DEVELOPMENT AND OPTIMIZATION OF THE NVIDIA CUDA APPLICATION FOR MILKYWAY@HOME

By

Anthony J. Waters

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

____

Boleslaw Szymanski, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

April 2010
(For Graduation May 2010)

CONTENTS

LIST OF TABLES . . . iv

LIST OF FIGURES . . . v

ACKNOWLEDGMENT . . . vi

ABSTRACT . . . vii

1. Introduction . . . 1

2. Volunteer Computer with BOINC . . . 4
   - 2.1 GPU Applications . . . 4

3. Implementation on the CPU and ATI GPU . . . 6
   - 3.1 CPU Application . . . 6
   - 3.2 ATI CAL Application . . . 7

4. Nvidia CUDA Application . . . 9
   - 4.1 Hardware Overview . . . 9
   - 4.2 Implementation Overview . . . 12
     - 4.2.1 Integral Kernel . . . 13
     - 4.2.2 Likelihood Kernel . . . 15
   - 4.3 Performance Bottlenecks . . . 15
   - 4.4 Optimizations . . . 17
     - 4.4.1 Alternate Memory Types . . . 18
     - 4.4.2 Division and Square Root . . . 20
     - 4.4.3 Loop Unrolling . . . 22
     - 4.4.4 Other . . . 23
   - 4.5 Single Vs. Double Precision . . . 25
   - 4.6 Single Precision Scalability . . . 25

5. Discussion and Conclusions . . . 27

LITERATURE CITED . . . 30

APPENDICES

A. Integral Kernel . . . 33

B. Hardware Specifications 37

LIST OF TABLES

4.1 Nvidia CUDA Memory Spaces [16, 11] 10

4.2 Execution Time (ms) of Global, Constant, Texture and Shared memory 20

4.3 Execution Time (ms) of the Division and Square Root Implementation . 21

4.4 Execution Time (ms) of Loop Unrolling 24

4.5 Precision and Performance of Single and Double Precision 25

LIST OF FIGURES

4.1 Uncoelesced Global Memory Access 13

4.2 Coalesced Global Memory Access 13

4.3 Performance plot of Global, Constant, Texture and Shared memory . . 20

4.4 Performance Advantage of the Division and Square Root Implementation 22

4.5 Performance Advantage of Loop Unrolling 24

4.6 Scalability of the application across generations 26

5.1 Advantage of Nvidia and ATI over the CPU 27

5.2 Advantage of Nvidia over the CPU 28

5.3 Speedup of the Optimized Versions over the Unoptimized Versions . . . 29

ACKNOWLEDGMENT

I would like to thank my family, friends, advisor, Nvidia and ATI. This work was partially supported by the NSF Grant IIS 0612213, however, the views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. Government.

ABSTRACT

Milkyway@Home takes advantage of maximum likelihood fitting in order to generate a model of fitting the Sagittarius Drawf Tidal Streams to the stars within the Milkyway Galaxy. The maximum likelihood calculation is simplified into a convolution integral and solved using the Gaussian quadrature method. Once a likelihood value is calculated it is used, along with the calculated parameters, in order to refine the model.

A disadvantage of this process is that it takes an exorbitant amount of CPU power and time. On a single core CPU, it was estimated that it would take 312 days to complete the analysis of a single stream. In order to get around the issue first grid computing was tried, through SALSA and MPI. Then, volunteer computing, through the BOINC framework, was experimented with. While analyzing a single stream through volunteer computing decreased the total amount of time spent, it was still time consuming in order to generate a single likelihood value.

To tackle this issue and decrease the time spent generating a single likelihood value GPU applications were explored. Initially there were concerns surrounding the fact that GPUs only supported single precision arithmetic, as opposed to double precision which is required. However, recent advances in the hardware have exposed double precision arithmetic in the next generation of GPUs. Therefore, this thesis explores the performance of a double precision Nvidia GPU application. In addition, a set of performance bottlenecks in the application are identified and a set of optimizations were fully explored and benchmarked. Lastly, the different applications that are used within Milkyway@Home are compared against each other in order to gain some insight on the computational power within consumer GPUs.

CHAPTER 1
Introduction

The primary purpose of the Milkyway@Home project is to generate a reliable model of fitting the Sagittarius Drawf Tidal Streams to the stars contained in the Milkyway Galaxy. This is accomplished through the use of supervised machine learning, in particular maximum likelihood fitting [4]. The secondary purpose is to take advantage of the GPUs and CPUs within a volunteer's computer, through the use of the BOINC middleware, in order to accelerate the generation of the model. Before going into detail about volunteer computing and the applications that are used to generate the likelihood it is necessary to gain a better understanding of the science behind the project.

The maximum likelihood calculation is used to find the ideal parameters in a parameterized model. This is done by being given real data and the parameterized model, from that the goal is to find the parameters in the model that would have been used to generate the original data. The likelihood is defined as the probability that the generated parameters can be used to observe the data [5]. This is summarized in the equation $L(Q) = \prod_{i=1}^{N} \mathrm{PDF}(l_i, b_i, g_i \| Q)$, where N is the number of stars and Q is the set of parameters in the model, of which there are eight [5]. By simplifying the equation it comes down to a convolution integral that is solved through the use of the Gaussian quadrature method, for more information see [4]. In order to actually solve the integral it is broken down into a mesh, with each space being solved independently of the other, then summed together generating the integral.

A single likelihood is generated for one input set of data, this likelihood and the calculated parameters are then used to generate more search parameters, in an attempt to explore the parameter space. This part is executed on the Milkyway@Home server and the generation of the new search parameters is done through various methods, genetic search, line search, and others as described in [7, 6]. Once it is determined that the parameters are good enough for the model they can be used to assign a probability to each star as to whether the star is part of the stream

or not, thereby serving the primary purpose of Milkyway@Home.

However, the generation of the likelihood value for each set of search parameters is time consuming. Using a single stream model it takes around 7.5 hours to obtain a single likelihood value. Then in order to refine the model's parameters more search parameters are generated creating the need to find more likelihood values. In order to find the final set of parameters using conjugate gradient descent a minimum of 1000 likelihoods would need to be calculated [5]. Thereby taking over 312 days to complete the analysis of a single stream. In order to reduce the amount of time to analyze the single stream two methods were employed.

The first was to take advantage of grid computing and volunteer computing[8]. With volunteer computing different volunteers would be responsible for computing the likelihood for a set of search parameters and then sending the result back to the server for processing. While it was still time consuming to perform a single likelihood calculation, the work is spread over many different computers.

The next method aimed to reduce the amount of computation time for computing a single likelihood value, and involved the use of programmable GPUs. The first known volunteer computing project to take advantage of GPUs was Folding@home. In order to use the GPU they took advantage of the Pixel Shader 3.0 specification contained in Microsoft's DirectX 9.0c API [10]. Since Pixel Shader 3.0, the development tools for programmable GPUs have come a long way. Nvidia created CUDA (Compute Unified Device Architecture) [17], which allows a programmer to use a subset of the C language to program the inner workings of the GPU. ATI also developed a framework called Brook+ [2]. Through these two frameworks it is possible to efficiently take advantage of the vast computing power of GPUs.

After the introduction of the GPU computing frameworks several other volunteer computing projects also explored GPU applications. Folding@Home released another GPU application, this time using CUDA and Brook+, in order to accelerate molecular dynamic simulations [13]. GPUGRID, another volunteer computing project using BOINC, also took advantage of Nvidia GPUs to perform molecular dynamic calculations [14]. In both of these projects the main goal of introducing the GPU applications was because of the performance that they could bring over

conventional CPU applications. Using single-precision math performance gains of 100x to 700x were reported in the case of Folding@Home.

Based on the potential performance gains by using a GPU for calculation a Nvidia CUDA application was developed for Milkyway@Home. Currently it is able to obtain a 40x performance increase over a single core CPU. However, in order to reach such high speeds the application went through several iterations of optimizations. Most of the optimizations were implemented based on finding performance bottlenecks in the application, while others were explored as a result the findings of other GPU programmers.

Chapter 2 describes how volunteer computing fits into the Milkyway@Home project, with a concentration on the GPU applications. Chapter 3 describes the version of the CPU application that volunteers are using on BOINC and also touches on the ATI application that was developed by a volunteer. The details of the Nvidia application and contributions of this thesis, which includes development of a double precision version, identification of performance bottlenecks and optimizations of the application, are described in Chapter 4. Chapter 5 compares the performance of the CPU, ATI, and Nvidia application and offers final conclusions of the thesis.

CHAPTER 2
Volunteer Computer with BOINC

At the heart of the Milkyway@Home project is volunteer computing, through this thousands of volunteers dedicate both CPU and GPU time in order to compute likelihood values that can be used to generate a reliable model. The infrastructure to support the volunteers is built upon the work of the Berkeley Open Infrastructure for Network Computing (BOINC) [3] middleware. Through this infrastructure many of the difficult tasks of setting up a volunteer computing framework are already completed. Generally, the only major part that needs to be done is the development of the applications that will execute on the volunteer's computer.

When a volunteer first joins Milkyay@Home they download an application that is designed to execute either on their CPU or GPU. The specifics of downloading a GPU application are elaborated on in Section 2.1. Then a three step process is repeated, first, the user downloads a set of project specific configuration files that the application uses to decide what to do, these files make up a work unit. After that the application that was downloaded executes the work unit, usually only while the computer has been idle. Once the execution is done the results of the computation are sent back to the server and the volunteer is given another work unit.

2.1 GPU Applications

To support GPUs within volunteer computing, and BOINC itself, several changes were made. The BOINC package was updated to version 6.10 which allowed detection of a volunteer's GPU along with the device driver version. It is important to know the device driver version because certain features are only available in specific drivers. Also, different drivers have different library names, so it is important to correctly distribute applications that are compiled for distinct driver versions.

Milkyway@Home had additional constraints besides the volunteer's device driver version, the GPU applications relied on the volunteer having a GPU that

was capable of performing double precision math. By default the server side software for BOINC makes no distinction between GPUs that support double precision and those that do not. Therefore the server side scheduler, the daemon that controls which volunteers are sent which applications, was updated to only send the GPU application to volunteers that had a double precision GPU. In order for the volunteer to be distributed the Nvidia CUDA application their GPU had to support compute capability 1.3, which signifies that it supports double precision. On the other hand, ATI GPUs do not have any compute capability to distinguish between features, but the ATI developer framework supports the ability to query whether a GPU has double precision or not. By adding in these constraints it ensured that volunteers with the correct hardware would be sent the correct application.

CHAPTER 3
Implementation on the CPU and ATI GPU

Currently on BOINC, volunteers have the opportunity to use either their CPU, ATI GPU, or Nvidia GPU in order to participate in Milkyway@Home. All three of these applications perform the same function, computing likelihood values, but they go about it in slightly different ways. Described below is an overview of how the CPU application is implemented and also an overview of the ATI application. In Chapter 4 there is a detailed explanation of the Nvidia application.

3.1 CPU Application

The first version of Milkyway@Home for BOINC was implemented in C/C++ and was designed to execute on a single core of a x86/x86_64 CPU. The source code is a direct implementation of the formula described in Chapter 1. The application can be broken down into three distinct steps, setting up the data structures, estimating the model parameters, and generating a likelihood based on the estimated parameters.

The first step is setting up all of the data structures required for computation along with their default values. Part of these values are taken from two configuration files, astronomy_parameters.txt and search_parameters.txt. The first file, astronomy_parameters.txt, contains the information specific to the current search that is being performed. Within this file there are three important parameters, the number of mu steps, nu steps, and r steps. As described in [5], these parameters are used to identify a point within a wedge. They are also the primary factors that determine how long the execution is going to take, because for each point within the wedge the convolution integral is calculated.

The second file, search_parameters.txt, contains more work unit specific information, such as the initial parameters for the search. These parameters guide the computation and affect the final likelihood. They are generated by the server when new results come in through various evolutionary algorithms [8]. There is also a

third data file, stars.txt, that contains the locations of the various stars within the current wedge. The stars are used in the final likelihood calculation to determine how well the estimated parameters are for the model [5].

After the setup work is complete the next step is to estimate the model parameters by evaluating the integral. The integral is evaluated for each potential value of mu and nu, this step is performed r times. Once the r iterations have completed for each mu and nu the final values for the background integral and each of the stream integrals are given.

The values of those integrals are used to generate the weights in the model, correspondingly called the background weight and the stream weight. In order to find the final likelihood value, for each star present a set of background and stream probabilities are generated and multiplied by the weights created earlier. Then a probability is generated for each star, and the average probability across all of the stars is taken to be the final likelihood.

Before the Nvidia application was started the CPU version was already running and implemented by another member of Milkyway@Home. It was, however, slightly updated so that the Nvidia version could share more the code with the CPU application. For example, most of the code used by the CPU version that was responsible for setting up the data structures was extracted into separate functions so that they could be called by both the CPU version and Nvidia version.

3.2 ATI CAL Application

A volunteer also noticed the lackluster performance in the CPU application and with the advent of programmable GPUs, decided to implement their own version of Milkyway@Home that took advantage of an ATI GPU. The programming was done using Brook+ and some IL [2]. IL is similar to an assembly language for a processor, it contains specific instructions to execute for the GPU. While it is generally harder to use straight IL it offers a performance benefit by allowing for fine tuning of how the device functions.

In addition to using IL to extract more performance, the ATI application also implements some of the same optimizations as the Nvidia application. The

biggest optimization was the enhanced square root and division functions, which are discussed in Chapter 4.

CHAPTER 4
Nvidia CUDA Application

With the advent of programmable GPUs and the relatively slow performance of the CPU version, the Nvidia application was designed in order to offload computation onto a Nvidia GPU. Most of the code is shared with the CPU application with a few major differences. First, in order to execute on the GPU some setup needs to be performed, mostly with copying the required memory from the host to the device. The host in this case is the computer containing the GPU and the device is the actual GPU. The second difference is the calculation of the integral and the likelihood are performed on the GPU as opposed to the CPU. The rest of the application is identical to the CPU version which facilitates code reuse. The application that is described below is the highly optimized, double precision version, that was derived from the initial single precision Nvidia application created by another member of Milkyway@Home. The optimization of this initial verion, discussed in the chapter, and testing and gathering performance data, presented in the next chapter, are the core and individual contributions of the author of this thesis.

4.1 Hardware Overview

The hardware present in the Nvidia GPU operates differently than that of a regular CPU. One thing to keep in mind is that GPU stands for Graphical Processing Unit, and GPU development revolves around rendering complex 3D scenes for video games [15]. As a result of this, the GPU is accustomed to executing instructions in parallel across the device with little to no branch type instructions. This is important to know when developing an application for the GPU, if the application will have a lot of branch type instructions or if is is very hard to parallelize it will not benefit from the parallel architecture of a GPU.

Another aspect that the GPU differs from the CPU is in the types of available memory. The GPU is composed of six different memory spaces: registers, shared memory, global memory, local memory, texture memory, and constant memory, a

Type	Location	Size	Scope
Register	On SM	16384 (GT200) per SM	Thread
Shared	On SM	16KB per SM, 16 banks	Block
Constant	Off SM, Cached	64 KB, 8 KB cache per SM	Global
Texture	Off SM, Cached	Limited by Video Memory	Global
Local	Off SM	16 KB per thread	Thread
Global	Off SM	1024 MB (GT200)	Global

Table 4.1: Nvidia CUDA Memory Spaces [16, 11]

table summarizing each memory space can be seen in Table 4.1. In order to achieve the best performance possible it is necessary to exploit these various memory spaces and to put critical pieces of data either within registers or shared memory for the fastest access. However, there are is a very limited amount of registers and shared memory, each streaming multiprocessor (SM) is limited to 16384 32 bit registers and 16KB of shared memory. Constant memory and texture memory are very useful for look up tables as these two memory spaces each have a separate cache for frequently accessed data. Global and local memory should try to be avoided at all costs as a result of their high access latency, however, much of this access latency can be hidden by having a lot of active threads[17]. GPUs are also limited by the total amount of memory they have access to, the GTX 285 used for testing has 1024MB[18] of video memory. The video memory is shared between global memory, texture memory, constant memory, and local memory.

The execution units within the Nvidia GPU are arranged into streaming multiprocessors (SMs), the number of SMs is dependent on the level of hardware purchased. The top of the line GTX 285 contains 30 SMs while towards the lower end the GTX 260 contains only 24 SMs [17]. The number of SMs present in the GPU directly corresponds to how fast the application will be, the more SMs, the more instructions that can be executed in parallel. Within each SM are eight Scalar Processor (SPs) cores, the shared memory, registers, and other hardware. Each SP is capable of handling one thread at a time, so at any given time while the GPU is executing it is possible for 240 threads to be executing instructions. In total the GPU is able to support 30720 threads that are active, 1024 per SM with 30 SMs in the GTX 285. Each SM also contains two Special Function Units (SFUs) that

have the ability to compute floating point multiply instructions and transcendental math. The SPs and SFUs can only do single precision floating point, so each SM also contains a double precision processing unit (DPU). While the GTX 285 contains 240 SPs to perform single precision, there exists only 30 DPUs to perform double precision, therefore the GTX 285 will be approximately 8 times slower at double precision compared to single precision [19, 22].

In order to keep the SMs and SPs busy the GPU has a broken down execution model. At the top there are thread blocks, this is just a group of 1 to 512 threads. Each thread block shares the same shared memory space, allowing threads to communicate and share data. Thread blocks are assigned to SMs, with a maximum of eight blocks per SM. The blocks are further divided into thread warps of size 32, all threads within the warp execute the same instruction, except operating on different data elements. If a group of threads in a warp decide to execute a branch type instruction the warp diverges into two groups and the execution of each group is serialized. It is also important to note that the shared memory within a block is divided into 16 banks and if more than one thread attempts to access the same bank of shared memory there will be a bank conflict. When a bank conflict occurs the access is serialized, resulting in a performance hit. However, access to shared memory is split into half-warps, meaning the first 16 threads in the warp access the shared memory then the other half does. Therefore if the correct precautions are taken there will be no bank conflicts. For more in depth information pertaining to the CUDA architecture the reader is referred to the Nvidia Programming Guide, [17].

The last part of the GPU that varies from the CPU is the method in which instructions are executed. With a CPU a program is developed, in a myriad of languages, and then compiled. From start to finish the program is designed to execute on the CPU. GPU programs operate slightly differently. There is a host portion, that can be developed in any language supporting CUDA, mostly C/C++, and a device portion. The host portion executes on the CPU, while the device portion is designed to be executed on the GPU. The set of instructions that make up the device portion is called the kernel. During the execution of the host portion

there comes a point in which the host instructs the GPU to execute the kernels. Then, the CPU waits for the GPU to finish and processes the results. For the kernel, in the case of Nvidia, there exists only two programming languages. The first is CUDA, which is a subset of C, and the second is PTX [20]. PTX can be thought of as assembly language, and the CUDA device code is actually compiled into PTX. For Milkyway@Home two device kernels are used, the integral kernel and likelihood kernel, which are elaborated on below.

4.2 Implementation Overview

Before executing the kernels on the Nvidia GPU a lot of initialization steps need to be performed. The first thing that is done is the actual GPU is chosen, this is especially important when the system has multiple GPUs. To do this all of the Nvidia GPUs in the system are enumerated and checked for compute capability 1.3, which signifies double precision support. Next, of the eligible GPUs the one that has the highest GFLOP/s, as reported by the CUDA run time, is chosen as the GPU to use. The GPU is only allowed to access memory that resides on the GPU itself, therefore the nest step is to copy all of the data that the algorithm would use to the GPU. However, two things need to be taken into consideration, 1) GPUs have a limited amount of memory and 2) in order to obtain the best performance the memory has to be aligned in a specific order. Since the application requires compute capability 1.3, which is only present in the latest generation of Nvidia GPUs, memory is not a big problem. These GPUs have around 1024MB of memory [18], while the largest work units that Milkyway@Home use take up around 128MB of memory.

The other important factor is the layout of the memory on the GPU, in particular the global memory. The global memory resides off-chip on the GPU and is very large compared to the on-chip memory, however, it is also very slow, with memory accesses taking around 400 - 600 cycles [17]. In order to have the fastest access possible to the global memory the reads and writes need to be coalesced [16]. Memory accesses are coalesced when thread 0 is accessing memory block 0, then thread 1 accesses memory block 1, and so on. If this pattern is not followed then

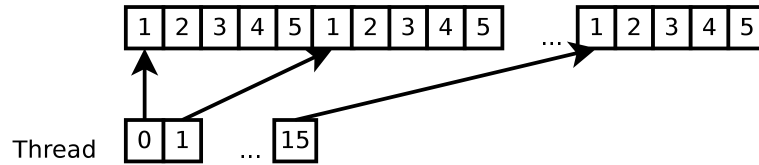

Figure 4.1: Uncoelesced Global Memory Access

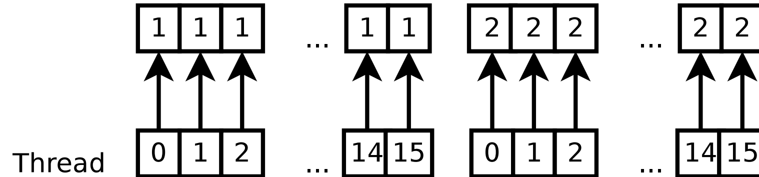

Figure 4.2: Coalesced Global Memory Access

the memory requests cannot be merged into one big request. Therefore, to have coalesced memory access, the ordering of some of the data needed to be changed. For example, each thread needs to read five parameters from a variable named `stars`. Within the CPU application the data is arranged such that for the first thread the five parameters are in locations 1 through 5, thread one's locations are from 6 to 10, see Figure 4.1. Since the accesses are non-coalesced they result in separate global memory reads. To over come this the data was arranged so that each thread would access parameter 0 in order, then parameter 1, and so on, as in Figure 4.2.

4.2.1 Integral Kernel

The integral kernel is where most of the computation takes place and is responsible for evaluating the convolution integral using the Gaussian quadrature method, as discussed in Chapter 1. The mathematical operations performed within the kernel are the same as the ones used by the CPU, however, the structure is different. As mentioned previously, the CPU version first loops over the parameter mu, nu, and lastly over the parameter r, as seen in Algorithm 1. This last loop is where the probabilities for the background integral and the stream integrals are calculated. The GPU version operates differently, it instead loops over the r parameter and then (mu * nu) threads are created to perform the calculations, visible in Algorithm 2. It is important to note that each thread is assigned a different mu and nu value, effectively emulating looping over the values of mu and nu.

Algorithm 1 Summing order in the CPU Application

for mu_min to mu_max with mu_steps
    for nu_min to nu_max with nu_steps
        for r_min to r_max with r_steps
            Compute value the point in the wedge at mu_step, nu_step, r_step
        end
    end
end

Algorithm 2 Summing order in the GPU Applications

for r_min to r_max with r_steps
    for each thread (there are mu_steps * nu_steps threads)
        Compute value the point in the wedge at mu_step, nu_step, r_step
    end
end

The different execution order causes some issues, mostly having to do with the differences in the floating point results. Since a lot of smaller numbers, on the order of $1 * 10^-8$, are being summed together, the order of the summations changes the final result. The floating point operations are not associative [1]. For example (A + B) + C will not equal the same answer as A + (B + C) due to the nature of the storage of the numbers. The CPU application performs the summation over mu and nu, while the GPU applications does it over r. Mathematically the results should be identical, but with floating point numbers, even with double precision, the numbers start to differ around the 12th decimal place.

A complete copy of the integral kernel can be seen in Appendix A. The first major part is the `for` loop that operates over the `convolve` variable on line `43`. The next step calculates the value for the variables `xyz2`, `xyz0`, and `xyz1`. While calculating these the value of `r_point` is obtained through a 2D texture fetch. Those values are used to calculate `rg` and `bg_int` (background integral), which use the optimized math functions `fsqrtd` and `divd` Then the integral kernel calculates the value of the stream integral for the different streams present from lines `57` to `140`. Once most of the math calculations are performed the results are written to global memory so that they can be processed by the next iteration of the integral kernel.

In total there will be `r_step` iterations. After the final values of the background and stream integrals are calculated per thread they are summed together on the CPU. This produces the final background integral and the final stream integrals, which are used to generate inputs for the likelihood kernel.

4.2.2 Likelihood Kernel

The purpose of the likelihood kernel is to check how well the estimated parameters fit the model. This is done by calculating the likelihood the parameters are correct for each star present in the wedge. Generally there are 100,000 stars present in the wedge, so the likelihood kernel represents a very small fraction of the total execution time, therefore not a lot of time was spent optimizing it. Typically the likelihood kernel takes around .088 seconds, while the integral kernel takes around 60-600 seconds depending on the size of the work unit. So even if the likelihood kernel was made five times faster it would not affect the total execution time a great deal.

Code wise, the likelihood kernel is very similar to the integral kernel, the major difference is that some of variables are generated based on the outputs from the integral kernel, most notably the result of the background and stream integral. In addition, instead of a thread being created for each value of mu and nu, a thread is created for each star in the wedge. Since each star needs to calculate different parameter values, these values cannot be pre-computed, as in the case with the integral kernel. These two things prevent the code already contained in the integral kernel from being extracted out into a generic function for use by both kernels.

4.3 Performance Bottlenecks

The first iteration of the Nvidia application was much faster than the CPU version. However, despite the speed advantage over the CPU version the Nvidia application was not showing the same performance when compared to the ATI application. Part of the reason for the performance difference between the two applications is the difference in the amount of double precision floating point operations per second (FLOPS) the hardware is capable of. As seen in Appendix B the sample

ATI Hardware exhibits 544 GFLOP/s and the Nvidia hardware contains only 78 GFLOP/s. Even if this difference was taken into account, the Nvidia version was still slow.

Before implementing the optimizations it was necessary to discover the causes of the low performance, when compared to the ATI application. The first bottleneck was the actual use of double precision, both GPUs have significantly less double precision FLOPS compared to single precision, mostly due to the fact that there is less hardware dedicated to double precision. In the case of the Nvidia GPU there exists 240 single precision floating point processing units, compared to only 30 double precision floating point processing units. However, double precision is required because of the magnitude of the numbers involved exceeds the storage space of single precision and single precision only provides accurate numbers to about the seventh decimal place.

Another part of double precision that makes it slower than single precision, besides the number of FLOPS, is the lack of intrinsic functions. For example, with single precision there exists two different versions of some math functions, like `sin` and `__sin`. The former version is performed within the CUDA library and is designed to follow the IEEE 754 specification for accuracy. While the latter version is implemented as an instruction within the SFU, which is faster, at the cost of lower precision [24]. Another example is with single precision division, it is implemented as a sequence of inlined instructions, however, in order to solve double precision division an iterative approach is used [22]. The iterative approach, while having more accurate results, leads to a performance hit in the application.

Another bottleneck that was present was the limited parallelism that was occurring in the GPU itself. As mentioned previously in the description of the hardware the Nvidia device has a limited number of stream multiprocessors (SM). Each SM is only allowed to support a limited number of blocks, the constraint being on the amount of shared memory used per block and the amount of registers used by each thread in the block. The concern here was mostly to do with registers. In the case in which there are three streams in the input file and using shared memory the nvcc compiler would attempt to use 49 registers per thread. According to the

CUDA Occupancy Calculator [21] this would result in a SM occupancy of only 25%. If the memory type was switched to be global memory the register usage jumped to be 82, reducing the occupancy to 12% with two active blocks per SM. The amount of shared memory used per thread block is also a constraint. Each SM is limited to 16 KB of shared memory that must be shared between all active blocks on the SM. If each block needed 4 KB of shared memory then that only allows four active blocks. This would be the worse case in which there are 128 threads with four streams, needing to store 32 bytes per thread.

In addition to the bottlenecks described above there were several other minor ones present. First was the issue with writing to shared memory using double precision. Shared memory is aligned into 16 banks of 32 bits per bank, this means that each thread in a half-warp (16 threads) can access one bank of memory with out any bank conflicts. In the event of a bank conflict the requests are serialized - causing a slowdown. However, keep in mind that a double type variable requires 64 bits, meaning each write to a double in shared memory requires accessing two banks of shared memory resulting in a bank conflict every time it is written [17]. Other minor issues included accessing global memory in a non-coalescing way and using a `for` loop, which includes a branch type instruction.

4.4 Optimizations

To reduce the bottlenecks above and increase performance several optimizations were taken into account. The three major ones where using alternate memory types, loop unrolling, and changes to the way the division and square root operations were implemented. Besides those enhancements several other smaller ones were looked into.

The benchmarks below were executed using a Nvidia GeForce GTX 285 with an AMD Athlon64 X2 Dual Core Processor 4400+ CPU on GNU/Linux (Gentoo) with kernel 2.6.30, for more details see Appendix B. Each benchmark was repeated five times and the execution time includes only the time of the likelihood and integral kernels, so the time on the GPU. The results were averaged, and the average was used for analysis. The GPU was not connected to a display, thereby avoiding any

potential slowdowns relating to the operating system using the GPU to accelerate the Graphical User Interface.

4.4.1 Alternate Memory Types

Table 4.1 summarizes the main memory types present in the Nvidia GPU. In order to obtain the best performance possible it is necessary to choose the correct memory type suitable for the application, taking into account the constraints on the amount of memory available. The main piece of code that memory access was an issue was within the inner loop of the integral kernel. At this location the variables `fstream_a` and `fstream_c` are fetched. The values in these variables are constant across the whole GPU, but change with each work unit so they cannot be hard coded. As a result of their volatility they can be stored in any of the memory types, most notably shared, texture, constant, and global memory. While registers could also be used it would offer very little benefit. It would use more registers, which would reduce the occupancy of the GPU because there would not be enough registers to have enough actively running blocks. Second, if the the compiler was forced to use less registers the data would spill over into local memory, which has a very high latency, effectively removing any benefit of using registers.

To test the performance of the four different memory types, four different versions of the kernel were created. The results are very interesting and a graph of the performance of each memory type can be seen in Figure 4.3 and in Table 4.2. Shared memory turned out be the fastest version, being about 3% faster than constant memory. This translates into an almost 15 second difference in execution time, just by changing the type of memory. Part of this speedup is due to the fact that the GPU does not have to access global memory at all within the inner loop, reducing the overall demand on global memory [23]. In addition, the latency of using shared memory is in the range of the same latency of a register, making it very fast to access. Also, each thread is accessing the same location in shared memory so a broadcast mechanism is used by the hardware, ensuring that there are no bank conflicts [16].

Using texture memory was approximately 1.25% slower with the three stream

work units, while with the two stream work units the slowdown was about 0.1%. Therefore, despite having the advantage of a texture cache, the speed was slower than using only global memory. Part of this is due to the fact that CUDA does not natively support using `double` data types in texture memory. By default only the `float` data type is supported. In order to get around this limitation the CUDA framework exposes the functionality of converting an eight byte `double` into two four byte `ints`. Therefore the data is stored in texture memory as an `int2`, which is a two component `int` vector, with each component representing four bytes of the `double`. When the texture is retrieved the `int2` is fetched from texture memory as seen in Algorithm 4.1, and combined into the correct `double` value. The slowdown could also be contributed to the fact that the cache used by the texture memory is not designed to reduce the latency of accessing memory, but is instead designed to reduce the amount of memory that is required to be fetched from global memory [17, 16].

Listing 4.1: Function to Retrieve Double Precision Textures

```
1 static __inline__ __device__
2 double
3 tex3D_double(texture<int2, 3> tex, int x, int y, int z)
4 {
5   int2 val = tex3D(tex, x, y, z);
6   return __hiloint2double(val.x, val.y);
7 }
```

The most surprising piece is that constant memory is slower than global memory, given the fact that they are both stored in the same memory space and constant memory has an 8KB cache. One possible explanation for the slowdown is the cache is not being taken advantage of effectively. As stated in [22] when the data is not in the constant cache it is actually slower to fetch from constant memory versus fetching it from global memory. This could explain the slowdown over global memory.

From looking at the numbers it is also possible to see that while using texture and constant memory is always slower than using global memory, the performance

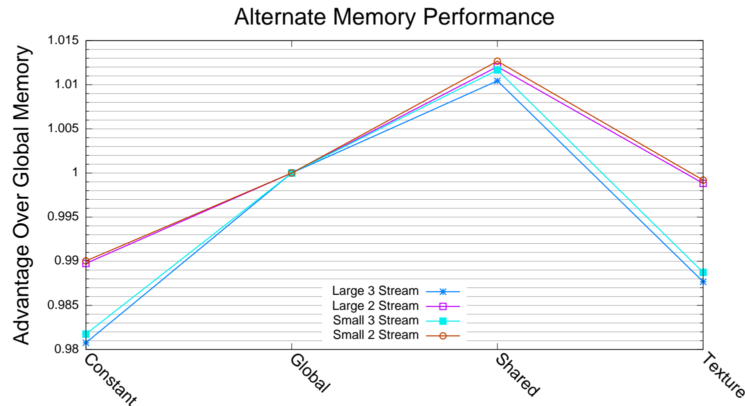

Figure 4.3: Performance plot of Global, Constant, Texture and Shared memory

	Work Unit			
Memory Type	3-S Large	2-S Large	3-S Small	2-S Small
Constant	537931.571	386745.631	67644.509	48714.355
Global	527584.387	382782.138	66409.764	48228.913
Shared	522134.869	378224.084	65643.893	47625.491
Texture	534162.763	383230.794	67166.101	48267.882

Table 4.2: Execution Time (ms) of Global, Constant, Texture and Shared memory

hit decreases in the two stream work units. With the three stream work units the constant memory was over 10 seconds slower, however, in the case of the two stream work units the difference was only about 4 seconds. In the case of texture memory the performance difference is 6.5 seconds and 0.5 seconds. It is possible that in the case of the two stream work units the data for the `fstream` variables fit all the way in the constant and texture cache.

4.4.2 Division and Square Root

In order to reduce the bottleneck associated with using double precision math an optimized version of division and square root were implemented. The implementer of the ATI application was the creator of the original version of the specialized functions and they were originally implemented in the ATI application. To get them to function correctly and efficiently on Nvidia hardware a few changes were

	Work Unit			
Memory Type	3-S Large	2-S Large	3-S Small	2-S Small
Constant	568881.729	71658.759	410171.221	51742.278
Global	547891.089	68936.260	396294.771	49945.827
Shared	539140.643	67824.010	390968.988	49256.236
Texture	549780.899	69195.699	396980.649	50028.304

Table 4.3: Execution Time (ms) of the Division and Square Root Implementation

made, mostly by changing some operations like 1.0/sqrt(x) into __rsqrtf(x), which is faster on a Nvidia GPU. This allowed for more performance while staying within the range of precision. These enhanced methods took advantage of both algorithmic enhancements and single precision in order to perform some math functions. This could be called mixed precision, as in [12], whereby using single precision in certain calculations increased the performance a great deal, while making little to no impact on the final result. An overview of the details of the optimized division and square root implementations are described in [9]. Besides using an optimized division and square root method a faster exponential method was looked into, however, it proved to be slower than the implementation contained within the CUDA run time therefore it was abandoned.

As seen in Figure 4.4 and Table 4.3 the performance gained by using these functions is from around 17% to 12% depending on whether the work unit contains two or three streams. This translate into a performance advantage of around 65 seconds for the large three stream work unit. Global memory and shared memory were able to take more advantage of the improved functions, likely due to the fact that they are able to access the memory faster for the division and square root operations. The large and small work units also show nearly the same performance advantage, given the fact that their execution times differed by at least 500 seconds, clearly showing the scalability of the enhancement. The speedup from this enhancement occurred for two reasons. First the enhanced versions perform less instructions, thereby increasing the throughput of the functions themselves. Also, some of the double precision math was substituted with single precision math, allowing the use of the single precision floating point units within the GPU.

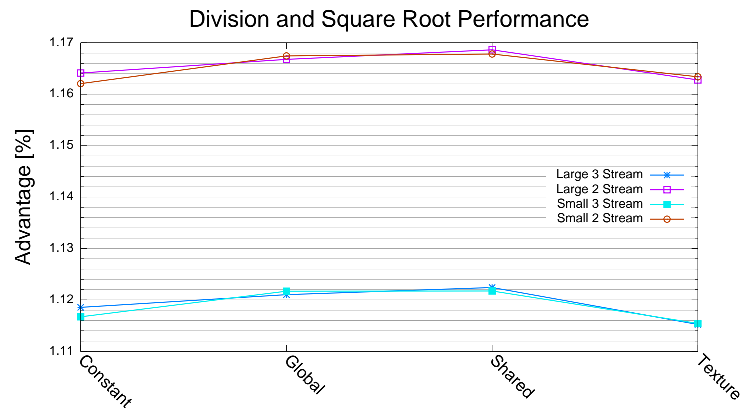

Figure 4.4: Performance Advantage of the Division and Square Root Implementation

4.4.3 Loop Unrolling

Loop unrolling is the process of taking the contents of a loop and instead of actually executing the contents of the loop x times the contents are copied over x times. This effectively eliminates the loop while keeping the behavior of the code block the same, at the expense of harder to read source code and a larger executable. The kernels described above are comprised of two nested `for` loops. The outer loop executes `convolve` times, which is typically around 120, this does not make it an ideal candidate for loop unrolling. Having to copy the code inside the loop 120 times would make the source code very difficult to maintain and correspondingly puts more constraint on the compiler to generate optimized source code. Therefore only the inner loop which executes `number_streams` times, which is from one to four, was unrolled.

The number of streams changes with each work unit, so `if` statements were inserted into the kernel to control when the calculations for a stream were done. Therefore if the kernel had only two streams, the `if` statements would ensure that only the math for those two streams is executed. The GPU kernels make use of C++ template parameters, and one of those template parameters is the `number_streams` variable. This means that when compiling the kernels four different versions are created, one for each possible value of `number_streams`. Another benefit of this is that the `if` statements mentioned above are actually compiled out of the kernel.

This does two things, first it eliminates the `if` statements so the GPU does not have to execute them and it allows for smaller stream work units to actually execute less code, hence the performance difference between the two stream and three stream work units.

In addition, loop unrolling eliminated the need for the loop itself, removing a conditional and the loop counter. Also, no math has to be performed to index the `fstream_a` and `fstream_c` arrays because the index was dependent on the loop variable, which is the current stream. The other benefit was the fact that the temporary variable `st_int`, for the stream integrals, no longer has to be contained in shared memory and instead can be placed within a register. As mentioned previously, in the Performance Bottlenecks section, using shared memory for double variables resulted in shared memory bank conflicts, by switching to registers this bottleneck is eliminated. There is another advantage of not using the shared memory as well, it allows for more parallelism by reducing the constraint on the number of active blocks per SM.

The results of the loop unrolling can be seen in Figure 4.5 and in Table 4.4. Overall constant memory benefited the most from loop unrolling, followed by global memory, shared memory, and lastly texture memory. The gains from loop unrolling were not as great as from the enhancement to the division and square root implementations, but still represents a performance advantage of around 2% to 5%. When using constant memory, global memory, and shared memory part of the performance advantage stems from the fact that those memory spaces previously had their memory locations calculated within the `for` loop. By removing the `for` loop these extra indexing calculations were eliminated. Texture memory never needed to perform any extra math to look up the memory locations, thereby exhibiting less of a performance gain. The rest of the performance gain can be attributed to the removal of the `for` loop itself and using registers to store the variable `st_int`.

4.4.4 Other

Besides the three main enhancements above several others were looked into, but provided little to no improvement in performance. The first was to introduce

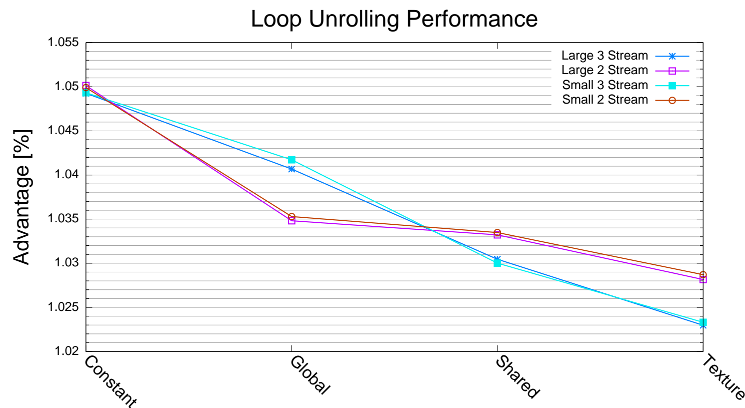

Figure 4.5: Performance Advantage of Loop Unrolling

	Work Unit			
Memory Type	3-S Large	2-S Large	3-S Small	2-S Small
Constant	606447.033	76260.728	454693.980	57269.315
Global	590187.810	74228.922	446832.900	56321.746
Shared	587253.033	73864.422	442208.945	55658.954
Texture	599357.472	75424.386	448953.353	56577.585

Table 4.4: Execution Time (ms) of Loop Unrolling

memory coalescing for the global memory reads that occur at the start of the kernels. As mentioned earlier, the data structure `stars` was transformed to be able to be coalesced, this change resulted in only a few seconds difference in execution time.

Another small modification was changing the number of threads that would be included in a single block. Previously the number of threads per block was set to be the same as the number of nu steps, which would generally by 160. This presented several disadvantages, first it was not a multiple of 32, which according to Nvidia generates the best performance [17]. Also, having the number of threads per block depend on an input parameter could lead to issues if the nu parameter is above 512 which is the maximum number of threads allowed per block. Therefore, this parameter was fixed at 64 threads per block, which allowed for the best performance.

In order to get around the issue of reduced parallelism, due to a low amount of occupancy, a constraint was placed on the maximum amount of registers that are allowed to be used. Previously using 49 registers resulted in a low occupancy

	3-S Large		2-S Large	
	Likelihood	Time (ms)	Likelihood	Time (ms)
Single Precision	-3.238302 15350	75427.506	-2.9853108 48100	63759.320
Double Precision	-3.238302 24389	522134.869	-2.9853108 33254	378224.084
Difference	-	6.92x	-	5.93x

Table 4.5: Precision and Performance of Single and Double Precision

of 25%, by enforcing the compiler to set a limit of 48 registers per thread through the `--maxrregcount` argument the occupancy increased to 31%. This change in occupancy resulted in a performance difference of around 50 ms with the three stream large work unit. However, when taking into account the global memory version that used 82 registers, reducing that number to 48 resulted in a performance difference of around 37 seconds.

4.5 Single Vs. Double Precision

Milkyway@Home requires the use of double precision floating point math in order to achieve reliable results. With single precision the likelihood values are only accurate to about the 6-7th decimal place, while double precision doubles this to around the 12-13th decimal place, as seen in Table 4.5. By using a double variable to increase the amount of precision a sizable performance hit is taken. When using the fastest version of the single precision application it ranges from 6-7x faster than the fastest double precision version. This speedup is expected because the Nvidia hardware contains 8 single precision FPUs to 1 double precision FPU [22]. As in [12] a similar speedup is experienced by using single precision over double precision, in addition there is also a difference in the accuracy of the final calculations.

4.6 Single Precision Scalability

The GTX 285 series is the only generation of GPU from Nvidia that supports double precision math, therefore it is difficult to get an idea of how well the application will scale between different generations. However, since the current and previous generation of GPUs both support single precision it is possible to compare the performance with only single precision. The difference in the amount of single

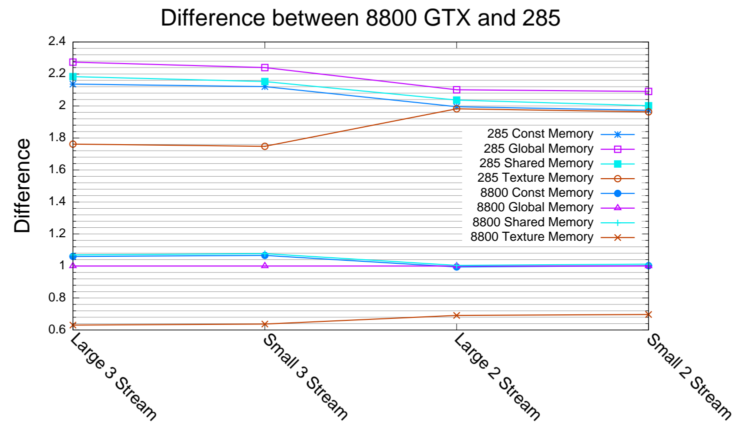

Figure 4.6: Scalability of the application across generations

precision GFLOP/s between the two GPUs is around 80%, as seen in Appendix B. Therefore the GTX 285 should be roughly 1.8x faster than the 8800 GTX.

Figure 4.6 depicts the performance difference between the current generation GPU, a GTX 285, and a previous generation GPU, the 8800 GTX, with the global memory version of the 8800 GTX being the baseline at 1. The fastest single precision version of the GTX 285 is approximately 2.3x faster than the 8800 GTX, which is comparing the performance between both global memory version. This shows that the application has good scalability, when the hardware improves by 1.8x the application is 2.3x faster.

CHAPTER 5
Discussion and Conclusions

The development of the Nvidia CUDA version of Milkyway@Home proved to be viable. The Nvidia application is roughly 40x faster than the CPU application when processing the same work units. This equates to a time difference of execution around 337 minutes for the large three stream work unit. The CPU took a total of 346 minutes, while the Nvidia GPU was able to complete it in 8.7 minutes. While this is still a sizable margin it was nothing compared to the performance attainable by using an ATI GPU. Figure 5.1 shows the performance of the ATI GPU compared to both the Nvidia GPU and the CPU. The ATI application was able to obtain a stable 249x speedup over the CPU. Granted the ATI GPU that was used was recently released in November of 2009 and represents around 544 GFLOP/s of double precision performance. While the Nvidia GPU was released earlier in January of 2009 and contains only 78 GFLOP/s of double precision performance.

In order to reach a high amount of performance the Nvidia application went through several iterations of performance enhancements. As seen in Figure 5.2, originally the Nvidia application was only around 34x faster than the CPU version. With the large three stream work unit it would take 10.2 minutes to complete. While

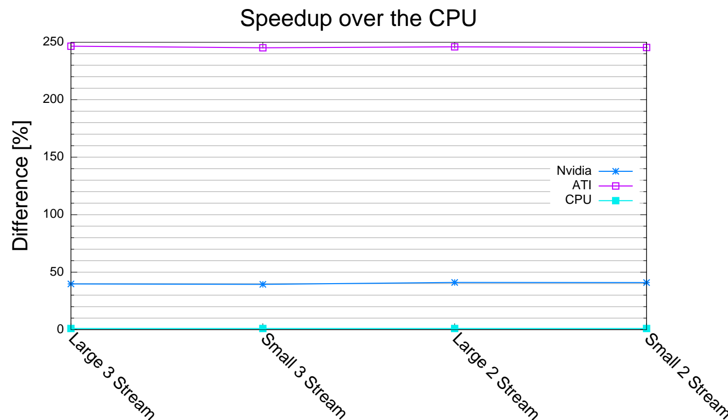

Figure 5.1: Advantage of Nvidia and ATI over the CPU

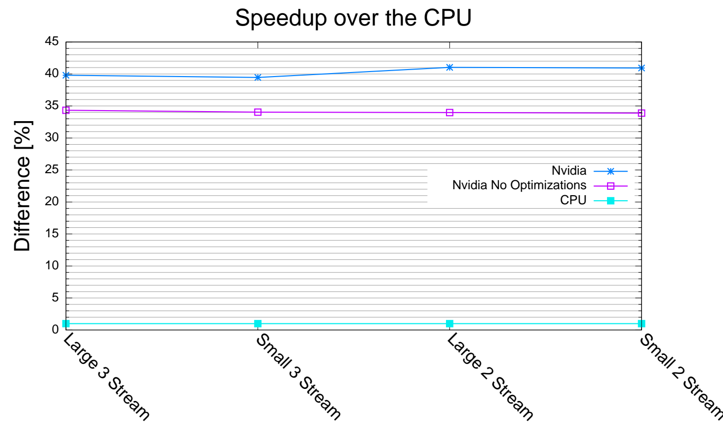

Figure 5.2: Advantage of Nvidia over the CPU

this is much faster than the CPU, it was far from the expected speedup and was slower than the ATI version, taking into account the difference in GFLOP/s.

To increase the performance of the Nvidia application first the performance bottlenecks were identified. They included such things as low GPU occupancy and the low performance of double precision math compared to single precision. Knowing the performance bottlenecks made it simpler to create the performance optimizations. The performance optimizations included using faster methods for division and square root, loop unrolling, and lastly by implementing small optimizations the speedup of the Nvidia version increased to 40x over the CPU version. Figure 5.3 portrays the speedup of the optimized Nvidia version over the unoptimized global memory version, which was the first one implemented. While some optimizations increased the performance over the original version, the shared memory version containing all of the optimizations was able to reach the highest performance. The optimized versions also experience more of a performance boost when calculating two stream work units over three stream work units.

The results obtained from the GPU application for Milkyway@Home match similar performance quotes of other GPU applications. The speedup of 40x obtained over the CPU version when using double precision is comparable to the speedups that others have obtained when using only single precison. With Nvidia's next generation of hardware (GTX 480/Fermi) it is reported that there exists 168 GFLOP/s of double precision floating point performance. With numbers like that it may very

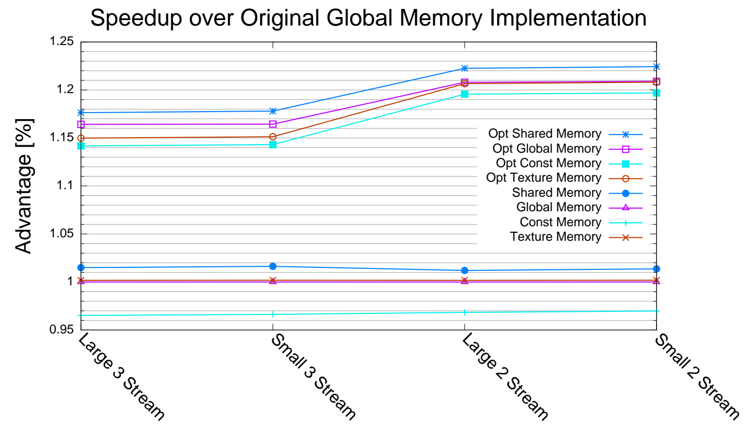

Figure 5.3: Speedup of the Optimized Versions over the Unoptimized Versions

well be possible to exceed the performance of the ATI application.

LITERATURE CITED

[1] IEEE Standard for Binary Floating-Point Arithmetic. http://grouper.ieee.org/groups/754/, 1985. ANSI / IEEE Std. 754-1985.

[2] AMD Corporation. ATI Stream Computing User Guide Version 1.4.0.

[3] D.P. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *e-Science*, pages 196–203. IEEE Computer Society, 2005.

[4] Nathan Cole. *Maximum Likelihood Fitting of Tidal Streams with Application to the Sagittarius Dwarf Tidal Tails*. PhD thesis, Rensselaer Polytechnic Institute, 2009.

[5] Nathan Cole, Heidi Newberg, Malik Magdon-Ismail, Travis Desell, Kristopher Dawsey, Warren Hayashi, Jonathan Purnell, Boleslaw Szymanski, Carlos A. Varela, Benjamin Willett, and James Wisniewski. Maximum likelihood fitting of tidal streams with application to the sagittarius dwarf tidal tails. *Astrophysical Journal*, 683:750–766, 2008.

[6] Travis Desell, Nathan Cole, Malik Magdon-Ismail, Heidi Newberg, Boleslaw Szymanski, and Carlos Varela. Distributed and generic maximum likelihood evaluation. In *3rd IEEE International Conference on e-Science and Grid Computing (eScience2007)*, pages 337–344, Bangalore, India, December 2007.

[7] Travis Desell, Malik Magdon-Ismail, Boleslaw Szymanski, Carlos A. Varela, Heidi Newberg, and Nathan Cole. Robust asynchronous optimization for volunteer computing grids. In *Proceedings of the 5th IEEE International Conference on e-Science (eScience2009)*, pages 263–270, Oxford, UK, December 2009.

[8] Travis Desell, Boleslaw Szymanski, and Carlos Varela. An asynchronous hybrid genetic-simplex search for modeling the milky way galaxy using volunteer computing. In *Genetic and Evolutionary Computation Conference*, Atlanta, Georgia, July 2008. To Appear.

[9] Travis Desell, Anthony Waters, Malik Magdon-Ismail, Boleslaw Szymanski, Carlos A. Varela, Matthew Newby, Heidi Newberg, Andreas Przystawik, and Dave Anderson. Accelerating the milkyway@home volunteer computing project with gpus. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics (PPAM 2009)*, page 13 pp., Wroclaw, Poland, September 2009. To appear.

[10] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V.S. Pande. N-Body simulation on GPUs. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 188, New York, NY, USA, 2006. ACM.

[11] Rob Farber. CUDA, Supercomputing for the Masses: Part 11. http://www.drdobbs.com/hpc-high-performance-computing/215900921, Date Last Accessed, 03/01/2010.

[12] Eric B. Ford. Parallel algorithm for solving kepler's equation on graphics processing units: Application to analysis of doppler exoplanet searches. *New Astronomy*, 14(4):406 – 412, 2009.

[13] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, and V.S. Pande. Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry*, 30:864–872, 2009.

[14] M.J. Harvey, G. Giupponi, and G. De Fabritiis. ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale. *Journal of Chemical Theory and Computation*, 5, 2009.

[15] Frank Jargstorff. Chapter 27. A Framework for Image Processing. http://http.developer.nvidia.com/GPUGems/gpugems_ch27.html, Date Last Accessed, 03/01/2010.

[16] NVIDIA Corporation. NVIDIA CUDA Best Practices Guide 2.3.

[17] NVIDIA Corporation. NVIDIA CUDA Programming Guide Version 2.3.1.

[18] NVIDIA Corporation. NVIDIA GeForce GTX 285. http://www.nvidia.com/object/product_geforce_gtx_285_us.html, Date Last Accessed, 03/04/2010.

[19] NVIDIA Corporation. NVIDIA GeForce GTX 200 GPU Architectural Overview. http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf, Date Last Accessed, 04/06/2010.

[20] NVIDIA Corporation. NVIDIA Compute PTX: Parallel Thread Execution ISA Version 2.0. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/ptx_isa_2.0.pdf, Date Last Accessed, 04/10/2010.

[21] NVIDIA Corporation. CUDA Occupancy Calculator. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls, Date Last Accessed, 3/16/2010.

[22] M.M. Papadopoulou, M. Sadooghi-Alvandi, and H. Wong. Micro-benchmarking the GT200 GPU. Technical report, Technical report, Computer Group, ECE, University of Toronto, 2009.

[23] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[24] S.S. Stone, J.P. Haldar, S.C. Tsao, W. m.W. Hwu, B.P. Sutton, and Z.-P. Liang. Accelerating advanced mri reconstructions on gpus. *Journal of Parallel and Distributed Computing*, 68(10):1307 – 1318, 2008. General-Purpose Processing using Graphics Processing Units.

APPENDIX A
Integral Kernel

```
1  template <unsigned int number_streams, unsigned int convolve>
2  __global__ void gpu__integral_kernel3(int mu_offset, int mu_steps,
3                                        int in_step, int in_steps,
4                                        int nu_steps, int total_threads,
5                                        double q_squared_inverse, double r0,
6                                        double *device__sinb,
7                                        double *device__sinl,
8                                        double *device__cosb,
9                                        double *device__cosl,
10                                       double *device__V,
11                                       double *background_integrals,
12                                       double *stream_integrals,
13                                       double *fstream_c,
14                                       double *fstream_a)
15 {
16   double *s_fstream_c = shared_mem;
17   double *s_fstream_a = &s_fstream_c[number_streams * 3];
18
19   if (threadIdx.x < number_streams * 3)
20     {
21       s_fstream_a[threadIdx.x] = fstream_a[threadIdx.x];
22       s_fstream_c[threadIdx.x] = fstream_c[threadIdx.x];
23     }
24
25   double bg_int = 0.0;
26   double st_int0 = 0.0;
27   double st_int1 = 0.0;
28   double st_int2 = 0.0;
29   double st_int3 = 0.0;
30
31   double sinb = device__sinb[threadIdx.x +
32                              ((mu_offset + blockIdx.x) * blockDim.x)];
33   double sinl = device__sinl[threadIdx.x +
34                              ((mu_offset + blockIdx.x) * blockDim.x)];
35   double cosb = device__cosb[threadIdx.x +
36                              ((mu_offset + blockIdx.x) * blockDim.x)];
37   double cosl = device__cosl[threadIdx.x +
38                              ((mu_offset + blockIdx.x) * blockDim.x)];
39
40   double cosb_x_cosl = cosb * cosl;
41   double cosb_x_sinl = cosb * sinl;
42
43   for (int i = 0; i < convolve; i++) {
44     double r_point = tex2D_double(tex_r_point,i,in_step);
45     double xyz2 = r_point * sinb;
```

```
46      double xyz0 = r_point * cosb_x_cosl - d_lbr_r;
47      double xyz1 = r_point * cosb_x_sinl;
48
49      double qw_r3_N = tex2D_double(tex_qw_r3_N,i,in_step);
50      {
51        double rg = fsqrtd(xyz0*xyz0 + xyz1*xyz1 + (xyz2*xyz2)
52                           * q_squared_inverse);
53        double rs = rg + r0;
54
55        bg_int += divd(qw_r3_N , (rg * rs * rs * rs));
56      }
57      if (number_streams >= 1)
58        {
59          // stream 0
60          double sxyz0 = xyz0 - s_fstream_c[0];
61          double sxyz1 = xyz1 - s_fstream_c[1];
62          double sxyz2 = xyz2 - s_fstream_c[2];
63
64          double dotted = s_fstream_a[0] * sxyz0
65            + s_fstream_a[1] * sxyz1
66            + s_fstream_a[2] * sxyz2;
67
68          sxyz0 -= dotted * s_fstream_a[0];
69          sxyz1 -= dotted * s_fstream_a[1];
70          sxyz2 -= dotted * s_fstream_a[2];
71
72          double xyz_norm = (sxyz0 * sxyz0) + (sxyz1 * sxyz1) + (sxyz2 * sxyz2);
73          double result = (qw_r3_N
74                           * exp(-(xyz_norm) *
75                                 constant_inverse_fstream_sigma_sq2[0]));
76          st_int0 += result;
77        }
78      if (number_streams >= 2)
79        {
80          //stream 1
81          double sxyz0 = xyz0 - s_fstream_c[3];
82          double sxyz1 = xyz1 - s_fstream_c[4];
83          double sxyz2 = xyz2 - s_fstream_c[5];
84
85          double dotted = s_fstream_a[3] * sxyz0
86            + s_fstream_a[4] * sxyz1
87            + s_fstream_a[5] * sxyz2;
88
89          sxyz0 -= dotted * s_fstream_a[3];
90          sxyz1 -= dotted * s_fstream_a[4];
91          sxyz2 -= dotted * s_fstream_a[5];
92
93          double xyz_norm = (sxyz0 * sxyz0) + (sxyz1 * sxyz1) + (sxyz2 * sxyz2);
94          double result = (qw_r3_N
95                           * exp(-(xyz_norm) *
96                                 constant_inverse_fstream_sigma_sq2[1]));
97          st_int1 += result;
```

```
98          }
99        if (number_streams >= 3)
100         {
101           //stream 2
102           double sxyz0 = xyz0 - s_fstream_c[6];
103           double sxyz1 = xyz1 - s_fstream_c[7];
104           double sxyz2 = xyz2 - s_fstream_c[8];
105
106           double dotted = s_fstream_a[6] * sxyz0
107             + s_fstream_a[7] * sxyz1
108             + s_fstream_a[8] * sxyz2;
109
110           sxyz0 -= dotted * s_fstream_a[6];
111           sxyz1 -= dotted * s_fstream_a[7];
112           sxyz2 -= dotted * s_fstream_a[8];
113
114           double xyz_norm = (sxyz0 * sxyz0) + (sxyz1 * sxyz1) + (sxyz2 * sxyz2);
115           double result = (qw_r3_N
116                            * exp(-(xyz_norm) *
117                                  constant_inverse_fstream_sigma_sq2[2]));
118           st_int2 += result;
119         }
120       if (number_streams >= 4)
121         {
122           //stream 3
123           double sxyz0 = xyz0 - s_fstream_c[9];
124           double sxyz1 = xyz1 - s_fstream_c[10];
125           double sxyz2 = xyz2 - s_fstream_c[11];
126
127           double dotted = s_fstream_a[9] * sxyz0
128             + s_fstream_a[10] * sxyz1
129             + s_fstream_a[11] * sxyz2;
130
131           sxyz0 -= dotted * s_fstream_a[9];
132           sxyz1 -= dotted * s_fstream_a[10];
133           sxyz2 -= dotted * s_fstream_a[11];
134
135           double xyz_norm = (sxyz0 * sxyz0) + (sxyz1 * sxyz1) + (sxyz2 * sxyz2);
136           double result = (qw_r3_N
137                            * exp(-(xyz_norm) *
138                                  constant_inverse_fstream_sigma_sq2[3]));
139           st_int3 += result;
140         }
141     }
142
143     //define V down here so that one to reduce the number of registers,
144     //because a register will be reused
145     int nu_step = (threadIdx.x + (blockDim.x * (blockIdx.x + mu_offset)))
146       % nu_steps;
147     double V = device__V[nu_step + (in_step * nu_steps)];
148     int pos = threadIdx.x + (blockDim.x * (blockIdx.x + mu_offset));
149     background_integrals[pos] += (bg_int * V);
```

```
150     if (number_streams >= 1)
151       {
152         stream_integrals[pos] += st_int0 * V;
153         pos += total_threads;
154       }
155     if (number_streams >= 2)
156       {
157         stream_integrals[pos] += st_int1 * V;
158         pos += total_threads;
159       }
160     if (number_streams >= 3)
161       {
162         stream_integrals[pos] += st_int2 * V;
163         pos += total_threads;
164       }
165     if (number_streams >= 4)
166       {
167         stream_integrals[pos] += st_int3 * V;
168       }
169   }
```

APPENDIX B
Hardware Specifications

AMD Athlon(tm) 64 X2 Dual Core Processor 4400+
2300 Mhz
L2: 512 KB (per core)
SSE2, SSE3, MMX
4 GB RAM
Gentoo GNU/Linux kernel 2.6.30 and Windows XP x64

Nvidia GeForce GTX 285 (GT200 GPU)
240 Shader cores
Compute Capability 1.3 (Double precision)
648 MHz Core Clock, 1242 MHz Mem Clock, 1476 MHz Shader Clock
1024 MB Graphics Memory GDDR3
PCI Express 2.0
78 GFLOP/s (Double Precision) 933 GFLOP/s (Single Precision)[1] [19]

Nvidia GeForce 8800 GTX (G80 GPU)
128 Shader cores
Compute Capability 1.0 (Single Precision only)
576 MHz Core Clock, 900 MHz Mem Clock, 1350 MHz Shader Clock
768 MB Graphics Memory GDDR3
PCI Express 1.0
518 GFLOP/s (Single Precision)[2] [19]

ATI Radeon HD 5870
1600 shader cores
850 Mhz Core clock, 1200 Mhz Mem clock
1024 MB Graphics Memory GDDR5
PCI Express 2.0
544 GFLOP/s (Double Precision) 2720 GFLOP/s (Single Precision)[3]

[1] http://www.nvidia.com/object/product_geforce_gtx_285_us.html
[2] http://www.nvidia.com/page/geforce_8800.html
[3] http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx