General-Purpose Computing on Graphics Processing Units: GPU Processing of Protein Structure Comparisons

Todd Sullivan, Heather Nelson, Travis McBee, Mathew Alvino
Mentor: Dr. Chi-Ren Shyu

CS 4970 – Senior Capstone Design I
Technical Report
Winter 2007

Table of Contents

Executive Summary 1

1 Problem Definition 2

1.1 Introduction 2

1.1.1 Needs Analysis 2

1.1.2 Market Analysis 2

1.2 Technical Background 3

1.2.1 Index-based Protein Substructure Alignments 3

1.2.2 General GPGPU Techniques 4

1.2.3 Current GPGPU Technologies 6

1.2.3.1 Accelerator Performance 7

1.2.3.2 BrookGPU Performance 8

1.2.3.3 Cg 10

1.2.4 GPGPU Performance Techniques 10

1.2.4.1 Cache-Efficient Memory Models 11

1.2.4.2 Vertex Processor Code Motion Technique 11

1.2.4.3 GPU Clustering 13

1.3 Goals and Objectives 14

1.4 Overall Approach 15

1.4.1 System Diagram 17

1.4.2 Advantages and Disadvantages 19

1.4.3 Costs 19

2 Requirements Analysis 20

2.1 System Requirements and Constraints 20

2.1.1 Operating Environment 20

2.1.2 Market Users and Characteristics 21

2.1.3 Environmental Constraints 22

2.1.3.1 Quality and Reliability 22

2.1.3.2 Human Factors 22

2.1.4 System Components 24

2.1.5 Software Interfaces and Libraries 24

2.1.6 Communication Interfaces 25

2.1.7 Hardware Interfaces 25

2.1.8 System Maintenance 26
2.2 Performance Requirements 26
2.3 Resource Requirements 27
2.3.1 Time 27
2.3.2 Resources 28
2.3.3 Facilities 28
2.3.4 Budget 28
2.4 Evaluation Metrics 28
3 Design Specification 29
3.1 Hardware 29
3.2 Data Requirements 29
3.2.1 IPSA Data Sources 29
3.2.2 Translation to a GPU Datatype 30
3.2.3 Matrix Translations 30
3.2.4 Protein Chain Translations 32
3.3 Software 33
3.3.1 IPSA Profiler 33
3.3.1.1 Class Diagram 34
3.3.1.2 Data Flow Diagram 35
3.3.2 Java-C++ Interaction 36
3.3.2.1 High-Level Process Diagram 38
3.3.3 Alternative Java-C++ Interaction 39
3.3.3.1 Alternative High-Level Process Diagram 39
3.3.3.2 Advantages and Disadvantages 40
3.3.4 The Compute_RMSD_D1 Function 40
3.3.4.1 Algorithm 41
3.3.4.2 Eigenvalues and Eigenvectors 42
3.4 Testing Methods 42
3.4.1 IPSA Profiler 43
3.4.2 Java-C++ Interaction 43
3.4.3 The Compute_RMSD_D1 Function 43
3.5 Scheduling and Task Assignments 44

4 System Implementation 46
4.1 GPGPU Limitations Demonstration 46
4.1.1 2D Arrays versus 1D Arrays 46
4.2 GPGPU Potential using Cg 47
4.2.1 Mathematical Computation 47
4.2.2 Average Random Walk Distance 49
4.3 IPSA 51
4.3.1 IPSA Profiler 52
4.3.2 Java-C++ Interaction 52
4.3.3 The Compute_RMSD_D1 Function 53
4.3.3.1 CPU Arrays to GPU Arrays 53
4.3.3.2 Code Translation 54
5 System Performance and Evaluation 58
5.1 GPGPU Limitation Demonstrations 58
5.1.1 2D Arrays versus 1D Arrays 58
5.2 GPGPU Potential using Cg 59
5.2.1 Mathematical Computation 60
5.2.2 Average Random Walk Distance 61
5.3 IPSA 62
5.3.1 IPSA Profiler 62
5.3.2 The Compute_RMSD_D1 Function 64
6 Summary and Conclusions 65
7 Future Work 66
8 References 67

Executive Summary

The project's purpose is to learn about General Purpose Computing on Graphical Processing Units (GPGPU), determine the feasibility of porting an algorithm designed by a University of Missouri-Columbia PhD student, and develop prototypes that compute a portion of IPSA on the GPU and demonstrate using GPU-based computations within IPSA. The algorithm, known as Index-based Protein Substructure Alignment (IPSA), has an average response time of twenty minutes. Dr. Shyu hopes that the team's research into GPGPU technologies will show that GPU-based versions of his biology algorithms can perform faster than the current CPU versions and, in particular, that the GPU-based portion of IPSA is faster than its CPU counterpart.

The team's study of GPGPU technologies and analysis of IPSA reveals that only 9.3% of IPSA's total processing time is parallelizable. The remaining 90.7% of the total processing time is consumed by sections with an abundance of branching and extensive nested looping, both of which are extremely slow on the GPU. Of the GPU-compatible 9.3%, the team created a GPU-based prototype of the Compute_RMSD_D1 function, which consumes 7% of the total processing time.

The GPU-based prototype calculates 102,400 chain comparisons simultaneously. The prototype is 9.828 times faster than the CPU-based computation. Without including any potential overhead from connecting the GPU-based computation with IPSA, the GPU-based IPSA algorithm is 1.076 times faster than IPSA and cuts 84 seconds off the total processing time. Any additional performance improvements require a redesign of IPSA that increases the amount of parallelizable code.

1 Problem Definition

1.1 Introduction

General-purpose computing on graphical processing units (GPGPU) is an emerging method for achieving high performance gains on various computing problems such as database operations, neural networks, and physics-based simulations. The higher number of cores and the more efficient processing of complex mathematical calculations on GPUs in comparison with CPUs makes GPGPU a fascinating new method for deploying algorithms. In an effort to reduce the runtime of protein-based search and retrieval algorithms, Dr. Chi-Ren Shyu of the University of Missouri-Columbia Medical Biological Digital Library Research Lab (MBDLRL) has asked the team to convert MBDLRL algorithms into comparable algorithms for processing on a GPU.

1.1.1 Needs Analysis

In particular, the team will study the potential for GPU-based MBDLRL algorithms and develop a prototype that implements a portion of PhD student Pin-Hao Chi's Index-based Protein Substructure Alignment algorithm (IPSA) [1] on the GPU. IPSA is part of the MBDLRL's Protein Database Search Engine project (ProteinDBS) [2]. The algorithm currently takes twenty minutes on average to complete one query. Dr. Shyu hopes that the team's research into GPGPU technologies will show that GPU-based MBDLRL algorithms can perform faster than the current CPU versions.

1.1.2 Market Analysis

The development of GPU versions of MBDLRL algorithms will benefit the study of proteins and various medical biology fields. Reducing processing time will allow ProteinDBS to service more queries. Protein retrievals, folding, and other algorithms will be more efficient and thus have a higher availability.

The experience the team will acquire and the framework that the team will develop is also applicable to many other research areas. Since GPGPU methods are still very new, the framework and documentation will provide a base for GPGPU applications in all computationally intensive research. Thus, the benefits of understanding GPGPU methods and developing a solid framework for converting algorithms to run on the GPU will have benefits beyond the protein retrieval algorithm that the team will study. In fact, InformationWeek chose the field of GPGPU as one of the five disruptive technologies to watch in 2007 [3].

1.2 Technical Background

The technical background includes Index-based Protein Substructure Alignments [1], general GPGPU techniques [4], current GPGPU technologies, and GPGPU performance techniques.

1.2.1 Index-based Protein Substructure Alignments

Through the use of computer algorithms, biologists have the ability to solve many computationally challenging problems. For example, one area of interest in molecular biology surrounds protein structure matching. Proteins, the essential building blocks of organisms, fold into complicated three-dimensional structures. Each structure has a unique shape that often determines its biological function. IPSA presents a fast structure retrieval system to find similar proteins in a database of over 40,000 elements [5]. With such a large dataset, speed often becomes an issue since biologists need to have access to a system that provides results in real time. As such, IPSA achieves much higher efficiency than its well-recognized competitors, DALI and CE. IPSA is 37.66 times faster than DALI and 2.78 times faster than CE.

Although IPSA performs structure retrieval much faster than many other available systems, the algorithm could benefit from additional increases in speed. By looking at overall

structure and important substructure elements, such as the commonly occurring alpha helix and beta sheet, the system can achieve higher accuracy than its predecessor, ProteinDBS. However, an average response time of twenty minutes places a burden on the system and proves an inconvenience to researchers. The project team will examine the data structures and algorithms used to determine how they may be processed using a GPU in order to increase efficiency.

IPSA features many large data structures that facilitate fast protein structure retrieval. Each protein is considered as a sequence of terms. The algorithm maps substructure representatives into an M-tree. Then, an inverted-protein index organizes the terms associated with a protein by considering topology constraints. Using this database, the system can easily map query proteins to other similar structures. The team could potentially achieve an increase in speed simply by moving IPSA's large data structures such as the M-tree and inverted-protein onto the GPU. The team outlines areas with GPU-based computation potential in Section 5.3.1.

1.2.2 General GPGPU Techniques

Modern day GPUs specialize in performing large floating-point matrix arithmetic in small amounts of time. Much of the power of the GPU has evolved due to the switch from hardwired pipelines to programmable components. For example, the NVIDIA GeForce FX replaced register combiners with programmable pixel shaders. GPUs also specialize in parallel processing, and it is because of this specialization that they are so effective in GPGPU. For example, modern GPUs can compute 330 billion floating-point operations per second. However GPGPU is not the best choice for many algorithms in their original form. Many of these algorithms have to be reworked or have a translator made to function on a GPU. [6]

GPGPU techniques must follow the stream programming model to make use of the parallelism within GPUs. GPGPU applications implement several basic operations of the stream

programming model. These operations are map, reduce, scatter and gather, scan, stream filtering, sort, and search.

Map and reduce are two simple operations. Map is essentially applying a function to a given set of data elements. For example, if one has a stream of data in the range [0.0, 1.0), then a map operation can convert the values into the range [0, 255] by multiplying each element by 256 and then taking the floor of the result.

Reduce computes a smaller stream from a larger stream. An example application of reduce is when calculating the largest element from a dataset. A fragment program reads two values and writes the largest one to the pass' result buffer. The process continues in parallel until the stream contains only one element, which is the dataset's maximum.

Scatter and gather are read and write operations that access memory indirectly. Scatter is the operation for assigning a value to an element of an array. Gather is the process of retrieving an array element's value. Gather is implemented on a GPU by fetching a pixel from a texture using specific texture coordinates. Due to the design of GPU hardware, a fragment program cannot implement scatter. Instead, developers use several tricks to implement scatter such as rewriting the problem in terms of gather or using the vertex processor for scatter operations.

The scan and stream filtering operations are also helpful tools. Given an operator + and an array of n elements $[a_0, a_1,\ldots, a_{n-1}]$, scan returns the array $[a_0, (a_0 + a_1), \ldots, (a_0 + a_1 +\ldots+ a_{n-1})]$. The stream filtering operation selects a subset of the overall dataset and only processes on the subset. This technique is useful for simple data partitioning and collision detection.

Sorting operations are difficult to implement on fragment processors in current GPUs since GPU hardware does not implement scatter and data-dependent operations are difficult to implement in parallel processors. Thus, all GPU-based sorting is impartial to the input data.

Many sorting operations are significantly faster on the GPU than on the CPU. For example, PBSN, a cache-efficient GPU-based sorting network algorithm, is almost six times faster than the CPU-based Qsort available in the Intel compiler.

Searching operations on the GPU are not able to beat CPU algorithms in terms of reducing the processing time of a single search. Instead, the parallelism of the GPU allows developers to increase throughput by processing multiple searches at the same time. Most basic searching algorithms such as binary search are possible on the GPU, but algorithms that are more complex require specialized GPU data structures.

1.2.3 Current GPGPU Technologies

The base technologies for converting algorithms to run on the GPU are the OpenGL Shading Language (GLSL) [7], DirectX [8], and Cg [9]. RapidMind [10] and PeakStream [11] both offer commercial products for deploying high-performance applications on a variety of processors including CPUs, GPUs, and the Cell, while NVIDIA is developing a technology called CUDA [12] that is specifically for GPGPU. Other languages and extensions that are available include Sh [13], Shallows [14], Accelerator [15], and BrookGPU [16].

GLSL, DirectX, and Cg are all APIs and toolkits for writing graphics programs. These APIs are the base for programming on popular video cards made by ATI and NVIDIA. GLSL is an open standard while DirectX is a proprietary Microsoft product and Cg is a proprietary NVIDIA product. Even though Cg is proprietary, its compiler supports OpenGL's standards and thus can create programs that run on both ATI and NVIDIA video cards. The basis for using GLSL, DirectX, or Cg for GPGPU is that one can hijack the graphics processing pipeline and force the GPU to perform other tasks instead of rendering graphics. All other languages,

extensions, and commercial products convert code into GLSL or DirectX methods during the compilation process.

RapidMind and PeakStream both offer commercial products for taking advantage of various types of processors. RapidMind is comprised of the former developers of Sh. Both companies provide APIs for C++. Unfortunately, the products are too costly for the project. For example, PeakStream's API costs $1,000 per GPU on an academic server and $295 per user on an academic workstation.

NVIDIA's CUDA technology is a new technology that enables standard C programming on a GPU. The technology supports both Linux and Windows XP operating systems and includes native multi-GPU support. While CUDA appears to be the most promising technology for future GPGPU applications, the technology is currently in closed testing and is not available for this project.

Sh and Shallows are C++ libraries that in simplest terms provide a high-level API above GLSL that allows developers to spend less time with the GLSL-specific programming details. Accelerator and BrookGPU are both toolkits for taking advantage of GPU processing power without requiring developers to learn GPU specific APIs or programming languages. The team is interested in the applicability of Accelerator and BrookGPU to the project because both technologies significantly reduce the time required to create GPU programs.

1.2.3.1 Accelerator Performance

Accelerator is a library for C# that provides a high level API for performing data parallelism operations. [15] includes a performance evaluation between C# using Accelerator, handwritten Pixel Shader 3.0 assembly code, and CPU-based C++ code. The evaluation benchmarks ten operations: sum, matrix-vector multiplication, matrix-matrix multiplication, life,

demosaic, convolve, rotate, corner detection, motion estimation, and stereo matching. Overall, the Accelerator code performs within 50% of the speeds of the handwritten Pixel Shader code. The Accelerator code performs better than the CPU-based C++ code on seven out of the ten operations, but when the Accelerator code performs worse, it performs significantly worse.

Accelerator's slow processing is from many design decisions. Accelerator uses just-in-time compilation for its fragment processor programs. This means that the library compiles each fragment processor program at runtime, when the fragment program is needed. Even though the library caches these programs for later use, the just-in-time compilation overhead and the library's other execution costs take up 9% of the running time on average.

Accelerator's worst performance was in the rotate and motion estimation operations. These operations involve many gather operations with special out-of-bounds access cases. While GPUs have hardware support for gather, the DirectX C# API does not provide access to the hardware support. Thus, Accelerator cannot utilize the GPU's resources while handwritten Pixel Shader code has access to this hardware support.

While Accelerator is typically faster than the CPU-based C++ code, it is not as fast as handwritten Pixel Shader 3.0 code. Additionally, Accelerator in only available for C#, and thus requires a Windows machine running DirectX and the .Net Framework. For these reasons, the team will not use Accelerator for the project.

1.2.3.2 BrookGPU Performance

BrookGPU extends C to include stream processing constructs that use the GPU to perform calculations. BrookGPU includes a compiler and a runtime environment that allows Brook programs to run on both ATI and NVIDIA GPUs with either OpenGL or DirectX. Brook virtualizes two important stream computing aspects. OpenGL, DirectX, and Cg limit the number

of outputs from a fragment program and limit stream dimensions and size. BrookGPU virtualizes these aspects and, for example, allows users to create streams that are not bound by texture memory restrictions within the GPU.

BrookGPU provides a high-level abstraction of the graphics pipeline through offering streams and kernels. Streams are analogous to textures within GPU memory, but they do not have limits on their size. Kernels are analogous to fragment programs that execute on fragment processors within the GPU. The main difference between kernels and fragment programs is that BrookGPU kernels allow for an arbitrary amount of output streams while fragment programs have a limit on the number of outputs and the size of textures used.

[16] includes a performance evaluation of BrookGPU. On average, BrookGPU's DirectX output is within 80% of hand-coded GPU implementation performance. BrookGPU's OpenGL output is significantly less efficient. Most of these inefficiency issues are due to the hand-coded GPU implementations using problem-specific knowledge to find "shortcuts." These optimizations are not possible when using BrookGPU.

BrookGPU is significantly faster than Accelerator. When using the DirectX runtime, BrookGPU is only marginally slower than hand-coded GPU implementations. Addtionally, BrookGPU's abstraction will allow the team to port a larger portion of the IPSA algorithm to the GPU within the limited timeframe for the project. Due to BrookGPU's significant decrease in programming time in comparison with using the base technologies of OpenGL, DirectX, or Cg, the team will use BrookGPU for porting applicable portions of IPSA to the GPU.

1.2.3.3 Cg

In order to reach the goals outlined in Section 1.3, the project group must choose a language that demonstrates the raw power of the GPU for high computation and computational biology problems in a short period of time. As such, Cg presents a programming language using C-like syntax and philosophy [9]. Each member of the group has worked extensively with the C language and will feel comfortable learning a similar language. In fact, one of the main reasons that the designers of Cg chose to implement an environment similar to C was to provide ease of programming to developers since they would feel familiar with the language.

Other goals met by the designers of Cg fit well with the project's purpose to perform computation on a GPU, contrary to the hardware's intended purpose. Since Cg implements a C-like interface, it is hardware-oriented while being general purpose [9]. The designers provided full support for hardware programming, similar to an assembler language, so that the developers could have more power. Therefore, the project group will have complete control over the underlying elements and will be able to demonstrate the best performance that GPGPU techniques can achieve for selected computationally intensive problems.

1.2.4 GPGPU Performance Techniques

GPGPU developers can use several techniques for improving program performance. Most of these techniques require direct programming with OpenGL, DirectX, or Cg. Three examples of GPGPU performance techniques are cache-efficient memory models, the vertex processor code motion technique, and GPU clustering.

1.2.4.1 Cache-Efficient Memory Models

Through the use of an intuitive and flexible language, successful GPU computation requires the effective representation of data in memory. Many applications beyond graphics can benefit from parallel GPU processing, including database indexes. These types of programs benefit since they can be expressed independently, and the GPU is able to process the same function on multiple elements of a two dimensional array. As shown by the model in [17], efficient algorithms achieve a two to five times improvement versus a CPU. Therefore, if researchers ignore the design of the hardware, they will not achieve much increase in speed since the GPU is intended for maximum efficiency through independent, simultaneous computation.

In order to maintain high throughput, one must develop a system that reduces that number of cache misses. Looking up elements that do not currently reside in the cache creates a high overhead. Therefore, [17] presents a technique that decomposes the input into smaller blocks, called quads. The GPU processes these separately so that a smaller dataset resides in memory. Then, a rendering pass performed at each stage reduces the dataset to its final return value. The project group will use the decomposition technique to reduce IPSA's protein chains into smaller pieces that the GPU can most efficiently process. By following an effective memory model, the project group expects to achieve a two to five times speedup on the GPU-based portions of IPSA in relation to their original CPU-based counterparts.

1.2.4.2 Vertex Processor Code Motion Technique

[18] includes the vertex processor code motion technique for improving the efficiency of GPGPU programs. The technique focuses on the fact that GPUs include both vertex processors (VPs) and fragment processors (FPs), but GPGPU programs typically only use fragment processors. By offloading appropriate instructions to the VPs, [18] was able to reduce execution

time of a Gaussian filter program by 40% and reduce the FP workload in ten of eighteen GPGPU programs tested.

Data in the GPU pipeline flows from VPs to rasterization and interpolation to FPs. The key to moving instructions from FPs to VPs is to identify movable instructions. [18] defined movable instructions as instructions that are executable, accessible, equivalent, and independent.

An instruction is executable if the VP instruction set includes it. For example, the txld instruction is not executable because it is not within the VP instruction set since VPs cannot read textures. Additionally, instructions are accessible if the VP registers can replace the registers used on the FPs. For example, registers on a VP cannot replace sampler registers on FPs because samplers involve textures and VPs cannot operate on textures.

An instruction is equivalent if the instruction is linear and does not use lower precision registers on the VP for high-precision data. Since FPs receive linearly interpolated data from VPs, one can only move linear operations to VPs. If an operation is not linear, then the rasterization and interpolation process between VPs and FPs in the GPU pipeline with mangle the data. Additionally, one cannot use lower precision registers within VPs to pass high-precision data to FPs because this will result in rounding errors.

An instruction is independent if it does not depend upon any unmovable instruction above it in the fragment program. For example, if an instruction I_1 is dependent upon a previous instruction that is not movable, then I_1 is not movable. This restriction is obvious because the previous instruction must execute before I_1 and moving I_1 to VPs will cause it to execute before the instruction that it depends upon.

[18] found that their technique of moving instructions from FPs to VPs improved the performance of ten out of eighteen GPGPU programs tested. The technique did not work for all

programs because offloading work from FPs to VPs increases the amount of data passed between the two. During this passing of data from VPs to FPs within the GPU pipeline, the rasterization and interpolation process became the bottleneck of some of the applications. While the technique can certainly increase efficiency of GPGPU programs, implementing the technique requires extensive knowledge of GPU programming. Thus, the team will not attempt this technique during the project.

1.2.4.3 GPU Clustering

[19] discusses the design, implementation, and results of using the Lattice Boltzmann Model (LBM) for flow simulation on a GPU cluster. The GPU cluster uses 32 nodes that are connected by a 1 Gigabit Ethernet switch. Each node has two Pentium Xeon 2.4GHz processors, 2.5GB of memory, and one GeForce FX 5800 Ultra with 128MB of memory. The GPU cluster executed each step of the simulation 4.6 times faster than [19]'s CPU cluster implementation.

[19] first developed an LBM algorithm on a single GPU. The algorithm ran 8 times faster on the GeForce FX 5900 Ultra when compared to their CPU version running on a Pentium IV 2.53GHz. One can easily visualize the design of the LBM used for the single GPU algorithm as a rectangle in three dimensions.

When scaling the LBM onto the GPU cluster, [19] roughly split the domain into cubes. Each node processes one cube. The data along the borders of each cube have to pass to neighboring GPUs for computation. This introduced a bottleneck while transferring data from a node's GPU to its CPU, and then across the network.

The primary challenge for [19] was minimizing the cost of communication between nodes. [19]'s cluster was hindered by using an AGP 8x bus, which only has a peak upstream of 133 MB/sec. They were also hindered by the fact that each GPU only had 128 MB of memory.

According to the team's test results, between 24 to 28 nodes was optimal. With more than 28 nodes the performance dropped significantly due to network collisions and other issues. In an effort to combat the network issues, the team set up a scheduling routine that made sure that not all of the nodes were transmitting border data at the same time.

Overall, [19] concluded that the price per performance of their GPU cluster was significantly better than purchasing a comparable CPU cluster. Today, with the performance increases in GPUs such as significantly more GPU memory and processing cores, the performance of [19]'s GPU cluster would only be greater. While a GPU cluster is not within the budget for the project, the promise of GPU clusters is certainly interesting.

1.3 Goals and Objectives

The team's ultimate goal is to port the most promising portion of IPSA into a working GPU-based prototype that performs at least twice as fast as its CPU-based counterpart. The ultimate goal of the project is certainly lofty, especially considering that none of the team members has any experience working with GPUs. Since the team is lacking in general experience with GPUs, the team must meet many preliminary learning, discovery, and testing goals before the team attempts to tackle the final goal.

First, the team will review the GPU programming tutorials in [20] and [21]. These tutorials demonstrate the basic concepts of using graphics processing pipeline to perform non-graphics related tasks. All of the tutorials that the team will review use Cg to complete the GPU programming tasks. After completing the tutorials, the team will produce GPGPU programs that demonstrate the best performance possible for select GPU-based algorithms barring hand-writing Pixel Shader 3.0 assembly code. These GPGPU case study programs are the deliverables of this first phase.

Next, the team will examine the BrookGPU example code and explanations in [16]. The team will learn to write simple GPU-based programs that implement general GPGPU techniques such as map and reduce. The GPGPU programs that demonstrate map and reduce are the deliverables for the second phase.

Once all team members have a grasp of GPGPU techniques and BrookGPU programming, the team will begin analyzing the feasibility of porting IPSA to the GPU. This will include a thorough study of the algorithm so that all team members understand the processing steps and data structures involved. During this third phase, the team will develop a profiling program, which will analyze the runtime spent in each section of IPSA and determine the most lucrative portions of IPSA for porting to the GPU.

During the final phase of the project, the team will implement on the GPU the portion of IPSA that has the most promise for parallelized processing. The team will also develop a prototype that demonstrates how the IPSA algorithm could transfer and receive data from the GPGPU program. The final deliverables of the project are the GPU-based program that implements a portion of IPSA and the IPSA-GPU interaction program that demonstrates how IPSA will access the GPU.

1.4 Overall Approach

The team will follow the waterfall model for the project. Since the project consists of porting a portion of an existing, well-defined algorithm to run on a GPU, the waterfall model is the best choice for a development model. After completing the initial learning and discovery goals listed under the Goals and Objectives section, the team will follow the general waterfall process model as shown in Figure 1.4a.

In addition to the waterfall model, the team will use a version of the iterative development model as shown in Figure 1.4b for determining the best possible translation of IPSA 's data structures into comparable data structures for the GPU and for determining the best fragment programs for performing the GPU computations. This process will include many design, implementation, and evaluation phases as the team refines the data structure and fragment program designs to achieve maximum performance. Since the design of the data structures is one of the most important issues to consider with GPGPU methods, this iterative development will be a large portion of the project.

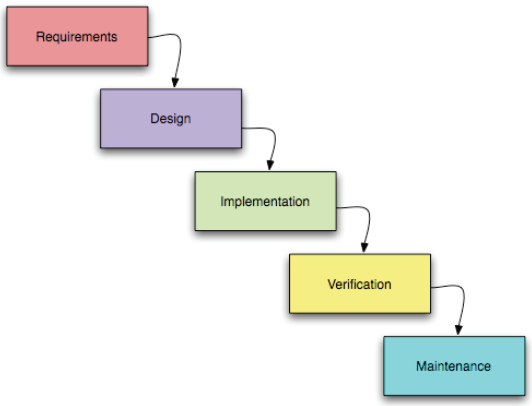

Figure 1.4a: The Waterfall Model

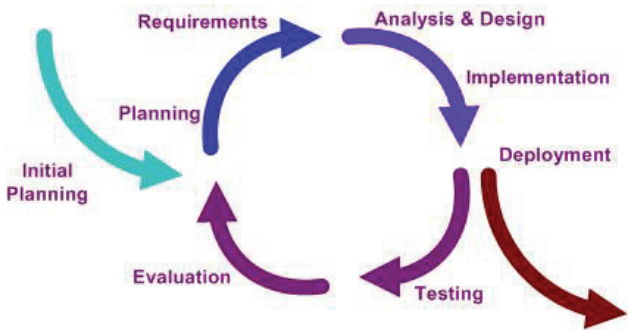

Figure 1.4b:
The Iterative Development Model

Graphics taken from Wikipedia.com under the Creative Commons Attribution ShareAlike License v. 2.5.

The team will use Cg and BrookGPU for developing the project's GPU-based programs. This decision is based on recommendations from Robert Luke and Derek Anderson, both graduate students that have experience porting algorithms to the GPU. The development, testing, and production environment includes two different graphics cards: an NVIDIA 8800 GTX with

128 processing cores and 768 MB of memory and an ATI x800 XT PE with 16 processing cores and 256 MB of memory.

1.4.1 System Diagram

The program will include many modules as detailed in Figure 1.4.1. An outside source such as an existing program within ProteinDBS will request to run the protein retrieval algorithm with a given set of parameters. The IPSA Request Handler, a component running on the CPU that is a modified version of the program that the original IPSA developer created, will receive the request and ultimately return the result. The IPSA Request Handler will load the nine IPSA index files and send the proteins to process to the GPU Program Loader. The GPU Program Loader will create the necessary data structures on the motherboard memory and load the GPU program onto the GPU.

Once the data is loaded onto the GPU, the goal of the GPU program is to execute the algorithm while minimizing the amount of references to non-GPU memory or procedures. The Retrieval Initializer will generate the GPU data structures that correspond to the CPU data structures created earlier. Once initialization of the memory is complete, the Stream Process Distributor will begin executing the algorithm.

The Stream Process Distributor executes the kernel programs that run on each data point in the GPU's memory. The component sends the datasets to be processed to the stream processors (pixel shaders) on the GPU. Each of these stream processors produces results that the Stream Processor Distributor receives and aggregates accordingly.

Once the stream processing is complete, the Result Handler component on the GPU sends the GPU Program Loader the result. The GPU Program Loader stores the result in the motherboard memory and sends the GPU result to the modified IPSA algorithm. The modified

IPSA algorithm executes the CPU-based IPSA algorithm while using the results that the GPU produced for the specific calculations that the team ported to the GPU. Once the Modified IPSA component sends the IPSA Request Handler the final result, the IPSA Request Handler sends the result to the outside source that requested the retrieval.

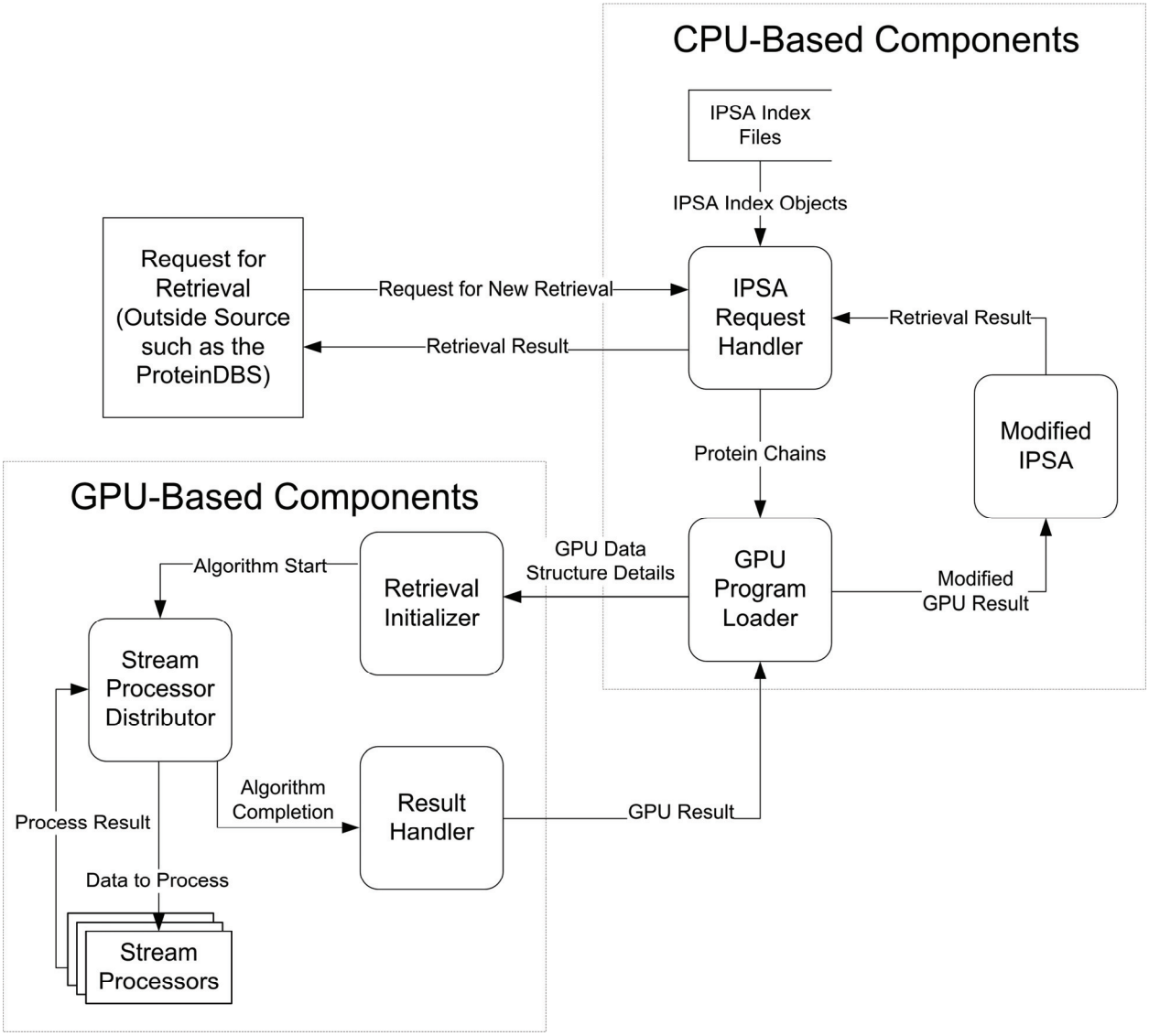

Figure 1.4.1: The Component Diagram

1.4.2 Advantages and Disadvantages

The advantage of the above component layout is that the layout ensures that the GPU-based algorithm minimizes its interaction with the CPU and motherboard memory. The key to gaining a speedup is to reduce the amount of GPU-CPU interaction. One needs to design the system to run on the GPU for the highest possible percentage of overall processing time.

The main disadvantage to the team's approach is the difficulty in coordinating efficient data structures between the GPU and CPU memory. This disadvantage is acceptable because it is currently the only way to use GPUs for general purpose computing. Since GPUs cannot store objects and data in the same way as CPU and motherboard memory, one must create a translation between the two representations. This translation and choice of data representation on the GPU is critical to the efficiency of GPU-based algorithms.

1.4.3 Costs

The most obvious costs of the project are the costs of the hardware. An NVIDIA 8800 GTX alone currently costs 575 dollars while an ATI x800 XT PE costs 250 dollars [22]. The hardware costs are definitely expensive for a college student, but Dr. Shyu's budget can sufficiently handle the cost. Section 2.3 presents a more thorough cost analysis.

2 Requirements Analysis

2.1 System Requirements and Constraints

The system has many requirements and constraints including required hardware, quality and reliability requirements, human factors, required software libraries, and performance requirements. While all constraints are important, the human factors in Section 2.1.3.2 had the largest impact on the project's timeline and goals. The time requirements in Section 2.3.1 were also significantly affected by the human factor constraints.

2.1.1 Operating Environment

The team will use two development machines as detailed in Table 2.1.1. The team will use multiple development machines in order to alleviate the human-based constraints described in Section 2.1.3.2 and to meet the project goals in the limited time span. Dr. Shyu provided Computer 1 while Todd Sullivan, one of the team members, provided Computer 2. Since both video cards support OpenGL and DirectX and both operating systems provide proper drivers for their respective video cards, the Cg and BrookGPU programs will execute correctly on both machines.

	Computer 1	Computer 2
Video Card	NVIDIA 8800 GTX	ATI x800 XT PE
Processing Cores	128	16
GPU Memory	768 MB	256 MB
CPU	Intel Pentium 4 2.8 GHz	AMD64 3400+
CPU Memory	4 GB DDR400	3 GB DDR400
Operating System	Fedora Core 6 (Linux)	Windows XP/Cygwin

Table 2.1.1: The Development Machines

As described in Section 1.2.3.2, BrookGPU's DirectX runtime is significantly faster than its OpenGL runtime. While this does not affect the development phase of the project, the restriction certainly affects the performance of the BrookGPU programs. Due to time and budget

constraints, the team cannot install a Windows operating system on Computer 1. Instead, the team has to use a free Linux distribution, which does not provide DirectX. Thus, in a real-world BrookGPU-based IPSA program, Dr. Shyu should run the program in a Windows environment in order to achieve maximum performance.

While BrookGPU's DirectX runtime is faster than its OpenGL runtime, working in Windows comes with its costs. In order to take advantage of widely available Linux libraries and other tools, the team must compile and execute the BrookGPU program using Cygwin, a Linux-like environment for Windows. As a result, in order to achieve the performance gain from the BrookGPU DirectX runtime, the team must manage working in an entwined Windows/quasi-Linux environment.

The team will not need to manage any web servers or database environments. IPSA's data structures are stored as nine index files. Each index file is a serialized Java object. IPSA's own algorithms and tools maintain its database. The team's GPU-based program will need to access IPSA's data structures, but the program will not need to alter them in any fashion. Thus, the team can access IPSA's local database by calling methods in Java in a similar fashion to Java's built-in data structures such as Vector and TreeMap.

2.1.2 Market Users and Characteristics

In compliance with U.S. regulations, the team will not violate any patents held by other GPGPU researchers and companies. The tools that the team will use for the project, such as Cg and BrookGPU, do not impose any regulatory or license-based restraints on the project. Sections 1.1.1 and 1.1.2 describe the project's market demand and customer requirements respectively.

2.1.3 Environmental Constraints

In order to satisfy customer requirements, the team must address quality and reliability constraints. Due to the nature of the system, the GPU-based program meets suitability and safety requirements because they are built into the original IPSA algorithm and, provided the GPU-based program is a direct port of portions of IPSA, are not actual requirements for the project. The team must also accommodate many human factors that are uncontrollable and sometimes unpredictable. The majority of these human factors reduce the time available for analyzing IPSA, designing the GPU-based solution, and implementing the prototype.

2.1.3.1 Quality and Reliability

The GPU-based solution must reliably provide similar results in comparison with the original algorithm. The original IPSA algorithm, written in Java, uses double precision floating point numbers. Current video cards use single precision floating point numbers. Due to this constraint, the GPU-based computations will offer less precision than the original CPU-based computations. This constraint is unavoidable in current hardware. The team hopes that since all computations will be less precise, the resulting values will remain the same in relation to each other. Since the resulting values will remain the same relative to each other, the ranking scheme will still return the same result.

2.1.3.2 Human Factors

Human factors severely reduced the amount of time available for the third and fourth phases of the project as described in Section 1.3. The team did not receive the IPSA source code until April 2. Upon receipt of the algorithm, the team did not receive the algorithm's dataset until April 6. The source code contained no inline comments and no documentation explaining the process of compiling and executing the various IPSA modules.

Additionally, the source code did not initially compile correctly because the team lacked several required libraries and the source code had multiple locations of hard-coded, system-specific paths. These library dependencies and hard-coded paths were not disclosed in any documentation and IPSA's original creator did not remember all of the locations where paths were hard-coded. As a result of these issues, the team was not able to successfully compile and execute IPSA until April 11.

Aside from human factors pertaining to source code issues, the team did not receive the primary development machine, Computer 1, until April 2. The machine had Red Hat Linux preinstalled, which was partially corrupted and unusable. Thus, the team spent a few days trying to fix the system. Once the proposition of fixing the system became futile, the team reformatted Computer 1's hard drive, installed Fedora Core 6, and installed all of the required libraries as detailed in Section 2.1.5. The primary development machine was functional on April 5.

Due to these human factors, the team had approximately two and a half weeks to profile IPSA, choose a suitable section to port to the GPU, and develop the prototype. The team remained steadfast (to no avail) in requesting the required source code, dataset, and development machine prior to the delivery of the items. To counter these human factors, the team spent the time before delivery of the items learning to program in Cg and BrookGPU and completing the first and second phases of the project's goals as described in Section 1.3. The team also countered these limiting factors through time management that allowed the team members to dedicate significant portions of each day to the project during the two-week period before the project's presentation at the Advisory Board.

2.1.4 System Components

The primary system components are the IPSA index files, the Java-based IPSA programs, and the GPU-based program that the team will develop. The team will develop a modified version of the Java-based IPSA algorithm that demonstrates how to use the GPU-based program for performing select calculations. The team will not manipulate the IPSA index files. Instead, the team will access the indexes through the Java-based IPSA program.

2.1.5 Software Interfaces and Libraries

The team will use Cg and BrookGPU to develop the deliverables described in Section 1.3. These libraries require OpenGL or DirectX to run. As covered in Section 2.1.1, both development machines contain support in hardware and software for OpenGL, while Computer 2 contains support for DirectX. When using OpenGL, the Cg and BrookGPU programs also both require GLUT, the OpenGL Utility Toolkit, and GLEW, the OpenGL Extension Wrangler.

Since the IPSA algorithm is written in Java, the team will use a modern Java Virtual Machine for Java 1.5 to execute the IPSA algorithm. Java includes constructs such as the Runtime.exec() method for executing operating system-specific commands. The team will use Runtime.exec() to execute and control the GPU-based program from within Java.

Since this method uses commands specific to the machine's operating system, the team will have to use different commands for invoking the GPU-based program on a Linux machine versus a Windows machine. The primary difference is that in Windows the team will have to use Runtime.exec() to load a new Cygwin instance and then load and control the GPU-based program from within the Cygwin process. On a Linux machine the team will simply use Runtime.exec() to load and control the GPU-based program directly.

2.1.6 Communication Interfaces

The team will manage the communication between the Java-based program and the GPU-based program using the features of the Process object in Java. After loading the GPU-based program using the Runtime.exec() method in Java, the team will control the GPU-based program by using the process' standard input and output streams. Java's Process object allows the team to read from the GPU-based program's standard output and write to its standard input as if the GPU program is running in a console and the Java program is the keyboard and screen.

Cg and BrookGPU manage the communication interface between the CPU processes and the GPU processes. Additionally, the IPSA programs manage the communication channels responsible for receiving retrieval requests and returning results. Thus, the team has little oversight responsibilities and abilities in regards to these communication interfaces.

2.1.7 Hardware Interfaces

Table 2.1.1 details the hardware for each development system. The only notable interface for the project is the connection between the video card and the CPU in each machine. Computer 1 contains a PCI Express 1.0 bus that connections the 8800 video card with the CPU. Computer 2 uses an AGP 8x bus for connecting the x800 video card with the CPU. The PCI Express 1.0 bus has a maximum data rate of 4 GB/s while an AGP 8x bus has a maximum data rate of 2.1 GB/s. While these constraints limit the maximum performance that the GPU programs can achieve, they do not affect the design or development details of the GPU-based programs.

2.1.8 System Maintenance

The team must provide full documentation for compiling and executing the deliverables outlined in Section 1.3. Since none of the team members will be available after the project's completion, the documentation must provide enough information for a new team to continue the research with as few relearning periods as possible. In order to maintain the system, the administrator will either need prior experience with GPGPU or be willing to learn GPGPU techniques. The team strongly recommends that any maintenance staff at least have experience with graphics programming using OpenGL or DirectX.

2.2 Performance Requirements

The portion of IPSA that the team ports to the GPU must execute faster than performing the calculations on the CPU. The definition of "faster" is important because the GPU cannot outperform the CPU when executing simple calculations on small vectors or matrices. For example, the GPU performs significantly worse in comparison to the CPU when performing calculations on three-by-three matrices because the GPU is designed to process large quantities of data at the same time. Thus, determining which is "faster" by having each program perform one protein chain comparison at a time will always result in the CPU-based program being faster.

In order to take advantage of the GPU, the GPU-based program should pack thousands of three-by-three matrices into a single matrix. Instead of performing the calculation on one three-by-three matrix at a time, the GPU-based program should calculate all of the results simultaneously. The team should then compare the time elapsed for this bulk processing with the average time from the CPU-based program's sequential processing of an equal number of three-by-three matrices. Thus, the pertinent performance requirement is that the GPU-based

program processes all of the protein chains in bulk faster than the original CPU-based section of code processes the protein chains sequentially.

2.3 Resource Requirements

The human factors significantly affected the time requirements for the project. The team also required many important resources in Section 2.3.2, such as Computer 2, because of the environmental constraints. The project's facilities were fairly small for the team size, but the accommodations were paid for by other projects and thus acceptable.

2.3.1 Time

As described in Section 2.1.3.2, Human Factors, time management is crucial for adapting to unpredictable events. Each team member averaged eight hours per week over a ten-week development period, totaling 320 work hours. At an average hourly rate of fifty dollars per hour, the time costs are $16,000, as detailed in Table 2.3.4.

Due to the environmental constraints in Section 2.1.3, each team member did not consistently work eight hours each week. Instead, the team's work hours included a sharp, sustained rise during the three and a half weeks at the beginning of April. In fact, over the weekend of April 21st and 22nd two team members practiced Extreme Programming [23] and each clocked two twelve to fourteen hour work days in a row. These irregular work patterns were necessary to accommodate the unavoidable constraints described in Section 2.1.3.2 and were achieved by early planning that eliminated other obligations of each team member during the heavy development period.

2.3.2 Resources

Aside from the IPSA source code and protein dataset, the only resources required are the development machines detailed in Table 2.1.1. Computer 1's video card cost $575 while its other parts total $1,200. Computer 2's video card cost $250 while its other parts total $950. The total resource costs are $2,975, but an important note is that most of the development machines' parts were reused parts from other projects.

2.3.3 Facilities

The team requires at least one location where team members can operate both development machines side-by-side. Dr. Shyu's student cubicle space meets this requirement. The project does not incur additional expenses from using the cubicle space because the particular cubicle is not in use by another project and Dr. Shyu's other projects pay for the costs of the space.

2.3.4 Budget

Employment	$16,000
Computer 1	$1,775
Computer 2	$1,200
Total	$18,975

Table 2.3.4: The Cumulative Project Budget

2.4 Evaluation Metrics

The project's evaluation metrics are the production of the deliverables described in Section 1.3 and a performance evaluation of the GPU-based IPSA Compute_RMSD_D1 function that shows that the function is faster than its CPU-based counterpart. Section 2.2 includes the project's definition of the term "faster" and describes how the team will assess the GPU-based program's performance in comparison with the CPU-based calculations. The process is a complete success if the team produces the deliverables and positive performance evaluation.

3 Design Specification

The majority of the project's design process centers around designing data structures and designing software modules. Since IPSA includes its own data structures, the data structure designs involve translating IPSA's data structures into GPU-compatible data. The software, on the other hand, involves modifying IPSA's program flow to include GPU computations and designing software for profiling IPSA and for performing the GPU-based calculations.

3.1 Hardware

The project does not require any specialized hardware designs, which allowed the team to focus primarily on software designs. While the team did not create any hardware designs, the project requires knowledge of the GPU pipeline and the general design of video cards. Section 1.2 discusses this general GPU design.

3.2 Data Requirements

IPSA requires many different data sources as described in Section 3.2.1. The team's GPU-based computations involve one particular data structure – a protein chain consisting of an array of doubles. Sections 3.2.2 and 3.2.3 describe translating these protein chains into GPU-compatible data.

3.2.1 IPSA Data Sources

The IPSA algorithm includes nine data files as shown in Figure 3.2.1. These nine data files include two protein lists and seven indexes. IPSA stores these data structures as serialized Java objects. The project's goals do not require the team to alter these data structures in any way. Instead, the team only has to access the proteins in the data structures using method calls similar to Java's built-in Vector and TreeMap classes.

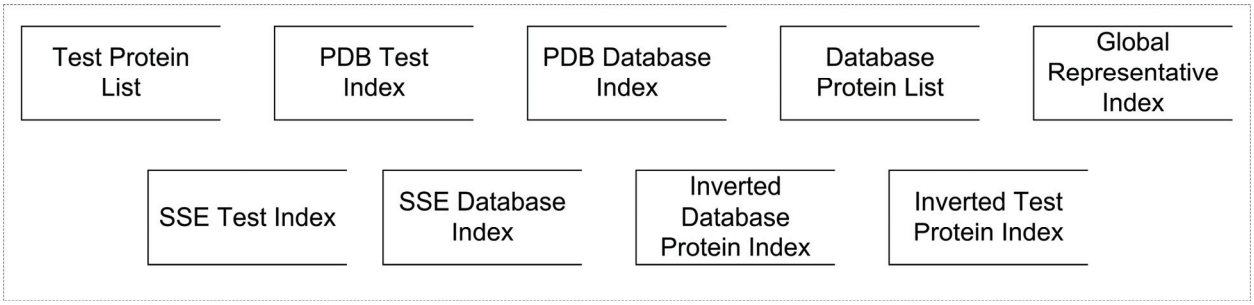

Figure 3.2.1: IPSA Data Sources

3.2.2 Translation to a GPU Datatype

In order to process the proteins on the GPU, the team must translate the protein's structure in Java into a suitable structure in GPU memory. IPSA stores the protein chains as arrays of doubles (double precision floating point numbers) that are either thirty or forty-five elements long. Since current GPUs can only process floats (single precision floating point numbers), the protein chains will lose half of their precision in GPU memory.

Additionally, GPUs store four floats at each pixel on a texture. These four floats are for the red, green, blue, and alpha values of the pixel. Thus, each element of an array on the GPU is considered a float4, which can be thought of as a C struct with floats denoted by pixel.r, pixel.g, pixel.b, and pixel.a. Since IPSA calculates most chain values in groups of three, the GPU arrays use the red, green, and blue floats while disregarding the alpha float.

3.2.3 Matrix Translations

When performing comparisons on chains, IPSA uses many three-by-three matrices. Since the GPU's main skill is performing calculations in bulk, the GPU-based algorithm needs to pack many three-by-three matrices into a single giant matrix. Figure 3.2.3 demonstrates this process of storing many three-by-three matrices as a single texture on the GPU.

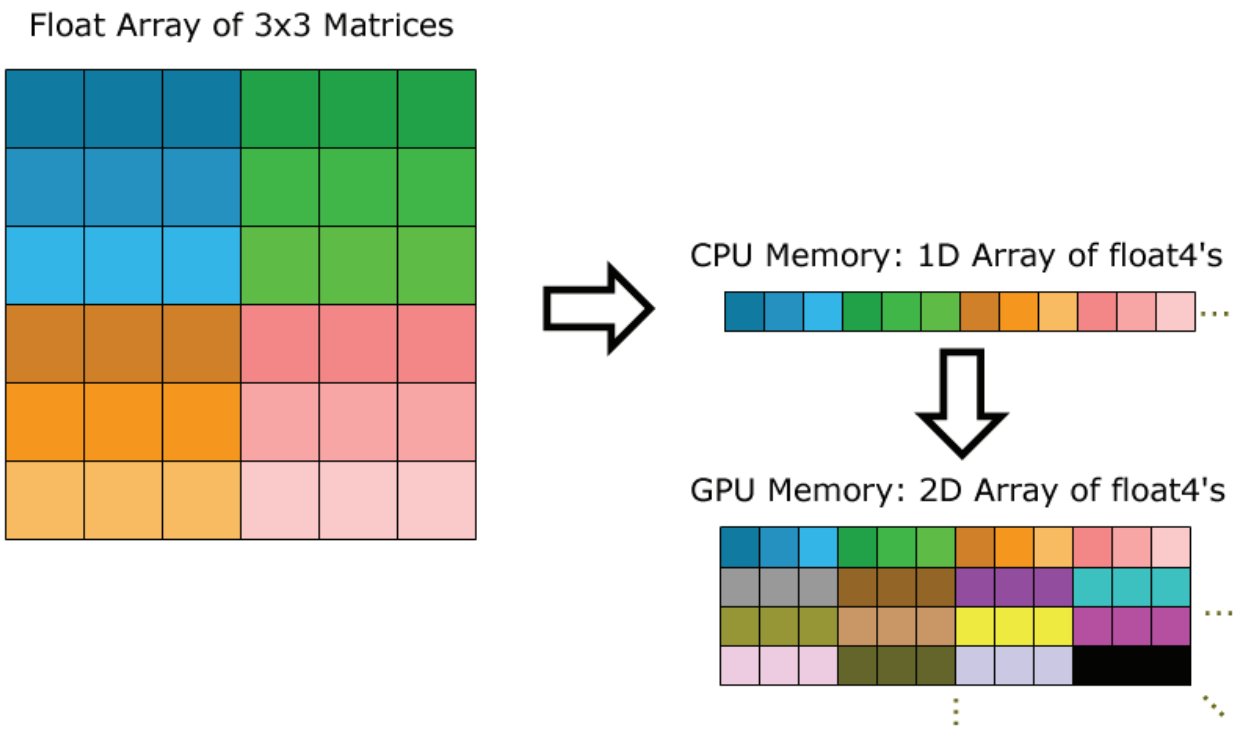

Figure 3.2.3: Three-by-Three Matrix Translation

First, the three-by-three matrices are placed in a larger square matrix with sides that are a multiple of three. Second, each row of each three-by-three matrix is then placed into a single float4 with the float4s red value storing the row's first element, green value storing the row's second element, and blue value storing the row's third element. These float4s are arranged in CPU memory as a one-dimensional array of float4s.

Figure 3.2.3 demonstrates this second step as the first arrow in the center of the figure. The top row of the blue three-by-three matrix collapses into the first float4 element in the 1D array. Similarly, each three-by-three row collapses into the single element in the 1D array that contains the same color as the original row of floats.

After creating a one-dimensional array of float4s in CPU memory, the array is written to a texture in GPU memory as shown by the second array in Figure 3.2.3. While writing the array

to a texture in GPU memory, the one-dimensional array wraps around into a two-dimensional texture. The resulting 2D texture is three times as long as it is high.

For clarity, Figure 3.2.3 demonstrates the process with a two-by-two matrix of three-by-three matrices. In reality, the second step (first arrow in the figure) writes the first row of three-by-three matrices before moving to the second row. Thus, on a larger set of three-by-three matrices the matrix denoted by orange is not placed next to the matrix denoted by green.

3.2.4 Protein Chain Translations

Since the GPU needs to compute many protein chain comparisons at once, the GPU also needs to pack protein chains into a single large texture. As stated earlier, protein chains are either thirty or forty-five floats long. Thus, each protein chain requires either ten or fifteen float4s where each float4 contains three floats in the RGB values.

For efficiency reasons, the large texture containing these chains needs to store each chain as a symmetric matrix of float4s. The smallest symmetric matrix that contains at least fifteen elements is a four-by-four. Figure 3.2.4 shows how multiple protein chains map into the large texture.

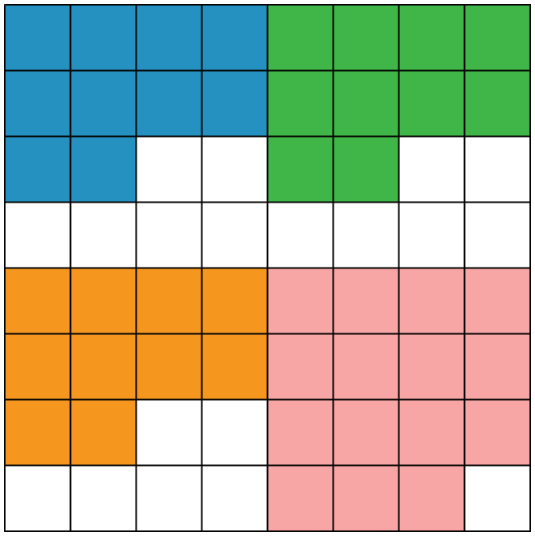

Figure 3.2.4: Protein Chain Mapping into Texture

The colored blocks in Figure 3.2.4 denote float4s of the texture that each chain occupies. The white blocks are float4 elements that contain all zeroes and are necessary wasted space. The blue, green, and orange chains are thirty floats long. The last chain is forty-five floats long. Each thirty-float chain wastes six float4s while each forty-five-float chain wastes one float4.

3.3 Software

The project's software designs include the IPSA Profiler, Java-C++ Interaction, and the Compute_RMSD_D1 function. The IPSA Profiler and Java-C++ Interaction programs will integrate into the original IPSA program. The Compute_RMSD_D1 function is a GPU-based program that is a translation of IPSA's Compute_RMSD_D1 function.

3.3.1 IPSA Profiler

The IPSA Profiler's purpose is to provide a convenient method for recording the processing time of each section of IPSA and analyzing the results. While analyzing a program, the profiler maintains a stack of each layer of executing code. The layer on the top of the stack is the current, most narrow section of code that is executing in the program.

When entering a new section of the program, the profiler places a new layer on the top of the stack. The profiler records the new layer's start time, depth, and layer number when adding the layer to the stack. When exiting a section of the program, the profiler pops off the layer at the top of the stack. The profiler records the popped layer's end time and writes the layer's information to the current session's log file. The profiler also maintains execution time totals for each layer in relation to its name and its path.

Once the program completes execution, the profiler sends the log file to an analyzer that generates a summary report of the profile. The analyzer creates a plain-text report that displays total time by layer, percent total time by layer, total time by layer with respect to its path, and

percent total time by layer with respect to its path. The analyzer sorts these values so that the report clearly shows the sections of the program that consume the majority of the execution time.

3.3.1.1 Class Diagram

The IPSA Profiler contains three classes as shown in Figure 3.3.1.1. The Profiler class and ProfileAnalysis class are public classes. The Layer class is an internal class of the Profiler class.

When entering a new section of a program, the program calls the Profiler class' push method. When exiting a section of a program, the program calls the Profiler class' pop method. The Profiler class' close method writes the final profile to disk.

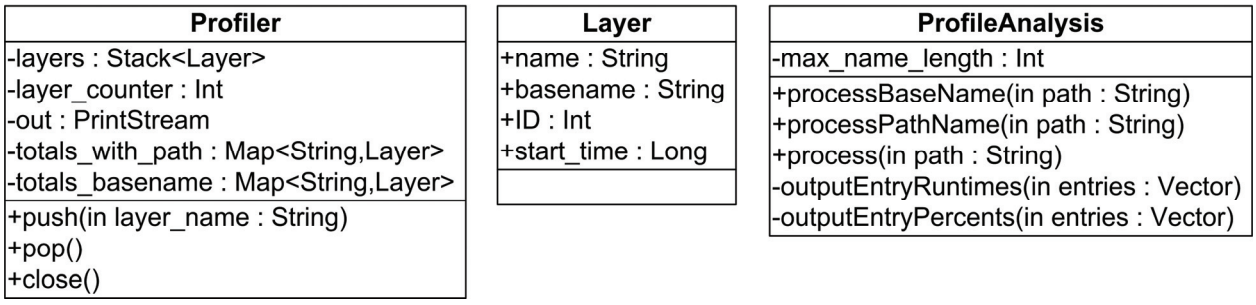

Figure 3.3.1.1: IPSA Profiler Class Diagram

3.3.1.2 Data Flow Diagram

Figure 3.3.1.2 illustrates the activity of an instance of the Profiler class when profiling a program. The actions correspond to the previous English language description of the IPSA Profiler. The Profile Analysis class does not need to a data flow diagram because the process simply involves parsing a text file, calculating percents, sorting the results, and output the sorted results.

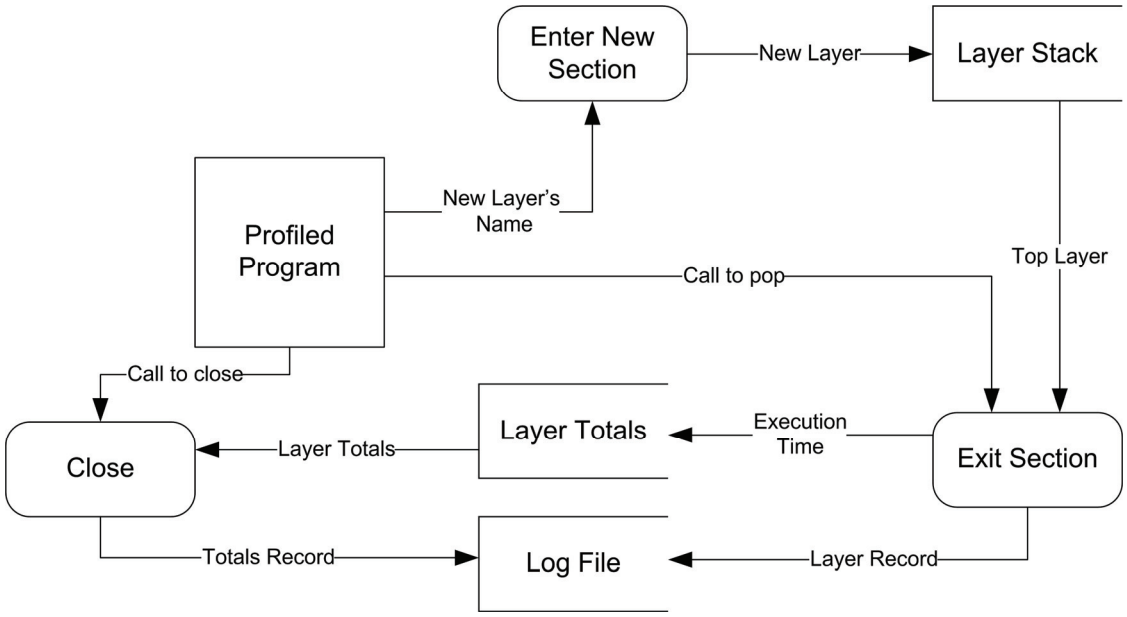

Figure 3.3.1.2: IPSA Profiler Data Flow Diagram

3.3.2 Java-C++ Interaction

Figure 3.3.2 illustrates the control flow of the original IPSA algorithm. The program loads the index objects from disk as serialized Java objects, sends the proteins to a Rank Proteins procedure, and then sends the protein scores to a Find Best Score procedure. The Find Best Score procedure sorts the scores in descending order and outputs the results.

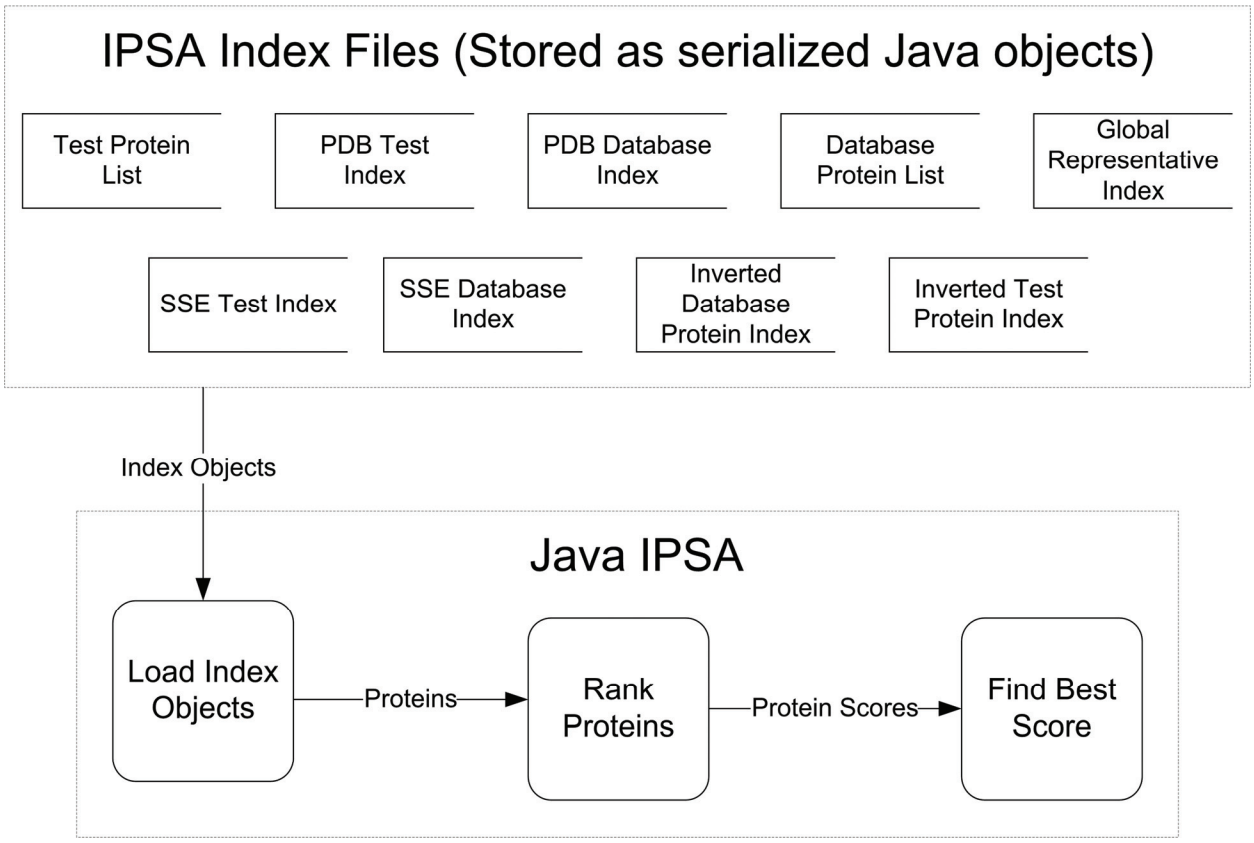

Figure 3.3.2: Java-C++ Interaction High-Level Process Diagram

IPSA's designer implemented the algorithm in Java. Regardless of the library or method used for GPGPU, the GPU-based program is ultimately a C++ application. In order to integrate the two, the Java program needs to start and control the C++ application.

The Java program starts the C++ program by executing an operating system-specific command and attaching the new process to a Process object. The Java program then extracts the input and output streams from the Process object. The Java program uses the output stream in a

similar way to a human using keyboard input to a command line program. The Java program uses the input stream to read data from the C++ program.

When the C++ program reads from its standard input, it reads commands sent to it by the Java program through the Process object's output stream. Similarly, when the C++ program writes to its standard output, it writes data to the Process object's input stream. This process is similar to connecting programs together using sockets. The difference is that neither program has to know anything about sockets and the C++ program is completely oblivious to who controls it. Thus, when testing developers can execute the C++ program from the command line and provide input using the keyboard.

Once the Java program starts the C++ program, it provides the C++ program with the list of protein chains to compare. The C++ program simultaneously computes all of the comparisons. Once the computations are complete, the C++ program sends the results to the Java program. The Java program stores all of the results and begins the normal IPSA algorithm. Whenever the Java program's execution reaches a computation that the C++ program calculated, the Java program retrieves the result from memory instead of computing the result with the CPU.

3.3.2.1 High-Level Process Diagram

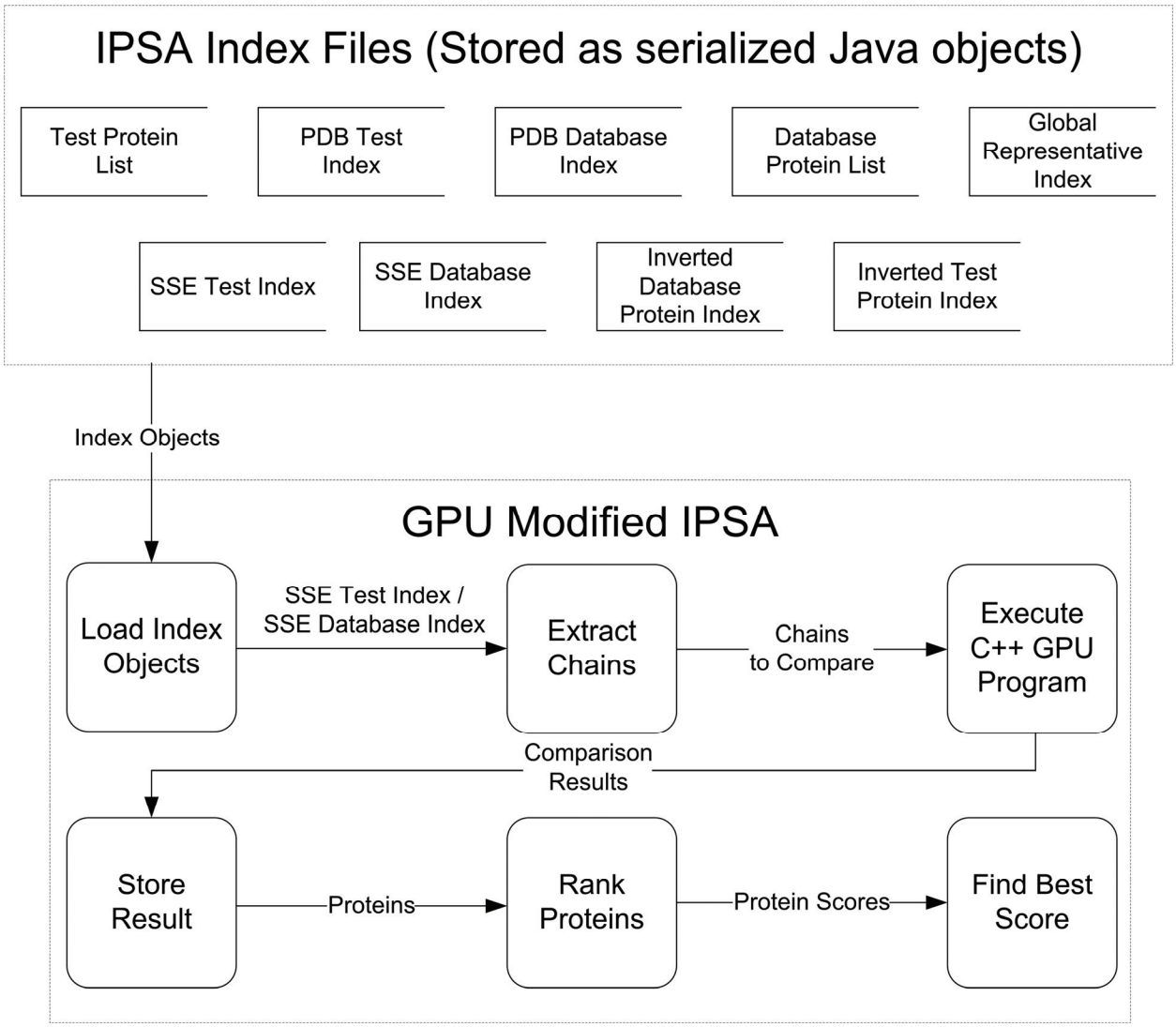

Figure 3.3.2.3: Java-C++ Interaction High-Level Process Diagram

3.3.3 Alternative Java-C++ Interaction

Alternatively, the Java program could send a protein chain comparison request to the C++ program when the Java program needs the comparison result. The two programs would communicate in the same way, but instead of bulk processing the comparisons in the beginning of the algorithm, the C++ program would process each comparison as they are needed. Figure 3.3.3.1 depicts this alternative design.

3.3.3.1 Alternative High-Level Process Diagram

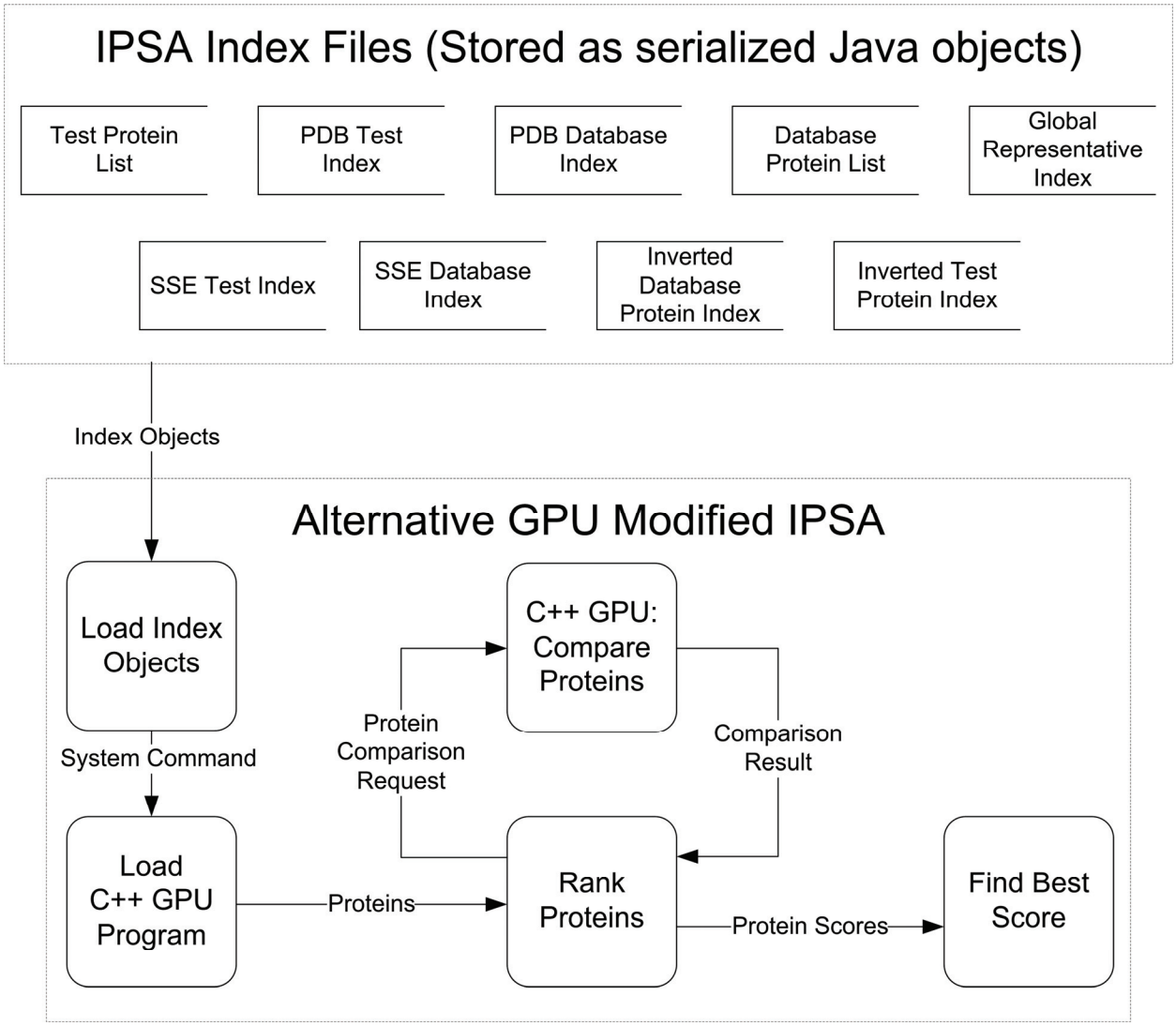

Figure 3.3.3.1: Alternative Java-C++ Interaction High-Level Process Diagram

3.3.3.2 Advantages and Disadvantages

The primary disadvantage to the original Java-C++ interaction design in Figure 3.3.2.3 is the high memory usage. While the C++ program is calculating protein comparisons, the C++ program and the Java program both have all of the protein chains stored in memory. Once the C++ program finishes calculating the comparisons, the Java program stores the results in memory for the entire duration of the IPSA algorithm.

The alternative design eliminates this memory usage issue because the C++ program only stores the two proteins that it is currently comparing. Additionally, the Java program does not store all of the results in memory for the entire algorithm. Instead, the Java program only stores one comparison result at any point in time.

Unfortunately, the alternative design does not take advantage of the GPU's ability to calculate thousands of protein comparisons simultaneously. While requiring less memory, the alternative design has significant performance issues. Thus, the higher memory usage of the original design is necessary to take advantage of the GPU's bulk computation ability.

3.3.4 The Compute_RMSD_D1 Function

Section 5.3.1 describes the results of profiling IPSA with the IPSA Profiler. The results of the profiling clearly show that the Compute_RMSD_D1 function within IPSA has the most potential for GPU-based computation. The Compute_RMSD_D1 function performs a comparison on two protein chains. Each protein chain is either thirty or forty-five floats long. The algorithm produces many intermediate three-by-three matrices during the process. The result is a single float.

3.3.4.1 Algorithm

The following bullets and pseudo code describe the Compute_RMSD_D1 algorithm. The input chains that the algorithm compares are denoted by chain8 and chain9. All of the calculations are matrix calculations. Most of the calculations perform computations on three different lines of the matrices at the same time. The pseudo code is a broad generalization of the actual process for one comparison. The GPU-based version of the algorithm must adapt each computation to work on a two dimensional array containing thousands of comparisons that are computed simultaneously.

1) Calculate the mean of each chain

```
mean8 = Sum( chain8 ) / Size( chain8 );
mean9 = Sum( chain9 ) / Size( chain9 );
```

2) Shift each chain's floats by its corresponding mean values.

```
PointsOf( chain8 ) = PointsOf( chain8 ) – mean8;
PointsOf( chain9 ) = PointsOf( chain9 ) – mean9;
```

3) Calculate a three-by-three matrix *tR*.

```
Row1( tR ) = Sum( PointsThatAreMultiplesOfThree(chain8) * chain9 );
Row2( tR ) = Sum( PointsThatAreMultiplesOfThreePlus1(chain8) * chain9 );
Row3( tR ) = Sum( PointsThatAreMultiplesOfThreePlus2(chain8) * chain9 );
```

4) Transpose *tR* into *R*.

```
R = Transpose( tR );
```

5) Calculate the three-by-three matrix *tR* * *R*.

```
tRR = MatrixMultiply( tR, R );
```

6) Calculate the eigenvalues *mu* and eigenvectors *a* of the *tR* * *R* matrix.

```
mu = GetEigenvalues( tRR );
a = GetEigenvectors( tRR );
```

7) Calculate and normalize the *b* vectors.

```
b = Sum( R * ColumnsOf( a ) ) / SquareRoot( mu );
```

8) Calculate the rotation matrix *tU* using the *b* vectors and the eigenvectors *a*.

```
tU = ColumnMultiply( b, a );
```

9) Transpose *tU* into *U*.

```
U = Transpose( tU );
```

10) Calculate the root mean square error.

```
tempchain = Sum( ColumnsOf( chain9 ) * Columns of U );
rms = Sum( Square( chain8 -  tempchain ) );
```

11) Return the root mean square deviation.

```
rmsd = SquareRoot( rms / chain_size );
```

3.3.4.2 Eigenvalues and Eigenvectors

Step 6 in Section 3.3.4.1 involves calculating eigenvalues and eigenvectors. The process of computationally determining a matrix's eigenvalues and eigenvectors is complicated. This process requires more time than the project's constraints allow. Thus, the GPU-based prototype does not calculate the eigenvalues or eigenvectors. To ensure the accuracy of the output, the team will hardcode these values while testing the program. To ensure an accurate performance evaluation, the team will remove the eigenvalue and eigenvector computation time from the execution time of the CPU-based algorithm as described in Section 5.3.2.

3.4 Testing Methods

The team will use comparisons between program output and either hand-determined correct output or output from the original IPSA program to determine whether each piece of software is working correctly. The IPSA Profiler and Java-C++ Interaction will use hand-determined output. The Compute_RMSD_D1 function will naturally use the output from the original IPSA function.

3.4.1 IPSA Profiler

The team will test the accuracy of the IPSA Profiler by running the profiler on a program that simply contains multiple nested layers that each cause the execution thread to sleep for a predefined number of seconds. Since each layer sleeps for a predefined number of seconds, the team can hand calculate a report that summarizes the execution time at each layer and along each execution branch. The team will compare the hand-calculated report with the report that the profiler generates. If the reports are the same, then the IPSA Profiler does not have any bugs and works properly.

3.4.2 Java-C++ Interaction

The team will test the Java-C++ interaction by using a predefined list of protein chain comparisons and comparison results. The Java program will send the list of protein chain comparisons to the C++ program. The C++ program will match each protein chain comparison with its predefined comparison result and send the comparison results back to the Java program. The Java program will output the results to the console. If the results are the same as the hand-calculated results then the Java-C++ interaction works correctly.

3.4.3 The Compute_RMSD_D1 Function

The team will test the GPU-based Compute_RMSD_D1 function by outputting to the console the result of each step described in Section 3.3.4.1. The team will compare these results with the corresponding results from the original Java function. If all of the results are the same, then the GPU-based function calculates the values correctly for a single protein chain comparison.

Since the GPU-based function calculates thousands of comparisons at the same time, the team needs to determine if all of the comparisons are calculated correctly. To do this, the team

will write the textures to CPU memory after each step described in Section 3.3.4.1 and then output to the console selected results from the matrix of results. The team will compare these results with the corresponding results from the original Java function.

3.5 Scheduling and Task Assignments

Table 3.5a: Schedule

ID	Task Name	Start	Finish	Duration
1	Form Group	2/5/2007	2/9/2007	1w
2	Determine Project	2/12/2007	2/16/2007	1w
3	Problem Statement	2/23/2007	3/6/2007	1.5w
4	Literary Review	3/7/2007	3/16/2007	1.5w
5	Combine PS's and LR's	3/16/2007	3/22/2007	1w
6	Explore How GPGPU Works	2/16/2007	3/1/2007	2w
7	Study CG, BrookGPU, other options.	2/23/2007	3/9/2007	2.2w
8	BrookGPU Case Studies	3/9/2007	3/30/2007	3.2w
9	Cg Case Studies	3/19/2007	4/17/2007	4.4w
10	IPSA Profiler	4/2/2007	4/6/2007	1w
11	Program IPSA Segments with GPGPU	4/9/2007	4/24/2007	2.4w
12	Prepare Presentation	4/16/2007	4/23/2007	1.2w
13	Prepare Final Report	3/22/2007	5/23/2007	9w

Table 3.5b: Task Assignments

	Mathew Alvino	Travis McBee	Heather Nelson	Todd Sullivan
Problem Statement				
Writing	◆	◆	◆	◆
Editing	◆	◆	◆	◆
Requirements Analysis				
Writing	◆	◆	◆	◆
Editing		◆	◆	◆
Design Specification				
Designs			◆	◆
Writing			◆	◆
Editing			◆	◆
System Implementation				
2D Arrays versus 1D Arrays				◆
Cg Mathematical Computation		◆	◆	
Cg Average Random Walk Distance			◆	
IPSA Profiler				◆
IPSA Java-C++ Interaction				◆
IPSA Compute_RMSD_D1 Function				◆
Writing			◆	◆
Editing			◆	◆
System Performance and Evaluation				
2D Arrays vs. 1D Arrays Data Collection	◆			◆
Cg Mathematical Computation Data Collection			◆	
Cg Average Random Walk Distance Data Collection			◆	
IPSA Compute_RMSD_D1 Function Data Collection				◆
Writing			◆	◆
Editing			◆	◆
Summary and Conclusions				
Writing		◆		◆
Editing			◆	◆
Future Work				
Writing	◆		◆	◆
Editing	◆		◆	◆
Advisory Board Presentation				
Presentation Creation	◆	◆	◆	◆
Presentation Editing	◆	◆	◆	◆

4 System Implementation

The system implementations involve a wide variety of programming languages and technologies. The GPGPU Limitations Demonstration and the Compute_RMSD_D1 function use C++ and BrookGPU while Mathematical Computation and Average Random Walk Distance use C++ and Cg. As indicated earlier by the name, Java-C++ Interaction includes a Java program and a C++ program.

4.1 GPGPU Limitations Demonstration

GPU-based programs have many limitations that the team must take into account when implementing the deliverables. GPUs do not have random access memory, so programmers need to avoid lookups. Branching on the GPU and transferring data from the CPU to the GPU are also very costly. Two dimensional textures are also significantly more efficient than one dimensional textures.

4.1.1 2D Arrays versus 1D Arrays

The 2D Arrays versus 1D Arrays program demonstrates the superiority of using two dimensional arrays in GPU computations. The program generates a set of random floats in the range [0.0, 1.0) and loads the numbers onto a 2D texture and a 1D texture on the GPU. Next, the program multiplies each float by 256 and then takes then floor of the result. The program compares the execution time for each texture with the execution time of performing the same operations on the CPU.

The team implemented the 2D Arrays versus 1D Arrays program using BrookGPU. The 2D texture is a two dimensional Brook stream of size 1280 by 1280. Similarly, the 1D texture is a one dimensional Brook stream of size 1280. The previously described operations execute on datasets of 1,000 elements to 1,000,000,000 elements. Since a single Brook stream cannot hold

the required amount of elements, the program repeats the process multiple times until the number of elements operated on equals the target dataset size. One Brook kernel performs the multiplication and floor operation on each element.

4.2 GPGPU Potential using Cg

In order to facilitate the group's understanding of GPGPU programming techniques, the team implemented two fundamental programs as case studies for GPGPU performance evaluation. Before creating Cg programs, the hardware must include the correct libraries and development environment as described in Section 2.1.5. For this portion, the group used Computer 1, which includes an NVIDIA 8800 GTX graphics card and an Intel Pentium 4 2.8 GHz processor. The necessary software includes the OpenGL Utility Toolkit (GLUT) and OpenGL Extension Wrangler (GLEW) libraries, the Cg Toolkit for shader support, and a C/C++ compiler for the actual development.

4.2.1 Mathematical Computation

Initially, the group implemented a complex parallel mathematical computation over each element in a large array. This example, derived from the Basic Math Tutorial in [21], served as the basis for understanding the GPU pipeline and allowed the team to analyze the performance of the GPU in its ideal use. In order to perform this computation, the group performed several initialization steps. In all, a simple four-line CPU program was translated into 532 lines of code, suitable for execution using the GPU. Each step, as outlined in the following paragraphs, enables the use of computational GPU processing.

Before attempting to utilize the GPU processors, the program must fill a CPU array with the data that will later be passed to GPU textures. Since the CPU array lies in a one-dimensional

space, the program will translate the elements into a two dimensional representation. Furthermore, several libraries enable the program to pass data onto the GPU.

First, the GLUT library allows the program to create a context for OpenGL by opening an invisible window. For graphics related GPU programming, this window displays the images and textures. However, GPGPU programming only requires the window in order to provide a framework for computation, since no output will be displayed. In addition, the algorithm initializes GLEW in order to load important OpenGL extensions, such as those that handle floating-point numbers. The libraries will enable Cg to create a context for general purpose computation.

Following the library initialization steps, the program creates a framebuffer object that will allow the CPU to interpret the GPU result. For example, the GPU represents data using four channels, where each channel represents eight bits. In graphics programming, these channels correspond to the amount of red, green, blue, and alpha transparency in an image. However, the use of this offscreen buffer to render calculations allows the program to bypass the limitations caused by eight bit channels. An offscreen framebuffer object is essential to GPGPU programming since it provides better consistency between the CPU and GPU.

Once the program initializes the environment, the algorithm may create GPU textures for the CPU data and initialize the shader. A function generates each texture with a specific identification and transfers the CPU data to the GPU. Then, the algorithm creates a fragment program, or computational kernel. The kernel behaves as a loop where it acts on each element in the texture, independently of the other elements. Furthermore, the actual kernel, or shader, is written in Cg and loaded into the fragment program. Figure 4.2.1 shows the actual shader source used for this computation.

```
float4 saxpy (
      in float2 coords : TEXCOORD0,
      uniform samplerRECT textureY,
      uniform float alpha )  : COLOR {
            float4 y = texRECT (textureY, coords);
            return alpha*y+ (alpha+y)/(alpha*y)*alpha;
      }
```

Figure 4.2.1:
Cg Shader Source for Mathematical Computation

After each initialization step, the GPU can finally perform the computation on each element in the texture. This algorithm uses a ping-pong technique to compute the result using the value of the texture element and stores the result back into the same element. The algorithm renders each pixel, or element, in the texture to a quad. Finally, the program transfers the resulting data back to the CPU so that it may be displayed to the user.

4.2.2 Average Random Walk Distance

In order to further study the potential for GPU implementations of CPU programs beyond mathematical computations, the team implemented a common problem in protein folding algorithms. The Random Walk Distance algorithm generates the average distance between points in a walk using a two dimensional space. Since this technique is often used in protein folding algorithms, it serves as a useful model to examine the potential of GPGPU algorithms to serve protein prediction and retrieval research.

Many of the steps outlined in the preceding section were also followed to initialize the Random Walk environment. As before, the program included and initialized the GLUT and GLEW libraries, prepared the offscreen buffer, and initialized four arrays with random points to follow during the walk. After creating the four CPU arrays, the program created textures and transferred the CPU data onto the GPU. Finally, the algorithm used the ping-pong technique to

store results and implemented the reduction technique to return a single value, the average distance of the walk.

Although the program follows many similar steps as the complex mathematical computation, the group made significant improvements to the algorithm. For example, the CPU algorithm performs a complex computation on each element, sums the results, and returns the average. This type of computation therefore requires the use of multiple Cg shaders in order to first run the equation and finally sum all of the results. Therefore, the Random Walk Distance algorithm uses two shaders, which each binds to a fragment program and executes using the same textures.

Since Random Walk Distance uses the ping-pong technique during the initial mathematical computation, the program does not need to create an additional texture for use during the summation. First, the function maps the shader described in Figure 4.2.2 to the elements in each texture. Texture y, via the ping-pong technique, switches from a read-only to a write-only texture in order to store the returned results. After mapping the equation to each pixel, the algorithm follows the reduction technique in order to sum the entire result. In general, a reduction will not prove faster on a GPU than on a CPU since the result is dependent on *each* element in the texture, thereby eliminating the potential for parallel computation. However, since the group has chosen to represent each element as a float4, including the result, each channel of the result is summed independently, as shown in Figure 4.2.2. This allows any reduction algorithm, such as summation, to perform approximately four times faster on a GPU than on a CPU. Once the GPU returns the result, the CPU must take one final step to sum the four channels of the result.

```
float4 map(
       in float2 coords: WPOS,
       uniform samplerRECT textureX,
       uniform samplerRECT textureU,
       uniform samplerRECT textureV,
       uniform samplerRECT textureY) : COLOR {
              float4 x = texRECT(textureX, coords);
              float4 y = texRECT(textureY, coords);
              float4 u = texRECT(textureU, coords);
              float4 v = texRECT(textureV, coords);
              return sqrt(pow((x-u),2) + pow((y-v),2));
       }

float4 summation(
       float2 coords: WPOS,
       uniform samplerRECT textureY) : COLOR {
              float2 topleft = ((coords-0.5)*2.0)+0.5;
              float4 result;
              float4 val1 = texRECT(textureY, topleft);
              result.r = val1.r + val1.g + val1.b + val1.a;
              float4 val2 = texRECT(textureY, topleft+float2(1,0));
              result.g = val2.r + val2.g + val2.b + val2.a;
              float4 val3 = texRECT(textureY, topleft+float2(1,1));
              result.b = val3.r + val3.g + val3.b + val3.a;
              float4 val4 = texRECT(textureY, topleft+float2(0,1));
              result.a = val4.r + val4.g + val4.b + val4.a;
              return result;
       }
```

Figure 4.2.2:
Cg shaders for Average Random Walk Distance

4.3 IPSA

Section 2.1.3.2 describes the compiling and organizational problems of the original IPSA source code. In order to compile the program, the team repackaged all of the source code's classes into a Java package called IPSA. The team added a class to the IPSA package called Protein_Path.

Protein_Path contains several static variables that all of the classes within the IPSA package can access. These variables include the path to the protein files, the path to the profile directory, and counters for keeping track of execution times of different areas of the code. The team replaced all of the hard-coded, system-specific paths in the original source code with the path variables in the Protein_Path class. The repackaging of the source code and addition of

Protein_Path gives the team the ability to quickly port the source code to a new system by simply changing the variables within the Protein_Path class.

4.3.1 IPSA Profiler

Since the creator of IPSA implemented the algorithm in Java, the team wrote the IPSA Profiler in Java. The layers variable is a standard Java Stack instance. The two maps, totals_with_path and totals_basename, are standard Java TreeMap instances. The out variable is a standard Java PrintStream instance of a FileOutputStream instance.

The analysis portion of the application parses the log file based on the carriage return character and the ": " delimiter. The entries for each layer implement the Comparable interface. The analyzer uses multiple implementations of the Comparator interface to sort the results based on layer name, and ascending/descending execution times.

4.3.2 Java-C++ Interaction

The Java program uses the Runtime.exec() method to execute an operating system-specific command that runs the C++ program. The Runtime.exec() method returns a standard Java Process object. The Process object provides methods for controlling the process. These methods include accessing the process' input and output streams.

The Java program creates a BufferedReader and BufferedWriter from the process' input and output streams. The BufferedReader instance contains a method for checking if the stream has data available and a blocking method for reading a line. The team uses these two methods to perform nonblocking reads from the C++ program. Similarly, the BufferedWriter contains a method for writing characters and strings to the output stream and a method for flushing the output stream.

The C++ program reads and writes data to the Java program using standard C printf and scanf functions or C++'s cout and cin. Both programs write data as characters. When reading numbers, the C++ program uses atoi(), atof(), and atoll() to convert the strings into numbers. The Java program uses comparable conversion functions as well.

4.3.3 The Compute_RMSD_D1 Function

The Compute_RMSD_D1 function performs comparisons on thousands of chains at once. The two dimensional arrays of chains are stored on the GPU as two Brook streams with length and height of size N, where N is a power of two. N is currently set to 1,280. The program uses eight N-by-N Brook streams to store the two sets of chains to compare, the three concurrent execution lines on the sets of chains, and three extra arrays for intermediate calculations. The program also uses three N/4-by-N/4 Brook streams for storing reduce operations and intermediate sets of three-by-three matrices.

4.3.3.1 CPU Arrays to GPU Arrays

When writing the arrays to GPU memory and retrieving the results, the corresponding CPU arrays must be contiguous blocks of memory. Thus, to store an N-by-N array of float4s in CPU memory, the program allocates a one dimensional array of size N^2. When referencing this CPU array, the program uses a simple mapping function that maps to dimensional coordinates to the one dimensional index space.

At the beginning of program execution, the program translates each protein chain into a four-by-four array of float4s as described in Section 3.2.4. After translating a protein chain into an array of float4s, the program inserts the protein chain into the appropriate giant one dimensional CPU array using the previously mentioned mapping function. The program writes

the final one dimensional CPU arrays to the Brook streams once all of the protein chains are placed in the arrays.

4.3.3.2 Code Translation

After initialization, the program begins the GPU-based computations. Once the GPU-based computations begin, the program does not perform any read/write operations between the CPU and GPU memory until the end of the algorithm. Since the algorithm includes numerous kernels and summation operations, this section will only describe the translation process for a few cases.

As described in Section 3.3.4.1, Step 4 of the Compute_RMSD_D1 function calculates the three-by-three matrix tR. Figure 4.3.3.2a depicts Step 4 as the Java code in the original IPSA source code. The for loop performs a summation on a series of multiplication operations. The first step towards a GPU-based algorithm is to convert the for loop into a calculation on float4s as shown in Figure 4.3.3.2b's C++ code. The three-by-three matrix tR translates to a three-by-one array of float4s as described in Section 3.2.3 while the two chains translate into an array of float4s with the array length divided by three.

```
for( i = 0; i < chain1.length/3; i++ ){
    a1 = i * 3;
    a2 = a1 + 1;
    a3 = a2 + 1;

    tR[0][0] += chain1[a1] * chain2[a1];
    tR[0][1] += chain1[a1] * chain2[a2];
    tR[0][2] += chain1[a1] * chain2[a3];

    tR[1][0] += chain1[a2] * chain2[a1];
    tR[1][1] += chain1[a2] * chain2[a2];
    tR[1][2] += chain1[a2] * chain2[a3];

    tR[2][0] += chain1[a3] * chain2[a1];
    tR[2][1] += chain1[a3] * chain2[a2];
    tR[2][2] += chain1[a3] * chain2[a3];
}
```

Figure 4.3.3.2a:
Original Java Calculation of tR

```
for( i = 0; i < sizeof(chain1); i++ ){
    tR[0].r += chain1[i].r * chain2[i].r;
    tR[0].g += chain1[i].r * chain2[i].g;
    tR[0].b += chain1[i].r * chain2[i].b;

    tR[1].r += chain1[i].g * chain2[i].r;
    tR[1].g += chain1[i].g * chain2[i].g;
    tR[1].b += chain1[i].g * chain2[i].b;

    tR[2].r += chain1[i].b * chain2[i].r;
    tR[2].g += chain1[i].b * chain2[i].g;
    tR[2].b += chain1[i].b * chain2[i].b;
}
```

Figure 4.3.3.2b:
Translation of tR Calculation to Float4s

Once in float4 format, the programmer can break the for loop into Brook kernels that perform a uniform operation across all of the elements of a Brook stream. As Figure 4.3.3.2b shows, the loop consists of three separate sections of calculations. The first element of the float4 version of tR is the summation of all of the multiplications of chain1's red value with chain2's red, green, and blue values with tR's red, green, and blue values summing the multiplications with chain2's corresponding color. The second element of the float4 version of tR is similar, except it uses chain1's green value, while the third element uses chain1's blue value.

After analyzing the float4 version of the tR calculation, one sees that this operation is six separate operations mapped onto all of the elements of chain1 and chain2. The tR calculation consists of the three separate multiplication operations, and then three summation (reduce) operations that sum the results from the three multiplication operations. Figure 4.3.3.2c shows these steps in C++ code.

Multiplication Operations

```
for( i = 0; i < sizeof(chain1); i++ ){
  temp1[i].r = chain1[i].r * chain2[i].r;
  temp1[i].g = chain1[i].r * chain2[i].g;
  temp1[i].b = chain1[i].r * chain2[i].b;
}

for( i = 0; i < sizeof(chain1); i++ ){
  temp2[i].r = chain1[i].g * chain2[i].r;
  temp2[i].r = chain1[i].g * chain2[i].g;
  temp2[i].r = chain1[i].g * chain2[i].b;
}

for( i = 0; i < sizeof(chain1); i++ ){
  temp3[i].r = chain1[i].b * chain2[i].r;
  temp3[i].r = chain1[i].b * chain2[i].g;
  temp3[i].r = chain1[i].b * chain2[i].b;
}
```

Summation Operations

```
for( i = 0; i < sizeof(chain1); i++ ){
  tR[0].r += temp1[i].r;
  tR[0].g += temp1[i].g;
  tR[0].b += temp1[i].b;
}

for( i = 0; i < sizeof(chain1); i++ ){
  tR[1].r += temp2[i].r;
  tR[1].g += temp2[i].g;
  tR[1].b += temp2[i].b;
}

for( i = 0; i < sizeof(chain1); i++ ){
  tR[2].r += temp3[i].r;
  tR[2].g += temp3[i].g;
  tR[2].b += temp3[i].b;
}
```

Figure 4.3.3.2c:
tR's Three Multiplication and Three Summation Operations

An important observation is that each operation, which is shown as a for loop, operates on a single element of each array and the elements of all arrays in an operation have the same

index. Due to these facts, the operations are parallelizable and thus suitable for GPU-based computation. Figure 4.3.3.2d demonstrates the three multiplication operations and the summation operation as three Brook kernels and one Brook reduction.

```
kernel void tR1( float4 a<>, float4 b<>, out float4 c<> ){
    c.x = a.x * b.x;
    c.y = a.x * b.y;
    c.z = a.x * b.z;
    c.w = 0.0;
}

kernel void tR2( float4 a<>, float4 b<>, out float4 c<> ){
    c.x = a.y * b.x;
    c.y = a.y * b.y;
    c.z = a.y * b.z;
    c.w = 0.0;
}

kernel void tR3( float4 a<>, float4 b<>, out float4 c<> ){
    c.x = a.z * b.x;
    c.y = a.z * b.y;
    c.z = a.z * b.z;
    c.w = 0.0;
}

reduce void sum( float4 a<>, reduce float4 result<> ){
    result += a;
}
```

Figure 4.3.3.2d:
Brook Kernels and Brook Reduction

The kernels and reduction are similar to normal C++ functions. Each variable a, b, c, and result represent a single pixel of their respective textures. The sum function performs a reduce operation that is dependent on the size of the input and output textures. For example, if the first texture is a four-by-four matrix and the result is a two-by-two matrix then the sum function reduces each two-by-two matrix in the first texture into a single element in the result texture.

In practice, the tR calculation requires eight textures. chain1, chain2, extrac1, extrac2, and extrac3 are N-by-N Brook streams while r1_x, r2_x, and r3_x are N/4-by-N/4 Brook streams. chain1 and chain2 contain thousands of chains as described in Section 3.2.4. Figure 4.3.3.2e describes the consecutive texture operations for calculating tR.

```
tR1( chain1, chain2, extrac1 );
tR2( chain1, chain2, extrac2 );
tR3( chain1, chain2, extrac3 );
sum( extrac1, r1_x );
sum( extrac2, r2_x );
sum( extrac3, r3_x );
```

Figure 4.3.3.2e:
The tR Calculation's Kernel Calls

The three kernels tR1, tR2, and tR3 write the appropriate multiplication results to all of the pixels of extrac1, extrac2, and extrac3. After the multiplication operations, the three summation operations on the three extra textures produce the tR matrix. Since the extra textures are N-by-N and r1_x, r2_x and r3_x are N/4-by-N/4, the summation operation effectively sums each four-by-four matrix within the extra textures and places them into a single element of the result textures. Thus after completing all of the summation operations, r1_x contains the first element of each chain comparison's tR float4 array while r2_x contains the second element and r3_x contains the third element.

5 System Performance and Evaluation

The team used a millisecond timer for all performance evaluations. Windows, and most other modern computer systems, do not offer timers that are better than one millisecond. Even though the team used a millisecond timer, all computations need to take at least several seconds in order to accurately estimate the relative performance.

The team compared the relative performance of each program by timing the GPU-based computation and timing the same computation performing on the CPU. In all cases the computations were repeated hundreds or thousands of times in order to make the computations last for several seconds to several minutes. In the case of large dataset sizes that do not fit into a single texture, the team continually wrote values to and read results from the GPU memory to attain timings for the large datasets.

5.1 GPGPU Limitation Demonstrations

5.1.1 2D Arrays versus 1D Arrays

The 2D Arrays versus 1D Arrays program clearly shows that 2D textures are vastly superior to 1D textures. Figure 5.1.1 shows the results from running the map operations described in Section 4.1.1 on 2D textures and 1D textures. The left graph shows the 2D texture's performance relative to the CPU's performance while the right graph shows the 1D texture's performance relative to the CPU. Both horizontal axes are logarithmic. An important note is that the left graph's vertical axis is logarithmic while the right graph's is linear. Thus, at 10,000,000 elements, the 2D texture implementation was 10,000 times faster than the CPU while the 1D texture implementation speed was about equal to the CPU speed.

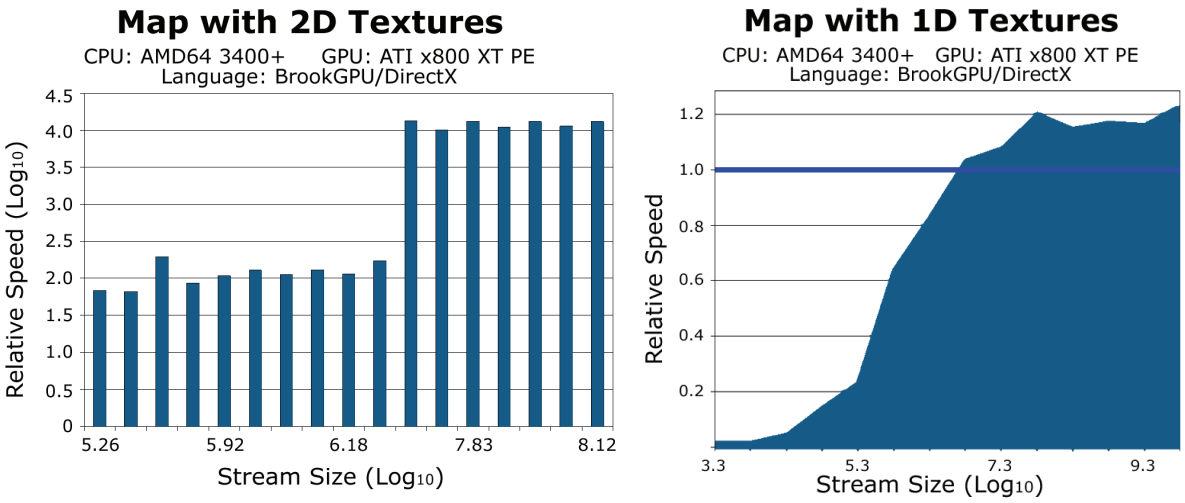

Figure 5.1.1: 2D Arrays versus 1D Arrays Results

5.2 GPGPU Potential using Cg

For the experienced GPGPU developer, Cg provides an excellent low-level interface to the underlying hardware. Therefore, much more optimization is possible using Cg than with BrookGPU. Despite the programmer's increased control when using Cg, the language still suffers from many of the same limitations as BrookGPU. Indeed, highly mathematical computations that run in parallel over large datasets prove much faster on the GPU. However, common CPU operations, such as the summation, limit the realized performance gain. The choice of language will require future developers to consider the need to quickly implement large sections of code with BrookGPU versus the desire to optimize algorithms for an even greater speedup using Cg.

The team compared each Cg algorithm to a CPU counterpart. This comparison ensured the accuracy of the Cg algorithms and provided the basis for performance evaluation. Each algorithm features a significant speedup over the comparable CPU version due to the use of fundamental GPGPU programming techniques.

5.2.1 Mathematical Computation

The highly mathematical and parallel nature of the Mathematical Computation algorithm demonstrates the keen ability of the GPU to evaluate numbers over large datasets. As shown in Figure 5.2.1, small arrays have little or no gain over the CPU since the cost of data transfer remains very high. However, the algorithm grows faster with the size of the data. At the largest texture size, 4096-by-4096 pixels, the GPU performs nearly 175 times faster than the CPU.

Although the GPU only features 128 parallel processors, this speedup is still extremely plausible. Since the algorithm computes the function for over sixteen million items, the cost of data transfer is low in comparison to the cost of CPU computation. Furthermore, the GPU has been optimized for graphics processing and can process multiple floats in a single clock cycle. As such, an increase greater than 128 times should be expected given the intended purpose of the GPU to efficiently process numerical data.

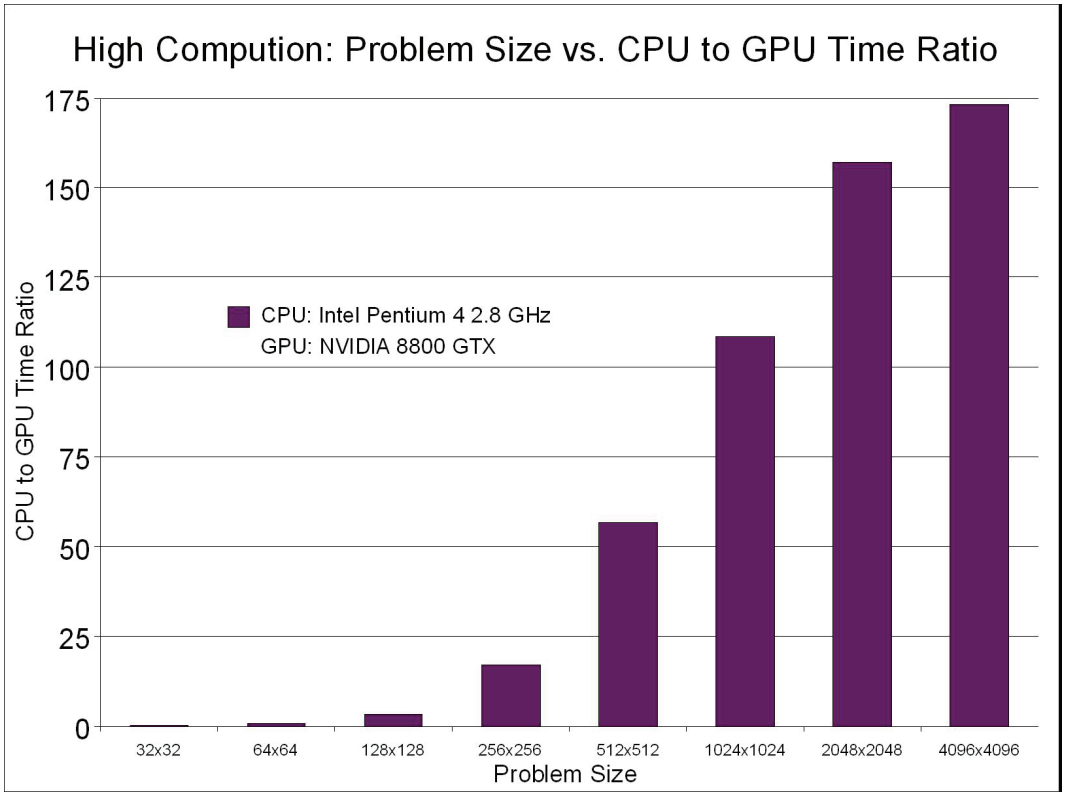

Figure 5.2.1: High Computation: Problem Size vs. CPU to GPU Time Ratio

5.2.2 Average Random Walk Distance

Average Random Walk Distance benefited greatly from the use of GPGPU techniques in Cg. As with Mathematical Computation, Random Walk Distance features an extensive mathematical function that will be evaluated over each element. Furthermore, protein folding algorithms that feature random walks will likely require the evaluation of very large datasets to cope with the growing database of proteins in the Protein Data Bank [5]. At the heart of this algorithm's increase in speed is its use of parallel computation over a very large texture.

As shown in Figure 5.2.2, Random Walk Distance achieves a significant speedup on the GPU while not as great as the speedup realized by Mathematical Computation. The GPU evaluates the equation that Random Walk Distance implements much faster than the CPU. However, the need to reduce the texture into a single value in order perform a summation hinders the ability of the GPU version to outperform the CPU. Therefore, Random Walk Distance performs a respectable 40 times faster than the CPU algorithm using the largest texture size.

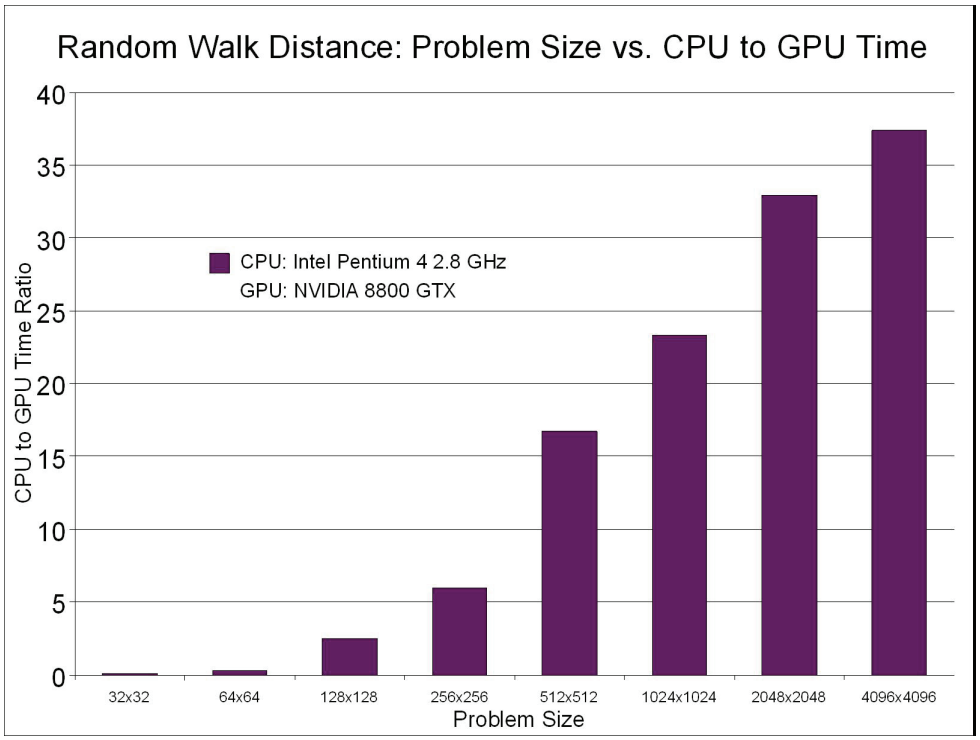

Figure 5.2.2: Random Walk Distance: Problem Size vs. CPU to GPU Time Ratio

5.3 IPSA

The results of the IPSA programs show that the team successfully improved the performance of IPSA, and that further performance gains are unlikely given IPSA's current design. The IPSA Profiler results show that the team translated the majority of GPU-compatible code when implementing the Compute_RMSD_D1 function. The Compute_RMSD_D1 function results shows that GPU processing can significantly increase performance.

5.3.1 IPSA Profiler

The team used the IPSA Profiler's results to determine which portions of IPSA to transfer to the GPU. Figure 5.3.1a shows the highest-level results from the IPSA Profiler. IPSA spends eighty-eight percent of its processing time in a section called Extend. IPSA spends the remaining twelve percent in a function called Align Items.

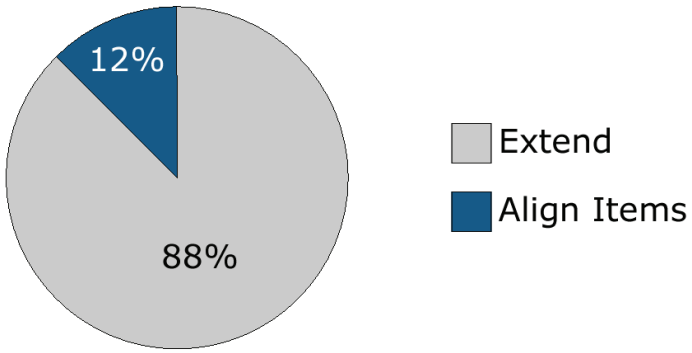

Figure 5.3.1a: IPSA Profile Layer 1

Figure 5.3.1b breaks the two top layers down into subsections. Align Items breaks down into two seconds, Compute D1 and Align Items Branching. Compute D1 consumes 7% of IPSA's total processing time while Align Items Branching consumes 5% of the total processing time. The Extend section consists of Extend Branching, Compute D2, and Matrix Multiply.

Extend Branching consumes 85.7% of IPSA's total processing time while Compute D2 and Matrix Multiply consume 2% and 0.3% respectively.

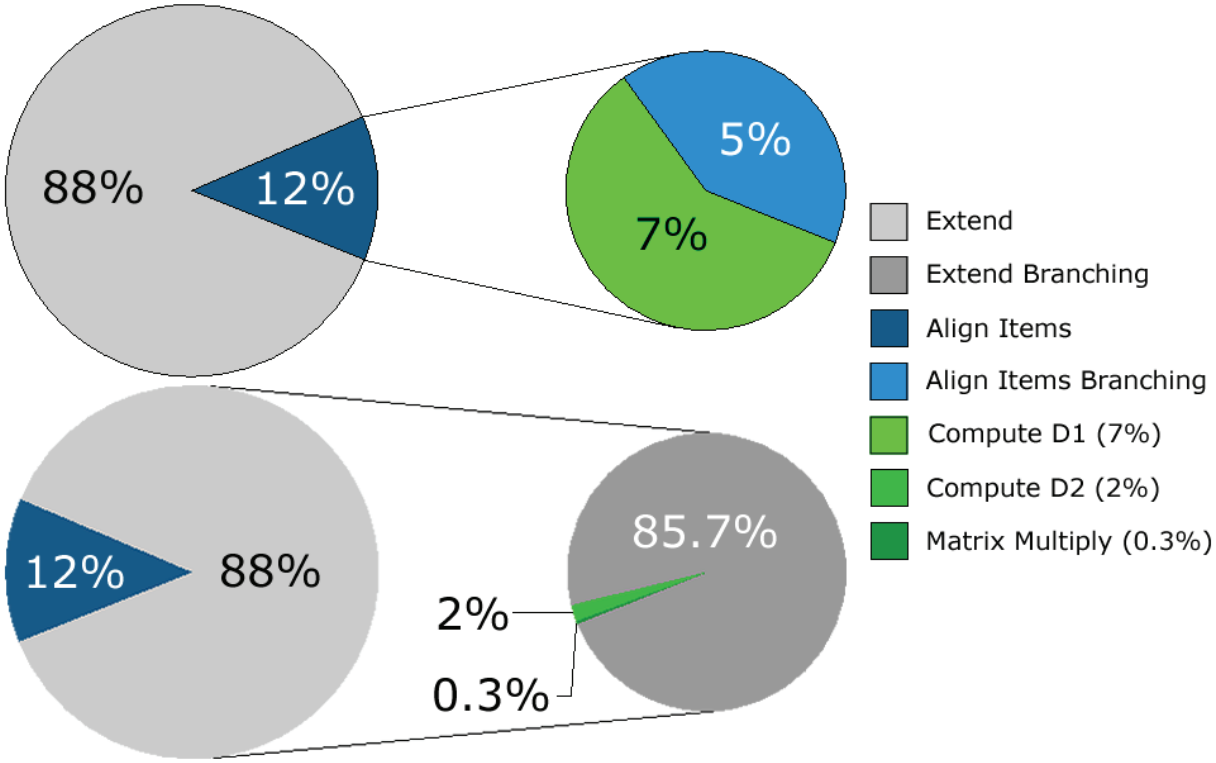

Figure 5.3.1b: IPSA Profile Layer 2

As the section names suggest, Align Items Branching and Extend Branching both contain significant amounts of branching and control structures. As described in Section 4.1, these branching and control structures are very costly to perform on a GPU. Thus, Align Items Branching and Extend Branching, which collectively account for 90.7% of IPSA's total processing time, are not suitable for GPU-based processing in their current form. These sections average three to five instructions per if statement and contain multiple sections of up to three nested for loops.

The sections in Figure 5.3.1b that are labeled in green colors are the only sections of IPSA that are applicable for GPU-based processing in their current form. These sections

collectively account for only 9.3% of the algorithm's total processing time. Thus, if the team were able to completely eliminate this processing time, the algorithm would only be 1.1 times faster than the CPU-based IPSA. Logically, the team chose to translate Compute D1, which is shorthand for the Compute_RMSD_D1 function, to the GPU.

5.3.2 The Compute_RMSD_D1 Function

As described in Section 3.3.4.2, the Compute_RMSD_D1 function includes calculating eigenvalues and eigenvectors. Due to project constraints, the team hard-coded the eigenvalues and eigenvectors into the prototype. The team eliminated the eigenvalue and eigenvector processing time from the CPU-based timing result by timing the calculation of thousands of eigenvalues and eigenvectors and dividing by the number of sets of eigenvalues computed. After determining the average time for computing one comparison's eigenvalues and eigenvectors, the team multiplied the average time by the number of comparisons in the actual test and subtracted the result from the CPU-based processing time.

The team performed the Compute_RMSD_D1 function tests on Computer 2. Due to limitations of Computer 2's video card, the largest texture that the computation could use was 1280-by-1280. The GPU-based Compute_RMSD_D1 function can compute 102,400 chain comparisons simultaneously. The GPU-based function is 9.828 times faster than the CPU-based function. Not including any potential overhead from connecting the GPU-based function with IPSA as detailed in Section 3.3.2, the GPU-based IPSA algorithm is 1.076 times faster than IPSA and cuts 84 seconds off the total processing time.

6 Summary and Conclusions

The team successfully completed the goals and objectives in Section 1.3. The Cg programs described in Section 4.2 and Section 5.2 fulfill the requirements of the project's first phase. The map program described in Section 4.1 and Section 5.1, and the Map-Reduce program included in the report's documentation meet the requirements of the second phase. The team completed the third phase by developing and using the profiling program described in sections 3.3.1, 3.4.1, 4.3.1, and 5.3.1. The team completed the final phase by developing the GPU-based Compute_RMSD_D1 function prototype described in sections 3.3.4, 3.4.3, 4.3.3, and 5.3.2 and by developing the IPSA-GPU interaction prototype described in sections 3.3.2, 3.4.2, and 4.3.2.

The team successfully overcame the project's many constraints, including the team members' complete lack of experience with GPU programming, the significant human factor constraints described in Section 2.1.3.2, and the sheer difficulty of the problem. The team has proven that GPU-based computation can significantly improve the average response time of algorithms that are parallelizable. The team's GPU-based Compute_RMSD_D1 function could potentially eliminate 84 seconds of IPSA's total processing time. The team has also shown that further GPU-based improvements of IPSA will require a redesign of the algorithm that reduces branching and extended nested looping.

7 Future Work

The GPU performance gain is limited by the small percentage of total processing time for IPSA's parallelizable code sections (Section 5.3.1), the use of only three of the four floats in each pixel on the GPU (Section 3.2.2), and the unused float4's from the texture packing strategy (Section 3.2.4). In order to overcome the two limitations pertaining to float4 conversion, the GPU-based algorithm would need to be completely redesigned. This redesign would most likely provide a performance gain that is not worth the amount of time required to complete the redesign. The performance gain would also be insignificant because the Compute_RMSD_D1 function is only 7% of IPSA's total processing time.

Future development can only achieve worthwhile performance gains through redesigning the IPSA algorithm in a way that increases the number of parallelizable sections. IPSA's current design includes an abundance of branching and extensive nested looping. Improvements would also be easier to develop if the original IPSA algorithm was thoroughly documented, and if the algorithm and all of its supporting data structures were rewritten in C++.

Aside from potential performance gains, integrating the GPU-based program into the IPSA algorithm requires that the calculation of eigenvalues and eigenvectors be ported to the GPU. This process, which is long and tedious, could be a semester or yearlong project in itself. Additionally, the team did not study whether the GPU's single precision floating point numbers will affect the accuracy of IPSA, which currently uses double precision floating point numbers. Eventually, video cards will support double precision floating point numbers. In the meantime, IPSA must accept the single precision limitation in order to reduce the average response time.

8 References

[1] Chi, P. and Shyu, C., "Efficient SCOP fold classification and retrieval using Index-based Protein Substructure Alignments (IPSA)," *Submitted to Intelligent Systems for Molecular Biology (ISMB)*, 2007.

[2] Chi, P.; Scott, G.; Shyu, C., "A fast protein structure retrieval system using image-based distance matrices and multidimensional index," *in Int J. Softw. Eng. Know., Special Issue on Software and Knowledge Engineering Support in Bioinformatics*, 2005, pp. 527-545.

[3] Strom, D., "5 Disruptive Technologies To Watch In 2007," Information Week, Jan. 2007. Available: http://www.informationweek.com/internet/showArticle.jhtml?articleID=196800208

[4] Owens, J.; et al, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, Vol. 26 No. 1, pp. 80-113, Mar. 2007.

[5] "Yearly Growth of Total Structures." RCSB Protein Data Bank, [Online Document], 2007 Feb 13, [cited 2007 Feb 18], Available HTTP: http://www.pdb.org/

[6] Luebke, D.; Humphreys, G., "How GPUs Work," *Computer*, Vol. 40, No. 2, pp. 96-200, Feb. 2007.

[7] Treseler, Mary, Ed., *The Redbook*. Addison Wesley Publishing Co., Jan 1997.

[8] "DirectX Developer Center," Microsoft. Available: http://msdn.microsoft.com/directx/

[9] Mark, W.; et al, "Cg: a system for programming graphics hardware in a C-like language," *in ACM SIGRAPH*, 2003, pp 896-907.

[10] "RapidMind," RapidMind. Available: http://www.rapidmind.net

[11] "PeakStream," PeakStream. Available: http://www.peakstreaminc.com

[12] "NVIDIA CUDA," NVIDIA. Available: http://developer.nvidia.com/object/cuda.html

[13] "Sh: A high-level metaprogramming language for GPUs," Sh. Available: http://libsh.org

[14] "Shallows: Making GPGPU programming fast and easy," Shallows. Available: http://shallows.sourceforge.net

[15] Tarditi, D.; Puri, S.; and Oglesby, J. “Accelerator: using data parallelism to program GPUs for general-purpose uses,” *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 325-335, Oct. 21 - 25 2006.

[16] Buck, I.; et al. “Brook for GPUs: stream computing on graphics hardware,” *ACM SIGGRAPH 2004 Papers*, pp. 777-786, Aug. 8 - 12 2004.

[17] Govindaraju, N.; Larsen S.; Gray, J.; and Manocha, D., “A Memory Model for Scientific Algorithms on Graphics Processors,” *in Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, no. 89.

[18] Ikeda, T.; Ino, F.; Hagihara, K., "A code motion technique for accelerating general-purpose computation on the GPU," *2006 Parallel and Distributed Processing Symposium*, 10 pp, Apr. 25-29 2006.

[19] Fan, Z.; Qiu, F.; Kaufman, A.; and Yoakum-Stover, S., “GPU Cluster for High Performance Computing,” *Supercomputing. Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pp. 47-59, Nov. 2004.

[20] "CIRL GPU," Robert Luke and Derek Anderson. Available: http://cirl.missouri.edu/gpu/

[21] "GPGPU Tutorials," Dominik Göddeke. Available: http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/

[22] "GeForce 8800 GTX," Pricewatch.com. Available: http://www.pricewatch.com/video_cards/geforce_8800gtx_768mb.htm

[23] "Extreme Programming: A Gentle Introduction," ExtremeProgramming.org. Available: http://www.extremeprogramming.org/map/project.html