2014

GPU optimizations for a production molecular docking code

Landaverde, Raphael J.

Boston University

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

GPU OPTIMIZATIONS FOR A PRODUCTION MOLECULAR DOCKING CODE

by

RAPHAEL J. LANDAVERDE

B.S., Boston University, 2012

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2014

Approved by

First Reader

Martin Herbordt, PhD
Professor of Computer Engineering

Second Reader

Sandor Vajda, PhD
Professor of Biomedical Engineering

Third Reader

Florian Raudies, PhD
Research Assistant Professor of Computational Neuroscience

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
–Martin Fowler

Acknowledgments

I would like to thank Professor Herbordt for taking me as a Research Assistant after receiving my undergraduate degree. His help in my research and his knowledge of computer architecture has helped me grow in my knowledge and love of computers.

I would also like to thank Professor Vajda and Professor Kozakov for their help in understanding the PIPER code. Dr. David Hall also proved invaluable in helping me understand PIPER, as well as providing me testing data, result files for verification, and resources for benchmarking.

Finally, I would like to thank Cali Stephens, Austin Alexander, and the rest of the staff in the Boston University ECE office for their help throughout graduate school, and in assembling this thesis.

GPU OPTIMIZATIONS FOR A PRODUCTION MOLECULAR DOCKING CODE

RAPHAEL J. LANDAVERDE

ABSTRACT

Scientists have always felt the desire to perform computationally intensive tasks that surpass the capabilities of conventional single core computers. As a result of this trend, Graphics Processing Units (GPUs) have come to be increasingly used for general computation in scientific research. This field of GPU acceleration is now a vast and mature discipline.

Molecular docking, the modeling of the interactions between two molecules, is a particularly computationally intensive task that has been the subject of research for many years. It is a critical simulation tool used for the screening of protein compounds for drug design and in research of the nature of life itself. The PIPER molecular docking program was previously accelerated using GPUs, achieving a notable speedup over conventional single core implementation. Since its original release the development of the CPU based PIPER has not ceased, and it is now a mature and fast parallel code. The GPU version, however, still contains many potential points for optimization.

In the current work, we present a new version of GPU PIPER that attains a 3.3x speedup over a parallel MPI version of PIPER running on an 8 core machine and using the optimized Intel Math Kernel Library. We achieve this speedup by optimizing existing kernels for modern GPU architectures and migrating critical code segments to the GPU. In particular, we both improve the runtime of the filtering and scoring stages by more than an order of magnitude, and move all molecular data permanently to the GPU to improve data locality. This new speedup is obtained while retaining a computational accuracy virtually identical to the CPU based version. We

also demonstrate that, due to the algorithmic dependencies of the PIPER algorithm on the 3D Fast Fourier Transform, our GPU PIPER will likely remain proportionally faster than equivalent CPU based implementations, and with little room for further optimizations.

This new GPU accelerated version of PIPER is integrated as part of the ClusPro molecular docking and analysis server at Boston University. ClusPro has over 4000 registered users and more than 50000 jobs run over the past 4 years.

Contents

1 Introduction **1**

1.1 The Molecular Docking Problem 1

1.2 GPU Acceleration 3

1.3 Contributions 5

1.3.1 Newly Accelerated PIPER 5

1.3.2 Limit of GPU Speedup 6

1.4 Organization of the Paper 6

2 Graphics Processing Units **8**

2.1 Overview 8

2.2 Kepler Architecture 10

2.3 The CUDA Model 15

2.4 Goals of GPU Computing 17

3 PIPER and Molecular Docking **19**

3.1 Overview 19

3.2 Molecular Docking 20

3.3 PIPER 21

3.4 PIPER Program Flow 25

3.5 Recent GPU Work 26

4 Optimizations **28**

4.1 GPU PIPER 28

4.2 Odd Size Grid Volume 31

4.3 Filtering and Scoring Stage Optimizations 32
4.4 GPU Idle Time Optimizations 35

5 Results **39**

5.1 Target Hardware 39
5.2 Protein Complexes 39
5.3 Optimization Results 40
5.4 Validation 46

6 Conclusion **49**

6.1 Optimization Discussion 49
6.2 Further Work 50

References **52**

Curriculum Vitae **57**

List of Tables

5.1 Comparison of target hardware. 40

5.2 Comparison of GPU and CPU outputs after clustering phase. The number within 10 Angstrom are the best selected poses for the complex. The near native rank and interface RMSD are for the best pose selected for each complex. 48

List of Figures

2·1 SMX structure for the Kepler Architecture [Nvi09] 12

2·2 Memory Hierarchy for the Kepler Architecture. The memory architecture has been simplified since the Tesla generation from the point of view of the programmer [Nvi09] 13

2·3 Thread Hierarchy for the CUDA model. Different levels of memory are accessible by different groups of threads at varying levels of granularity. [Nvi12] 17

3·1 Visualization of two proteins docking [SH09c] 21

3·2 Examples of protein poses [SH09c] 22

3·3 CPU work distribution during a single rotation [SH09c] 26

4·1 The flow of the PIPER program. Light green boxes are on the CPU, dark green boxes are on the GPU, and light blue boxes are being moved to the GPU in the current work. 29

4·2 Proportion of rotation step spent on various stages of the computation. The FFT and modulation steps are grouped as the GPU core computations. Filtering/Scoring and the CPU calculations take the majority of the time. 30

4·3 Filtering Stage 1 kernel. Each SM may write multiple partial scores to global memory based on how many blocks were assigned to that SM. 34

4·4 Filtering Stage 2 kernel. A single SM performs the work of reducing the partial scores in global memory. 35

4·5 NVProf output memory latency relative to correlation compute time on the K20c GPU. A longer bar indicates a longer execution time. . . 36
4·6 Grid volume for the protein complexes in the Zlab Protein Docking benchmark. Each bar represents an individual protein complex and its grid volume. 37

5·1 GPU runtime improvement with bar segments separating the different computations. Prior to the CPU idle time optimizations, the correlation time was actually hidden by the memory transfer time, as it was larger. 42
5·2 Relative improvement for various configurations. The red bar is CPU PIPER, blue bars on Kepler HW, and the green bar is Fermi HW. . . 43
5·3 Portion of rotation step spent on various parts of the computation after optimizations. 44
5·4 Comparison of the runtime for 3D FFTs with varying dimensions to the improvement in runtime demonstrated in PIPER docking for Intel, Fermi, and Kepler HW configurations 45

List of Abbreviations

AA	.....	Antigen/Antibody
API	.....	Application Programming Interface
CPU	.....	Central Processing Unit
CUDA	.....	Compute Unified Device Architecture
CUFFT	.....	CUDA Fast Fourier Transform
FFT	.....	Fast Fourier Transform
FLOPs	.....	Floating-point operations per second
GB	.....	Gigabytes, 2^{30} (10^9) bytes
GFLOPs	.....	10^9 floating-point operations per second
GHz	.....	Giga Hertz, 10^9 cycles per second
GP	.....	General Purpose
GPU	.....	Graphics Processing Unit
HPC	.....	High Performance Computing
KB	.....	Kilobytes, 2^{10} (10^3) bytes
MB	.....	Megabytes, 2^{20} (10^6) bytes
MHz	.....	Mega Hertz, 10^6 cycles per second
MKL	.....	Intel Math Kernel Library
MPI	.....	Message Passing Interface
PC	.....	Personal Computer
PCI	.....	Peripheral Component Interconnect
PCIe	.....	PCI express
RAM	.....	Random Access Memory
RMSD	.....	Root mean square deviation
SIMD	.....	Single Instruction, Multiple Data
SIMT	.....	Single Instruction, Multiple Thread
SM	.....	Streaming Multiprocessor
SP	.....	Streaming Processor
TB	.....	Terabytes, 2^{40} (10^12) bytes
TFLOPs	.....	10^{12} floating-point operations per second

Chapter 1

Introduction

1.1 The Molecular Docking Problem

The cost of drug development over the past decades has been steadily increasing, with many studies indicating that the cost is already immense and constantly increasing [AB10; DHG03]. This is in addition to the long drug development time, often ten years or more, and investment losses when drugs fail before ever making it to market [DHG03]. Computational methods have therefore been in use for many years to help alleviate the cost, particularly by helping screen drug candidates from large libraries of chemical structures [SCFR10]. In this way only likely candidate drugs can have the proper resources invested. These computational methods are specific to many aspects of the drug development and research process, and their development and use is an important aspect of modern drug design.

One field where a significant amount of computational research has been invested is the modeling of the interaction between two molecules, otherwise known as molecular docking. Docking is critical both for the development of novel drugs and for research into the nature of life itself. More clearly, the main goal of molecular docking is to accurately model the structure of molecular interactions as well as correctly predict chemical activities that will occur when the molecules come into contact [KDFB04]. With this in mind, molecular docking algorithms are typically grouped into protein-ligand docking, where the protein is a large molecule while the ligand is small, and protein-protein docking, where both of the molecules are large. The former is pri-

marily used in drug development, since the drug is typically a smaller ligand used to affect the behavior of the protein. The latter is fundamentally used for understanding basic life processes, but can also be useful in drug design as well.

Molecular docking is a computationally expensive task, even after many modeling assumptions have been made to minimize the complexity. Nonetheless, the assumptions and simplifications to the algorithm are critically important in the development of fast and accurate algorithms. A common simplification relevant to the current work is that of rigid-body molecular docking. In rigid-body docking, the molecules are assumed to not be flexible during the evaluation of the docked conformations. Many state-of-the-art docking algorithms use the rigid docking assumption in order to be effective for analyzing protein interaction in a high performance manner, especially since drug development time is a key factor in the overall cost [DHG03].

One such program that was developed for the express purpose of accurately modeling the docking of protein complexes with high performance is PIPER, a molecular docking code developed by the Structural Bioinformatics Lab at Boston University [KBCV06]. This program is high performing and highly parallel, and as part of the ClusPro molecular analysis server, is considered the best computational predictor of protein interaction by the CAPRI ranking, an experiment for the evaluation of protein interaction [LW13]. However, the program is still computationally expensive and time consuming, and for this reason, the Computer Architecture and Design lab at Boston University has accelerated the PIPER program, as well as other ClusPro applications, first with FPGAs [VGH04; VH06; SH08; SH09b; SH10b] and later with graphics processing units (GPUs) [SH09c; SH10a; SH09a; SH14].

Since this acceleration, the field of computational molecular docking has progressed, with many other docking programs accelerated by GPUs [SIS+13; RV10; PF11; SCTP13] and GPU hardware significantly improving since the late 2000s.

This thesis presents optimizations for the GPU version of PIPER on modern GPUs, attaining notable speedup over the previous version, as well as speedup over the mature and parallel version of PIPER currently available and in use. This optimized GPU version of PIPER has been fully integrated into the public ClusPro server at Boston University, and is available for use.

1.2 GPU Acceleration

Scientists have always felt the desire to "perform more and larger computations" that quickly began to surpass the capabilities of conventional single core computers [LG09]. The first solution to this was the introduction of multicore computers that could process multiple operations in parallel. However, for the level of parallel computation expressed in many scientific fields, Nvidia introduced the concept of the general purpose GPU for scientific computing, that could effectively handle certain parallel problem spaces [LNOM08].

Acceleration of scientific computations via GPUs has been a hot field in the 7-8 years since the introduction of the Compute Unified Device Architecture (CUDA) and the general purpose GPU [LNOM08]. This development brought a surge of accelerator based implementations of important algorithms that presented notable performance improvements over CPU implementations of the same computations. Examples of some of these accelerated programs include acceleration of pair potential calculations in molecular modeling [RHS+08], acceleration of numerical weather prediction [MV08], and acceleration of signal processing for geophysical signal processing [WH11].

However with the large gains come limitations. Development of GPU accelerated applications is difficult for a number of reasons, including:

- **Amdahl's Law:** Even with most computations accelerated in an application,

the overall system will always be limited by the non accelerated components of the computation. This is true with the original GPU version of PIPER as it is now limited in its performance by a few costly program stages.

- **Difficulty of Parallel Programming:** Parallel programming of any kind introduces significant complexity and often requires extensive refactoring. For this reason, many GPU application require repeated tuning and extensive development to attain significant performance improvements.

- **Difficulty of Mapping Algorithms to GPU:** Not all algorithms that are computational intensive map well to a GPU. For many applications, complex refactoring of the algorithm is necessary for achieving any performance gains. Along with this, the developers must be aware of the limited resources as well as the unique architecture in order to design algorithms suitable for the highly parallel GPU.

- **Hardware Dependence:** As GPUs improve, the micro-architecture of the GPU changes as well, requiring code already running on old GPU architecture to be updated to take advantage of architectural changes, even simply to maintain performance benefits over CPU implementations. This particular issue is significant to the current work.

For these reasons, there have been discussions as to the limit of GPU acceleration, and the performance gains that should be expected in the future [VCC+10]. The current work touches on the limits possible via GPU acceleration, but as will be discussed, GPU performance still outperforms equivalent CPU based approaches for molecular docking.

In this thesis, we take the GPU accelerated version of PIPER [SH09c] originally developed and in use, and accelerate it even further in order to find the limit for

improving its performance, as well as ensure that it always performs better than the now mature, optimized, and parallel CPU based version of PIPER [KBCV06].

1.3 Contributions

This thesis has two main contributions. The first is a newly accelerated version of PIPER that performs 3.3x better than the best CPU version available. The second is in finding the limits of the GPU acceleration possible for the PIPER algorithm as a result of the limits of GPUs and the nature of the algorithm itself.

1.3.1 Newly Accelerated PIPER

Our new version of PIPER is accelerated on both a readily available C2075 Fermi architecture GPU [Nvi11] and a state of the art K20x Kepler architecture GPU [Nvi13]. The C2075 is one of the most commonly used computational GPUs available on the market today and provides key improvements over the architecture PIPER was accelerated on originally, while the K20c has architectural improvements that provide even greater speedup, but is not as widely distributed. With our new version and using the brand new K20c GPU, we achieve a 3.3x speedup in rotation time over the fastest, 8 core version of PIPER available, as well as a 6.6x speedup over the original version of GPU PIPER. With the more widely distributed C2075 GPU, we still attain a 2X improvement over the CPU based PIPER. These improvements are substantial, as they are optimizations layered on top of the original optimizations already designed. This newly accelerated version of PIPER is fully integrated into the ClusPro molecular analysis server at Boston University, and is now available for use.

1.3.2 Limit of GPU Speedup

Our new version of PIPER also demonstrates a key facet of limited GPU speedup. We discuss how after our optimizations, the GPU version of PIPER is tied to the performance of the Fast Fourier Transform (FFT) library provided by Nvidia (CUFFT) [Nvib], and its speedup with respect to the analogous Intel FFT library. As we discuss later, if the algorithms were to be minimized down to the FFT computations, the GPU version will never have greater than 3.3x speedup over the CPU version for the Kepler architecture. Although we move all computations to the GPU, effectively hiding the latency of any left over CPU computations, we now experience similar effects to Amdahl's Law on the GPU scale now, as the algorithm is limited by the CUFFT library. Along with this aspect, the PIPER algorithm is innately scalable to many processors, owing to the fact that each iteration of the critical loop is independent from one another. Thus, accelerating the inner loop will always be at odds with simply distributing iterations across many cores.

1.4 Organization of the Paper

The rest of the thesis provides a detailed discussion on the various aspects of the GPU acceleration, including GPU architecture, molecular docking and a discussion of the PIPER program, optimization methods applied, our experimental results, and a discussion of the implications. These topics are organized into 5 chapters.

Chapter 2 discusses GPU architectures. We provide an overview on the internals of an Nvidia GPU, particularly the K20c card that we use for the optimization process. We also provide a discussion on the techniques used for programming the card itself.

Chapter 3 gives a background on molecular docking and the PIPER program. We discuss the details of molecular docking, and in particular, the details of the PIPER algorithm and the computational assumptions made. A brief overview of the

state of the art of GPU docking algorithms is also presented.

Chapter 4 presents the analysis on the key optimization points in the original version of GPU PIPER. We then discuss the optimizations we implemented to improve performance, as well as the reason as to why they were hindering performance.

Chapter 5 provides the results of the optimizations via a series of experiments. The target hardware is presented in detail, and we discuss the different configurations and the meaning of our results.

Chapter 6 concludes the thesis with a discussion of our results and direction for future work in the acceleration of molecular docking.

Chapter 2

Graphics Processing Units

2.1 Overview

GPUs originated from a need for dedicated to rendering computer graphics. They were created solely as coprocessors for the CPU, used for offloading the task of pixel rendering, a task which is typically computationally intensive. They could then take this intense computational workload of rendering graphics and accelerate it significantly when compared to similar CPU rendering techniques [LNOM08]. The graphics card, which contains the chip and on board memory, is an independent processor connected to the motherboard via PCI or PCIe slots, and is then queried by the CPU, a paradigm that has remained dominant over the past decades. Data is transferred from system memory to the GPU, which then processes the data, generating millions of pixels data per second, and directly rendering the data to a display device [LNOM08].

Early GPUs contained a fixed graphics pipeline optimized for fast and fixed vertex and pixel computations on data that required very low precisions [LNOM08]. Over time, GPUs evolved to handle higher levels of precision, run a more complex graphics pipeline for more flexible and programmable computations, and contain significantly more processing cores for generating a higher throughput of data. These capabilities were all designed with the express purpose of improving graphics performance in fields such as simulations, computer generated imagery, and video games. However, the greater flexibility created as well as the sheer compute power capable on the chips

made GPUs prime candidates for high performance computing acceleration, a field where no graphics output is necessary.

With the benefits of GPUs evident in terms of raw computing power, the development of high level programming languages and interfaces such as Compute Unified Device Architecture (CUDA) [Nvia] and OpenCL [Grob] facilitated the mapping of preexisting algorithms to graphics processors. The innate massively parallel design of GPUs accessed via programming interfaces provided a gateway for accelerating parallel scientific computations without the need for mapping algorithms unnaturally as pixel computations. The combination of this parallel architecture design and the available interfaces led to many scientific applications being mapped to GPUs, such as with the topic of this thesis, molecular docking [SH09c]. Furthermore many of the top modern supercomputers, such as the Titan supercomputer, integrate GPUs using CUDA for increasing their throughput enormously [Lab].

Beyond the massive quantity of cores present on a GPU, a primary factor for the raw performance delivered by GPUs is the relative absence of typical caches, as they are found on normal CPUs. Unlike CPUs, where much of the die on a chip is devoted to on chip cache for hiding the large discrepancy in memory latency vs clock time, GPUs have very specialized caches that are lightweight and are primarily controlled via code [LNOM08]. Thus, most of the transistors on the chip are devoted to computation cores with comparably little control logic [Nvi09; Nvi12]. To hide memory latencies, GPUs rely on having a large number of active light-weight threads and a very fast and efficient single-cycle context switch between active threads waiting for data and those ready to execute [LNOM08]. Ideally, owing to the GPU heritage as a processor for graphics, each single thread would act on unique data, minimizing the need for data sharing and caching. This provides the groundwork for immense parallelism, with coarse-grained parallelism at a computation core level, and fine

grained parallelism with individual threads.

During the development of the GPU version of the PIPER program, the architecture used for development was the Nvidia Tesla Architecture [SH09c], and in particular the C1060 card in the Tesla family [Nvi08]. Nvidia graphics cards are organized by architecture and card, where each card is devoted to various purposes. For a given architecture, graphics cards are created designed for computation, like the C1060, K20c, and C2075, or designed for graphics, such as the GeForce GTX series of cards. As is discussed in the Kepler Architecture section, the majority of the concepts discussed for the Tesla architecture are still in use. The current state of the art is the Kepler architecture, and the most commonly available and readily affordable computational graphic cards are based off of the prior generation Fermi architecture, of which the Tesla C2075 is a member [Nvi09]. The Kepler architecture will primarily be discussed, and notable improvements over the Tesla architecture will be touched upon, particularly in the context of its benefit for the GPU version of PIPER. However, since our optimizations are based off of new architectural improvements introduced during the Fermi generation, the new version of GPU PIPER is compatible with both the Fermi and Kepler generation.

2.2 Kepler Architecture

The Kepler architecture of the K20c GPU has improved on various aspects of the Tesla architecture. Just like the Tesla architecture C1060 card used before, the smallest atomic unit of the GPU are the Scalar Processors (SP), grouped into clusters. In the Tesla architecture, these clusters are called Streaming Multiprocessors (SMs), a unit which the Kepler architecture refers to as an SMXs. For the sake of simplicity in comparison, the Kepler SMX will be referred to as an SM. Compared to the Tesla architecture, the Kepler architecture groups 192 SPs into a single SM, rather than

8 to a single SM, as is seen in Figure 2·1 [Nvi12]. On the C1060, there were 30 SMs for a total of 240 SPs, but now, the K20c contains 13 SMs into a total 2496 SPs, increasing computational cores by an order of magnitude. This update in the quantity of processing units lead to an increase in the FLOPs (floating-point operations per second) possible by the GPU to 3.52 single precision TFlops [Nvi13]. The C2075 card provides various improvements over the C1060 as well, with 32 SPs per SM and a new peak performance of 1.05 TFlops [Nvi09; Nvi11].

Identically with the C1060, the K20c is a Single Instruction Multiple Thread (SIMT) processor. A SIMT processor operates very similarly to the more common Single Instruction Multiple Data (SIMD) paradigm. In a SIMD processor, a single instruction operates on a vector of data simultaneously. With a SIMT processor, a single instruction operates on a group of threads in lock step [LNOM08]. This occurs by grouping computing threads together onto SMs, and applying instructions to groups of 32 threads (a logical group called a warp) at a time on each SM. Since the number of SPs within an SM is now greater than the warp size on the K20c, there is a notable performance increase over the Tesla architecture. However this also requires that the number of threads assigned per SM must be newly tuned to accommodate the architectural changes.

Although the number of SPs per SM has changed, the behavior of communication between SMs has not changed across these two architectures. Just as in the Tesla architecture, each SM can only communicate within itself, or more clearly, the threads placed within an SM can only communicate with other threads on the same SM [LNOM08]. Communicating across SMs requires a global resynchronization of the GPU via the host CPU.

The memory hierarchy of the K20c GPU changed in a few ways, and in particular, has been simplified for easier general purpose computational use, as is seen in Figure

Figure 2·1: SMX structure for the Kepler Architecture [Nvi09]

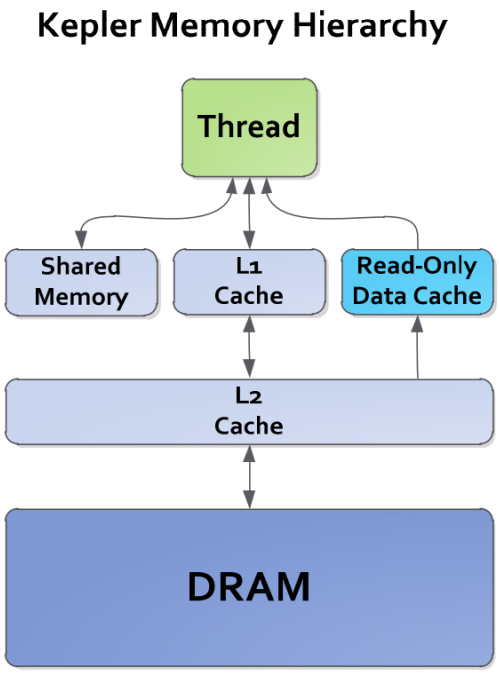

Figure 2·2: Memory Hierarchy for the Kepler Architecture. The memory architecture has been simplified since the Tesla generation from the point of view of the programmer [Nvi09]

2·2. For a Kepler GPU, there are four major types of memory [Nvi12]:

- **Shared memory** that can also now be configurable as an L1 cache. This shared memory has increased in size by 4x to 64KB to allow for more low latency high bandwidth memory. When used as a cache, it is no longer programmer managed, but automatically caches accesses to global memory.

- **L2 Cache**, a new level of memory hierarchy intended to minimize the apparent latency of global memory. Unlike the majority of GPU memory, this memory is no longer managed explicitly, and behaves much more closely to CPU caches.

- **Global memory**, which remains the bulk of the memory of the GPU, has the longest memory access latency, and is stored in off-chip DRAM. Global memory on the K20c card has increased from 4GB to 5GB [Nvi13]. Starting with the Fermi generation and continuing in the Kepler generation, new memory instructions were added that allow for floating point atomic operations that were not possible on the C1060 card [Nvi09; Nvi12].

- **Read-Only Data Cache**, a 48KB cache for read only data that was originally only accessible by the graphics based texture unit. For the Kepler architecture, is was made transparent for the use of developers.

With the exception of the read only data cache, the Fermi architecture uses this same memory hierarchy. The increased global memory size is an optimization point for improving on the GPU version of PIPER, and the appearance of the L2 cache already provides a distinct performance increase over the previous generation. Of note however, is that one of the important requirements of global memory is memory alignment. For many external libraries to function properly, memory addresses must be aligned to 8 or 16 bit boundaries [Nvi09; Nvia; Nvib].

Along with the increased size of memory, the memory transfer bandwidth from the CPU to the GPU has also increased [Nvi11; Nvi13]. However, minimizing data transfer to as little as possible is a focus of the optimizations performed on PIPER.

2.3 The CUDA Model

The programming model has not changed dramatically since the Tesla architecture was first released. Traditionally, graphics processors required graphics APIs such as OpenGL [Groa]. However, at the release of the Tesla architecture, Nvidia also released the Compute Unified Device Architecture (CUDA) [LNOM08]. CUDA is a thread-based data-parallel programming model built on top of C/C++ for programming NVIDIA GPUs [Nvia]. CUDA enables threads to be launched on a GPU, logically grouped onto SMs for execution in SIMT. CUDA also introduces the abstraction of a kernel, which is simply a function that runs in parallel on a GPU [Nvia].

Since CUDA is based on C/C++, it provides extensions for kernels, kernel launches, memory transfers to the GPU, shared memory, thread organization, and many other concepts. The general model for a CUDA program is:

1. CPU serial code and data initialization
2. Data transfer to the GPU
3. Parallel kernel execution
4. Data transfer back to the CPU.

For mapping algorithms to the GPU, CUDA provides a general framework with three main abstractions [Nvia]. These abstractions aid the programmer, but also help in simplifying the architectural model into a form that is much easier to interact with. These abstractions are:

- **Thread Hierarchy:** Threads can be organize into thread blocks. Thread blocks can be 1, 2, or 3 dimensions, but the dimensionality is used primarily for mapping threads to an algorithm. Each thread block is sent to a single SM. Thread blocks are then grouped into a 1 or 2 dimensional grid of blocks. The thread scheduler on a GPU assigns blocks to SMs. The quantity of blocks must be appropriate to fully utilize the GPU

- **Shared Memory:** Each SM contains low latency shared memory. Threads in each thread block can share this memory, and CUDA provides mechanisms for moving data from Global Memory to shared memory for data that is accessed regularly and must be shared amongst threads in a block.

- **Synchronization:** CUDA provides a mechanism for threads within a thread block to be synchronized. However, as it was in the Tesla architecture, there is no synchronization method across threads between blocks [Nvi09; Nvi12].

Figure 2·3 shows the hierarchy of threads to memory on a GPU. A single thread gets local memory in the form of registers, groups of threads in a block share access to shared memory, while the entire collection of blocks in a grid share access to global memory.

The main goal of the CUDA model, and how it is used in this paper, is organizing threads to best use the memory hierarchy and minimize the amount of time spent outside of a kernel, in data transfer, or using the CPU alone.

Another aspect of the CUDA model that is regularly used is the concept of streams. A CUDA stream is a sequence of operations that execute in issue order on the GPU [Nvia]. Streams are typically used to partition data into data transfer streams and execution streams. The Fermi and Kepler architectures allow for data transfer streams and kernel streams to execute concurrently. This was Nvidia's primary solution to

the memory and bandwidth limit for a GPU, as it allows data transfer to ideally be hidden by the execution of a kernel. Along with this, the Kepler GPU has multiple compute streams that can smartly choose to execute multiple compute kernels to execute in parallel, if SMs have available occupancy [Nvi12].

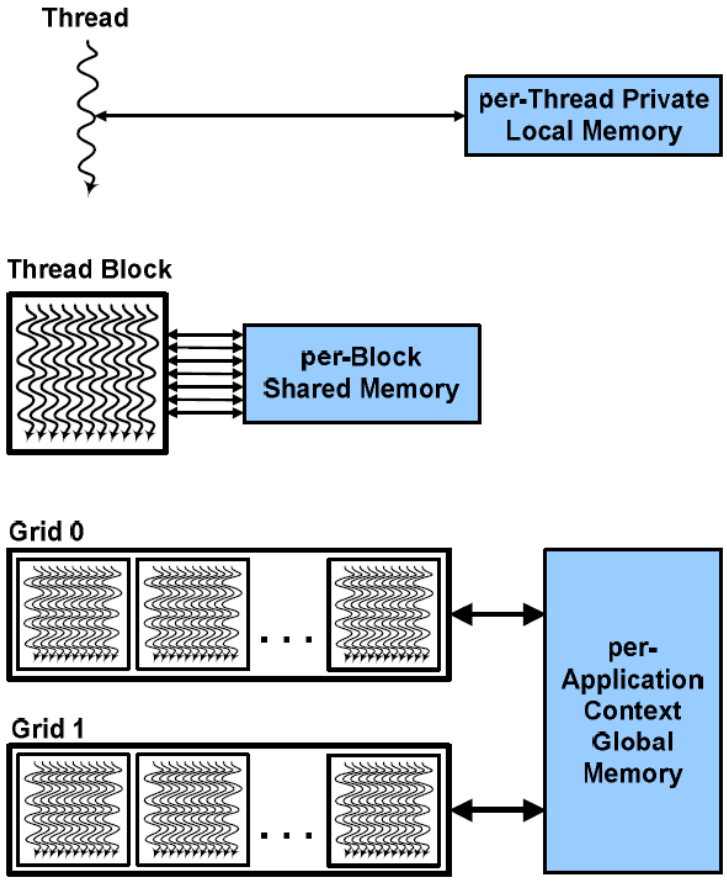

Figure 2·3: Thread Hierarchy for the CUDA model. Different levels of memory are accessible by different groups of threads at varying levels of granularity. [Nvi12]

2.4 Goals of GPU Computing

Just as it was during the development of the first version of GPU based Piper, there are challenges associated with GPU computing, and minimizing or alleviating their

effect is critical to attaining performance speedup. For optimizations discussed in the current work, we dealt with the following challenges:

- **Distribution of Work:** The initial design of GPU Piper emphasized distribution of work amongst threads. However, many of the optimizations we developed involved improving the work distribution even further so that at any given time, the GPU would not have idle SMs.

- **Data Transfer Overhead:** The explicit transfer of data to the GPU, if sufficiently large, can have a notable latency, and limits the performance gains attainable by parallel kernels. Minimizing the quantity of times data is transferred is critical for better performance.

- **CPU Overhead:** The performance of the GPU is limited by the speed of the CPU during non parallel code segments. Thus, minimizing the amount of CPU computations can improve GPU performance, allowing the GPU to be executing without delay.

The optimizations we develop in the current work all take into account the above challenges to attain the speedup we present.

Chapter 3

PIPER and Molecular Docking

3.1 Overview

The modeling of molecular activities and the interactions between proteins and ligands is a vast and important field of research. Molecular docking in particular is one of the fundamentals methods used for discovering and investigating the behavior of proteins, in the discovery and design of novel drugs, as well as for the modeling of entire protein groups[LW13]. Docking has been used effectively in the past in the design of drugs, including drugs such as HIV protease inhibitors and the peptide antigens for the MHC receptors [RVD95; KDFB04]. Yet for effective use in drug design in delivery, accuracy and speed are both important factors that tend to be at odds.

The main aims of molecular docking are accurate structural modeling of interaction, as well as correct prediction of activity [KDFB04]. Many different docking algorithms have many different approaches for analysis, but since many organic molecules have many degrees of freedom, modeling them with sufficient accuracy is computationally challenging [KDFB04]. Thus, to minimize the difficulty caused by having multiple degrees of freedom, approximations are typically used, with a common approximation being the assumption that the molecules being analyzed are rigid and not flexible.

Because of previous difficulties in computational intensity and time, molecular docking has remained a hot topic of study over the past several years. During this time, much development has occurred in creating novel docking programs that are

either faster, more accurate, or both. Many computational docking programs have been created such as PIPER (a part of the ClusPro molecular docking and prediction server) [KBCV06], the main focus of GPU acceleration in Sukhwanis work [SH09c] and also the focus of further optimizations in the current work. Other work in this field include FTDock [GJS97], ZDOCK [CLW03], Hex [MMV+10], AutoDock [PF11], Megadock [SIS+13], and GRAMM-X [TV06]. These other works use different algorithms in their docking phase for accuracy or speed purposes. However for the purposes of the current work, we focus on PIPER and its GPU version, namely because we place emphasis on improving the prior work without adjusting the fundamental docking algorithm.

3.2 Molecular Docking

Molecular docking is the computational prediction of molecular interactions and investigation of the intermolecular complex formed when two molecules, typically two proteins, interact. As was mentioned earlier, the field is vast, and there are many molecular docking programs that have been created to solve this problem. Figure 3·1 below is a visualization of molecular docking.

Docking begins by taking the atomic coordinates of two molecules, and then predicting their correct bound structure via computational methods [HMWN02]. These methods involve positioning the two molecules relative to each other, both their rotation and relative offset (the pose), as a way of predicting likely interacting poses. These poses are then evaluated via energy functions, the energy functions and their evaluation being one of the key variation between different molecular docking programs. Because of the computational run times and the complexity of the energy computation, these energy evaluation functions tend to make simplifying assumptions [KDFB04]. Molecules are also typically flexible and this can change the methods in

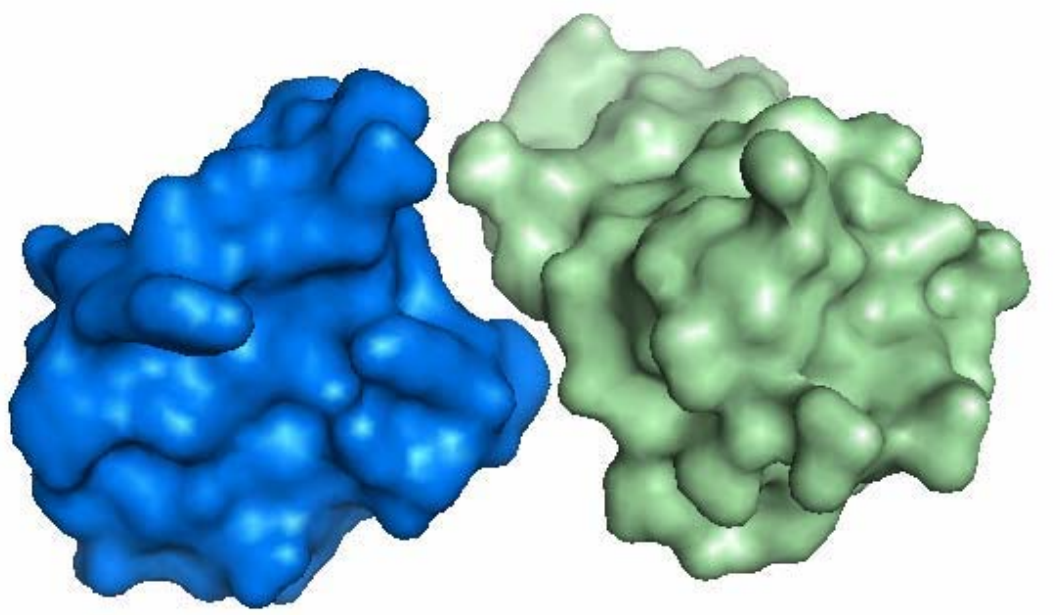

Figure 3·1: Visualization of two proteins docking [SH09c]

which a pose can be evaluated. Even with very fast docking and energy evaluation, modeling the full flexibility of the molecules is prohibitively slow [KDFB04]. For this reason, PIPER focuses on performing rigid docking as a modeling simplification that performs well for the majority of cases [KBCV06]. The evaluation stage of docking generates multiple candidate poses based on the various energy functions, and a scoring and filtering stage is then used to pick the best configurations for each pose.

During the rest of this chapter, the details of molecular docking will be discussed as it is relevant to the PIPER molecular docking program.

3.3 PIPER

PIPER is a rigid protein-protein docking program that uses a grid based representation for the molecules. In a grid based representation, the surface of the molecule is represented as points on a 3-D grid, and each point in the grid represents whether the protein surface lies inside, outside, or on the surface of the protein [KDFB04]. The

algorithm is exhaustive, evaluating every possible pose within each of the rotational positions that are considered [KBCV06]. Because PIPER operates on both protein-ligand and protein-protein interactions, the size of the grids and energy terms can tend to be very large, and it is for this reason that GPU accelerated PIPER was created, as this calculation is heavily time consuming [SH09c]. Figure 3·2 below shows examples of poses that may be evaluated by PIPER, where the proteins are moved in an attempt to try and fit within each other. The success of this fit is evaluated via the energy evaluation functions [KBCV06]

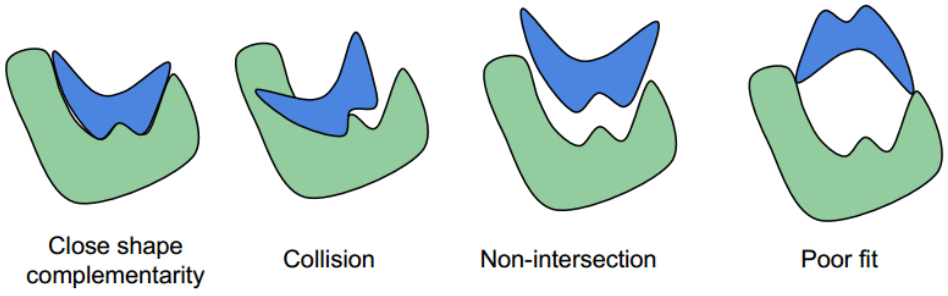

Figure 3·2: Examples of protein poses [SH09c]

For typical rigid docking, including the docking performed in PIPER, the process begins by mapping the two molecules onto individual 3D grids. To evaluate the poses of the grids, the receptor grid is fixed in position, while the ligand grid is then moved around it to evaluate the position. Grid size can typically be of size $N = 128$ in each dimension, and the total number of angles can be around 10,000, thus generating up to 10^{10} relative positions to be evaluated per single molecule pairs [SH09c]. The evaluation phase operates as a series of loops, where the outer loop consists of the rotations, and the inner loop explores the entire 3-axis translational space using a 3D correlation. This requires $O(N^6)$ operations, and is thus incredibly computationally intensive for typical molecule sizes. PIPER, along with many other rigid docking programs, uses a 3D FFT to reduce the complexity of each 3D correlation to $O(N^3 logN)$

for all of the energy contributions [KKSE$^+$92; GJS97; CW03; KBCV06].

The primary advancement that PIPER presented was the use of desolvation energies along with typical shape and electrostatics terms in the evaluation function, thus minimizing the number of candidates needing detailed scoring in the docking phase [KBCV06]. The improved sensitivity created by these docking terms has ensured that as of recently, PIPER within the ClusPro server is still the top performing molecular docking program in terms of accuracy, comparable even to human predictors [LW13]. The desolvation terms are generated as pairwise potential terms. These pairwise potential terms represent the interaction of atoms on the interacting molecules, and the terms are empirically determined prior to runtime [KBCV06]. For K atom types, there is a $K \times K$ interaction matrix, and this may significantly increase run time. However, the PIPER algorithm proposed that by using eigenvalue-eigenvector decomposition, the number of terms needed can be limited to the P largest eigenvalues, where $2 \leq P \leq 4$, limiting the added number of Fourier transforms to 2 to 4 forward transforms and one reverse. When the first version of GPU Piper was designed, it was under the assumption that up to 18 of these desolvation terms may still be used [SH09c]. However, since then, it has come to be known that in practice, only 4 terms are typically ever used, and this is a key optimization in the current work.

In PIPER, there are typically 4 non pairwise terms used for evaluation pertaining to shape and electrostatic behaviors, along with the pairwise desolvation terms [KBCV06]. For each rotation, the exhaustive search of 6D space is done using Fourier transforms for each of the pairwise and non pairwise terms, and the PIPER energy-like scoring function is computed to evaluated the goodness of fit between the molecules. This goodness of fit is expressed as the sum of P correlation functions for all possible translations α, β, γ of the rotated ligand relative to the receptor,

$$E(\alpha, \beta, \gamma) = \sum_{p} \sum_{i,j,k} R_p(i, j, k) L_p(i + \alpha, j + \beta, k + \gamma) \tag{3.1}$$

where $R_p(i, j, k)$ and $L_p(i + \alpha, j + \beta, k + \gamma)$ are the components of the correlation function defined on the receptor and the ligand grids, respectively.

Thus, for each rotation, the ligand energy function is evaluated on the grid and repeated FFT correlations are performed for each of the different energy function. Filtering is performed by scoring each pose within a rotation and subsequently selecting the top scoring poses. A top scoring pose is a pose which minimizes total energy based off of both the pairwise and non pairwise energy terms.

The scoring functions in piper are based on three criteria: shape complentarity, electrostatic energy, and desolvation energy via pairwise potentials [KBCV06]. Each is expressed as a 3D correlation sum evaluated via FFTs, and the total energy function is expressed as a weighted sum:

$$E = E_{shape} + w_2 E_{elec} + w_3 E_{pair} \tag{3.2}$$

Shape complentarity refers to how well the proteins fit geometrically, as was see in figure 3·2 earlier, and it is computed as a weighted sum of attractive and repulsive van der Waals terms:

$$E_{shape} = E_{attr} + w_1 E_{rep} \tag{3.3}$$

The electrostatic interaction is represented in terms of Generalized Born equations that are simplified for PIPER [CW03]

The energy function weights are provided at runtime, and are considered the coefficients for the evaluation of the scoring function. For a molecular docking run of PIPER, multiple sets of coefficients may be provided so that PIPER returns the top

scores for each set of coefficients. Along with returning top scores per coefficients, PIPER is capable of returning the top N scores for each pose and coefficient set as well.

3.4 PIPER Program Flow

The PIPER program has initial stages used to read in receptor and ligand information, compute the FFT size, create the receptor grids, compute the receptor FFT for all energy terms, and create the ligand grids. After this setup, PIPER begins performing the various rotations, and within each rotation, the following steps occur:

1. Rotation of the ligand grid
2. Assignment of the 3D energy grids for all terms
3. FFT correlation of the receptor and the ligand grids
4. Accumulation of the desolvation terms to obtain pairwise potential score
5. Weighted score computation of different energy functions
6. Scoring and filtering for the current rotation

The work distribution for a single rotation in the CPU based version of PIPER, as is seen in Figure 3·3, is primarily dominated by the computation time of the correlation. Filtering and grid assignment take only a very small percentage of the total rotation time.

The GPU version of PIPER follows an identical program flow for the computation; the fundamental algorithm is not changed. The only change is the migration of various computations, particularly the correlation and filtering steps, to the GPU for acceleration. Details and computational analysis of these steps will be discussed in the following chapter.

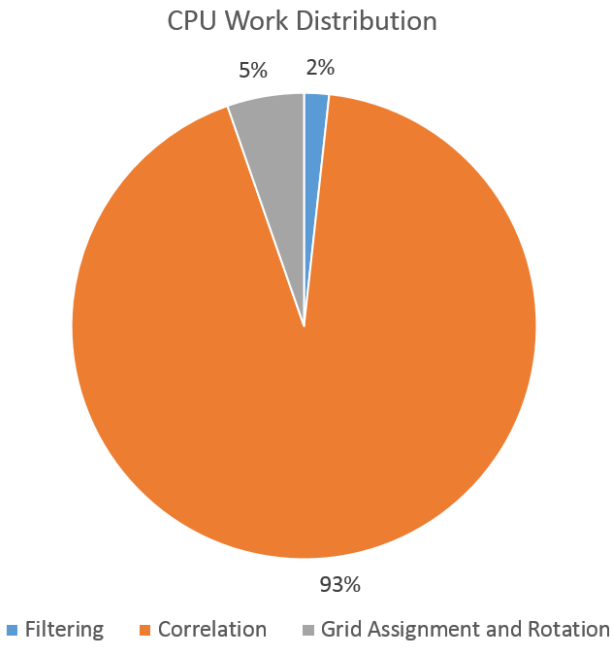

Figure 3·3: CPU work distribution during a single rotation [SH09c]

3.5 Recent GPU Work

Since the release of the original GPU version of PIPER in 2009, there have been several developments for GPU enhanced versions of molecular docking. In particular, MEGADOCK [SIS+13] in 2013 and Hex in 2010 [RV10] have displayed favorable speedups vs their respective CPU based versions of molecular rigid docking. Beyond rigid docking, Autodock [PF11] and Moldock [SCTP13] have also attained speedup in the realm of flexible molecular docking. The improvements these programs have gained are also based on varying the fundamental molecular docking algorithms used in their respective programs.

As was seen in the recent CAPRI test, PIPER still is best in class for automatic prediction of interaction [LW13]. Along with this, the CPU version of PIPER is a mature codebase that is now optimized using methods such as parallelization via Message Passing Interface (MPI) [Tea] and FFT acceleration using the Intel Math

Kernel Library (MKL) [Int]. Thus, to obtain even better speedup for this state of the art version of PIPER, it is important for the GPU version of PIPER to be further optimized.

Chapter 4

Optimizations

4.1 GPU PIPER

When the original version of GPU accelerated PIPER was created, there were two guidelines which were followed for attaining the best performance gains. The first was to use a pre-analysis of the PIPER program in order to find the most time consuming and optimal points for acceleration. The second guideline was in trying to find the program segments which innately showed the most propensity for parallelization.

What were found to be the most computationally time consuming segments were all of the 3D FFT computations, as is seen in Figure 3·3. The main optimization developed in the previous work was migrating all of these computations to the GPU via the CUFFT library [SH09c; Nvib]. Along with the FFT computation, the filtering and scoring stages were found to be highly parallel, and they were readily moved to the GPU[SH09c]. The grid computation as well as grid rotation were left on the CPU in this original version for each rotation. Figure 4·1 displays the program flow for the GPU version Piper with a few modifications relevant to the current work. All light green boxes are program segments which remain on the CPU, all dark green boxes are code segments that were moved to the GPU, and the light blue box is a code segment that was on the CPU in the original version of GPU accelerated PIPER [SH09c], and in the current work is being moved to the GPU.

Since the FFT computation is completely dependent on the CUFFT library provided by Nvidia, the points for optimization in this current work revolved around

both the data transfer between CPU and GPU, the filtering and scoring function, and eliminating any idle time on the GPU.

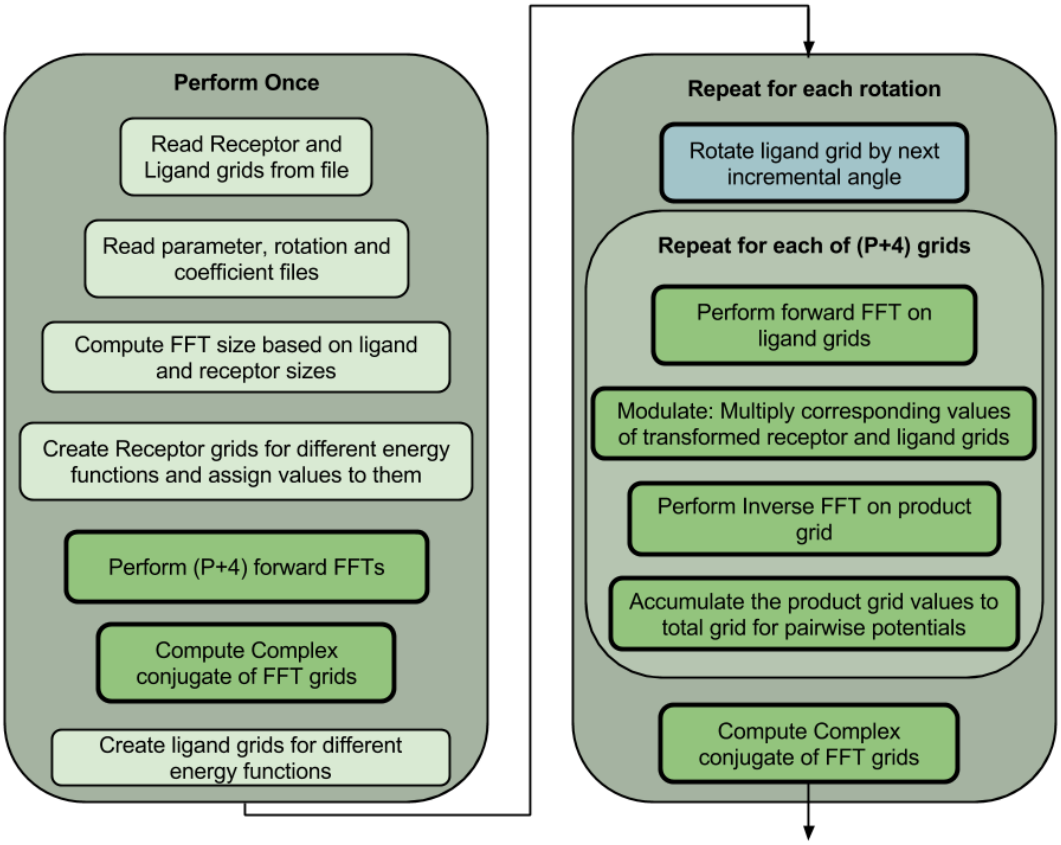

Figure 4·1: The flow of the PIPER program. Light green boxes are on the CPU, dark green boxes are on the GPU, and light blue boxes are being moved to the GPU in the current work.

In order to find the key points for optimization, an initial analysis of the original GPU PIPER performance was attained using Nvidias profiling tool, nvprof [Nvia], on the K20c card. This profiling tool analyzes the kernel runtime for PIPER and also displays the idle time of the card during execution, providing insight on the worst performing kernels. Figure 4·2 shows the results of the analysis via a pie chart. The chart represents the total time of a single rotation, and each slice is the proportion of time spent on a certain stage of rotation.

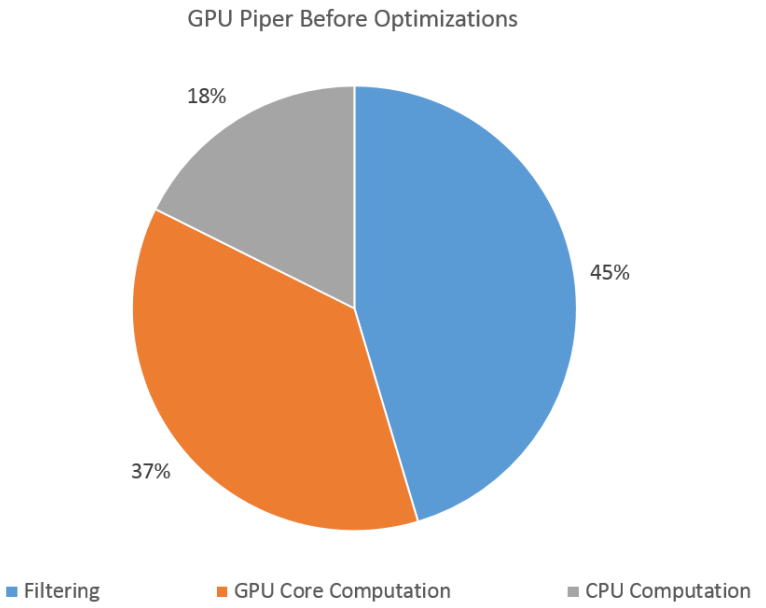

Figure 4·2: Proportion of rotation step spent on various stages of the computation. The FFT and modulation steps are grouped as the GPU core computations. Filtering/Scoring and the CPU calculations take the majority of the time.

The chart evidently shows a difference in work distribution, compared to the work distribution display in Figure 3·3 for the CPU. On the GPU, the filtering step takes a large portion of the time per rotation. The leftover CPU computations, consisting of the rotation step and the grid assignment steps for each of the non-pairwise and desolvation grids, takes a non-trivial portion of the rotation as well. This is larger than the proportion on the CPU simply because the correlation function on the GPU is significantly faster, so the proportion of time spent on CPU work is thus larger. What is even more critical is that because the grid assignment is on the CPU, data must be transferred to the GPU on each iteration.

4.2 Odd Size Grid Volume

Before implementing any new performance optimizations, an important fix was necessary for GPU PIPER to ensure that it adequately performs molecular docking for all suitable inputs. In the original version, the ligand grids for all energy terms were stored in large arrays in GPU Global memory. These arrays were allocated directly after each other in memory, and for the majority of protein complexes, this was suitable as the total grid volume became an even value, resulting in aligned memory addresses. However, in a few cases, the grid volume became oddly valued, and the original version of GPU PIPER was unable to handle this. This error arose because for the CUFFT library to operate properly, the start of the memory location allocated to the grids must be properly aligned in memory to a 16 byte boundary [Nvib], and this immediately failed for some of the grids allocated on the GPU.

Thus to alleviate the issue, the memory allocation for ligand grids was performed using pitched allocation on the GPU. This forces each ligand grid array to be aligned in memory properly, at the expense of a few extra bytes being allocated for memory padding. With the 6GB global memory size available on the C2075 cards and the

5GB global memory for the K20c, a few extra bytes to allow for proper alignment is of no consequence. All results for the original optimized version of GPU PIPER have this important fix included, as it is fundamental to proper execution of the algorithm.

4.3 Filtering and Scoring Stage Optimizations

The original version of GPU piper used a scoring and filtering kernel that attempted to find the top scores for a rotation for all provided scoring coefficients at the same time. This method worked the best when the max number of coefficient sets were provided at the same time, typically up to 8. In this case, the top score for each set of coefficients was calculated by a single SM on the GPU card. With 8 coefficient sets, 8 SMs were simultaneously occupied, and theoretically, the GPU would be fully utilized.

In practice, this method vastly underutilized the potential of the GPU by minimizing the work distribution and forcing individual SMs to work an inordinately large amount relative to their peers. Furthermore, for many molecular docking runs used in practice, only a single coefficient set is used, leaving only a single SM to be utilized. Thus, optimizing filtering changed the goal of the GPU filtering kernel from attempting to find the top score for all coefficient sets simultaneously, to quickly finding the top score for a single coefficient set and repeating the process for each coefficient set.

The new form of filtering and scoring on the GPU takes the form of two kernels which are then repeated for every set of coefficients, as well as being repeated for the number of top scores desired by the user for each coefficient set. The two kernels partition the work into two stages so that the shared memory on the GPU is utilized for fast memory access, and so that work is distributed across all SMs.

In the first kernel, the output data, which is the size of the molecular grid volume, or equivalently the FFT size, is partitioned between all available SMs. This is done

by launching the kernel with a sufficiently large number of blocks such that every SM has a suitable quantity of work. For reasons that will be explained later, the number of blocks must be a power of two. We found that a suitable quantity of blocks for the K20c card was 32. For each block, each thread in the block accesses a subset of the output molecular grid data, calculates the score using the energy equations described in Chapter 3, finds the best score within the subset, and finally places this result in shared memory for the block. The subset accessed by the thread is not contiguous, but rather strided by the total number of threads launched on the GPU, in order to properly coalesce GPU memory and improve performance [Nvia]. Once all threads in the block find their partial best scores and write it to memory, they are synchronized, and a single thread from each block finds the top score from the partial scores in shared memory, and writes this block partial score to global memory, letting the kernel finish. This first kernel utilizes shared memory as best possible to minimize the time needed to find the best score out of the scores made available to this block. Figure 4·3 is a visualization of this work distribution across the SMs on a GPU.

The second kernel differs greatly from the first kernel in that only a single block is used, and the number of threads assigned to this block is equivalent to the number of blocks used in stage 1. The reason for the power of two requirement from the prior stage lies in that in this stage, a classic log step reduction of the partial scores in memory is used. At each step, the number of active threads is equal to half the number of partial scores left. Each thread compares between two scores, and writes the best to the lower half of memory. After every comparison, a synchronization is performed to ensure all threads are always operating on the same step, and thus operating on the consistently updated memory. Eventually this results in only one top score. This best score is then marked in a separate array in global memory,

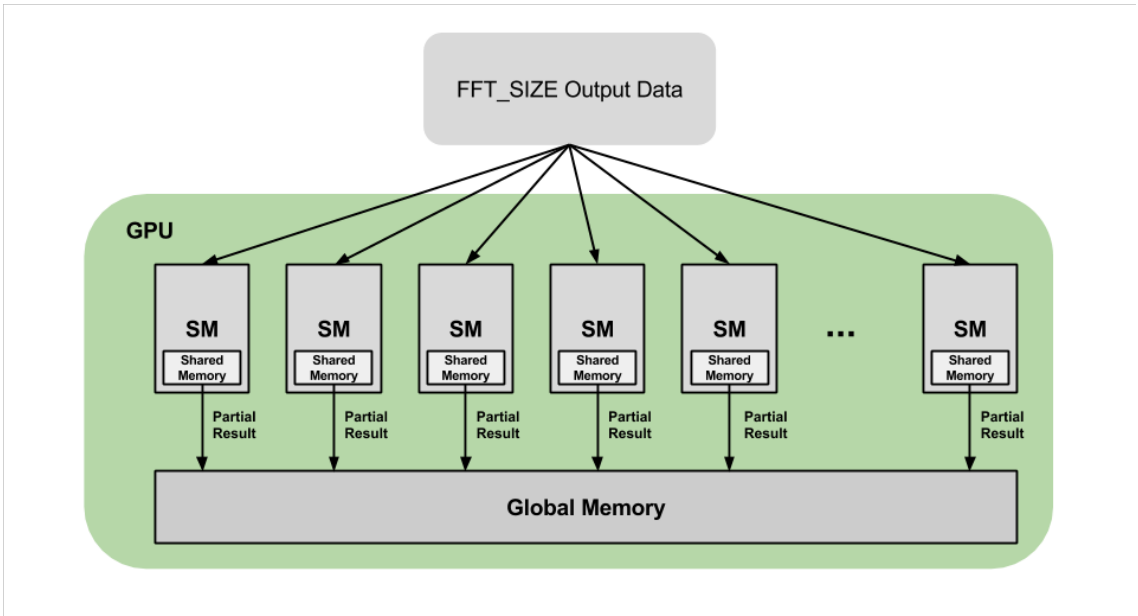

Figure 4·3: Filtering Stage 1 kernel. Each SM may write multiple partial scores to global memory based on how many blocks were assigned to that SM.

indicating that it is a top score. The necessity for this arises in that PIPER may require multiple top scores per coefficient set, so previously chosen top scores must be marked as to not be chosen again in later calls to the function. Along with marking the top score, positions in the molecular grid around this best score are also marked as being "top scores". This ensures that when computing multiple top scores, the results are not all trapped in the same local minima for the molecular grid, a key feature in filtering and scoring for PIPER [KBCV06]. Figure 4·4 displays this second stage process.

While this filtering stage no longer optimizes for all coefficient sets, we find that the GPU is better off fully utilized and repeating the computation for most efficiency. By doing this, the GPU can be kept fully utilized regardless of how many coefficient sets are provided at runtime. This new version performs dramatically better than the original filtering and scoring kernel, and will be discussed in more detail in the

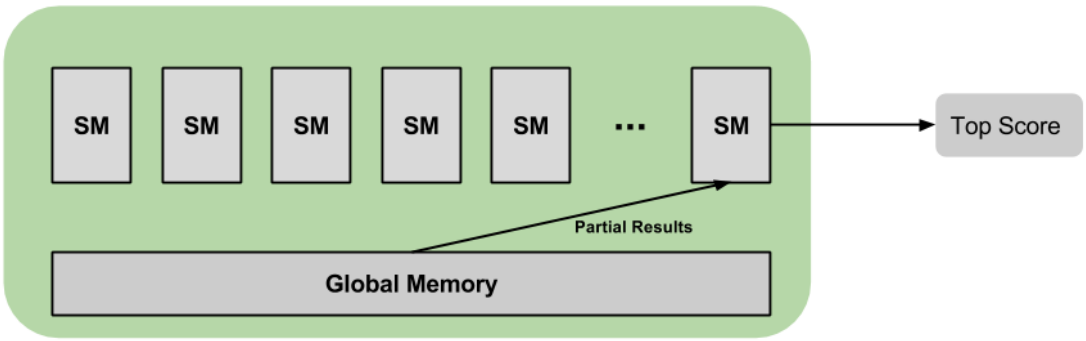

Figure 4·4: Filtering Stage 2 kernel. A single SM performs the work of reducing the partial scores in global memory.

following chapter.

4.4 GPU Idle Time Optimizations

The second major optimization applied was the elimination of the idle GPU time present when the rotation and grid assignment steps are performed on the CPU. In the original version, CUDA streams were used in an attempt to overlap GPU work with grid assignment [SH09c]. However, due to the result from filtering and scoring immediately being assigned to memory and moved around on the CPU, the GPU kernels were forced to execute filtering and synchronize, before the CPU could prepare for the next rotation.

Along with this issue, the grid data, since it was assigned on the CPU, must be copied to the GPU on each rotation. For the C2075 card, this data transfer overlapped with the CUFFT calls for each grid, so that the effects of this data transfer latency were hidden. However, on the K20c, the data transfer from CPU to GPU was longer than the CUFFT execution time, either due to a shortened FFT and modulation execution time or reduced bandwidth for the desired transfer size, making the GPU

core computations memory bound rather than compute bound.

This is visible in the sample nvidia visual profiler output shown in Figure 4·5. In this figure, the top bar is the memory latency while the smaller boxes below are the kernels for the FFT and modulation stages of correlation. As is clear from this figure, the memory transfer time for the molecular grid is longer than the entire correlation computation, forcing the GPU to remain idle during the data transfer before starting the next correlation step.

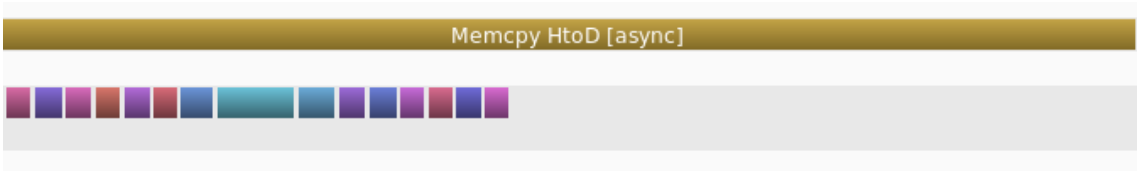

Figure 4·5: NVProf output memory latency relative to correlation compute time on the K20c GPU. A longer bar indicates a longer execution time.

To solve both of these issues, our solution was to move all of the grid assignment arrays permanently to the GPU, and perform grid assignment and ligand rotation directly on the GPU with no need for transfer between the host system memory and GPU memory. For this to be feasible, the GPU must contain enough global memory for all of the input and output data. As was discussed earlier, the number of energy grids used is $P + 4$, where P is the quantity of pairwise potential terms. In the original work, it was assumed that up to 18 of these terms were used in practice [SH09c]. However, since then, it has become known that for typical ClusPro operation, accurate results are obtained using only 4 pairwise terms. Thus, it is only required that enough memory for 8 grid arrays is available across the entire docking analysis for the vast majority of docking cases. Figure 4·6 is a chart showing the distribution of the protein complex grid volumes provided by the ZLab Protein docking benchmark [HVJW10]. As can be seen, the largest grid volume contains

around 7 million elements, while the smallest are below 1 million elements. When calculating the memory requirement for the GPU, the largest array for all portions of the computation comfortably fit within the 5GB of global memory on the K20c GPU. Since this benchmark is considered a representative sample of protein complex sizes, we assume that for the vast majority of computations, the GPU memory on the K20c, and similarly the C2075, will suffice. Both the K20c and C2075 GPU also have sufficient memory for greater than 4 pairwise potential terms, but the quantity of these terms that can fit depends on the size of the protein complex.

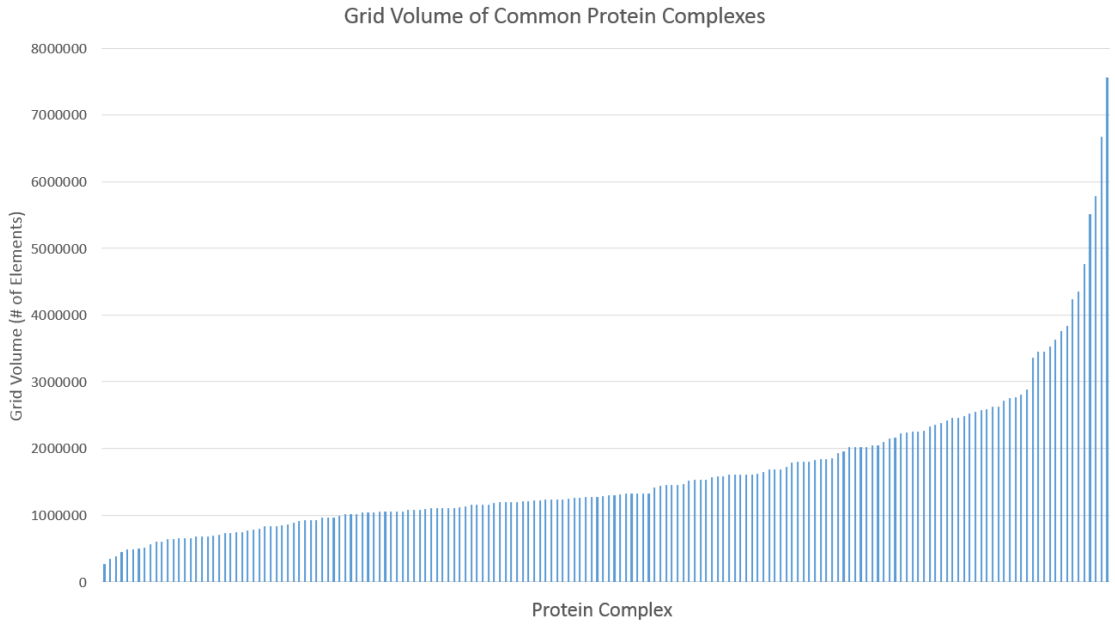

Figure 4·6: Grid volume for the protein complexes in the Zlab Protein Docking benchmark. Each bar represents an individual protein complex and its grid volume.

With all ligand grids of suitable size fitting in GPU memory, the ligand grids are immediately allocated on the GPU before any rotation steps. The functions for grid assignment for all energy terms, as well as the data for rotation and angle data, were also moved to the GPU.

For grid assignment, the functions mapped easily to the parallel nature of GPU

kernels. Every grid function iterates over all of the atoms in the protein, and assigns values to the appropriate grid location based on predetermined position and state information for every type of energy and desolvation term. Thus, on the GPU, each atom is assigned to a single thread, and multiple blocks of threads are launched. However, multiple atoms may affect the same element in the grid, and a race condition may occur if multiple threads attempt to update the same memory location. In order to alleviate this issue, we took advantage of a new feature introduced in the Fermi architecture: global memory floating point atomic operations [Nvi09]. In the Tesla architecture, only integer atomic operations were possible, but with floating point atomic operations, the grid assignment kernels were greatly simplified with the use of floating point atomic add operations for grid assignment.

Typically when using atomic operations on the GPU, it is under the implicit understanding that serializing the operations will result in significant performance degradation. However, in the case of grid assignment, the amount of threads which may interact with each other is incredibly small for each memory location. Thus, we claim that moving the grid assignment to the GPU will be a significant improvement over grid assignment on the GPU, even with the requirement for atomic operations.

With the grid data, grid assignment, and rotation function now on the GPU, the data transfers from CPU to GPU were entirely removed, and every portion of the rotation step now lies on the GPU. The results for all of these optimizations will be discussed in the following chapter.

Chapter 5

Results

5.1 Target Hardware

For attaining the timing results for the various runs of optimized CPU PIPER vs the newly accelerated GPU PIPER, we used a standard set of comparable hardware. The hardware we used was available on both the Boston University engineering grid and the Boston University Shared Computing Cluster. For all GPU experimental runs, PIPER was compiled using CUDA version 5.5 and g++ 4.4.7 . The tests were run on server nodes on the Boston University engineering grid with Intel Xeon E5530 processors; each server node contained either a single Nvidia C2075 Fermi class card or a single Nvida K20c Kepler class card. For MPI based CPU runs, PIPER was compiled using gcc 4.4.7 and with OpenMPI version 1.6.4 . The CPU version of PIPER used the Intel MKL FFT library, version 11.1 . The tests were run on server nodes on the shared computing cluster with Intel Xeon E5-2680 sandy bridge era cards, and PIPER was allowed full use of one of these 8 core chips. An overview of this hardware is shown in Table 5.1, with a full description of process size, release date, and frequency across the devices.

5.2 Protein Complexes

The protein complex used during benchmarking was the 1AHW complex from the ZLab Protein docking benchmark [HVJW10]. As was discussed in the previous chap-

Table 5.1: Comparison of target hardware.

Experimental Hardware								
Technology	Make	Model	Parallelism	Part #	Process	Frequency	Code	Release
CPU	Intel	Sandy Bridge	8 Core	E5-2680	32 nm	2.7 GHz	MPI+MKL	2012/Q1
GPU	Nvidia	Fermi	448 SPs	Tesla C2075	40 nm	1.15 GHz	CUDA	2011/Q3
GPU	Nvidia	Kepler	2496 SPs	Tesla K20c	28 nm	0.73 GHz	CUDA	2012/Q4

ter and shown in Figure 4·6, the distribution of protein complex grid volumes, or equivalently FFT size, is vast. The 1AHW protein complex is chosen because it is larger than the majority of protein complexes at approximately 3.5 million elements, but half the size of the largest protein complex. We believe this protein complex provides representative results for the analysis, especially since the majority of the results we present rely on the runtime of the Nvidia CUFFT library, of which scalability information is readily available. PIPER was also configured to run with a single coefficient set and a single top score per run in order to isolate the improvements per program segment. PIPER molecular docking runs iterate over 70,000 rotations, and the results are an average over all of the rotations.

5.3 Optimization Results

The results from the optimization are expressed in stages: the original version of GPU Piper, a version of PIPER with the filtering optimization, and finally the version of PIPER with both the filtering and CPU idle time optimizations. The benchmarking results are expressed for a version of PIPER using MPI and MKL on the 8 core Intel machine. Along with this, the results for the MPI version of PIPER were derived by dividing the average rotation time of all individual MPI tasks by the number of cores. This is because the MPI version of piper splits up the independent rotations across multiple MPI tasks. Thus, for the MPI version of PIPER, the rotation time

is actually the effective rotation time experienced by splitting the work amongst the various MPI tasks.

In figure 5·1, the results for the filtering optimization are presented. Prior to optimization, GPU PIPER filtering took a large bulk of the computation. Once the new filtering kernels were introduced, the filtering kernel improved from a latency of 108 ms to only 1 ms, a multiple order of magnitude improvement. After applying the second optimization, leftover CPU code that is not hidden by the GPU-CPU overlap requires only 0.8 ms, and grid assignment and rotation when done on the GPU only requires 1.2 ms. This is a 21x impovement over the original 42 ms required for grid assignment and rotation in the unoptimized GPU PIPER and CPU PIPER.

An important observation to note is that when all the computations are moved to the GPU, there is no longer a need for data transfer overlap with CUFFT. Prior to the CPU idle time observations, the GPU correlation time is actually the max of either the memory transfer latency time, or the correlation time. However, as is shown in Figure 4·5, the memory transfer latency is longer for the Kepler GPU, so the correlation time appeared longer. However, with data permanently moved to the GPU, repeated memory transfers are no longer needed. Thus, the GPU core computation time improves as well since the application is no longer memory bound.

Figure 5·2 is a comparison of the improvements the GPU based PIPER displays over the CPU for a single rotation time. The results are normalized to the CPU rotation time. Due to the original inefficient filtering method, the original version of GPU PIPER was actually slower than a modern MPI version of PIPER for the same protein complex on modern hardware. Along with this, due to the inefficiencies caused by the bandwidth bound FFT computations, even after the filtering optimization, the MPI based piper is faster. This limitation indicates that the original optimizations were limited to the previous GPU and CPU generation and were not scalable to future

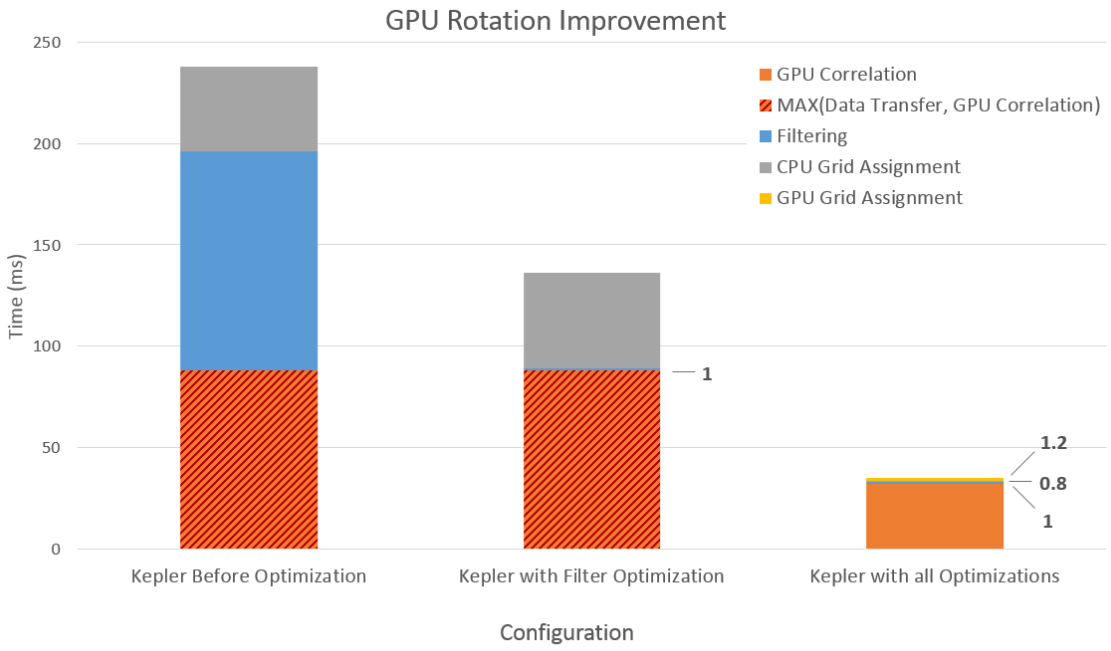

Figure 5·1: GPU runtime improvement with bar segments separating the different computations. Prior to the CPU idle time optimizations, the correlation time was actually hidden by the memory transfer time, as it was larger.

hardware. However, both improving the filtering and scoring stages allow the GPU version of PIPER to be approximately 3.3x faster than CPU PIPER, and 6.6x faster than the unoptimized version of GPU accelerated PIPER.

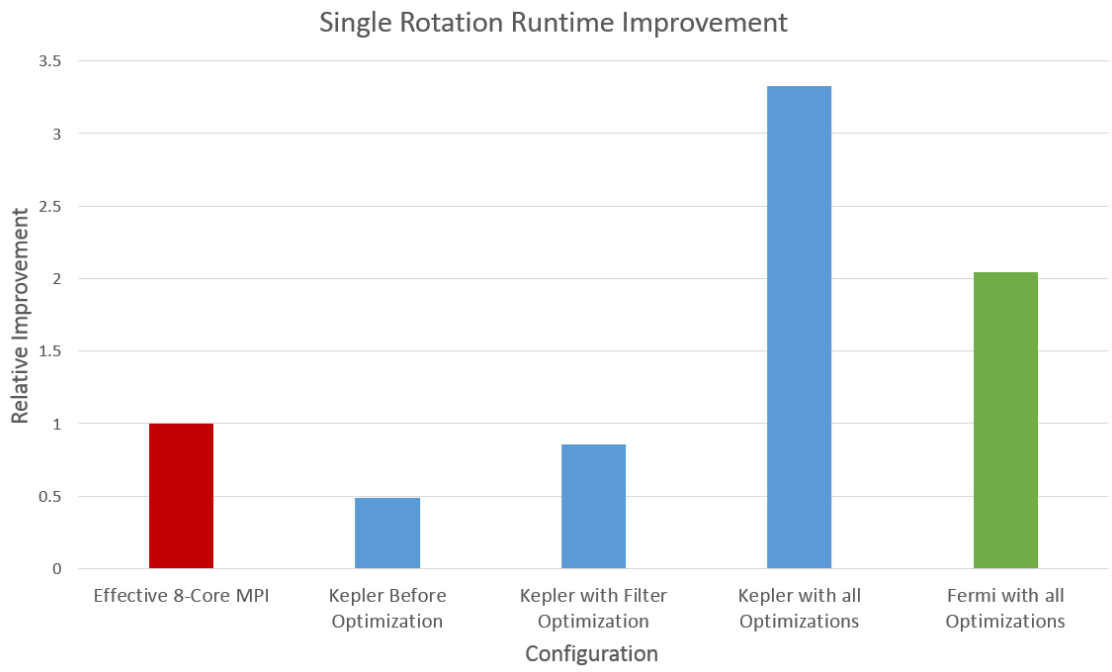

Figure 5·2: Relative improvement for various configurations. The red bar is CPU PIPER, blue bars on Kepler HW, and the green bar is Fermi HW.

The results obtained from all optimizations for the C2075 card are also favorable, attaining 2X speedup over the CPU based PIPER. Thus, for a chip to chip comparison, the GPU version has 2x improvement using the Fermi class card and 3.3x improvement using the Kepler class card over the mature and optimized CPU version of PIPER running on SandyBridge Intel hardware.

To further reflect the improvements in the GPU rotation runtime, Figure 5·3 below is a pie chart similar to the one shown in Figure 4·2. It reflects the respective proportions of the rotation spent on individual tasks. Compared to before, only 5% of the computation is spent on filtering and CPU based computations. 91%

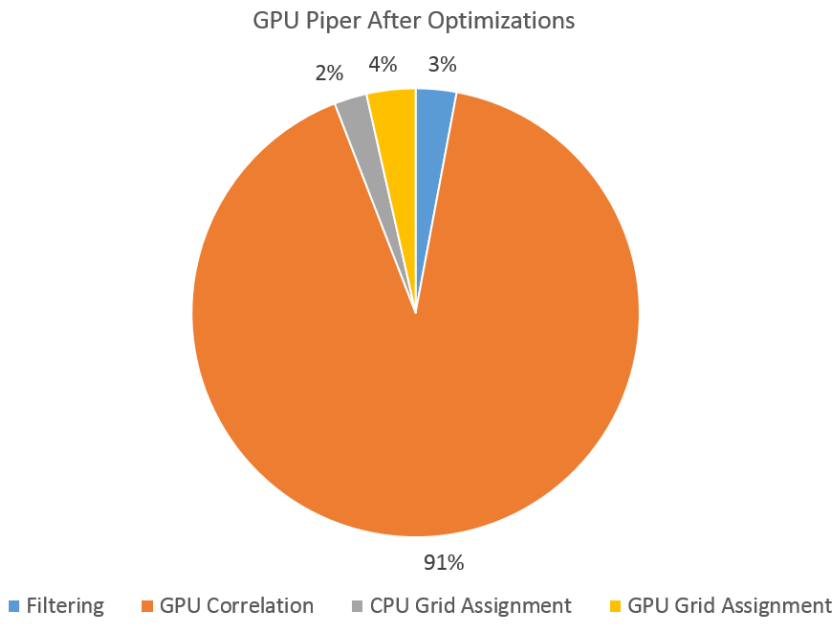

Figure 5·3: Portion of rotation step spent on various parts of the computation after optimizations.

of the computation is now left for core FFT and modulation GPU computations. The remainder is now the GPU based grid assignment operations. This new chart demonstrates how our new version much more closely matches the work distribution shown in Figure 3·3 for the CPU, but performs significantly faster.

Figure 5·3 brings up an important aspect of the result, namely that reducing the time spent in rotation to primarily FFT computations limits the level of optimizations possible to the runtime of the CUFFT library. In an ideal scenario, the computations of the inner loop would be reduced to only FFT computations for either the CPU or the GPU versions. In this case, the performance increase would be tied to the performance of the CUFFT and Intel MKL libraries respectively. Figure 5·4 directly addresses this scenario by comparing the average CUFFT and 8 core Intel MKL FFT run times when calculating the 3D FFT of length 64 in each dimension and 128 in each dimension, to the improvements shown for molecular docking on Intel and

Nvidia hardware. These sizes are comparable to the grid volume sizes in figure 4·6 and are illustrative of the FFT level improvements. For both of the FFT sizes, the max improvement demonstrated by the GPU are 2x and 3.3x respectively.

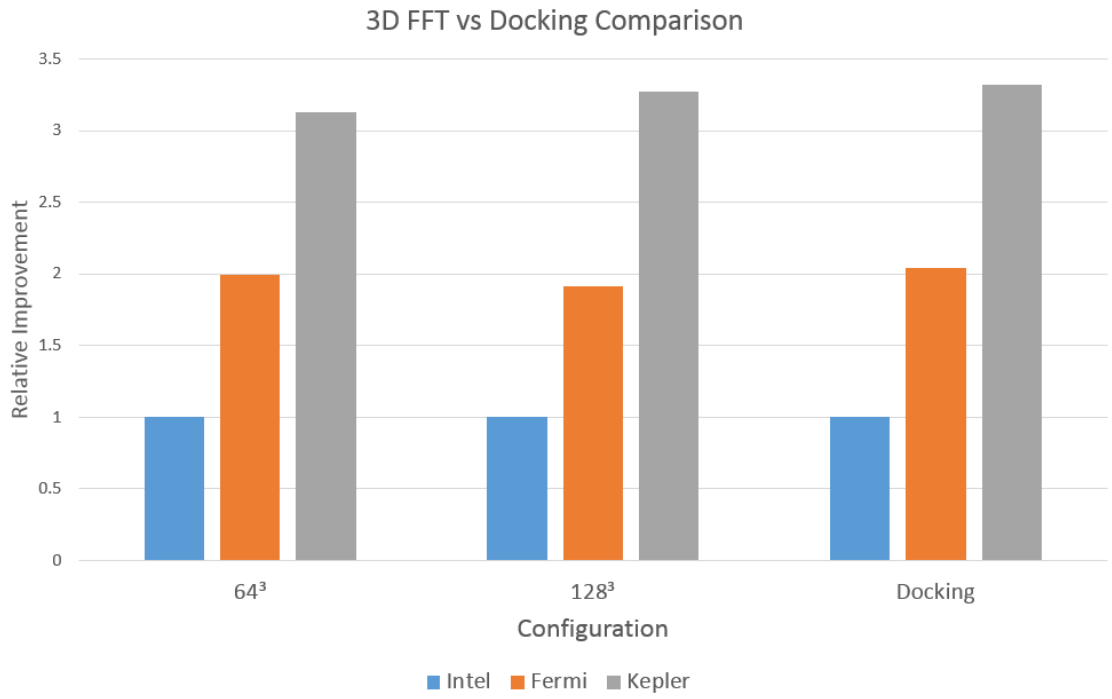

Figure 5·4: Comparison of the runtime for 3D FFTs with varying dimensions to the improvement in runtime demonstrated in PIPER docking for Intel, Fermi, and Kepler HW configurations

According to the CUFFT documentation, for non power of two dimensional lengths, the performance improvements are not as pronounced [Nvib]. Thus, for the improvements presented in this work, a 3.3x improvement on a per chip basis for kepler hardware can be thought of as near the limit of improvement if all other computations were minimized relative to the FFT computation.

These improvements match incredibly closely to the improvements we have demonstrated with our work for the C2075 and K20c GPUs, with the small differences attributed to the other nonzero computations required in molecular docking. Our

current rotational runtime is now closely tied to the performance of the CUFFT. Because of this, our new version of PIPER will stay faster than the CPU version of PIPER as long as the Nvidia CUFFT library remains faster than the MKL library for comparable chip architectures. However, this is now representative of Amdahl's law on the scale of the GPU. Now that all other algorithmic points have been optimized, any further performance improvements are directly limited by the FFT. Our presented optimizations approach this limit of possible speedup for the PIPER molecular docking algorithm.

In work by Ritchie [RV10], they discuss that a potential reason for the limited runtime improvement for 3D FFTs on GPUs lies in that 3D FFTs require significant global memory accesses, and are thus limited by the memory latency of global memory. Unfortunately, further optimizing the FFT on the GPU is not in our control without a custom implementation. Any further notable improvements in rotation runtime will no longer be gained by turning to the GPU architecture as an accelerator, but by changing the fundamental PIPER algorithm instead. As is the trend with many GPU algorithms, memory latency is a fundamental limit to runtime improvement.

The results we present demonstrate that the GPU version of PIPER did indeed require optimizations in order to ensure that it always outperforms its CPU counterpart. Importantly, we also show that by limiting the GPU computation to the FFT, we ensure that the GPU version remains faster than the CPU version as long as the CUFFT library is faster than Intel MKL for equivalent inputs.

5.4 Validation

Along with the performance results provided in this section, a primary focus of the work was to ensure that the accuracy of the results, the output of molecular docking,

remained identical to that of the original GPU version of PIPER [SH09c]. All of the experimental results presented here only accelerate the computation, and the docking output from the computation using the optimized version are identical to the results attained from analyzing the same protein complex using the original version of GPU PIPER.

For validation, we provided the output of the PIPER molecular docking to the clustering phase of the ClusPro program, which groups the near native poses for molecular complexes. Table 5.2 shows the output of clustering for both results attained from the CPU version of PIPER, and the GPU version presented in the current work. The complexes tested were the Antigen/Antibody (AA) complexes from the ZLab docking benchmark [HVJW10]. For each complex, the number of near native poses within 10 Angstrom of the interface root mean square deviation (RMSD) value in the top 1000 for all AA complexes is shown. The rank of the best pose for each complex, along with the interface RMSD for this best pose, is also shown. Any empty table entries indicate that PIPER returned no top poses for that complex, which occasionally occurs. The best pose output for the CPU and GPU are identical, while the top 1000 selected for the complexes are nearly identical. The small differences are attributed to small floating point differences between the GPU and CPU. These results coincide with the validation performed during the original development of GPU accelerated PIPER, and demonstrate that our newly accelerated version retains all accuracy.

Table 5.2: Comparison of GPU and CPU outputs after clustering phase. The number within 10 Angstrom are the best selected poses for the complex. The near native rank and interface RMSD are for the best pose selected for each complex.

CPU - GPU Validation Antigen/Antibody Complexes						
	CPU			GPU		
Protein Complex	# within 10A	Near Native Rank	Interface RMSD	# within 10A	Near Native Rank	Interface RMSD
1AHW	64	4	4.622	64	4	4.622
1BVK	30	9	7.288	30	9	7.288
1DQJ	36	1	9.84	36	1	9.84
1E6J	207	1	9.365	207	1	9.365
1JPS	84	3	4.898	84	3	4.898
1MLC	35			34		
1VFB	142	2	5.287	142	2	5.287
1WEJ	118	4	2.781	118	4	2.781
2FD6	13			13		
2I25	0			0		
2VIS	3			3		
1BJ1	72	4	2.218	72	4	2.218
1FSK	197	1	1.192	197	1	1.192
1I9R	111	1	6.482	112	1	6.482
1IQD	56	4	1.95	56	4	1.95
1K4C	9			9		
1KXQ	227	1	4.304	227	1	4.304
1NCA	27	11	0.852	27	11	0.852
1NSN	17	20	8.33	17	20	8.33
1QFW	113	2	2.626	113	2	2.626
2QFW	80	5	5.735	80	5	5.735
2JEL	61	5	1.79	61	5	1.79
1BGX	0			0		
1E4K	0			0		
2HMI	0			0		

Chapter 6

Conclusion

6.1 Optimization Discussion

After performing our optimizations, we have created a new version of PIPER that is 3.3x faster on state of the art Nvidia GPUs over an MPI and MKL based implementation running on an 8 core CPU. This was the immediate goal of the current work, and we were successful in attaining the speedup. Importantly, we attained this speedup without affecting the output from the molecular docking analysis.

Of note is our improvement of the filtering and scoring step in PIPER. During the initial development of GPU accelerated PIPER, the impact of the filtering and scoring step was minimal relative to the time spent in the FFT. However, as the FFT time improved over multiple generations, the filtering time became substantial, and via Amdahl's law, the speedup was limited by the runtime of the filtering step. Our approach directly addressed this bottleneck, as well as the leftover CPU computations in order to once again make the limiting step the FFT.

Along with the improvement in filtering, the migration of all data to the GPU without data transfers during a rotation allowed the computation to be entirely compute bound. For modern GPU architectures, the unoptimized version of PIPER became bandwidth bound and hindered potential improvements in runtime. Our new approach ensures that the benefits of new architectures and improved runtime of the CUFFT are reflected in an improved rotational runtime.

As our accelerated version of PIPER is now limited by the FFT, we also are able to

observe that our version of PIPER has approached the theoretical maximum speedup possible over an 8-core CPU if the algorithm was reduced to a series of 3D FFTs alone. This limitation is demonstrative that GPU acceleration is still limited by the architecture, as 3D FFTs are still computationally expensive on a GPU. However, because the new version of GPU PIPER utilizes most of the rotation runtime now on the FFT stages, we can ensure that this version will remain faster on a chip to chip basis than equivalent CPU version of PIPER.

Our new version completely offloads all rotation computations to the GPU. This is beneficial for two reasons. First, it enables the performance of the rotation scoring to be completely independent of the quality of the CPU. This may be a cost effective approach for molecular docking, as purchasing multiple GPUs without the worry of owning a high performance CPU cluster may be cheaper. Secondly, it allows the CPU to be free to perform other computationally intensive tasks without hindering the performance of the GPU rotation steps. This provides incredible flexibility for running PIPER in data centers.

GPU accelerated PIPER is now integrated and available for use as part of the ClusPro server at Boston University.

6.2 Further Work

Although we demonstrate that this version of GPU PIPER is close to optimal in terms of acceleration, there are still a few directions further research can progress for further improvements.

- **MPI GPU:** PIPER is able to be parallelized extremely well via MPI because every individual rotation is independent and can thus be exhaustively searched and scored independently. This is the primary reason as to why an MPI based CPU version of PIPER incredibly scalable. However, this same technique can

be used for accelerating the computation on the GPU. An MPI version of GPU PIPER, where different rotations are assigned to different GPUs could provide a highly scalable performance improvement over the current version. This can be particularly useful if PIPER rotations are distributed across multiple, inexpensive consumer grade GPUs, rather than a single, more expensive, computational GPU.

- **Improved PIPER algorithm:** Since the original release of the PIPER algorithm, work has not ceased to continue in developing a faster and more accurate molecular docking algorithm. For this reason, new versions of the PIPER algorithm are being developed that may fundamentally improve the complexity or runtime of the rotation loop. A GPU version of these new versions may provide further speedup.

- **GPU Optimization of Other Molecular Modeling Tools:** During the development of the original PIPER, work was also performed to accelerate Energy Minimization, a separate molecular modeling tool for predicting molecular interaction. This tool is also part of the ClusPro server, and finding new optimizations for this algorithm is also of importance.

Beyond molecular docking, the methods learned in analyzing GPU performance and optimization discovery can be applied to a wide variety of High Performance Computing domains. Many areas of research are now using GPUs for accelerating computations, and sharing optimization techniques from across fields may provide an even larger breadth of performance improvements to keep up with increasing scientific demand.

References

[AB10] Christopher Paul Adams and Van Vu Brantner. Spending on new drug development1. *Health Economics*, 19(2):130–141, 2010.

[CLW03] R. Chen, L. Li, and Z. Weng. ZDOCK: an initial-stage protein-docking algorithm. *Proteins*, 52(1):80–87, Jul 2003.

[CW03] R. Chen and Z. Weng. A novel shape complementarity scoring function for protein-protein docking. *Proteins*, 51(3):397–408, May 2003.

[DHG03] Joseph A DiMasi, Ronald W Hansen, and Henry G Grabowski. The price of innovation: new estimates of drug development costs. *Journal of Health Economics*, 22(2):151 – 185, 2003.

[GJS97] H. A. Gabb, R. M. Jackson, and M. J. Sternberg. Modelling protein docking using shape complementarity, electrostatics and biochemical information. *Journal of Molecular Biology*, 272(1):106–120, Sep 1997.

[Groa] Gold Standard Group. OpenGL - The Industry Standard for High Performance Graphics. `http://www.opengl.org/`. Accessed: 2014-4-1.

[Grob] Khronos Group. Opencl: The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`. Accessed: 2014-3-31.

[HMWN02] I. Halperin, B. Ma, H. Wolfson, and R. Nussinov. Principles of docking: An overview of search algorithms and a guide to scoring functions. *Proteins*, 47(4):409–443, Jun 2002.

[HVJW10] H. Hwang, T. Vreven, J. Janin, and Z. Weng. Protein-protein docking benchmark version 4.0. *Proteins*, 78(15):3111–3114, Nov 2010.

[Int] Intel. Intel Math Kernel Library. `http://software.intel.com/en-us/intel-mkl`. Accessed: 2014-4-4.

[KBCV06] Dima Kozakov, Ryan Brenke, Stephen R. Comeau, and Sandor Vajda. Piper: An fft-based protein docking program with pairwise potentials. *Proteins: Structure, Function, and Bioinformatics*, 65(2):392–406, 2006.

[KDFB04] D. B. Kitchen, H. Decornez, J. R. Furr, and J. Bajorath. Docking and scoring in virtual screening for drug discovery: methods and applications. *Nature Reviews Drug Discovery*, 3(3):935–949, July 2004.

[KKSE+92] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. A. Friesem, C. Aflalo, and I. A. Vakser. Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques. *Proceedings of the National Academy of Sciences of the United States of America*, 89(6):2195–2199, Mar 1992.

[Lab] Oak Ridge National Laboratory. Introducing titan: Advancing the era of accelerated computing. `http://www.olcf.ornl.gov/titan/`. Accessed: 2014-3-31.

[LG09] James Larus and Dennis Gannon. Multicore Computing and Scientific Discovery. In *The Fourth Paradigm: Data-Intensive Scientific Discovery*, pages 125–130. Microsoft Research, 2009.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[LW13] Marc F. Lensink and Shoshana J. Wodak. Docking, scoring, and affinity prediction in capri. *Proteins: Structure, Function, and Bioinformatics*, 81(12):2082–2095, 2013.

[MMV+10] Gary Macindoe, Lazaros Mavridis, Vishwesh Venkatraman, Marie Dominique Devignes, and David W. Ritchie. Hexserver: an fft-based protein docking server powered by graphics processors. *Nucleic Acids Research*, 38(suppl 2):W445–W449, 2010.

[MV08] John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.

[Nvia] Nvidia. Cuda c programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Accessed: 2014-3-31.

[Nvib] Nvidia. Cufft user guide. `http://docs.nvidia.com/cuda/cufft/index.html`. Accessed: 2014-3-31.

[Nvi08] Nvidia. Nvidia tesla c1060. `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C1060_US_Jan10_lores_r1.pdf`, 2008. Accessed: 2014-3-31.

[Nvi09] Nvidia. Nvidia fermi compute architecture whitepaper. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009. Accessed: 2014-3-31.

[Nvi11] Nvidia. Nvidia tesla c2075. http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf, 2011. Accessed: 2014-3-31.

[Nvi12] Nvidia. Nvidia kepler gk110 architecture. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. Accessed: 2014-3-31.

[Nvi13] Nvidia. Kepler family datasheet. http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf, 2013. Accessed: 2014-3-31.

[PF11] I. Pechan and B. Feher. Molecular docking on fpga and gpu platforms. In *2011 International Conference on Field Programmable Logic and Applications (FPL)*, pages 474–477, Sept 2011.

[RHS+08] Christopher I. Rodrigues, David J. Hardy, John E. Stone, Klaus Schulten, and Wen-Mei W. Hwu. Gpu acceleration of cutoff pair potentials for molecular modeling applications. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 273–282, New York, NY, USA, 2008. ACM.

[RV10] David W. Ritchie and Vishwesh Venkatraman. Ultra-fast fft protein docking on graphics processors. *Bioinformatics*, 26(19):2398–2405, 2010.

[RVD95] R. Rosenfeld, S. Vajda, and C. DeLisi. Flexible docking and design. *Annual Review of Biophysics and Biomolecular Structure*, 24(47):677–700, July 1995.

[SCFR10] S. F. Sousa, N. M. Cerqueira, P. A. Fernandes, and M. J. Ramos. Virtual screening in drug design and development. *Combinatorial Chemistry and High Throughput Screening*, 13(5):442–453, Jun 2010.

[SCTP13] Martin Simonsen, Mikael H. Christensen, Ren Thomsen, and Christian N.S. Pedersen. Gpu-accelerated high-accuracy molecular docking using guided differential evolution. In Shigeyoshi Tsutsui and Pierre Collet, editors, *Massively Parallel Evolutionary Computation on GPGPUs*, Natural Computing Series, pages 349–367. Springer Berlin Heidelberg, 2013.

[SH08] B. Sukhwani and M.C. Herbordt. Acceleration of a Production Rigid Molecule Docking Code. In *FPL*, pages 341–346, 2008.

[SH09a] B. Sukhwani and M.C. Herbordt. Accelerating CHARMM Energy Minimization Using Graphics Processors. In *Proc. Symp. on Application Accelerators in High Performance Computing*, 2009.

[SH09b] B. Sukhwani and M.C. Herbordt. FPGA-Acceleration of CHARMM Energy Minimization. In *Proc. High Performance Reconfigurable Computing Technologies and Applications*, 2009.

[SH09c] Bharat Sukhwani and Martin C. Herbordt. Gpu acceleration of a production molecular docking code. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 19–27, New York, NY, USA, 2009. ACM.

[SH10a] B. Sukhwani and M.C. Herbordt. Fast Binding Site Mapping using GPUs and CUDA. In *Proc. High Performance Computational Biology*, 2010.

[SH10b] B. Sukhwani and M.C. Herbordt. FPGA Acceleration of Rigid Molecule Docking Codes. *IET Computers and Digital Techniques*, 4(3):184–195, 2010.

[SH14] B. Sukhwani and M.C. Herbordt. Increasing Parallelism and Reducing Thread Contentions in Mapping Localized N-body Simulations to GPUs. In V. Kindratenko, editor, *Numerical Computations with GPUs*. Springer Verlag, 2014.

[SIS+13] Takehiro Shimoda, Takashi Ishida, Shuji Suzuki, Masahito Ohue, and Yutaka Akiyama. Megadock-gpu: Acceleration of protein-protein docking calculation on gpus. In *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, BCB'13, pages 883:883–883:889, New York, NY, USA, 2013. ACM.

[Tea] Open MPI Team. Open Source High Performance Message Passing Library. `http://www.open-mpi.org/`. Accessed: 2014-4-4.

[TV06] Andrey Tovchigrechko and Ilya A. Vakser. Gramm-x public web server for proteinprotein docking. *Nucleic Acids Research*, 34(suppl 2):W310–W314, 2006.

[VCC+10] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association.

[VGH04] T. VanCourt, Y. Gu, and M.C. Herbordt. FPGA acceleration of rigid molecule interactions. In *FPL*, 2004.

[VH06] T. VanCourt and M.C. Herbordt. Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing*, v2006:1–10, 2006.

[WH11] Shih-Chieh Wei and Bormin Huang. Gpu acceleration of predictive partitioned vector quantization for ultraspectral sounder data compression. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 4(3):677–682, Sept 2011.

CURRICULUM VITAE

Raphael J. Landaverde

I am an aspiring computer engineer at the cusp of hardware and software working to solve a wide variety of computing problems. Graduated Salutatorian of the College of Engineering at Boston University with a Biomedical Engineering B.S. in 3 years. Currently studying for a Computer Engineering M.S. with an emphasis on software engineering at Boston University. I specialize in software engineering, high performance computing, and GPU computing, but I dabble in everything.

Contact Raphael J. Landaverde

College of Engineering, Boston University, 44 Cummington Mall, Boston, MA 02215, USA

Education **Boston University**, B.S., Biomedical Engineering,
Salutatorian, Summa Cum Laude,
September 2009 – May 2012.

Boston University, M.S. Candidate, Computer Engineering,
September 2012 – present.
Thesis advisor: Professor Martin Herbordt, PhD.

Relevant Coursework Cybersecurity, Operating Systems, Networking the Physical World, Advanced Computing Systems and Architectures, Intro to Computer Graphics, Operating Systems, Advanced Algorithms and Data Structures, Advanced Signals and Systems in Biomedical Engineering, High Performance Programming, Biological Database Analysis

Notable Projects **GPU PIPER** Further accelerated a production molecular docking code using Nvidia GPUs and CUDA. Achieved 3.3x speedup over a comparable 8 core CPU.

Automous Car Developed an autonomous car for the course Networking the Physical World. The car used infared sensors and SunSPOT processors to self navigate through a hallway and turn corners.

Mario Vivarium Created a simple vivarium using openGL and Java in which a simple version of Mario runs around chasing goombas, while being targeted by a lakitu from above. Developed for the course Intro to Computer Graphics.

Algorithmic Odyssey Developed a small tactical RPG video game for the cource Advanced Algorithms and Data Structures. Used perlin noise to generate unique maps with increasing difficulties and obstacles, implemented pathfinding algorithms, and created various AIs for the enemy units using alpha-beta pruning.

Ramdisk Filesystem Designed a basic filesystem using the linux kernel and an ioctl interface, capable of all basic file operations. Created for the Operating Systems course.

Vulnerable Site Created a vulnerable website with various integrated security challenges for use as an in class exercise in the course Cybersecurity.

Work Experience

Microsoft
Software Development Engineer Intern,
June 2013 – August 2013
Worked on the Skydrive Pro sharepoint server backend for document syncing, investigating file schema, and creating tools for analyzing the infrastructure.

Boston University ECE Department
Undergraduate Teaching Fellow,
January 2012 – May 2012
Provided support for course EC527, High Performance Programming. Held regular lab hours and graded assignments.

Boston University Neuromuscular Research Center
Lab Assistant,
September 2010 – May 2011
Used MATLAB to assist in developing a muscle force model with a physiological basis in conjunction with research associates. Developed tools for more clearly presenting raw data.

Publications

B. Humphries, H. Zhang, J. Sheng, R. Landaverde, and M.C. Herbordt (2014): *3D FFT on a Single FPGA*.Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM 2014), pp. TBD.