A MULTI-GPU COMPUTE SOLUTION FOR OPTIMIZED GENOMIC SELECTION ANALYSIS

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Trevor DeVore

June 2014

© 2014

Trevor DeVore

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	A Multi-GPU Compute Solution for Optimized Genomic Selection Analysis
AUTHOR:	Trevor DeVore
DATE SUBMITTED:	June 2014
COMMITTEE CHAIR:	Assistant Professor Chris Lupo, Ph.D., Department of Computer Science
COMMITTEE MEMBER:	Professor Bruce Golden, Ph.D., Department of Dariy Science
COMMITTEE MEMBER:	Associate Professor John Seng, Ph.D., Department of Computer Science

ABSTRACT

A Multi-GPU Compute Solution for Optimized Genomic Selection Analysis

Trevor DeVore

Many modern-day Bioinformatics algorithms rely heavily on statistical models to analyze their biological data. Some of these statistical models lend themselves nicely to standard high performance computing optimizations such as parallelism, while others do not. One such algorithm is Markov Chain Monte Carlo (MCMC). In this thesis, we present a heterogeneous compute solution for optimizing GenSel, a genetic selection analysis tool. GenSel utilizes a MCMC algorithm to perform Bayesian inference using Gibbs sampling.

Optimizing an MCMC algorithm is a difficult problem because it is inherently sequential, containing a loop carried dependence between each Markov Chain iteration. The optimization presented in this thesis utilizes GPU computing to exploit the data-level parallelism within each of these iterations. In addition, it allows for the efficient management of memory, the pipelining of CUDA kernels, and the use of multiple GPUs. The optimizations presented show performance improvements of up to 1.84 times that of the original algorithm.

Keywords: Bioinformatics, Multi-GPU, Markov Chain Monte Carlo

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The field of Bioinformatics relies heavily on statistical modeling to analyze biological data. To improve the accuracy of these statistical models, larger quantities of data are being analyzed by increasingly complex algorithms. The Markov Chain Monte Carlo (MCMC) family of algorithms is a good example of this. Recently, this family of algorithms has seen increased usage in Bioinformatics applications, and as such, a great deal of effort has been put into the optimization of MCMC.

When optimizing algorithms, exploiting either task-level or data-level parallelism allows for major speedups. Unfortunately, not all algorithms fall into the embarrassingly parallel category, and others resist parallelization altogether. MCMC algorithms fall into the latter category due to their inherently iterative nature and numerous data dependencies. Previous research in parallelizing MCMC algorithms has shown that good speedups are possible if the algorithm is designed around parallel programming principles [7, 12, 13].

This thesis will explore an alternative method for optimizing MCMC that does not require major algorithm redesign. These optimizations will be achieved by isolating sections of the algorithm that will benefit most from parallelization. Once these sections are found, the proper parallelization technique will be applied. The general idea behind this method is to shrink the computation time required by each link of the Markov Chain, reducing the runtime for the entire algorithm.

The optimizations explored in this thesis leverage multi-GPU computing, kernel (GPU function calls) pipelining, and efficient memory management to exploit data-level parallelization. The analysis performed by GenSel requires the computation of many matrix operations. The size of these computations scale with the number of observations given as input to the program, and are executed for every genetic marker in each iteration of MCMC.

The remainder of this thesis will be organized as follows: Chapters 2 and 3 will discuss background information and related works respectively. Chapters 4, 5, and 6 will detail the methodologies used to analyze optimization opportunities, an overview of the algorithm being optimized, and the implementation of a host of GPU optimizations. Next, Chapters 7 and 8 will discuss the results of the optimizations implemented and the test environment used to generate these results. Finally, Chapters 9 and 10 discuss potential future optimizations and provide conclusions for the optimizations provided in this thesis.

Chapter 2

Background

This Chapter will provide background information on both the statistical analysis behind GenSel and the CUDA programming language.

2.1 Statistical Analysis

Bayesian inference is a widely used form of statistical inference that is used in a number of computer science fields, one of which is Bioinformatics. Bayesian inference is derived from the Bayes theorem, which essentially allows for the updating of prediction probabilities based on previous experience. The probability of the original prediction is called the prior, and the probability with knowledge of the previous experience is called the posterior.

Bayesian inference allows for relatively simple yet accurate predictions of the effect specific genes have on a given trait [6]. Unfortunately, simply implementing a Bayesian inference algorithm is not very feasible as there is no guarantee of a closed form solution for the posterior. In the cases where a solution is possible, integration over multiple dimensions is required, which can lead to very long computation times. In order to side step this issue, GenSel uses a Gibbs sampler, which is a member of the Markov Chain Monte Carlo (MCMC) family of algorithms. MCMC algorithms provide a way to compute the posterior distribution when it is not possible to sample the distribution of interest directly.

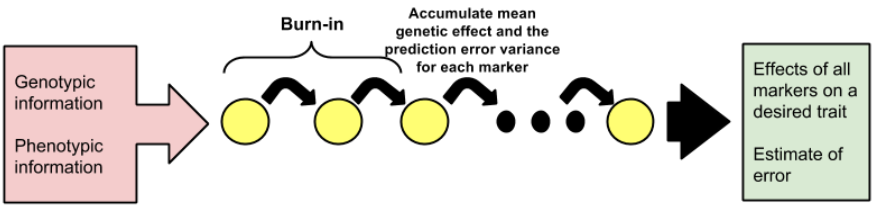

Figure 2.1: General overview of algorithm

While the use of Gibbs sampling allows for more efficient computation of the posterior distribution [6], it is by no means a trivial computation. To find the correct posterior distribution, sampling must be done until the probability distribution converges. To make matters worse, determining the number of iterations needed for the distribution to converge is a difficult problem [8]. In addition, due to the bias introduced by the initial values in the first iteration, the beginning portion of the chain cannot be used.

In GenSel, each genetic marker is evaluated to determine if it has an effect on the desired trait. A statistical model is then built containing the mean effect all of the markers have on the desired trait. In addition to this, the estimated errors of prediction are also captured. To generate this model, GenSel iterates through each genetic marker and determines whether or not it is to be included in the model. This decision is made based on whether or not the traits fully-conditional posterior exceeds a set threshold [6]. This process is done multiple times, forming links in a chain where each link builds on the results of the previous one. This algorithm is shown in Figure 2.1 and is the target of the optimizations presented in this thesis.

When studying the way a MCMC algorithm works, it is easy to see the inherent issues with parallelization. Each link in the chain relies on values computed

in the previous link. Instead, the optimizations discussed focus more on the work being performed inside each link of the chain. In GenSel, a majority of the work is performed in the BayesC algorithm, which is presented in [6]. This adds additional data dependencies between the analysis of each genetic marker. Fortunately, this dependence is only present when the marker being analyzed is to be included in the current model or was included in the previous model. This allows for parallelization optimizations that would otherwise not be possible.

2.2 CUDA

NVIDIA's Compute Unified Device Architecture (CUDA) is a parallel computing architecture that allows a graphics processing unit (GPU) to be used by developers. GPUs can contain thousands of cores, allowing them to process large amounts of data in parallel, as that is their intended purpose in graphics. To aid developers and improve portability, CUDA threads have been abstracted away from the physical hardware. The CUDA thread model consists of a two-dimensional grid of three-dimensional blocks. The dimensions of the grid are defined by the developer for each kernel. Each block in the grid contains a three-dimensional layout of threads. Much like the grid, the dimensions of the blocks are also defined by the developer. While these are highly configurable, some limitations on maximum dimensions are set, and vary based on the version of CUDA being used. The thread, the basic unit in the CUDA thread model, is what is executed on each core in the GPU. Behind the scenes, 32 threads are grouped into what is called a warp. All threads in a warp are executed at the same time, and each thread executes the same instruction. Due to this, care must be taken that the work given to each thread does not diverge.

CUDA operations on the GPU can be executed asynchronously from the CPU, allowing both to work on separate tasks at the same time. This is integral to squeezing the maximum performance out of a system. In order to allow asynchronous data transfers, memory on the CPU must be pinned, which forces the system to keep that memory in RAM and not move it to swap space. To further facilitate asynchronous operations, CUDA allows users to queue GPU operations in a stream. Streams are associated with a specific CUDA device, and each device can have multiple streams. CUDA kernels (functions) and memory operations executed on the same stream are guaranteed to execute in a first-in-first-out manner. Tasks in different streams can be executed concurrently on the device. This allows for better utilization of the GPU's compute resources. Efficient use of streams is critical in applications that wish to utilize both the CPU and GPU simultaneously.

GPU memory is different than that of a traditional system. The majority of a GPUs memory is global memory. This would correlate to the RAM in a traditional system, and is the slowest but largest memory on the GPU. Shared memory, on the other hand, is only accessible by threads in the same block, but is much faster and much smaller. This would most closely correlate to a cache in a traditional system. One major difference is that the shared memory in CUDA is controllable by the developer. This allows key pieces of data to be put in shared memory for significantly faster access times. Additionally, if multiple threads are working on the same data, changes made by one thread in a block are immediately visible to all other threads in the block.

Chapter 3

Related Works

In researching the topic of this thesis, it has become clear that very little research has been done in this area. This does not come as too much of a surprise given the specific nature of the problem. Another potential reason for the lack of published work in this specific area is the competitive advantage that can be gained from this research.

As with many bioinformatics algorithms, this algorithm is comprised of many components from many fields of study. This section will discuss the work related to the major components of this algorithm.

3.1 Statistical Analysis of Genetics

One of the largest components of this algorithm is the genetic analysis performed using various statistical methods. While this does not directly relate to the work of this thesis, it is crucial to the algorithm being optimized and is therefore important to discuss.

While statistics has been used since the 1800s for genetic analysis in the form of probabilities, the application of statistical modeling to genetics began to increase in the late 1900s as computers became more available [4, 11]. As the power of computers has increased, larger datasets are able to be analyzed using more complex statistical approaches than before, which yields increasingly more

exact results. Currently, compute time is the limiting factor to these large scale statistical analyses, as can be seen by the topic of this thesis.

A few of the more recent related works in this are come, unsurprisingly, from the same authors of this algorithm. In this work, the breeding values of animals were estimated by analyzing which genomic regions have an effect on a specific trait of the animal, and how this information can be used to determine which animals should be bred [3].

3.2 MCMC on the GPU

Another large component of this algorithm is the use of graphics processing units (GPUs) to improve the runtime of statistical computations. One way to achieve these speedups is to move the entire statistical computation to the GPU for processing. While this is in fact the fastest way to do these analyses, it is not always feasible due to data dependencies or algorithm divergence during the analysis.

While the parallelization of MCMC is very difficult, it has become a hot topic for research in recent years. One approach is to shrink the computation time for each iteration of the MCMC chain [13]. This method does provide good performance improvements, but the speedup is still bounded by Amdahl's quantity. This means that the performance improvements can only be as great as the portion of the algorithm that is optimized. In the case of MCMC, which is a very iterative algorithm, only small portions can be optimized, and therefore only small speedups can be expected.

Another approach is to restructure the MCMC algorithm entirely to make it more GPU friendly. DE-MCMC, as proposed by Zhu and Li, uses differential evolution on a population-based MCMC sampler, which allows for parallelization

[16]. While massive speedups can be seen from this approach, it is not always possible to restructure the algorithm in this manner. The algorithm optimizations discussed can only be applied to certain members of the MCMC algorithm family, which can limit the usefulness of this type of approach.

3.3 Multi-GPU Computing

Leveraging multiple GPUs can significantly improve the runtime of large data computations. One approach to using multiple GPUs is to decompose a single large problem into multiple pieces, and allow each GPU to compute a piece of the problem [1]. This is best suited to problems that contain only a few very large computations, and will often require inter-GPU communication. Another approach is to allow each GPU to compute a single problem on its own [13]. This is most beneficial when a large number of difficult computations must be performed at the same time.

Another consideration when using multiple GPUs is load balancing. Without load balancing, the problem can only be solved as fast as the slowest GPU can solve it. This means that a single slow GPU can hamper the performance of multiple faster GPUs. Dynamic load balancing helps tackle these issues [2]. This approach can improve performance when hardware or workloads are unbalanced.

3.4 Heterogenous Computing

In order to squeeze the most performance possible from a system, both the CPUs and GPUs must be utilized to their maximum potential. Many high performance compute solutions, both in statistics and other fields, leverage both the CPU and GPU in their optimizations [13–15]. Heterogeneous compute solutions provide the

maximum performance by not only allowing both the CPU and GPU to perform useful work at the same time, but also by allowing each to play to its strengths. The CPU is much better at exploiting coarse-grained parallelism, while the GPU is much stronger when it comes to fine-grained parallelism [13].

Chapter 4

Methodology

Although the algorithm used in GenSel is not completely parallelizable, many smaller areas of the algorithm contain various levels of parallelism. In order to exploit these areas, the correct programming paradigms must be matched with the corresponding levels of parallelism. When searching for these areas of parallelism, it is important to analyze the amount of computation time spent in an area, whether or not the code diverges, and what dependencies need to be maintained.

Performing parallelization optimizations on algorithms generally limits the flexibility and portability of the algorithm. The optimizations made are based largely on the hardware available. This leads to an optimized algorithm requiring a specific set of hardware. The GenSel algorithm exhibits both fine-grained and coarse-grained areas of parallelism, therefore a heterogeneous compute solution has been implemented, utilizing both CPU threading and GPU computing. This thesis will focus primarily on the GPU optimizations used in the solution. The hardware used in these optimizations is discussed further in Chapter 7.

GPUs are extremely effective when an algorithm exhibits areas of data level parallelism with no divergence [9]. The thousands of cores available on modern GPUs allows for very high levels of throughput on large computations. In the CUDA programming model, each thread on the GPU is organized into a warp.

Each of these warps executes the same instruction at the same time. In the case of branches or other forms of divergence, some threads in the warp will not be doing any useful work.

In order to help increase the utilization of a GPU, CUDA includes a feature called streams. Every kernel call to a device is sent over a stream, and each device can have multiple streams associated with it. Kernels and other memory functions that are sent down the same stream are guaranteed to be executed in order, but there is no execution order guarantee for kernels sent down different streams. A major advantage of using streams is that, hardware permitting, this allows the GPU to run multiple kernels concurrently. Additionally, this allows for kernels and memory operations to be executed asynchronously from the CPU, allowing it to continue performing useful tasks while the GPU is executing its kernels.

The ability to execute kernels and perform memory transfers asynchronously allows for maximizing GPU utilization without hindering the CPUs ability to work. One of the biggest issues with using the GPU is the data transfer latency. By asynchronously executing these transfers ahead of when they needed on the GPU, most of this transfer latency can be hidden, vastly improving the latency of GPU computation.

Additionally, it is important to re-analyze the system's bottlenecks after an optimization is made. In some cases, the optimizations will speedup one section of an algorithm, shifting the bottleneck to a different component. In the case of this implementation, even after optimizing computations by parallelizing them on the GPU, the bottleneck still existed in the same area. To remedy this, the implementation was extended to support multiple GPUs, which helps further reduce the bottleneck.

Chapter 5

Algorithm Overview

This chapter contains a brief description of the algorithms being used in GenSel. Algorithm 1 shows the original algorithm in pseudo code.

Algorithm 1 Initial Bayes Algorithm

```
1: for each marker do
2:     compute probability marker is included in model
3:     if marker is included in model then
4:         compute mean genetic effect
5:         compute error prediction variance
6:         include marker in model
7:     else
8:         remove marker from model
9:     end if
10: end for
```

It is important to understand the basics of what occurs within each iteration of the MCMC algorithm. Without this, many of the considerations and design decisions made in later sections of the thesis may not make sense. In the initial algorithm, a single **for** loop iterates through each genetic marker that is being analyzed. For each marker, the probability that it should be included in the current model is computed. The statistics behind this are covered further in [6], but for the purposes of this discussion, the focus will be on the large dot product

computation between observations of the genetic marker and the current state of the statistical model. This probability is then compared to a threshold value in the **if** statement. If the probability surpasses the threshold, the marker is included in the model, which in turn updates the current model, which is then used in the next marker's probability computation. If the probability does not surpass the threshold then the marker is not included in the model. This leads to what will be referred to as a sporadic loop dependency.

5.1 Increasing Data Level Parallelism

The statistics behind the inclusion of markers in the statistical model cause the true branch to execute about once every 10 iterations on average. While this still does not allow for the entire loop to be parallelized, it does allow for chunks of the loop to be computed together to improve performance. Algorithm 2 shows a restructured algorithm that allows for chunks of data to be computed in parallel.

In the new algorithm, a chunk of probabilities are computed at the same time. Once the probabilities are computed, each marker in the chunk is tested to see if it would have surpassed its threshold. If one of the markers is to be included in the model, which would in turn require the model to be updated, that marker is considered the terminal marker of the chunk. All of the computations in the chunk up to the first terminating marker are valid, while the rest are discarded as their dot products are invalid. The included marker is added to the model, and the process repeats itself until every marker has ben evaluated. By carefully selecting the chunk size, it is possible to use about 85% of the computations on average. Additional details on the precomputing of dot products can be found in

Algorithm 2 New Bayes Algorithm

```
1: while markers processed < total number of markers do
2:    for each marker in chunk do
3:       compute probability marker is included in model
4:       if marker is to be included in model then
5:          set as terminating marker for chunk
6:       end if
7:    end for
8:    begin precomputing next chunk's dot products
9:    initiate observation data transfer to GPUs
10:   for each marker before terminal marker do
11:      if marker is to be included in model then
12:         compute mean effect
13:         compute error prediction variance
14:         include marker in model
15:      else
16:         remove marker from model
17:      end if
18:   end for
19:   update markers processed
20:   advance chunk to terminal marker
21: end while
```

Section 6.1 and Section 6.3 and additional information on selecting chunk sizes is available in Section 8.1.1.

Chapter 6

Implementation

This chapter contains an in-depth discussion of the various GPU optimizations used to improve the runtime of GenSel.

6.1 Parallelizing Dot Product Computation

The dot product computation that is used to determine whether or not a marker is to be included in the model is the primary focus of the optimizations performed in this chapter. The dot products being computed scale with the number of observations in the analysis, which can range from 1,000 to 1,000,000 observations. The GPU is a powerful tool when attempting to do linear algebra operations. The dot product operation, for instance, consists of many small operations that are completely independent of each other. This is well suited for the GPU as it is able to leverage the massive number of cores available to perform these operations simultaneously.

This implementation leverages the cuBLAS linear algebra library [10], which provides an optimized CUDA dot product implementation. Unfortunately, this library is not a simple plug and play replacement for a CPU dot product implementation. Data must still be transferred across the PCIe bus to and from the GPU. Transferring the data across the bus can cause major slow downs if not done correctly. Efficient and well placed memory transfers are the key to

the speedups seen in this optimization. Without hiding memory transfers, the latency of each dot product would be significantly increased, most likely resulting in a slowdown for any dataset. Sections 6.4, 6.5, and 6.6 go into more detail on the memory management used in this implementation.

6.2 Hiding Memory Transfers

In most applications that utilize the GPU, a large portion of time is spent transferring memory back and forth between the CPU and GPU. There are several known techniques for amortizing this cost. One option is using asynchronous memory transfers. These types of transfers return control to the CPU immediately, instead of waiting for the transfer to complete. This allows the CPU to do additional work while the data is transferring to the device. The GPU can also be performing work while data is being transferred. Additionally, three of the four GPUs used in this paper have two copy engines, enabling them to transfer data to and from the CPU simultaneously.

In this implementation, all memory transfers occur asynchronously to overlap data transfer with execution as much as possible. To minimize or eliminate the amount of time that the CPU is waiting on the GPU or vice versa, they are performed as early as possible. This optimization operating in conjunction with the data buffering technique discussed in Section 6.5 all but eliminate the time the GPU is waiting for data to perform its calculations. The initial transfer of data to the GPU is done during the setup phase of the GenSel application. All subsequent transfers occur while the CPU is updating the current model and the GPU is precomputing the next chunk's dot products.

6.3 Precomputing Dot Products

The purpose of chunking genetic markers together, as discussed in Section 5.1, is to allow multiple dot product computations to be performed at the same time, which can be done quickly by leveraging the GPU. In order to fully benefit from the high throughput of the GPU, the high latency of the system's bus must also be taken care of. Instead of sending the data to the GPU, waiting for the GPU to compute the dot products, and then shipping the results back to the CPU when a dot product result is requested, this implementation precomputes the next chunk's dot products as soon as possible. As can be seen in Algorithm 2, this occurs directly after the first **for** loop, on line 8. At the completion of this loop, the terminating marker of the chunk is known. Any markers in the chunk after the terminating marker are discarded, the model attribute vector on the GPU is immediately updated, and the GPU begins computing the next chunk of dot products, transferring the results back as soon as they are complete. Since this is all done asynchronously, the CPU is free to continue on to the bottom **for** loop as the GPU is computing the next chunk's dot products. Then, on the next iteration of the **while** loop, when the dot products for the next chunk are requested, the CPU does not have to wait on the GPU and data transfers, as the computations should already be complete. More discussion on tuning the chunk size is given in Section 8.1.1.

6.4 Pipeling Kernel Calls

Making sure the GPU is saturated with work is extremely important for maximum efficiency. Traditionally, a single GPU kernel is launched at a time. With the addition of asynchronous calls and streams, it is possible to execute multiple

kernels at once. In CUDA, a stream allows a queue of jobs to be created for a device. GPU operations that are launched in a stream are guaranteed to be executed in order, and a device can execute tasks from multiple streams concurrently. This implementation launches each dot product of a chunk in a separate stream. This allows for maximum saturation of the GPU as it is given a steady stream of work as opposed to executing a dot product and returning control to the CPU, just to have the CPU launch the next dot product kernel. At a high level, this can be thought of as a pseudo pipeline for the GPU. Ultimately, this reduces the time wasted between dot product kernels and considerably improves the latency for each chunk of dot products to be computed. While it is possible to create a specialized CUDA kernel that can compute a chunk size worth of dot products, this method is less flexible and would require code re-writes for changes to the size of each chunk or the underlying hardware in the system. Instead, by leveraging CUDA streams, more of the dot product concurrency is handled by the hardware itself.

6.5 Circular Buffering of Data on the GPU

As discussed in Section 6.2, one of the largest drawbacks to GPU computing is amount of time it takes to transfer data across the bus. Optimally, all of the data needed for any GPU operations would be transferred to the GPU ahead of time, allowing kernels to begin executing as soon as possible, without awaiting memory transfers. The first iteration of this implementation contained a single bulk data transfer of all the observation data to the GPU at the beginning of the program. This allowed for excellent performance as the only data transfers needed were the occasional update of the statistical model on the GPU and the dot product results. However, as the input data for GenSel becomes large, this

implementation will quickly fail. For reference, the size of the observation matrix can grow to about 40 GB of data at 1,000,000 observations and 10,000 genetic markers, which is only a fraction of the eventual expected data. While compute class GPUs tend to have much larger amounts of memory than desktop class GPUs, neither are anywhere near the 40 GB or more required by GenSel.

To deal with this issue, this implementation instead uses a circular buffer to store observation data on the GPU. The size of this buffer is defined as the chunk size multiplied by a multiplier. This multiplier must be large enough to allow each observation vector to be completely transferred to the GPU before it is needed in the dot product computation. This allows for the benefits of already having the necessary data on the GPU with a much smaller memory footprint. Further analysis on tuning the size of the circular buffer can be found in Section 8.1.3.

Management of the circular buffer is done through a separate stream. All of the data transfers to the GPU are done in this stream. At the beginning of each iteration of the MCMC chain, the circular buffer is filled with the first **n** values, where **n** is the total size of the buffer. Due to this, it is important to ensure that the buffer is correctly sized to avoid extra initial delays as the buffer is initially filled at the beginning of each MCMC iteration. Once the terminating marker of the chunk is found, each marker up to and including the terminating marker is considered consumed, and the next values are sent to take their place in the buffer. The advantage of using a separate stream for transferring data is that the GPU can facilitate the data transfer at the same time as it is executing other kernels.

Figure 6.1 provides a visual for the circular data buffer. In this example, the chunk size is set to six and the chunk size multiplier is set to five, leading to a 30

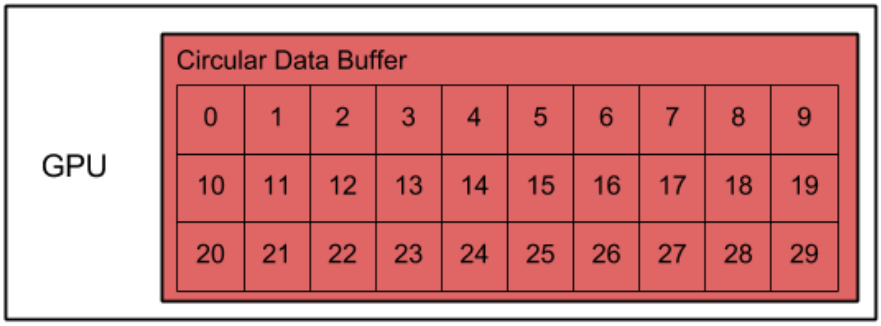

Figure 6.1: GPU memory layout illustrating how data is stored a circular data buffer

slot data buffer on the GPU. If the first five entries of the buffer are consumed, they would be replaced by the observation vectors for markers 30-34. As can be seen, this allows for much more efficient memory usage on the GPU.

6.6 Multi-GPU and Striping

Executing a batch of dot product computations on the GPU scales extremely well when additional GPUs are added to the system. This will allow more dot products to be computed in a shorter amount of time by distributing the computations across the GPUs. Due to the flexibility and design decisions discussed in the previous sections, additional GPUs can be easily leveraged for massive performance gains. While additional considerations must be made to accommodate multiple GPUs, the implementation can largely remain the same.

For starters, stream data structures must be capable of scaling to multiple devices. This includes an individual data stream for each GPU, and streams for each dot product that needs to be computed, which is based on the chunk size. The simplest way to do this is a 2D array of streams, where the **x**-dimension is

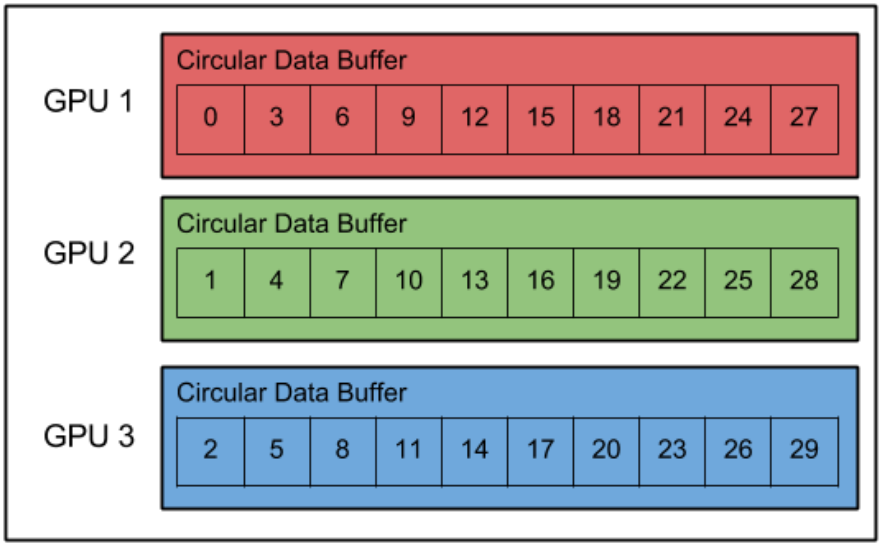

Figure 6.2: Striped memory layout for multiple GPUs

the number of compute streams per GPU and the **y**-dimension is the number of GPUs. Since there is a known maximum for how many dot products will be executed on each GPU for a given chunk size, only a set number of compute streams must be opened to each GPU. If, for example, three GPUs are in use and the chunk size is six, only two compute streams are required. When each dot product kernel is launched, a stream associated with the GPU containing the necessary observation vector must be chosen. This can easily be done by using modulo arithmetic, as the observation vectors are distributed consistently across the GPUs. Observation data streams can be organized and used in the same manner as the computation streams.

Next, the circular buffer design must be slightly altered. Since each GPU will only be computing the dot product for a portion of the current chunk, each GPU only needs the observation data for a fraction of the total observations. This allows us to stripe the data across the GPUs in the system. Additionally, the fact that observations will be consumed in order allows the implementation

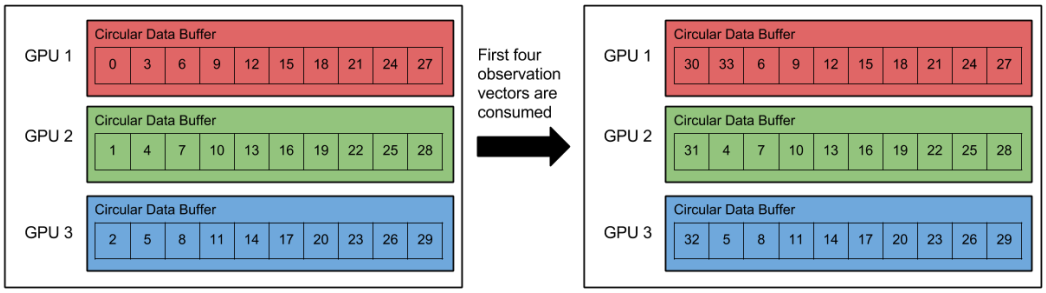

Figure 6.3: Circular buffer observation vector replacement on multiple GPUs

to naturally load balance across the GPUs. Figure 6.2 provides an example of how data is striped across the GPUs. In this example, the system contains three GPUs, each of which are holding a third of the observation vectors for each marker. Figure 6.3 illustrates how observation vector replacement works in the multi-GPU circular buffer design. In this example, the chunk size is set to six and the multiplier is set to five. If, after computing the first chunk's dot products, the first four observation vectors are to be consumed, the observation vectors for markers zero through three would be replaced, on their respective GPUs, with the observation vectors for markers thirty through thirty-three. Then, the dot products for the next chunk (i.e., for markers four through nine) would be computed. As can be seen, no matter how much of a chunk is consumed, the next chunk will be distributed evenly across the GPUs, unless there is not a full chunk's worth of observations remaining. In this case, the load is still distributed as best as possible. Managing the data in this manner also helps further reduce the memory footprint on each GPU, as the space needed is divided by the number of GPUs.

The additional performance improvements gained from leveraging multiple GPUs is well worth the slightly more complicated GPU initialization and memory management. These extensions are also fully scalable to any number of GPUs,

including a single GPU. Ultimately, this allows more dot products to be computed in the same amount of time, and allows for larger chunk sizes to be used without a negative impact on performance.

Chapter 7

Experimental Setup

7.1 Test System

The results gathered by this thesis were all performed on one test system. This system contains two Intel Xeon E5-2650 CPUs, each of which contains 8 physical cores. Both processors are clocked at 2.00 GHz and support Intel's Hyper-Threading technology. This system also contains 64 GB of RAM. In addition, the system also contains 4 GPUs. A majority of the tests were done using the 3 compute class cards in the system, which includes a NVIDIA Tesla K40 and two NVIDIA Tesla K20Xs. The Tesla K40 GPU has 2,880 cores clocked at 745 MHz and 12 GB of GDDR5 of memory, while the Tesla K20X GPUs have 2,688 cores clocked at 732 MHz and 6 GB of GDDR5 memory. To demonstrate scalability, the fourth GPU in the system was also used to perform a four GPU run. This is a GeForce GTX TITAN Black, which contains 2,880 cores clocked at 889 MHz, and 6 GB of GDDR5 memory. The four GPUs are connected to rest of the system via two PCI buses. The Tesla K40 and GTX TITAN Black share one bus, and the two K20Xs share the other. This can lead to some additional bandwidth congestion when using multiple GPUs that are on the same bus.

All source code used in these experiments were compiled using GCC version 4.8.2 and CUDA 6.0. The test system was running Arch Linux with version 3.14.0 of the Linux kernel, and Nvidia driver version 334.21.

7.2 Test Data

The purpose of these optimizations is to help GenSel accommodate the larger quantities of data that are expected to be available in the near future. In order to perform these tests before this data is available, simulated genotypic and phenotypic data are used instead. A data generator has been created that can simulate datasets of any number of genetic markers and observations. This allows for the testing of a number of different dataset sizes. To keep runtimes within reasonable bounds, the number of genetic markers simulated has been limited to 10,000 markers.

7.3 Reference Implementation and Validation

The initial implementation that only utilizes the CPU was developed with the Eigen library [5]. This library is highly optimized and is one of the fastest linear algebra libraries available, taking advantage of vectorized instructions, cache optimizations, and loop unrolling.

For each dataset tested, both the initial implementation and the optimized implementation were run, and the results of both were compared for validation. In addition, timing results for both algorithms were stored. The recorded timing data only includes the runtime of the algorithm, file I/O is excluded from the timing.

Chapter 8

Results

This chapter of the thesis presents the results of the optimizations made to GenSel. Any speedup results given in this chapter are the speedup of the optimized algorithm relative to the initial CPU only implementation that uses Eigen. The timing results used to compute these speedups only contain the time for the algorithm to run, and therefore do not include file read in time. The times do however contain all of the bus transfers between the CPU and GPU, as well as the set up needed for the GPU. Unless otherwise stated, all of the results shown in this chapter were done with the three Tesla class GPUs, a chunk size of six and a buffer size of four chunks.

Figure 8.1 shows the overall speedup of the optimized implementation over the initial implementation. The datasets tested range from 1,000 to 1,000,000 observations, and the number of genetic markers ranges from 500 to 10,000. As discussed earlier, the reason for these cutoffs is to keep the runtimes reasonable. There are no actual limitations in the algorithm, which can easily scale to more observations and genetic markers. As can be seen, the crossover point is at about 150,000 observations, and the best case speedup is about 1.84.

The similar shapes of the curves in Figure 8.1 illustrate that the optimizations made scale primarily with the number of observations in the model. This also means that increasing or decreasing the number of genetic markers does

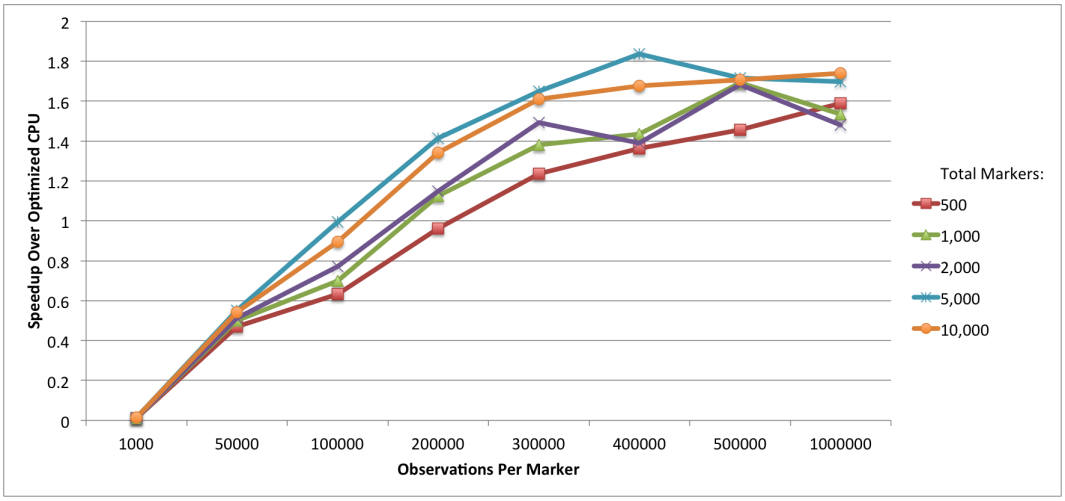

Figure 8.1: Overall speedup with varied number of markers

not really affect the performance of the optimizations. This occurs because the chunking process that takes place in the optimized algorithm essentially decreases the number of iterations required to compute each link of the MCMC chain, and by shrinking the time it takes to compute each link, the chain can be computed in a much shorter amount of time. This is important because the number of observations is expected to continue to rise, but the number of genetic markers being analyzed is expected to eventually decrease as the more important genetic markers for a given trait are isolated.

For perspective, the actual runtimes for the 500 marker dataset is shown in Table 8.1. The runtimes scale in a roughly linear manner for the number of genetic markers (e.g., the 1000 marker runtimes will take about two times as long).

Number of Observations	CPU Runtime (s)	GPU Runtime (s)	Speedup
1,000	14	1,284	0.01
50,000	797	1,692	0.47
100,000	1,579	2,493	0.63
200,000	3,184	3,316	0.96
300,000	4,761	3,855	1.24
400,000	6,375	4,678	1.36
500,000	7,920	5,435	1.46
1,000,000	15,653	9,850	1.59

Table 8.1: Runtimes for 500 marker data set

8.1 Tuning

In high performance computing, tuning an application can make or break an optimization. It is crucial to tailor the implementation to the system it is running on. To make tuning the application easier, the optimizations have been designed to allow easy tweaking without changing any code. The primary tuning variables for these optimizations are: the chunk size, the number of GPUs, and the buffer size on each GPU.

8.1.1 Tuning the Chunk Size

The chunk size is one of the most critical tuning variables. If the chunk size is set to be too large, a larger portion of each chunk will need to be discarded. On the other hand, if the chunk size is set to be too small, little benefit will come from using the system, and the ability to leverage multiple GPUs will be diminished.

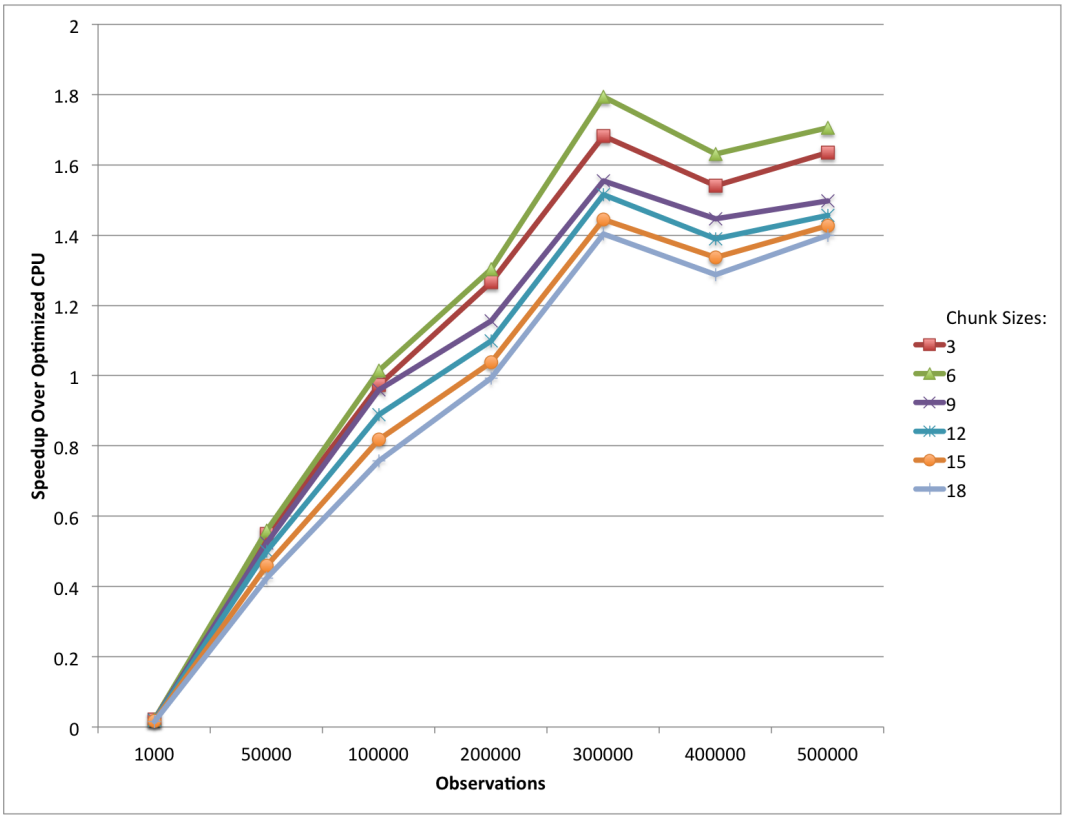

Figure 8.2: Effect of different chunk sizes on speedup

When tuning the chunk size, the number of GPUs used is also important to consider. If the chunk size does not divide evenly amongst the GPUs, some GPUs will be sitting idle while others are working, leaving room for additional performance gains.

Figure 8.2 shows the performance effects of changing the chunk size. These results were collected using three GPUs, which is why the chunk size scales in multiples of three. As the results show, selecting a chunk size of 6 is the best for the test system. It is also important to note that the next the best chunk size is 3 as opposed to 9. This is due to the amount of each chunk that needs to be discarded, which should be minimized if possible. In general, it is best to set the chunk size so that the GPUs can compute their workloads and return

their results in roughly the same time it takes for the CPU to get through the bottom loop of the algorithm. Based on these results, the test system is capable of computing whether or not two observations should be included in the model per GPU in this amount of time.

8.1.2 Tuning the Number of GPUs

The number of GPUs to be used when running GenSel is mostly limited by the system itself. GPUs are specialized hardware, and many systems do not contain more than one. For this reason, the optimizations made are very flexible and allow the user to easily configure the number of GPUs, as well as, which GPUs to use.

Figure 8.3 shows the effect of running GenSel with multiple GPUs. All runs were performed on the same system, the number of markers was set to 500, and the chunk size was set to 6 for all runs except the four GPU run, where the chunk size was set to 8. The single GPU run uses the Tesla K40, the two and three GPU runs add one and two Tesla K20x cards respectively, and the four GPU runs use all of the system's GPUs, including the GTX TITAN Black. The speedups improve with the addition of each GPU, which is expected as the same amount of work can be compute faster with multiple GPUs. Additionally, the lowered crossover point is also nice as it allows for speedups at a much lower number of observations.

As can be seen in Figure 8.3, there is a large gap between the the speedups of the 2 and 3 GPU runs. To further investigate this issue, the order of the GPUs was changed. In the first run, the order was Tesla K40, Tesla K20x, Tesla K20x, and TITAN Black. The ordering of the GPUs was changed to Tesla K20x,

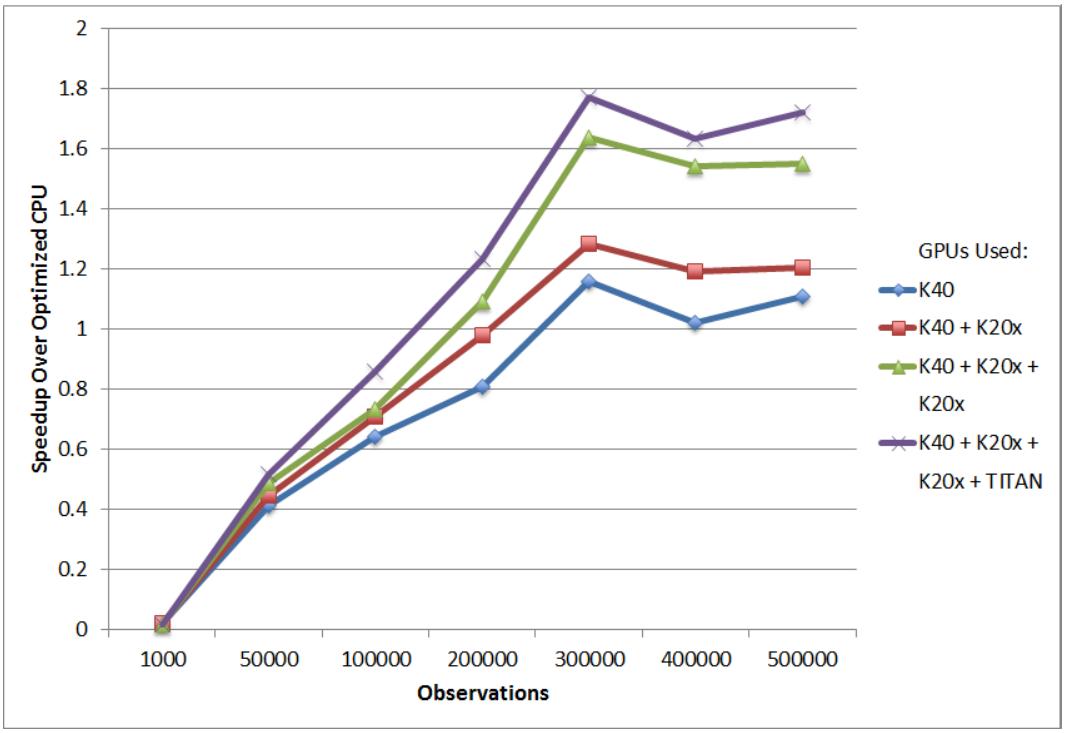

Figure 8.3: Effect of scaling the number of GPUs utilized on speedup

Tesla K20x, Tesla K40, and TITAN Black. The results of this run can be seen in Figure 8.4. The reordering of GPUs has little effect on the speedups achieved by 2, 3, and 4 GPUs, but has a negative impact on the speedups of the single GPU runs. This is most likely caused by the K20x being a weaker card than the newer K40. These results show that the culprit of the speedup gap is the workload each GPU has in a given chunk computation. In the single GPU runs, the GPU is computing 6 dot products for each chunk, which is causing the CPU to wait for the GPU when it needs the dot product results. On the 2 and 3 GPU runs, each GPU is computing 3 and 2 dot products respectively. This drastically reduces the amount of time the CPU spends waiting for the dot product results from the GPU. On the 4 GPU run, the chunk size is set to 8, allowing for even spreading of the workload, which also has each GPU computing 2 dot products for each chunk. This means that the CPU is effectively waiting the same amount

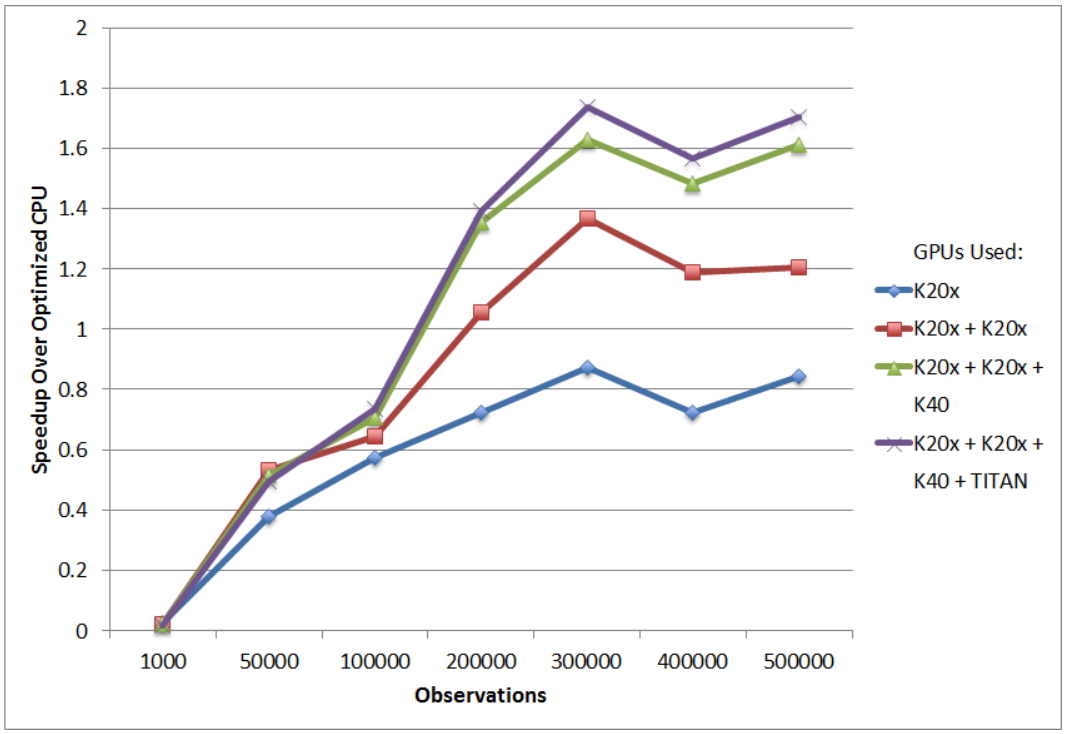

Figure 8.4: Effect of scaling the number of GPUs utilized on speedup with re-ordered GPUs

of time for each chunk of dot products to be computed in the 3 and 4 GPU runs, as each GPU is computing two per chunk. By having a chunk size of 8, the 4 GPU implementation is capable of consuming more markers per chunk than the 3 GPU implementation, which gives it a greater speedup.

8.1.3 Tuning the buffer size on each GPU

The buffer size that is used on each GPU can have a large effect on the algorithm's performance. If this buffer is not large enough, the data transfers between the CPU and GPU will not be hidden, and will cause the GPUs to idle while the memory transfers complete. The buffer size also cannot be set too high. The reason that the buffer is needed in the first place is the limited memory available

on each GPU. The ideal setting for the buffer size is just large enough to mask the data transfers from the CPU to the GPU. This will largely be dependent on the system in use. While setting the buffer size too high will not have a major negative impact on performance, it will cause a slight performance loss as the initial data transfer will be larger for each iteration in the MCMC chain.

Figure 8.5 shows the speedups obtained by using different buffer sizes, specifically two, three, four, and five times the chunk size, which in these runs was set to six. These results show that a buffer size multiplier of four is the best, meaning it allows data transfers to complete before the data is needed, but is not so large that an excess amount of data must be transferred at the start of each MCMC iteration. This graph also shows that a buffer size larger than what is needed is worse than one that is smaller. This may be true with a small number of markers, 500 in these data runs, but may not be true in data runs with a larger number of markers. This is because in data runs with smaller marker counts, the excessive initial start up cost is a larger component of the MCMC iteration than the small performance losses that occur when the GPU is waiting on memory transfers. With larger marker counts, the initial transfer time will remain constant, but the net performance loss that occurs for each chunk transfer will increase linearly with the marker count. This goes to show that tweaking the buffer size can have a great effect on performance and needs to be tuned for every system.

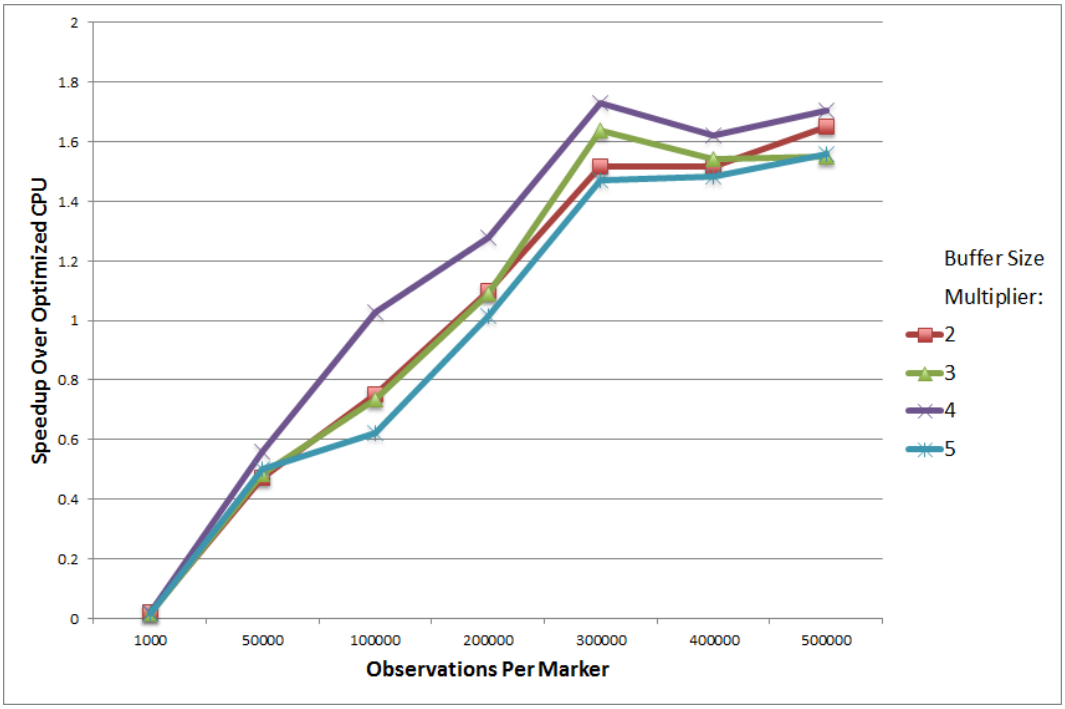

Figure 8.5: Effect of changing the buffer size multiplier

Chapter 9

Future Work

As with all optimized algorithms, there is always more that can be done to further improve performance. The optimizations presented in this thesis show good performance improvements for large datasets, but take a performance hit for smaller datasets. In the future, extending the implementation to automatically decide which algorithm is more appropriate to run for the given input will yield good performance for all input cases. This will also prevent the user from having to manually decide which algorithm to run. In addition, the decision on which algorithm to run can be made more dynamic by analyzing the runtimes for a set of inputs. This will further improve the user's experience and prevent code changes when a different system is used. Along with determining which algorithm to run, some form of automatic tuning can also be implemented. This would set the chunk size and chunk size multiplier to the most optimal settings automatically, further reducing the amount of work done by the user.

Optimizing GenSel for a cluster may also yield performance gains. Many modern clusters contain GPUs on each node, and an approach like this would provide opportunities to leverage fine-grained data level parallelism, coarse-grained data level parallelism, and task-level parallelism. Each of these forms of parallelism could optimize a different aspect of GenSel, ultimately leading to massive performance improvements.

GenSel contains many other opportunities for improvements. The optimizations made in this thesis do not change the underlying statistical analysis being performed. In the future, other statistical algorithms that perform a similar analysis but are easier to parallelize can be used instead of Gibbs sampling. Altering the algorithm used to a block Gibbs sampler may provide an opportunity to parallelize a larger chunk of the algorithm without sacrificing accuracy in the statistical model.

Another potential area for improvement is better load balancing on multi-GPU implementations. Currently, the memory layout and distribution among the GPUs handles the even spread of work across the GPUs. In the system used for testing, this is a completely adequate form of load balancing as the GPUs are very similar in compute power and are able to handle their workloads in very similar amounts of time. In other systems, this may not be ideal. Some systems may have a much larger variance in their hardware, which could include multiple GPUs that are of differing performance. In these cases, the work should be distributed in a manner such that more work is being directed to the more powerful GPUs. While an implementation for this type of system may not be quite as flexible, it will allow the for the highest throughput and lowest latency, which is important in high performance computing.

Finally, the current implementation can be expanded to utilize different devices. Currently, the system is confined to NVIDIA GPUs. Future iterations can provide the ability to run on GPUs from other manufacturers, as well as, other high performance computing solutions, such as Intel's MIC. Memory layouts and other performance optimizations made in this thesis could largely be applied to both of these paradigms. This would further improve the usefulness of these

optimizations by allowing them to be used on whatever hardware a system may have available.

Chapter 10

Conclusion

Identifying areas of MCMC algorithms that contain parallelism and applying the correct programming paradigm can lead to strong performance improvements. In the case of a loop that contains a sporadic dependency, chunking portions of the computation together allows for portions of the loop to be parallelized.

However, it is important to note that careful tuning is required to achieve these performance benefits. More specifically, identifying the correct number of iterations to chunk together is crucial. Without this, either too much data will be discarded, or the loop will execute in an almost sequential fashion. Additionally, creating the correct buffer size on each GPU can make or brake the optimization. It is necessary that the GPU is never waiting for the data it needs to compute, and a properly sized data buffer will prevent this from occurring. In addition, it is also important to note that each system will require a different set of tuning parameters.

Leveraging the computing power of GPUs allows for the optimization of MCMC based algorithms using Gibbs sampling, or other statistical algorithms that contain sporadic loop dependencies. By adding multi-GPU support to GenSel, speedups of up to 1.84 are possible for large datasets.

Bibliography

[1] Benjamin Block, Peter Virnau, and Tobias Preis. Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model. *Computer Physics Communications*, 181(9):1549–1556, 2010.

[2] Long Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[3] Dorian J Garrick, Jeremy F Taylor, Rohan L Fernando, et al. Deregressing estimated breeding values and weighting information for genomic regression analyses. *Genet Sel Evol*, 41(55):44, 2009.

[4] Daniel Gianola and Rohan L Fernando. Bayesian methods in animal breeding theory. *Journal of Animal Science*, 63(1):217–244, 1986.

[5] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[6] David Habier, Rohan L Fernando, Kadir Kizilkaya, and Dorian J Garrick. Extension of the Bayesian alphabet for genomic selection. *BMC bioinformatics*, 12(1):186, 2011.

[7] Anthony Lee, Christopher Yau, Michael B Giles, Arnaud Doucet, and Christopher C Holmes. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. *Journal of Computational and Graphical Statistics*, 19(4):769–789, 2010.

[8] Adrian E Raftery, Steven Lewis, et al. How many iterations in the Gibbs sampler. *Bayesian statistics*, 4(2):763–773, 1992.

[9] NVIDIA Corporation. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, February 2014.

[10] NVIDIA Corporation. CUDA Toolkit Documentation – cuBLAS. `http://docs.nvidia.com/cuda/cublas/`, February 2014.

[11] Derek A Roff and Paul Bentzen. The statistical analysis of mitochondrial DNA polymorphisms: chi 2 and the problem of small samples. *Molecular Biology and Evolution*, 6(5):539–545, 1989.

[12] Marc A Suchard and Andrew Rambaut. Many-core algorithms for statistical phylogenetics. *Bioinformatics*, 25(11):1370–1376, 2009.

[13] Marc A Suchard, Quanli Wang, Cliburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. *Journal of Computational and Graphical Statistics*, 19(2), 2010.

[14] Jeffrey S Vetter, Richard Glassbrook, Jack Dongarra, Karsten Schwan, Bruce Loftis, Stephen McNally, Jeremy Meredith, James Rogers, Philip Roth, Kyle Spafford, et al. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *Computing in Science and Engineering*, 13(5):90–95, 2011.

[15] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, and Kai Lu. Adaptive optimization for petascale heterogeneous CPU/GPU computing. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 19–28. IEEE, 2010.

[16] Weihang Zhu and Yaohang Li. GPU-accelerated Differential Evolutionary Markov Chain Monte Carlo Method for Multi-objective Optimization over Continuous Space. In *Proceedings of the 2Nd Workshop on Bio-inspired Algorithms for Distributed Systems*, BADS '10, pages 1–8, New York, NY, USA, 2010. ACM.