REGEN: OPTIMIZING GENETIC SELECTION ALGORITHMS FOR HETEROGENEOUS COMPUTING

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

Scott Winkleblack

June 2014

COMMITTEE MEMBERSHIP

TITLE:	ReGen: Optimizing Genetic Selection Algorithms for Heterogeneous Computing
AUTHOR:	Scott Winkleblack
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 Dairy Science
COMMITTEE MEMBER:	Associate Professor John Seng, Ph.D., Department of Computer Science

ABSTRACT

ReGen: Optimizing Genetic Selection Algorithms for Heterogeneous Computing

Scott Winkleblack

GenSel is a genetic selection analysis tool used to determine which genetic markers are informational for a given trait. Performing genetic selection related analyses is a time consuming and computationally expensive task. Due to an expected increase in the number of genotyped individuals, analysis times will increase dramatically. Therefore, optimization efforts must be made to keep analysis times reasonable.

This thesis focuses on optimizing one of GenSel's underlying algorithms for heterogeneous computing. The resulting algorithm exposes task-level parallelism and data-level parallelism present but inaccessible in the original algorithm. The heterogeneous computing solution, ReGen, outperforms the optimized CPU implementation achieving a 1.84 times speedup.

Heterogeneous computing, Bayes methods, Bioinformatics

ACKNOWLEDGMENTS

I would like to especially thank my parents and family for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

GenSel is a genetic selection analysis tool used to determine which genetic markers are informational for a given trait. It can also be used to determine the genetic estimated breeding value of a particular individual. To determine this information, Bayesian methods are applied to genotypic (genetic markers) and phenotypic (observable traits) information collected from a set of individuals. For example, this tool could be used in breeding turkeys with larger breasts.

Performing this form of analysis on a large group of individuals is very computationally expensive and, as a consequence, time consuming. Historically, this process has been performed by a single CPU thread. This presents a substantial opportunity for performance improvement by realizing untapped potential parallelism. This thesis focuses on reengineering the existing algorithm to take advantage of the many opportunities for parallelism that exist within the heterogeneous computing environment.

Recently, the scientific community has moved to embrace heterogeneous computing as the technological base for high performance computing. Heterogeneous computing is the use of multiple types of processors in a single system, typically CPUs and GPUs. The different strengths of the various processors allow for differing levels of data and task level parallelism to be more readily exploited. In

addition, they are an economical way of providing a large amount of computing power and scale easily.

This thesis presents a heterogeneous computing version, named ReGen, of the genetic selection analysis tool, GenSel. ReGen attempts to efficiently exploit both task-level parallelism (TLP) and data-level parallelism (DLP) for performance benefits. Using a combination of CPU threads, GPU threads, and parallel programming best practices a speedup of up to 1.84 times over the original, highly optimized CPU based implementation is achieved.

Chapter 2 presents the motivation behind this thesis. Chapter 3 presents the necessary background information as well as introducing the algorithm of interest. Related works are detailed in Chapter 4. The methodology behind the optimizations performed is outlined in Chapter 5. The algorithm is discussed at length in Chapter 6. Implementation details are presented in Chapter 7. The experimental setup and results are discussed in Chapters 8 and 9, respectively. Finally, Chapters 10 and 11 present conclusions and future work.

CHAPTER 2

Motivation

Runtimes for GenSel are strongly dependent on the number of markers and the number of observations per marker. Currently, a typical analysis involves hundreds of thousands of observations for tens of thousands of genetic markers. For the foreseeable future, the number of genetic markers is likely to stay the same. However, recent technological advances are driving down the cost of genotyping individuals. This is expected to cause the number of observations to climb as high as 1,000,000 per marker for a single analysis [10].

While the increase in observations will likely lead to better knowledge of what genetic markers are informational it will come at the cost of time. As it stands, GenSel is simply not equipped to perform an analysis at that scale; runtimes would be measured in days, potentially weeks! It is therefore imperative that GenSel be optimized to accommodate the expected load.

Optimizing this tool is not as straightforward as it would seem. The statistical basis from which GenSel makes its predictions is Bayesian inference, implemented as Gibbs sampler. Gibbs sampling is a Markov chain Monte Carlo (MCMC) algorithm that is often employed in the field of bioinformatics. MCMC based algorithms are notoriously difficult to optimize, especially using parallelization as an optimization technique. However, recent successes have shown that by applying parallelization principles within the context of heterogeneous computing

environments it is possible to achieve speedups in excess of 10 times in certain cases [13, 21, 22].

Taking inspiration from the mentioned success stories, this thesis represents the first step in incorporating heterogeneous computing ideas into GenSel. The major contribution of this thesis is the redesign of the algorithm used by GenSel. The resulting algorithm exposes TLP and DLP present but inaccessible in the original algorithm. Using this algorithm as a foundation, ReGen, a heterogeneous compute solution for optimized genetic selection analysis was created.

CHAPTER 3

Background

3.1 Bayesian Inference

Bayesian inference is a form of statistical inference. In this form of inference, unknowns are expressed as probabilities, which are continuously refined as new evidence emerges. Bayes' rule provides the framework by which new evidence is incorporated. Bayes' rule is expressed in Equation 3.1 where | stands for conditional probability, H for hypothesis, and E for evidence.

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)} \tag{3.1}$$

In this equation, the probability of the original hypothesis, $P(H)$, is known as the prior distribution. A likelihood function, $P(E|H)$, indicates if the evidence is coherent with the current belief. This information is combined resulting in the probability of the hypothesis given the evidence, expressed as $P(H|E)$. This probability is known as the posterior.

There are several important ideas that come out of Equation 3.1. First, beliefs are updated not only on the probability of the evidence but also on the probability of the belief itself. Secondly, Bayes rule can easily be applied iteratively. To do so, the posterior derived after observing a piece of evidence is simply treated as

the prior for the piece of evidence that follows. These important properties allow estimated parameters to be continuously refined.

Bayesian inference is a powerful tool that has a wide number of applications in fields ranging from law to computer science. In computer science, Bayesian inference is used in artificial intelligence [4, 11] and spam filtering [5]. It has also increasingly appealed to and been utilized by the bioinformatics community, specifically in the area of phylogenetics [21].

3.2 Gibbs Sampling

Directly implementing Bayesian inference is problematic. More specifically, the problem comes down to computing the posterior. The problem is twofold. First, there is not guaranteed to be a closed form solution for computing the posterior. Secondly, in the cases where a closed form solution does exist, it may be computationally infeasible requiring the integration of high-dimensional functions.

Gibbs samplers offer a solution to this problem for certain classes of Bayesian problems. By repeatedly drawing sampling from the posterior and employing a technique called Monte Carlo integration, an approximate posterior distribution can be created. In essence, Monte Carlo integration works by simply averaging the randomly selected points [6, 23]. This avoids the problems with exponential growth in complexity associated with integrating over high-dimensions. Gibbs samplers provide a practical, feasible method for generating the posterior.

3.3 Markov Chain Monte Carlo

Gibbs samplers belong to a class of algorithms referred to as Markov chain Monte Carlo (MCMC) methods. MCMC methods get their name from the two algorithms that comprise them, Monte Carlo methods and Markov chains. Monte Carlo methods rely on random sampling to estimate numeric results, as mentioned in Section 3.2.

A Markov chain defines a set of states and the probability of transitioning between states. This process is memoryless as the transitional probabilities are only a function of the current state. Repeated transitions define the chain. Markov chains are good for modeling nonrandom events. For example, how traits relate to genetic markers.

Taken together, MCMC methods allow a Markov chain to be constructed whose equilibrium represents the desired integral needed for Bayesian inference. To arrive at the correct distribution, enough iterations of the chain must pass for the probability distribution to converge. This task proves to be non-trivial [15]. Also, adding to the problem, the beginning of the chain is loosely correlated with the starting position. This presents a problem as this algorithm relies on randomness for success. Discarding the first portion of the chain, a technique known as burn in, is used to mitigate this problem. Due to the amount of iterations needed for burn in and to achieve convergence MCMC algorithms are very compute intensive.

Besides being compute intensive, MCMC based algorithms are notoriously difficult to optimize, especially by leveraging parallelism. This stems from the serial, undeterministic nature of Markov chains. Despite the challenges, there have been some significant successes in optimizing this class of algorithm [22].

3.4 Heterogeneous Computing

Heterogeneous computing refers to a computing platform composed of multiple types of processors, typically CPUs and GPUs. The idea being each type of processor is designed with a particular type of task in mind. By making different types of processors available and using them as intended it is possible to reap the benefits of improved performance and power consumption. In addition, using GPUs as accelerators has proven to be an economical way of obtaining a large amount of computing power [7, 19].

3.4.1 CPU Based Computing

CPUs are optimized to minimize latency and offer high clock speeds. This makes them very efficient at handling sequential tasks. Their ability to quickly switch contexts makes them very good at multitasking. However, clock rate increases have stalled in recent years leading to, among other things, more parallelism concepts being incorporated in CPUs [1].

The availability of more threads, multiple cores, and even multiple CPUs on one system are some of the ways CPU parallelism is manifesting itself. CPU threads operate independently from one another, unlike GPU threads. This makes it very easy to exploit TLP.

Parallelism constructs are also helping CPUs handle DLP more efficiently. Modern CPU instruction sets often support vectorized instructions. These instructions are often referred to as single instruction multiple data (SIMD) instructions. These instructions make the CPU more effective at computing math intensive tasks, such as multimedia.

3.4.2 GPU Based Computing

GPUs were originally designed for rendering 3D computer graphics. As such, GPUs are optimized for throughput not latency. They feature lower clock speeds than CPUs but make up for it with massive amounts of parallelism. GPUs provide thousands of cores that allow them to exploit large amounts of DLP.

GPUs' unique architecture allows them to outperform CPUs when performing certain types of mathematical calculations on very large data sets. Additionally, GPUs can deliver these capabilities at much lower power consumption relative to CPUs. Leveraging GPUs makes computational power not possible in multi-core and unfeasible in distributed computing solutions not only possible but also accessible.

Building applications around GPU computing paradigms requires a very different mindset than CPU computing. General purpose GPU (GPGPU) computing follows the single program, multiple data (SPMD) model. Developing the necessary skills to unlock the power of the GPU can be time consuming but advances in tools such as CUDA, are lowering this barrier.

3.5 CUDA

Compute Unified Device Architecture (CUDA) is the parallel computing platform developed by NVIDIA [16]. It allows developers to easily perform GPGPU processing on NVIDIA GPUs. The CUDA toolkit extends the C, C++, and Fortran languages for ease of use.

CUDA provides powerful ways to interact with GPU threads by abstracting thread organization. Threads are organized into blocks composed of up to three

dimensions. Blocks in turn make up the grid, which may be organized into up to two dimensions.

Threads are organized into groups called warps by the GPU. When executing a kernel (GPU function call) every thread in a warp executes on a single streaming multiprocessor (SM). SMs are capable of executing multiple warps at once. Threads in a warp execute instructions in lock step. Thread divergence within an SM is undesirable as it causes very poor performance in CUDA. When divergence occurs, all threads in the warp must execute all relevant branches. Thus, programmers must take extreme care to organize threads in such a way as to achieve warp level execution path agreement [16].

There are three types of memory available to the programmer for general purpose computing: global memory, shared memory, and registers. Global memory is by far the most abundant but also the slowest. Shared memory and registers can both be accessed in one clock cycle. All threads in a block can access the same shared memory. Registers are the least available form of memory and are associated with SMs. The GPU is composed of many SMs.

In CUDA, there is a concept of a stream. Streams are a way of organizing kernel calls in a way that indicates call dependency and allows for the exploitation of concurrent execution of kernels. When called, every kernel is associated with a stream either explicitly by the caller or implicitly by CUDA. Each stream essentially acts like an independent queue. Within a stream kernels are guaranteed to execute in the order they were called. However, there is no guarantee of ordering between streams. The GPU will run kernels from different streams as concurrently as possible. As SMs become available, the next kernel pending immediately fills them as long it does not violate ordering guarantees. This helps achieve higher occupancy of the GPU and leads to a pipelining effect.

Equally as important as allowing concurrent execution of kernels, streams allow transfer to and from the GPU to occur simultaneously, in devices of compute capability 2.x and higher with two copy engines. If memory that cannot be swapped out, known as page-locked or pinned memory, is used the transfer can even occur asynchronously. Using this technique, data can be transferred to the GPU during preparatory phases on the CPU, results can be transferred back to the CPU so they are ready when needed, and data that is no longer needed on the GPU can be replaced at any time, even while execution is occurring. This has very powerful performance implications. It can significantly reduce or eliminate the amount of time the GPU or CPU is waiting for data. Thus, both the CPU and GPU can always be performing work reducing idle time.

Data transfer across the bus is the Achilles heel of GPGPU computing. GPGPU computing is only useful if the performance gain is not dwarfed by the cost of moving the data to and from the GPU. By intelligently designing algorithms around heterogeneous computing principles it is possible to lessen the impact of transfer times.

CHAPTER 4

Related Works

As MCMC based algorithms have become more prevalent in many fields they have drawn a lot of attention due to their computational intensity and serial nature. This has led to many attempts at optimizations [2, 13, 14, 21, 22]. These attempts can be broadly classified under two optimization strategies. One attempts to increase performance by breaking the chain into many smaller pieces and running them concurrently; the other seeks to parallelize a single MCMC run by shrinking the links of the chain [3].

In the first strategy, the chain is essentially split into smaller pieces. Each chain is then run independently. Determining the point of convergence can be more difficult with this method. Nonetheless, convergence can be determined by comparing between and within sequence variance [2]. The performance boost from this method is due to the smaller chain lengths and because each chain is independent, they can be computed concurrently. There are some drawbacks associated with this approach. The burn in period, which must be discarded, is repeated across multiple processes. Additionally, to increase the randomness of the process a practice known as thinning is often used. When thinning, after an iteration is kept a set number of iterations immediately following that iteration are discarded [8]. Some implementations have managed to achieve near linear speed ups using this approach [18].

The second strategy does not attempt to break up the chain. Instead it recognizes that by shrinking the individual links the overall length of the chain can be reduced. This strategy can be more finely broken down into two sub-strategies associated with exploiting TLP and DLP within the chain.

The more successful strategy exploits DLP by utilizing GPU technology to parallelize general sampling methods [13] or the evaluation of likelihoods [21]. The results reported in research performing this type of optimization saw speed ups of 10's to 100's of times [13, 21, 22].

In general, exploiting TLP turns out to be less successful than DLP for MCMC based algorithms. This is largely due to the fact that there is a dependence between each chain link which incurs major communication overhead due to the need for global model parameter updates, if a distributed approach is taken [22].

The algorithm employed by the heterogeneous solution presented in this thesis, ReGen, follows the second strategy. It combines both of the sub-strategies, exploiting both TLP and DLP. By correctly identifying regions of code that exhibit parallelism and associating these regions with the correct parallel programming paradigm based on system architecture considerations. This idea is set forth in [8] which mainly focuses more efficient matrix operations on the CPU. The work presented in this thesis expands the spectrum of parallelizable code and utilizes different technologies. Furthermore, it presents a method for breaking sporadic loop carried dependencies that allows for still more parallelism to be exploited.

Leveraging GPU technology as a complete solution, as in [13] or [22], proves to be problematic because of an additional data dependence between markers within each chain link. The same issue that causes the additional data dependency

leads to problems with thread divergence. Divergent algorithms do not map well into the GPGPU computing paradigm. Pre-computing the data responsible for control flow has the potential to solve the problems with divergence. But, pre-computing this data substantially reduces the amount of parallelizable code to the point that communication overhead between the host and device overshadows any performance improvement. Instead, use of GPU technology is limited to performing large matrix operations, which still provides a sizable speedup for very large matrices.

CHAPTER 5

Methodology

To achieve better performance, ReGen embraces parallelism in all its forms. Choosing parallelism as an optimization strategy is ideal when changing the fundamental process of an algorithm is undesirable. It relies on identifying independent tasks and duplicated processes that can be performed concurrently. After identifying areas of parallelism it may be necessary to reorganize and restructure the original algorithm to expose them and make them accessible. It is critically important to the success of such optimizations to correctly map parallelizable regions to the correct parallel programming practice.

All parallelizable regions of code do not exhibit the same levels of parallelism. Parallelism exists on a spectrum. Very coarse-grained parallelism can easily be exploited by paradigms such as MPI, which allows for work to be easily spread across a cluster. Tasks that fit well into this form of parallelism generally exhibit large amounts of TLP and have a large compute to communication ratio. Slightly finer grained parallelism is more suited to local parallelism constructs as opposed to distributed parallelism principles. Local CPU threads are more suited to tasks that exhibit large amounts of TLP due to the design of the CPU. However, because many threads can reside on the same device communication overhead is much less of an issue than in a distributed system. Local parallelism can also take the form of multiple GPUs, these devices may collaborate and perform peer to peer communication. Even finer levels of parallelism can exist within threads in

the form of vector constructs. Vector constructs allow threads to take advantage of DLP.

Due to the large number of data dependencies present in the MCMC based algorithm used by GenSel very coarse grained parallelism, such as distributed computing, offers no help in terms of performance. The massive communication cost rapidly exceeds the benefits of distributing computation. The rest of the parallelism spectrum is present in GenSel in one form or another waiting to be capitalized on.

The heterogeneous computing environment provides the needed tools to exploit the spectrum of parallelism present, due to the different types of processors. In general, the CPU is the best choice for highly divergent code or when compute time is low enough that bus transfer times prevent the GPU from being effective. The GPU is the most effective choice for highly parallel tasks, such as math operations, that can be organized so that threads within warps rarely follow divergent branches of code or when dealing with very large sets of data. For smaller sets of data, vectorized instructions allow the CPU to fill the gap between traditional CPU and GPU computing.

It is necessary when designing a high performance computing solution to know and understand the limitations of the system the process will be run on. Engineering decisions may not hold across systems; they depend on the particular CPU, GPU, and type of interconnects involved. Unfortunately, this leads to custom tailored solutions and the loss of some flexibility seen in deploy anywhere systems. The details of the environment ReGen was optimized for can be found in Chapter 8.

CHAPTER 6

Algorithm

6.1 Overview

GenSel is used to infer the effects of genetic markers on a desired trait or to determine the genomic estimated breeding value (GEBV) of genotyped individuals. As previously mentioned, GenSel employs Bayesian inference to make these predictions. In this application, the contribution of every genetic marker to the desired trait can be seen as a variable. A statistical model represents the mean effect of all markers on the desired trait. Another distribution captures an estimate of the error associated with these predictions.

Genotypic and phenotypic information is provided by the user and used to construct the original models. Afterwards, during each iteration of the chain, each marker is evaluated to determine if it shall be included or excluded in the current model. A very high level overview of the algorithm is presented in Figure 6.1.

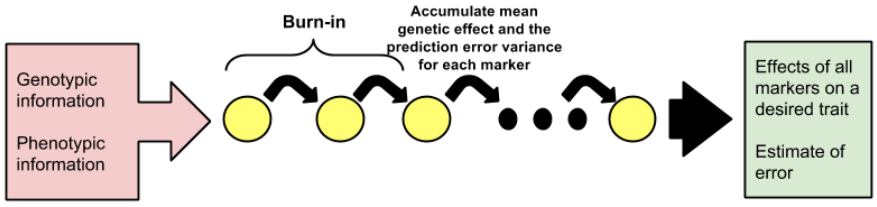

Figure 6.1: Overview of algorithm

6.2 Identifying Regions of Parallelism

In GenSel, the same process is repeated every iteration of the chain. The statistical model modified in iteration i is then used in iteration $i + 1$. This data dependency makes optimizing GenSel technically challenging. In fact, this is the same issue experienced by all MCMC algorithms. Luckily, optimizing GenSel does not depend on breaking this data dependence.

The search for regions of parallelism continues within each iteration of the chain. Algorithm 1 presents a high level overview of the original algorithm that is executed within each chain link. It is here that many opportunities can be found for exploiting parallelism.

Algorithm 1 Initial Bayes Algorithm

```
1: for each marker do
2:    compute probability marker is included in model
3:    if P_model(marker) < threshhold 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
```

The pseudo code does not make it apparent but there is an additional data dependence that occurs within each chain link. The BayesC algorithm presented in [12], which forms the basis of this thesis, contains another data dependence that occurs between markers when building the model in each chain link. A very

important aspect of this data dependence is that it is conditional. The data dependence is only present between adjacent markers when the first marker is chosen to be included in the current model or is excluded but was included in the model of the previous iteration. Because the data dependence only occurs some of the time it is possible to work around but it does limit the amount of parallelism possible.

Computing the probability a marker is included in the model is a based on linear algebra that the GPU excels at. Moreover, the size of the matrices involved scales with the number of observations in the analysis. This makes for a prime target for GPGPU acceleration, especially when the matrices become very large. The challenge becomes hiding the memory transfers to and from the GPU. For more details on what defines the threshold and how the probability of a marker being included is determined refer to [12] and [20].

There are several small-scale matrix operations present within this algorithm as well. For these operations any communication overhead proves to be prohibitively expensive and taking advantage of SIMD instructions on the CPU is the best choice.

6.3 Refactoring the Original Algorithm

Algorithm 2 presents the restructured algorithm that has been refactored to expose additional parallelism, some of which is mentioned above. This algorithm forms the basis of the heterogeneous computing solution for genetic selection related analyses, ReGen. In Algorithm 2, the **for** loops on Lines 2 and 10 are performed concurrently using CPU threads, Lines 3, 12-14, and 16 leverage SIMD instructions, and Lines 3, 8-9, and 19 involve steps on or using the GPU.

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 P_model(marker) < threshhold 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
```

In the first **for** loop, everything that is needed to determine which markers from the chunk are going to be included in the model is computed. The first marker that triggers the loop carried dependence is marked as the terminating marker of the chunk and the computed values of all following markers are discarded; they are no longer valid because they are based on the old model as opposed to the updated model. In the best case scenario, the entire chunk is valid and no computations are wasted, and in the worst case scenario, only one marker in the chunk is valid and the rest must be discarded. The optimal chunk size for a given system is determined partially on the statistics behind GenSel and partially on the system itself. The number of GPUs available is a large contributing factor in the latter. Properly selecting a chunk size results in upwards of 85% of the computations being valid. This keeps runtimes much closer to the best case than the worst case by keeping wasted computation down.

The loop carried dependency was determined empirically to only occur on average every 10 iterations. This means those 10 iterations exhibit TLP, which can be exploited by threads to obtain a performance advantage. To realize the performance advantage, speculative execution of c iterations of the **for** loop are performed concurrently. The results are validated to see if the data dependence occurred in any of the loops before the results are incorporated into the statistical model. The speculative execution performed adds very little overhead because the computations are executed concurrently. The value c is referred to as the chunk size and can be set by the user. The chunking method allows TLP to be exploited despite the presence of a sporadic data dependence. Chunking effectively reduces the amount of iterations required to build the model within each chain link. This has a big performance impact. The overall speedup offered by the heterogeneous

solution is a product of the per marker speedup and the average number of markers that are parallelized by this method.

Next, the bottom **for** loop is executed for every marker in the chunk up to and including the terminating marker. This is responsible for updating the current model. There is no wasted computation associated with the second **for** loop. The GPU is not leveraged for this portion of the algorithm. The inability to predict which marker will be the terminal marker and transfer times associated with moving the necessary data to the GPU makes the CPU more suited for the tasks. To increase efficiency for the matrix operations involved SIMD instructions are leveraged via the Eigen library [9].

Finally, the chunk is adjusted to the marker immediately following the terminating marker. The process is repeated until all the markers have been processed.

The algorithm is structured to leverage the heterogeneous computing environment to the fullest. The structure is crafted around one simple idea: never be idle. Eliminating idle time means keeping all processors working full time. This is a very powerful idea that is easy to say and much more difficult to achieve. It requires going beyond common tactics such as taking advantage of TLP via CPU threads and DLP via GPU threads. Often times one processor will get stuck waiting for another processor's result. All data dependencies need to be removed, if possible, or structured in a way that they minimize waiting. Eliminating all data dependencies from this algorithm is not possible. However, by starting computation as early as possible wait times can be minimized. Sections 7.2, 7.3, and 7.5 describe optimizations aimed at reducing wait time. Everything should be as overlapped as much as possible including data transfers and computation. Section 7.4 describes this process in detail. This algorithm also overlaps GPU computation by taking advantage of TLP on the GPU. One way this occurs

is by pipelining kernel calls, discussed in Section 7.6. Additionally, ReGen offers multi-GPU support, covered briefly in Section 7.7.

6.4 Complexity Analysis

The computational complexity for Algorithm 2 is technically worse from that of Algorithm 1. For all computational complexity presented in this paper, n represents the input size in markers and c represents the chunk size, which is a constant defined by the user. The **for** loop in Algorithm 1 executes for each marker and is therefore $O(n)$ algorithm.

The analysis for Algorithm 2 is slightly more complex. The worst case occurs when every marker is included in the model for every iteration. In this case, the first marker in the chunk is the terminating marker but computation is performed on all other markers. This leads to a lot of wasted computation in the upper **for** loop. The second **for** loop only executes one iteration in this case. The amount of wasted computation is determined by c which can range from 1 to n. This leads to a worst case computational complexity shown in Equation 6.1. In practice, $c << n$ (i.e. 6 vs. 50,000) causing performance to to be closer to the original $O(n)$.

$$O(n(c+1)) \rightarrow O(n^2) \tag{6.1}$$

In the best case, the terminating marker in every chunk is the last marker. This occurs when no marker in the chunk causes the data dependence or the final marker causes the data dependence. In this case, the upper and lower **for** loops

execute c times and enclosing while loop is executed $\lceil n/c \rceil$ times. This leads to a best case computational complexity shown in Equation 6.2.

$$O(\left\lceil \frac{n}{c} \right\rceil (c + c)) \rightarrow O(n) \tag{6.2}$$

The slight degradation in terms of computational complexity between Algorithm 1 and Algorithm 2 indicates that any performance gains observed are a result of the application of parallel paradigms and not algorithmic change.

CHAPTER 7

Implementation

7.1 Multi-Threading

Using the refactored algorithm presented in Algorithm 2 it is easy to take advantage of task level parallelism. Each iteration of the two **for** loops is associated with a separate marker and can be viewed as a separate task. The tasks in the upper **for** loop can only be considered to be independent when the conditions discussed in Section 6.2 are satisfied. This means that a group of tasks may be executed concurrently but they must be checked for correctness and results from any markers after a data dependence issue occurs must be discarded. This creates a chunking effect as the total number of tasks remaining is decreased by all or some portion of the original group of tasks.

The size of the chunk ran concurrently is a parameter of the system. It is chosen based on hardware constraints and easily determined experimentally, see Section 9.1 for more details. After the size of the chunk is known threading concepts are easy to apply. One strategy would be to assign one task per thread. This works well if the tasks assigned to the CPU and the GPU have approximately the same runtime. This is not the case for larger data sets, which are the target area of this thesis. In the upper **for** loop, the amount of GPU compute time is proportional to the number of observations while the CPU compute time stays relatively constant. This is true even with all the optimizations discussed above.

As a result, in practice, one CPU thread per GPU is capable of managing several tasks.

7.2 Dot Product Computation

The statistics involved in performing a genetic analysis rely heavily on linear algebra. For example, the dot product computation performed to determine if a marker should be included or excluded from the current model. This is computed every iteration of the chain, for each marker. The size of the computation grows proportionally with the number of observations. Together, these factors make this operation one of the most time consuming portions of the algorithm. Luckily, it is also one of the easiest and most effectively parallelized portions of the algorithm.

Dot products are very math intensive operations composed of many small, identical, independent operations. GPUs excel at this sort of problem. Their large number of cores allows the GPU to take advantage of DLP within the problem and compute the smaller operations concurrently.

The cuBLAS linear algebra library provides an optimized set of linear algebra operations [17]. The downside of GPU computing is that data must be explicitly transferred to and from the device. This adds additional overhead costs creating a threshold that must be met before GPU computing is advantageous over CPU computing. For this application it was generally observed that the GPU was superior at performing this calculation when the number of observations exceeded 150,000.

7.3 Pre-computing Dot Products

Except when bound by data dependence issues, the GPU and the CPU operate independently from one another. This offers another opportunity to exploit parallelism, by overlapping computation on the GPU and CPU. By doing so, the idle time of the processors is decreased and the overall performance increases. If possible, work should be done ahead of time, before it's needed. By working ahead, wait times due to data dependencies can be decreased or eliminated.

An example of this method in action occurs after the first **for** loop in Algorithm 2. At this point in the algorithm, all the information necessary for computing the next chunks worth of data is known. By beginning computation on line 8, the GPU has a head start calculating the results the CPU will need at the beginning of the next iteration of the algorithm. The sets of observations needed to carry out this calculation are made possible by the optimization discussed in Section 7.5.

7.4 Hiding Memory Transfers

Decreasing the amount of time spent waiting on data transfers to and from GPU helps to amortize the cost of GPU computing. In ideal cases, this cost can be completely removed. The most basic technique is to simply use asynchronous memory transfers between the CPU and GPU. Asynchronous memory transfers are a fire-and-forget form of memory transfer. Unlike ordinary data transfers, once kicked off control returns immediately to the caller, instead of waiting for the transfer to complete. Both the CPU and GPU can perform work while the transfer is happening. If done in separate streams and the GPU has two copy

engines, data transfer to and from the GPU can occur simultaneously. The catch is the data transfer can only happen to and from page locked memory.

To hide memory transfers as much as possible data flow patterns that exist within the process must be analyzed. Ideally, all the data required by the CPU or GPU for its current calculation would be available without delay. By beginning transfers as soon as data becomes available, instead of immediately before it is needed, idle times can be minimized.

In this application, the set of observations needed on the GPU is constantly changing. But, as soon as the first **for** loop executes in Algorithm 2 how many markers are valid is known. This indicates how many sets of observations must be replaced on the GPU. Splitting the old **for** loop in two allows for the next set of observations to begin transferring before it is needed. All the while, computation is performed on the GPU and CPU. The CPU updates the statistical model to reflect for the current chunk and the GPU begins computing the dot products for the next chunk. This is shown in lines 8 and 9 of Algorithm 2.

7.5 Circular Data Buffering

As discussed previously, bus transfer times are one of the biggest killers of GPU computing performance and all data needed by the GPU would be transferred before any calculations began. However, due to memory capacity limitations on GPUs this is not always feasible. Instead a subset of data needed for the next n iterations can be kept on the GPU. If n is large enough it can effectively hide the fact that the necessary data is constantly being transferred to the GPU. By ensuring the data necessary for the next iteration is available, this buffering

technique prevents the GPU from sitting idle and allows for the pre-computing of dot products.

For this application, the maximum number of observation sets that must be replaced is set by the user supplied chunk parameter. Due to the data dependence issues in ReGen the number of observation sets that must be replaced varies. A circular buffer was selected to store the observation sets as it naturally lends itself to the problem.

In this implementation, n is a configurable value. A sufficient value for n can easily be empirically determined. The n value used throughout this thesis is 4.

7.6 Pipelining Kernel Calls

Besides offering large amounts of DLP the GPU also offers a limited amount of TLP. Achieving TLP on the GPU involves executing multiple kernels at once. This can be accomplished by calling kernels in separate streams. Recall that this indicates to the GPU that there is no dependence between the two kernels. As a result, hardware permitting, the kernels can be run concurrently. The limiting factor becomes the number of SMs available on the GPU. This can lead to a kernel pipelining effect and helps reduce the latency of each kernel call.

Taking advantage of TLP on the GPU has the added advantage of making sure the GPU is saturated with work. Higher occupancy, the ratio of active warps to the maximum possible number of active warps, often lead to better performance. Yet, this is not always the case [24]. ReGen takes advantage of this form of parallelism by executing all dot product kernel calls in separate streams.

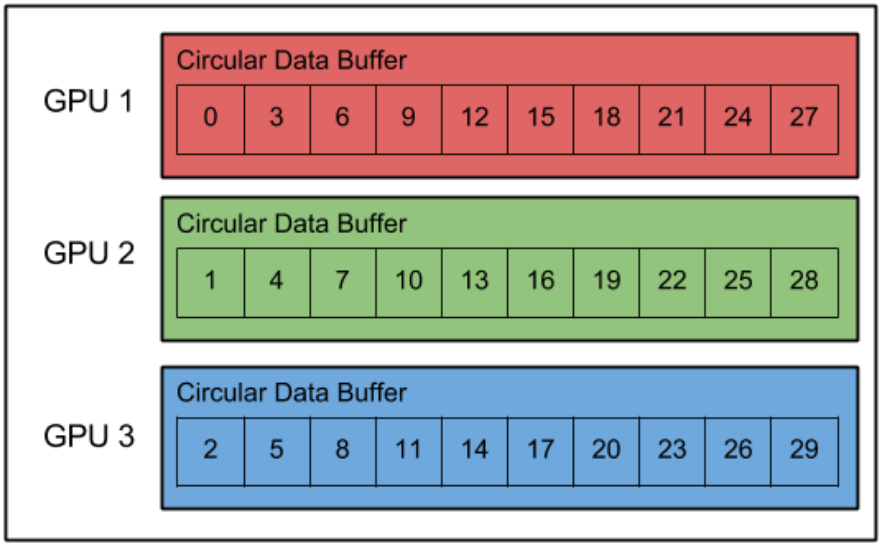

Figure 7.1: GPU memory layout demonstrating striping paradigm

7.7 Multiple GPU Support

Adding more hardware can increase parallelism and performance. Utilizing more GPUs allows the workload to be spread around, reducing the time to compute independent tasks, such as computing a chunk of dot products. Placing the GPUs on different buses can remove bus contention, which can lower performance.

Each GPU is not required to hold all the observation sets. Data is striped across all the GPUs to decrease bus transfer times. To ensure an even workload, the index of the set of observations is divided by the total number of GPUs and the remainder is used to index into the set of GPUs. Figure 7.1 illustrates how observations are divided among the GPUs.

ReGen has been designed to seamlessly scale to accommodate additional GPUs. It can be instructed to use all or a subset of the GPUs available on the system.

CHAPTER 8

Experimental Setup

8.1 Testing Environment

All performance measurements were taken on a single machine. This machine contains two physical CPUs and four GPUs. Only three GPUs were used for testing, leaving one to drive the display. Both of the CPUs are Intel Xeon E5-2650 8-core CPUs. These CPUs have support for hyper-threading and run at 2.00 GHz. The system has 64 GB of RAM available.

The GPUs used for testing consisted of one NVIDIA Tesla K40 and two NVIDIA K20Xs. The Tesla K40 has 2880 cores, 12 GB of memory, and run at a speed of 745 MHz. The Tesla K20Xs have 2688 cores, 6 GB of memory, and run at a speed of 732 MHz.

At the time that tests were conducted, the machine was running Arch Linux containing version 3.14.0 of the Linux kernel. CUDA 6.0 and gcc version 4.8.2 were used to compile and link source code. The NVIDIA driver version was 334.21.

8.2 Test Data

The purpose behind the optimizations presented in this thesis is to enable GenSel to scale and accommodate the dramatic increase in genotypic information that is

expected to occur in the near future. Testing with real data sets is problematic because data sets of the magnitude needed are not currently available. Consequently, in order to conduct relevant tests, simulated genotypic and phenotypic data was used. A tool was created that allowed the creation of data sets containing an arbitrary number of markers and observations. For the tests performed in this thesis the number of markers were limited to 10,000 markers in order to keep runtimes manageable.

8.3 Validation

All test data sets were run on the original algorithm as well as the refactored algorithm. To confirm correctness, the results were compared between the two implementations. Timing data was also recorded during these runs. All timing data reported in the results section includes only compute time, not I/O time. I/O time is common to both the heterogeneous and original solutions and can therefore be removed without distorting the results.

It should be noted that the original algorithm is highly optimized for single thread CPU computing. It leverages highly optimized libraries, such as the Eigen library [9] for math operations. This library makes use of vectorized instructions, loop unrolling, and cache optimizations. In addition, the entire program is compiled using the most aggressive optimization flags.

CHAPTER 9

Results

The results presented in this section compare the results of the heterogeneous computing solution against the optimized CPU based solution. Runtimes and performance comparisons are based off timing data that does not include I/O times. The I/O times are present in both systems and not of interest. The reported timing data does include data transfer times for the heterogeneous solution, both to and from the GPU.

Unless specifically noted, ReGen configuration parameters were consistent across the various tests performed. ReGen was configured to use 3 GPUs. The GPUs had error-correcting code memory (ECC) enabled. Each GPU contained a circular buffer that held 4 chunks worth of data. One chunk of data is defined as the data necessary to process 6 markers.

The effects of individual optimizations were not measured. The heterogeneous computing solution was intended to work as a system. On their own, the optimizations are out of place with the rest of the solution leading to performance decreases. Taken together, the optimizations perform quite well against the highly optimized CPU based implementation.

Figure 9.1 depicts the performance data for ReGen over a wide range of markers and observations. In the interest of time the number of markers was capped at 10,000. Runtimes for larger marker sets can easily be approximated

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 9.1: Runtimes for 500 marker data set

as runtimes scale linearly with marker count. For example, the 10,000 marker case takes about two times as long to run as the 5,000 marker case if all else is held constant. Table 9.1 presents the runtimes for the 500 marker case to provide perspective.

As expected, the CPU based implementation outperforms the heterogeneous solution when the number of observations is small. High clock speeds and limited DLP capabilities, in the form of SIMD instructions, allows the CPU to outperform GPU based computing for the lower end of the observation range.

Closer inspection of Figure 9.1 reveals that the crossover point at which heterogeneous computing becomes beneficial is not constant and is dependent on both the number of markers and observations. The crossover point can be considered to be approximately 150,000 observations for the data presented. At this point, only analyses with a small number of markers ($< 1,000$) are not faster. The runtimes associated with such analyses are significantly smaller than the

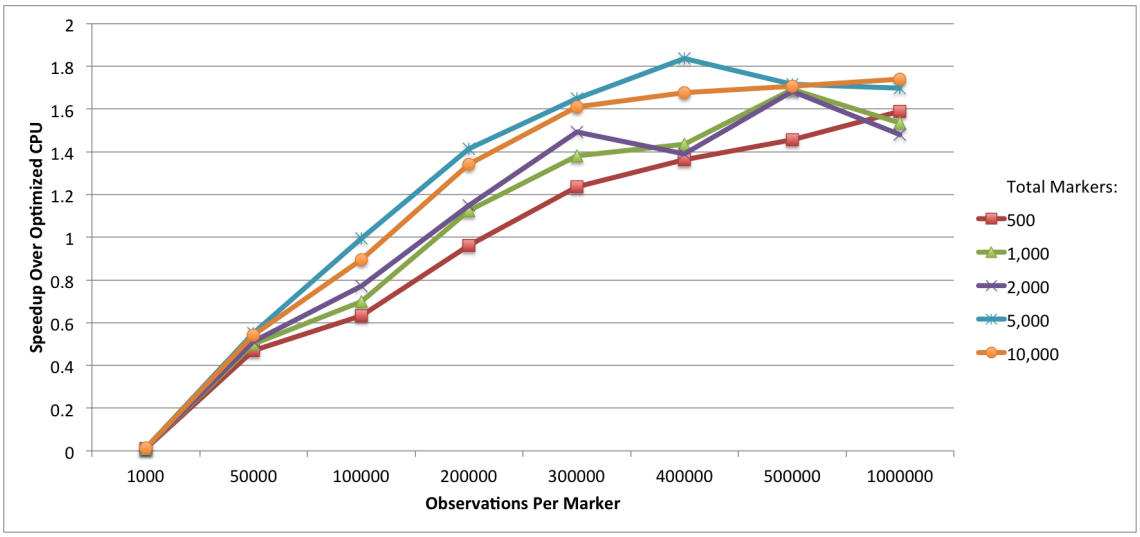

Figure 9.1: Overall speedup with varied number of markers

larger analyses causing small performance differences to be negligible in terms of the overall runtime.

The optimizations presented in this thesis are aimed at assisting in the handling of the increasingly large amount of data GenSel is expected to handle. The heterogeneous solution succeeds in this endeavor. In all cases tested, the heterogeneous solution consistently outperforms the CPU based solution as optimizations grow large. The largest speedup achieved was 1.84.

The best predictor of potential speedup is the number of observations. For a given number of observations, the speedup associated with differing marker totals is largely clustered with larger marker sets tending to see a slightly higher performance improvement. This is promising and indicates that the current solution will scale well to higher marker counts. Observations are much more informative about performance because the success of ReGen is predicated on the increasingly large amount of DLP GPGPU computing can exploit as data sets grow.

This computational advantage is amplified by the chunking method as discussed previously.

9.1 Tuning

ReGen was designed to be highly configurable. This allows the solution to be tailored to the computing environment it finds itself in leading to greater flexibility in deployment. Proper tuning can result in significantly faster runtimes at the cost of increased complexity and time invested in setup. The number of chunks buffered on each GPU and chunk size are the parameters with the most significant effect on performance.

The number of chunks buffered on each GPU is the least interesting parameter. From a performance perspective the GPU should never sit idle and should always have the data it needs for the next set of computations. To ensure this happens the buffer must meet or exceed the minimal size that allows work to always be readily available to the GPU. Falling below the minimum size negatively impacts GPU performance. Exceeding the minimum buffer size can also have a negative performance impact. This is due in a large part to the amount of time spent in setting up the initial buffer exceeding the time necessary to perform other setup routines, delaying the start of computation. Figure 9.2 illustrates the effect of buffer size on performance.

Determining the optimal chunk size can be extremely difficult. Every aspect of the system has an effect from the number of GPUs to the underlying statistics that enable the analysis. For simplicity and accuracy, it is best to determine empirically. This process can be done relatively quickly. The effect of different size chunks on the overall speedup is shown in Figure 9.3.

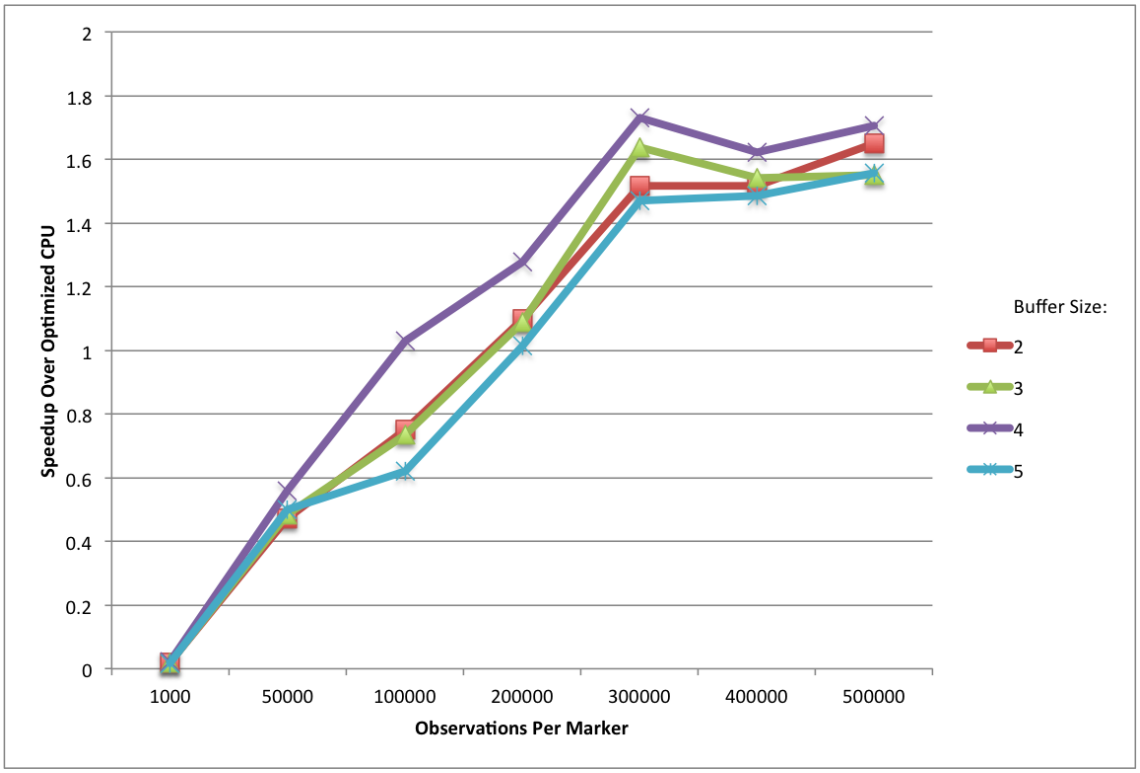

Figure 9.2: Effect of different buffer sizes on speedup

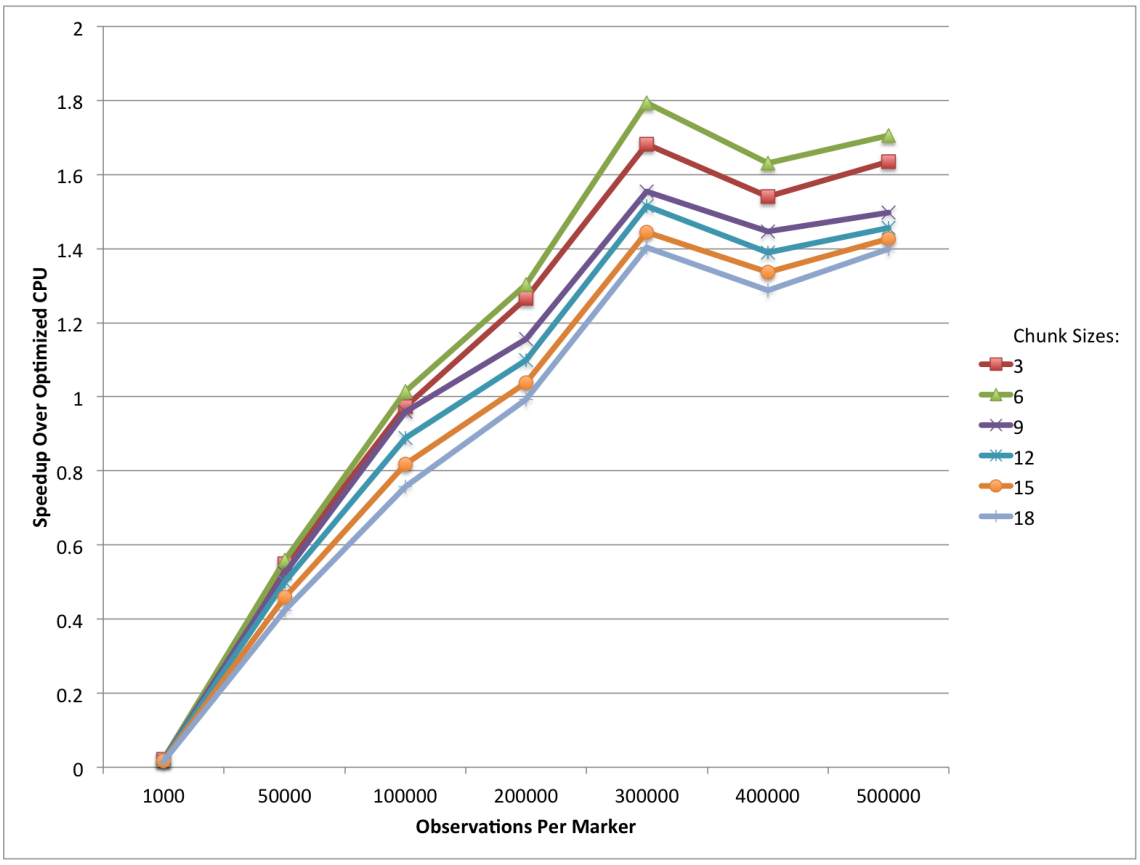

Figure 9.3: Effect of different chunk sizes on speedup

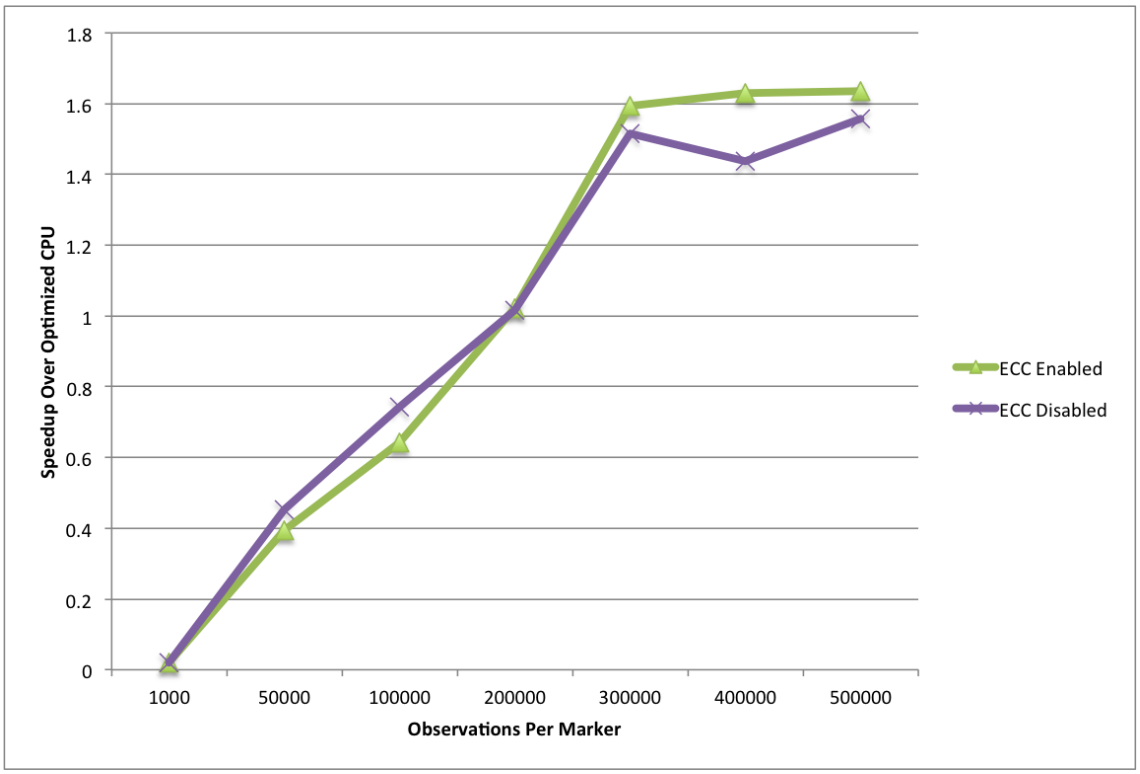

Figure 9.4: Effect of ECC memory on speedup

Enabling or disabling ECC memory on the GPUs has the potential to effect ReGen's runtimes as well. ECC memory protects against data corruption but in turn may lower performance due to additional overhead costs. For this application, enabling or disabling ECC memory was not found to have a significant effect on performance. Figure 9.4 shows the performance difference between two analyses, one performed with ECC memory enabled and the other disabled.

9.2 Scalability

ReGen was designed to scale easily and use all computing resources available. One of the most impactful improvements on performance can be achieved by adding more GPUs to the system. As the number of observations increases the

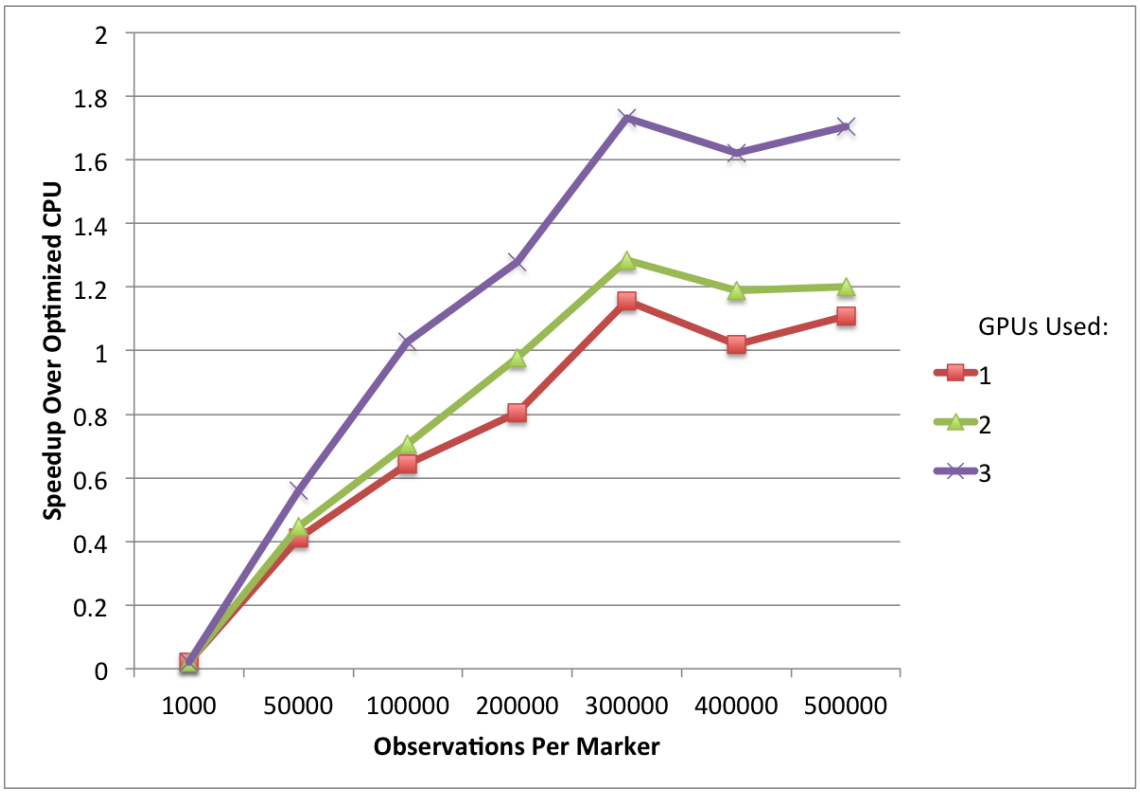

Figure 9.5: Effect of scaling the number of GPUs utilized on speedup

GPU computing load increases much faster than the CPU computing load. This imbalance can lead to inefficiency. By spreading the load across multiple GPUs this imbalance can be mitigated. Figure 9.5 shows the effect of adding additional GPUs to the system. The GPUs that compose the system have a priority assigned to them. So, in this test the K40 was always used due to its high priority. As the number of GPUs was increased the K20Xs were added.

CHAPTER 10

Conclusions

The heterogeneous computing environment offers many advantages for high performance compute problems. It opens up a whole new world of parallel computing possibilities. Leveraging this technology is a non-trivial task. A paradigm shift in the design and implementation of solutions is required.

ReGen illustrates how even notoriously difficult to parallelize algorithms can benefit from the heterogeneous compute environment. By carefully restructuring the original algorithm found in GenSel, ReGen embraces the parallel computing paradigms to the fullest. Concepts of TLP and DLP are taken to extreme measures to make sure work is always being performed on the CPU and GPU, results are computed as early as possible to reduce wait times, transfer times are hidden, and kernels are pipelined. ReGen is able to achieve a 1.84 times speedup over the highly optimized CPU implementation. This performance gain will ensure that genetic analyses continue to be performed in a reasonable timeframe as the size of data sets increases.

CHAPTER 11

Future Work

The current implementation of ReGen offers significant improvement over the original implementation but there remain several areas that could be improved or explored. In its current state, ReGen relies heavily on hand configuration to achieve optimal performance. This limits ReGen's usability in two ways. First, it makes learning to use the tool significantly more difficult and time consuming. Second, the lack of an automated method for setting configuration parameters makes setting up ReGen on a new system very time consuming. Ideally, ReGen should be able to detect the configuration of the system and choose the best parameters by default while still maintaining the ability to be manually overridden. Taking this a step further, ReGen should also choose the optimal computation strategy between CPU and heterogeneous computing. These modifications would significantly improve the usability and flexibility of the system.

ReGen is currently a very greedy application. When running on large data sets it has a very large memory footprint. Much of this memory is page locked to allow for asynchronous data transfer. This can be problematic because it can negatively impact other users on the system and if the data set is truly massive can cause the system to grind to a halt. If data sizes increase faster than the memory capacity of the systems ReGen is being run on these problems may occur in the future. To prevent this, the existing solution should be enhanced to

efficiently predict what data needs to be transferred and swap data in and out of a small portion of pinned memory as necessary.

Several alternative algorithms for performing Bayesian inference could provide significant speedups. Block Gibbs sampling builds upon the basic idea of current algorithm and seeks to shrink the individual links of the chain. Block Gibbs sampling enables larger chunks of data to be processed at once leading to increased parallelism.

Another potential avenue to squeeze performance out of is through generating more efficient machine code. The use of industrial compilers, such as PGI, should be explored. It would be interesting to see how much, if at all performance is affected.

BIBLIOGRAPHY

[1] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. *Clock rate versus IPC: The end of the road for conventional microarchitectures*, volume 28. ACM, 2000.

[2] Anton A Béguin and Ceec AW Glas. MCMC estimation and some model-fit analysis of multidimensional IRT models. *Psychometrika*, 66(4):541–561, 2001.

[3] Stephen Brooks. Markov chain Monte Carlo method and its application. *Journal of the royal statistical society: series D (the Statistician)*, 47(1):69–100, 1998.

[4] Gregory F Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2):393–405, 1990.

[5] Gordon V Cormack and Thomas R Lynam. Online supervised spam filter evaluation. *ACM Transactions on Information Systems (TOIS)*, 25(3):11, 2007.

[6] Michael Evans, Tim Swartz, et al. Methods for approximating integrals in statistics with special emphasis on Bayesian integration problems. *Statistical Science*, 10(3):254–272, 1995.

[7] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004*

ACM/IEEE conference on Supercomputing, page 47. IEEE Computer Society, 2004.

[8] James E Gentle, Wolfgang Härdle, and Yuichi Mori. *Handbook of computational statistics*. Springer, 2004.

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

[10] 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.

[11] Cecil Huang and Adnan Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 15(3):225–263, 1996.

[12] K Kizilkaya, RL Fernando, and DJ Garrick. Genomic prediction of simulated multibreed and purebred performance using observed fifty thousand single nucleotide polymorphism genotypes. *Journal of animal science*, 88(2):544–551, 2010.

[13] 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.

[14] Anthony O'Hagan, Jonathan Forster, and Maurice G Kendall. *Bayesian inference*. Arnold London, 2004.

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

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

[17] NVIDIA Corporation. CUDA toolkit documentation – cuBLAS. `http://docs.nvidia.com/cuda/cublas/`, February 2014.

[18] Fredrik Ronquist and John P Huelsenbeck. Mrbayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics*, 19(12):1572–1574, 2003.

[19] 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 *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[20] Mahdi Saatchi, Mathew C McClure, Stephanie D McKay, Megan M Rolf, JaeWoo Kim, Jared E Decker, Tasia M Taxis, Richard H Chapple, Holly R Ramey, Sally L Northcutt, et al. Accuracies of genomic breeding values in american angus beef cattle using k-means clustering for cross-validation. *Genet Sel Evol*, 43:40, 2011.

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

[22] 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.

[23] Luke Tierney. Markov chains for exploring posterior distributions. *the Annals of Statistics*, pages 1701–1728, 1994.

[24] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.