GPU-OPTIMIZED GRAPH THEORY ANALYSIS OF ALLOSTERIC PROTEIN SIGNALING NETWORKS

BY

CODY STEVENS

A Thesis Submitted to the Graduate Faculty of
WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES
in Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE
Computer Science
May, 2015
Winston-Salem, North Carolina

Approved By:
Samuel S. Cho, Ph.D., Advisor
Todd Torgersen, Ph.D., Chair
William Turkett, Ph.D.
# Table of Contents

List of Figures ................................................................. iv

Abstract ................................................................. v

Chapter 1 Introduction ......................................................... 1

Chapter 2 History of Supercomputing ........................................ 3
  2.1 Overview of Supercomputing ........................................ 3
  2.2 Supercomputing Milestones ........................................ 4
  2.3 Supercomputing Accessibility ....................................... 7

Chapter 3 Overview of GPU Technology ................................... 8
  3.1 Computer Graphics Technology and Parallel Calculations ........ 9
  3.2 General Purpose GPU Computing .................................. 10
  3.3 General Purpose GPU Programming Languages ................. 11
    3.3.1 CUDA for General Purpose GPU Programming ............. 12
    3.3.2 GPU Programming Parallel Performance Advantages ....... 17
    3.3.3 GPU Programming Bottlenecks .............................. 18

Chapter 4 Introduction to Graphs ......................................... 20
  4.1 Overview of Graphs ............................................... 20
  4.2 Floyd-Warshall All-Points Shortest Path ......................... 21
    4.2.1 Parallelizing the Floyd-Warshall Algorithm ............. 22
  4.3 Dijkstra’s Single Source Shortest Path .......................... 24
    4.3.1 Parallelizing Dijkstra’s Algorithm ........................ 26
  4.4 Girvan-Newman Algorithm for Community Analysis .............. 27

Chapter 5 Optimizing Graph Analysis for GPU Computation ........... 30
  5.1 Parallel Betweenness Algorithm ................................. 31
  5.2 Adapting a Dijkstra’s based Betweenness Algorithm .......... 33
    5.2.1 Performance Analysis of Parallel Betweenness Algorithm .... 39
  5.3 Extending Betweenness Algorithm to Perturbation .............. 42
### List of Figures

<table>
<thead>
<tr>
<th>Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>GPU Program Lifecycle</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>GPU Execution Model</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>GPU Memory Hierarchy</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Parallel Floyd Warshall Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Dijkstra’s Algorithm</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Community Detection for NCAA Conferences</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Floyd Warshall Betweenness</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>GPU Graph Representation</td>
<td>34</td>
</tr>
<tr>
<td>5.3</td>
<td>Parallel Dijkstra’s based Betweenness Algorithm</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Betweenness Speed Up vs. Node Number</td>
<td>40</td>
</tr>
<tr>
<td>5.5</td>
<td>Betweenness Speed Up vs. Edge Saturation</td>
<td>41</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance Analysis as Kernel Block Count Increases</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>Graph Representation of MetRS:tRNA$^{C\text{ys}}$</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Graph Representation of CysRS:tRNA$^{C\text{ys}}$ Protein:RNA</td>
<td>46</td>
</tr>
<tr>
<td>6.3</td>
<td>Observed $K_m$ vs. Perturbation Analysis</td>
<td>49</td>
</tr>
<tr>
<td>6.4</td>
<td>Observed $k_{Cat}$ vs. Perturbation Analysis</td>
<td>50</td>
</tr>
<tr>
<td>6.5</td>
<td>Observed $k_{Cat}/K_m$ vs. Perturbation Analysis</td>
<td>51</td>
</tr>
<tr>
<td>6.6</td>
<td>Community Detection and Analysis for CysRS</td>
<td>53</td>
</tr>
<tr>
<td>6.7</td>
<td>Community Detection and Analysis for MetRS</td>
<td>54</td>
</tr>
<tr>
<td>6.8</td>
<td>Community Detection and Analysis for RNAse-P</td>
<td>55</td>
</tr>
</tbody>
</table>
Abstract

Cody A. Stevens

Graph algorithms have been widely applied to science and engineering problems, but the practical applications of graph algorithms to ever-growing, large data sets requires significant computational time, motivating the need for efficient, parallel solutions on modern computing architectures. Here, we develop a novel GPU-optimized parallel algorithm for measuring the betweenness centrality of edges of a graph that is over 1000x faster than the CPU implementation of the algorithm in the NetworkX software package for 1,000-5,000 nodes. Inspired by biochemical mutagenesis experiments, we also introduce a new approach to measure how the betweenness centrality changes with the perturbation of a graph. As a direct application, we performed MD simulations of several protein-RNA complex, where we abstracted the residues and their interactions as nodes and edges, respectively, to characterize the molecular interaction network. We then systematically compare the change in the betweenness upon the deletion of the residue, akin to mutagenesis experiments performed in the laboratory, to create a predictor for mutagenesis experiments. Finally, we compared communal detection results to previously determined protein domains, finding a high correlation between residue-residue interactome communities and larger protein domains.
Chapter 1: Introduction

Computational biophysics is one of the most researched subjects in the academic community. More specifically, protein folding dynamics and protein-protein interactomes have been the focus of many studies [15, 46, 33], as these biological processes occur constantly in every organism. Complications within these processes have been linked to many diseases and medical conditions [34].

Proteins are large macromolecules made up of a linear chain of residues, and they fold into specific structures to perform their biological functions in cells. There can be hundreds of residues within a protein and for researchers, and dissecting how they carry out their biological functions remains a great unsolved problem in biology. Molecular Dynamics (MD) Simulations are able to simulate the folding process of a protein, by calculating the force exerted on the individual atoms within a protein. The velocity and positions of these atoms can be computed over multiple time steps [5]. While MD simulations can help us study the folding dynamics of a protein through a computer simulation, the overall time to complete this computation may take several hours to many days just to compute as little as 10 nanoseconds of protein folding.

In Computer Science, it is possible to map many problems, including those involved in protein dynamics, to that of a graph data structure. Graphs are heavily used in the fields of Computer Science and Mathematics, and algorithms to efficiently analyze these data structures have existed ever since the first computers were built. By mapping the protein dynamics to a graph data structure, our aim is find a correlation between metrics used to describe graph structures, primarily shortest path and edge betweenness, as well as determine if previously known methods for community
detection can be applied to the individual residues within a protein.

Graph data structures contain large amounts of data though, and the algorithms previously used to analyze these graphs are asymptotically slow. Thus for large problems, such as abstracting an entire protein to a graph data structure, it is not feasible to perform graph analysis within a reasonable amount of time. By creating parallel GPU-optimized algorithms, the same analysis can be performed in a much more reasonable amount of time. Our aim is to provide researchers with a metric for identifying key residues within protein structures that can be the focus of future \textit{in vitro} studies.

Throughout the remainder of the thesis, I will detail an overview of supercomputing and how it has impacted GPU programming today. Then, I will discuss common graph algorithms, such as the Floyd Warshall and Dijkstra’s algorithm for computing shortest path, as well as their common CPU parallelization methods. These methods will be recast into a GPU-Optimized edge betweenness algorithm, and I will discuss how it relates to the Girvan-Newman algorithm for community detection. Lastly, I will compare our graph metrics across three proteins, analyze correlations between mutation experiments done \textit{in vitro} to node perturbations within a given graph, and apply community detection for each protein and previously determined subsets found within each protein.
Chapter 2: History of Supercomputing

Since they were first created, computers have been powerful tools used by researchers in a very wide variety of disciplines such as mathematics, physics, chemistry, and economics to provide insight into some of the world’s most complex and difficult problems. The nature of these calculations can vary greatly, from solving Newton’s method for determining the zero-crossings [28] of a function for numerical analysis to modeling a dynamic economic system using agent-based simulations [45] over time. While each of these problems may vary in complexity, they are each still solved by performing a finite number of operations. We measure the performance and power of a computer based on the number of operations it can perform over a given set of time, so while it may be possible to any computer to solve the above problems the amount of time needed to compute the results may vary greatly based on the performance capabilities of the machine [25]. While computers have improved greatly since their creation, some problems are still so complex that it is infeasible to perform them on standard machines. However, as computers continue to improve the number of complex problems, we are able to solve within a reasonable amount of time also increase [18].

2.1 Overview of Supercomputing

The term supercomputer is reserved for machines that are at the frontline of computational power and capacity. One comparison of two computers’ speeds is to determine the number of floating point operations that each computer can perform over a given period of time. This metric, ”FLOPS” (floating point operations per second) is often prefixed prefaced with metric prefixes giga-, tera-, etc., when dealing with large
values. It is a strong indicator of speed and performance because even though computers may utilize the same number of processors, each running at identical speeds, the overall performance of the machine can be influenced by architecture utilized [21]. According to the Top 500 Supercomputers, the Tianhe-2 located at Sun Yat-sen University, Guangzhou, China is the fastest among them. The Tianhe-2 is capable of 33.86 peta-FLOPS and reportedly cost 390 million dollars to construct utilizing 16,000 nodes, each containing two Intel Xeon IvyBridge processors and three Xeon Phi processors [3].

2.2 Supercomputing Milestones

Physicist John Mauchly first proposed an all-electronic calculating machine in 1942. With the U.S. Army needing to perform complex trajectory tables for ballistics, the ENIAC was commissioned and built between 1943 and 1945 [8]. The ENIAC (Electronic Numerical Integrator And Computer) was the first Turing-complete electronic computer and is considered to be the very first supercomputer. The ENIAC was the first computer to run at electronic speed without being slowed by mechanical parts. Like many supercomputers today, the ENIAC took up 1,800 square feet and cost roughly 500,000 dollars [47]. ENIAC was capable of performing 300 FLOPS and could perform 5,000 additions, 357 multiplications or 38 divisions in one second [7]. The ENIAC’s size can be attributed to the large number of vacuum tubes within its construction, with almost 18,000 tubes housed within it [35]. Primarily built with vacuum tubes, it was not uncommon for the ENIAC to suffer a malfunction every one or two days.

Over the next several years, engineers transitioned to the silicon based transistors that we use today because of advances in lithography led to computers shrinking from the size of warehouses to the size of a smartphone used today. Technology continued
to evolve at a rate in which the number of transistors that could fit on any given space would double every 18 months, thus leasing to exponential increases in memory and processor speed, a trend described by Moore’s Law \[32, 44\].

Over time as computer hardware became more powerful and affordable, the construction of supercomputers would change from custom built hardware to a stronger use of consumer-grade parts. These supercomputers were often combinations of individual computers that were wired together and unified through a common operating system. Beowulf was the name of a cluster of these consumer-grade computers first built in 1994 by Thomas Sterling and Donald Becker at NASA. Beowulf clusters often have several individual computers connected together through local area networks and use software packages such as MPI and NFS to transfer data and communicate with one another. Beowulf clusters act as a single computer using a unified operating system. Of all the individual computers connected together, typically one or two of these are considered to be head nodes. These head nodes are a user’s portal to the remaining computers known as clients, or compute nodes. Work is submitted to the cluster through the head nodes and scheduled to run on the clients. The scheduling of work is transparent to the user and the end result is transferred back to the head node and outputted to the user. This model is still heavily used in supercomputers today with work scheduled from head nodes to compute nodes and stored through shared file systems across all computers in a cluster \[36\]. The use of consumer-grade hardware was often much cheaper to use and implement than custom built hardware and with the overall cost decreasing more and more supercomputers began to be constructed.

Constructed in 1997, ASCI Red was notable for being one of the first supercomputers to be constructed using consumer-based technology and was the first computer to perform a TeraFLOP or one trillion floating point operations per second \[1 \[1\].
Utilizing a total of 9,298 Intel Pentium Pro processors [1], ASCI Red was roughly the same size as the ENIAC from 1946. By utilizing mass market architecture as opposed to custom built hardware, the overall maintenance of the machine was much easier than that of its predecessor. Where programming changes made to the ENIAC computer would take several weeks and many engineers to design and build the hardware, many of the components within ASCI Red were readily available as they were already being used within many personal computers. Not only did ASCI Red have a computational advantage that far exceeded its predecessor, it was also vastly more adaptive to advances in technology as well.

Until recently, many of the world’s fastest supercomputers were constructed in a similar manner as ASCI Red. These implementations relied heavily on the power of Central Processing Units (CPUs) to perform a majority of the calculations. With recent advances in programmability for Graphics Processing Units (GPUs) and with the performance gains of GPUs increasing over CPUs, many engineers have begun creating supercomputers that balance the workload of calculations between CPUs and GPUs. The first supercomputer to utilize a heterogeneous CPU/GPU architecture was the Tianhe-1A, built in 2010. Using a combination of 14,335 Xeon X5670 processors along with 7,168 NVIDIA Tesla GPUs, it was estimated that the Tianhe-1A was able to theoretically achieve a speed of 4.701 petaflops. With the current leader, Tianhe-2 able to achieve 33.86 petaflops, supercomputing has progressed substantially in only a short amount of time [12]. It is estimated that by the year 2018 the Summit supercomputer, located at Oak Ridge National Labs, will overtake the Tianhe-2 as the world’s fastest super computer. Improving upon Oak Ridge’s Titan supercomputer, Summit is estimated to out perform the Titan’s 18,688 compute nodes by a factor of five with only 3,400 nodes, each utilizing NVIDIA Volta GPUs connected with NVIDIA’s high-speed NVLink.
2.3 Supercomputing Accessibility

While common parallel processing libraries exist, such as MPI [6] and OpenMP [2], in order to allow parallel applications to be written more easily, there are far more considerations to be made when developing code designed for parallel architectures. When writing code for distributed systems, it is important for each node or resource within the system to be utilized at all times without any nodes being idle throughout the length of the calculation to maximize available resources. Since these nodes are performing calculations in parallel, it is also important to make special considerations when allocating memory among all nodes and to manage the transfer of data between nodes as needed. When developing a single threaded application, the programmer does not have to make these considerations and therefore does not need to spend as much time optimizing their code.

Even with these tools, access to the world’s fastest supercomputers is limited among the entire researching community. Many of these organizations that maintain these large supercomputing facilities charge a fee in order to use their machine for a period of time. This creates a financial barrier for many researchers hoping to gain access to these state of the art supercomputers and, in most cases, one must be an established researcher in order to be considered to use them. If these financial and developmental barriers were to decrease within the coming years, supercomputing would be much more accessible to researchers that are trying to solve the world’s most complex problems.
Throughout the 1980s, companies began engineering accelerator hardware that was specifically designed to offload computationally complex graphical calculations from the CPU. These components, which would later become modern day GPUs, were primarily used in the manipulation and display of 2D graphics. These "graphics cards" sacrificed computational complexity that many general purpose CPUs had in order to solely focus on calculations involved within computer graphics. As programmers began creating applications to render and model 3D graphics on standard monitors, a heavier emphasis was placed on hardware that was dedicated to providing the best graphics. It wasn’t until 3D video games that a consumer-level market was developed for powerful graphics cards.

Just like any financial market, as the demand for more powerful graphics cards increased, so did competition between different graphics cards vendors. This competition led to large improvements in graphics card technology, boosting performance and minimizing the cost in which to produce them. It wasn’t until 1999 with the introduction of NVIDIA’s GeForce 256 graphics card that NVIDIA would introduce the GPU to computing, defining it to be ”a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second” [41]. With the GeForce 256 boasting more graphical functions than what previous graphics cards displayed, it would go on to be the first in a long series of GPUs that would be produced by some of the largest vendors that include NVIDIA and ATI (which later merged with the company AMD). Over the next few years, the improvements made in GPU technology would continue to grow and even outpace that of CPUs.
3.1 Computer Graphics Technology and Parallel Calculations

The process by which computers render graphical images is inherently parallel and can be broken down into a series of tasks known as the "graphics pipeline." Throughout each of these tasks, data relating to an image being processed can be manipulated independently and thus GPU manufacturers design hardware that can perform these calculations in parallel at a speed in which to display an image to the screen at a rate of at least 20 frames per second. Throughout the graphics pipeline, a computer representation of a 3D scene is composed of a series of points known as vertices that are used to generate thousands of 2D primitives, most of which consist of triangles, that create a 3D scene. These primitives are then processed by the GPU through rasterization to translate the 3D scene into individual pixel values for the monitor [29]. The GPU must perform calculations that approximate various properties pertaining to lighting and opacity, performing each operation independently for each primitive. As most of these parallel calculations are floating point operations, GPUs are optimized for these types of computations.

The number of operations that are involved in the graphics pipeline are relatively small for a specific set of data points, thus GPU manufacturers primarily focus on creating hardware that are optimized for this limited set of instructions [29]. This limitation on the GPU contrasts that of the multipurpose CPU but allows the GPU to perform its tasks with a higher level of efficiency. Therefore, GPUs focus on the utilization of a large number of parallel processors dedicated to performing graphical computations with a high throughput, whereas CPUs focus on performing a wide variety of operations very quickly. Though computer manufacturers have begun creating computers with multi-core CPUs, there still exists a substantial difference in resources available between the CPU and GPU. While CPU manufacturers favor processor speed over parallelism, GPU manufacturers generally favor creating hardware
with a larger number of individual processors with slower speeds [29]. Even though an individual GPU core may be slower than a single CPU, the overwhelmingly large difference in the number of cores on the GPU make them better suited for high throughput parallel tasks.

3.2 General Purpose GPU Computing

With GPUs becoming more powerful with each successive generation, the potential for GPUs to be used for general purpose computing became more and more apparent. It is common for most applications to have at least some portion that can be parallelized. For many problems that are solved using supercomputers, there are at least some independent floating point calculations that can be optimized for GPUs. Unfortunately, for a GPU to be used for general purpose computing, instructions had to be translated so that a GPU could understand them in terms of vertices and primitives. It is much more difficult to write a non-graphical program to run on a GPU than to write an equivalent application to run on a CPU.

Even though GPUs were not being designed specifically for general purpose applications, the increase in complexity of 3D video game graphics motivated GPU manufacturers to build new hardware that was more flexible and allowed video game programmers to have more control while developing games. The rendering pipeline was changed from fixed-function units with hardwired algorithms to more flexible units that could perform custom computations.

Video game programmers would then be able to write their own custom shaders to calculate different properties of the 3D scene rather than being limited to the shader implementations were already built in the GPU hardware. This transition from GPUs with a limited set of instructions to a piece of hardware that was much more flexible and highly programmable was not only welcomed by game designers but
also by researchers who wished to take advantage of the computational power of the GPU [29].

3.3 General Purpose GPU Programming Languages

With GPU manufacturers NVIDIA and AMD producing generic, unified GPU architectures, both companies wanted to provide programmers the ability to program their GPUs for a variety of purposes besides graphical applications. Even though it was possible to program GPUs through graphical APIs, it still created a barrier for programmers as problems had to be approached in completely different ways. Several software packages have been released with the intention of providing a general way to program GPUs for a variety of applications.

Developed by researchers at Stanford University, Brook was the first attempt at a general purpose GPU programming language [9]. By creating extensions for the C programming language, the goal of Brook was to allow programmers to more easily access GPUs for non-graphical applications. Despite Brook being implemented on hardware that was not as well suited for general purpose computing as modern day GPUs, researchers were still able to produce programs that were able to perform up to eight times faster than equivalent applications programmed for the CPU [9].

Building on the advances that Brook made in the area of GPU general purpose computing, NVIDIA designed their own general purpose GPU programming language, the Compute Unified Device Architecture (CUDA) library [12]. With CUDA, programmers are allowed to write C/C++ applications that are able to execute parallel code on NVIDIA GPUs. ATI also began development on their own general purpose GPU programming language known as Stream [48].

Even though the previously mentioned libraries are hardware dependent, the OpenCL project intended to create a programming language that would allow code
to be run on both GPUs and CPUs that was independent on the hardware manufacturer [22]. Initially proposed by Apple and later adopted by Intel, AMD, and NVIDIA, OpenCL ensures that a wide array of hardware is supported, making it the most common framework for developing programs to be implemented across multiple brands of hardware. Even though OpenCL is based on the C/C++ languages, it does have some restrictions, limiting the ability for C/C++ programs to be compiled by OpenCL toolchain. This limits the ease of porting existing serial code to OpenCL for parallel execution [23]. For the remainder of this thesis, we will focus on CUDA.

3.3.1 CUDA for General Purpose GPU Programming

CUDA is designed as a collection of language constructs that allow parallel code to be written for NVIDIA GPUs without the need to learn many new programming methods. Created by NVIDIA, one of the primary goals of CUDA is to allow developers with a background in programming languages to be able to easily access NVIDIA GPUs for parallel computation with as much ease as possible. CUDA uses a heterogeneous computational environment, meaning that it uses two types of processors, CPUs and GPUs, during program execution [40]. This allows a programmer to utilize the strengths of both types of processors by writing a CPU program that is capable of offloading a certain amount of the calculations to the GPU for faster processing.

To execute parallel code meant for the GPU, a CUDA program will launch a "kernel", which is the equivalent of a parallel function call. The process for calling a CUDA kernel is very similar to calling a C/C++ function, but, when a kernel is called, it executes in parallel on the GPU [40]. Each kernel call generates a collection of subprocesses on the GPU called "threads." Each thread executes all code within the launched kernel independently of all other threads running on the GPU (Figure 3.1).
Threads created by a kernel call are arranged on the GPU in a multi-level hierarchy that can be configured to fully utilize the hardware of any CUDA-capable GPU. Threads created by the kernel are arranged into either one, two, or three dimensional arrays known as "blocks" that will execute independently on different processors within a GPU [40] (Figure 3.2). Each thread is assigned a unique thread ID for each dimension of the block and is stored as a local variable for each thread. The blocks that are created can be thought of as modular subdivisions of a computation that are executed independently and in arbitrary order. All of the blocks within a kernel are arranged into either a one or two dimensional collection of blocks known as a "grid." Blocks are also assigned a unique block ID for each dimension within the grid [40].

On any given GPU, there are only a finite number of blocks that can execute at the same time, depending on the number of computational cores that are available on the GPU. If there is a relatively small number of computational cores, a subset of the blocks will execute simultaneously, and, when they are completed, a second set of blocks will execute. This process continues until all blocks have finished execution.

Figure 3.1: Example execution of a CUDA program. Initially the code executes serial commands until a kernel call is executed from the host CPU. Afterwards, blocks of threads are arranged into a grid and parallel computations are performed. Once the kernel call is finished, the CPU host continues to execute commands serially until another kernel call is executed or the end of the program is reached.
Figure 3.2: Just as blocks are arranged into a two-dimensional grid, threads within each block can be arranged in three-dimensions. Each block has two global parameters, blockId.x and blockId.y, that is can use to distinguish itself from other blocks. These values are passed to each thread, and each thread also has parameters corresponding to threadId.x, threadId.y, and threadId.z that it can use to distinguish itself from all other threads.

If there is a relatively high number of computational cores available on the GPU, it is possible for all of the blocks to execute simultaneously and be complete at the same time. This method of arranging blocks provides a flexible framework to adapt to different GPUs that contain different numbers of computational cores.

Memory distribution within CUDA is distributed in a similar hierarchical manner (Figure 3.3). Grid-level memory exists at the highest level and consists of general purpose read/write global memory, read-only constant memory, and read-only texture memory [40]. Each thread and block within a kernel call has access to this level of memory, and it is in many ways analogous to RAM in a CPU computing environment. At the next level down, each block has its own shared memory that is accessible by each thread within that block that is much faster than accessing global memory. This memory is primarily used as a type of cache for operations that would otherwise require many redundant reads or writes to global memory. Shared memory exists for
Figure 3.3: Memory model of the GPU. On the GPU there resides global memory that can be accessed by all blocks and threads within a kernel call. This memory is the slowest for reads and writes, although it persists throughout the life of a CUDA program. Each block is capable of allocating block shared memory that can only be accessed by threads within the block. This memory is faster than global memory, but the fastest memory is thread local memory only readable and writable by individual threads. Thread and block memory are only allocated during the lifetime a kernel call.

The lifetime of the block and is freed once a block finishes execution. At the lowest level, each CUDA thread possesses a relatively small amount of private memory only accessible by that thread. This is the fastest and most limited type of memory available on NVIDIA GPUs [40].

The execution model implemented by NVIDIA GPUs is a generalization of the single instruction, multiple data (SIMD) model known as single instruction, multiple threads (SIMT). In SIMD computer hardware, a set of computational cores execute the same operations on different pieces of data over the same time interval. SIMD architectures are well suited for massively parallel tasks where calculations over a set of data are repeated, but each calculation is independent of one another. A common approach when developing for SIMD architectures is to assign each processor to a
subset of memory and then have each processor perform an operation on that piece of data. Once all cores have performed their operation, all cores are synchronized and the program continues execution. By comparison, in a single instruction, single data (SISD) programming model, each thread would need to be synchronized at the end of execution, resulting in an increase in overhead to the overall computations. From the programmer’s perspective the SIMT model used by NVIDIA GPUs is identical to the SIMD model, meaning that no further understanding is needed in order to utilize CUDA.

CUDA also allows for barrier synchronization similar to that of OpenMP. Barriers used within CUDA allow the programmer to halt CPU or GPU computation and synchronize threads through simple methods before program execution resumes. This gives the programmer much more freedom when developing CUDA code as the programmer can choose to halt the CPU while the GPU finished performing calculations or allow it to continue at the same time as the GPU. Without a fully flexible barrier synchronization scheme in place, many application could not be fully optimized for a heterogeneous computing environment.

With the introduction of CUDA 5, NVIDIA has also provided a framework for dynamic parallelism, allowing a programmer to execute kernel calls within kernel calls. In previous versions of CUDA it was only possible to achieve recursion through the use of GPU functions that could be called by threads within a kernel. The use of these functions were relatively slow, but with the addition of dynamic parallelism a single thread can make a kernel call and create a new set of threads for a completely parallel recursive algorithm. Dynamic parallelism is limited though in the number of levels of recursion that a single thread can obtain. When a thread executes a kernel call, it performs just a CPU would continuing computation without waiting for its kernel to finish execution.
3.3.2 GPU Programming Parallel Performance Advantages

The greatest advantage of utilizing GPUs for general purpose computing is their inherently parallel nature. When processing graphics, it is important to minimize the amount of overhead introduced when processing data. With each graphics rendering scenario being a parallel process, designers of GPUs have always focused on ways to set up multiple concurrent operations with the lowest possible cost. The low overhead involved with performing parallel graphics processes allows GPUs used in general computing applications to perform very well when a high number of parallel calculations are needed. By comparison, CPUs often require much more overhead when invoking a large number of parallel processes \[29\]. GPUs then create an advantage during parallel computation by reducing the bottlenecks due to overhead involved in setting up massively parallel computations.

Parallelism in GPUs is also easier to manage than with CPUs. While thread scheduling and synchronization are nontrivial issues with CPUs, GPUs are designed to manage these problems natively on the hardware, reducing the amount of time that both the programmer and the application are required to spend in order to manage parallelism. With GPUs, programmers do not manage individual threads, but these are instead handled with relative ease through the simple kernel calls that manage the allocation of threads to blocks and execute multiple threads simultaneously.

The native parallelism in GPU architecture allows it to scale without requiring major hardware changes. The computing power of a GPU can be increased just by adding more computational cores and a small amount of circuitry to issue commands to these new cores. In contrast, the addition of more cores to a CPU is often more complex due their more intricate, multipurpose nature. This advantage in scalability has allowed GPUs with hundreds to thousands of computational cores to be designed allowing an equally large number of threads to be executed concurrently.
Though each computational core on the GPU may be weaker than individual cores on the CPU, the high throughput capabilities of the GPU makes up for this difference in performance.

### 3.3.3 GPU Programming Bottlenecks

While GPUs are well suited for solving certain problems, writing efficient code for them is not without its difficulties. One of the largest bottlenecks involved in GPU programming is the transfer of data between the CPU and GPU via the PCI Express bus. In a CPU, a program loads data in RAM from a hard drive or some other data source, and it can quickly access the data without significant performance issues with regard to the access. However, a GPU must first transfer all of the data needed for the execution before it can proceed. During this transfer, the GPU will not be able to execute and may sit idle for a significant amount of time. Depending on the size and frequency of these data transfers, the amount of overhead generated can severely impact the performance of an application.

Also, even though the parallel nature of GPUs makes them efficient at processing large amounts of data in parallel, serial portions of algorithms will not benefit from implementations on a GPU. With many modern GPUs containing hundreds of computational cores, if all but one GPU cores sit idle, the overall performance gains granted by the GPU are no longer present because we are essentially performing serial operations that could be computed much faster on a CPU. For GPUs to provide the best computational advantage, programs must be written in order to take advantage of as many resources as possible on the GPU at the same time. If a program performs a portion of its algorithm in serial, we lose the advantage of GPU programming because a large number of resources will sit idle during that phase of computation.

While working using distributed systems allows for greater parallelization, hard-
ware is not always perfect and may sometimes fail during computation. With standard computers containing one or two GPUs, this problem rarely arises, but consider one of the world’s largest supercomputers in the world made up of thousands of GPUs. During a program’s execution, if a GPU breaks or fails, that portion of data processing will be lost, and this could mean that the entire calculation must be restarted. With larger laboratories, it is not uncommon for individual pieces of hardware to break and require replacements before programs can execute once again. This concept of preserving information once a resource fails in order to not compromise the entire calculation is a bottleneck that can face some people using larger distributed systems.
Graphs are abstract data structures in computer science that are used to represent a set of objects and the relationships between those objects. Graphs can be used to model many different systems and are used often within the fields of computer science, mathematics, biology, and economics. Just like any other data structure within computer science, there are a number operations associated with graphs, and there has been a considerable amount of work regarding the research and study of algorithms designed to analyze the composition and layout of graphs.

4.1 Overview of Graphs

The simplest representation of a graph consists of two sets, $V$ and $E$, where $V$ is a set of vertices and $E$ is a set of edges that connect vertices within $V$ \[4\]. Graphs can be classified as 1) directed or undirected and 2) weighted or unweighted \[4\]. Within an undirected graph, each edge that connects node $i$ to node $j$ implies that there is also an equal connection in the opposite direction between node $j$ and node $i$. For directed graphs, each edge directed from node $i$ to node $j$ no longer implies that there is a connection from node $j$ to node $i$. Edges of unweighted graphs, are uniform and cannot be distinguished from one another. Weighted graphs have edges that are assigned a cost value, implying that flow between two nodes has an associated cost equal to the weight of the edge that connected them \[3\].

Graphs are most commonly implemented as either an adjacency list, or an adjacency matrix. For a graph with $N$ vertices, an adjacency list will represent the graph within a list of size $N$. Each index $i$ of the array list will contain a subset of vertices from the vertices within the graph that all all connected to vertex $i$ \[4\]. Array lists are
beneficial when representing sparse graphs, because the amount of memory needed to represent the graph is considerably less than that of an adjacency matrix. An adjacency matrix consists of a two-dimensional matrix with each dimension of size $N$ where indices $i$ and $j$ are the distance between nodes $i$ and $j$ [4]. Weighted graphs use the weights of edges to describe the cost between nodes, while for undirected graphs it is common to only use 1’s to denote edge connections between nodes. While adjacency matrices are expensive to maintain in terms of memory needed to represent a graph, they have an advantage when accessing the data within. In order to determine whether vertex $i$ is connected to vertex $j$, using an adjacency matrix only requires a simple table look-up at index $i$ and $j$ of the matrix, which is $O(c)$. However, for an adjacency list, in order to determine whether vertex $i$ was connected to vertex $j$, the adjacency list would have to be searched at index $i$ in order to find vertex $j$ within its list of adjacent nodes. This search is $O(N)$ because, within a graph of $N$ vertices, the largest number of vertices that a node can be connected to is at most $N$ [4].

There are a variety of algorithms used to search, traverse, and analyze numerous properties of graphs, but we will focus on algorithms used to calculate the shortest distance between vertices in graphs. For the remainder of this thesis, ”distance” refers to the accumulated cost of edges between a start and end vertex and all intermediate vertices within the path between the start and end.

### 4.2 Floyd-Warshall All-Points Shortest Path

In its currently recognized form, the Floyd-Warshall algorithm was first published by Robert Floyd in 1962 [16], but is essentially the same algorithm for determining the transitive closure of a graph, as described by Bernard Roy and Stephen Warshall, published in 1959 and 1962, respectively. The algorithm is used to compute shortest path between all vertex pairings $i$ and $j$ for a graph and store the results in a shared
adjacency matrix. The algorithm has an overall time complexity of \(O(N^3)\), for a graph containing \(N\) vertices.

The algorithm begins with an initialized adjacency matrix of size \(N \times N\), with each index \(i\) and \(j\) set to “infinity”, typically implemented as some extremely large number. Then, the distance between each vertex and itself is set to 0 while vertices connected by edges are updated, such that the distance between two vertices is the weight of the edge that connects them [16]. The algorithm then begins computing the shortest paths between all points utilizing three nested for-loops. The two inner most for-loops represent all vertex pairings \(i\) and \(j\) within the graph, while the outer most for-loop represents an auxiliary vertex \(k\). For each auxiliary vertex \(k\), the algorithm will update the distance between vertices \(i\) and \(j\) if the distance between \(i\) and \(k\), along with the distance between \(k\) and \(j\), are shorter than the previously found shortest distance that did not use vertex \(k\). Once all three for-loops have finished all iterations, the final output will be an adjacency matrix where each pair \(i, j\) is the shortest distance between vertices \(i\) and \(j\) [16].

4.2.1 Parallelizing the Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is a fundamental algorithm within graph theory analysis, making it a prime target for parallelization due to its utility. Unfortunately, there exists a loop carried dependence on the outer for-loop considering intermediate vertices, \(k\). The algorithm compares pathways using an intermediate node \(k\) to a previously found distance, optimizing over time. This creates a barrier in the amount of parallelization that can be applied to the algorithm because of the distinct time intervals that each element of the comparison is calculated. Even though we many not be able to parallelize the entire algorithm, it is still possible to achieve reasonable speed up and decrease the overall computation time.
Given 16 processors, each processor is assigned a block in the 4 x 4 grid. When considering a new node $k$, the processors containing the distance information to node $k$ broadcasts those values to all other processors within its row. The processor then, with the distances from node $k$, broadcasts those values to each other processor in its column. Once all messages have been broadcasted and received, each processor updates their own distances with these values.

The are two primary methods for parallelizing the Floyd-Warshall algorithm. The first method relies on a row decomposition on the adjacency matrix and dividing this decomposition across $P$ processors [30]. Given an $N \times N$ adjacency matrix, each processor would then be responsible for $N/P$ rows of the matrix. With each iteration of $k$ in the outer for-loop, the work of the second for-loop iterating over $i$, is split across all processors [30]. This approach assumes that each processor has their own private memory that contains their assigned rows of the decomposed matrix. This results in increased overhead for the calculation for each iteration of $k$, and the processor that is responsible for row $k$ must perform a universal broadcast with a message containing information of the distance between vertex $k$ and all other vertices. This message introduces some latency so the overall speed up gained from dividing the work of an inner for-loop across multiple processors must be greater than the overhead of broadcasting a message for each iteration of $k$ in order to justify this parallel approach [30].
Another parallelization approach of the Floyd-Warshall algorithm arranges processors in a grid, and assigns each processor a block of the adjacency matrix. Assuming \( P \) processors and an \( N \times N \) adjacency matrix, we would arrange our grid of processors into a \( \sqrt{P} \) by \( \sqrt{P} \) grid. Each processor would then be responsible for an \( N/\sqrt{P} \) by \( N/\sqrt{P} \) section of the adjacency matrix \([30]\). For each iteration of \( k \), all processors that contain a block of the matrix containing row \( k \) would broadcast a message to all processors within the same column of the processor grid, while all processors containing a block of the matrix with column \( k \) would broadcast a message with this information to all processors in the same row of the processor grid. Once each processor has all the information needed for iteration \( k \), it performs the necessary calculations. By sending these messages in parallel, there is a benefit in sending smaller messages that create less overhead. This approach is also not constrained by needing all processors to synchronize after each iteration of \( k \). When a processor finishes its calculation for a given iteration of \( k \), if the processor contains information for the next iteration of \( k \), it can go ahead and broadcast its information out without having to wait for all processors to finish. This allows processors to compute their section of the adjacency matrix as soon as it receives the information needed without having to wait for all processors to finish the previous iteration of \( k \). An example illustration is shown in Figure 4.1.

While the Floyd-Warshall algorithm is not the most ideal algorithm for parallelization, there are still considerable gains to be made because of the overall high asymptotic cost in its serial implementation.

### 4.3 Dijkstra’s Single Source Shortest Path

Dijkstra’s algorithm was first conceived by Edsger Dijkstra in 1956 and later published in 1959 \([14]\). The algorithm differs from the Floyd-Warshall algorithm, in that it only
Figure 4.2: Example execution of Dijkstra’s algorithm of single source shortest paths with five nodes. (a) Initially, the source node $s$ is added to the working set and that ends the first iteration. (b) Next, all nodes that are connected to the source node, $y$ and $t$, are considered, with node $y$ being added with the shortest distance of five. (c) As the working set grows, the number of nodes to consider increases, as well as the number of paths to nodes. (d) During the next iteration, node $t$ is found to have a shortest distance to the source node $s$, by using node $y$ as an intermediary node. This distance is shorter than the previously recorded distance, ten, and node $t$ is added to the working set and its shortest distance to the source node is recorded as eight. (e,f) The algorithm continues until all nodes have been added to the working set.

solves the shortest path problem from a single source vertex, $s$, and all other vertices in the graph. This greedy algorithm begins with a single source vertex, and over time builds a shortest path tree containing all vertices in the graph, where there exists a connected path to the source vertex. The algorithm has an asymptotic complexity of $O(N^2)$, where $N$ is the number of vertices within the graph [14].

The algorithm maintains two sets of vertices, one representing the working set of vertices that have been added to the shortest path tree, and the other representing all vertices that still need to be added [14]. The algorithm initially starts with a working set of only the source vertex, and the distance for every other vertex being the weight of the edge connecting it to the source vertex or ”infinity.” At each iteration of the
algorithm, the vertex with the current shortest path to the source is added to the working set and then the distance to the source for all vertices outside of the working set is updated. After a node is added to the working set, all neighboring nodes, outside the working set, will update their distance to the source node, if there exists a shorter path through the most recently added node [14]. There is once again a loop carried dependence in this algorithm due to the fact that, when we update the distance, we must wait for a vertex to be added to the working set. However, there have still been many attempts to parallelize Dijkstra’s algorithm.

4.3.1 Parallelizing Dijkstra’s Algorithm

In order to utilize Dijkstra’s algorithm to compute the all points shortest path, the serial algorithm must be performed $N$ times, once for each vertex to be considered as the source node. The first parallel implementation divides these $N$ serial algorithms across $P$ processors, where each processor is then responsible for $N/P$ serial algorithms [30]. Each execution of the serial algorithm is independent of one another, as only the source vertex changes. This approach replicates the shortest path tree across all processors and parallelizes the calculation for the all points shortest path.

A second approach parallelizes the actual Dijkstra’s serial algorithm, when adding the closest vertex to the working set. This approach divides the vertices across $P$ processors, such that each processor is responsible for $N/P$ vertices [30]. When calculating the closest vertex to the working set, each processor calculates a local minimum by deciding which one of its assigned vertices is the closest to the working set. Once all processors have found their local minimum, all processors perform a reduction to find the global minimum, and then the algorithm adds that global minimum to the working set. This approach has an added benefit, as it allows for a number of processors $P$ where $P > N$ when computing the all points shortest path.
Figure 4.3: Graph representation of the NCAA Division-1 Football schedule. Each node represents a single team, while each edge represents a game was scheduled between two teams. The Girvan-Newman algorithm was used to accurately place teams into communities representing the different conferences in the NCAA.

### 4.4 Girvan-Newman Algorithm for Community Analysis

Community Analysis has been a growing study within graph theory analysis, and has been applied to many biological studies over the past few years [15]. The goal of identifying communities within a network is to identify nodes that may be closely related to one another. The concept can be seen in looking at college football teams, and grouping them together by conference (Figure 4.3). Each conference can be considered a community, and those communities can be determined by looking at the scheduled opponents for each college team. Community analysis does not tell you what properties the nodes have in common, but only that certain nodes may be closely related to one another. This is applicable in protein-protein interactomes, as
the biological function of certain proteins can be inferred from other proteins within the same community, when some proteins have known functions and others do not.

The Girvan-Newman algorithm [20, 39] determines the communities within a network graph, by calculating the betweenness of each edge, and deleting the edge from the graph with the highest betweenness. This process continues until there are two distinct sub-graphs, at which point a modularity function is applied to the graph to determine whether or not the algorithm will be efficient in finding more communities. The two sub-graphs are considered to be within separate communities, and the Girvan-Newman algorithm is applied recursively to each sub-graph. The algorithm will terminate once all edges have been removed from the graph. The output of the algorithm is a dendogram, with each node being a leaf within the dendogram, and the inner nodes representing the collection of nodes within their combined subtrees [20].

The initial betweenness calculation is dependent on the all points shortest path, and it is the standard betweenness computation for edges. Edges are weighted based on the number of times they are used within all $N^2$ paths within the graph. The edge with highest betweenness is deleted, and the calculation is repeated. The calculation must be repeated because the deletion of one edge may affect the betweenness of the other edges within the graph. For efficiency’s sake, only the shortest paths for nodes using the deleted edge as an intermediary edge are recomputed. Once the edge has been deleted, the modularity function is applied. The modularity function generates a ratio based on the degree on each node, and the total number of edges within each subgraph. Then, they are summed over all sub graphs within the network. The intention being, that subgraphs that have potential distinct communities still within them will create a lower modularity score. This score means that all nodes within the sub graph are not heavily interconnected, and that there is some subset of nodes
that are more interconnected, allowing the algorithm to continue execution. At some point, a large set of nodes consisting of a community will be found and this subset of nodes will produce the highest modularity value. Any further subdivision of this graph will still produce reasonable modularity scores. However, since the number of nodes in each of these subgraphs is lower, the overall modularity score will be lower, indicating that the more optimal community was already previously found. The state of the graph when the modularity was the highest is the output of the graph.

To date, very little effort has been made to parallelize the Girvan-Newman algorithm, as research topics in developing an efficient betweenness algorithm has taken precedent in the recent years. Since the Girvan-Newman algorithm is computed using the betweenness algorithm, developing a faster betweenness algorithm will thus produce a faster community analysis detection method.
Chapter 5: Optimizing Graph Analysis for GPU Computation

As discussed in the previous chapter, graph algorithms have a wide variety of utility, as almost any system can be represented using a graph data structure. While the algorithms to perform these graph analyses may be very useful, that does not mean that they can be easily recast for parallel computations. Common characteristics of graph algorithms, particularly shortest path algorithms, are that there are inherent loop carried dependencies between time-steps, as answers computed in previous iterations of a loop are then needed during later iterations of the same loop [31]. Other issues in optimizing these algorithms for GPU computations involve the amount of memory available on a given graphics card, as it may not necessarily be large enough to store the entire graph data structure. Parallel computations achieve their highest efficiency when performing calculations on larger problem sizes, but when the amount of memory available on a given GPU is relatively small, there exists a physical hardware limit on the size of networks that can be analyzed. Finally, graph data structures are nonuniform, and are typically very imbalanced. Given that the GPU is a SIMD architecture, this nonuniform structure forces us to either optimize an algorithm capable of balancing work across all threads, or suffer a performance penalty, as some threads perform more work than others. In this chapter, I will present my efforts in 1) creating a GPU-optimized parallel algorithm for computing the betweenness metric of a given network, and 2) applying the algorithm to computations focused on the community analysis of graph networks. There have been many studies conducted previously on optimizing graph algorithms for GPU computations, many of which focus on mapping the irregular graph data structures to a SIMD architecture [26, 37], while others have approached more complex problems, including shortest path [24].
5.1 Parallel Betweenness Algorithm

As previously described, to compute the betweenness for a given graph, the all-points shortest path must be computed in order to find the individual shortest paths between all pairs of nodes. Therefore, developing a faster shortest path algorithm, will then give us a faster algorithm for computing the betweenness. Our first attempt at creating a parallel betweenness algorithm, involved developing an algorithm based on Floyd-Warshall’s shortest path algorithm, similar to the parallel CPU implementation. In the parallel CPU algorithm, portions of the adjacency matrix are distributed across all processors, with work being assigned as a grid of blocks, consisting of portions of the adjacency matrix. An advantage of using the GPU for this algorithmic approach, is that a GPU is not as limited in the number of threads that it can assign to the computation because the CPU is in assigning cores to do the same amount of work. The GPU has many more resources readily available, that it is possible to assign an individual thread to a single entry in the adjacency matrix, resulting in $N^2$ threads being assigned per kernel call. As the Floyd-Warshall algorithm iterated over the outer most for-loop, it considers paths utilizing a new node $k$ for all $i, j$ pairings. A parallel kernel call would pass a parameter $k$, acknowledging which node should be considered as the new intermediary node, to $N^2$ threads, each one representing a different $i,j$ pairing. This approached requires $N$ kernel calls to consider each node as a parameter $k$. This algorithm was successful in computing the all points shortest path, but did not lend itself well to computing the betweenness of a graph.

The other main component of calculating the betweenness of a graph involves recording the individual shortest paths, and creating a weight for each edge in the graph. Unfortunately, the Floyd-Warshall algorithm does not inherently record these paths, and we are also unable to update the betweenness of edges until the algorithm has fully completed, as a path using an intermediary node $k$ could be completely
In the above example $i$ and $j$ represent nodes 2 and 9, respectively. When a shorter path is discovered between nodes $i$ and $j$ using intermediary node $k$, the paths from $i$ to $k$ and $k$ to $j$ must be recorded as the path from $i$ to $j$. This is done by copying the third dimension at index $i,k$, inserting the intermediary node $k$ into the list (In the example, node 6), and finally copying the values at index $k,j$. This results in all intermediary nodes being recorded for the new path from $i$ to $j$.

![Diagram](image.png)

Figure 5.1: ]

In the above example $i$ and $j$ represent nodes 2 and 9, respectively. When a shorter path is discovered between nodes $i$ and $j$ using intermediary node $k$, the paths from $i$ to $k$ and $k$ to $j$ must be recorded as the path from $i$ to $j$. This is done by copying the third dimension at index $i,k$, inserting the intermediary node $k$ into the list (In the example, node 6), and finally copying the values at index $k,j$. This results in all intermediary nodes being recorded for the new path from $i$ to $j$.

Different from a path using an intermediate node $k + 1$. In order to record the paths for each $i, j$ pairing, the adjacency matrix was extended from a 2D representation to a 3D representation, with the third dimension recording all intermediary nodes along the path between each paring of $i$ and $j$. As the number of intermediary nodes for each path is unknown at the beginning on the computation, the adjacency matrix had to be extended by a factor of $N$, in order to account for the worst case scenario, where a path between two nodes used the remaining $N - 2$ nodes as intermediary nodes.

Whenever a thread updates its distance value after considering a new intermediate node $k$, it then copies the contents of the path from $i, k$, and the current path from $k$ to $j$ into the third dimension of the array, for index $i, j$. Once all nodes have been considered as intermediate nodes, each thread then uses a sliding window to update the betweenness for the intermediary edges between nodes $i$ and $j$. See Algorithm 1 for a pseudocode representation and Figure 5.1 for an illustrated example. This
approach was also successful in computing the betweenness of a graph, but it was not without limitations. The extra dimension in the adjacency matrix required \( N^3 \) entries, in order to record the shortest distance as well as the intermediary nodes for all \( i, j \) pairs. This caused a severe limitation on the input size of a graph that could be allocated on the GPU. This memory limitation was the main motivator for discovering more efficient methods of optimizing a betweenness algorithm for the GPU.

Algorithm 1 Parallel Floyd-Warshall Betweenness Algorithm for GPU

1: procedure SHORTEST PATH KERNEL
2: initialization:
3: \( K \leftarrow \) intermediary node \( k \)
4: distance loop:
5: if \( \text{Distance[threadId.x][threadId.y][0]} > \text{Distance[threadId.x][K][0]} \)
6: \( +\text{Distance[K][threadId.y][0]} \) then
7: \( \text{Distance[threadId.x][threadId.y][0]} \leftarrow \text{Distance[threadId.x][K][0]} \)
8: \( +\text{Distance[K][threadId.y][0]} \)
9: goto updatePath()
10: updatePath:
11: for \( i \) in \( N \) do
12: while \( \text{Distance[threadId.x][K][i]} \neq \text{null} \) do
13: \( \text{Distance[threadId.x][threadId.y][i]} \leftarrow \text{Distance[threadId.x][K][i]} \)
14: \( \text{Distance[threadId.x][threadId.y][i]} \leftarrow K \)
15: for \( j \) in \( N \) do
16: while \( \text{Distance[K][threadId.y][j]} \neq \text{null} \) do
17: \( \text{Distance[threadId.x][threadId.y][i+j]} \leftarrow \text{Distance[K][threadId.y][j]} \)
18: syncthreads():

5.2 Adapting a Dijkstra’s based Betweenness Algorithm

Another approach would be to use a Dijkstra’s inspired algorithm, to compute the betweenness for each edge within the shortest path between a given \( i,j \) pairing. Other studies have created single source shortest path algorithms optimized for the GPU [13, 10] using Dijkstra’s algorithm, computing the shortest path to a source vertex by creating a working set, and greedily adding vertices to the working set by determining the closest vertex to the source via any vertex within the working set. For our implementation, our graph is assumed to be unweighted and undirected, although it
Figure 5.2: This sample graph consisting of five nodes is represented using two arrays. Array B contains the neighbor list for each node, colored accordingly. Array A contains an index for each node, such that the array entry at index $i$ contains the starting index within B that contains the neighbor list for node $i$. A node $i$ can determine the length of their neighbor list by subtracting index $i$ within A from index $i + 1$.

can trivially be extended to directed graphs. The data is stored as an adjacency list using two arrays, A and B, on the GPU. Array B is a one-dimensional compression of a standard adjacency list, containing the values of all neighboring nodes for each node ordered 0 to $N$, one after another. The overall length of array B is thus twice number of edges within the graph. Array A then contains a single entry for each individual node within the graph, containing the index location within B that lists the neighboring nodes for that individual node. Each index $i$ within A, thus contains the starting index within B that contains the adjacency list for node $i$. See Figure 5.2 for an illustrated example.

Once the initial graph information is transferred to the GPU, a kernel call is launched and the threads begin to compute the betweenness for each edge in the graph. Threads execute in blocks, with each block being responsible for a different source node, each one similar to Dijkstra’s approach. For a fully parallel calculation, it is optimal for a kernel call to be launched with a number of blocks equal to the
number of nodes within the graph. If, due to either memory constraints on the GPU or the number of blocks needed, a number of blocks allocated is less than number of nodes within the graph, separate kernels calls are launched until all source nodes are considered. Each block contains a set number of threads each being responsible for a different node with the graph. If the number of nodes is greater than the number of threads available per block, nodes are then assigned to threads in an even distribution such that each node is monitored by one thread.

Algorithm 2 Parallel Dijkstra’s Betweenness Algorithm for GPU

1: procedure BETWEENNESS KERNEL
2: initialization:
3: \( S \leftarrow \) source node
4: \( \text{Buddy[threadId]} \leftarrow -1 \)
5: \( \	ext{Weight}[S] \leftarrow 1 \)
6: \( \	ext{Weight[threadId]} \leftarrow 0 \)
7: buddy loop:
8: \( \text{while } \text{Weight[threadId]} \neq 1 \text{ do} \)
9: \( \text{for each neighbor } i \text{ do} \)
10: \( \text{if } \text{Weight}[i] = 1 \text{ then} \)
11: \( \text{Weight[threadId]} \leftarrow 1 \)
12: \( \text{Buddy[threadId]} \leftarrow i \)
13: \( \text{goto } \text{syncthreads}() \)
14: \( \text{syncthreads}() ; \)
15: weight loop:
16: \( \text{while } \neg \text{addedWeight do} \)
17: \( \text{if } \text{threadId} \notin \text{Buddy} \text{ then} \)
18: \( \text{Weight[\text{Buddy[threadId]}]} \leftarrow \text{Weight[\text{Buddy[threadId]}]} + \text{Weight[threadId]} \)
19: \( \text{Buddy[threadId]} \leftarrow \infty \)

In our algorithm, a single source node is treated as a sink and all information will eventually travel from each node to the source via the shortest path. As such, for a given source \( s \) and an intermediate node \( i \), if node \( j \) uses node \( i \) to reach node \( s \), then all other nodes that use \( j \) will also use \( i \). Thus, each node only needs to keep track of a single node that it will use to reach the source node. We refer to this intermediate node as the a ”buddy.” Each node is responsible for updating the betweenness of the edge between itself and its buddy.
Within each block, the shortest path from all nodes to the source node is computed. One node is designated a source node, and we maintain arrays for the buddy list and weight values, shared within each block. To generate the two lists, initially, the source node is evaluated and its weight is set to 1. We then compute, using our algorithm, the buddy list and weights for the entire graph from the single source. See Algorithm 2 for a pseudocode representation.

In the "buddy loop", each node attempts to add itself to the working set of nodes by determining the neighboring node that results in the shortest path the source node (Figure 5.3C). This neighboring node is referred to as the buddy node. Once the buddy loop begins, each thread that is responsible for an individual node, will check each of its neighbors to see if a neighbor is within the working set. If a node finds a neighbor within the working set, that neighbor could potentially be that node’s buddy. In the serial implementation of Dijkstra’s algorithm, only the node with the shortest distance to source node is added to the working set. When analyzing a weighted graph, each node computes its shortest distance to the source node using a neighboring node that is currently within the working set as an intermediary. The threads then compare their distances, and the node with the shortest distance is added to the set. However, if there are multiple nodes that share this smallest distance, each node may add itself to the working set during the same iteration, as we are guaranteed there will be no shorter distance, if there are no negative edge weights. In unweighted graphs, this comparison is even easier. If a node finds a neighboring node that is in the working set, that neighboring node can be used as the intermediate node, as nodes added to the working set in earlier iterations are closer to the source node than nodes added during later iterations. Once all nodes have determined their buddy, the algorithm moves to the weight loop.

Within in the "weight loop", each single source betweenness is evaluated in a
Figure 5.3: Example execution with a sample graph consisting of seven nodes. A) Two arrays depicting how the graph is stored on the GPU. Each index $i$ within array $a$ contains the index $j$ within array $b$ where the neighbor list for node $i$ can be found. B) Depiction of the sample graph, using colored nodes to depict the order in which they are updated within the algorithm.
reverse breadth-first manner from the outermost nodes to the source node (Figure
5.3D). Each node begins by scanning the buddy list to see if there are any nodes that
use it as an intermediary node to the source. If a node is not used as an intermediary
node, that node adds its weight to its buddy’s weight and sets a flag to denote to its
buddy that there is no longer a dependence on it. The nodes continue to check the
buddy list to determine if their dependencies have resolved and continue to add their
weight to its buddy’s until each node has updated their buddy’s weight value. We
illustrate our algorithm with a seven node graph example in Figure 5.3.

For the sample graph above, a kernel call would create 7 blocks, each computing
the betweenness for a different source node. The example shows the sample execution
for the buddy and weight loops using node 0 as the source node. Initially, only the
source node 0 has a weight of 1, denoting it is a part of the working set, while all other
nodes have a weight of 0. Within the first iteration of the buddy loop, only nodes 1
and 2 are able to add themselves to the working set, as they are the only two nodes
with a neighboring node, node 0, already within the working set. Their weights are
updated to 1, signaling to other nodes that they are within the working set, and their
buddy is recorded as node 0. Within the next iteration, nodes 3 and 4 are now able
to add themselves to the working set, recording their buddies as 1 and 2, respectively,
and setting their weights to 1. Finally, within the last iteration, nodes 5 and 6 add
themselves to the working set while recording node 4 as their buddy, and the loop is
able to terminate because all nodes are now within the working set. Within the next
step to the algorithm, each node will update the weight of their buddy node with
their own weight, starting with the leaf nodes of the graph. From the example, the
exterior nodes 3, 5, and 6 are the first to update their neighbors, and each add their
weight to the weight value of their buddy, while also setting a negative flag within
the buddy array. This flag is used to ensure that each node only updates the weight
of its buddy once during this process. The buddy information is retained though in order to update a global betweenness array after the weight loop has terminated, as described above. Once the exterior nodes have updated their buddies, nodes 1 and 4 are able to do the same, adding their weights to nodes 0 and 2 respectively. Finally, node 2 is able to update the weight of its buddy, node 0, and the loop terminates since each node has now updated the weight of its buddy. When a node adds its weight to the weight of its buddy, it notifies the buddy node how many times it is ultimately used as an intermediary node, allowing it to pass this information forward to its buddy node in the future.

The parallel betweenness algorithm we described above is for a single source betweenness calculation. In the final step, once all blocks have finished execution, each node updates the global all-points betweenness array by adding the betweenness of the edge between itself and its buddy to the global array. The global betweenness array is then copied back from the GPU to the CPU, and the edge betweenness values are printed.

5.2.1 Performance Analysis of Parallel Betweenness Algorithm

We compared our algorithm to the CPU implementation of the betweenness centrality algorithm, as implemented in the NetworkX software package, an open-source Python language tool for the creation, manipulation, and analysis of complex networks [38]. CPU calculations were performed on a single workstation utilizing an Intel i7 quad-core processor running at 3.2 GHz per core, and 16 GB of RAM. GPU calculations were completed with all calculations being performed on a single NVIDIA Tesla S2050 GPU, composed of 448 CUDA cores with a GPU clock rate of 1.15 GHz. For both the NetworkX CPU and our GPU implementations, we used input graphs generated by the NetworkX software consisting of $N$ nodes and $E$
edges. Values for $N$ ranged between 100 and 600, while $E$ was was adjusted to be between 20 and 40 percent of the total possible number of edges within the graph. Each graph was unweighted and undirected, and it was possible that the randomly generated graphs were not be fully connected. We observed an $N$-dependent speedup ranging from 150 to 450 across all input graphs (Figure 5.5). The overall increase in execution time as graph size increases is proportional between both the CPU and GPU algorithm. It can therefore be assumed that our GPU algorithm suffers from the same asymptotic complexity of the CPU betweenness algorithm.

We also analyzed the performance of our GPU-optimized betweenness algorithm with respect to the number of blocks concurrently executing on the GPU during the kernel call. In a GPU, the individual threads that are executed independently using the SIMT (single instruction, multiple thread) paradigm, and they are organized into blocks that are executed by a multiprocessing unit. The GPU hardware is designed such that there exists a finite number of stream multiprocessors (SMs) that can be executed concurrently. As we discussed above,
our algorithm computes the betweenness from all nodes to a single source in a single block, thus for maximum parallelization it would be beneficial to have $N$ blocks concurrently executing for a graph consisting of $N$ nodes.

We observed, that as the number of nodes increased, there was a diminishing return on the speedup observed by the parallel algorithm (Figure 5.4). Using an input graph of 5,000 nodes, we observed that the execution times using different number of blocks, ranging from a single block to the maximum of 5,000 blocks (Figure 5.6). While we observe a significant decrease in the execution times as the number of blocks, after about 10 blocks, we observe only a modest decrease in execution time with increasing number of blocks. This is most likely attributed to the number of SMs available on the GPU that can be executed concurrently. One way to address this
hardware limitation is to divide the load across multiple GPUs, which would increase the number of SMs available for calculations.

5.3 Extending Betweenness Algorithm to Perturbation

There are many instances where hub nodes, having large amounts of betweenness, are important nodes to a given system [49, 17]. Calculating the node betweenness does not necessarily indicate the impact that an individual node has on the network though as a whole. Even if one hub node is deleted, there may be a secondary hub node that may alleviate the impact on the shortest path between all points. Node perturbation is the measure in the change of edge betweenness, when a node is removed from a network. When hub nodes that are the main source of communication between opposite parts of the graph are removed from the graph, more signals may be redirected through multiple nodes making the distance between nodes much larger, than when the hub node existed within the graph. This change in distance, is something that cannot be measured by solely looking at the node betweenness of the hub node. Perturbation attempts to measure the change in communication pathways between nodes, as opposed to only recording the number of nodes that use a hub node as an intermediary node.

In order to calculate the perturbation of a given graph, first the betweenness of the fully intact graph must be computed. These values are used as control values for the remainder of the calculation. Then, for each node within the graph, a kernel call is made where a node value is passed to the GPU. This value is the node that is to be removed from the graph. The same calculation is then performed using the above parallel Betweenness algorithm, but the thread responsible for the node to be deleted, never adds it to the working set. Without ever getting added to the working set, no other node will use it as a intermediary node to the source node. When updating
Figure 5.6: Execution time as the number of blocks per kernel call is increased. As each block computes the betweenness for a single node as the source, a fully parallel implementation would consider all nodes simultaneously using $N$ blocks given a graph with $N$ nodes. This graph demonstrates diminishing returns as the number of blocks exceeds eight.

The betweenness for each edge, the thread responsible for the node to be deleted will never update its betweenness because it never computed a buddy node to the source. Once the betweenness is computed with the node removed from the graph, the new betweenness values are compared to the previously found control values, and the absolute difference in betweenness for each edge is then summed together and a final perturbation value is recorded for the deleted node. The algorithm benefits from each kernel call computing a node to be deleted being independent from one another, after the control betweenness values have been computed. This means that multiple GPUs can be used to perform these calculations independently from one another and then combine their results once the overall calculation is completed, resulting in an overall time complexity equal to the betweenness algorithm given that there are enough GPUs to distribute a single kernel call to each coprocessor.
Proteins are responsible for thousands of biological functions that occur constantly in all organisms in the world, and they have been the focus of many studies in order to learn their behavior and purpose in biological processes \[146, 15\]. Proteins are large macromolecules comprised of one, or more chains of amino acids linked together, with each protein varying in size. Proteins form specific structures that can form communication networks within themselves to carry out their biological functions. These networks are formed by the amino acids interacting with each other. In our study, we abstract several proteins into graph data structures, and apply the previously explained graph analyses on these proteins in order to correlate these statistics to protein dynamics.

6.1 Abstracting a Protein as a Graph

Each atom within the protein has a position in Three-Dimensional space, and MD simulations simulate the change of position of these individual atoms over multiple time steps. The individual positions of these atoms are calculated by computing the force that each atom exerts on each other atom, and along with the trajectory of an atom at any given time step, the position of each atom is updated for the next time step. MD simulations are performed for an arbitrary amount of time, but it is possible to restrict the simulations to only the time it takes the protein to reach a stable state. The output of these simulations is a list of position coordinates for each atom within the protein, at each time step within the simulation. For our studies, we used the position coordinates for each residue within the protein, with each residue consisting
Figure 6.1: Edge frequency during an MD simulation of the MetRS:tRNA$^{Cys}$ Protein:RNA complex. The MetRS is shown in a standard protein ribbon diagram (magenta) and the location of each constituent residue is a silver spherical node. The tRNA$^{Met}$ is shown in a standard cartoon diagram (blue); it is transparent because it was included in the MD simulation but not analyzed. The interaction frequency between pairs of residues across all of the time steps during the MD simulation is represented as a straight line edge colored from blue (low frequency) to red (high frequency).

of one or more atoms. The position of the residue was achieved by averaging the positions of all atoms within the residue.

To directly apply our perturbation measure to a biomolecular system, we performed MD simulations of protein:RNA biomolecular complexes that have been well studied using experiments. Proteins and RNA are linear chains that consist of constituent residues and nucleotides, respectively. The residues and nucleotides can form interactions with each other if they are in close proximity. Proteins and RNA are also dynamic, and they can move based on the interactions. A well-established method of determining how biomolecules move based on their interactions is MD simulations.

We performed MD simulations of the CysRS:tRNA$^{Cys}$ complex using the CHARMM36 force field [27], and the NAMD simulation suite [43]. We used explicit water MD
Figure 6.2: Edge frequency during an MD simulation of the CysRS:tRNA\textsuperscript{Cys} Protein:RNA complex. The CysRS is shown in a standard protein ribbon diagram (magenta) and the location of each constituent residue is a silver spherical node. The tRNA\textsuperscript{Cys} is shown in a standard cartoon diagram (blue); it is transparent because it was included in the MD simulation but not analyzed. The interaction frequency between pairs of residues across all of the time steps during the MD simulation is represented as a straight line edge colored from blue (low frequency) to red (high frequency).

simulations, using a standard protocol. The MD simulations resulted in a 50 ns trajectory "movie" resulting in 50,000 individual "snapshots" that had the coordinate of the residue Cartesian positions. For the CysRS protein, used in our analysis, there are 461 residues. We defined an interaction between residues if a pair of residues in CysRS was within 8.0 Angstroms of each other. To represent the structure as a graph, the residues were abstracted as the nodes, and the interactions were the edges. Since residues are linked together in a linear chain, we always defined an interactions between consecutive residues.

Using this method, each time step of the simulation can be analyzed as an unweighted graph, where each edge is an interaction. For the \( N \) nodes in our graph, there can be at most \( O(N^2) \) edges. We counted the number of times an edge occurred during the MD simulation and normalized each of these values over the number of time steps within the simulation to compute the edge frequency (Figures 6.1 and 6.2).
We subsequently performed our betweenness perturbation calculation on the MD simulations of the CysRS:tRNA$_{\text{Cys}}$ complex for each of the 461 residues. Ghosh et al. performed experimental mutagenesis studies of the CysRS:tRNA$_{\text{Cys}}$ complex, where individual residues were replaced, and observed the change in the kinetics rates of enzymatic reactions [19]. In their analysis, they measured two kinetic rates, $K_m$ and $k_{\text{cat}}$, after each mutation. A high value of the kinetic rates indicates that the time for the enzymatic process to occur is higher. Therefore, a mutation at a critical residue would be expected to result in a longer time required to perform its enzymatic function. This is analogous to the removal of a critical node in a graph, resulting in a high perturbation in our analysis.

In order to abstract each protein within our study as a graph, we analyzed the positions of each residue within the protein at each time step of the simulation. Since a graph is comprised of a set of vertices and a set of edges, each residue was considered to be an individual vertex within our graph representation of the protein. Edges were defined within the graph between two vertices if two residues were able to interact with each other, at a given time step. For our study, we determined that two residues were able to interact with one another as long as the distance between two residues was equal to or less than eight angstroms. As residues are linked together within a chain, sequential residues were always considered to have an edge between them, despite the distance between said residues. Since the residues change position between time steps, it is possible for edges to be removed in later time steps while others will be added as residues come closer and closer to one another. Analyzing each time step of the MD simulation presented a weighted graph where each edge had a weight ranging between 0 and 1. These weights represented the percentage of time that the edge existed within the simulation. Edges with a weight of 1 existed within every time step of the simulation, while edges that had a weight of 0 never existed within the
simulation, and were not included in the graph. Individual time steps could also be analyzed in order to create an unweighted representation of the graph, where each edge is a binary interaction between residues.

6.2 Perturbation Analysis of Protein Residues

Mutation experiments have been performed previously in vitro, analyzing the change in protein dynamics as proteins are mutated prior to folding, by removing selected residues from the protein [19]. These experiments are performed in an attempt to identify the individual residues that are the most important to the communication networks of a given protein necessary to carry out its function. These studies measure the catalytic efficiency of the protein by measuring three values, $K_m$, $k_{Cat}$, and the ratio between the two, $k_{Cat}/K_m$. The value of $K_m$ is a constant value that represents the concentration of substrate needed for an enzyme to operate at half of its maximum velocity. $k_{Cat}$, on the other hand, represents the ratio of the maximum velocity of the enzyme to a given concentration, describing the amount of substrate that can be converted to product over time. This leaves the final value of $k_{Cat}/K_m$ representing the catalytic efficiency, or how fast the enzyme reacts to the substrate after encountering one another. In order to perform these experiments in vitro, it takes time to perform the experiments and analyze the results, unfortunately there is no easy way to determine which residues will have the strongest impact on these values, or which ones will affect the catalysis the most. With some proteins consisting of only a few residues, the proteins used in this study were composed of several hundred residues, thus it is impractical to perform these mutation experiments on each residue within the protein.

By comparing the perturbation values of individual residues to these catalytic efficiency measurements, our goal is to create a predictor for which residues have the
greatest impact on the catalysis. The perturbation value thus does not replace the three aforementioned values, but may describe which residues are key targets for these experiments, as the resources to perform these experiments are limited. By analyzing a protein using graph theory metrics, we assume that during the folding process, residues must communicate with one another in order to successfully reach a stable state. The structure as a whole must communicate, so signals must be sent from each residue to every other residue, with 8 Angstroms being the maximum distance that a signal can be sent. By generating these perturbation values, we are calculating the latency created in signal sending when residues are removed. Two proteins were analyzed using this metric, the first being MetRS with the second being CysRS. The proteins differ from one another, as MetRS is composed of 546 residues and no binding partner, while CysRS consists of 461 residues and was analyzed with its binding partner during the folding process. The results differed for each protein across each experiment performed. This could be attributed to the fact that CysRS does have a binding partner, while a binding partner for MetRS is yet to be found.

Figure 6.3: Comparison of the experimentally observe kinetic rate, $K_m$, and our perturbative values. A) Correlation between $K_m$ values during mutation experiments versus perturbation value, and with the best-fit line and correlation coefficient. B) The same data plotted for each residue.
Perturbation results for MetRS showed a relatively large disparity across all residues within the protein. As the size of each graph is different, perturbation results cannot be compared across different graphs, but the values are relative to other nodes within the same graph. MetRS showed seven residues with perturbation values over 100,000. These values referred to residues 12, 22, 51, 53, 132, 235, 491. Comparing these large perturbation results to other residues, the smallest values for perturbation ranged between 3,000 and 4,000 and over 80 percent of all residues had perturbation values less than 25,000. Comparing these values though to the $K_m$, $k_{Cat}$, and $k_{Cat}/K_m$ values of in vitro experiments though showed that there was little, if any correlation, between these perturbation results and the observed catalytic metrics.

The second protein CysRS showed more promising results, but with limited in vitro data, it is hard to determine whether or not our perturbation analysis has a true correlation to the catalytic metrics. Performing the perturbation analysis on CysRS revealed an even greater disparity in perturbation values between residues. The highest values recorded were by residues 32 and 405, with perturbation values of...
Figure 6.5: Comparison of the experimentally observe kinetic rate, $k_{\text{Cat}}/K_m$, and our perturbative values. A) Correlation between $k_{\text{Cat}}/K_m$ values during mutation experiments versus perturbation value and with the best-fit line and correlation coefficient. B) The same data plotted for each residue.

98,370 and 91,725, respectively. While these values are not as high as values observed in MetRS, it is expected because of CysRS being a small protein, and thus represented using a smaller graph. The smallest value for perturbation was a mere 43 by residue 454, with almost two-thirds of all residues showing perturbation below 10,000. This greater disparity and fewer number of identified hub residues showed that the perturbation results for both proteins varied greatly between the two of them. For CysRS there were six residues that had observed in vitro results for the $K_m$, $k_{\text{Cat}}$, and $k_{\text{Cat}}/K_m$ metrics, with those residues being 40, 42, 294, 297, 354, and 427. While some of the residues showed a strong correlation across all metrics, not all residues show that same weight of correlation. Residue 294 was an outlier in all three metrics,
along with residue 427 being an outlier and showing no correlation in the $K_m$ metric (Figure 6.3) while residue 40 showed no correlation in the $k_{Cat}$ and $k_{Cat}/K_m$ values (Figures 6.4 and 6.5). For the remaining three residues, there were trend lines displayed with $R^2$ values in the range of 0.99, a near one to one correlation. Residue 427 displayed this same correlation in the $k_{Cat}$ and $k_{Cat}/K_m$ comparisons, while residue 40 showed the same correlation in the $K_m$ value. While the results for three residues were promising, with two of the residues showing good results in different categories, the limited data does not support that perturbation has a direct correlation with these catalytic efficiency values. We are unable to explain why this metric performs better for some residues and not for others, or why there is strong correlation in one metric versus the other. It has been proposed that the individual function of the residue may impact these results, with residues of different functions producing different values for $K_m$, $k_{Cat}$, and $k_{Cat}/K_m$, but to justify that perturbation is linked with these metrics would require further analysis and larger sets of data.

6.3 Community Analysis of Residue-Residue Interactions

Community detection is used to identify similar nodes, based on the degree of each node and the neighbors of each node. Nodes within the same community are often thought to be closely related, and in the life sciences, studies have been performed to analyze communities in biological systems. On a higher level, community analysis has been performed on protein-protein interactomes in order to identify proteins that share the same biological process [46, 15], the goal being that knowing the answer for one protein will give the answer to all other proteins within the same community. Community analysis has been done within individual proteins, and for the aforementioned proteins, MetRS and CysRS, distinct communities have been determined through experimental results and analysis. In this analysis, we validate the commu-
Figure 6.6: Community analysis of protein CysRS. A) Results of Girvan-Newman analysis protein. Results showed there to be 12 communal structures found within the structure compared to the B) five domains previously observed through other classification. Many of the communities within A seem to be sub-components of elements within B.

Communities determined through previous *in vitro* experiments and identify communities in a protein not previously studied, 1a6f Ribonuclease-P (RNAse-P).

For this analysis the Girvan-Newman algorithm was performed on our graph representations of MetRS and CysRS, as well as the new RNAse-P protein. For MetRS, several domains have been previously determined based on the functions of different strands within the protein chain. A total of four domains have been identified within MetRS, and include the catalytic domain (CD), connective peptide (CP), stem contact fold (SCF), and anticodon binding domain (ABD). The CD is comprised of residues 1 through 99 and with residues 251 to 325. The CP domain consists of residues 100 through 250 while the smallest domain, the SCF domain, is the chain of residues 326 to 387. The ABD domain consists of the residues numbered form 388 to 547. The results of the Girvan-Newman algorithm detected seven communities within MetRS, ranging across all domains (Figure 6.6). There was only one community that was
Despite being larger than CysRS, there have been fewer observed communal structures within MetRS. Though the A) seven communities found with the Girvan-Newman analysis do not correlate as well with the B) four communal structures previously observed as well as within the CysRS analysis, there is still some correlation between the two analyses.

All known domains were subdivided into small communities, with many communities spanning multiple domains. These results do not support nor disprove the previously studied domains identified within MetRS, but only suggest that some changes could be made to what residues are considered to be in which domain.

Even though CysRS is smaller than MetRS in the total number of residues, there have actually been more domains identified within CysRS compared to MetRS. For CysRS there have been five domains identified, including CD, CP domain, SCF, HBD, and ABD. The CD domain for CysRS consists of the residues 22 to 131, and residues 208 to 254. The CP domain contains the residues from 132 to 207, and the SCF domain is composed of the residues from 255 to 325. Finally, the last two domains, the HBD and ABD, are composed of the residues 326 to 400 and 388 to 547, respectively. The analysis of the Girvan-Newman results revealed there to be
The Girvan-Newman analysis of RNAse-P identified seven communities despite its smaller size. Without an empirical comparison though we can only carry these results forward into future work as a basis for communal detection.

12 communities, more than doubling the number found through experimental results. The communities detected through the Girvan-Newman analysis fall much more inline with the previously found domains in CysRS (Figure 6.7). Many of these domains were subdivided into two or more communities, meaning that these domains could potentially be further subdivided, and that the current assumption of the domain composition may not be optimal. Even though we may be able to justify that the number of domains currently perceived may not be optimal, we can also find some justification that the Girvan-Newman analysis along with our graph representations can be used to detect communities within specified proteins.

Finally the last protein that our community analysis was performed was RNAse-P, the smallest of the three, composed of only 113 residues. Domains had not been previously identified in RNAse-P, and so our analysis is the first attempt to identifying these communities within the protein structure. Our analysis found that there were seven communities within RNAse-P, despite its smaller size (Figure 6.8). The largest community consisted of 25 residues, almost a quarter of the total number of residues.
within the protein. The smallest community consisted of only seven residues with all other communities having roughly 10 to 15 residues within them. While there are no experimental results to compare these results to, these results will be used in future research endeavors in order to justify our graph representations of proteins, and to justify the Girvan-Newman method for detecting domains within protein structures.

The comparisons between graph analysis metrics and the in vitro experiments on protein folding dynamics have provided some promising results, but the results so far are still inconclusive as it will still take further analysis and experimental data to make a valid claim that there is a correlation between in vitro experiments, and perturbation or community detection algorithms, that are found using our graph abstraction model. While our model can be changed, there are still a lot of variables to consider as each protein is diverse and the individual residues within these complex macromolecules are equally as complex. It will be interesting to see how further study into graph analysis on protein dynamics fares, and the results that come about from it.
Chapter 7: Conclusions

Optimizing graph algorithms for execution on the GPU is hard work. Graphs are non-uniform data structures with many of the algorithm not easily lending themselves to parallel solutions. The single instruction multiple data architecture of the GPU thus does not create an easy outlet for these load imbalanced jobs. While graph algorithms are prime targets for speedup as their CPU algorithms are asymptotically the slowest that does not mean that there is always an easy solution to find the optimal way to compute these graph metrics. In this thesis, we present new GPU-optimized parallel graph algorithms for connected components, shortest path, and graph traversal but many of the algorithms left to be explored are these algorithms such as betweenness and community analysis that are compounded algorithms of these smaller graph analysis. I expect in the near future for more people to explore parallel solutions to these more complex graph problems, along with problems such as min cut max flow, and detecting sink components.

After attending the GPU Technology Conference last spring, there were many talks about the the optimization of graph algorithms for applications on the GPU, but there were very few research endeavors devoting GPU computation for the scientific community. I see GPU computing to still be primarily focused on computations related to computer graphics and image processing as that is what the GPU was originally intended to do. Many of the industry leaders have shifted their focus towards these graphics programmers, and many of the latest optimizations for GPUs have been made with their goals in mind. Unless the focus is also highly directed towards scientific applications too, high performance scientific computing experts will have unnecessary technical hurdles for implementing their algorithms on the GPU.
While there are still many scientific applications that could benefit from optimizing parallel solutions on GPUs, many of the interesting problems found in science are too big to solely run on a single or even a few GPUs. The world’s biggest supercomputing clusters use thousands of GPUs, but they are also utilizing thousands of the fastest CPU hardware on the market. I see parallel computations taking a more hybrid approach of utilizing the benefits of both the CPU and GPU in order to achieve the fastest speed up available for these larger problems.
Bibliography


62


[38] NetworkX. High-productivity software for complex networks. [https://networkx.github.io/](https://networkx.github.io/)


Cody A. Stevens

5099 Winster Drive Apt. 304
Winston-Salem, NC 27103
stevca9@wfu.edu
(301) 514-1617

EDUCATION

*Wake Forest University*, Winston-Salem, NC, May 2015

Master of Science in Computer Science, Area of Specialization: Parallel Computing

Thesis: "GPU Optimized Graph Theory Analysis of Allosteric Protein Signaling Networks"

GPA: 3.9

*Wake Forest University*, Winston-Salem, NC, December 2012

Bachelor of Science in Computer Science

RESEARCH EXPERIENCE

*Research Assistant*, Dept. of Computer Science, Wake Forest University, Fall 2012 – Present

Research Advisor: Dr. Sam Cho

- Developed C code using the CUDA library to perform Graph Theory Analysis on NVIDIA Graphics Processing Units
• Applied Graph Analysis to Molecular Dynamic Simulations of protein folding for \textit{E. coli} Methionyl-tRNA synthetase (MetRS), CysRS:tRNA^Cys (CysRS)

• Conducted studies using NVIDIA Graphics Processing Units for digital currency mining using script-based proof of work

\textbf{WORK EXPERIENCE}

\textit{Unix Systems Administrator}, Information Systems, Wake Forest University, Fall 2014 - Present

• Perform daily operational tasks regarding user and server management

• Experience writing puppet modules for server automation, including hierra

• Experience with website hosting, SSL certificate authorities, DNS

• Installed physical hardware to build new systems for Red Hat Enterprise Linux 6, was responsible for decommissioning old hardware

\textit{HPC Intern}, Information Systems, Wake Forest University, Summer 2014

• Experience with high performance cluster administration on the Wake Forest University DEAC cluster

• Updated CUDA language libraries on DEAC Cluster GPU compute nodes

• Created a virtual machine test environment for cluster emulation, experience with general UNIX system administration practices

• Research and Development of SLURM scheduler to replace past TORQUE/MAUI scheduler on Red Hat Enterprise Linux 6
TEACHING EXPERIENCE

Teaching Assistant, Data Structures and Algorithms I, Wake Forest University, Spring 2013

• Met with students individually to answer questions and provide additional tutoring
• Assisted in grading of project assignments

Teaching Assistant, Computer Systems, Wake Forest University, Spring 2013

• Attended weekly lab to assist students and help introduce new topics
• Met with students individually to answer questions and provide additional tutoring
• Assisted in grading of lab and homework assignments

GRANTS and AWARDS

Center for Molecular Communication and Signaling Research Grant, Fall 2013 - Spring 2014

Wake Forest University Graduate School Summer Research Grant, Summer 2013
PROFESSIONAL AFFILIATIONS

Member, Upsilon Pi Epsilon Honor Society, Epsilon Chapter, Spring 2014 - Present

Member, Association for Computing Machinery, Spring 2014 – Present

Member, Alpha Phi Omega, Kappa Theta Chapter, Fall 2009 – Fall 2013

PUBLICATIONS and PRESENTATIONS


”GPU Optimized Allosteric Communication Network Analyses of Molecular Dynamic Simulations.” Poster presented at the NVIDIA GPU Technology Conference, March 24-27, San Jose, California

”Graph Theory Analyses of Molecular Dynamic Simulations.” Poster presented at the Wake Forest University Center for Molecular Communication and Signaling Research Retreat, October 11, 2013, Winston Salem, North Carolina

REFERENCES

References available upon request.