Enabling Implementation and Optimization of Scientific Algorithms via Graphics Processing Units

James C Sweet

Publication Date

17-10-2017

ENABLING IMPLEMENTATION AND OPTIMIZATION OF SCIENTIFIC ALGORITHMS VIA GRAPHICS PROCESSING UNITS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

James C. Sweet

Douglas Thain, Co-Director

Jesús Izaguirre, Co-Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
October 2017

ENABLING IMPLEMENTATION AND OPTIMIZATION OF SCIENTIFIC ALGORITHMS VIA GRAPHICS PROCESSING UNITS

Abstract

by

James C. Sweet

GPUs provide a fast but simplified architecture that leads to major challenges in implementation of algorithms. This document looks at how this can be applied to scientific problems that require both rigor and performance.

This work looks at solving three different problems by utilizing the GPU for visualization and computation. Firstly, it looks at exploring better methods of creating accurate and realistic 3D models from laser scan data and photographic images. Secondly, it explores the implementation of a coarse-grained molecular dynamics algorithm on the GPU and the how the factors of the simulation affect the performance and accuracy. Finally, it looks at transitioning part of an MPI particle laden flow simulation code to utilize the GPU as well as the challenges associated.

This work demonstrates that by implementing algorithms to utilize the power of the GPU provides an opportunity for providing scholars to be able to research more effectively and efficiently. Firstly, by using GPUs, effective on-site feedback can be provided for architects and archaeologists who are collecting data as well as off-site scholars attempting to explore this data after collection. Secondly, the performance of molecular dynamics simulations can be improved by $\approx 6\times$ whilst keeping the accuracy, allowing for faster drug research and development. Finally, a particle laden simulation's MPI particle update calculations can be migrated to the GPU to improve

the performance by $\approx 14\times$, allowing for more accurate exploration of areas such as pollution dispersion in the atmosphere.

To my remarkable wife Rachael, whose sacrifices afforded our adventures together, and I look forward to many more.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

I am extremely grateful for many people throughout my time at the University of Notre Dame, who have welcomed me to the United States and helped me succeed.

Firstly, to Dr. Jesús Izaguirre, for helping me take the knowledge that I had gained from my undergraduate degree and finding a way to apply it to something that could change the world. I am thankful to Dr. Douglas Thain, for accepting me as his student when Dr. Izaguirre took a job in the real world, as well as his guidance to help me complete my Ph.D.

To my dissertation committee members, Dr. Paul Brenner and Dr. Michael Niemier, who have supported and guided me. I would like to especially thank Dr. Krupali Krusche for providing me with a rare opportunity to carry out research in a field new to me, as well as allowing me to explore and document historical sites in Rome and India. Thanks to Dr. Charles Vardeman II for his help proof reading, many discussions on a range of topics and divergent ideas. Thanks to my colleagues, now good friends in the Laboratory for Computational Life Science group, RJ Nowling, Badi' Abdul-Wahid, Kevin Kastner and Haoyun Feng, for providing a great amount of support, knowledge and enjoyment. I would like to thank Dr. Jaroslaw Nabrzyski, for initially encouraging me to apply to join the Ph.D. program at Notre Dame. I would also like to thank the Center for Research Computing, for providing access to a vast verity of resources, which allowed this research to be possible. Finally, a big thank you to my parents, Chris and Fiona, for their never ending love and support. They've inspired me from a young age to pursue my passions, and I am proud to be who I am today.

CHAPTER 1

INTRODUCTION

In the last 20 years, the gaming industry's quest for ever greater visual immersion has created graphics processing units (GPUs) with immense power, which are becoming widely available at a low cost. These GPUs are designed to work on large numbers of pixels on the screen simultaneously as a Single Instruction Multiple Data (SIMD) architecture. Software developers became aware that the GPU's computational power, measured in terms of floating point operations, dwarfed that of a typical CPU. This spawned many efforts to harness this power, with initial attempts using the graphics pipeline to perform the calculation through OpenGL's built in texturing and blending operations. This progressed to the use of programmable shaders as graphics cards evolved. These shaders allowed developers to change how the GPU interacted with data at different places within the pipeline. Eventually, the GPU manufacturers realized the financial opportunities of general purpose programming on the GPU. They began to implement programming languages that abstracted away the use of the graphics pipeline and added additional functionality, such as CUDA and OpenCL.

However, implementing software to optimally utilize these highly performant acceleration tools is generally difficult because knowledge of the underlying hardware, which changes rapidly for each new GPU architecture iteration, is required. The introduction of GPU programming languages (CUDA and OpenCL) has bought "C"-like programming techniques to the platform. Although this allows straightforward implementation of parallel algorithms, this simplistic approach produces an

executable which often fails to achieve the expected performance due to the architectural complexity. For instance, there is a large overhead when transferring data to and from the GPU, data access is less optimal as there is much less cache available, and due to the simplicity of the cores, branch prediction is much worse than a CPU. Despite these limitations, experience shows these architectures can yield spectacular results, which makes it a worthwhile goal for software developers to pursue this field. This is helped by the continual improvement of available analytical tools, although this may be restricted to the latest generation of GPUs.

In this dissertation I explore the techniques available for implementing scientific algorithms for modeling physical systems in the GPU/CPU environment to harness the phenomenal computational power available. I begin by exploring GPGPU calculations in the OpenGL pipeline and document a use of the resulting developed platform. I extended this work to simulating molecular dynamics using the CUDA language in the context of a single GPU in a single compute node. I further extended these techniques to the hybrid GPU/CPU solution of particle laden flow simulations in a multi-node cluster with the addition of a single node equipped with multiple GPUs, using CUDA and Message Passing Interface (MPI).

1.1 Problems

In the following chapters I will introduce three example problems that I have decomposed to utilize the GPU in an efficient manner.

Firstly, I will talk about implementing algorithms and tools to allow researchers to combine laser point cloud data and digital images to produce highly accurate 3D models and visualize them. This work looks at the challenges of rendering high fidelity data using GPUs. This utilizes the generalized graphics capabilities of the GPU such as texturing and requires generation and loading of view matrices to produce a holistic model.

Second, I will talk about implementing a Molecular Dynamics (MD) integration algorithm on the GPU to gain an order of magnitude performance improvement over the reference CPU version. This work looks at the challenges with implementing C++ algorithms on the GPU using CUDA.

Finally, I will talk about implementing a Fortran Lagrangian particle calculation algorithm on the GPU. This work takes the knowledge learnt from the previous problem and introduces the added complexity of interfacing between Fortran and C++ and using Message Passing Interface (MPI) for multi-node execution. I have also extended this solution to multiple GPUs in a single node.

1.2 Graphics on the GPU

Although current data acquisition technologies, such as LIDAR and GigaPan, allow the collection of vast amounts of data on Heritage sites, the end users for the data are Archaeologists and Architects who require the presentation of the data as a Holistic model in order to navigate it. The target users for the LIDAR systems are Civil Engineers who are used to working with the point-clouds, often enhanced by representing the points a colored spheres. There exists a "gap" in the technology to build a surface from the point-clouds in order for the GPU to be able to texture the model. In addition, the images acquired from a robotic tripod/DSLR camera need to be scaled, rotated and registered with the point-cloud for the texturing to take place. The panoramic "stitching" software could not be used since it requires the camera to be set at a fixed focus, impractical for the proposed use here.

Despite the mass of work on the rapidly evolving field of graphics on the GPU, there was no solution available that could merge multiple data sets through the concept of data registration, where the spacial mapping is distinct from the images themselves. Current solutions could only build a textured model with a fixed resolution in both point-cloud density and image resolution.

1.3 Molecular Dynamics

Molecular dynamics (MD), a type of N-body simulation, is a computer simulation method for studying the physical movements of atoms and molecules. By allowing atoms to interact with each other over a period of time, a dynamical evolution of the system can be observed. Most commonly, the trajectories of atoms and molecules are determined by numerically solving Newton's equations of motion for a system of interacting particles. Forces between the particles and their potential energies are calculated using inter-atomic potentials or molecular mechanics force fields.

Traditional MD simulations are constrained in length by timestep limits. Studies by our group and others have shown that traditional MD is limited to timesteps of about 2 fs due to high-frequency resonance [39, 51, 74]. Even the most basic biologically relevant motions occur on the microsecond to millisecond range, which is 9 to 12 orders of magnitude greater than the timesteps possible with traditional MD. Further, each step requires a costly force calculation ($O(N)$ to $O(N^2)$). As such, simulating medium-size proteins often requires months of computer time on a large distributed system such as Folding@home [8, 80] to simulate milliseconds of dynamics. Similarly, simulating a large protein (e.g. the β-2 Adrenergic Receptor) on the more interesting biologically-relevant time scales (milliseconds through hours) using a standard desktop computer would take years. Thus, it is not feasible to simulate timescales of real biological interest without substantial advances in MD methods.

1.4 Lagrangian Particle Laden Flow

Computational fluid dynamics is typically carried out by designing a parallel program that is decomposed in the spatial domain, such that each processor operates on a portion of the fluid flow, and periodically exchanges a halo of states with its neigh-

bors. This long-used method is highly effective at Eulerian computational methods, such that the partitioning is static relative to Cartesian dimensions. An example of this is the NCAR-LES simulation code [55], which has been ported to multiple HPC architectures and used to simulate atmospheric turbulence in many different applications.

Recently, this code has been augmented by adding spray particles superimposed over the turbulent flow. This collection of particles represents small droplets that are carried by the flow, such as might be seen at the crest of an ocean wave. The most direct way to add particles to the existing flow simulation code is to again perform a spatial partitioning and then alternate the particle and flow calculation at each node, exchanging states as needed between each time step. Formally this is referred to as Operator Splitting. However, due to the nature of the particle step, the time to compute the particle step quickly exceeds the flow step as the number of particles is scaled up.

1.5 Overview

In this chapter, three problems were defined as well as their importance. Chapter 2 explores the related work for the three separate problems. Chapter 3 examines the hardware design of GPUs and explores how features are implemented using CUDA. Chapter 4 presents work to solve the problem of data collection and fusion to produce highly accurate 3D representations of historical preservation sites. It allows the computed result to be used in a multi-resolution setting where the complexity can be defined depending on the hardware available and the response time required. Chapter 5 presents work that utilizes a GPU implementation of a coarse grained algorithm (LTMD). It shows that through this implementation, a $\approx 6\times$ performance improvement is achievable and that as the problem size increases, the performance improvement over traditional methods also increases. Chapter 6 presents work that

moves the Lagrangian particle update step from a parallel MPI implementation to use four GPUs. It shows that this implementation is able to achieve a $\approx 14\times$ performance improvement. It demonstrates that, for this work, it is more efficient to scale the number of GPUs available to the system rather than the number of machines/-cores available to MPI. Finally, Chapter 7 summarizes this work and explores future possibilities.

CHAPTER 2

RELATED WORK

2.1 Digital Heritage Preservation and Visualization

In the recent years, laser scanning or Light Induced Detection and Ranging (LIDAR), has become a major technique that has started replacing the use of hand measuring and surveying for survey and documentation work. A laser scanner is an automated surveying apparatus that uses a laser beam to collect location coordinates from the surface of a desired object. These measured coordinates are recorded in the form of a point cloud that identifies the shape of the object by converting the spatial geometry established through the x, y, and z position of that point in space. The largest producer of these LIDAR devices is Leica Geosystems [46].

A number of algorithms have been created to convert a point cloud data set to a triangular mesh. Their complexity depends on if the input point cloud data is saved in a structured, or unstructured way. If the data is structured, then a simple algorithm can be used to connect the points in a known way to produce a mesh. For unstructured data, algorithms such as the Poisson merging algorithm [43], or the Visual Computing Lab's merging filter [14], attempt to estimate the shape of input data set.

The simplest method to map images to a 3D mesh is to have a person manually drag the picture onto the model or to select features on the picture and model to connect the two. Work has been done to reduce the human error or amount of human intervention required. Janko Z. et al. [42] implemented a genetic algorithm to reduce

the pixel error caused by human error. Franken, T. et al [25] attempted to reduce the human error and level of work required by defining correspondences between pictures so that once a single picture is mapped to the model, the rest can be inferred. Whilst these methods can map the images to the 3D point clouds accurately, it is costly in time.

Automated methods aim to map the images onto the 3D model by using techniques to find correlation between the pictures and the model. Lensch, H. et al. [48] uses a silhouette-based method, which attempts to calculate the mapping by rendering a silhouette of the 3D model in different configurations and generates a fitting metric for the image. Liu. L. et al. [49] extracts geometric features from a 3D model and matches horizontal and vertical lines with 2D images to then calculating the camera parameters by using these 2D lines to find the vanishing point. Corsini. M. et al [13] follow a similar path to [48], but instead of silhouettes, they render the model and calculate a correlation between it and a grayscale 2D images. These methods provide an automated method, at the cost of loss of accuracy and a high computational workload.

Another method of creating models and mapping images to them is to use a large collection of pictures of the object and then extracting the 3D data from them pictures to create the model. Zhengyou Z. [95] calibrates a single camera with a number of different planar images and from there are able to extract 3D information from images captured with the calibrated camera. This method allows for the creation of high relative accuracy at the cost of computational time and a requirement of a large number of images of the model taken from many different angles.

Once data has been collected from these LIDAR scanners, it must be processed or visualized to be of use. LIDAR manufacturers provide tools, such as Leica's Cyclone [47] or Faro's SCENE [22], that allow for registration of multiple data sets, visualization and basic editing. A number of open source tools have been developed to

provide added functionality. Meshlab [12], designed for working with 3D meshes, is a common tool allowing people to visualize their output or edit it with a greater number of options than that of the manufacturer software. It provides users with a visual way of editing, cleaning, healing, inspecting, rendering, and converting data. For developers, Point Cloud Library (PCL) [71] has become one of the leading libraries. PCL provides developers with a wide array of functionality to work with point clouds. It implements feature detection, filtering, I/O, partitioning, searching, recognition, segmentation, registration, surface generation and visualization.

2.2 Molecular Dynamics

Due to the high computational cost to simulate molecular dynamics simulations, a number of approaches have been implemented to attempt to push the performance further.

Langevin dynamics, which solves a stochastic differential equation under dissipation-fluctuation constraints and provides an attractive thermostat, can be used to overcome instabilities due to resonances of constant energy integrators [74] and achieve larger timesteps [40]. However, even Langevin integrators require relatively small time steps for stability due to fast frequency motions within the bonded forces. As a result, algorithms such as Normal Mode Langevin (NML, [41, 88]) which can achieve timesteps 25 - 50x larger have been developed.

Distributed systems, such as Folding@Home [8, 80] attempt to simulate protein folding by running a large ensemble of simulations on as many computers as possible and then using the results to infer statistics.

Specialized hardware is also an area that has been explored. The D. E. Shaw group has developed Anton, a specialized supercomputer where MD algorithms are implemented in hardware using application-specific instruction chips (ASICs) [76–79]. For explicitly solvated systems, it has been shown that Anton can provide speed

ups of up to 2 orders of magnitude over simulations run in HPC environments.

It has been shown that the computation of MD simulations can be sped up considerably by taking advantage of GPUs [84]. NAMD [67, 83] adapted GPU support for running on large clusters. GPUs are used accelerate the computation of electrostatics and Generalized Born [62] implicit solvent model while the remaining computations and communications are handled by CPUs. Overlapping GPU non-bonded force calculation (parallelized in a similar way using blocks) with CPU communication protocols yielded a five to seven fold improvement in efficiency on NAMD [66] when running simulations of solvated models such as Apolipoprotein A1 (ApoA1, 92000 atoms) and Satellite Tobacco Mosaic Virus (STMV, 1.06 million atoms).

Other MD packages have focused on running entire MD simulations on one or more GPUs on a single workstation [33, 36]. In doing so, GPU-enabled workstations are capable of running simulations on the same timescales as large clusters at a fraction of the cost, which significantly increases access and availability for the average researcher. Friedrichs, et al. have shown that OpenMM [19, 26], a library for performing MD on GPUs, is capable of speeding up simulations of implicitly-solvated systems more than 500 times over an 8-core CPU.

2.3 Lagrangian Particle Laden Flow

Particle laden flows have a wide variety of engineering and scientific applications such as pollution dispersion in the atmosphere, fluidization in combustion processes, aerosol deposition in spray medication, along with many others. These problems often take the following form: a set of partial differential equations governing the spatial variation of some continuous quantity is discretized, parallelized, and solved on a fixed Eulerian computational mesh, while at the same time Lagrangian elements, each carrying its own information, travel throughout the domain independent of the mesh but are modified based on the computed Eulerian fields.

The implementation of the Eulerian flow methods is split into two different approaches to solving, depending on the domain size required.

Direct Numerical Simulation (DNS), used by [20], provides a method for very high accuracy, but the domain size is restricted due to the computational cost.

Large Eddy Simulation (LES), was first implemented by Moeng et al. [55] aiming to model small scales of fluid motion and directly simulate larger scales. This allows for a much larger domain size, whilst trading accuracy of very fine details. Sullivan et al. [86], aimed to make the model more accurate by enhancing the sub grid scale eddy-viscosity model Finnigan et al. [23] utilized LES to compare the turbulence statistics of the canopy/roughness sublayer and the inertial sublayer above it. Sullivan et al. [85], in 2011, implemented a massively parallel LES flow calculation software.

Richter et al. [69] extended the work of Sullivan et al. to simulate particle laden flow to explore sea spray. This research was motivated by the possible effects of spray on the drag felt by the ocean surface in high winds and used direct numerical simulation coupled with Lagrangian particle tracking to investigate how suspended inertial particles alter momentum flux in an idealized turbulent flow. These simulations showed that when inertial particles are introduced into a turbulent flow, they carry a portion of the total vertical momentum flux, and that this contribution can be significant when the particle concentration is sufficiently large.

The software used in [69] was extended in Chapter 6 to implement the particle update calculation using multiple GPUs in a single computer.

For comparable work in the Literature, GPUs are now becoming popular as accelerators and to increase the fidelity of simulations for SPH and CFD. In [56] the massive parallelism of GPUs is not only harnessed but enhanced numerics, required as the number of particles increase, are proposed. The open sourced gpuSPHASE [94] GPU implementation for SPH discusses the shared memory caching implementation required for optimal performance with CUDA. GPU acceleration of major CFD

codes is ongoing, packages such as PETSc [11] have shown excellent results, although an awareness of the implementation issues is critical [2].

Recent work in the field of Large Eddy Simulations has focused on targeting accelerators. In [73] the issues related to GPU architecture is addressed where the choice between explicit and implicit time integration is shown to rely on the convergence of explicit solvers and the efficiency of preconditioners on the GPU. In [92] Turbulent Wall-Bounded Flows, using the Lattice Boltzmann method, are solved in a Multiple GPU environment. Atmospheric boundary layer flows are computed on a GPU in [91], presenting an implementation of two time-stepping methods on the GPU and highlighting the different challenges on the programming approach. In addition the authors introduce a classification of basic CFD operations, found on the degree of parallelism they expose, and study the potential of GPU acceleration for every class. In [50] the authors highlight the comparison of GPU and CPU calculations by presenting verification and validation of HiFiLES, a High-Order LES unstructured solver on multi-GPU platforms. GPU platform specific analysis can be found in [35] where particle simulation for Kepler GPU is optimized.

CHAPTER 3

GRAPHICS PROCESSING UNIT

3.1 Hardware

In this section, we will look at the hardware design of GPUs and how it affects computation on the GPU.

A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. CPUs have evolved to support a complex instruction set and have advanced branch prediction. Many of these advancements are missing from GPU cores. These deficiencies in the GPU cores, in the context of advanced algorithms, lead to major challenges in transferring algorithms from CPU to GPU to take advantage of the massively parallel architecture.

The efficiency of using the GPU to accelerate code will ultimately depend on the GPU architecture and the algorithm implementation, for example algorithms that require global memory access will have major bottlenecks compared to algorithms with local access only. A good example of optimizing algorithms to be efficient on GPUs would be to re-formulate them to have local access only.

Some of these limitations are mitigated by enhancements implemented by the GPU manufactures to make their products more ubiquitous. A choice of a good GPU would generally focus on the number of cores as a primary measure, see Table 3.1, but this can be offset by improvements in the architecture of the Streaming

Processors (SP). In general the trend is still to fit as many cores as possible on the chip (currently 5,000) and to increase the memory capacity and performance on the card (currently 16GB/ 900GB/sec). A consequence of the increase in cores is, if anything, to decrease their complexity exacerbating the challenges of GPU programming. One variation to this trend is the addition of TensorCores (by NVIDIA) to meet the ever growing demands of the Machine Learning Community. I focus on NVIDIA's GPUs as currently, they are the most performant for GPU computation and have been the target hardware for my work.

In the following discussion I describe Streaming Processors, a key feature of GPU devices, and the impact of these on the efficiency that can be obtained.

NVIDIA's Tesla architecture introduced the idea of Streaming Processors (SP) to their platform. These SP are the main drivers of work for NVIDIA GPUs. These SP were a shift in the design of the architecture from vector Processing (used for graphics) to scalar processing (GPGPU) and only operate on one component at a time. This reduced the maximum throughput of the SP, but also made them much less complex. Because of this reduced complexity, they are more efficient in a large number of cases as VPs relied on getting an ideal instruction mix and ordering to reach the peak throughput. To compensate for the reduced throughput, more SP could fit onto the die due to the improved efficiency and the clock speed can be set higher due to the simplicity.

Figure 3.1 shows an overview of NVIDIA's SP, now called Streaming Multiprocessors (SM), design for both Pascal (2016) and Volta architectures (2017). On the left, Pascal's SM is divided into "Cores", which can calculate one FP32 or INT32 operation at a time, and "DP Units", which can calculate one FP64 operation at a time. On the right, Volta's SM is divided into FP32, FP64, INT32 and "Tensor Cores" We can see that each SM is split into multiple separate processing blocks, allowing multiple "Warps" of 32 threads at once. Each of these blocks contains its

(a) Pascal [59]

(b) Volta [60]

Figure 3.1. An architectural overview of a Streaming Multiprocessor in different architectures

own cache, scheduler, dispatcher, register file and processing units. Shared between all processing blocks is a configurable L1 Cache which allows the developer to choose between having it act as a data cache or provide a section of it as memory to be shared between threads in a warp. We can see from this figure that both architectures contain half as many FP64 ALUs compared to FP32 or INT32 ALUs in both cases. This leads to a performance drop in software that requires FP64 computation only in comparison to FP32.

Three main differences are obvious between these SMs. First, Volta's SM has four processing blocks, compared to Pascal's two. This allows Volta's SMs to have twice as many threads being executed at once, allowing for more throughput. Secondly, Volta's SM has separate INT32 and FP32 ALUs. This also allows mixed format

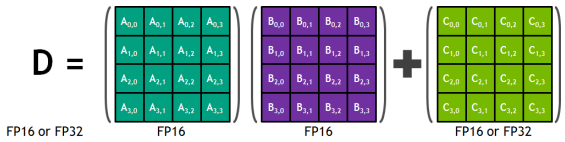


Figure 3.2. Tensor Core fused multiply add operation with mixed precision [60]

computation to occur, further improving the throughput. Finally Volta's SMs, has the addition of a number of Tensor Cores. These Tensor Cores are flexible, but still programmable, cores geared specifically towards deep learning Tensor operations. These cores are a collection of ALUs designed for performing 4x4 Matrix operations, specifically a fused multiply add (A*B+C) which multiplies two 4x4 FP16 matrices together, and adding that result to an FP16 or FP32 4x4 matrix to generate a final 4x4 FP32 matrix as shown in Figure 3.2. This divergence from the previous generation's architecture is due to the rapid growth of neural network computations. By providing this option, developers who capitalize on this functionality should be able to achieve large performance improvements over previous generation hardware.

Table 3.1, examines a number of different parameters for single GPU server cards. From this, we can see that the major factor for change between architectures is the number of cores available for computation, ranging from 240 for the original Tesla architecture, up to 5120 for Volta. We can also see that the amount of memory available and the memory bandwidth are increasing to allow larger problem sets to be solved on the GPU. We can see that although the clock speed of the cores fluctuates, it has not changed much. Finally, we can see that with Maxwell, the GPUs have stabilized at 250 watts for their maximum thermal design power.

TABLE 3.1

COMPARISON OF NVIDIA GPU ARCHITECTURAL DESIGN PARAMETERS

Architect	FP32 GFLOPS	CUDA Cores	Clock (MHz)	RAM (MB)	Bandwidth (GB/s)	TDP (Watts)
Tesla	622	240	1296	4096	102.4	187.8
Fermi	1030	448	1150	6144	144	225
Kepler	4666	745	875	12288	288	235
Maxwell	6335	948	1114	12288	288	250
Pascal	8706	1126	1303	16384	720	250
Volta	14028	5120	Unknown	16384	900	250

3.2 Rendering

Rendering was originally the primary driver of GPU technology, allowing developers to produce ever more realistic visualizations. To utilize a GPU for rendering, two main programming libraries exist for software developers to use, OpenGL and DirectX. OpenGL is an graphics library which allows rendering on many different operating systems such as Windows, Linux, OSX, Android, iOS. DirectX is Microsoft's graphics library which is restricted to developing for Windows and XBox. In this section, I will look at a number of aspects of implementing graphics rendering using OpenGL. Many of the techniques have large similarities between OpenGL and DirectX, due to the end result rendering on the same hardware.

3.2.1 Pipeline

Originally, OpenGL had a fixed functionality pipeline without access to programmable shaders. In this version, Objects were defined as a collection of primitives, made up of triangles. To aid in optimization, primitives such as quadrilaterals, line stripes and polygons were also definable. These larger primitives provided a way of defining and drawing an object by sending less data to the GPU. After OpenGL moved to the programmable pipeline with shaders being available, the ability to use these optimized primitives was removed leaving only triangular primitives available to the end user.

To represent a primitive, a set of vertices are defined. These vertices represent

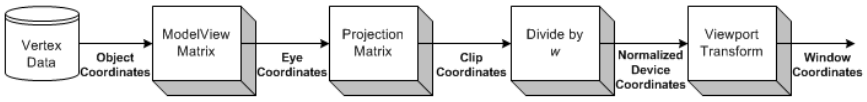


Figure 3.3. OpenGL transformation pipeline from vertex data to window coordinates [1]

a point in space, but can also have other attributes applied to them, such as color, normal vector and texture mapping coordinates.

In the fixed function pipeline, the order that the vertices in a triangle are sent to the GPU is important for both lighting and culling. This is because the normal vector of the triangle is calculated based on the three vertices if it is not provided. OpenGL uses a counter clockwise winding to represent the front of the object. By knowing this normal vector, it can be used within the lighting calculation itself as well as for culling if it is invisible from the camera's viewpoint.

3.2.2 Texturing

Texturing on the GPU is done by first uploading raw, or compressed image data if the graphics card supports it, then binding it when drawing an object. Vertices must have mapping data associated to them for the texture to be rendered onto the object. Normally, texture coordinates are floating point values in the range of $[0, 1]$ denoting how far along the texture it should access data from.

Textures stored on the GPU have a maximum possible size, 4096×4096 for newer GPUs.

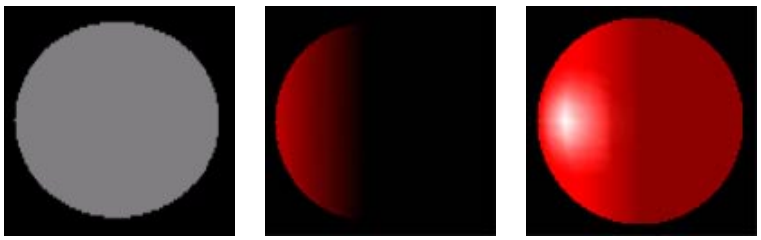
Due to having to switch to the correct texture and then draw its respective vertices, a common optimization is to combine multiple textures together into a texture "atlas". An example of this, taken from [61], is shown in Figure 3.4.

3.2.3 Lighting

In OpenGL 1.x, lighting was implemented as part of the fixed function pipeline. It provided the ability to add three different types of lighting to objects in the scene. These were as follows:

Ambient This light is the average volume of light that is created by emission of light from all of the light sources surrounding (or located inside of) the lit area.

Figure 3.4. An example of a texture atlas on the left with its final mapped result on the right



(a) Ambient (b) Diffuse (c) Specular

Figure 3.5. An example of lighting a sphere with different lighting types.

Diffuse This light represents a directional light cast by a light source and can be described as the light that has a position in space and comes from a single direction.

Specular This light is a directional type of light which comes from one particular direction and relies on the angle between the viewer and the light source. It reflects off the surface in a sharp and uniform way.

An example of different lighting styles applied to a colored sphere is shown in 3.5, taken from [81]

After OpenGL 1.x, implementation relied on developers writing shaders to handle lighting. This led to more accurate representation of light by allowing the calculations to be performed on a per-pixel basis rather than a per-vertex basis.

3.2.4 Shaders

Vertex This shader allows calculation to be done for each vertex in the scene. This is usually used to calculate per-vertex lighting or to setup values to be used later on in the rendering pipeline.

Fragment This shader allows for calculations to be executed on every pixel generated in the scene.

Tessellation This shader allows for the specification of how to tessellate primitives in the scene.

Geometry This shader takes an input of a single primitive and produces zero to N output primitives. It's main uses are to render the primitive to multiple different targets and to store calculated primitives to be used for computation.

Examples of GLSL code to implement simple diffusion lighting from [10] are shown below:

```
varying vec3 N;
varying vec3 v;

void main(void) {
    v = vec3(gl_ModelViewMatrix * gl_Vertex);
    N = normalize(gl_NormalMatrix * gl_Normal);

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

varying vec3 N;
varying vec3 v;
```

```
void main(void) {
    vec3 L = normalize(gl_LightSource[0].position.xyz - v);
    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L),
        0.0);
    gl_FragColor = clamp(Idiff, 0.0, 1.0);
}
```

In these source listings, I define the following:

ModelViewProjectionMatrix This is the product of the Model, View and Projection matrices which convert a vertex from object space to device space as shown in 3.3

ModelViewMatrix This is the product of the Model and View matrices which convert a vertex from object space to the camera's eye space as shown in 3.3

NormalMatrix This is a user specified matrix allowing for the global transformation of the provided normal vectors.

FrontLightProduct This is a derived product of the light's parameters combined with the vertex's material parameters to give the final diffuse color.

Normal This is the input vertex's normal vector.

Vertex This is the input object space vertex position.

Position This is the final device space position of the vertex.

FragColor This is the color of the current pixel.

All matrices within the OpenGL pipeline are 4×4, although this is counter-intuitive for 3D graphics, operations such as translation of a point would not be possible with 3×3 matrices. We can see this from the simple case of translating the point $(0, 0, 0)^{\mathrm{T}}$, multiplication of the zero vector by any matrix will yield the zero vector so it cannot be translated. In practice we extend a vector $(a, b, c)^{\mathrm{T}}$ to $(a, b, c, 1)^{\mathrm{T}}$ for use in the pipeline. It is interesting to note that the value of

the 4^{th} element will not always be 1 after multiplication and this variation is used in perspective calculations allowing for an accurate representation of how a three-dimensional object appears to the eye.

3.3 Programming

At the time of writing, there are two competing programming APIs to implement direct GPU calculation. NVIDIA provides the CUDA programming framework and the Khronos Group manage OpenCL, both are extensions of the C programming language. These two APIs provide the same basic functionality, and for features supported by both, it is simple to translate between the two. However, they have a number of distinct differences.

NVIDIA's CUDA provides good tool support to debug and profile applications and is usually more performant. It's main limitation is that it only executes on NVIDIA graphics cards.

OpenCL is designed to execute on multiple different hardware architectures including GPUs, CPUs or Accelerator Cards such as Intel's Xeon Phi. It's main benefit is that it is not locked into a specific vendor. However vendors, such as NVIDIA, may not provide drivers to support the latest standards.

Due to NVIDIA's dominance in the General Purpose Programming on GPU (GPGPU) domain, this was the target for my software implementations. This section will talk about how implementation of code on the GPU is achieved, as well as some common optimizations, using CUDA.

3.3.1 Kernels

Code that is run on the GPU is called a kernel and a simple example is show in Listing 3.3.1 which sums two arrays together and stores it in a third.

```
__global__ void add(int n, float *x, float *y) {
```

```
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```

There are two major differences compared to standard C code. Firstly, a prefix is added to the function, in this case *__global__*, which defines what code is able to execute this kernel. Three main prefixes are:

__global__ This allows any code to execute this kernel.

__device__ This only allows other kernels executing on the GPU to execute this kernel and can be combined with the *__host__* prefix to compile for both host and device.

__host__ This only allows the CPU to execute this kernel.

The second addition is the two undefined variables *threadIdx* and *blockDim*, which are available in device kernels to provide information on the current thread and how large the block of execution is.

3.3.2 Execution

To execute a kernel, another extension to the C standard had been added.

Listing 3.3.2, shows execution of a kernel using the <<< >>> syntax.

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

This syntax requires specifying two parameters, a number of blocks and then how many threads per block. Threads per block defines how many threads are executed at once, usually in the range of 64 to 512. Number of blocks defines the number of

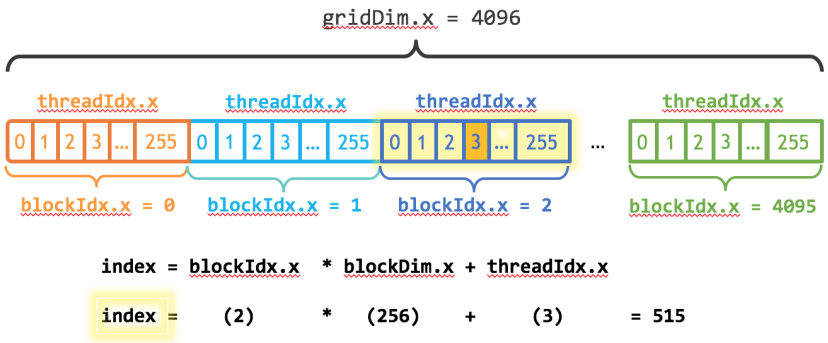


Figure 3.6. A depiction of how grids, blocks and threads are related in CUDA [58]

blocks of work to schedule, usually defined by the total number of items to work on divided by the number of threads per block.

Figure 3.6, shows how the grid size (total number of items), is divided into blocks and threads.

Blocks are important because they define a collection of threads that are able to access a small chunk of memory (usually 64KB) that can be shared between them.

3.3.3 Libraries

Along with the other tools provided by NVIDIA, a number of GPU optimized libraries have also been developed. These libraries aid developers with implementing optimal solutions without the need to develop and tune in-house versions.

cuDNN NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks, it is designed to be integrated into higher-level machine learning frameworks.

cuFFT NVIDIA CUDA Fast Fourier Transform Library (cuFFT) provides a simple interface for computing FFTs up to 10x faster, without having to develop your own custom GPU FFT implementation.

TensorRT NVIDIA TensorRT is a high performance neural network inference library for deep learning applications.

cuSolver A collection of dense and sparse direct solvers which deliver significant acceleration for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications.

cuSparse NVIDIA CUDA Sparse (cuSPARSE) Matrix library provides a collection of basic linear algebra subroutines used for sparse matrices that delivers over 8x performance boost.

cuBLAS NVIDIA CUDA BLAS Library (cuBLAS) is a GPU-accelerated version of the complete standard BLAS library that delivers 6x to 17x faster performance than the latest MKL BLAS.

cuRAND The CUDA Random Number Generation library performs high quality GPU-accelerated random number generation (RNG) over 8x faster than typical CPU only code.

3.3.4 Optimization

To optimize algorithms implemented on the GPU NVIDIA provides a number of profiling tools to ease the development of GPU applications. Firstly, it provides a command line tool "nvprof" which allows for the collection of performance statistics and simple exploration of the results. They also provide the "Visual Profiler" which allows for a much deeper exploration of important features by collecting statistics with "nvprof" and then providing an Eclipse based tool for exploration.

One of the largest performance issues developers are faced with is caused due to the GPU being poor at branching code. This is an issue because most software branches often. Because a GPU executes in a Single Instruction Multiple Thread pattern, whenever a branch occurs it must execute all sides, one at a time, until the calculation is finished. This is demonstrated in Figure 3.7.

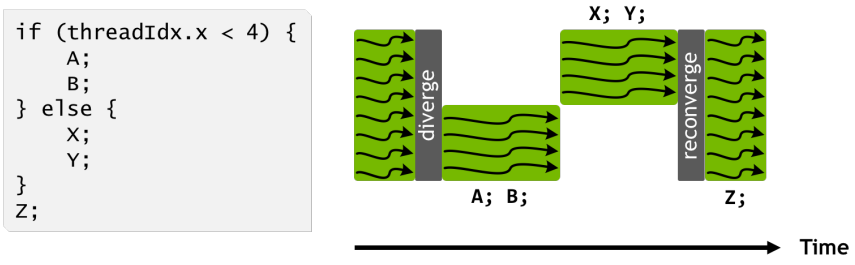


Figure 3.7. An example of divergence within a warp [60]

Pointer aliasing is another area where optimization is possible. Aliasing is where two pointers have the same value and is mainly an issue in C derivatives due to the ability for pointers to point to memory locations. This stops the compiler from being able to reorder instructions optimally. To combat this, there is a compiler hint, *__restrict__*, which allows developers to specify that two pointer arguments to a function do not alias each other. For example, we can modify Listing 3.3.1, to restrict the pointer parameters, as shown in Listing 3.3.4.

```
__global__ void add(int n, float * __restrict__ x, float * __restrict__
    y) {
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

To optimize the memory access of threads executing on the GPU, the device coalesces global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. Accessing memory offset by a stride for each element, such as indexing into a multidimensional array, forces the GPU to be unable to coalesce these memory accesses because they are so far apart, degrading the memory access performance massively. To overcome this problem,

shared memory can be utilized. Shared memory is an on-chip cache, usually around 64KB, which is shared between all of the threads in a warp. The shared memory's latency is roughly 100x lower than uncached global memory latency. A common paradigm is to have the first thread load the data from global memory into the shared memory, synchronize all of the threads in the block and then have all the threads execute together.

CHAPTER 4

MODELING DIGITAL HERITAGE

4.1 Introduction

New technologies have made it possible to convert the archaeological and architectural records of the past into digital formats and then record that information in databases, Computer Aided Design (CAD) maps and digital images. Data created using state-of-the-art 3D laser scanners and high-resolution panoramic digital images offer scholars the unprecedented opportunity to recreate accurate, searchable digital copies of the sites they are researching. Digital records of historic sites and structures with an accuracy of ±4mm potentially give researchers, educators, students and the general public an opportunity to virtually explore all the details of a site.

Researchers working at archaeological and architectural sites typically produce a diverse array of 3D data allowing them to make more accurate claims about historical sites. Here we present the results of our research combining 3D scanner and high resolution digital camera images to create unique data virtualizations, allowing users to frame new research questions and generate innovative interactive explorations of historic properties and world heritage sites. Using data already collected at the Roman Forum, we developed software capable of fusing extremely high-resolution (giga-pixel) panoramic images produced by the GigaPan system, with dense point clouds generated by Leica 3D scanner to create accurate, interactive 3D images of archaeological sites. Because the technique affords unique ways to manipulate and interpret 3D virtualizations of historic sites, its potential contribution to the field of humanistic scholarship and education is vast.

In addition to offering unique ways to study historic sites, this Chapter presents methods to spur new scholarship, enabling new methods of analysis, site manipulation and reverse blueprinting (a way of retroactively generating plans should the site ever be destroyed). Learning materials generated will allow better study of the conservation arts, what it means to think in 3D, and connections across pencil, paint and pixel modeling. Our approach should add an invaluable resource for educators/scholars seeking to enrich their curricula or research agenda with the study of historic sites.

3D visualizations of historic, world heritage and cultural sites will not only expand and enhance our collective understanding of the historical record; it will bring the past to life for a population who is increasingly visually oriented. This precise type of 3D modeling allows humanities students and researchers to pose questions surrounding what it means to practice conservation and preservation of crucial sites in the age of digital virtualization.

In Section 4.2 we introduce the DHARMA group and the history behind their work at the Roman Forum. In Section 4.3 we describe the details of scanning the Roman Forum and the workflow employed. In Section 4.4 we discuss the GigaPan photographic work done at the Roman Forum and in Section 4.5 we discuss the methods we have developed to combine these data sets. Finally in Section 4.6 we describe the workflow for data combination, followed by conclusions in Section 4.7.

4.2 DHARMA - Roman Forum Project

Digital Historical Architectural Research and Material Analysis (DHARMA) is a research team founded in 2007, based at the University of Notre Dame School of Architecture. The team, under the direction of Prof. Krupali Krusche, works on documenting historic monuments and World Heritage Sites around the world with the use of Leica 3D laser scanners. These high-speed, long-range scanners are ideal

for projects that are difficult to document by traditional methods. The scanner provides researchers with the most field-efficient means of data collection. Recently the team has also used 3D scanning to assess aging effects on historic buildings and reconstruction processes of buildings with historical value.

In the summer of 2010, the DHARMA team, led by Prof. Krusche in cooperation with Archaeologist James Packer, and with special permissions from Soprintendenza Speciale per i Beni Archeologici di Roma, Ministry of Heritage and Culture and the Archaeological Service, digitally and traditionally documented the Roman Forum site, Rome, Italy in a number of different formats.

4.3 3D Scanning

In recent years, laser scanning or Light Induced Detection and Ranging (LIDAR), has become a major technique that has started replacing the use of hand measuring, surveying, and the use of Total Station in the field. A laser scanner is an automated surveying apparatus that uses a laser beam to collect location coordinates from the surface of a desired object. These measured coordinates are recorded in the form of a point cloud that identifies the shape of the object by converting the spatial geometry established through the x, y, and z position of that point in space. Depending on the way the data is collected, the scanners have different capacities and are of various types. This chapter concentrates on the "time of flight" type of scanner, which is specifically used in the historic and archaeological documentation of large sites. The data produced from the scanner can be used to create measured-drawing documentation of historic monuments and existing buildings, to save costs on as-built drawings used in restoration and reconstruction.

The Roman Forum data was collected using hand measuring, photogrammetry, 3D Scanner and GigaPan technologies. A team of two graduate research assistants scanned the Roman Forum site, using the Leica Scan station model, with a resolution

of 1cm x 1cm throughout the site (Figure 1 (a)). The team worked on the central Forum i.e., the open space (the "Area Fori" including the east and west rostra, the small monuments around the latter, and the Diocletianic columns) and the following buildings: the Temple of Caesar, the Temple of Antoninus and Faustina, the Basilica Aemilia, the Curia, the Arch of Severus, the Temple of Concord, the Temple of Vespasian, the facade of the Tabularium, the Temple of Saturn, and the Basilica Julia.

All spaces were assessed, analyzed, documented and scanned from outside, as well as inside the ruins. It took the team seven, twelve-hour workdays to complete the project with one scanner and two battery units being charged interchangeably. The site is approximately 150m x 250m in dimension with very large variations in contour, and about fourteen monuments and change of grade existing throughout the area. It took the team 27 scans, with more than 100 million data points of cloud data, and five strategic target locations to completely document the site. A team of four DHARMA undergraduate members documented the whole site with hand measuring techniques. They completed measuring the overall site in the same stipulated time with sketch drawings containing dimensions of individual buildings and their location from a set zero point.

While the information collected by hand measuring was adequate to give an overall understanding of individual monument dimensions, a lot of time was spent maneuvering through the complex site, and the detail of information was not exact or refined because of the ruinous state of the site. It is important to note however, that there were very important observations recorded during the hand surveying of the site that would have been missed while scanning the site from remote locations (Figure 2 (a) and (b)). The scanner did exceptionally well in capturing data from points that were not visible due to their remote location or issues of accessibility. Even in places where the scanner couldn't be positioned because of time, target and location constraints,

Figure 4.1. (a) Luke Golesh operating the Leica ScanStation at the Roman Forum. This image is taken from top of the ruins of Vespasian temple. (b) The Roman Forum as captured by the Leica ScanStation.

we were still able to document around 95% of the Forum site by being strategic about scanner positions throughout the field work.

Off site, the registered and unified data from the 27 scans revealed information that the scanner had collected filling "holes" in the scan results for inaccessible positions. The plan and sectional views of the combined 3D scan data revealed information regarding the spatial correlation of the individual ruins as never seen before. It also revealed the comparative level change of the archaeological site in relation to the present city of Rome (Figure 1 (b)).

4.4 Photographic Data Acquisition

The DHARMA team also photographed the site using GigaPan technology, originally developed by robotics scientists from Carnegie Mellon University in cooperation with NASA Ames Research Center for use by the Mars Rover program.

The GigaPan system uses a programmable robotic mount to precisely control a Digital Single Lens Reflex (DSLR) camera to take hundreds or thousands of pictures of a scene automatically. The practical resolution limit of the panorama imagery created using GigaPan technology is directly related to the zoom level of the lens and

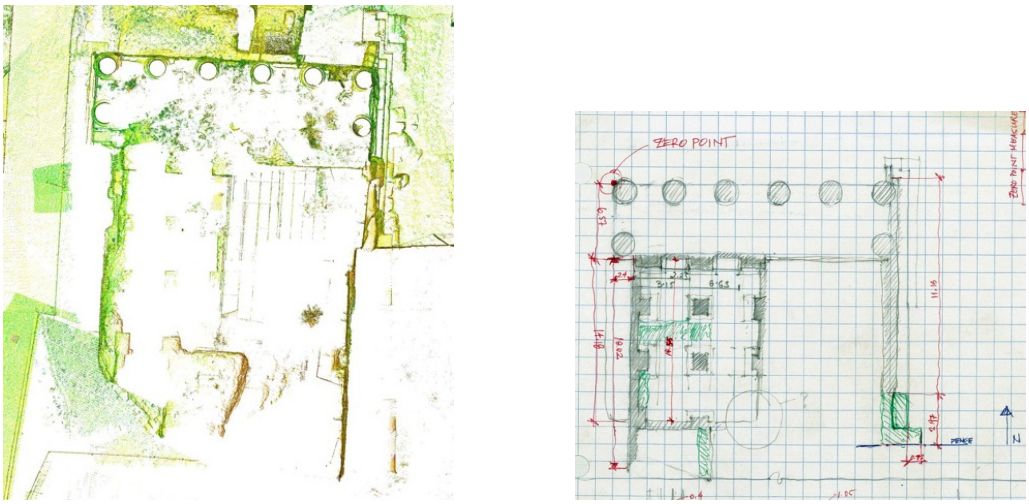


Figure 4.2. Results of the measurements taken at Temple of Saturn, at the Roman Forum, with the Leica ScanStation (a) and hand measure drawings (b).

quality of the camera's image sensor.

Although there exists software to produce a combined panoramic image for each acquisition point, such as GigaPan's Stitch or Kolor's Autopano Pro software, this was made redundant when mapping the images to the Laser Scanner pointcloud as described in Section 4.5. To illustrate to capabilities of Robotic Tripods, Figure 4.3 shows an example where 150 separate high resolution images, taken using a DSLR mounted onto the GigaPan robotic tripod, were stitched, using Kolor's Autopano software, into a single panorama image. The inset image represents a single image within the set. Zooming into small sections of the arch in the GigaPan image reveals fine details not apparent to the naked eye when viewing from ground level.

4.5 Combining Leica Scan Data and Panorama Images

We explored methods for combining the two primary technologies into a single interactive system for cataloging, analyzing and displaying both point and raster data

Figure 4.3. Under-arch of Septimus Severus at the Roman Forum. The inset illustrates the high resolution available from the GigaPan image.

at both standard and extremely high resolution. Previous methods have manually adjusted the registration between the data sets visually, which does not guarantee the fidelity of registration over the full extents of the data sets and, in general, cannot correctly align the data sets. In addition, for many images an automatic process is desirable. Computer programs also exist to approximately align the data to produce single point visualizations, in general this visualization cannot be rotated through all viewpoints and it is not possible to measure surface features based on the underlying 3D data.

In this work, 3D data is combined with 2D surface data such that:

1) The registration between the data sets, and hence the validity of observations and measurements based on them, is determined geometrically and maintained over the surface of interest.

2) Where multiple sets of 2D data are available, the optimal set can be determined for each rendered pixel set. In addition, visual artifacts generally present with

multiple images can be reduced.

3) The surface images do not have to be produced from fixed viewpoints in relation to the 3D representation (or map directly to individual 3D features of the object), allowing the acquisition of data from the architectural to nano scales, such as sub-surface information. This is particularly important for complex surfaces.

To accomplish this, we have a main source of point data (a Leica scanner), and two potential sources of image data (a Leica scanner or a DSLR attached to a robotic panorama head). To work with the Leica scanner's picture data, we had to understand the camera's motion during scanning. We also had to export data from Cyclone (Leica's scanner software) which describes some extra information about the camera's specifications. Finally, we determined that the camera data is arranged in a gimbal format i.e. z,y,x rotation order.

To extend this to using panorama picture mapping, it becomes more complicated. This is because, unlike the Leica scanner, the robotic panorama head will not be able to take images from the exact same location as the scanner. Also, due to the nature of robotic panorama head, end users attach a DSLR to the device and each of these will affect the field of view and stepping, which are important parameters to mapping the pictures onto the data.

The following content appears in a provisional patent application [87] by the authors.

4.5.1 Multiple Scans

To fully document a single monument requires several 3D scans and GigaPan's. Since the model requires all of the scans in the same reference frame, a transformation matrix must be determined for each scan. To determine this matrix requires at least

3 points, although more are desirable to reduce the effect of a 'rogue' point, these points generally take the form of targets that can be acquired by the scanner. A good workflow that determines target placement is required to get the highest fidelity data. Once this data is available it can be inserted into the model XML file and the DHARMA Clone analysis program can use this to determine each matrix relative to a scan that is arbitrarily picked as the main reference scan.

Given three or more common targets in two scans, denoted here u_i and v_i where v_i are the targets in the main reference scan, DHARMA Clone determines the transformation by first calculating the centroids for both u_i and v_i, denoted c_u and c_v. The centroids are subtracted from u_i and v_i to give $\bar{u}_i$ and $\bar{v}_i$ respectively and the sum of the outer-products taken to yield matrix $M = \sum_i \bar{u}\bar{v}^{\mathrm{T}}$. The next step is to perform a Single Value Decomposition [93] on the matrix M to get $[U, S, V] = SVD(M)$. The rotation matrix is then found as $R = VU^{\mathrm{T}}$ (S is a scaling matrix which is not required for this analysis as scaling is unity). The translation is then $T = -R \times c_u + c_v$, and this can be combined with R to produce a single 4×4 transformation matrix.

4.5.2 Surface Generation

To map the point cloud data and image data together we need to first create a surface from the point cloud data. Analysis of the point cloud data from Cyclone, in the PTX file format (text file containing point cloud coordinates), showed that the data is arranged as points in a vertical line with any cut out points being replaced with zero vectors. From this, we created an algorithm that looks at four points adjacent to each other and tests if there are any missing points. We then create triangles from all the points that are available. This algorithm is described in Listing A.1.1.

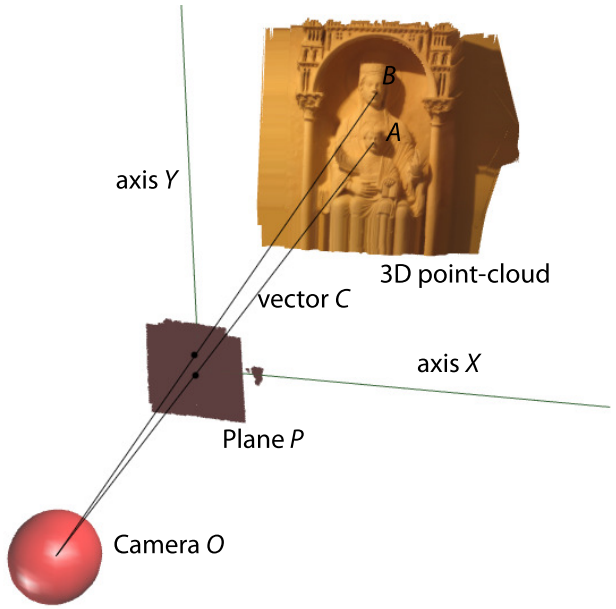


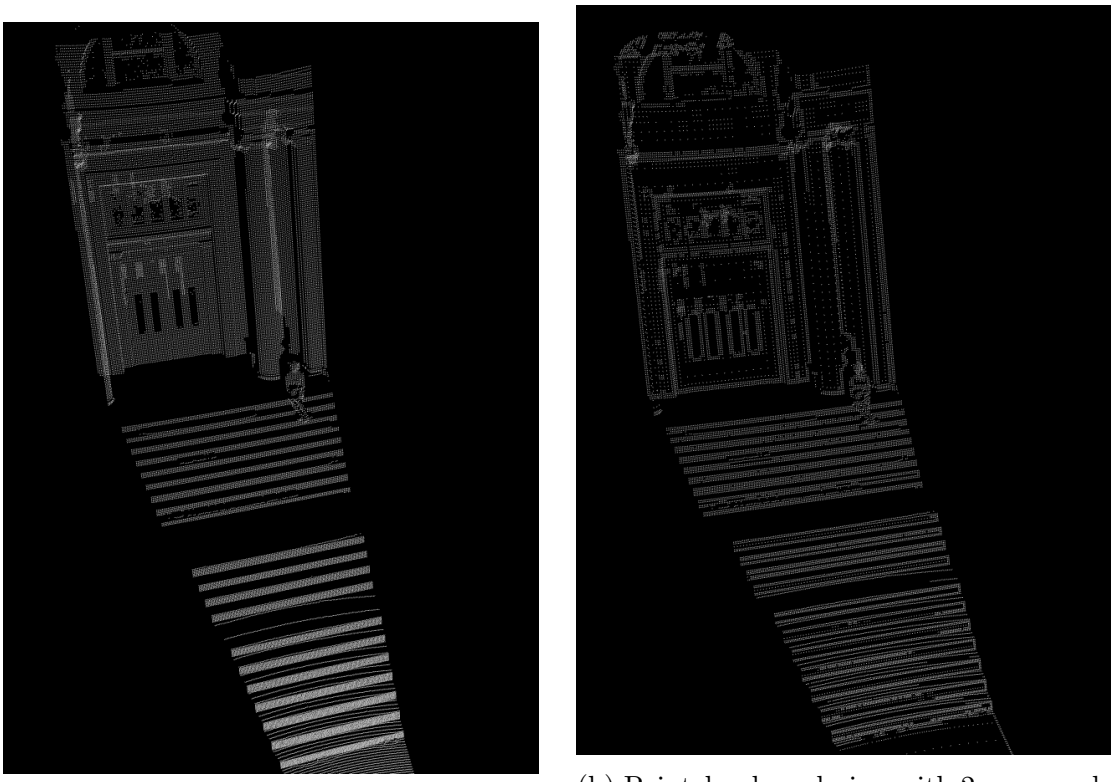
Figure 4.4. Mapping 2D image to 3D point-cloud.

4.5.3 Point Reduction in $\mathcal{O}(N)$

To improve the speed of rendering our mapped models, we came up with an algorithm that removes redundant points from the data set whilst keeping important features, such as edges. This can be seen in Figure 4.5 and the algorithm has a time complexity of $\mathcal{O}(N)$.

Due to the structure of our data set (a grid of points where holes are symbolized by a predefined "empty" point) we can test four squares of increasing sizes to see if all of the points within the four squares face the same direction. If the points do face the same way within a user defined threshold angle, we can reduce the four squares into a single square.

Two comparison metrics are available for the squares reduction. The first comparison metric compares the angle between the center point and all eight surrounding points. If any of the surrounding points are greater than the threshold angle, then we know that the current set can not be reduced and we move on. This method provides



(a) Pointcloud rendering without reduction

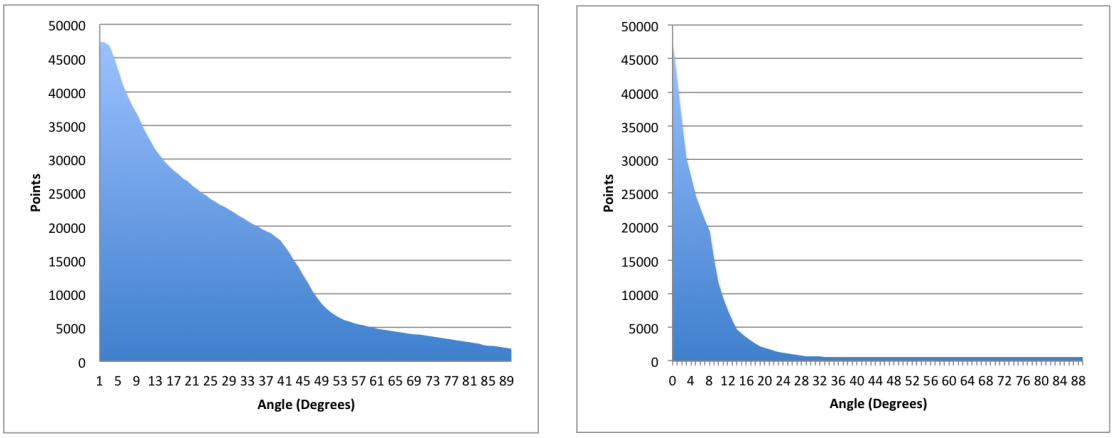
(b) Pointcloud rendering with 2-norm reduction

Figure 4.5. Example of compression by removal of points while retaining important structural features such as edges.

the most accurate final representation as it does not remove any sets of points that do not all pass the test. This method for a test system provides reduction in triangles as shown in Figure 4.6(a).

The second comparison metric compares the square root of the sum of the squares of all of the angles between the center point and all eight surrounding points. If this value is less than the threshold then we know that we can reduce this set. This provides a less accurate representation due to the way that the two norm calculation works, however it allows for a much greater reduction of points as shown in Figure 4.6(b).

The algorithm, shown in Listing A.1.2, has a time-complexity of $\mathcal{O}(n)$ because



(a) Center comparison method

(b) Two norm comparison method

Figure 4.6. Point reduction behaviors for different reduction methods.

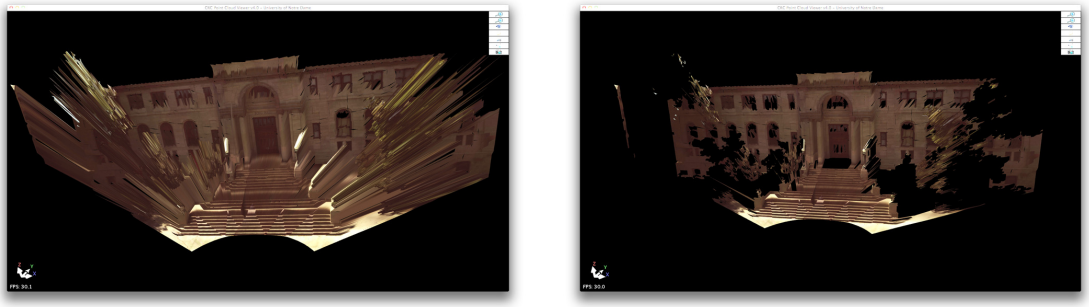
every iteration works on a progressively smaller fraction of points, $\frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{n}$, which is a geometric series that always increases but never actually reaches n.

4.5.4 Mesh Pruning to Remove "Poor" Triangles and Overlap

When multiple scan locations are combined together and meshed, there are regions of the final model that have overlap from multiple scans. If we mapped these, we would see that the resolution of the scan (and hence the size of the triangles calculated) is poor where the scanner is not directly aligned with the monument or object. We would also notice artifacts in the form of very large (stretched) triangles where the angle that the scanner scanned from was not direct towards the model or where adjacent scan points have a large difference in distance from the scanner. We classify these triangles as poor because to attain the highest resolution model, we want as many small triangles as possible.

To overcome this issue, we calculate the average *compensated* size of a triangle and then prune all triangles that are greater than a threshold. This threshold is based off of the average *compensated* size and can be tuned to an individual model if required

to achieve the greatest quality end result. The compensated area is calculated as the area divided by the distance from the scanner. In some situations it may be desirable to use the reduction algorithm without using compensation. Given points A, B, C representing the vertices of the triangle we find the area by forming vectors AB and AC, then the magnitude of the cross product $AB \times AC$ is twice the area of the triangle. To compensate for distance we divide by the magnitude of A. This is illustrated in Figure 4.7.



(a) No pruning

(b) Pruning by area

Figure 4.7. Example of artifacts generated by near and far objects appearing adjacent to each other in the point cloud. In the right hand image the triangle areas are calculated and normalized and removed if they are larger then a given threshold.

Two algorithms were created to solve this problem. The first, shown in Listing A.1.3, calculates the average triangle size and the second, shown in Listing A.1.3, prunes triangles greater than a specified threshold.

4.5.5 Mapping Algorithm

Once we have the surface calculated, we can then progress to map the image data onto the surface, based on Figure 4.4.

We now discuss mapping 2D images onto the surface.

1) **Determine two coincident points** A and B in the image and 3D data, we will assume point A is close to the center of the image. In Figure 4.4 the 'noses' of the two statues are used.

2) **Find the viewpoint.** Find the vector from the camera origin O to point A, denoted vector C. Without loss of generality (w.l.o.g.) we assume here that point O is at the system origin

$$O = (0, 0, 0). \tag{4.1}$$

3) **Find the viewing plane.** Construct a plane P with normal vector C, w.l.o.g. we construct the plane at a distance of 1 from the camera origin O. Hence, using Eqn. (4.1), a point on the plane is $\hat{A} = A/||A||$ and the vector from the origin O to point $\hat{A}$, $\hat{A} - O$, is a unit normal vector to the plane. Then for point p on plane P, using Eqn. (4.1),

$$p.(\hat{A} - O) - \hat{A}.(\hat{A} - O) = p.\hat{A} - 1 = 0. \tag{4.2}$$

4) **Find the points in the viewing plane.** For each point P_i on the 3D surface, draw a line L_i from the camera origin O to the point P_i and find the point of intersection with plane P, PP_i [53]. For point l on line L_i and parameter t we have

$$l = O + t(P_i - O), \tag{4.3}$$

which intersects the plane when

$$t = \frac{-1 - O.(\hat{A} - O)}{(P_i - O).(\hat{A} - O)}. \tag{4.4}$$

Using Eqns. (4.1 - 4.4) this reduces to

$$PP_i = \frac{P_i}{P_i.\hat{A}}. \tag{4.5}$$

5) **Find Y axis in the viewing plane.** Define a 'vertical' plane VP containing vector C, find the line Y where this plane intersects plane P. This will be the 2D 'y' axis. This can be accomplished by adding a vertical offset $VOFFS$ to point A to give point $V = A + VOFFS$, a new point on the plane $\bar{V}$ can be calculated using the technique in Item 4) Eqn. (4.5)

$$\bar{V} = \frac{V}{V.\hat{A}}. \tag{4.6}$$

We require a unit vector $\hat{Y}$ in the 'y' direction from the plane origin $\hat{A}$ hence

$$\hat{Y} = \frac{\bar{V} - \hat{A}}{||\bar{V} - \hat{A}||}. \tag{4.7}$$

6) **Find X axis in the viewing plane.** Rotate the line Y through 90 degrees clockwise in plane P around the point where vector C intersects plane P. This will be the 'x' axis which we denote line X. Given

$$O = (a, b, c),\ \bar{V} = (x, y, z), \tag{4.8}$$

then we find point H rotated around $\hat{A}$ in plane P [31] (for this method $u =$

$a,\ v = b,\ w = c$)

$$\begin{aligned}
\text{dotp} &= u*x + v*y + w*z, \\
H &= (\ a*(v*v + w*w) + u*(-b*v - c*w + \text{dotp}) \\
&\qquad +(-c*v + b*w - w*y + v*z), \\
&\quad b*(u*u + w*w) + v*(-a*u - c*w + \text{dotp}) \\
&\qquad +(c*u - a*w + w*x - u*z), \\
&\quad c*(u*u + v*v) + w*(-a*u - b*v + \text{dotp}) \\
&\qquad +(-b*u + a*v - v*x + u*y)\ ).
\end{aligned}$$

We require a unit vector $\hat{X}$ in the 'x' direction from the origin $\hat{A}$ hence

$$\hat{X} = \frac{H - \hat{A}}{||H - \hat{A}||}. \tag{4.9}$$

7) **Project planar points onto 'x-y' axes.** Project each of the points PP_i onto the x and y axes to determine their 'x-y' coordinates PPX_i and PPY_i

$$PPX_i = (PP_i - \hat{A}).\hat{X},\ PPY_i = (PP_i - \hat{A}).\hat{Y}. \tag{4.10}$$

8) **Find scale factors and offsets.** Calculate a scale factor S and 'x-y' offsets $XOFF$ and $YOFF$ for the 2D picture. Given that points A and B on the 3D point-cloud correspond to points $\hat{A}$ and $\hat{B} = B/||B||$ in plane P, and assuming these points on the 2D image have 'x-y' coordinates (a_x, a_y) and (b_x, b_y)

respectively, we have

$$S = ||B - A||/\sqrt{(b_x - a_x)(b_x - a_x) + (b_y - a_y)(b_y - a_y)}, \quad (4.11)$$

$$XOFF = -a_x; \quad (4.12)$$

$$YOFF = -a_y. \quad (4.13)$$

9) **Calculate actual 'x-y' coordinates** ($\mathbf{ppx\_i}, \mathbf{ppy\_i}$) for all points

$$\mathbf{ppx\_i} = PPX_i * S - XOFF, \ \mathbf{ppy\_i} = PPY_i * S - YOFF. \quad (4.14)$$

Once we have these actual 'x-y' coordinates, we then have all of the information that we require to render a textured surface to the screen.

4.5.5.1 Examples

A Leica [46] scan of a statue at Bond Hall at the University of Notre Dame has been textured with a photograph of the same statue, taken from the same point as the scanner. The surface generated from the point-cloud data and the resulting textured surface can be found in Figure 4.8.

A Leica [46] scan of the Arch of Septimius Severus at the Roman Forum has been textured with multiple scanner photographs from a number of scanner locations. The partially textured surface can be seen in Figure 4.9.

4.5.6 DHARMA Interceptor

Initial use of the GigaPan robotic tripod at the Roman Forum showed some interesting problems. Where edges of the monuments were photographed the camera often failed to auto-focus if the center of focus was in the area of clear sky. Apart from the missing image, this exacerbated other issues. For instance the spherical

Figure 4.8. Example of the stages of mapping the statue of bond hall.

coordinates of each photograph could not be determined.

To overcome this problem I built an Arduino interface module that negotiates between the GigaPan and DSLR camera and adds additional data EXIF to the image, denoted the DHARMA Interceptor (DI) (see Figure 4.10). This relays the GigaPan command to take a picture to the camera and pauses the GigaPan until information on the success or failure to take the picture is received. If the picture fails then a new point of focus is chosen and the process is repeated until a picture is successfully taken or all focus points are tried. This dramatically reduced the number of missing images. The DI unit also has a three axis accelerometer and magnetometer to determine the orientation and tilt of the camera, this data is inserted into the image as EXIF data which can be used by the DHARMA Clone analysis program to automatically sort the images.

4.6 Method of Data Processing

In this section, we will describe the workflow required to transform a Leica scan of an object along with corresponding pictures, all the way through to the creation of the final viewable model. We assume that the reader has already registered their

Figure 4.9. Partial rendering of the Arch of Septimius Severus at the Roman Forum.

data within Leica's Cyclone software [47].

The first stage is to extract the required data from Cyclone so that we can use it in our software. To do this, the registered point cloud data needs to be exported in the PTX format. This is due to the fact that other formats do not keep a rigid structure that allows the point data to be converted into a surface for applying the texturing. The next stage is to export every picture and the corresponding picture data from each "Scan World", where each "Scan World's" files are stored in separate folders.

Once we have extracted the data from Cyclone, we must first guarantee that the

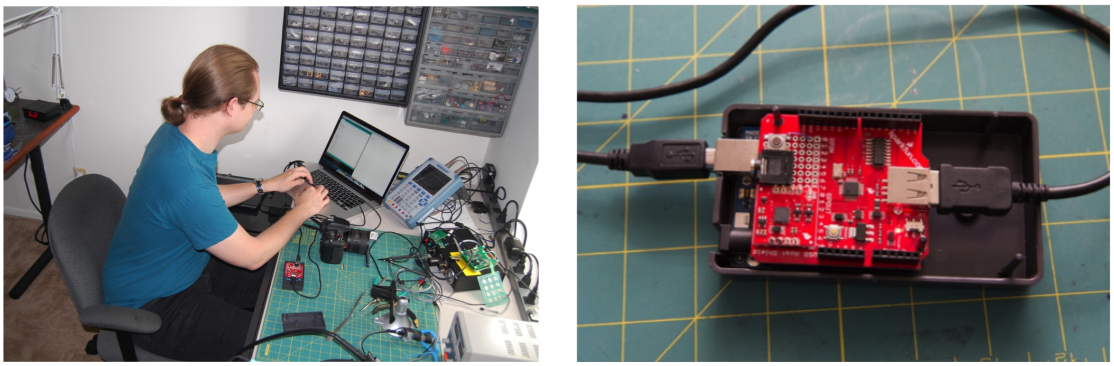


Figure 4.10. DHARMA Interceptor based on Arduino.

data that has been exported is correct. We run a utility on the PTX file to make sure that each model has the number of points specified at the start of the model description (the number of horizontal and vertical points that it should contain). This has become necessary since we have discovered that some large registrations have a small number of models that do not have the correct number of points compared to what we expect.

Once we have verified the data is correct, we then move on to creating a structure that our software can understand. This allows for the automated mapping of the pictures and surface generation. The first step is to convert the PTX file to a binary format. This reduces the data size of the file and allows for faster processing within the application. We then need to execute a utility in each of the "Scan World's" picture directories to extract the required information from the photo information taken from Cyclone. This creates a corresponding info file for each image. Finally, we need to create a map file for each "Scan World" which contains the starting and end number for the pictures, which "Scan World" it is and which image is the "master" image.

Next, we can use our software to do the final processing. Firstly, the binary file is loaded into the software, which allows for the creation of the surface of the object.

Next, we load each of the individual map files, which calculate which parts of the surface relate to which picture. Finally, the software writes out a valid DHZ file that can be loaded with the viewing application for visualization.

4.7 Conclusion

In this work, we explored methods of creating realistic accurate 3D models of historical monuments from data collected using LIDAR scanners and traditional photography. This has many areas of interest, such as scientific research, interactive education for high school students, who may not necessarily be able to visit sites such as the Roman Forum, and for students of architecture, history and archaeology that study such sites in greater detail.

Although it has been possible to acquire the data, consisting of multiple 3D point-clouds and arrays of images from different viewpoints, using the data in a Heritage Preservation and Architectural Education setting has been challenging. Software was provided with the LIDAR scanner to stitch together point-clouds from different vantage points but it was not possible to remove overlapping data that is inherent in the process. In addition there was no way to merge the images and point-clouds to form a complete navigable model at the resolutions of interest. Further issues presented themselves with distortions in the resulting panoramic vista produced by the GigaPan software, partly caused by the GigaPan requirement of a fixed focal distance which is problematic in real Heritage site viewpoints. We find that the current options, such as manual draping or stereo through motion, do not solve this problem effectively and propose a set of algorithms, tools and workflow for collecting data and automatically processing it to produce these 3D models.

Working with Dr. Krusche's team of architects and archaeologists, we develop an on-site workflow and a software platform to ingest and present data. We wrote software to merge multiple point-clouds into the same viewpoint and to cull the re-

dundant overlapping data. We implemented algorithms to generate a surface for the model, including compression techniques to remove redundant surface features. Finally, we implemented algorithms to scale, rotate and position images on the generated surface as well as choosing the optimal image pixels where images overlap. The resulting work has also been published as a Patent application [87] and was displayed in the Curia Julia at the Roman Forum. Using seed funding from the University of Notre Dame, we were able to acquire both a LIDAR scanner and a GigaPan robotic tripod, and document sites such as the Roman Forum and Taj Mahal in unprecedented detail.

CHAPTER 5

LONG TIMESTEP MOLECULAR DYNAMICS ON THE GRAPHICS PROCESSING UNIT

5.1 Introduction

Molecular dynamics (MD) involves solving Newton's equations of motion for a system of atoms and propagating the system over a small time step. MD finds uses in studies of protein folding, virtual drug screening, design of polymers, and sampling of molecular configurations.

Traditional MD simulations are limited in length by timestep limits. Studies by our group and others have shown that traditional MD is limited to timesteps of about 2 fs due to high-frequency resonance [39, 51, 74]. Even the most basic biologically relevant motions occur on the microsecond to millisecond range, which is 9 to 12 orders of magnitude greater than the timesteps possible with traditional MD. Further, each step requires a costly force calculation ($O(N)$ to $O(N^2)$). As such, simulating medium-size proteins often requires months of computer time on a large distributed system such as Folding@home [8, 80] to simulate milliseconds of dynamics. Similarly, simulating a large protein (e.g. the β-2 Adrenergic Receptor) on the more interesting biologically-relevant time scales (milliseconds through hours) using a standard desktop computer would take years. Thus, it is not feasible to simulate timescales of biological interest without substantial advances in MD methods.

Approaches for reducing the computational cost of MD have generally followed two tracks: improved algorithms and hardware acceleration. Langevin dynamics,

which solves a stochastic differential equation under dissipation-fluctuation constraints and provides an attractive thermostat, can be used to overcome instabilities due to resonances of constant energy integrators [74] and achieve larger timesteps [40]. However, even Langevin dynamics integrators require relatively small time steps for stability due to fast frequency motions within the bonded forces. As a result, algorithms such as Normal Mode Langevin (NML, [41, 88]) which can achieve timesteps 25× - 50× larger have been developed. NML, referred to here as Long Timestep Molecular Dynamics (LTMD), runs Langevin dynamics but splits the motions into fast and slow frequency, then over-damps the fast frequency motions by applying Brownian dynamics [17]. Thus acceleration is only assumed to occur among the slow frequency motions which allows for the increase in time step. This method has been shown to achieve adequate sampling over long timescales of microseconds to milliseconds and to scale well with system size. For instance, it yields 11-fold speedups over conventional Langevin dynamics for 882-atom Bovine Pancreatic Trypsin Inhibitor (BPTI) and WW domain folding simulations.

Hardware acceleration allows each timestep to be computed more quickly. D. E. Shaw's group has developed Anton, a specialized supercomputer where MD algorithms are implemented in hardware using application-specific instruction chips (ASICs) [76–79]. For explicitly solvated systems, it has been shown that Anton can provide speed ups of up to 2 orders of magnitude over simulations run in HPC environments. However, whilst Anton's design is flexible, it requires expert programmers to implement new algorithms or functionality.

It has been shown that the computation of MD simulations can be sped up considerably by taking advantage of GPUs [84]. NAMD [67, 83] adapted GPU support for running on large clusters. GPUs are used accelerate the computation of electrostatics and Generalized Born [62] implicit solvent model while the remaining computations and communications are handled by CPUs. Overlapping GPU non-bonded force

calculation (parallelized in a similar way using blocks) with CPU communication protocols yielded a five to seven fold improvement in efficiency on NAMD [66] when running simulations of the Apolipoprotein A1 (ApoA1, 92000 atoms) and Satellite Tobacco Mosaic Virus (STMV, 1.06 million atoms). This allowed long-range electrostatic algorithms such as Particle-Mesh Ewald (PME, [16]) to proceed on the CPU while bonded and short range non-bonded forces took advantage of the GPU power. We note that D. E. Shaw's group's replacement for Anton will be GPU based.

Other MD packages have focused on running entire MD simulations on one or more GPUs on a single workstation [33, 36]. In doing so, GPU-enabled workstations are capable of running simulations on the same timescales as large clusters at a fraction of the cost, which significantly increases access and availability for the average researcher. Friedrichs, et al. have shown that OpenMM [19, 26], a library for performing MD on GPUs, is capable of speeding up simulations of implicitly-solvated systems more than 500 times over an 8-core CPU. Any MD software package that links against OpenMM can take advantage of the speed ups offered by GPUs. OpenMM implements all MD algorithms needed to run constant energy and constant temperature simulations, implicit and explicit solvent, and different AMBER force fields. OpenMM has been benchmarked at 127 ns/day for implicit solvent simulations of DHFR with roughly 2,500 atoms on an NVIDIA GTX 580. OpenMM has a strong emphasis on hardware acceleration, providing not only ease of development but very high performance as well.

We present a graphical processing unit (GPU) implementation of LTMD in OpenMM, thus combining the capabilities of LTMD to integrate timesteps 25× - 50× larger than conventional MD with the hardware acceleration of OpenMM. We use the force calculators in OpenMM to construct numerical Hessians, and have implemented CUDA kernels that provide minimization, projection, and propagation routines for LTMD. We demonstrate correctness of the implementation and speedups of up to 50-fold

over GROMACS with 6 CPU cores and 6-fold over conventional Langevin Leapfrog in OpenMM. This results in nearly 5 µs per day for implicit solvent simulations of the Villin NLE headpiece.

In the remainder of the Chapter we discuss my contribution to the literature in this field in Section 5.2, the LTMD method (Section 5.4), our GPU Implementation (Section 5.5), numerical results that show superior performance (Section 5.6) and correctness (Section 5.6.2), and conclusions and future work (Section 5.7).

5.2 Contribution to the Literature

In the following Sections of this Chapter I will present my work in implementing the Long Time Molecular Dynamics method [88] within the OpenMM framework [64]. Previous work within the LCLS group had developed the mathematical representation of LTMD, the basic numerical methods and derived the analytical Hessians for the CHARMM force field. This implementation had the following limitations which I resolved through my work presented here:

- The LTMD implementation is ProtoMol [54](developed specifically for prototyping MD algorithms) with a limited user base.
- No GPU implementation. Most MD packages now support GPU which was not available for LTMD implemented in ProtoMol. This is a significant disadvantage for an accelerated method.
- The analytical Hessians need to be derived for each force within a force field. Given the number of force fields (AMBER, CHARMM, OPLS)[74] and the variations for such elements as Generalized Bourn[74] for implicit solvent, this is challenging task.

My approach was to implement LTMD within the OpenMM platform, developed by the Pande group at Stanford University and with a large user community (includ-

ing Folding@Home [8]). The platform includes both a reference CPU implementation and a high performance GPU interface, both of which were leveraged during my work.

To remove the requirement for mathematically derived Hessian equations I implemented a Numerical Differentiation scheme. Given that we have highly efficient force calculations (negative of the derivative of the potential energy w.r.t. positions) we can generate the Hessian by numerically differentiating these, again w.r.t. the positions. In practice since we do not need the actual Hessian, which would be $O(N^2 \log N)$, we use a scheme (flexible block method) to differentiate w.r.t. a set of vectors spanning the space of the first few eigenvectors of the Hessian.

5.3 Background

Molecular dynamics (MD) simulations form one of the main methods used in the theoretical study of chemical and biological molecules, wherein the time dependent behavior of a molecular system is computed. These MD simulations can provide detailed information on molecular fluctuations and conformational changes and are used routinely to investigate the thermodynamics, dynamics and structure of chemical and biological molecules. MD methods date back to the 1950's, when Alder and Wainwright [3–5] studied the interactions of hard and elastic spheres leading to important insights into the behavior of simple liquids, and have been refined to the point where realistic simulations of solvated proteins, and the folding of small proteins, is possible.

MD simulations solve the equations of motion of the particles within the system and hence the information generated is at the microscopic level, such as atomic positions and velocities, which can be converted to macroscopic quantities, such as pressure, energy and heat capacity, by the use of statistical mechanics as shown in Section 5.3.1. Statistical mechanics provides the mathematical expressions that relate these macroscopic quantities to the distribution and motion of the atoms and

molecules of an N-body system. One of the main advantages of MD simulations over other schemes, such as the Monte-Carlo method, is that it is possible to study both thermodynamic and time dependent properties.

When considering macroscopic quantities, an ensemble is a collection of all possible systems which have different microscopic states but have an identical macroscopic or thermodynamic state. Examples of a number of ensembles with different characteristics are,

Microcanonical Ensemble (NVE) The thermodynamic state characterized by a fixed number of atoms, N, a fixed volume, V, and a fixed energy, E. This corresponds to an isolated system.

Canonical Ensemble (NVT) This is a collection of all systems whose thermodynamic state is characterized by a fixed number of atoms, N, a fixed volume, V, and a fixed temperature, T.

The ensemble average of some quantity $A(q,p)$ is then defined as,

$$\langle A(q,p)\rangle_{Ensemble} = \int A(q,p)\rho(q,p)dqdp, \tag{5.1}$$

where $\rho(q,p)$ is the probability density of the ensemble. This integral is generally difficult to evaluate as it is necessary to calculate all possible states of the system, and a molecular dynamics simulation calculates the points in the ensemble sequentially in time. For MD simulations we instead determine a time average of $A(q,p)$ which is expressed as, for time $\mathcal{T}$,

$$\langle A(q,p)\rangle_{Time} = \lim_{\mathcal{T}\to\infty} \frac{1}{\mathcal{T}} \int_0^{\mathcal{T}} A(q(t),p(t))\ dt \approx \frac{1}{M}\sum_{i=1}^{M} A(q_i,p_i), \tag{5.2}$$

where M is the number of steps of time Δt and $A(q_i,p_i)$ is the value of $A(q,p)$ at the discreet points $q_i = q(i\Delta t), p_i = p(i\Delta t)$. From this it is possible to calculate

time averages by molecular dynamics simulations, but these experimental averages are then assumed to be ensemble averages. This apparent problem is resolved by the ergodic hypothesis, one of the most fundamental axioms of statistical mechanics, which states that the ensemble average equals the time average i.e.,

$$\langle A(q,p)\rangle_{Ensemble} = \langle A(q,p)\rangle_{Time}. \tag{5.3}$$

The basic concept here is that if the system is allowed to evolve in time indefinitely it will eventually pass through all possible states. Because of this it is important in MD simulations to generate enough representative conformations such that this equality is satisfied and, since the simulations are of fixed duration, a sufficient amount of phase space must be sampled. The proof of sampling from the correct ensemble, for systems thermostatted by Nosé's method, is dependent on the system being ergodic, which is not always true particularly for small or stiff systems. The definition of ergodic as time average being equal to ensemble average is used throughout this Chapter.

The MD simulation method is generally based on Newton's second law or the equation of motion $F = ma$, where F is the force exerted on the particle, m its mass and a its acceleration. From a knowledge of the forces acting within the system it is possible to determine the acceleration of each atom or particle. The equations of motion are then integrated to give a trajectory that describes the positions, velocities and accelerations of the particles as they vary with time, allowing the average values of properties to be determined. The method is deterministic, once the positions and velocities of each atom are known the state of the system can be predicted at any time in the future or the past. Due to the complicated nature of the potential energy functions found in all but the simplest of systems there will be no analytical solution to the equations of motion and they must be solved numerically. Many numerical methods have been developed for integrating these equations but the most effective

for use in MD simulations should conserve energy and momentum and permit a large integration time step. A class of integrators which meet these requirements are Geometric integrators which preserve geometric properties of the original system. The most common of these are time-reversible, a property found in Newtonian mechanics, and symplectic which are applicable for Hamiltonian systems, discussed in Section 5.3.2.

Molecular dynamics simulations are generally computationally expensive, mitigated to some extent by the availability of increasingly faster and cheaper computers. Despite this, simulations of solvated proteins are routinely calculated up to the nanosecond time scale, with simulations into the millisecond time scale reported. Since a significant part of the simulation can be taken up by equilibration, which must be completed before averages can be taken, methods which converge quickly to the correct ensemble are desirable.

5.3.1 Microcanonical and Canonical Ensembles

Although constant energy simulations are straightforward it is not as convenient to derive statistical mechanical formulae from the microcanonical ensemble as it is from the canonical ensemble, as considered by Lebowitz, Percus and Verlet [45]. As a motivation for developing methods which sample from the canonical ensemble both ensembles are studied, and are shown schematically in Figure 5.1.

5.3.2 Microcanonical Ensemble

The microcanonical ensemble in statistical mechanics is equivalent to constant energy conditions, the external control parameters being number of particles N, total energy, E, and the volume V. For a single harmonic oscillator, with angular frequency $\omega = 1$, sampling from the microcanonical ensemble, the q histogram and q, p phase

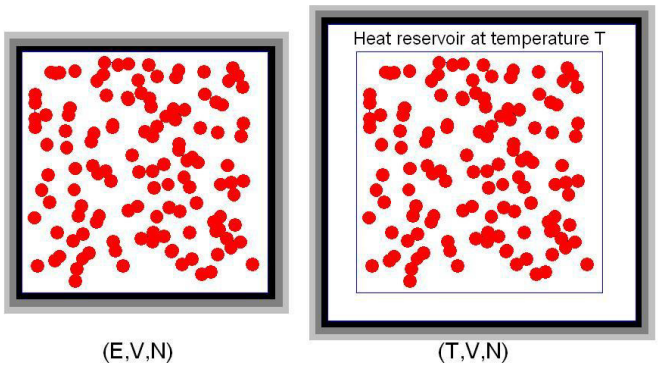


Figure 5.1. Systems in the microcanonical (left) and canonical (right) ensembles. The shaded outlines represent heat insulating walls.

space are shown in Figure 5.2. For a Hamiltonian,

$$H(q,p) = \sum_{i=1}^{N} \frac{p_i^2}{2m_i} + V(q), \tag{5.4}$$

where $V(q)$ is the potential energy, the equations of motion,

$$\dot{q}_i = \frac{p_i}{m_i}, \quad \dot{p}_i = -\nabla_{q_i} V(q), \tag{5.5}$$

conserve the total energy $H(q,p)$, the only phase-space points (q,p) allowed are those on the constant energy hypersurface satisfying $H(q,p) = E$. It is assumed that that every allowed point in phase-space has equal weight in microcanonical ensemble averages, the principle of of equal a priori probability in statistical mechanics. This is closely related to the assumption of ergodicity, where the trajectory of a phase-space vector (q,p) will pass through almost all points within the allowed portion of phase-space, which is integral to the proof of the correct sampling for Nosé schemes. The probability that a phase-space point (q,p) appears in an average is defined by

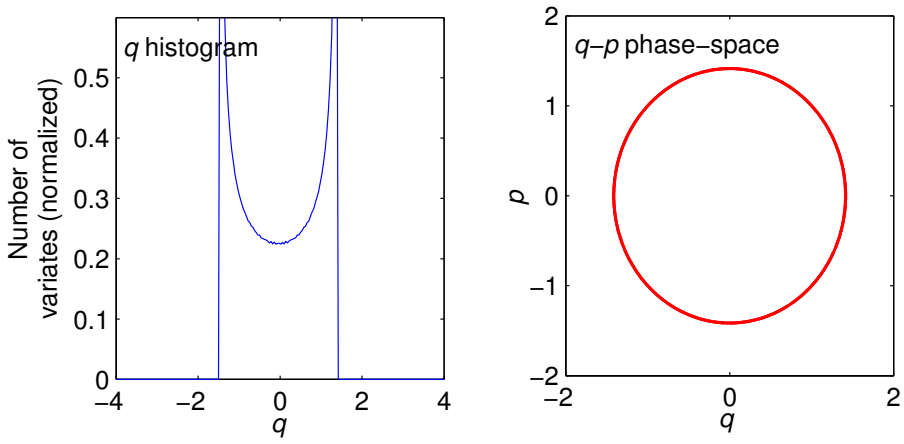


Figure 5.2. Harmonic oscillator for angular frequency $\omega = 1$, q histogram and q, p phase-space for the microcanonical ensemble.

the equilibrium density function $f(q,p)$ and, for the microcanonical ensemble,

$$f_{mc}(q,p) \propto \delta[H(q,p) - E]. \tag{5.6}$$

The Dirac Delta function δ form reflecting the constraint $H(q,p) = E$ with $\delta(x-a) = 0,\ x \neq a$ and $\int_{a-\epsilon}^{a+\epsilon} \delta(x-a)dx = 1,\ \forall \epsilon > 0$. The ensemble average for some quantity $A(q,p)$ is then defined as,

$$\langle A(q,p) \rangle = \frac{\int A(q,p) f(q,p) dqdp}{\int f(q,p) dqdp}. \tag{5.7}$$

By using thermodynamic relations the macroscopic properties of the system can be derived. The Boltzmann relation for entropy is,

$$S = k \ln W, \tag{5.8}$$

where W is the number of microscopic states which, for the microcanonical ensemble,

is given by,

$$\begin{aligned} W &= \frac{1}{N!h^{N_f}} \int^E dE' \int f_{mc}(q,p) dq dp \\ &= \frac{C_1}{N!h^{N_f}} \int \theta(E - H(q,p)) dq dp, \end{aligned} \tag{5.9}$$

for constant C_1 and Planck's constant h. Here $\theta(x)$ is the Heaviside function with $\theta(x) = 1,\ x > 0$, $\theta(x) = 0,\ x < 0$ and $\delta(x) = d\theta(x)/dx$.

The statistical mechanical expressions can then be derived using the methods of Pearson, Halicioglu and Tiller [65]. For systems where the kinetic energy is given by a quadratic form of the momenta, where it is possible to perform the integration in $3N$ dimensional momentum space, (5.9) simplifies to,

$$W = C_2 \int \frac{2}{3N} (E - V(q))^{(3/2)N} dq, \tag{5.10}$$

for constant C_2. Substituting (5.10) into (5.8),

$$S = k \ln \left(C_2 \int \frac{2}{3N} (E - V(q))^{(3/2)N} dq \right). \tag{5.11}$$

From (5.7) the average of a quantity $A(q)$, where $\langle\ \ \rangle_{mc}$ is the average in the micro-canonical ensemble is,

$$\langle A(q) \rangle_{mc} = \frac{\int A(q)(E - V(q))^{(3/2)N-1} dq}{\int (E - V(q))^{(3/2)N-1} dq}. \tag{5.12}$$

Temperature is defined by the thermodynamical relationship,

$$\frac{1}{T} = \left(\frac{\partial S}{\partial E} \right)_V = k \frac{\int \frac{3N}{2} (E - V(q))^{(3/2)N-1} dq}{\int (E - V(q))^{(3/2)N} dq} = \frac{3Nk}{2\langle K \rangle}, \tag{5.13}$$

for kinetic energy $K = E - V(q)$. Then the temperature is related to the average

kinetic energy by the equipartition theorem,

$$T = \frac{2}{3Nk}\langle K\rangle_{mc}. \tag{5.14}$$

The heat capacity is,

$$C_V = \left(\frac{\partial E}{\partial T}\right)_V = \left(\frac{\partial T}{\partial E}\right)_V^{-1} = k\left(1 - \left(1 - \frac{2}{3N}\right)\langle K\rangle_{mc}\left\langle\frac{1}{K}\right\rangle_{mc}\right)^{-1}. \tag{5.15}$$

The average of the inverse of the kinetic energy in the thermodynamical limit is approximated by,

$$\left\langle\frac{1}{K}\right\rangle_{mc} = \frac{1}{\langle K\rangle}\left(1 + \frac{\langle(\delta K)^2\rangle}{\langle K\rangle^2}\right), \tag{5.16}$$

where $K = \langle K\rangle + \delta K$ and $\langle(\delta K)^2\rangle = \langle K^2\rangle - \langle K\rangle^2$. Substituting (5.16) into (5.15) we get,

$$C_V \approx k\left(\frac{2}{3N} - \frac{\langle(\delta K)^2\rangle}{\langle K\rangle^2}\right)^{-1}, \tag{5.17}$$

an equation obtained by Lebowitz, Percus and Verlet [45]. The fluctuation of the kinetic energy in the microcanonical ensemble is then,

$$\langle(\delta K)^2\rangle_{mc} = \frac{2}{3N}\langle K\rangle^2\left(1 - \frac{3Nk}{2C_V}\right). \tag{5.18}$$

5.3.3 Canonical Ensemble

The canonical ensemble relates to simulations where temperature T is fixed instead of total energy E. This ensemble is shown schematically in Figure 5.1 where the original system is surrounded by large external system and energy, but not particles, can be exchanged between them. The external system, or heat bath, must be large in relation to the original system so that temperature changes caused by any energy transfer will be negligible. If we define the temperature of the original system by the average total kinetic energy (5.14), the temperature will be maintained at a constant

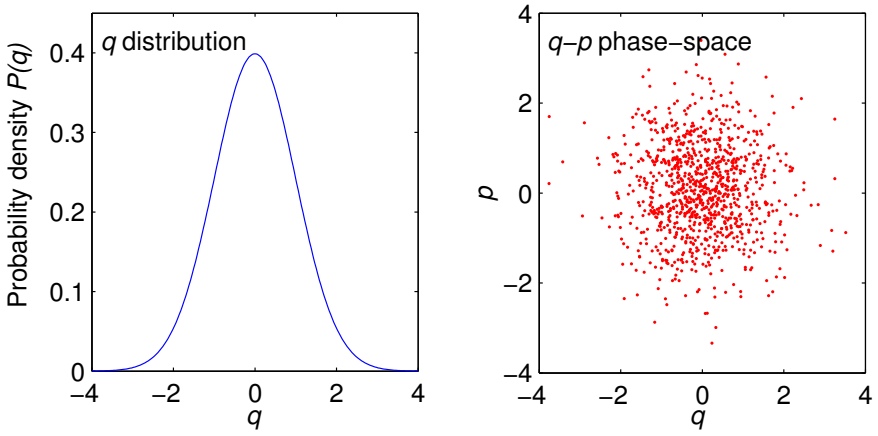


Figure 5.3. Harmonic oscillator with $\omega = 1$, q distribution and q, p phase-space for the canonical ensemble.

value by thermal contact with the heat bath. Since temperature is now constant the total energy of the system fluctuates and the distribution is now the canonical distribution,

$$f_c(q,p) = \frac{1}{\sqrt{2\pi kT}} \exp\left(-\frac{H(q,p)}{kT}\right). \tag{5.19}$$

For a single harmonic oscillator, with $\omega = 1$, sampling from the canonical ensemble the q distribution and q, p phase space are shown in Figure 5.3. The relationship between the distribution functions (5.6) and (5.19) is given by the Laplace transformation, with energy E,

$$f_c(q,p;T) = \int dE \exp\left(-\frac{E}{kT}\right) f_{mc}(q,p;E). \tag{5.20}$$

The thermodynamical potential in the canonical ensemble is the Helmholtz energy

$F(T,V,N)$ given by,

$$\begin{aligned} F(T,V,N) &= -kT\ln\left(\int f_c(q,p)dqdp\right) \\ &= -kT\ln\left(\frac{1}{\sqrt{2\pi kT}}\int \exp\left(-\frac{H(q,p)}{kT}\right)dqdp\right). \end{aligned} \tag{5.21}$$

The heat capacity is then expressed as a fluctuation of the total energy,

$$C_V = \frac{\langle H^2\rangle_c - \langle H\rangle_c^2}{kT^2}. \tag{5.22}$$

The average and fluctuation of kinetic energy are then,

$$\langle K\rangle_c = \frac{3N}{2}kT, \tag{5.23}$$

and,

$$\langle(\delta K)^2\rangle_c = \frac{2}{3N}\langle K\rangle^2 = \frac{3N}{2}(kT)^2. \tag{5.24}$$

It is noted that quantities which are first order derivative of the thermodynamical potential, such as total energy E and pressure P, are independent of the ensemble but second or higher order derivatives, as we see with heat capacity, are not.

The fluctuation of kinetic energy in the canonical ensemble (5.24) is greater than that in the microcanonical ensemble 5.18c, and this inequality can be used to confirm that the sampling is correct for constant temperature simulations.

5.3.4 Constant Temperature Methods

The most common constant temperature methods, required for sampling from the canonical ensemble, are: the constraint method, the stochastic method and the extended system method. Brief descriptions of these methods follow.

5.3.5 Stochastic Method

The thermal motion of a particle, in a macroscopic scale, appears to be driven by a random force and hence stochastic methods, such as Monte Carlo and Brownian dynamics, are applicable. Equations similar to Langevin's equation for Brownian dynamics were proposed by Schneider and Stoll [75],

$$m_i \frac{d^2 q_i}{dt^2} = -\nabla_{q_i} V(q) - \gamma \dot{q}_i + R_i(t), \tag{5.25}$$

where a friction force, with coefficient γ, and a random force $R_i(t)$ are added. The random force, temperature T and friction coefficient γ are related by the second fluctuation dissipation theorem,

$$\langle R_i(t_1) R_j(t_2) \rangle = \delta_{ij} 2kT\gamma\delta(t_1 - t_2). \tag{5.26}$$

Thermal agitation due to the random force and slowing due to the friction force balance to keep the temperature constant.

Andersen [6] has proposed a more direct method where occasional collisions between a particle and hypothetical particles cause the particle to lose its memory, the velocity is reset to a value randomly selected from a Maxwell distribution at temperature T.

Both of these approaches provide sampling from the canonical ensemble for position q, however care needs to be exercised in their use. For example if the frequency of the random collisions in Andersen's method is too high the particle loss of memory occurs in too short a time, leading to the velocity autocorrelation function damping quickly [90].

5.3.6 Forces

The calculation of the force contributions is one of the most costly parts of molecular dynamics simulations. This is because most force calculations are calculated between all pairs of atoms, leading to a $O(N^2)$ algorithmic complexity. This itself is not a problem for small systems (100s of atoms), but as the problem starts scaling up to use larger and larger proteins (100,000s of atoms) it quickly dominates.

In molecular dynamics a number of different forces are used to approximate the physical interactions occurring in reality. These include:

Bonds The force created between two bonded atoms. It is represented as a spring in most molecular dynamics simulations, for potential energy:

$E_{\text{bonds}} = \sum_{\text{bonds}} k_b(b - b_0)^2$,

for coefficient k_b, rest length b_0 and length b.

Angles The force between three atoms where two are connected only to the third, for potential energy:

$E_{\text{angles}} = \sum_{\text{angles}} k_\theta(\theta - \theta_0)^2$,

for coefficient k_θ, rest angle θ_0 and angle θ.

Dihedrals The force between four atoms connected in a chain that are able to be rotated as two planes around a single axis for potential energy:

$E_{\text{dihedrals}} = \sum_{\text{dihedrals}} k_\phi[1 + \cos(n\phi + \delta)]$,

for coefficient k_ϕ, offset δ and angle ϕ and integer n.

Torsion/Impropers This force is generated by the angle between the two planes defined by four atoms for potential energy:

$E_{\text{impropers}} = \sum_{\text{impropers}} k_\omega(\omega - \omega_0)$,

for coefficient k_ω, rest angle ω_0 and angle ω.

Lennard-Jones This force provides repulsion of atoms when they get very close to each other and a long range attraction force at distances for potential energy:

$E_{\text{LJ}} = \sum_{\text{all pairs}} \epsilon \left[\left(\frac{R_{\min_{ij}}}{r_i j} \right)^{12} - \left(\frac{R_{\min_{ij}}}{r_i j} \right)^{6} \right]$,

for coefficient ϵ, atomic distance between atoms i and j of r_{ij} and parameters $R_{\min_{ij}}$.

Coulomb This represents the electrostatic potential between two charged atoms for potential energy:

$E_{\text{coulomb}} = \sum_{\text{all pairs}} \frac{q_i q_j}{\epsilon r_{ij}}$,

for coefficient ϵ, charge for atom i of q_i, charge for atom j of q_j and atomic distance between atoms i and j of r_{ij}.

Generalized Born A force that simulates the protein being in a solvent by calculating the born radius to determine surface accessible area.

To calculate the force from the Potential Energy we assume a Conservative Force [57], this can be formulated from a Hamiltonian or Lagrangian assumption. So force F is given by,

$$F = -\nabla E, \tag{5.27}$$

i.e. the negative of the gradient of E, the potential energy.

For example, for the Bond force between two atoms, where the atomic distance for atoms i and j is $b_{ij} = ||\mathbf{r}_j - \mathbf{r}_i||$ for atomic positions $\mathbf{r}_i, \mathbf{r}_j$,

$$\begin{aligned} \mathbf{F}_{\text{bonds}_{ij}} &= -\nabla E_{\text{bonds}_{ij}}, \\ &= -\nabla \left[k_{b_{ij}} (||\mathbf{r}_j - \mathbf{r}_i|| - b_0)^2 \right], \\ &= -2k_{b_{ij}} (||\mathbf{r}_j - \mathbf{r}_i|| - b_0)[-\hat{\mathbf{r}}_{ij}, \hat{\mathbf{r}}_{ij}]^{\text{T}}, \end{aligned} \tag{5.28}$$

where $\hat{\mathbf{r}}_{ij} = \frac{\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||}$.

5.3.7 Hessian

The LTMD method relies on the Hessian of the Potential Energy to partition the system into the Dynamical and statistical sub-spaces. The Hessian is the second derivative of the potential energy w.r.t. positions. For the Bond force this is given by,

$$\mathbf{H}_{\text{bonds}_{ij}} = 2k_{b_{ij}} \frac{||\mathbf{r}_j - \mathbf{r}_i|| - b_0}{||\mathbf{r}_j - \mathbf{r}_i||} \begin{bmatrix} I & -I \\ -I & I \end{bmatrix} + \frac{2k_{b_{ij}} b_0}{||\mathbf{r}_j - \mathbf{r}_i||} \begin{bmatrix} \hat{\mathbf{r}}_{ij}\hat{\mathbf{r}}_{ij}^{\mathrm{T}} & -\hat{\mathbf{r}}_{ij}\hat{\mathbf{r}}_{ij}^{\mathrm{T}} \\ -\hat{\mathbf{r}}_{ij}\hat{\mathbf{r}}_{ij}^{\mathrm{T}} & \hat{\mathbf{r}}_{ij}\hat{\mathbf{r}}_{ij}^{\mathrm{T}} \end{bmatrix} \quad (5.29)$$

5.4 LTMD

The central concept of LTMD is to increase the timescale of an MD simulation by computing coarse-grained normal modes (CNMA) and dividing the system's degrees of freedom (DOF) into fast and slow frequency motions. Then the fast frequency motions, which limit the step size, are approximated using Brownian Dynamics (or minimization). True dynamics are only used for the slow frequency motions, which represent a smoother energy landscape and where the timestep sufficient for stability is much larger. This introduces two additional costs to the LTMD method: calculating the mass weighted Hessian matrix $H = M^{-\frac{1}{2}}\mathcal{H}M^{-\frac{1}{2}}$, where $\mathcal{H}$ is the system Hessian, and diagonalizing it. This process must be done repeatedly throughout a simulation such that an accurate frequency division is always available. The time between diagonalizations is in the range of 10 ps to 100 ps for typical biomolecules. Since naive diagonalization is prohibitively expensive, $O(N^3)$, we have devised an approximate diagonalization method that has complexity $O(N^{9/5})$. We will show that this is sufficient to produce real speedups over conventional MD.

We consider first the propagation of the biomolecule in the fast and slow domains, and then the method of partitioning the dynamical space into fast and slow domains.

5.4.1 Propagator

The LTMD propagator uses the same basic technique as the Langevin method. The canonical Langevin equation is given as

$$\begin{aligned} \mathrm{d}X &= v\mathrm{d}t, \\ M\mathrm{d}v &= f\mathrm{d}t - \Gamma M v\mathrm{d}t + (2k_BT)^{\frac{1}{2}}\Gamma^{\frac{1}{2}}M^{\frac{1}{2}}\mathrm{d}W(t), \end{aligned} \tag{5.30}$$

where $f = -\nabla U(X)$, t is time, $W(t)$ is a collection of Wiener processes, k_B is the Boltzmann constant, T is the system temperature, v are the velocities and Γ is the diagonalizable damping matrix. The system diffusion tensor D gives rise to $\Gamma = k_BTD^{-1}M^{-1}$. D is chosen to model the dynamics of an implicit solvent.

In LTMD, the forces and random perturbations (heat bath) are partitioned, using a projection matrix P_f (and its complement $P_f^{\perp}$), so that the normal Langevin equation is approximated in the slow space but is over-damped in the fast space. Here,

$$P_f = M^{\frac{1}{2}}QQ^{\mathrm{T}}M^{-\frac{1}{2}},\ P_f^{\perp} = M^{\frac{1}{2}}\left(I - QQ^{\mathrm{T}}\right)M^{-\frac{1}{2}}, \tag{5.31}$$

where Q is the set of low frequency eigenvectors as columns and M is the diagonal system mass matrix.

Thus, in LTMD, the projected Langevin equation that models the coarse-grained dynamics of implicitly-solvated proteins is given as:

$$\begin{aligned} \mathrm{d}X &= v\mathrm{d}t, \\ M\mathrm{d}v &= P_f f\mathrm{d}t - \Gamma M v\mathrm{d}t + P_f(2k_BT)^{\frac{1}{2}}\Gamma^{\frac{1}{2}}M^{\frac{1}{2}}\mathrm{d}W(t). \end{aligned} \tag{5.32}$$

For the high frequency dynamics Brownian motion (over-damped) is often solved with the Euler-Maruyama method. This is similar in form to a "steepest descent"

minimizer, which we use for efficient damping. Then,

$$X^{n+1} = X^n + \eta P_f^{\perp} f, \tag{5.33}$$

where η is determined by a "line search" algorithm, X^n is the current position and X^{n+1} is the new position. This algorithm is iterated until the difference in system energy for successive steps is less than some threshold value.

After minimization is performed, the rest of the Euler-Maryuma approximation is computed by adding noise (random terms) to the fast space according to

$$X^{n+1} = X^n + \sqrt{2\Delta t \bar{\Gamma}^{-1} k_B T} M^{-1} P_f^{\perp} M^{1/2} z \tag{5.34}$$

where z is a random variable vector sampled from a Gaussian distribution, $\bar{\Gamma}$ is the damping matrix for the fast space, and Δt is the timestep.

5.4.2 Partitioning of the Dynamical Space of Biomolecules

To partition the dynamical space of the biomolecule we need to calculate the system's mass weighted Hessian and then diagonalize it to find a quadratic approximation to the system. This then identifies collective motions, or normal modes, and their associated frequency. A choice of cutoff frequency defines a set of normal modes, ordered according to their eigenvalues, that span the "slow" modes of interest.

The sparsity of the Hessian matrix is dependent on how we calculate the long range forces in the force field. For methods such as Ewald Summation [21] the Hessian is full. Studies[88] of biomolecules have shown that the important low frequency motions are dominated by motions of the backbone alpha carbon atoms rather than long range forces. In this case switches are generally used to reduce the forces to zero beyond a given distance so that the Hessian is sparse and the calculation cost is $O(N)$ (for system size N) for *analytical* Hessian calculation. In MD codes such as

OpenMM, where analytical Hessians are not available, the Hessian must be calculated numerically at a cost of $O(N^2)$ assuming force calculation cost of $O(N)$.

Diagonalization of a matrix ("brute force") is generally $O(N^3)$ which would become prohibitive for large systems, although there are other methods that may offer reduced cost. The Rotation Translation Block (RTB) method groups sequential residues into blocks, which are then treated as rigid bodies, and the movement of the entire protein is expressed as the rotations and translations of these blocks. Thus the dimensionality of the diagonalization problem is reduced and a diagonalization cost proportional to $O(N^{9/5})$ is achieved [18, 89]. Variants of this method exist that perform different approximations [27–30].

LTMD uses a coarse grained diagonalization method called Flexible Block Method (FBM) [41] that reduces the expected cubic run-time to $O(N^{9/5})$. FBM is similar to RTB, including the sequential partitioning method, but also includes the internal flexibility of the blocks, greatly increasing accuracy of the resulting eigenvectors with the same complexity. FBM avoids the calculation of the full Hessian and diagonalizes smaller matrices based on a knowledge of the structure of the biomolecule. Coarse-graining involves computing instead a *block* mass weighted Hessian:

$$\begin{bmatrix} H_{11} & & & \\ & H_{22} & & \\ & & \ddots & \\ & & & H_{mm} \end{bmatrix}$$

where each H_{ii} is a mass weighted Hessian matrix of the potential energy accounting for interactions only within some group of one or more residues i, and each block is assumed to be independent of other blocks.

Computing the Hessian of the potential energy U requires calculating the second-order derivatives of $U(X)$ where X is the vector of atomic positions. We obtain the force $F(X) = -\nabla U(X)$, the gradient of U w.r.t. X, and thus can approximate the

Hessian using the first-order derivatives of $F(X)$. This can be accomplished for the j^{th} column of H using the central difference method where we perturb the atomic positions by δx_j, a small value added to the j^{th} degree of freedom, and compute both $F(X + \delta x_j)$ and $F(X - \delta x_j)$, then calculate the column vector as,

$$H_j = \frac{F(X - \delta x_j) - F(X + \delta x_j)}{\delta x_j}. \tag{5.35}$$

We can then diagonalize each individual block to obtain a set of eigenvalues and eigenvectors. If the blocks are not at a minimum, then the Hessian will not contain the true rotational degrees of freedom[32, 44]. Therefore, the translation and rotational degrees of freedom are computed explicitly for each block using Eq. (5.36) - (5.38), (5.40). A new set of eigenvectors is formed by combining the sets of eigenvectors and translation and rotational degrees of freedom and using a modified Gram-Schmidt process to orthogonalize the set. Should any vector's norm become less than $1/20^{th}$ of its original norm after orthogonalization, it is assumed that the vector is a duplicate of the explicitly-calculated translation or rotation vectors and removed from the set.

$$T_1 = \left\{ \frac{\sqrt{m_1}}{M}, 0, 0, \frac{\sqrt{m_2}}{M}, 0, 0, \cdots, \frac{\sqrt{m_N}}{M}, 0, 0 \right\}, \tag{5.36}$$

$$T_2 = \left\{ 0, \frac{\sqrt{m_1}}{M}, 0, 0, \frac{\sqrt{m_2}}{M}, 0, \cdots, 0, \frac{\sqrt{m_N}}{M}, 0 \right\}, \tag{5.37}$$

$$T_3 = \left\{ 0, 0, \frac{\sqrt{m_1}}{M}, 0, 0, \frac{\sqrt{m_2}}{M}, \cdots, 0, 0, \frac{\sqrt{m_N}}{M} \right\}. \tag{5.38}$$

where

$$M = \sqrt{m_1 + m_2 + \cdots + m_N}\ . \tag{5.39}$$

$$R_i = \frac{\hat{R}_i}{||\hat{R}_i||},\ \hat{R}_i = \{r_{i,1}, r_{i,2}, \cdots, r_{i,N}\}, \tag{5.40}$$

where

$$r_{1,j} = \sqrt{m_j}\{0, d_{j,z}, -d_{j,y}\}, r_{2,j} = \sqrt{m_j}\{-d_{j,z}, 0, d_{j,x}\}, r_{2,j} = \sqrt{m_j}\{-d_{j,y}, -d_{j,x}, 0\} \tag{5.41}$$

for vector d, with xyz coordinates, representing the difference between the atom position and the center.

A $3N \times k$ matrix E is assembled from the block eigenvectors corresponding to the k lowest eigenvalues. An appropriate value of k which still spans the low frequency space will vary based on the particular composition of the protein's residues, but we have determined a typical value of k is around 12 per residue[41].

The reduced set of eigenvectors is used to compute the quadratic product $S = E^T HE$. The matrix S will be of smaller dimension $(k \times k)$ than H $(3N \times 3N)$ but will still account for the appropriate degrees of freedom. S is then diagonalized to obtain a set of eigenvectors Q which by definition satisfy the equation $Q^T SQ = D$ where D is a diagonal matrix. Combining our two equations, we get: $(EQ)^{\mathrm{T}} HEQ = D$, and so can finally represent $V = EQ$ as an orthogonal set of vectors that span the low frequency space and with the property that $V^{\mathrm{T}} HV = D$. If we sort the eigenvectors of Q by corresponding eigenvalue, V will be ordered as well. We can then select the m slowest frequency modes by simply choosing the first m vectors of V.

We note that rather than forming the quadratic product $S = E^T HE$ in the usual way, which would require the calculation of the Hessian H, we can calculate an approximation to the matrix HE directly using a first order numerical differentiation scheme. This is accomplished by perturbing the positions by $\epsilon M^{-\frac{1}{2}} E_i$ for some small scalar value ϵ and the i^{th} column of E (denoted E_i), we then find the force difference

scaled by $1/\epsilon$ and multiply by $M^{-\frac{1}{2}}$. Pre multiplication by E^{T} will then yield S. For example, using Taylor series expansion for each E_i perturbation,

$$\nabla U\left(X+\epsilon M^{-\frac{1}{2}}E_i\right) \;=\; \nabla U(X)+\epsilon\mathcal{H}M^{-\frac{1}{2}}E_i+\ldots, \tag{5.42}$$

for potential energy $U(X)$ and positions X. We note that the system force at positions X is given by $f(X) = -\nabla U(X)$. Then, multiplying Eq. (5.42) by $M^{-\frac{1}{2}}$ and re-arranging, we have,

$$HE_i \;=\; M^{-\frac{1}{2}}\mathcal{H}M^{-\frac{1}{2}}E_i \;=\; M^{-\frac{1}{2}}\left[\frac{f(X)-f\left(X+\epsilon M^{-\frac{1}{2}}E_i\right)}{\epsilon}\right]+O(\epsilon), \tag{5.43}$$

which represents the i^{th} column vector of HE. By repeating this n times, for each i, we can assemble the complete matrix HE.

A second order centered difference method is also possible at twice the cost:

$$HE_i \;=\; M^{-\frac{1}{2}}\left[\frac{f\left(X-\epsilon M^{-\frac{1}{2}}E_i\right)-f\left(X+\epsilon M^{-\frac{1}{2}}E_i\right)}{2\epsilon}\right]+O\left(\epsilon^2\right). \tag{5.44}$$

5.5 Implementation

OpenMM provides an API that performs GPU molecular dynamics calculations while masking the underlying details of GPU programming. The software employs the Plugin [24] design pattern which allows package extensions to be externally compiled into libraries which are loadable at run-time by (for example) setting a flag. We designed our implementation of LTMD as an OpenMM plugin that runs in parallel on either or both the GPU and CPU. MD force calculations are the slowest portions of our algorithm when running on the CPU, and we thus reserve those for the GPU while running some sparse matrix calculations using parallel libraries such as Intel's Math Kernel Library (MKL, [38]). We further divide our implementation into two

libraries, one for our API which is responsible for user interaction and the second is the plugin itself which is invoked dynamically when a GPU calculation takes place.

5.5.1 Propagator

The implementation of the LTMD propagator GPU kernel follows the same format as the OpenMM implementation of Langevin. In addition to the propagator kernel mapping sets of atoms to CUDA threads we also need to project the forces using a local product with reduction across all nodes. In addition we also implement the minimizer in the kernel to relax the fast sub-space of the biomolecule after propagation in the slow sub-space.

5.5.2 Diagonalization with Flexible Block Method

A major aspect of the GPU implementation of LTMD is the frequency partition, which involves three main areas of the algorithm:

5.5.2.1 Computation of Block Hessian

To compute the block Hessian, we create a separate OpenMM context in which all interactions between atoms in different blocks are removed. New force objects are instantiated for the block context where bonds, angles, dihedrals, and RB dihedrals that span atoms in multiple blocks are removed. Custom forces which allow the removal of interactions between atoms in multiple blocks are used for the non-bonded forces. Implicit solvent (Generalized Born) is not used in the calculation of the blocks.

As the blocks have no interactions, perturbations to a DOF in one block i have no effect on the forces of the atoms in the other blocks. This is equivalent to saying that $H_{i,j} = 0$ where $j \neq i$ for blocks i and j. We exploit this behavior to reduce the number of required force calculations. With each step of the numerical differentiation, we perturb the k^{th} DOF in each block and then compute the forces once for

the context. The components from the resulting column vector $\mathbf{v}$ are then copied into their respective block Hessians such that element $H_{i,l_i+k} = \mathbf{v}_{l_i+k}$, where i is a block and l_i is the first DOF in block i. With this approach, we only need one or two force calculations (depending on whether first-order or second-order numerical differentiation is used) for each DOF in the largest block instead of for every DOF in the system.

Since OpenMM does not allow two CUDA contexts to be instantiated at the same time, the forces for the blocks are computed using an OpenCL [63] context.

5.5.2.2 Block Diagonalization

Since individual blocks H_{ii} of the block Hessian are independent of one another, we can diagonalize them in parallel using OpenMP [15]. The procedure for finding the eigenvectors for each block Hessian is as follows: The block Hessian is diagonalized using the `dysevr` routine in LAPACK or Intel MKL. The eigenvectors are sorted by the magnitude of their eigenvalues. The block's rotation and translation vectors are computed geometrically. A new set of eigenvectors is formed from the rotation and translation vectors and original eigenvectors with care taken to ensure that the vectors are inserted so as to preserve their relative ordering. Lastly, the new set of eigenvectors is orthogonalized vector-by-vector using a modified Gram-Schmidt process. If a vector's norm is less than $1/20^{th}$ after orthogonalization, the vector is removed from the set. Otherwise, the vector is normalized. The process is repeated for each block Hessian, using OpenMP to parallelize the process such that each block is handled into its own a separate thread.

5.5.2.3 Computation of S

As described in Section 5.4.2, we can calculate the matrix $S = E^{\mathrm{T}}HE$ by first calculating the columns of the matrix $E^{\mathrm{T}}H$ and then post multiplying by E. The

columns of the matrix $E^{\mathrm{T}}H$ can be found using Eq. (5.44). The original OpenMM context, which is also used for propagation, is used to compute the forces for the numerical differentiation.

5.6 Results

In the following sections we present results for benchmarking, validation and choice of parameters. In Section 5.6.1 we compare relative performance of OpenMM LTMD, OpenMM Langevin and GROMACS when simulating a range of proteins consisting of 512 to 1251 atoms. In Section 5.6.2 we validate the LTMD method using Ala5, a small helical peptide, and Villin NLE a fast folding protein. Finally, in Section 5.6.3 we consider the effects of the parameters required for LTMD on accuracy.

5.6.1 Benchmarks

OpenMM LTMD was benchmarked against OpenMM with Langevin dynamics and GROMACS for four different protein systems (see below). We evaluated differences in absolute performance given as ns/day as well as relative speed ups. OpenMM Langevin was run with the CUDA platform, while OpenMM LTMD used the CUDA platform for propagation and the OpenCL platform for computing the block forces for the Flexible Block Method (FBM). GROMACS was configured to run with 6 threads. A single machine with an Intel Xeon E5645 2.6GHz processor, 24GB of RAM and two NVIDIA GeForce 580GTX graphics cards, running a 64-bit version of Red Hat Linux 6, was used for all of the tests. The protein system models were prepared with Amber96-SB and the Generalized Born OBC implicit-solvent model. All models were run at 370 ° K.

1. Villin NLE (512 atoms, 35 residues)

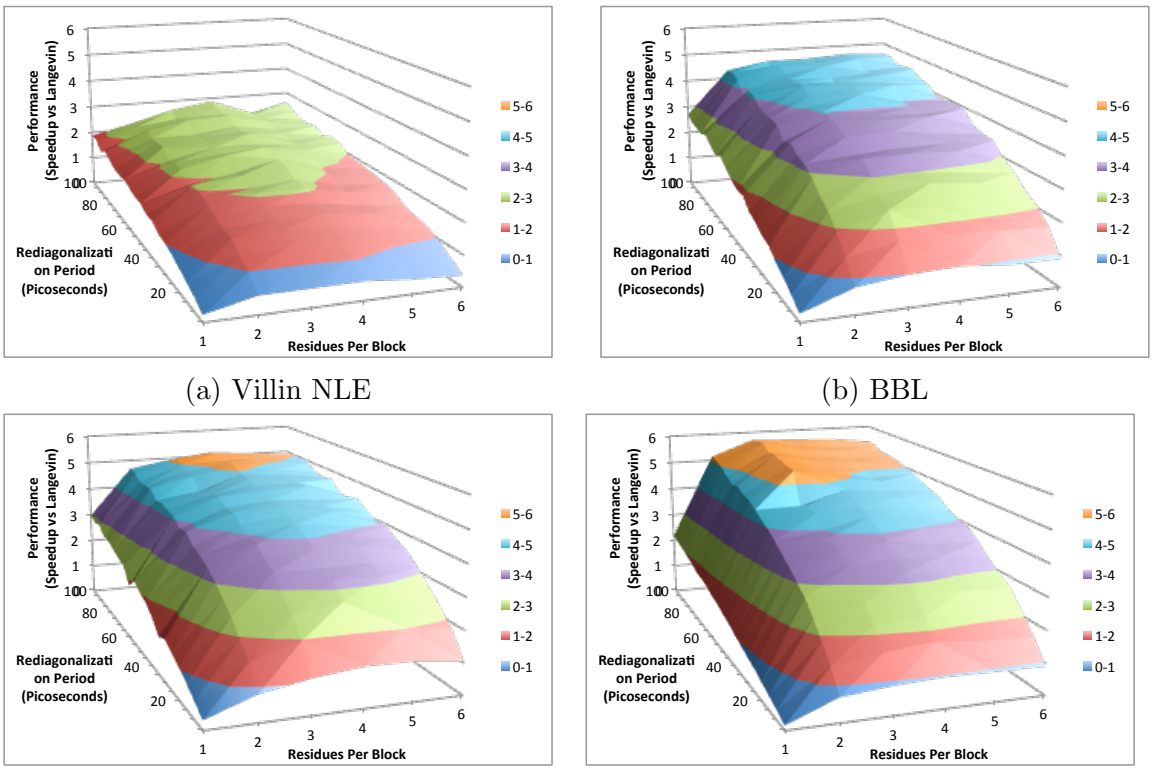
2. BBL (707 atoms, 47 residues)
3. N-Terminal Domain of L9 (881 atoms, 55 residues)
4. Lambda Repressor (1251 atoms, 80 residues)

5.6.1.1 Parameter Choices for Optimal Performance

The performance of LTMD is dependent on a number of parameters. The main determinant is how often new eigenvectors are calculated (re-diagonalization of the Hessian). The cost of the eigenvector calculation is, in turn, influenced by the size of the block Hessians (number of residues per block). It was shown that the cost of calculating the eigenvectors was $O(N^{\frac{9}{5}})$ using the Flexible Block Method[41] (FBM) with sparse and analytical Hessians. OpenMM LTMD uses numerical differentiation for calculating the block Hessians and quadratic-product matrix S. To quantify the effect of both the period of eigenvector calculation and the number of residues per block, we calculated the speed up (ns per day for LTMD divided by ns per day for OpenMM Langevin) for eigenvector calculation periods in the range of 10 ps to 100 ps and residues per block in the range of 1-6.

OpenMM LTMD's performance for the four test systems is presented as a function of the number of residues and rediagonalization period in Figure 5.4. The parameter space has a relatively convex shape which makes the method's performance easy to optimize and relatively robust to different choices of parameters. Although OpenMM LTMD performance was optimal with 3 residues per block and a rediagonalization period of 100 ps, similar choices of parameters (e.g., 2 or 4 residues per block and rediagonalization periods of 60 ps to 100 ps) provide similar performance.

When compared with OpenMM Langevin, speed ups for the smallest model, Villin NLE, were limited to just over two times, while speed ups in excess of 5 times were seen for the larger models (Table 5.1). LTMD's absolute performance reached 4.6 µs/day for Villin NLE.



(a) Villin NLE

(b) BBL

(c) N-Terminal Domain of L9

(d) Lambda Repressor

Figure 5.4. A comparison of the performance speed up on four proteins versus OpenMM Langevin for different values of the number of residues per block and the rediagonalization period.

5.6.1.2 Comparisons of Relative and Absolute Performance

Using the optimal choices for parameters from Section 5.6.1.1, OpenMM LTMD's performance was compared to that of OpenMM Langevin and GROMACS with 6 cores. Benchmark simulations were run for each of the four protein systems (Villin NLE, BBL, NTL9, and Lambda Repressor). Absolute performance numbers (given in ns / day) are shown in Figure 5.5, while relative speed ups were computed and presented in Table 5.1.

OpenMM LTMD showed significant increases on absolute performance in comparison with GROMACS and OpenMM Langevin. OpenMM LTMD's absolute perfor-

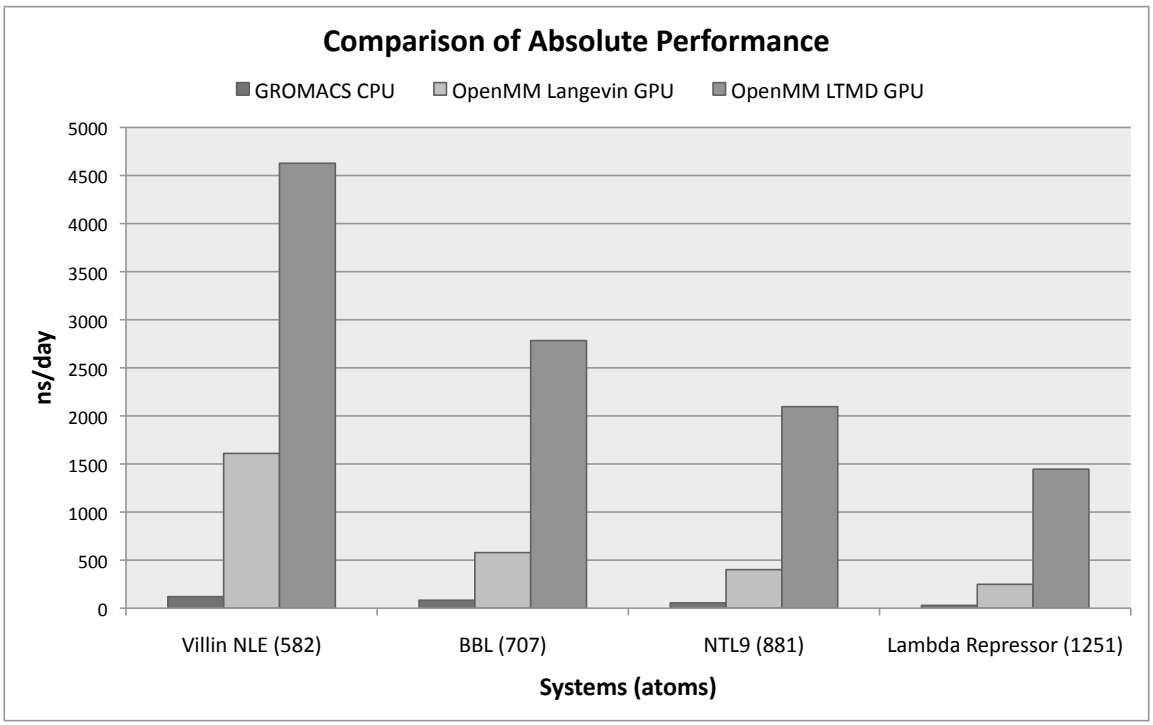


Figure 5.5. Comparison of absolute performance (ns/day) between GROMACS with 6 CPU cores, OpenMM Langevin, and OpenMM LTMD.

mance peaked at 4.6 µs/day for Villin NLE compared to 0.1 µs/day for GROMACS and 1.6 µs/day for OpenMM Langevin. On larger systems, OpenMM LTMD achieved 2.8 µs/day for BBL, 2.1 µs/day for NTL9, and 1.4 µs/day for Lambda Repressor compared with 0.08 (BBL), 0.06 (NTL9), and 0.03 (Lambda Repressor) µs/day with GROMACS and 0.6 (BBL), 0.4 (NTL9), and 0.2 (Lambda Repressor) µs/day with OpenMM Langevin.

Comparing relative performance (speed up) of OpenMM LTMD over GROMACS and OpenMM Langevin shows that OpenMM LTMD scales better for larger systems versus the other methods. Table 5.1 details the speed up of OpenMM LTMD over OpenMM Langevin and GROMACS. OpenMM LTMD provides speed ups of up to 5.8× over OpenMM Langevin and 49.7× over GROMACS. Of particular interest is that the speed up provided by OpenMM LTMD *increases* for larger protein systems,

implying, along with absolute performance displayed in Figure 5.5, that OpenMM LTMD scales better than either OpenMM Langevin or GROMACS with system size.

TABLE 5.1

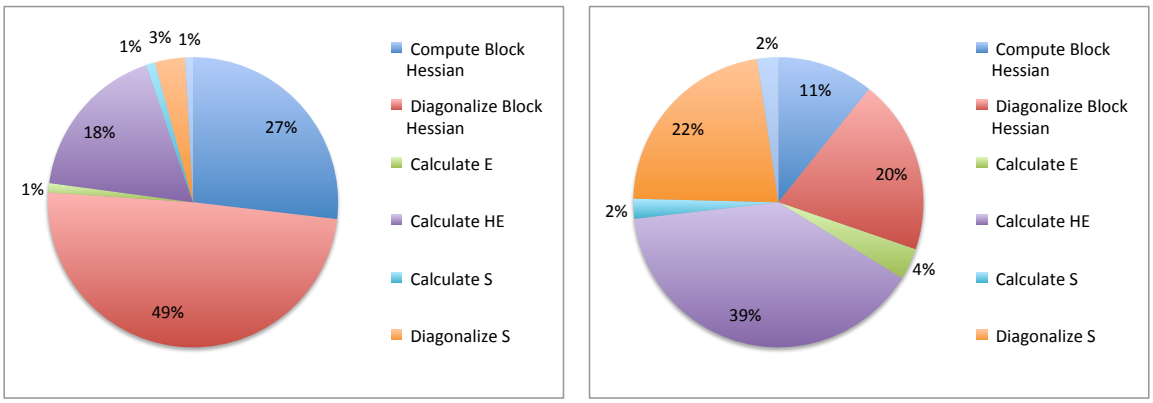
COMPARISON OF RELATIVE PERFORMANCE OF OPENMM LTMD VS GROMACS AND OPENMM LANGEVIN

Proteins		Speed Up of OpenMM LTMD vs	
Name	Number of Atoms	OpenMM Langevin	GROMACS
Villin NLE	582	2.9×	38.5×
BBL	707	4.8×	33.6×
NTL9	881	5.2×	37.9×
Lambda Repressor	1251	5.8×	49.7×

5.6.1.3 Run-time Breakdown

To provide insight as to where time is being spent, we detailed the steps of the method and profiled the run-time of each step individually. Simulations were run using the optimal parameters choices detailed in Section 5.6.1.1. The steps are as follows:

1. Computation of the block Hessians using numerical differentiation of the forces
2. Diagonalization of the block Hessians
3. Creation of E by sorting the calculated eigenvalues from the hessian and culling those below a threshold



(a) Villin NLE

(b) Lambda Repressor

Figure 5.6. A breakdown of the time spent in each section of the analysis portion of the code for the smallest (Villin NLE) and largest (Lambda Repressor) systems tested.

4. Computation of HE using numerical differentiation of the forces
5. Multiplication HE and E^T to find S.
6. Diagonalization of S to find Q
7. Finding the approximate normal modes U by multiplying E and Q.
8. The propagation of the system.

Table 5.2 gives the absolute time for each step in milliseconds for each of the four protein systems, while Figure 5.6 gives the percent time spent on each step for Villin NLE (our smallest system) and Lambda Repressor (our biggest system). For smaller systems, such as Villin NLE, the run time is dominated by the numerical differentiation of the block Hessians and their diagonalization. Whereas, for larger systems such as Lambda Repressor, run time is dominated by the numerical calculation of S and its diagonalization. The large amount of time spent on diagonalization versus propagation explains why longer rediagonalization periods provide better performance as found in Section 5.6.1.1.

TABLE 5.2

MILLISECOND RUNTIME BREAKDOWN PER SIMULATION STEP

System	Atoms	Comp. Blocks	Diag. Blocks	Calc. E	Calc. HE	Calc. S	Diag. S	Calc. U	Propagate
Villin NLE	582	103.4	189.5	4.0	67.8	3.9	12.9	3.4	37.1
BBL	707	135.1	325.5	6.1	121.2	6.7	22.0	7.5	62.5
NTL9	881	122.8	224.4	11.6	200.4	10.5	51.6	13.3	85.9
Lambda Repressor	1251	125.0	226.7	40.6	458.3	25.7	258.6	27.3	119.7

5.6.2 Validation

OpenMM LTMD was validated against OpenMM Langevin through simulations of Ala5[9], a small peptide of five alanines, and Villin NLE, a variant of the Villin headpiece that folds in 0.5 µs.

5.6.2.1 Dynamics and Sampling of the Small, Helical Peptide Ala5

Ala5 was simulated with OpenMM LTMD and Langevin to validate that LTMD properly reproduces dynamics and sampling. The Ala5 model was prepared with the Amber96 forcefield and the Generalized Born OBC implicit-solvent model. Eighteen simulations for both Langevin and LTMD, with total aggregate times of 5.4 µs and 4.8 µs each, were run from an extended confirmation at 300 ° K. The LTMD simulations used the following choice of parameters: 16 modes, 5 fs timestep, 625 fs rediagonalization period, 1 residue per block, 18 vectors per block, a block epsilon of 1×10^{-5} Å, and an s epsilon of 1×10^{-4} Å. The folded "helical" state for each alanine residue was defined by having ϕ and ψ angles such that $-180° \leq \phi \leq 0°$ and $-120° \leq \psi \leq 15°$.[9]

To compare sampling accuracy we measured the state distributions, where the states of the individual residues are deemed folded or unfolded by the criterion defined above. Ala5 has 32 states defined by the permutation of folded and unfolded states for each of its five residues. A comparison of the state distributions from the LTMD and Langevin simulations shows that LTMD samples the states with similar probability as Langevin, indicated by a correlation of 0.99 (Figure 5.7). Both methods show Ala5 spending most of its time in state 32 (all of the residues are folded), with the majority of the remaining time in state 16 (one of the end residues is unfolded). LTMD has a lower probability of being in state 16 and a higher probability of being in state 32.

To compare the dynamics we define the folded state of the peptide by all three inner alanine residues being folded, as defined above. We observed the time for each

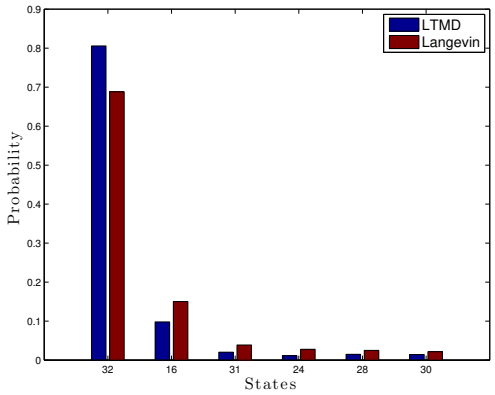


Figure 5.7. Populations of the 6 most-populated of the 32 defined states of Ala5 from Ala5 Amber96, GB-OBC implicit-solvent simulations run with LTMD (blue) and Langevin (red). All simulations started from an extended structure.

of the simulations to reach this folded state, from its initial extended state. The mean folding time for the Langevin simulations was 4.1 ns with a standard deviation of 4.6 ns. The LTMD Ala5 simulations produced mean folding times of 9.5 ns with a standard deviation of 6.9 ns, which is comparable to the Langevin simulations.

To measure the dynamics qualitatively we consider the RMSD, relative to the folded helical structure, during the folding process RMSD was computed for one representative Langevin simulation (black) and one representative LTMD simulation (blue) (Figure 5.9). Both simulations folded in 5.0 ns. The RMSD plots for both simulations show similar collapses at 5.0 ns as well as subsequent folding and unfolding.

5.6.2.2 Folding of Villin NLE

Villin NLE simulations were run with LTMD and OpenMM Langevin to validate the correctness of the LTMD implementation over longer timescales. The Villin NLE model was prepared with the Amber99-SB forcefield and the Generalized Born OBC implicit-solvent model. The dynamics of the simulations were analyzed with respect

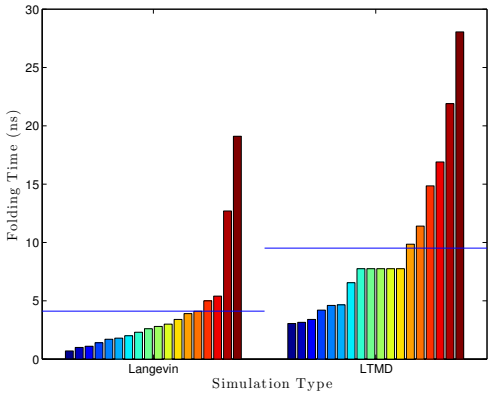


Figure 5.8. Folding times of Ala5 Amber96, GB-OBC implicit-solvent simulations run with Langevin and LTMD. The blue lines give the average folding times for each simulation method. Eighteen simulations of each type were run, and all simulations started from an extended structure.

to their ability to fold Villin NLE. It should be noted that our emphasis is purely on validating the correctness of the LTMD implementation, not on a large-scale analysis of Villin NLE dynamics.

Figure 5.10 gives the RMSD plots for two representative LTMD simulations run at 370 °K with a 50 fs timestep, 10 modes, and a rediagonalization period of 25 ps. The RMSDs were calculated against the native structure using the C_α atoms, excluding the first and last two residues. The first simulation (Figure 5.10a) folds Villin NLE to within 3.6 Å of the native structure, where it remains for around 1 µs. In the second simulation (Figure 5.10b), the protein undergoes multiple folding and unfolding events, occurring approximately every 0.5 µs, and is able to fold to within 3.5 Å of the native structure. While Langevin simulations fold Villin NLE to within 3 Å of the native structure, it should be noted that the higher RMSD of the LTMD simulations is an expected result given the coarse-graining used by the method to obtain better performance. As evidenced by these two simulations, LTMD is able to capture the dynamics of the proteins, while enabling a significant speed up over traditional MD.

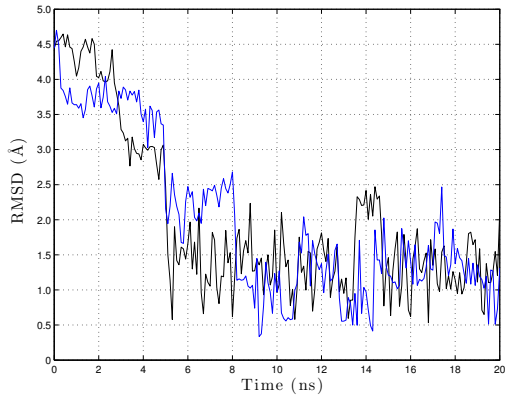
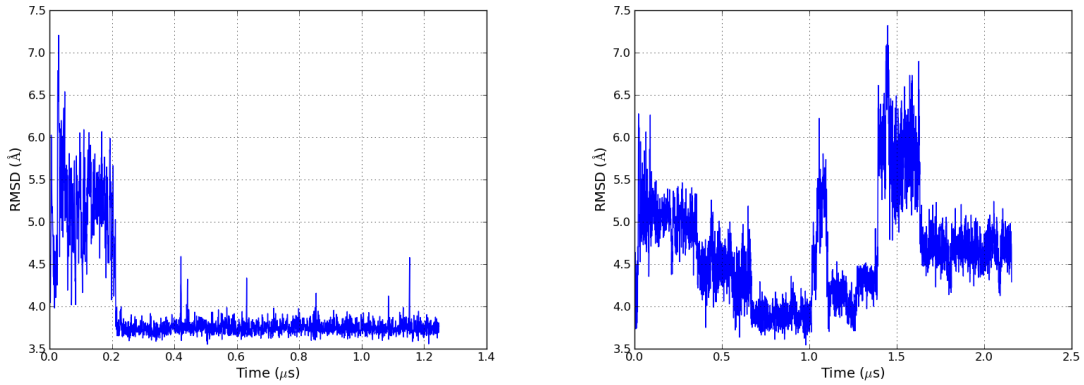


Figure 5.9. RMSD against the folded structure was computed for Langevin (black) and LTMD (blue).

5.6.3 Parameter Choices and Diagnostics

OpenMM LTMD uses multiple user supplied parameters in order to propagate and diagonalize the system of interest. In Section 5.6.3.1 the effect of varying the rate of rediagonalization on the dynamics is measured and in Section 5.6.3.2 the effects of the number of modes used on the dynamics is measured. We look at the effect of parameters on the accuracy of FBM in Sections 5.6.3.3 - 5.6.3.5. Section 5.6.3.3 compares the FBM implementation in OpenMM against the reference implementation in ProtoMol and the approximate eigenvectors with the full eigenvectors. In Section 5.6.3.4 the effect of the epsilons, ϵ, used as the perturbation for the numerical differentiation operations required within FBM. In Section 5.6.3.5 the effects of different partitioning schemes within FBM are compared. In Section 5.6.3.6 we consider the effects of the "noise" term in the Euler-Maruyama approximation in the fast space.



(a) (b)

Figure 5.10. RMSD plots for two Villin NLE Amber99-SB, GB-OBC implicit-solvent simulations run with LTMD. RMSD was calculated using C_α atoms, excluding the first two and last two residues, against the folded structure. The minimum RMSDs of each simulation are 3.6 Å and 3.5 Å, respectively. Figure 5.10b shows multiple folding and unfolding events, occurring approximately every 0.5 µs.

5.6.3.1 Rediagonalization Period

The theory behind LTMD assumes that the modes are always valid which, in theory, would require diagonalization at every step. In practice it has been observed that, since we only use a few low frequency modes, this requirement can be relaxed.

The effect of the rediagonalization period on the folding rate was measured by running Ala5 simulations with different rediagonalization periods (0.625 ps, 1.25 ps, and 2.5 ps) for 300 ns with parameters: 300 °K, 5 fs timestep, 12 modes, 1 residue per block, 14 vectors per block, and epsilons of 1×10^{-4} Å for the S matrix and 1×10^{-5} Å for the blocks. Rediagonalization period proved to affect the folding rate significantly. Simulations run with a rediagonalization period of 2.5 ps folded in an average of 62.5 ns, much larger than the average folding time of 3.7 ns found from Langevin simulations. Decreasing the rediagonalization period to 0.625 ps reduced

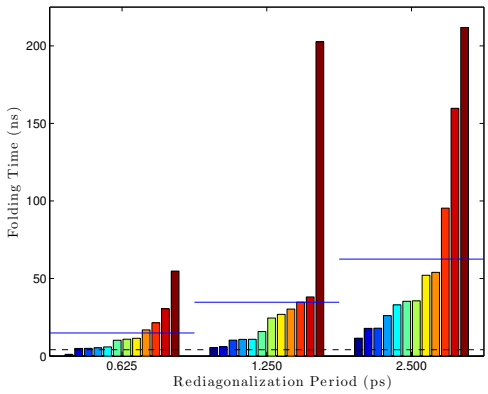


Figure 5.11. Rediagonalization period's effect on the folding time using 12 modes. Each bar is a single simulation, where all bars within a group all use the same parameters. The groups are sorted within themselves. The blue lines drawn over each set of bars are the average folding time for that set, while the dashed black line gives the "reference" folding time from Langevin simulations.

the average folding time to 14.3 ns.

This test was repeated, using 16 modes instead of 12 (which was found to be more accurate in Section 5.6.3.2). Here, we see that simulations run with a rediagonalization period of 2.5 ps had an average folding time of 20.94 ns. When the period is dropped to 1.25 ps or 0.625 ps, the average folding time drops to around 10 ns. This seems to indicate that picking the best number of modes to run, could allow a larger rediagonalization period before losing accuracy, allowing a trade-off to optimize performance.

5.6.3.2 Number of Modes

A sweep of the number of modes was performed. Eight Ala5 LTMD simulations were run for each mode setting (8, 10, 12, 14, and 16 modes) for 300 ns at 300 ° K, 5 fs timestep, 1 residue per block, and epsilons of 1×10^{-5} Å for the blocks and 1×10^{-4} Å for the S matrix. The number of vectors per block was set to be number of modes

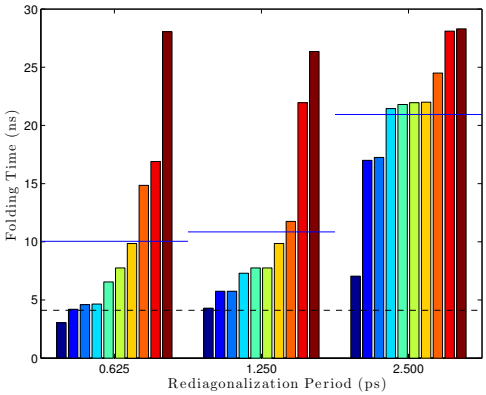


Figure 5.12. Rediagonalization period's effect on the folding time using 16 modes. Bars represent single simulations, where all bars within a group all use the same parameters. The groups are sorted within themselves. The blue lines drawn over each set of bars are the average folding time for that set, while the dashed black line gives the "reference" folding time from Langevin simulations.

plus two (10, 12, 14, 16, and 18 vectors per block, respectively). With 16 modes, the LTMD Ala5 simulations were able to consistently achieve folding times of 5.5 ns on average with a standard deviation of 2.6 ns, which compares favorably to the values obtained from Langevin simulations (average of 3.7 ns and standard deviation of 3.9 ns).

5.6.3.3 Approximate Eigenvector Overlap

The implementation of the Flexible Block Method (FBM) in LTMD was validated against the implementation in ProtoMol using a metric called "overlap." Overlap (as described in Tama, et al. [89]) measures how well a vector is represented by a space spanned by a set of vectors. The overlap P_j for a reference eigenvector u_j with the set of approximate eigenvectors $(v_1, v_2, \ldots v_m)$ is given by Eq. (5.45). An overlap value of 1 indicates that the reference eigenvector is represented completely by the approximate eigenvectors, while a value of 0 means that the reference eigenvectors is

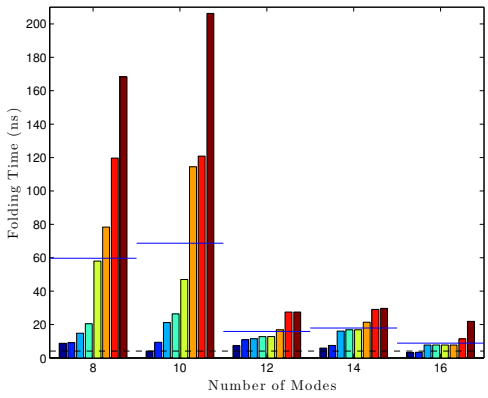


Figure 5.13. The effect on the folding time, caused by changing the number of modes. Each bar is a single simulation, where all bars within a group all use the same parameters. The groups are sorted within themselves. The blue lines drawn over each set of bars are the average folding time for that set, while the dashed black line gives the "reference" folding time from Langevin simulations.

not represented at all.

$$P_j = \sum_{i=1}^{m} (\mathbf{v_i} \cdot \mathbf{u_j})^2. \tag{5.45}$$

We used overlap to compare the approximate eigenvectors generated from both the reference CPU Flexible Block Method (FBM) implementation in ProtoMol[54] and our implementation in LTMD. Figure 5.14 shows the overlap for the approximate eigenvectors from LTMD with those produced by the FBM implemented in ProtoMol in WW Fip35. The two implementations show very good agreement for the first 20 modes. We also computed the overlap between the approximate eigenvectors from LTMD with the full eigenvectors. The approximate eigenvectors agree with the full eigenvectors for the first 15 modes but are less accurate for modes 16, 18, and 19. Since only ten or twelve modes are used in simulations, FBM is able to approximate the eigenvectors reasonably well for our purposes.

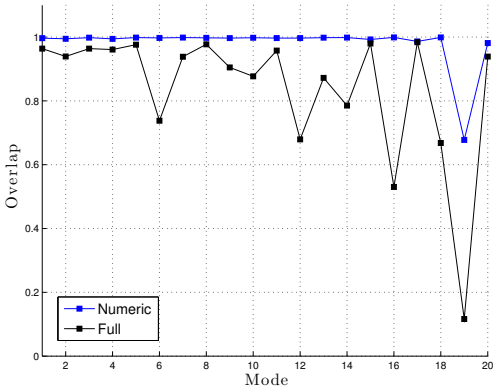


Figure 5.14. Overlap between the approximate eigenvectors for WW Fip35 for LTMD and ProtoMol implementations of FBM (blue) and the approximate eigenvectors and full eigenvectors (black).

5.6.3.4 Magnitude of Epsilon for Numerical Differentiation Perturbation

As detailed in the Section 5.4.2, LTMD uses the Flexible Block Method (FBM) to perform a fast normal mode analysis which utilizes numerical differentiation (ND). The choice of perturbation ϵ is usually made to be as small as possible so that accuracy is maintained, while being large enough to avoid round-off errors due to the finite precision of floating-point arithmetic. For single precision arithmetic, an optimal value for ϵ can generally be found as $\epsilon = \sqrt{2^{-23}} = 3.4 \times 10^{-4}$.[68] Note that OpenMM LTMD uses units of nm internally; for our configuration files this equates to $\epsilon = 3.4 \times 10^{-5}$ Å. For FBM there are additional considerations when finding the ϵ for the formation of the S matrix since it is important that our perturbation lies in the low frequency space of interest. Since, when projected from mode space our perturbation vector, δm, is given by $\delta m = \epsilon M^{-\frac{1}{2}} E_i$ for the i^{th} mode, all of the values in δm must not have significant round-off. Below, we document the tests carried out to determine the optimal values for the ϵ used in the block and S matrix ND and compare to theory.

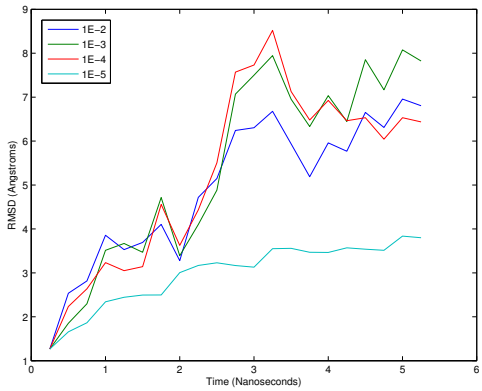


Figure 5.15. Comparison of the collapse of WW-Fip35 from the extended confirmation with different magnitudes for the numerical differentiation perturbations. RMSD was computed against the extended confirmation. The other simulations were run with LTMD using different values of ϵ. The same choice of ϵ was used for both the blocks and quadratic product.

From Langevin simulations, it is known that the extended confirmation of WW-Fip35 rapidly collapses. We have found that LTMD simulations of the collapse are particularly sensitive to the choice of ϵs and thus are an excellent test for validation.

To measure the effect of the choice of ϵ on the dynamics, WW-Fip35 was simulated from an extended confirmation with a range of values for ϵ. The WW-Fip35 model was prepared from the extended confirmation using the Amber96 forcefield and Generalized Born OBC implicit-solvent model.

LTMD simulations were run with a 50 fs timestep, 10 modes, a 100 ps rediagonalization period, and a range of values for ϵ. RMSD to the extended structure was calculated between the C_α atoms and plotted as a function of time in Figure 5.15. Simulations run with values between 1×10^{-2} Å and 1×10^{-4} Å show similar rates of collapse, while the simulation with $\epsilon = 1 \times 10^{-5}$ Å shows that the dynamics are damped and the protein is prevented from collapsing.

To further explore the effects of the S matrix epsilon on the simulation folding time, a sweep of simulations were run with varying S matrix epsilons using Ala5.

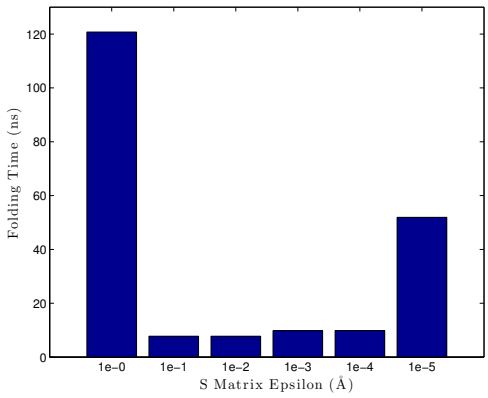


Figure 5.16. A sweep of simulations executed with varying S matrix epsilons, with the Ala5 model. Each bar is a single simulation, where all bars within a group all use the same parameters. The groups are sorted within themselves. The blue lines drawn over each set of bars are the average folding time for that set, while the dashed black line gives the "reference" folding time from Langevin simulations.

In Figure 5.16, we can see that this epsilon value should not be a critical factor in simulation accuracy (at least for the Ala5 model), beyond being within the right range.

5.6.3.5 Effect of Partitioning Method

The Flexible Block Method (FBM) requires choosing a method for partitioning the atoms into blocks[27–30, 89]. The simulations in this Chapter, group entire residues into blocks using a uniform number of residues per block (the last block may contain fewer if the number of residues is not an integer multiple of the number of residues per block). To look at the effect of the partitioning, an alternative scheme was devised where the partitioning occurs after the C_α atoms along the backbone. The atoms in the blocks for both partitioning methods represent a sequential set.

The two schemes were compared with respect to the overlaps of the first 20 approximate eigenvectors against the first 20 real eigenvectors (Figure 5.17) and folding

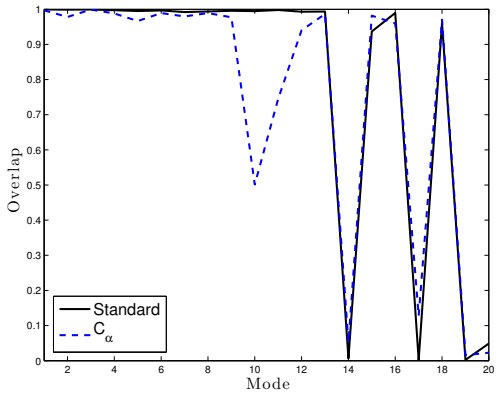


Figure 5.17. Overlap for Residue (standard) and C_α partitioning schemes for Ala5 compared with modes from a "brute force" diagonalization.

time of Ala5 LTMD simulations. The eigenvectors were calculated using 1 residue per block, 18 vectors per block, and epsilons of 1×10^{-5} Å for the blocks and 1×10^{-4} Å for the S matrix. The standard partitioning scheme reproduces the first 11 modes very well but is not able to reproduce modes 14, 17, or 19. The C_α partitioning scheme has similar accuracy for all modes except for modes 10-12, which it is not able to reproduce well. Reflecting the difference in the accuracy of the modes with the two partitioning schemes, a LTMD simulation with the standard partitioning scheme folded Ala5 in 24.8 ns, more quickly than a simulation with the C_α partitioning scheme at 33.7 ns.

The two schemes were also tested on WW Fip35 (Figure 5.18). The accuracy of the two schemes is similar, suggesting that the two schemes do not make a significant difference in the accuracy of the modes.

5.6.3.6 Magnitude of Fast Noise

As an approximation to Euler-Murayama method, after performing minimization in the fast space LTMD adds random noise Eq. (5.34). Ala5, with different

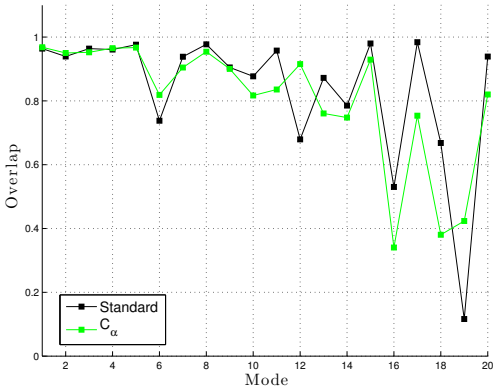


Figure 5.18. Overlap for Residue (standard) and C_α partitioning schemes for WW Fip35 compared with modes from a "brute force" diagonalization.

amounts of fast noise, was simulated to measure the effect of the noise on dynamics (Figure 5.19). The simulations were run for 300 ns with the following parameters: 300 ° K, 5 fs timestep, 16 modes, a rediagonalization period of 0.625 ps, 1 residue per block, 18 vectors per block, and epsilons of 1×10^{-5} Å for the blocks and 1×10^{-4} Å for the S matrix. The figure shows that the noise factor, for this model, did not have a great effect. However, from these results, a noise scaling value of 1.0 allows for the closest average folding time compared to the Langevin tests.

5.7 Conclusion

We presented a hybrid CPU/GPU implementation of the LTMD propagator with speed ups of over 5.8× compared with traditional MD integrators on GPUs. This result illustrates great potential for testing larger protein systems over a longer biological period of time. Analysis of the cost of individual sections of the method have yielded insight into how we may improve performance in the future.

Validation with Ala5 and Villin NLE has shown excellent agreement between Langevin and LTMD. With Ala5, we showed that LTMD is able to sample the confor-

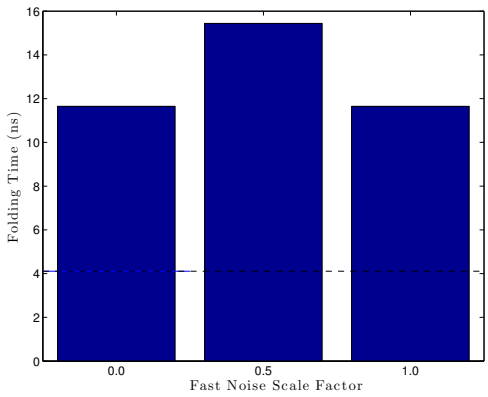


Figure 5.19. Folding times of LTMD Ala5 simulations run with different values for the amount of noise added in the fast space. Each bar is a single simulation, where all bars within a group all use the same parameters. The groups are sorted within themselves. The blue lines drawn over each set of bars are the average folding time for that set, while the dashed black line gives the "reference" folding time from Langevin simulations.

mational states with a similar distribution as Langevin, produces comparable folding times, and captures similar dynamics. Similar folding times and dynamics between Langevin and LTMD were also reported for Villin NLE.

Analyses of the effects of the various parameters such as rediagonalization period, the number of modes, epsilons used with the numerical differentiation, partitioning methods, and fast noise were presented. We found that with the proper parameters, especially the rediagonalization period and number of modes, LTMD agrees well with Langevin. Further, by choosing a larger number of modes, rediagonalization can be performed less frequently, leading to increased performance. The choice of block epsilons for FBM were shown to agree with the known values from the theory, while the choice of S epsilons were shown to be robust over a range of values.

This implementation of LTMD, and future improvements to the performance and numerics promise an order of magnitude improvement over conventional GPU implementations of MD.

CHAPTER 6

ACCELERATING A MIXED EULERIAN-LAGRANGIAN PARTICLE-LADEN FLOW SIMULATION USING A HYBRID CPU-GPU SYSTEM

6.1 Introduction

Computational fluid dynamics is typically carried out by designing a parallel program that is decomposed in the spatial domain, such that each processor operates on a portion of the fluid flow, and periodically exchanges a halo of states with its neighbors. This long-used method is highly effective at Eulerian computational methods, such that the partitioning is static relative to Cartesian dimensions. An example of this is the NCAR-LES simulation code [55], which has been ported to multiple HPC architectures and used to simulate atmospheric turbulence in many different applications.

Recently, this code has been augmented by adding spray particles superimposed over the turbulent flow. This collection of particles represents small droplets that are carried by the flow, such as might be seen at the crest of an ocean wave. The most direct way to add particles to the existing flow simulation code is to again perform a spatial partitioning and then alternate the particle and flow calculation at each node, exchanging states as needed between each time step. However, due to the nature of the particle step, the time to compute the particle step quickly exceeds the flow step as the number of particles is scaled up.

To address this problem, we reconfigured the application to perform the (Eulerian) flow calculation on distributed CPUs in the usual way, but moved the particle spray

to a GPU, which is better suited to support the (Lagrangian) interactions on a per-particle basis. We demonstrate that this technique achieves a speedup of 14.4× compared to 64 CPUs with MPI. A single GPU is able to outperform the distributed CPUs on systems of up to 240 million particles in the available device. Looking forward, we consider how this hybrid simulation technique can be scaled up to even larger systems making use of larger GPU memories, multiple centralized GPUs, and multiple GPUs distributed among the CPUs.

6.2 Background

In a wide variety of engineering and scientific applications, great need exists for simultaneously computing Eulerian and Lagrangian quantities, since each brings a different set of numerical advantages. Such a problem often takes the following form: a set of partial differential equations governing the spatial variation of some continuous quantity is discretized, parallelized, and solved on a fixed Eulerian computational mesh, while at the same time Lagrangian elements, each carrying its own information, travel throughout the domain independent of the mesh but are modified based on the computed Eulerian fields. From an algorithmic perspective, the underlying code is therefore split between two main jobs at each time step: solving for the Eulerian field and advancing the Lagrangian elements. Parallelization is typically done by decomposing the computational domain across tasks, transferring halo data before each time step for the Eulerian portion and transferring particle data across task boundaries for the Lagrangian portion.

A specific example motivating the present work is that of particle-turbulence interaction, a common system encountered in science and engineering where a turbulent flow field transports some dispersed phase (dust grains, spray droplets, etc.) throughout a domain of interest [7, 70]. In the physical sciences, these particles may be for example cloud droplets [34] or sea spray droplets [69] which are carried through-

out atmospheric turbulence, giving rise to a systematic and controlled way to study processes such as rain formation, air-sea energy exchange, atmospheric visibility, etc.

For the particular application of interest in this work, the incompressible Navier-Stokes equations are solved for the velocity, temperature, humidity, and pressure fields on a fixed computational Eulerian mesh:

$$\nabla \cdot \vec{u} = 0, \tag{6.1}$$

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = -\frac{1}{\rho} \nabla p + \nu \nabla^2 \vec{u}, \tag{6.2}$$

$$\frac{\partial T}{\partial t} + \vec{u} \cdot \nabla T = \alpha \nabla^2 T \tag{6.3}$$

$$\frac{\partial q}{\partial t} + \vec{u} \cdot \nabla q = \Gamma \nabla^2 q \tag{6.4}$$

where $\vec{u}$ is the fluid velocity, p is the pressure, T is the fluid temperature, q is the specific humidity of the fluid, ρ is the density, ν is the kinematic viscosity, α is the thermal diffusivity, and Γ is the diffusivity of water vapor.

At the same time, individual Lagrangian particles are inserted into the flow, which are each transported according to momentum, energy, and mass conservation as they change velocity, temperature, and radius:

$$\frac{d\vec{x}_p}{dt} = \vec{v}_p, \tag{6.5}$$

$$\frac{d\vec{v}_p}{dt} = \frac{1}{\tau_p} \left(\vec{u}_f - \vec{v}_p \right). \tag{6.6}$$

$$\frac{dT_p}{dt} = \dot{Q}_{conv} + \dot{Q}_{evap} \tag{6.7}$$

$$\frac{dr_p}{dt} = \dot{m} \tag{6.8}$$

Here, $\vec{v}_p$ refers to the velocity of a single particle, τ_p is a material time constant representing the inertia of the particle, and $\vec{u}_f$ is the surrounding flow velocity, interpolated to the location of the particle. It is this interpolation procedure that links the Eulerian solution of the velocity field $\vec{u}$ to the Lagrangian particle dynamics. T_p refers to the particle temperature, which can change based on convective heat transfer $\dot{Q}_{conv}$ or evaporative (i.e. latent) heat transfer $\dot{Q}_{evap}$. r_p is the particle radius which can change based on the evaporation rate $\dot{m}$. The radius and temperature are likewise linked to the Eulerian field via interpolation of the surrounding fluid temperature and humidity (contained in $\dot{Q}$ and $\dot{m}$).

Thus overall, the particle position $\vec{x}_p$ changes according to $\vec{v}_p$, which itself responds to the local flow velocity $\vec{u}_f$ according to Equation 6.6. At the same time the particle temperature and radius can change based on local thermodynamic properties interpolated to the particle location.

Numerically, Equations 6.1 — 6.4 are solved using a pseudospectral discretization in the periodic x and y directions (i.e., derivatives are based on fast Fourier transformations), and second-order finite differences are used in the z direction. Time integration is performed using a third-order Runge-Kutta scheme [82] for all flow and particle variables. A sample snapshot of the particle/flow solution is provided in Figure 6.1.

The underlying code is written in Fortran and based on MPI for parallelization, and has been used in the past to study a wide variety of turbulent flows in the atmosphere and ocean [23, 55, 69, 86]. It has been scaled up to 16,384 processors on

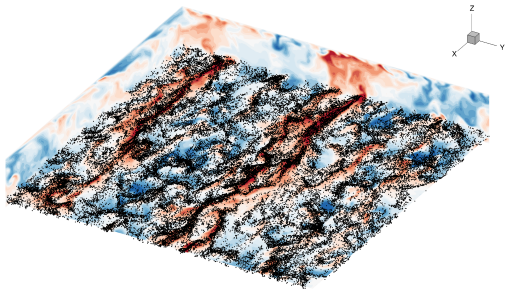


Figure 6.1. Snapshot of the particle-turbulence simulation computed via Equations 6.1 - 6.8. Black dots are the instantaneous locations of the particles (shown only in the bottom half of the domain), and colors represent velocity fluctuations.

multiple architectures including Cray, SGI, and IBM (see e.g. reference [85]), and routinely runs on computer clusters housed at various locations, including the National Center for Atmospheric Research (NCAR), the U.S. Army Engineer Research and Development Center (ERDC), and the National Energy Research Scientific Computing Center (NERSC).

In its original configuration, the flow and particle solutions are computed using MPI over multiple CPUs. The flow computation decomposes the Cartesian domain over a two-dimensional array of processors as shown in Figure 6.2(a). Each processor contains the entire solution array in the x direction at each (y, z) point for ease in performing fast Fourier transforms (FFTs), while MPI communication is required to perform FFTs in the y direction. Each processor constructs a halo in the z direction for computing the approximations to vertical derivatives.

For this particular application, the dynamics of the particles is such that they tend to drift and accumulate near the top and bottom walls of the domain [72], and therefore for purposes of load balancing, the particles must reside on a slightly different processor decomposition. This is shown schematically in Figure 6.2(b). Since Equation 6.6 depends on the local interpolated fluid velocity $\vec{u}_f$, the processor

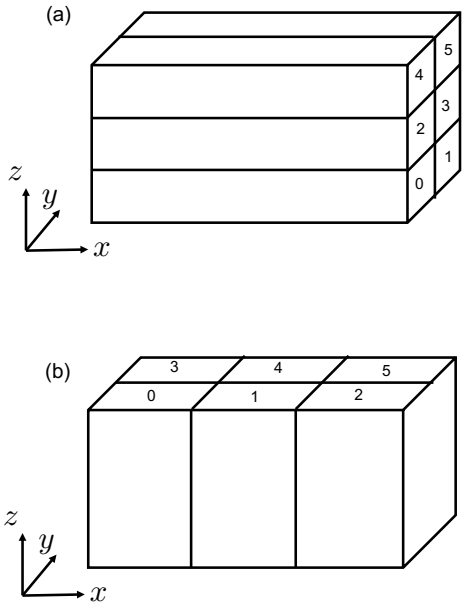


Figure 6.2. (a) Schematic of the MPI domain decomposition for the Eulerian flow calculations. Example here shown for 6 MPI processes. (b) Schematic of the MPI domain decomposition for the Lagrangian particles, shown for the same 6 MPI processes

domains shown in Figure 6.2(b) must acquire Eulerian flow information from the processor domains of Figure 6.2(a). For instance in the setup described in Figure 6.2, processor 0 must communicate with processors 2 and 4 in order to construct a "transposed" velocity (and temperature and humidity) field from which to interpolate the to the locations of the particles which live on processor 0. This nearly all-to-all communication step is one of the most expensive of the Lagrangian particle solver.

Furthermore, as the particles are transported throughout the domain, they cross processor boundaries and must be communicated via MPI. Since the particles housed on each processor are stored as a linked list, all particles crossing to each neighboring processor are collected and transferred using an MPI derived data type.

For many applications in the context of particle-turbulence interaction, the highest possible number of particles is desired for statistical convergence purposes; how-

ever, in the original MPI implementation, a typical simulation with an Eulerian grid size of $[N_x, N_y, N_z] = [128, 128, 128]$ run on a cluster of 64 processors becomes dominated by particle computations after roughly 10^6 particles are introduced. This limitation prevents the simulation of problems where particle numbers exceed 10^6. Since the particles are independent of each other, and since they are not confined to a computational mesh, this problem is ripe for GPU-based acceleration since it will remove the need for transferring flow information according to Figure 6.2 as well as exchanging particle information between processors. This capability is intended to allow for simulating significantly higher numbers of particles on the same Eulerian grid, but without prohibitive increases in computational cost.

6.3 The Scalability Problem

The challenge, then, is to simulate tens to hundreds of millions of particles, to model the phenomenon accurately, and to improve the statistical accuracy, while running on easily available resources. In this instance, the available Infiniband linked cluster consists of 16 compute nodes each with 16 Intel Xeon E5-2650 cores. This gives a total compute resource of 2.8 TFLOPS, with each processor providing 0.179Tf [37]. However, this resource is shared between a number of researchers causing most jobs to be limited out of practicality to 64 cores or 0.72 TFLOPS. This typically leads to a limitation in simulation parameters to 10^6 particles, with a practical upper bound of 10^7 particles for the wall clock time deemed appropriate. For reference, figure 6.3 demonstrates that the particle calculation dominates as the number of particles increases. In this example for 64 MPI processes, the flow/particle computation time crosses over at 2×10^6 particles.

Even a modest upgrade to double the available resource compute power would be prohibitively expensive and require additional rack space. An attractive alternative is to add GPU resources to the current cluster; a Titan X Pascal provides 11 TFLOPS

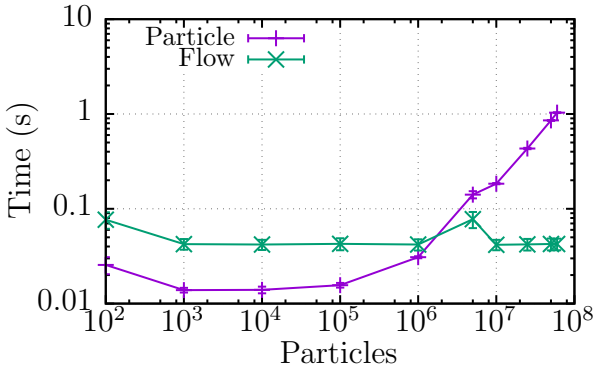


Figure 6.3. Comparison of the time taken for the flow vs particle calculation on 64 MPI processes. The particle calculation begins to dominate after 2×10^6 particles.

of compute resource, four times that available from the entire current cluster. Costs for adding a single Titan X card are of the order of $2000, well within available budgets.

Extracting optimal performance from GPUs is a challenging exercise, however, especially where the algorithm is a hybrid CPU/GPU code and data transfer must take place at every step. However the potential rewards make the exercise worthwhile even if only part of the potential can be harnessed.

As discussed previously, the underlying flow solver is a well-established code base that would be impractical to convert within the scope of this project. In contrast, the particle code is naively parallel, so ideal for GPU architecture exploitation. This choice also has advantages since the particle calculation is essentially a black box where the flow field is passed in and statistics are returned back out. From this perspective, porting the particle solver to the GPU is deemed worthwhile if it outweighs the cost of sending the data to the device.

6.4 Technical Challenges

Any such radical change in software architecture will bring interesting challenges, and these are briefly outlined in this section. Throughout, testability was built into the code from the beginning to facilitate regression testing as the code was moved from Fortran, to C/C++, and then to CUDA.

6.4.1 GPU Architecture

Two possible GPU/cluster architectures were considered; one or more GPUs per node, or a single master GPU node.

The traditional architecture for GPU clusters would have one or more GPUs per node. For our specific example, although still economic compared to extending the cluster, this was not practical as the current nodes could not accommodate the GPU cards or their power requirements. For our specific requirements, an additional node was purchased that could accommodate one or more Titan X Pascal GPUs and attached to the cluster via an Infiniband connection.

6.4.2 Data Transfer

By choosing a single GPU master process, we introduce a fixed cost to the simulation which is only affected by the flow grid size. Figure 6.4 shows that this fixed cost scales cubically with the grid size as expected. The cost is dominated by the MPI transfer as seen in Table 6.1.

6.4.3 Array Access

One of the biggest challenges was that the software is Fortran based, and the requirement for the GPU code to be in C++ with CUDA. This was chosen for the availability of tools as currently CUDA Fortran is only available via the PGI

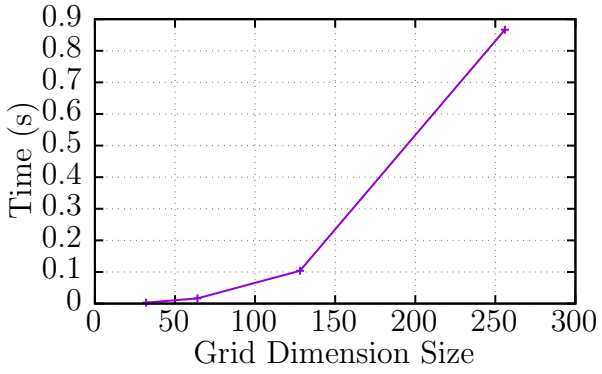


Figure 6.4. Time taken for different Grid sizes (32^3, 64^3, 128^3, 256^3). The time taken to transfer the grid to the GPU scales cubically with the grid size.

TABLE 6.1

COMPARISON OF COST BREAKDOWN FOR DIFFERENT GRID SIZES

Grid size	Flow Transfer (s)	Halo Generation (s)	GPU Upload (s)
32	0.00160	0.00018	0.00039
64	0.00710	0.00100	0.00160
128	0.03570	0.00810	0.01180
256	0.36900	0.04590	0.08750

compiler. CUDA itself was chosen over OpenCL due to the experience of the team, although transitioning it to provide an OpenCL implementation is made easier due to the work done for this project. To reduce data duplication between Fortran and C++, we pass the three dimensional velocity arrays as a linear array to the GPU. This produced its own problem due to the fact that Fortran and C use a different

memory layout for arrays, Column-Major for Fortran and Row-Major for C. This meant that the standard equation to calculate a three dimensional index into a linear array Equation 6.9 is incorrect i.e.

$$\begin{aligned} array[x][y][z] &\neq linear[a], \\ \text{where } a &= (x \times size(z) \times size(y) + \\ &\quad y \times size(z) + z) \end{aligned} \tag{6.9}$$

To correct this, we must rearrange the order of the operations to give us the correct equation as shown in Equation 6.10.

$$\begin{aligned} array[x][y][z] &= linear[b], \\ \text{where } b &= (x + y \times size(x) + \\ &\quad z \times size(x) \times size(y)). \end{aligned} \tag{6.10}$$

While this gives the correct access, Fortran also provides the ability to have offset indices for its arrays. To handle this, the minimum bound for the array dimensions in Fortran need to be added to each of the indexes as shown in Equation 6.11.

$$\begin{aligned} array[x][y][z] &= linear[c], \\ \text{where } c &= ((x + lbound(x)) + \\ &\quad (y + lbound(y)) \times size(x) + \\ &\quad (z + lbound(z)) \times size(x) \times size(y)). \end{aligned} \tag{6.11}$$

6.4.4 Batch System Support

Owners of large computational clusters utilize scheduling systems to allow as efficient usage of the owned resources as possible. However, these systems are still limited when working with GPU machines.

Most scheduling systems lack the ability to request a set of machines with varying requirements (in our case the request includes a single machine that has a GPU and any set of machines which have 16 cores.) Solving this required specifying a list of hard coded machine names that the software should run on, crippling the scheduler's ability to distribute the usage of resources. This limitation is the target of future system-level optimization and configuration.

6.4.5 Testing

While testing and debugging C++ or Fortran code is generally straightforward thanks to a large number of available tools, NVIDIA's CUDA only provides a small number of such features. The pre-processing system available in Fortran or C/C++ compilers allows the CUDA code to be wrapped with definitions so that it can be compiled as C/C++ code to be executed on the host instead of the GPU. This setup allows the implementation of a regression and unit test framework which also allows utilization of the tools NVIDIA provided as efficiently as possible.

6.5 Results

The presented results compare the CPU particle computation versus the GPU particle computation, since this calculation dominates the computation time after 2 million particles on the original full CPU configuration (see Figure 6.3). Since the architecture of the two solutions is different there are a number of important factors governing the relative CPU/GPU efficiency.

The original CPU implementation of the particle calculation is described in Section 6.2, and is distributed across the MPI nodes. During each time step (whose size is calculated based on the flow computation), each particle update follows the flow update, and after each particle update, particles are exchanged between MPI nodes based on their new locations. This results in two distinct timing events: CPU particle calculation and MPI particle position transfer.

The GPU algorithm updates the particles after each flow calculation step, and since the chosen architecture is a multi GPU-enhanced CPU node, the complete flow field now needs to be transferred to the GPU node. Again, we have two sources of computational cost; GPU particle calculation, and MPI flow transfer.

We note that the MPI flow calculations are effectively gated by a single step of the particle calculation in both the CPU/GPU implementations.

In the following results we measure the combined cost of both the calculation, and data transfer time. Figure 6.5 shows that until 1.0×10^6 particles, the CPU performs better than the GPU, primarily due to the transfer cost of sending the data to the GPU.

Due to the fact that it scales better than the CPU, the GPU performance overtakes 16 cores at $\approx 1.0 \times 10^6$, 32 cores at $\approx 5.0 \times 10^6$ and 64 cores at $\approx 2.5 \times 10^7$. At its most optimal, four GPUs are 14.4× faster than 64 cores, 29.2× faster than 32 cores and 35.4× faster than 16 cores. We note that we are comparing a fixed resource with GPUs compared to scaling in number of CPU cores, so clearly at some point the greater number of CPUs will overtake the GPU performance. However we show a good performance increase at our target sub-cluster of 64 cores. In addition, we discuss scalability as we add additional GPUs in the Section 6.5.1.4.

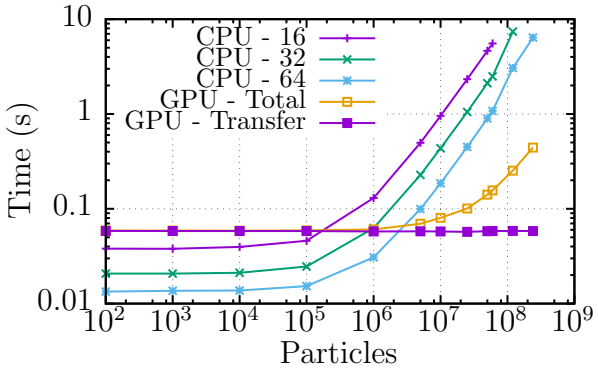


Figure 6.5. Performance scaling with varying MPI tasks on CPU vs four GPUs. The GPU has a constant transfer time but scales better than the CPU, providing a 14.4× performance improvement at 2.4×10^8 particles.

6.5.1 Optimization

With an initial reference GPU implementation, common optimization methods were explored to further improve the performance. Reducing the algorithmic complexity of the interpolation provided a large performance improvement whilst still keeping the statistical accuracy of the simulations. Shared memory, which is a common GPU optimization, provided no performance benefits for this software. Finally, restricting pointers to avoid aliasing, allowing the compiler to produce more optimized code, provided a large performance improvement for the original interpolation method, but no benefit for the reduced complexity version.

6.5.1.1 Interpolation

In the original CPU version of the code, two interpolation schemes are available for calculating flow properties at the location of each particle: sixth-order Lagrange interpolation and a second-order trilinear interpolation. The former requires a $6 \times 6 \times 6$ grid stencil surrounding each particle, which proved to be a large portion of the computation on the GPU due to the large amount of branching and random memory

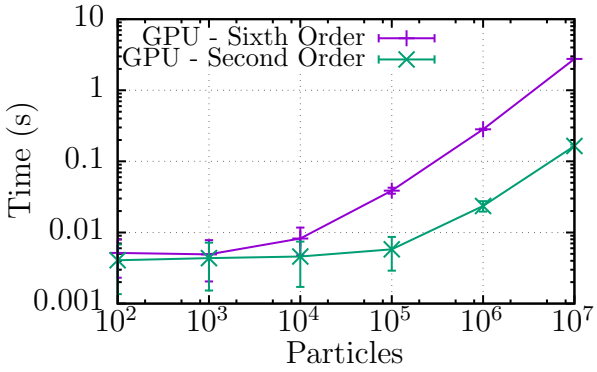


Figure 6.6. Comparison of sixth and second order interpolation on a single GPU. This shows that switching from sixth order to second order interpolation provided a 10× performance improvement for that GPU kernel.

access, both of which GPUs handle poorly.

By implementing the second order interpolation, which required less branching and random memory access, the performance was improved greatly. Figure 6.6 shows that switching from sixth order to second order interpolation provided a 10× performance improvement in that GPU kernel.

The reduced complexity second order method is proved to be statistically accurate enough to match the sixth order method (shown in Section 6.6).

6.5.1.2 Shared Memory

Shared memory is a common optimization within GPU code to improve the memory bandwidth. This is recommended because local variables within the kernel are either stored in registers (fastest access) or a private section of the GPUs global memory (slowest access).

Figure 6.7 shows that we receive no benefit for sixth order interpolation and only 10% improvement for second order interpolation. This is likely due to the fact that these kernels use double precision exclusively, and the performance benefits of the

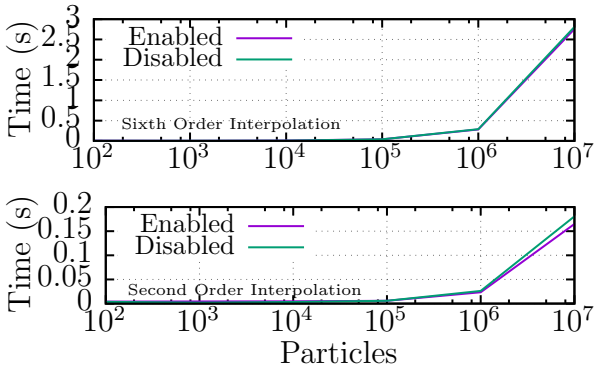


Figure 6.7. Performance improvement by using shared memory on a single GPU. Shows that the implementation does not benefit greatly from using shared memory.

shared memory are shadowed by the cost of loading them into the shared memory instead of relying on the L2 cache.

6.5.1.3 Restricted Pointers

In C/C++ it is possible for a function to take multiple pointers that point to the same memory location (aliasing). This aliasing requires extra safeguards to make sure that the program performs correctly. If two writes happen after each other, they must be preserved in the final machine code, whereas two reads can occur in either order without causing any problems. These things impose restrictions on the compiler's ability to produce the most optimal code.

Most compilers will allow the usage of the "restrict" keyword, which guarantees that the pointers do not alias each other allowing the compiler to arrange instructions in the most optimal way.

In CUDA, if parameters are defined as constant and restricted then the compiler allows the usage of a read only cached memory. Due to the fact that most of the work in our software relies on accessing the flow fields which are constant within each step, we are able to utilize this to our benefit.

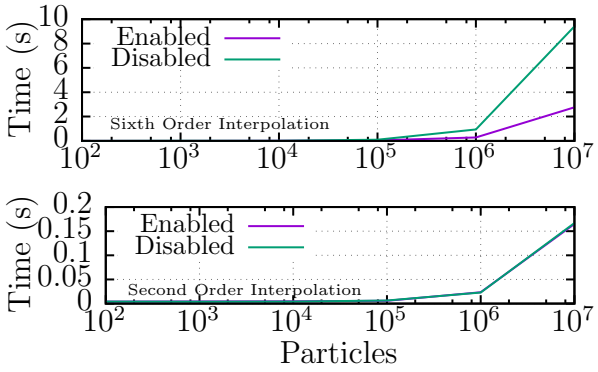


Figure 6.8. Performance improvement by using restricted pointers on a single GPU. Shows that the implementation provides a large benefit for the sixth order interpolation but not the second order interpolation

Figure 6.8 shows that a 5× performance improvement was achieved for the sixth order interpolation only. This is likely due to the fact that the sixth order interpolation access much more data and it accesses it in a more random pattern than the second order interpolation and so benefits more from the read only cache that this improvement affords.

6.5.1.4 Multiple GPUs

On a NVIDIA GTX Titan X card which has 12GB RAM, 60 million particles and the flow fields can be fit into the GPU's memory. We expanded the single GPU code to work with multiple GPUs on the same machine.

By scaling up our system to contain four GTX Titan X GPUs, it allowed for the simulation of up to 240 million particles at the cost of transferring the field to each GPU. Figure 6.9 shows how the performance scales for four GPUs with a varying number of particles. It shows that for 240 million particles, dividing the work over four GPUs provides a 6× performance improvement over 64 CPU cores.

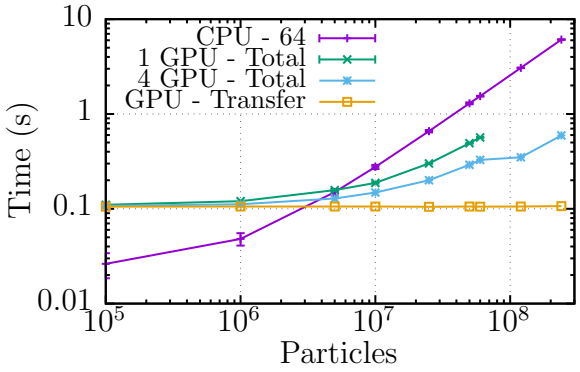


Figure 6.9. Particle update timings for four GPUs showing a 6× improvement with four GPUs compared to 64 CPU cores.

6.5.2 Single Precision Transfer

A major cost with the initial implementation was transferring the double precision fields from the nodes to the GPU master, by reducing the field to single precision before sending we reduce the time taken for this transfer by 60%.

Figure 6.10 shows the performance improvement using a single precision flow field. Reducing the transfer time by half moves the point where the GPU becomes faster than the CPU from ≈ 5 million particles to ≈ 1.25 million.

6.6 Validation

To validate the GPU implementation, we turn to the benchmark case of reference [52], who gathered results from several international research groups in order to provide a test bed of multiple flow/particle codes, all simulating the same particle-laden turbulent flow. Turbulent channel flow, solved via direct numerical simulation (i.e. the Eulerian computation), laden with one-way coupled particles (i.e. the Lagrangian computation) is computed, and statistics of the flow and particle velocities are provided from each of the groups. For purposes of the present work, we use this test case as a benchmark to ensure that the GPU calculations produce results which match

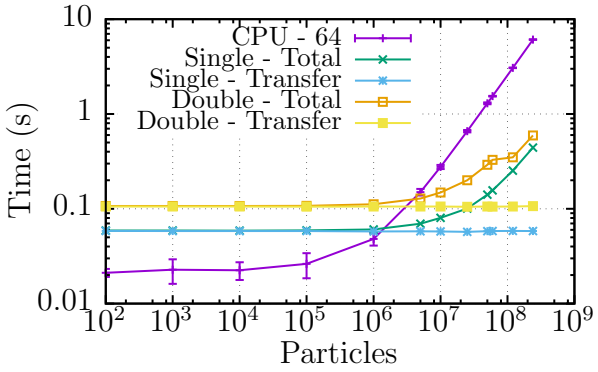


Figure 6.10. Performance comparison single precision and double precision flow field transfer transfer yielding a 30% reduction in overall particle update time for 240 million particles using four GPUs.

both the CPU calculation as well as published benchmark data.

As as a representative example, we plot in Figures 6.11 and 6.12 the mean streamwise particle velocity and the root-mean-square streamwise particle velocity fluctuation for one of the specified particle sizes (case $St = 5$ in [52]). In the figures, *UUD*, *TUE*, *ASU*, and *HPU* refer to four groups solving the same flow with slightly varying numerical discretizations. From our simulations, we present three curves: one based on the GPU using 10^7 particles with linear interpolation for the flow velocity at the particle location, one using the CPU and linear interpolation (10^5 particles), and one using the CPU and sixth-order interpolation (also 10^5 particles). These figures demonstrate that the GPU recovers the exact same result as the CPU for the same interpolation type, and that the speedup gained by using linear interpolation on the GPU does not significantly affect these statistics, making it a viable cost for the enhanced speedup.

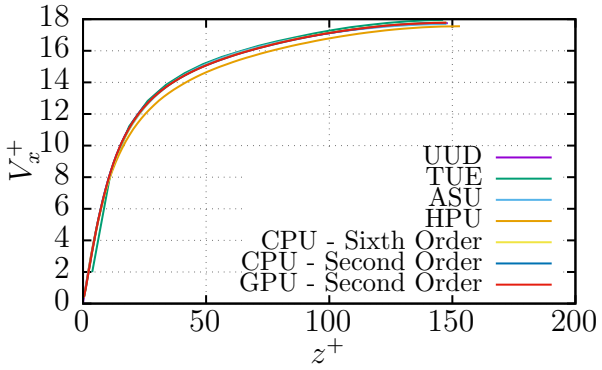


Figure 6.11. The horizontally averaged streamwise particle velocity (V_x^+) as a function of wall-normal distance (z^+) computed by the GPU, the CPU, and the CPU using sixth-order interpolation compared against research groups *UUD*, *TUE*, *ASU*, and *HPU* in the benchmark case of [52] for $St = 5$.

6.7 Extrapolation

With the current work optimized and validated, we look to the future at areas that we feel would provide the largest benefit.

The current code utilizes double precision exclusively to guarantee the results are as accurate as possible. However, this provides a major hit to performance on GPUs (which are designed to work efficiently on single precision floating point numbers), ranging from a 2× slowdown with NVIDIA's Tesla cards to a 32× slowdown for the GPUs we used in this project. Again, we provide an approximation of the maximum possible performance benefits gained by moving the code completely to single precision.

6.7.1 Single Precision Calculation

As this implementation was focused on moving the current code to use the GPU, all of the calculation on the GPU were kept in double precision to provide a consistent system. For GPUs, double precision computation is much more costly than single

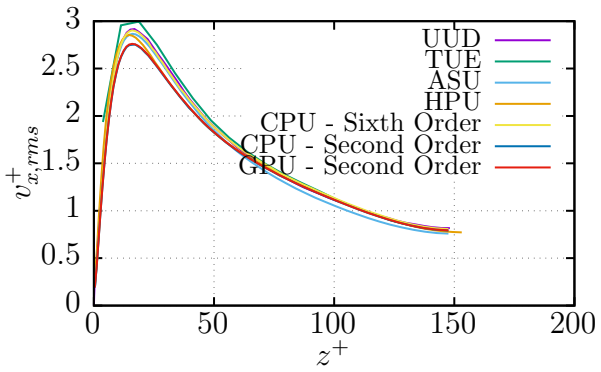


Figure 6.12. The root-mean-square streamwise particle velocity fluctuation ($v^+_{x,rms}$) as a function of wall-normal distance (z^+) computed by the GPU, the CPU, and the CPU using sixth-order interpolation compared against research groups *UUD*, *TUE*, *ASU*, and *HPU* in the benchmark case of [52] for $St = 5$.

precision (from 2× slower for Server cards, to 32× slower for the Titan X cards that were used.)

Figure 6.13 shows an extrapolation of the potential performance with single precision. For a Tesla GPU, the performance compared to the CPU improves to provide a 30× speedup. For the GTX Titan X, a 75× performance improvement compared to the CPU could be possible.

Finally, by transitioning the particle data and flow field from double to single precision, the number of particles that are able to fit into the GPU's memory should double to 125 million.

6.8 Conclusion

This work shows that by migrating the particle calculation from MPI to a four GPU system, we are able to calculate the trajectories for 240 million particles with a 14.4× speed up compared to the particle calculation on our target cluster of 64 cores. This allows for simulations to greatly enhance the statistical accuracy of the

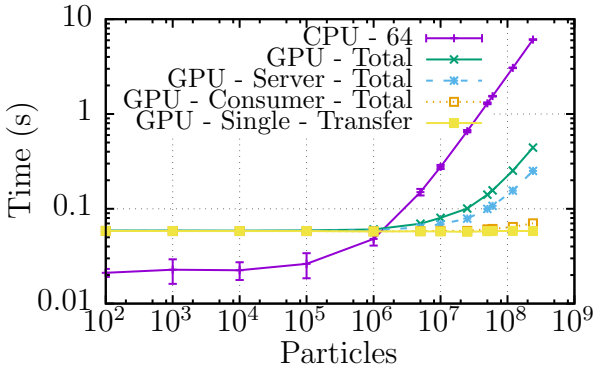


Figure 6.13. Single precision extrapolated particle update on consumer and enterprise GPUs. This shows that we expect the consumer card to achieve a 75× speedup over the CPU. It also shows that for a enterprise GPU we expect to achieve a 30× speedup over the CPU.

particles within a flow field.

Overall, with GPU acceleration of the flow/particle solver, system sizes (i.e. particle numbers) and flow regimes can now be reached on traditional compute clusters that were previously unobtainable. Furthermore, this demonstration highlights the unique ability of GPU devices to provide cost-efficient speedup over mere extension of CPU clusters, even when favorable scaling exists.

CHAPTER 7

CONCLUSION

For my initial experimentation at the most basic levels of the GPU arithmetic operations, shaders, I worked within the context of the OpenGL pipeline. Although this was quickly superseded by the transition to CUDA, an interesting spin-off of the technology was the "VisArray" platform that I developed. This platform found extensive use in mapping layered data onto basic point-clouds for Heritage Preservation, see Section 7.1.

The transition to CUDA allowed a more ambitious implement of algorithms such as the Normal Mode Langevin (NML) Course Grained Molecular Dynamics method in a hybrid CPU/GPU version of the OpenMM platform, see Section 7.2.

The work on LTMD was generally restricted to a single GPU in a single computer node. Extension of GPU techniques to the sort of environment that is generally available to researchers was a new challenge. This led to the extension of these techniques to the solution of particle laden flows in a typical HPC environment using Message Passing Interface (MPI), see Section 7.3.

7.1 Modeling Digital Heritage

In this work, we explored methods of creating realistic accurate 3D models of historical monuments from data collected using LIDAR scanners and traditional photography. This has many areas of interest, such as scientific research, interactive education for high school students, who may not necessarily be able to visit sites

such as the Roman Forum, and for students of architecture, history and archaeology that study such sites in greater detail.

Although it has been possible to acquire the data, consisting of multiple 3D point-clouds and arrays of images from different viewpoints, using the data in a Heritage Preservation and Architectural Education setting has been challenging. Software was provided with the LIDAR scanner to stitch together point-clouds from different vantage points but it was not possible to remove overlapping data that is inherent in the process. In addition there was no way to merge the images and point-clouds to form a complete navigable model at the resolutions of interest. Further issues presented themselves with distortions in the resulting panoramic vista produced by the GigaPan software, partly caused by the GigaPan requirement of a fixed focal distance which is problematic in real Heritage site viewpoints. We find that the current options, such as manual draping or stereo through motion, do not solve this problem effectively and propose a set of algorithms, tools and workflow for collecting data and automatically processing it to produce these 3D models.

Working with Dr. Krusche's team of architects and archaeologists, we develop an on-site workflow and a software platform to ingest and present data. We wrote software to merge multiple point-clouds into the same viewpoint and to cull the redundant overlapping data. We implemented algorithms to generate a surface for the model, including compression techniques to remove redundant surface features. Finally, we implemented algorithms to scale, rotate and position images on the generated surface as well as choosing the optimal image pixels where images overlap. The resulting work has also been published as a Patent application [87] and was displayed in the Curia Julia at the Roman Forum. Using seed funding from the University of Notre Dame, we were able to acquire both a LIDAR scanner and a GigaPan robotic tripod, and document sites such as the Roman Forum and Taj Mahal in unprecedented detail.

To push this work forward, a number of improvements are possible.

Firstly, the current model generation is not "air tight" because it simply maps the separate scans to the same coordinate space. This is restrictive as it makes the models unusable for further work, such as creating replicas through 3D printing, or structural analysis to determine stresses or weaknesses within the object.

Secondly, the current implementation is somewhat performant but it's performance is still a burden whilst working in the field. By using the GPU for calculation, as well as the rendering, we should be able to improve the performance. This would allow for easier work on-site to create models from the collected data to assure the surveyors that they have not missed crucial data.

7.2 Long Timestep Molecular Dynamics

We presented a hybrid CPU/GPU implementation of the LTMD propagator with speed ups of over 5.8× compared with traditional MD integrators on GPUs. This result illustrates great potential for testing larger protein systems over a longer biological period of time. Analysis of the cost of individual sections of the method have yielded insight into how we may improve performance in the future.

Validation with Ala5 and Villin NLE has shown excellent agreement between Langevin and LTMD. With Ala5, we showed that LTMD is able to sample the conformational states with a similar distribution as Langevin, produces comparable folding times, and captures similar dynamics. Similar folding times and dynamics between Langevin and LTMD were also reported for Villin NLE.

Analyses of the effects of the various parameters such as rediagonalization period, the number of modes, epsilons used with the numerical differentiation, partitioning methods, and fast noise were presented. We found that with the proper parameters, especially the rediagonalization period and number of modes, LTMD agrees well with Langevin. Further, by choosing a larger number of modes, rediagonalization can be

performed less frequently, leading to increased performance. The choice of block epsilons for FBM were shown to agree with the known values from the theory, while the choice of S epsilons were shown to be robust over a range of values.

This implementation of LTMD, and future improvements to the performance and numerics promise an order of magnitude improvement over conventional GPU implementations of MD.

To make this work perform better and become more generally usable for simulating MD, two areas will be explored in the future.

Firstly, the current implementation of the software does not fully utilize the GPU's potential as only part of the algorithm is implemented to use it. The CPU still handles a portion of the algorithm, including block diagonalization, diagonalization of S and numerical differentiation. We anticipate that porting this code to the GPU should help reduce data transfers between the two processors and take advantage of the GPU's ability to parallelize operation, which will improve performance. Given that the diagonalization dominates the run-time of the method, performance improvements can lead to even greater gains in raw simulation performance (e.g., ns/day).

Secondly, the LTMD method is only implemented with an implicit solvent model, which approximates the water molecules rather than simulating them, which provides a large performance improvement. However, we would like to implement it with an explicit solvent model, which simulates the protein within a box of water molecules to more accurately capture the dynamics of the system.

7.3 Lagrangian Particle Laden Flow Simulation

This work shows that by migrating the particle calculation from MPI to a four GPU system, we are able to calculate the trajectories for 240 million particles with a 14.4× speed up compared to the particle calculation on our target cluster of 64 cores. This allows for simulations to greatly enhance the statistical accuracy of the

particles within a flow field.

Overall, with GPU acceleration of the flow/particle solver, system sizes (i.e. particle numbers) and flow regimes can now be reached on traditional compute clusters that were previously unobtainable. Furthermore, this demonstration highlights the unique ability of GPU devices to provide cost-efficient speedup over mere extension of CPU clusters, even when favorable scaling exists. To enhance these results further and push particle-laden turbulence research into unexplored areas, there are three optimization's we would like to target.

Firstly, we would like to convert the computation to a mixed precision format rather than the solely double precision computation that the current implementation supports. This would allow for more particles to fit into the GPU pushing the statistical accuracy of simulations further, as well as benefit from GPUs higher performance with single precision calculations. These performance improvements could range anywhere from 2× for a enterprise GPU, to 32× for consumer GPUs.

Secondly, the implementation currently does not provide the same functionality that the CPU version does. We would like to find an optimal way of implementing particle-particle interaction to allow for more realistic simulations of real world phenomena.

Finally, we would like to extend the method to implement creation and destruction of particles. This would allow us to provide a more realistic method of achieving a steady state for the flow and particles, compared to the current version which reflects particles off of the boundaries.

7.4 Closing Notes

GPU development is complex due to the specialized knowledge required to achieve optimal efficiency with implementations. I feel that whilst the tooling and libraries available will improve, reducing the barriers to entry for applications that do not re-

quire the maximal benefit, it will not provide the maximum available benefit. Developers who wish to reach this optimal performance goal will still require the knowledge of the architecture to effectively implement their solutions.

As described in Chapter 3, we can expect GPUs to become more powerful, whilst keeping a realistic power budget. I feel that the trend of many simple cores will most likely continue, allowing the current programming model to remain effective.

As I expect the architectural challenges to remain the same, I would like to explore expanding the Lagrangian particle code to utilize multiple GPUs per node to allow for extreme numbers of particles to be simulated concurrently.

One of the exciting architectural changes is the introduction of the Tensor Cores. Current deep learning packages that can utilize the GPU, such as TensorFlow, have provided extensive linear algebra packages, which could be leveraged in a non deep-learning environment. These Tensor Cores, could become the "new GPU" for innovative algorithmic development. This is highly applicable to LTMD as it would allow a full GPU algorithm implementation, compared to the current hybrid implementation. For the digital heritage modeling project, a large number of algorithms could be replaced with deep learning approaches, as well as allowing implementation of structural recognition.

APPENDIX A

SOURCE CODE

A.1 DHARMA

A.1.1 Point to Surface Mapping

```
def CalculateSurface(Horizontal, Vertical, Points):
  Vertices = []
  for i in range(Horizontal - 1):
    for j in range(Vertical - 1):
      lineA = i * Vertical + j; lineB = (i+1) * Vertical + j
      a = Points.get(lineA); b = Points.get(lineB)
      c = Points.get(lineB + 1); d = Points.get(lineA + 1)

      count = 4
      if(a.isZero) count -= 1
      if(b.isZero) count -= 1
      if(c.isZero) count -= 1
      if(d.isZero) count -= 1
      if(count < 3) continue

      if(a.isZero):
        Vertices.append(b, c, d)
        continue
      if(b.isZero):
        Vertices.append(a, c, d)
        continue
      if(c.isZero):
```

```
        Vertices.append(a, b, d)
        continue
      if(d.isZero):
        Vertices.append(a, b, c)
        continue
      Vertices.append(a, b, c)
      Vertices.append(c, d, a)
  return Vertices
```

A.1.2 Point Reduction in $\mathcal{O}(N)$

```
def PrunePoints(Threshold, Vertex, Polygons, Normal, Vertical,
    Horizontal,
  PointUses, Criteria):

  result = Polygons[:]
  pointUses = PointUses[:]

  step = 2
  while step < max(Horizontal,Vertical):
    hStep = step / 2
    pUses = [0] * len(Vertex)

    for i in range(hStep, Horizontal,step):
      if i + hStep > Horizontal:
        continue

      for j in range(hStep,Vertical,step):
        if j + hStep > Vertical:
          continue

        lineA = ( ( i - hStep ) * Vertical ) + j
        lineB = ( i * Vertical ) + j
        lineC = ( ( i + hStep ) * Vertical ) + j
```

```
if pointUses[lineB] == 4:
  normals = [
    Normal[lineA+hStep], Normal[lineB+hStep], Normal[lineC+
        hStep],
    Normal[lineA], Normal[lineB], normal[lineC],
    Normal[lineA-hStep], Normal[lineB-hStep], Normal[lineC-
        hStep]
  ]

  removable = true

  if Criteria == Center:
    for k in range(10):
      top = dot(normals[4], normals[k])
      bottom = normals[4].Magnitude()*normals[k].Magnitude()

      angle = acos(top / bottom) * (180.0/3.14159)
      if angle >= Threshold:
        removable = false
        break
  else:
    angle = 0.0
    for k in range(10):
      if k != 4:
        top = dot(normals[4], normals[k])
        bottom = normals[4].Magnitude()*normals[k].Magnitude()

        angle += pow(acos(top / bottom) * (180.0/3.14159), 2)
    angle = sqrt(angle) / 8

    if angle >= Threshold:
```

```
                removable = false
                break

            if removable:
              // Remove Bottom Left
              result -= Polygon(lineA-hStep, lineB-hStep, lineB)
              result -= Polygon(lineB, lineA, lineA-hStep)

              // Remove Top Left
              result -= Polygon(lineA, lineB, lineB+hStep)
              result -= Polygon(lineB+hStep, lineA+hStep, lineA)

              // Remove Bottom Right
              result -= Polygon(lineB-hStep, lineC-hStep, lineC)
              result -= Polygon(lineC, lineB, lineB-hStep)

              // Remove Top Right
              result -= Polygon(lineB, lineC, lineC+hStep)
              result -= Polygon(lineC+hStep, lineB+hStep, lineB)

              // Create Replacement Points
              result -= Polygon(lineA-hStep, lineC-hStep, lineC+hStep)
              result += Polygon(lineC+hStep, lineA+hStep, lineA-hStep)

              // Update Usages for Next Step
              pUses[lineA-hStep]++
              pUses[lineA+hStep]++
              pUses[lineC-hStep]++
              pUses[lineC+hStep]++
    pointUses = pUses[:]
    step += step
  return result
```

A.1.3 Mesh Pruning to Remove "Poor" Triangles and Overlap

```
def FindAverage(Verticies, Polygons):
    average = 0
    for poly in Polygons:
        bDiff = Verticies[poly.b] - Verticies[poly.a]
        cDiff = Verticies[poly.c] - Verticies[poly.a]
        size = CrossProduct(bDiff, cDiff).Magnitude() * 0.5
        average += retVal / Verticies[poly.a].Magnitude
    return Absolute(average / len(Polygons))

def Prune(Threshold, Verticies, Polygons):
    remaining = []
    for poly in Polygons:
        bDiff = Verticies[poly.b] - Verticies[poly.a]
        cDiff = Verticies[poly.c] - Verticies[poly.a]
        size = CrossProduct(bDiff, cDiff).Magnitude() * 0.5
        area = size / Verticies[poly.a].Magnitude
        if area < Threshold:
            remaining.append(poly)
    return remaining
```

BIBLIOGRAPHY

1. S. H. Ahn. OpenGL transformation. `http://www.songho.ca/opengl/gl_transform.html`, September 2017.

2. M. Aissa, T. Verstraete, and C. Vuik. Toward a gpu-aware comparison of explicit and implicit cfd simulations on structured meshes. *Computers and Mathematics with Applications*, 74(1):201–217, 2017. doi: 10.1016/j.camwa.2017.03.003.

3. B. Alder and T. Wainwright. Phase transition for a hard sphere system. *J. Chem. Phys.*, 27:1208, 1957.

4. B. Alder and T. Wainwright. Studies in molecular dynamics. i. general method. *J. Chem. Phys.*, 31:459, 1959.

5. B. Alder and T. Wainwright. Studies in molecular dynamics. ii. behavior of a small number of elastic spheres. *J. Chem. Phys.*, 33:1439, 1960.

6. H. Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *J. Chem. Phys.*, 72:2384, 1980.

7. S. Balachandar and J. K. Eaton. Turbulent dispersed multiphase flow. *Annual Review of Fluid Mechanics*, 42:111–133, 2010. doi: 10.1146/annurev.fluid.010908.165243.

8. A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, May 2009. doi: 10.1109/IPDPS.2009.5160922. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5160922`.

9. N.-V. Buchete and G. Hummer. Coarse master equations for peptide folding dynamics. *The Journal of Physical Chemistry B*, 112(19):6057–69, May 2008. ISSN 1520-6106. doi: 10.1021/jp0761665. URL `http://www.ncbi.nlm.nih.gov/pubmed/18232681`.

10. M. Christen. OpenGL: Per fragment lighting. `https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/lighting.php`, September 2017.

11. P.-Y. Chuang and L. A. Barba. Accelerating petsc-based cfd codes with multi-gpu computing. *The International Conference for High Performance Computing, Networking, Storage and Analysis organized by ACM*, November 2016, 2016.

12. P. Cignoni, P. Cignoni, M. Callieri, M. Callieri, M. Corsini, M. Corsini, M. Dellepiane, M. Dellepiane, F. Ganovelli, F. Ganovelli, G. Ranzuglia, and G. Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. *Sixth Eurographics Italian Chapter Conference*, pages 129–136, 2008. doi: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

13. M. Corsini, M. Dellepiane, F. Ponchio, and R. Scopigno. Image-to-geometry registration: A Mutual Information method exploiting illumination-related geometric properties. *Computer Graphics Forum*, 28(7):1755–1764, 2009. ISSN 01677055. doi: 10.1111/j.1467-8659.2009.01552.x.

14. B. Curless and M. Levoy. A volumetric method for building complex models from range images. *Proceedings of the 23rd annual conference on ...*, pages 303–312, 1996. ISSN 00978930. doi: 10.1145/237170.237269. URL `http://portal.acm.org/citation.cfm?doid=237170.237269`.

15. L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Eng.*, 5(1):46–55, 1998.

16. T. Darden, D. York, and L. Pederson. Particle Mesh Ewald: An N log N method for Ewald sums in large systems. *J. Chem. Phys.*, 98(12):10089–10092, 1993.

17. J. M. Deutch and I. Oppenheim. Molecular theory of Brownian motion for several particles. *J. Chem. Phys.*, 54:3547, 1971.

18. P. Durand, G. Trinquier, and Y. Sanejouand. A new approach for determining low-frequency normal modes in macromolecules. *Biopolymers*, 34(6):759–771, June 1994. ISSN 0006-3525. doi: 10.1002/bip.360340608. URL `http://onlinelibrary.wiley.com/doi/10.1002/bip.360340608/abstract`.

19. P. Eastman and V. S. Pande. A hardware-independent framework for molecular simulations. *Comput. Sci. Eng.*, 12:34–39, 2010.

20. S. Elghobashi. On predicting particle-laden turbulent flows. *Applied scientific research*, 52(4):309–329, 1994.

21. P. P. Ewald. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Ann. Phys.*, 369:253–287, 1921.

22. FARO. Faro laser scanner software - scene - overview, September 2017. URL `http://www.faro.com/en-us/products/faro-software/scene/overview`.

23. J. J. Finnigan, R. H. Shaw, and E. G. Patton. Turbulence structure above a vegetation canopy. *Journal of Fluid Mechanics*, 637:387–424, 2009. doi: 10.1017/S0022112009990589.

24. M. Folwer. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, 2002.

25. T. Franken, M. Dellepiane, F. Ganovelli, P. Cignoni, C. Montani, and R. Scopigno. Minimizing user intervention in registering 2D images to 3D models. *Visual Computer*, 21(8-10):619–628, 2005. ISSN 01782789. doi: 10.1007/s00371-005-0309-z.

26. M. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. Beberg, D. Ensign, C. Bruns, and V. Pande. Accelerating molecular dynamic simulation on graphics processing units. *J. Comp. Chem.*, 30(6):864–872, 2009. ISSN 1096-987X. doi: 10.1002/jcc. URL `http://onlinelibrary.wiley.com/doi/10.1002/jcc.21209/full`.

27. A. Ghysels, D. Van Neck, V. Van Speybroeck, T. Verstraelen, and M. Waroquier. Vibrational modes in partially optimized molecular systems. *J. Chem. Phys.*, 126(22), June 2007. ISSN 0021-9606. doi: 10.1063/1.2737444. URL `http://www.ncbi.nlm.nih.gov/pubmed/17581039`.

28. A. Ghysels, D. Van Neck, and M. Waroquier. Cartesian formulation of the mobile block Hessian approach to vibrational analysis in partially optimized systems. *J. Chem. Phys.*, 127(16), Oct. 2007. ISSN 0021-9606. doi: 10.1063/1.2789429. URL `http://www.ncbi.nlm.nih.gov/pubmed/17979320http://link.aip.org/link/?JCPSA6/127/164108/1`.

29. A. Ghysels, D. Van Neck, B. R. Brooks, V. Van Speybroeck, and M. Waroquier. Normal modes for large molecules with arbitrary link constraints in the mobile block Hessian approach. *J. Chem. Phys.*, 130(8), Mar. 2009. ISSN 1089-7690. doi: 10.1063/1.3071261. URL `http://www.ncbi.nlm.nih.gov/pubmed/19256597`.

30. A. Ghysels, V. Van Speybroeck, E. Pauwels, D. Van Neck, B. R. Brooks, and M. Waroquier. Mobile Block Hessian approach with adjoined blocks: an efficient approach for the calculation of frequencies in macromolecules. *J. Chem. Theory Comput.*, 5(5):1203–1215, May 2009. ISSN 1549-9618. doi: 10.1021/ct800489r. URL `http://pubs.acs.org/doi/abs/10.1021/ct800489r`.

31. C. S. o. M. Glenn Murray. Rotation about an arbitrary axis in 3 dimensions, September 2017. URL `http://inside.mines.edu/~gmurray/ArbitraryAxisRotation/ArbitraryAxisRotation.html`.

32. J. Goldstone, A. Salam, and S. Weinberg. Broken Symmetries. *Phys. Rev.*, 127 (3):965–970, 1962.

33. A. W. Gotz, M. J. Williamson, D. Xu, D. Poole, S. LeGrand, and R. C. Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs. *J. Chem. Theory Comput.*, 8(5):1542–1555, 2012.

34. W. W. Grabowski and L.-P. Wang. Growth of cloud droplets in a turbulent environment. *Annual Review of Fluid Mechanics*, 45:293–324, 2012. doi: 10.1146/annurev-fluid-011212-140750.

35. Y. Hanada, S. Kitaoka, and Y. Xinhua. Optimizing particle simulation for kepler gpu. *Procedia Engineering*, 61:376 – 380, 2013. ISSN 1877-7058. doi: http://dx.doi.org/10.1016/j.proeng.2013.08.030. URL http://www.sciencedirect.com/science/article/pii/S1877705813012125. 25th International Conference on Parallel Computational Fluid Dynamics.

36. M. J. Harvey, G. Giupponi, and G. D. Fabritiis. ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale. *J. Chem. Theory Comput.*, 5: 1632–1639, 2009.

37. Intel. Intel xeon e5-2600 metrics, September 2017. URL http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf.

38. Intel. The math kernel library. http://software.intel.com/en-us/articles/intel-mkl/, September 2017.

39. J. A. Izaguirre and R. D. Skeel. *The Five Femtosecond Time Step Barrier*, pages 303–318. Springer-Verlag, Berlin, 1998.

40. J. A. Izaguirre, D. P. Catarello, J. M. Wozniak, and R. D. Skeel. Langevin stabilization of molecular dynamics. *J. Chem. Phys.*, 114(5), 2001. ISSN 00219606. doi: 10.1063/1.1332996. URL http://link.aip.org/link/JCPSA6/v114/i5/p2090/s1&Agg=doi.

41. J. A. Izaguirre, C. R. Sweet, and V. S. Pande. Multiscale dynamics of macromolecules using normal mode langevin. In R. B. Altman, A. K. Dunker, L. Hunter, T. Murray, and T. E. Klein, editors, *Pacific Symposium on Biocomputing*, pages 240–251. World Scientific Publishing, 2010. ISBN 978-981-4295-29-1. URL http://dblp.uni-trier.de/db/conf/psb/psb2010.html#IzaguirreSP10.

42. Z. Janko and D. Chetverikov. Photo-consistency based registration of an uncalibrated image pair to a 3d surface model using genetic algorithm. In *Proceedings of the 3D Data Processing, Visualization, and Transmission, 2Nd International Symposium*, 3DPVT '04, pages 616–622, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2223-8. doi: 10.1109/3DPVT.2004.94. URL http://dx.doi.org/10.1109/3DPVT.2004.94.

43. M. Kazhdan and H. Hoppe. Screened poisson surface reconstruction. *ACM Transactions on Graphics*, 32(3):1–13, 2013. ISSN 07300301. doi: 10.1145/2487228.2487237. URL http://dl.acm.org/citation.cfm?doid=2487228.2487237.

44. I. Kolossváry and C. McMartin. On the degeneracy of the Hessian matrix. *J. Math. Chem.*, 9(January 1993):359–367, 1992. URL http://www.springerlink.com/index/N801140471420746.pdf.

45. J. Lebowitz, J. Percus, and L. Verlet. Ensemble dependence of fluctuations with application to machine computations. *Phys. Rev.*, 153:250, 1967.

46. Leica. Leica scanners, September 2017. URL http://hds.leica-geosystems.com/en/index.htm.

47. Leica-Geosystems. Leica geosystems hds cyclone, September 2017. URL http://www.leica-geosystems.com/en/HDS-Software_3490.htm.

48. H. P. A. Lensch, W. Heidrich, and H. P. Seidel. Automated texture registration and stitching for real world models. *Proceedings - Pacific Conference on Computer Graphics and Applications*, 2000-January, 2000. ISSN 15504085. doi: 10.1109/PCCGA.2000.883955.

49. L. Liu and I. Stamos. Automatic 3D to 2D registration for the photorealistic rendering of urban scenes. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2:137–143, 2005. ISSN 10636919. doi: 10.1007/3-211-32318-X_30.

50. M. R. Lopez, A. Sheshadri, J. R. Bull, T. D. Economon, J. Romero, J. E. Watkins, D. M. Williams, F. Palacios, A. Jameson, and D. E. Manosalvas. Verification and validation of hifiles: a high-order les unstructured solver on multi-gpu platforms. *32nd AIAA Applied Aerodynamics Conference*, 2014, 2014. doi: https://doi.org/10.2514/6.2014-3168.

51. Q. Ma, J. A. Izaguirre, and R. D. Skeel. Verlet-I/r-RESPA/Impulse is limited by nonlinear instabilities. *SIAM J. Sci. Comput.*, 24(6):1951–1973, 2003. ISSN 10648275. doi: 10.1137/S1064827501399833. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.2183&rep=rep1&type=pdf.

52. C. Marchioli, A. Soldati, J. G. M. Kuerten, B. Arcen, A. Taniere, G. Goldensoph, K. D. Squires, M. F. Cargnelutti, and L. M. Portela. Statistics of particle dispersion in direct numerical simulations of wall-bounded turbulence: Results of an international collaborative benchmark test. *International Journal of Multiphase Flow*, 34:879–893, 2008. doi: 10.1016/j.ijmultiphaseflow.2008.01.009.

53. W. Mathworld. Line-plane intersection, September 2017. URL http://mathworld.wolfram.com/Line-PlaneIntersection.htm.

54. T. Matthey, T. Cickovski, S. Hampton, A. Ko, Q. Ma, M. Nyerges, T. Raeder, T. Slabach, and J. A. Izaguirre. ProtoMol , An Object-Oriented Framework for Prototyping Novel Algorithms for Molecular Dynamics. *ACM Trans. Math. Soft.*, 30(3):237–265, 2004.

55. C.-H. Moeng. A large-eddy-simulation model for the study of planetary boundary-layer turbulence. *Journal of the Atmospheric Sciences*, 41(13):2052–2062, 1984. doi: 10.1175/1520-0469(1984)041⟨2052:ALESMF⟩2.0.CO;2.

56. A. Mokos, B. D. Rogers, and P. K. Stansby. A multi-phase particle shifting algorithm for sph simulations of violent hydrodynamics with a large number of particles. *Journal of Hydraulic Research*, 55(2):143–162, 2017. doi: 10.1080/00221686.2016.1212944. URL http://dx.doi.org/10.1080/00221686.2016.1212944.

57. C. R. Nave. Potential energy. http://hyperphysics.phy-astr.gsu.edu/hbase/pegrav.html#cfor, September 2017.

58. NVIDIA. An even easier introduction to cuda. https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/, September 2017.

59. NVIDIA. Inside pascal: Nvidias newest computing platform. https://devblogs.nvidia.com/parallelforall/inside-pascal/, September 2017.

60. NVIDIA. Inside volta: The worlds most advanced data center GPU. https://devblogs.nvidia.com/parallelforall/inside-volta/, September 2017.

61. C. of the Dutchman. Creating a character. http://www.crossofthedutchman.com/2012/03/creating-character/, September 2017.

62. A. Onufriey, D. Bashford, and D. A. Case. Effective Born radii in the generalized Born approximation. *J. Comp. Chem.*, 23(14):1297–1304, 2002.

63. OpenCL. The open standard for parallel programming of heterogeneous systems. http://www.kronos.org/opencl/, September 2017.

64. V. Pande and P. Eastman. Openmm: A hardware-independent framework for molecular simulations. *Computing in Science and Engineering*, 12:34–39, 2010. ISSN 1521-9615. doi: doi.ieeecomputersociety.org/10.1109/MCSE.2010.27.

65. E. Pearson, T. Halicioglu, and W. Tiller. Laplace-transform technique for deriving thermodynamic equations from the classical microcanonical ensemble. *Phys. Rev. A*, 32:3030, 1985.

66. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message driven parallel application to GPU-accelerated clusters. In *Proceedings of SC08: The 2008 ACM/IEEE Conference on Supercomputing*, pages 1–9, Austin, TX, 2008.

67. J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters. In *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, number November, pages 1–9, 2008. ISBN 9781424428359.

68. W. H. Press, S. A. Teukolsky, W. T. Vetteringly, and B. P. Flannery. *Numerical Recipes in C++*. Cambridge, New York, 2002.

69. D. H. Richter and P. P. Sullivan. Sea surface drag and the role of spray. *Geophysical Research Letters*, 40:656–660, 2013. doi: 10.1002/grl.50163.

70. D. H. Richter and P. P. Sullivan. Momentum transfer in a turbulent, particle-laden Couette flow. *Physics of Fluids*, 25:053304, 2013. doi: 10.1063/1.4804391.

71. R. B. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). *Proceedings - IEEE International Conference on Robotics and Automation*, pages 1 – 4, 2011. ISSN 10504729. doi: 10.1109/ICRA.2011.5980567. URL `http://pointclouds.org/`.

72. G. Sardina, P. Schlatter, L. Brandt, F. Picano, and C. M. Casciola. Wall accumulation and spatial localization in particle-laden wall flows. *Journal of Fluid Mechanics*, 699:50–78, apr 2012. doi: 10.1017/jfm.2012.65.

73. J. Schalkwijk, H. J. J. Jonker, A. P. Siebesma, and E. V. Meijgaard. Weather forecasting using gpu-based large-eddy simulations. *Bulletin of the American Meteorological Society*, 96(5):715–723, 2015. doi: 10.1175/BAMS-D-14-00114.1. URL `https://doi.org/10.1175/BAMS-D-14-00114.1`.

74. T. Schlick. *Molecular Modeling and Simulation: An Interdisciplinary Guide*, volume 21. 2010. ISBN 978-1-4419-6350-5. doi: 10.1007/978-1-4419-6351-2.

75. T. Schneider and E. Stoll. Molecular-dynamics study of a three-dimensional one-component model for distortive phase transitions. *Phys. Rev. B*, 17:1302, 1978.

76. D. Shaw, M. Deneroff, R. Dror, J. Kuskin, R. Larson, J. Salmon, C. Young, B. Batson, K. Bowers, J. Chao, and Others. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, 35(2):91–97, 2008. doi: 10.1145/1364782. URL `http://portal.acm.org/citation.cfm?id=1250664`.

77. D. Shaw, R. Dror, J. Salmon, J. Grossman, K. Mackenzie, J. Bank, C. Young, M. Deneroff, B. Batson, K. Bowers, and Others. Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC09)*, number c, page 65. ACM, 2009. URL `http://portal.acm.org/citation.cfm?id=1654126`.

78. D. E. Shaw, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. M. Deneroff, M. a. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, S. C. Wang, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, and K. J. Bowers. Anton, a special-purpose machine for molecular dynamics simulation. *ACM SIGARCH Computer Architecture News*, 35(2):1, June 2007. ISSN 01635964. doi: 10.1145/1273440.1250664. URL `http://portal.acm.org/citation.cfm?doid=1273440.1250664`.

79. D. E. Shaw, P. Maragakis, K. Lindorff-Larsen, S. Piana, R. O. Dror, M. P. Eastwood, J. a. Bank, J. M. Jumper, J. K. Salmon, Y. Shan, and W. Wriggers. Atomic-Level Characterization of the Structural Dynamics of Proteins. *Science*, 330(6002):341–346, Oct. 2010. ISSN 0036-8075. doi: 10.1126/science.1187409. URL `http://www.sciencemag.org/cgi/doi/10.1126/science.1187409`.

80. M. Shirts and V. S. Pande. Screen savers of the world unite! *Science*, 290 (5498):1903–1904, 2000. doi: 10.1126/science.290.5498.1903. URL `http://www.sciencemag.org/content/290/5498/1903.short`.

81. F. Software. OpenGL: Light and creating light sources. `http://www.falloutsoftware.com/tutorials/gl/gl8.htm`, September 2017.

82. P. R. Spalart, R. D. Moser, and M. M. Rogers. Spectral methods for the Navier-Stokes equations with one infinite and two periodic directions. *Journal of Computational Physics*, 96:297–324, 1991. doi: 10.1016/0021-9991(91)90238-G.

83. J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *J. Comp. Chem.*, 18:2618–40, 2007. doi: 10.1002/jcc. URL `http://onlinelibrary.wiley.com/doi/10.1002/jcc.20829/full`.

84. J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten. GPU-Accelerated Molecular Modeling Coming of Age. *J. Mol. Graphics*, 29(2):116–125, 2010. doi: 10.1016/j.jmgm.2010.06.010.GPU-Accelerated.

85. P. P. Sullivan and E. G. Patton. The effect of mesh resolution on convective boundary layer statistics and structures generated by large-eddy simulation. *Journal of the Atmospheric Sciences*, 68:2395–2415, 2011. doi: 10.1175/JAS-D-10-05010.1.

86. P. P. Sullivan, J. C. McWilliams, and C.-H. Moeng. A subgrid-scale model for large-eddy simulation of planetary boundary-layer flows. *Boundary-Layer Meteorology*, 71:247–276, 1994. doi: 10.1007/BF00713741.

87. C. Sweet and J. Sweet. Method for mapping a 2d image to a 3d surface, 03 2012.

88. C. R. Sweet, P. Petrone, V. S. Pande, and J. A. Izaguirre. Normal mode partitioning of Langevin dynamics for biomolecules. *Journal of Chemical Physics*, 128(14):145101, 2008. ISSN 00219606. doi: 10.1063/1.2883966.

89. F. Tama, F. X. Gadea, O. Marques, and Y. H. Sanejouand. Building-block approach for determining low-frequency normal modes of macromolecules. *PROTEINS: Struc., Func., and Genetics*, 41(1):1–7, Oct. 2000. ISSN 0887-3585. URL `http://www.ncbi.nlm.nih.gov/pubmed/10944387`.

90. H. Tanaka, K. Nakanishi, and N. Watanabe. Constant temperature molecular dynamics calculation on lennard-jones fluid and its application to water. *J. Chem. Phys.*, 78:2626, 1983.

91. C. C. van Heerwaarden, B. J. H. van Stratum, T. Heus, J. A. Gibbs, E. Fedorovich, and J.-P. Mellado. Microhh 1.0: a computational fluid dynamics code for direct numerical simulation and large-eddy simulation of atmospheric boundary layer flows. *Geoscientific Model Development Discussions*, 2017:1–33, 2017.

doi: 10.5194/gmd-2017-41. URL https://www.geosci-model-dev-discuss.net/gmd-2017-41/.

92. X. Wang, Y. Shangguan, N. Onodera, H. Kobayashi, and T. Aoki. Direct numerical simulation and large eddy simulation on a turbulent wall-bounded flow using lattice boltzmann method and multiple gpus. *Mathematical Problems in Engineering*, 2014, 2014. doi: 10.1155/2014/742432.

93. Wikipedia. Single value decomposition, September 2017. URL http://en.wikipedia.org/wiki/Singular_value_decomposition.

94. D. Winkler, M. Meister, M. Rezavand, and W. Rauch. gpusphase—a shared memory caching implementation for 2d {SPH} using {CUDA}. *Computer Physics Communications*, 213:165 – 180, 2017. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2016.11.011. URL http://www.sciencedirect.com/science/article/pii/S0010465516303666.

95. Zhengyou Zhang. Flexible camera calibration by viewing a plane from unknown orientations. *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 00(c):666–673 vol.1, 1999. ISSN 01628828. doi: 10.1109/ICCV.1999.791289. URL http://ieeexplore.ieee.org/document/791289/.