Parallel Methods for Protein Coordinate Conversion

A Thesis Presented
by

Mahsa Bayati

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

April 2015
To my family.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>List of Figures</strong></td>
<td>iii</td>
</tr>
<tr>
<td><strong>List of Tables</strong></td>
<td>iv</td>
</tr>
<tr>
<td><strong>Acknowledgments</strong></td>
<td>vi</td>
</tr>
<tr>
<td><strong>Abstract of the Thesis</strong></td>
<td>vii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Thesis Organization</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1 Chemistry</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Amino Acid Building Blocks</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2 Internal Coordinates</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Parallel Programming Techniques</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1 GPU Architecture</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 SIMT Architecture</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3 CUDA: Compute Unified Device Architecture</td>
<td>8</td>
</tr>
<tr>
<td>2.2.4 Memory Hierarchy</td>
<td>9</td>
</tr>
<tr>
<td>2.2.5 Heterogeneous Programming</td>
<td>9</td>
</tr>
<tr>
<td>2.2.6 OpenMP</td>
<td>12</td>
</tr>
<tr>
<td>2.2.7 Parallel Computing Toolbox MATLAB (PCT)</td>
<td>12</td>
</tr>
<tr>
<td>2.3 Related Work</td>
<td>13</td>
</tr>
<tr>
<td><strong>3 Methodology and Design</strong></td>
<td>15</td>
</tr>
<tr>
<td>3.1 Cartesian to Internal Coordinate Conversion</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Internal to Cartesian Coordinate Conversion</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1 Local Cartesian Coordinates</td>
<td>21</td>
</tr>
<tr>
<td>3.2.2 Merge Local Coordinates</td>
<td>22</td>
</tr>
<tr>
<td>3.2.3 Global Cartesian Coordinates</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Summary</td>
<td>24</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Diagram of internal coordinates representation. The atom positions are, in Cartesian coordinates, represented by the vectors \( \mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k, \) and \( \mathbf{r}_l. \) The distance between two bonded atoms is written \( d_{ij}, \) the angle between three bonded atoms is written \( \theta_{ijk}, \) and the dihedral angle between four bonded atoms is written \( \tau_{ijkl}. \) ....... 3

2.1 Connecting two amino acid through peptide bond [1] ............... 5
2.2 Amino acids differ in side chains (This is not all amino acids) ................. 6
2.3 Thread hierarchy in GPU [2] ................................................. 10
2.4 Memory hierarchy in GPU [2] ................................................. 10
2.5 Heterogenous programming model [2] ....................................... 11
2.6 Master and worker thread in OpenMP [3] .................................... 12

3.1 Bond distance [4] ................................................................. 16
3.2 Bond angle [4] ................................................................. 17
3.3 Proper dihedral [4] ................................................................. 17
3.4 Improper dihedral [4] ................................................................. 17
3.5 Overview of computation ......................................................... 17
3.6 Protein amino acid [5] ................................................................. 22
3.7 Merging local coordinates using residue representation for 8 protein residues ......................................................... 26
3.8 Merging local coordinates using segment representation for 8 protein residues ......................................................... 27
3.9 Merge in level \( h \) and \( h + 1 \) using notation ......................................................... 27
3.10 Transformation operation storing process ......................................................... 28
3.11 Pattern used to accumulate transformation operation and calculate global Cartesian coordinates for each residue ......................................................... 28

4.1 Total time (ms) for tripeptide ......................................................... 30
4.2 Total time (ms) for lysozyme ......................................................... 31
4.3 Proline amino acid [6] ................................................................. 34
4.4 Disulfide bond [1] ................................................................. 34
4.5 Alanine amino acid ................................................................. 36
4.6 Total time (ms) for reverse conversion ......................................................... 36
List of Tables

3.1 Notation Table .................................................. 20
4.1 Size of input and output for lysozyme and tripeptide ...................... 31
4.2 Forward conversion timing (ms) results for tripeptide ...................... 32
4.3 Forward conversion timing (ms) results for lysozyme ....................... 32
4.4 Size of LDH for reverse conversion .................................. 34
4.5 Reverse conversion timing (ms) results .................................. 35
4.6 Reverse conversion CUDA-C kernel and memory copy time (ms) ........... 35
4.7 Reverse conversion timing (ms) for a simple alanine chain .................. 36
List of Algorithms

1. Reverse Conversion 1 ................................................................. 20
2. Local Cartesian Coordinates .................................................... 21
3. Merge Cartesian Coordinates 1 ............................................... 23
4. Merge Cartesian 2 ................................................................. 25
5. Cartesian Rebuild ................................................................. 26
6. Internal to Cartesian Coordinates 2 ......................................... 26
Acknowledgments

The author would like to thank Mathworks, NVIDIA, and NSF (under award number CCF-1218075) for all their support during the process of the thesis work.
Abstract of the Thesis

Parallel Methods for Protein Coordinate Conversion

by

Mahsa Bayati

Master of Science in Electrical and Computer Engineering
Northeastern University, April 2015
Dr. Miriam Leeser, Adviser

Proteins contain thousands to millions of atoms. Their positions can be represented using one of two methods: Cartesian or internal coordinates (bond lengths, angles, etc.). In molecular dynamics and modeling of proteins in different conformational states, it is often necessary to transform one coordinate system to another. In addition, since proteins change over time, any computation must be done over successive time frames, increasing the computational load. To lessen this computational load we have applied different parallel techniques to the protein conversion problem. The Cartesian to internal coordinate translation computes bond distances, bond angles, and torsion angles for each time frame by using the protein chemical structure and atomic trajectories as inputs. This direction is easily parallelizable and we realized several orders of magnitude speed up using various parallel techniques including a GPU implementation. The reverse direction, is used in molecular simulations for such tasks as fitting atomic structures to experimental data and protein engineering. This computation has inherent dependency in the data structures because bond lengths and angles are relative to neighboring atoms. Existing implementations walk over a protein structure in a serial fashion. This thesis presents the first fast parallel implementation of internal to Cartesian coordinates, in which substructures of the protein backbone are converted into their own local Cartesian coordinate spaces, and then combined using a reduction technique to find global Cartesian coordinates. We observed orders of magnitude speedup using parallel processing.
Chapter 1

Introduction

For modeling proteins in conformational states, two methods of representation are used: internal coordinates and Cartesian coordinates. Each of these representations contain a large amount of structural and simulation information. Different processing steps require one or the other representation. This research addresses efficient, scalable algorithms to convert between two different representations of molecular coordinates, so that a scientist can choose whichever method he or she would like independent of the coordinate representation required. Representation in Cartesian coordinates is intuitive: each atom is associated with a point in Cartesian space, i.e. atom i’s center is located at $(x_i, y_i, z_i)$. This representation allows easy file I/O and simple manipulations involving rigid-body motion (rotations and translations). The other representation, known as internal coordinates, describes a molecule’s atomic positions using chemically relevant features such as the distance between two atoms that are covalently bonded, or the angle formed by a chain of three bonded atoms (Fig. 1.1). Physical forces between atoms are most naturally expressed in this representation; for example, the force between two bonded atoms is usually modeled as a harmonic spring. Internal coordinates require more complicated data structures and management, which indicate, for instance, all of the chemical bonds between atoms.

Standard molecular dynamics simulations convert Cartesian coordinates to internal ones (the forward coordinate transformation) at every time step, necessitating fast algorithms for the forward transformation. Many other kinds of calculations also require fast algorithms for the reverse transformation, which converts from internal to Cartesian coordinates. Examples include protein structure refinement (improving the quality of experimentally estimated protein structures using modeling) and understanding large changes in protein structure. In these types of applications, internal coordinates offer advantages because the relevant conformational changes
CHAPTER 1. INTRODUCTION

involve primarily dihedral angles (Fig. [1.1]), which effectively reduce the number of degrees of freedom. However, the reverse coordinate transform is less easily parallelized than the forward transform, necessitating optimization or search algorithms that use Cartesian coordinates and have to impose complicated (slow) constraints.

Note that a protein can contain thousands to millions of atoms. Our goal is to process these as well as much bigger molecules, such as DNA. In addition, molecule shapes change over time. Thus each atom is represented in a large number of time frames. MD programs go from Cartesian coordinates \((x,y,z)\) to internal coordinates (bond angles, dihedrals) because the forces on atoms are defined in terms of internal coordinates. Through the chain rule, you compute the Cartesian coordinate derivatives in terms of the internal-space coordinates. Converting Cartesian coordinate to internal coordinates is what we have called the forward problem. It is relatively easy to parallelize because internal coordinates can be computed independently.

However, most MD programs do not provide a fast or parallel way to go from arbitrary internal coordinates to Cartesian coordinates. This reverse problem is more difficult due to dependencies along the chain. Some packages provide such functionality (CHARMM, PyMOL, VMD) but not in parallel and not packaged with a generic software interface so that it can be easily used by other researchers. Our main goal is to implement rapid, efficient and scalable conversion between these two coordinate spaces in both directions, independent of other processing. The implementation of Cartesian to internal coordinate translation computes internal coordinates including bond distances, bond angles, and torsion angles for each time frame by using the protein chemical structure and atomic trajectories as two input files. We improve the speed of a serial implementation from minutes to milliseconds by using CUDA-C with data streaming and overlapping computation on modern GPUs. The reverse direction is more complicated. In the complex bonded structure of proteins, each atom is connected to other atoms, making internal coordinates of atoms dependent on each other. We have designed an algorithm to overcome this dependent structure. We first calculate the Cartesian coordinates of each residue of a protein in residue specific local coordinate system, and then merge into a global Cartesian coordinate system. Using GPU programming we observed orders of magnitude speedup.

The contributions of this thesis are a software package that accelerates forward and reverse coordinate conversion that is independent of other processing.

For the reverse conversion:

- To our knowledge, this is the first parallel implementation of internal to Cartesian coordinates,
CHAPTER 1. INTRODUCTION

a problem of interest in protein engineering.

• The approach works via an initial hierarchical reconstruction of the protein backbone – the linear chain that creates the major dependency problem. Side chains will be added in a separate, second stage. Each amino acid’s side chain atoms can be reconstructed independently, making this stage easily parallelizable.

• This method can be applied to large structures that cannot be handled by molecular dynamics packages, such as polymers where each monomer (repeated unit of the polymer) is represented as a bead.

![Diagram of internal coordinates representation. The atom positions are, in Cartesian coordinates, represented by the vectors $\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k$, and $\mathbf{r}_l$. The distance between two bonded atoms is written $d_{ij}$, the angle between three bonded atoms is written $\theta_{ijk}$, and the dihedral angle between four bonded atoms is written $\tau_{ijkl}$.]

1.1 Thesis Organization

Chapter 2 presents the background on parallel programming techniques we used during this research including GPU programming, as well as background on chemistry and protein structure. Later in this chapter we discuss related work.

Chapter 3 covers the methodology and design of an accelerated Cartesian to internal coordinate translation as well as the novel method for parallelizing internal to Cartesian conversion by reducing dependencies.

Chapter 4 describes the hardware and software setup as well as the input test cases used in evaluation of the coordinate conversion implementation in both directions. Then the experimental results are presented.

Chapter 5 explains our future work and the conclusions from this research.
Chapter 2

Background

2.1 Chemistry

2.1.1 Amino Acid Building Blocks

Proteins and polypeptides are composed of linked amino acids. That amino acid composition of the polymer is known as the primary structure or sequence. A polypeptide is formed when amino acids join together with a peptide bond. The carboxyl carbon of one amino acid joins the amino nitrogen of another amino acid to form the peptide bond with the release of one water molecule (See Fig. 2.1). Each amino acid has the same fundamental structure called backbone, this basic geometry of amino acid residues is quite well determined. The amino acid backbone contains a nitrogen, two carbons and an oxygen atom \[12\]. Amino acids differs only in the side-chain, designated the R-group. The carbon atom to which the amino group, carboxyl group, and side chain (R-group) are attached is the alpha carbon \((C_a)\). The sequence of side chains determines all that is unique about a particular protein, including its biological function and its specific three-dimensional structure. Each of the side groups has a certain ”personality” which it contributes to this task (See Fig. 2.2).

2.1.2 Internal Coordinates

We can think of molecules as mechanical assemblies made up of simple elements like balls (atoms), rods or sticks (bonds), and flexible spring-like joints:

- Bond distance: In molecular geometry, bond length or bond distance is the average distance between nuclei of two bonded atoms in a molecule. Bond length is related to bond order: when
more electrons participate in bond formation the bond is shorter. Bond length is also inversely related to bond strength.

- Bond Angle: A bond angle is the angle formed between three atoms across at least two bonds.

- Dihedral Angle: In geometry, a dihedral or torsion angle is the angle between two planes. The structure of a molecule can be defined uniquely using bonds, angles, and dihedral angles between three successive chemical bond vectors. An improper dihedral angle is a similar geometric analysis of four atoms, but involves a central atom with three others attached to it rather than the standard arrangement of all four of them bonded sequentially each to the next. One of the vectors is the bond from the central atom to one of its attachments. The other two vectors are pairs of the attachments, and thus together represent the plane of the attachments. Improper dihedral angles are useful for analyzing the planarity of the central atom: as the angle deviates from zero, the central atom moves out of the plane defined by the three attached to it.

2.2 Parallel Programming Techniques

In this section we introduce the programming models used in this research: GPU programming using CUDA-C, OpenMP, and MATLAB PCT toolbox.
CHAPTER 2. BACKGROUND

2.2.1 GPU Architecture

To accelerate protein coordinate conversion, we use a NVIDIA GPU.

The NVIDIA GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called SIMT (Single Instruction, Multiple-Thread) that is described in section 2.2.2. The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively through simultaneous hardware multithreading. Unlike CPU cores, they are issued in order and there is no branch prediction or speculative execution [2].
CHAPTER 2. BACKGROUND

2.2.2 SIMT Architecture

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address. The term warp originates from weaving, the first parallel thread technology. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually. The threads of a warp that are on that warp’s current execution path are called the active threads, whereas threads not on the current path are inactive (disabled). Threads can be inactive because they have exited earlier than other threads of their warp, or because they are on a different branch path than the branch path currently executed by the warp, or because they are the last threads of a block whose number of threads is not a multiple of the warp size. If a non-atomic instruction executed by a warp writes to the same location in global or shared memory for more than one of the threads of the warp, the number of serialized writes that occur to that location varies depending on the compute capability of the device, and which thread performs the final write
CHAPTER 2. BACKGROUND

is undefined. If an atomic instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each read/modify/write to that location occurs and they are all serialized, but the order in which they occur is undefined [2].

2.2.3 CUDA: Compute Unified Device Architecture

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high level programming language [2].

Kernel: CUDA C extends C by allowing the programmer to define C functions, which called kernel, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>> execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable [2].

Thread Hierarchy: For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume [2].

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks [2].

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated in Fig 2.3. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed. Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.
CHAPTER 2. BACKGROUND

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores. Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the `syncthreads()` intrinsic function; `syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed. For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and `syncthreads()` is expected to be lightweight [2].

2.2.4 Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Fig 2.4. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application [2].

2.2.5 Heterogeneous Programming

As illustrated by Fig 2.3 the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case when the kernels execute on a GPU and the rest of the C program executes on a CPU. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory [2].
CHAPTER 2. BACKGROUND

Figure 2.3: Thread hierarchy in GPU

Figure 2.4: Memory hierarchy in GPU
Figure 2.5: Heterogenous programming model [2]
CHAPTER 2. BACKGROUND

2.2.6 OpenMP

OpenMP is an Application Program Interface (API), jointly defined by a group of major
computer hardware and software vendors. OpenMP allows higher level of abstraction and provides a
portable, scalable model for developers of shared memory parallel applications. The API supports
C/C++ and Fortran on a wide variety of architectures. OpenMP is a pragma based method and scoping
of thread-safe data is simplified, so it is easy to modify a serial code into a parallel version \[13\].
These pragmas are ignored for serial compilation. OpenMP starts out executing the program with
one master thread which forks worker threads. Worker threads die or suspend at the end of parallel
code (See Fig 2.6).

![Figure 2.6: Master and worker thread in OpenMP](image)

2.2.7 Parallel Computing Toolbox MATLAB (PCT)

PCT lets you solve computationally and data-intensive problems using multicore processors,
GPUs, and computer clusters. High-level constructs (parallel for-loops, special array types,
and parallelized numerical algorithms) let you parallelize MATLAB applications without CUDA or
MPI programming. You can use the toolbox with Simulink to run multiple simulations of a model
in parallel. The toolbox lets you use the full processing power of multicore desktops by executing
applications on workers (MATLAB computational engines) that run locally. Without changing the
code, you can run the same applications on a computer cluster or a grid computing service (using
MATLAB Distributed Computing Server). You can run parallel applications interactively or in batch
mode \[14\].
CHAPTER 2. BACKGROUND

2.3 Related Work

Molecular dynamics and protein modeling are extremely computationally demanding which makes them natural candidates for implementation on GPUs. With currently available molecular dynamics codes, we can only simulate small and fast protein folding on a desktop. Some previous studies have implemented specific algorithms used in molecular dynamics and protein modeling. For example, [15] used a GPU to implement a simple implicit solvent (distant dependent dielectric) model. Several algorithms have been implemented [16], including integrator, neighbor lists, and Lenard-Jones potential. GPU implementation of the traditional force field [17] and the challenges and accuracy of it on a GPU [18] have also been presented.

Molecular dynamic simulations require a realistic description of the underlying physical system and its molecular interactions [18]. The traditional force field method dates back to 1940 when F. Westhiemer formulated the molecular energy with its geometry; the spatial conformation ultimately obtained is a natural adjustment of geometry to minimize the total internal energy [12]. Since this method uses internal coordinates to calculate bonded forces, it has some similarity with our coordinate conversion. The difference is that their approach concentrates on forces introduced by internal coordinates and the accumulated results to find the energy, while the output from our approach is each set of internal coordinates in each time frame.

The choice of the coordinate system is of paramount importance in molecular geometry optimizations. Cartesian coordinates provide a simple and unambiguous representation for molecular geometries, and are used for calculating the molecular energy and its derivatives. However, bond lengths, valence angles, and torsions about bonds are more appropriate coordinates to describe the behavior of molecules. Because they express the natural connectivity of chemical structures, there is much less coupling between these internal coordinates, so internal coordinates are therefore the preferred coordinates in molecular geometry optimizations [19]. If \( q(x) \) denotes the internal coordinates associated with a given set of Cartesian coordinates \( x \). For a given internal coordinate step \( \Delta q \), the task is to find a Cartesian displacement vector \( \Delta x \) that satisfies equation [2.1]

\[
q(x_0 + \delta x) = q_0(x_0) + \delta q.
\]

(2.1)

In the field of geometry optimization, different methods have been proposed to solve this equation; for example, Nemeth et al. [20] uses an iterative back transformation method called (IBT) to solve this equation. The iteration is terminated with a threshold, when the root mean-square-error change in the internal or Cartesian coordinates is less than \( 10^{-6} \). Baker et al. [21] have proposed
an alternative method for performing the back transformation. Their algorithm finds the Cartesian
displacements by setting up a Z matrix.

One more approach to the back-transformation problem was used by Dachsel et al. [22] and V. V. Rybkin et al. [23] for visualization of curvilinear molecular vibrations. This method is also based on treating nonlinear relations between internal and Cartesian coordinates. It finds a discrete path in Cartesian coordinates corresponding to a set of finite displacements in curvilinear normal coordinates using the Taylor expansion of the former with respect to the latter. The derivatives are calculated with reciprocal vector bases and covariant metric tensor. All these proposed methods are sequential; to our knowledge our method is the first parallel design of internal to Cartesian coordinates.
Chapter 3

Methodology and Design

In this chapter we explain the design and challenges of implementing the conversion from Cartesian to internal coordinates and vice versa for all of a protein’s atoms in different time frames. The first section focuses on Cartesian to internal coordinates and the different parallelization techniques we used. The second section concentrates on the reverse conversion and our new method which facilitates parallelization of this translation.

3.1 Cartesian to Internal Coordinate Conversion

In this section we focus on the forward direction: Cartesian to internal coordinates. $a$ atoms in 3D space can move and change their position over time due to the forces between atoms. This moving structure can be described directly by a list of Cartesian coordinates. Alternatively, internal variables such as bond length, bond angles and dihedral angle may also be used. The Cartesian coordinates of the atom trajectories of $X$ can be represented as:

$$\{X(t_0), X(t_0 + \Delta t), \ldots, X(t_0 + n\Delta t)\}$$

(3.1)

where $t_0$ is the initial time reference and $\Delta t$ is the time step. To specify the position of atoms in a molecular structure, scientists define an analytic expression. For the molecular system of “a” atoms in Cartesian coordinate space let: $\vec{r}_i = (x_i, y_i, z_i)$. There are four different internal coordinates that need to be computed from Cartesian coordinates. Each computation, calculated for atoms that have bonds with each other.

(i) Calculation of the bond length between two bonded atoms (Fig. 3.1):

$$d_{i,j} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}$$

(3.2)
(ii) Calculation of the angle $\theta_{ijk}$ formed by a bonded triplet (Fig. 3.2):

$$\cos \theta_{ijk} = \left( \frac{\vec{r}_{ji} \cdot \vec{r}_{jk}}{|r_{ji}| |r_{jk}|} \right)$$  \hspace{1cm} (3.3)$$

where $\vec{r}_{ij}$ is a distance vector from i to j.

(iii) Computation of torsion (dihedral) angle (proper and improper) $\tau_{ijkl}$ defining the rotation of bond $i - j$ around $j - k$ with respect to $k - l$ [12]. In other words, when four bonded atoms are in two planes the angle between these 2 planes is called the torsion angle. The torsion angle based on the position of 4 atoms can fall into two groups: (I) Proper (Fig. 3.3) meaning all 4 atoms are connected consecutively, (II) Improper dihedral (Fig. 3.4) in which 3 atoms are connected to a central atom. For the forward conversion, both dihedral angles use the same formula 3.4:

$$\cos \tau_{ijkl} = \vec{n}_{ab} \cdot \vec{n}_{bc} = \frac{\vec{a} \times \vec{b}}{||a|| ||b|| \sin \theta_{ab}} \cdot \frac{\vec{b} \times \vec{c}}{||b|| ||c|| \sin \theta_{bc}}$$  \hspace{1cm} (3.4)$$

$$\sin \tau_{ijkl} = \frac{(\vec{c} \times \vec{b} \cdot \vec{a}) \cdot \vec{b}}{||b|| ||c|| \sin \theta_{bc} ||a|| ||b|| \sin \theta_{ab}}$$

The vectors $\vec{n}_{ab}$ and $\vec{n}_{bc}$ denote unit normals to planes spanned by vectors $\vec{a}, \vec{b}$ and $\vec{b}, \vec{c}$, respectively, where the distance vectors are: $\vec{a} = r_{ij}^{-1}, \vec{b} = r_{jk}^{-1}, \vec{c} = r_{kl}^{-1}$.

To perform the forward conversion, we are given each atom’s coordinates in each time frame, then apply equations 3.2, 3.3 and 3.4. The steps for this conversion are shown in Fig. 3.5. If there are $a$ number of atom and $m$ time frames for the protein then, for the parallel implementation, we can launch $ma$ threads, where each calculation is independent from the others. Therefore, the potential for parallelization is considerable. The initial version of the computation was done in MATLAB, and had long run times. To accelerate it, we initially tried the MATLAB Parallel Computing Toolbox (PCT) [14], then translated the code to C, tried OpenMP [3] and finally implemented it to run on an NVIDIA GPU by rewriting it in CUDA-C [2]. Each of these versions is discussed below.
CHAPTER 3. METHODOLOGY AND DESIGN

Figure 3.2: Bond angle

Figure 3.3: Proper dihedral

Figure 3.4: Improper dihedral

Figure 3.5: Overview of computation
CHAPTER 3. METHODOLOGY AND DESIGN

Input data All versions of the code require two input files. The first is a protein structure file (.psf) listing each pair of atoms that has a bond, groups of three atoms which form an angle, and groups of four atoms forming a torsion angle. The second input is a DCD \cite{24}; each DCD file contains the trajectory of all protein atoms in a number of time frames. To read these files, MATLAB toolboxes are used. The protein structure files are read by MDToolbox \cite{25}, a toolbox for analysis of molecular dynamics (MD) simulation data. The DCD files are read with MatDCD, a MATLAB package for reading/writing DCD files \cite{24}. After the data files are read, the program is ready to calculate the internal coordinates by calling the relevant functions.

Output data The program returns a text file listing the internal coordinates of the molecular structure in each of the input time frames.

MATLAB and MATLAB PCT We started with a serial CPU implementation written in MATLAB. In most functions all the computation is nested in a for loop, so we use the parfor instruction to get a multi-threaded version of MATLAB using PCT.

Translation to C and OpenMP To reduce the runtime and also move towards the GPU-based implementation using CUDA, the program was rewritten in C. Some preprocessing on both input files is done using MATLAB toolboxes to put them in a readable format for C. We used OpenMP to create a multithreaded C version by putting pragmas before for loops and collapsing all nested loops.

CUDA In the GPU-based implementation, we designed three kernels: bond length, bond angle, and torsion angle. Each kernel initiates number of threads which compute the associated internal coordinate for that kernel in a specific time frame. Another important step is data transfer between host and GPU device. Since proteins have a large number of atoms, the coordinate conversion has large amounts of data and thus transferring it between host and device is costly. The typical approach is to transfer the data to the device, do all the computation, and then copy the results back to save transfer time. In this approach the kernels are executed serially. However, in this coordinate conversion problem, the kernels are independent, so we can take advantage of streaming and use asynchronous memory transfer to copy data to or from the device while simultaneously doing computation on the device \cite{26}. To achieve better performance, we do asynchronous copying and launch the kernel for each chunk of data to overlap computation and communication.
CHAPTER 3. METHODOLOGY AND DESIGN

3.2 Internal to Cartesian Coordinate Conversion

Internal coordinates are relative to the position of neighboring atoms which makes parallelization difficult. We propose a novel implementation to perform the reverse conversion with considerable speed up.

Proteins are large biological molecules consisting of one or more long chains of amino acids. These amino acids lose their hydrogen atoms, during the formation of peptide bond resulting in amino acid residues. In many computational protein design models, the backbone structure is assumed to be a rigid body and constant, but the side-chains are allowed to vary among a finite set of discrete conformations \[27\]. Our main approach is to use divide and conquer to perform the reverse conversion. We consider each residue as an independent unit to start. For now we concentrate on the rigid body of the protein structure. The main steps of the approach are:

- Calculate Cartesian coordinates locally for atoms within each residue, using internal coordinates.

- Merge residues in different coordinate systems until one unique coordinate system remains.

**Input Data** Internal coordinates are not commonly saved. For reverse conversion we use the VMD application \[9\] to preprocess a protein structure file (.PSF) and protein data bank file (.PDB) to calculate internal coordinates.

Algorithm 1 shows the pseudo-code for our approach. If \( n \) is the number of different residues in the backbone then at the first step of algorithm 1 there are \( n \) different local Cartesian coordinate systems. Inside the for each two residue segments in different local coordinate systems are combined into one coordinate system. Therefore, at each iteration the number of coordinate systems is reduced by a factor of two. For example, after the first iteration the number of local coordinates system is reduced to \( \frac{n}{2} \) and after the second iteration it equals \( \frac{n}{4} \). Thus, the number of loop (algorithm 1 line 5) iterations is \( h_{\text{max}} = \lceil \log_2 n \rceil \). In other words, the algorithm is similar to a reduction operation on a binary tree with the hight of \( h_{\text{max}} \) as shown in Fig. 3.7. If the number of residue is 8, then \( h_{\text{max}} = \log_2 8 = 3 \).

Algorithm 1 shows the high level design of the algorithm. To investigate the implementation more deeply, we need to look into the two main functions: (I) calculate Cartesian coordinates locally, (II) Merge neighboring residue segments.

The notation used in this explanation is summarized in Table 3.1.
Algorithm 1 Reverse Conversion 1

1: procedure REVERSE
2:    \( n \leftarrow \) Number of Backbone Residues
3:    for \( i = 0 \) to \( n - 1 \); \( i++ \) do
4:        Calculate Cartesian coordinates locally.
5:    for \( h = 0 \) to \( h_{\text{max}} - 1 \); \( h++ \) do
6:        Merge and reduce neighboring pairs of residue segments in different coordinate systems to one system.

Table 3.1: Notation Table

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>Total number of residues</td>
</tr>
<tr>
<td>( h )</td>
<td>level of the reduction tree ( 0 \leq h \leq \lceil \log_2 n \rceil )</td>
</tr>
<tr>
<td>( h_{\text{max}} )</td>
<td>( \lceil \log_2 n \rceil )</td>
</tr>
<tr>
<td>( 2^h )</td>
<td>number of residues within one segment in level ( h ) of the tree</td>
</tr>
<tr>
<td>( S_{k,h} )</td>
<td>( k ) segment at level ( h ) of the tree ( 0 \leq k \leq \lceil \frac{n}{2^h} \rceil )</td>
</tr>
<tr>
<td>( S_{0}^{h}_{k,h} )</td>
<td>first residue of ( k ) segment at level ( h ) of the tree. Its residue index is equal to ( (2^h) \times k )</td>
</tr>
<tr>
<td>( S_{2^h-1}^{h}_{k,h} )</td>
<td>last residue of ( k ) segment at level ( h ) of the tree. Its residue index is equal to ( (2^h) \times (k + 2^h - 1) )</td>
</tr>
<tr>
<td>( S_{\frac{k}{2},h+1}^{2^h} )</td>
<td>first residue in the second half of ( \frac{k}{2} ) segment at tree level ( h + 1 ). Its residue index is equal to ( (2^{h+1}) \times \frac{k}{2} + 2^h = 2^h \times (k + 1) ) same index as ( k + 1 ) segment at level ( h ) of the tree</td>
</tr>
<tr>
<td>( S_{\frac{k}{2},h+1}^{2^h+1-1} )</td>
<td>last residue of ( \frac{k}{2} ) segment at level ( h + 1 ) of the tree. Its residue index is equal to ( (2^{h+1}) \times \frac{k}{2} + 2^{h+1} - 1 = 2^h \times (k + 1) + 2^h - 1 ) same index as last residue of ( k + 1 ) segment at level ( h ) of the tree</td>
</tr>
</tbody>
</table>
3.2.1 Local Cartesian Coordinates

There are four main atoms in each residue (see Fig. 3.6), including nitrogen, carbon-alpha, carbon and oxygen. Assume each residue has its own coordinate system, and the atom $N$ is the origin of that coordinate system, so its $(x, y, z)$ coordinates are assumed to be zero. Other atoms’ coordinates are calculated based on the bonds and angles between these four atoms. Atom $C_\alpha$ differs only in its $x$-coordinate and is equal to the distance between $N$ and $C_\alpha$. Atom $C$ is in the same $z=0$ plane, its $x$ and $y$ coordinates are calculated base on the angle $N-C_\alpha-C$ and the bond between $C_\alpha-C$. Finally, to find the position of atom oxygen $O$ we need to use the other 3 atoms’ Cartesian coordinates, and the bond, angle and dihedral involving $O$. To do this we call the function “FindXYZWithDihedral”.

Algorithm 2 represents the pseudo code for calculating the local coordinates of residue $i$.

**Parallel Local Cartesian Coordinates** Algorithm 2 is invoked for each residue, so its parallel implementation has a potential to run $n$ times (number of residues) in parallel simultaneously.

**Algorithm 2 Local Cartesian Coordinates**

```
procedure LOCAL_CARTESIAN_COORDINATES(i)
  Local.N(i).x ← 0
  Local.N(i).y ← 0
  Local.N(i).z ← 0
  Local.C_\alpha(i).x ← bond.NC_\alpha(i)
  Local.C_\alpha(i).y ← 0
  Local.C_\alpha(i).z ← 0
  Local.C(i).x ← bond.NC_\alpha(i) + (bond.C_\alpha(i) \times \cos(\pi - angle.NC_\alpha.C(i)))
  Local(i).C.y ← bond.C_\alpha(i) \times \sin(\pi - (angle.NC_\alpha.C(i)))
  Local(i).C.z ← 0
  Local.O(i) ← FindXYZWithDihedral(bond.CO(i), angle.C_\alpha.CO(i)...,
    dihedral.NC_\alpha.CO(i), Local.N(i), Local.C_\alpha(i), Local.C(i))
  return Local / structure containing local coordinate systems for each residue */
```
3.2.2 Merge Local Coordinates

As mentioned earlier, the merging step is similar to the reduction of a binary tree. Assuming we have $n$ backbone residues, the maximum height of the reduction tree is $h_{max} = \lceil \log_2 n \rceil$. At level $h$ the number of residues in each segment with the same coordinate system is $2^h$. If $S$ denotes the segment that has index $k$, with $0 \leq k \leq \lfloor \frac{n}{2^h} \rfloor$, then $S_{k,h}$ indicates segment $k$ at level $h$ of the tree, which has $2^h$ residues. $S^0_{k,h}$ denotes the first residue of segment $k$ at level $h$ of the tree. Likewise, the last residue is shown with $S^{2^h-1}_{k,h}$ (see Fig. 3.8). At level $h+1$ of the tree, the new Cartesian coordinates of the residues $S_{\frac{2^h}{2},h+1}^0$ to $S_{\frac{2^h}{2},h+1}^{2^h-1}$ are calculated by merging segments $S_{k,h}$ and $S_{k+1,h}$ (See Fig. 3.8). To perform the merge, we join the last residue $S_{k,h}^{2^h-1}$ with the first residue $S_{k+1,h}^0$ through their connecting angles and bonds. The naive approach is to use the same technique as will be used for the rest of the residues in segment $S_{k+1,h}$ i.e. $S_{k+1,h}^1$ to $S_{k+1,h}^{2^h-1}$. Algorithm 3 summarizes the exact procedure.

As algorithm proceeds to the bottom root, in segment $S_{k+1,h}$ there are more residues to convert to the $S_{k,h}$ coordinate system. Therefore we lose our parallelization efficiency as the number of residues in the chain grows. To optimize our design, instead of using internal coordinates of all residues within the $S_{k+1,h}$ coordinate system and pin their position, we only use internal coordinates to merge $S_{k,h}^{2^h-1}$ and $S_{k+1,h}^0$. Using 3 points from one coordinate system, the transformation operation mapping to that coordinate system can be calculated. Therefore, using the newly calculated coordinates of three atoms $N$, $C_\alpha$ and $C$ of $S_{k+1,h}^0$, we can find the transformation matrix to go form $S_{k+1,h}$ coordinate system to $S_{k,h}$ which is the same coordinate system $S_{\frac{2^h}{2},h+1}$ at level $h + 1$. To perform the next levels of merges in the reduction tree correctly, we need to have the last residue of $S_{k+1,h}$ in the correct coordinate system i.e. in $S_{k,h}$. So we use the transformation matrix to only
CHAPTER 3. METHODOLOGY AND DESIGN

Algorithm 3 Merge Cartesian Coordinates 1

procedure MERGE_CARTESIAN1(h, n)

for $S_{k,h} = S_{0,h}$ to $S_{\lfloor \frac{n}{2p} \rfloor ,h}$; $k = k + 2$ do

for $S_{k+1,h} = S_{0+1,h}$ to $S_{k+1,h}^{2h-1}; i++$ do

if $i == 0$ then

$N(S_{\frac{2h+1}{2},h+1}) \leftarrow \text{FindXY ZWithDihedral}(\text{bond.CN}(S_{i}^{k+h}), \text{angle.CN}(S_{i}^{k+h})), \ldots$

dihedral.NCaCN(S_{i}^{k+h}, Local.N(S_{k,h}^{2h-1}), Local.Ca(S_{k,h}^{2h-1}), Local.C(S_{k,h}^{2h-1}))

$C_{\alpha}(S_{\frac{2h+1}{2},h}) \leftarrow \text{FindXY ZWithDihedral}(\text{bond.}\alpha(S_{i}^{k+h}), \text{angle.CNCa}(S_{i}^{k+h})), \ldots$

dihedral.CaCNCa(S_{i}^{k+h}, Local.Ca(S_{k,h}^{2h-1}), Local.C(S_{k,h}^{2h-1}), Local.N(S_{k,h}^{2h-1}))

$O(S_{\frac{2h+1}{2},h+1}) \leftarrow \text{FindXY ZWithDihedral}(\text{bond.CO}(S_{i}^{k+h}), \text{angle.CaCO}(S_{i}^{k+h})), \ldots$

dihedral.NCaCO(S_{i}^{k+h}, Local.N(S_{k,h}^{2h-1}), Local.Ca(S_{k,h}^{2h-1}), Local.C(S_{k,h}^{2h-1}))

else

$N(S_{\frac{2h+1}{2},h+1}) \leftarrow \text{FindXY ZWithDihedral}(\text{bond.CN}(S_{i}^{k+h}), \text{angle.CN}(S_{i}^{k+h})), \ldots$

dihedral.NCaCN(S_{i}^{k+h}, Local.N(S_{k,h}^{2h-1}), Local.Ca(S_{k,h}^{2h-1}), Local.C(S_{k+h}^{2h-1}))

$C_{\alpha}(S_{\frac{2h+1}{2},h}) \leftarrow \text{FindXY ZWithDihedral}(\text{bond.}\alpha(S_{i}^{k+h}), \text{angle.CNCa}(S_{i}^{k+h})), \ldots$

dihedral.CaCNCa(S_{i}^{k+h}, Local.Ca(S_{k,h}^{2h-1}), Local.C(S_{k,h}^{2h-1}), Local.N(S_{k,h}^{2h-1}))

$C(S_{\frac{2h+1}{2},h+1}) \leftarrow \text{FindXY ZWithDihedral}(\text{bond.CaC}(S_{i}^{k+h}), \text{angle.CaCa}(S_{i}^{k+h})), \ldots$

dihedral.CaCNCa(S_{i}^{k+h}, Local.Ca(S_{k,h}^{2h-1}), Local.C(S_{k,h}^{2h-1}), Local.N(S_{k,h}^{2h-1}))

$O(S_{\frac{2h+1}{2},h+1}) \leftarrow \text{FindXY ZWithDihedral}(\text{bond.CO}(S_{i}^{k+h}), \text{angle.CaCO}(S_{i}^{k+h})), \ldots$

dihedral.NCaCO(S_{i}^{k+h}, Local.N(S_{k,h}^{2h-1}), Local.Ca(S_{k,h}^{2h-1}), Local.C(S_{k,h}^{2h-1}))
place the last residue correctly. Doing the same operation for all merges in all tree levels, algorithm 4 returns all transformation operations which are calculated and stored in each merge. For each merge the transformation operation is saved at the $S_{k+1,h}^0$ index of the global transformation operation, this index denotes the first residue of the merge in segment $S$ at level $h$. Fig. 3.10 represents how we store this transformation operation. Calculating the transformation operation is performed by calling the FindTransformation function.

Parallel Merge  As discussed, the merge design is similar to a reduction [26]. So at level $h$ there is a potential to run $\lceil \frac{n}{2^h} \rceil$ threads simultaneously. The number of potential threads reduces as $h$ increases, i.e., moving towards the root of the tree.

3.2.3 Global Cartesian Coordinates

By calculating the transformation matrices the global Cartesian coordinates can be found using algorithm 5. Each residue with index $i$ will be merged into the global coordinate system at the tree level. When placing the Cartesian coordinate system of the residue $i$ it is important to find correct indices of transformation matrices and multiply them to place the residue in its exact position. At each level of the tree we should find the segment number $i$ belongs to, then uses the index of the first residue of the segment to find the transformation matrix, denoted $S_{k+1,h}^0$. As an example, Fig. 3.11 it is shown how to traverse the tree for 8 number of residue.

Parallel global Cartesian coordinates  Performing the same transformation operation in algorithm 5 for all the $n$ residues of the backbone, the problem has the potential to run with $n$ threads simultaneously. Because threads with lower indices merge to the global coordinate system sooner and become idle, the design have a potential to be investigated for more optimization. Algorithm 6 represents the invocation of all the discussed functions within our design.

3.3 Summary

As discussed in section 3.1 the design of Cartesian to internal conversion is straightforward and has high parallelization potential. Section 3.2 explained the challenges that we encounter for parallelizing the reverse conversion. Our new design makes parallelization more promising. The next chapter will discuss our experiments and results based on these implementations.
Algorithm 4 Merge Cartesian 2

procedure MERGE_CARTESIAN 2( n, Local)
    for h ← 0 to \( h_{\text{max}} - 1 \); h++ do
        for \( S_{k,h} \) ← \( S_{0,h} \) to \( S_{\left\lfloor \frac{n}{2^h} \right\rfloor ,h} \); \( k = k+2 \) do
            \[
            \begin{align*}
            \text{Local.}N(S_{\frac{n}{2^h}}^{2h}) & \leftarrow \text{FindXYZWithDihedral}(\text{bond.CN}(S_{k+1,h}), \ldots, \\
            \text{angle.C}_\alpha \text{CN}(S_{k+1,h}), \text{dihedral.NC}_\alpha \text{CN}(S_{k+1,h}), \text{Local.}N(S_{k+1,h}^{2h-1}), \text{Local.C}_\alpha (S_{k+1,h}^{2h-1}), \\
            \text{Local.C}(S_{k+1,h}^{2h-1})) \\
            \text{Local.C}_\alpha (S_{\frac{n}{2^h}}^{2h+1}) & \leftarrow \text{FindXYZWithDihedral}(\text{bond.NCA}(S_{k+1,h}), \ldots, \\
            \text{angle.CNC}_\alpha (S_{k+1,h}), \text{dihedral.C}_\alpha \text{CNC}(S_{k+1,h}), \text{Local.C}_\alpha (S_{k+1,h}^{2h-1}), \text{Local.C}(S_{k+1,h}^{2h-1}), \\
            \text{Local.N}(S_{k+1,h}^{2h-1})) \\
            \text{Local.C}(S_{\frac{n}{2^h}}^{2h+1}) & \leftarrow \text{FindXYZWithDihedral}(\text{bond.C}_\alpha \text{C}(S_{k+1,h}), \ldots, \\
            \text{angle.NC}_\alpha \text{C}(S_{k+1,h}), \text{dihedral.CNC}_\alpha \text{C}(S_{k+1,h}), \text{Local.C}(S_{k+1,h}^{2h-1}), \text{Local.N}(S_{k+1,h}^{2h-1}), \\
            \text{Local.C}_\alpha (S_{k+1,h}^{2h-1})) \\
            \text{Transformation}(S_{k+1,h}^{2h}) & \leftarrow \text{findTransformation}(S_{k+1,h}^{2h}), \text{CA}(S_{k+1,h}^{2h}), \text{C}(S_{k+1,h}^{2h})) \\
            \text{Local.O}(S_{\frac{n}{2^h}}^{2h+1}) & \leftarrow \text{Transformation}(S_{k+1,h}^{2h}) \ast \text{Local.O}(S_{k+1,h}^{2h}) \\
            \text{Local.N}(S_{\frac{n}{2^h}}^{2h+1}) & \leftarrow \text{Transformation}(S_{k+1,h}^{2h}) \ast \text{Local.N}(S_{k+1,h}^{2h-1}) \\
            \text{Local.C}_\alpha (S_{\frac{n}{2^h}}^{2h+1}) & \leftarrow \text{Transformation}(S_{k+1,h}^{2h}) \ast \text{Local.C}_\alpha (S_{k+1,h}^{2h-1}) \\
            \text{Local.C}(S_{\frac{n}{2^h}}^{2h+1}) & \leftarrow \text{Transformation}(S_{k+1,h}^{2h}) \ast \text{Local.C}(S_{k+1,h}^{2h-1}) \\
            \text{Local.O}(S_{\frac{n}{2^h}}^{2h+1}) & \leftarrow \text{Transformation}(S_{k+1,h}^{2h}) \ast \text{Local.O}(S_{k+1,h}^{2h-1}) \\
            \end{align*}
            \]
        return Transformation
CHAPTER 3. METHODOLOGY AND DESIGN

Algorithm 5 Cartesian Rebuild

procedure \textsc{Cartesian Rebuild}(n, Transformation, Local)

for \(i = 0\) to \(n-1\); \(i++\) do

for \(h \leftarrow 0\) to \(<\log_2(i)\); \(h++\) do

\[
\frac{k}{2} = \left\lfloor \frac{i}{2^{h+1}} \right\rfloor 
\]

Local.\(N(i) \leftarrow Transformation(S_{k+1,h}^0) \ast Local.\(N(i)\)

Local.\(C_a(i) \leftarrow Transformation(S_{k+1,h}^0) \ast Local.\(C_a(i)\)

Local.\(C(i) \leftarrow Transformation(S_{k+1,h}^0) \ast Local.\(C(i)\)

Local.\(O(i) \leftarrow Transformation(S_{k+1,h}^0) \ast Local.\(O(i)\)

\[
\text{Global}(i) \leftarrow \text{Local}(i)
\]

return \text{Global}(i)

Algorithm 6 Internal to Cartesian Coordinates 2

procedure \textsc{Internal to Cartesian Coordinate 2}(Protein)

\(n \leftarrow \text{Number of Backbone Residues}\)

for \(i = 0\) to \(n-1\) do

Local \(\leftarrow \text{Local Cartesian Coordinate}(i)\)

Transformation \(\leftarrow \text{Merge Cartesian 2}(n, \text{Local})\)

\text{Cartesian Rebuild}(n, \text{Transformation, Local})

Figure 3.7: Merging local coordinates using residue representation for 8 protein residues
CHAPTER 3. METHODOLOGY AND DESIGN

Figure 3.8: Merging local coordinates using segment representation for 8 protein residues

Figure 3.9: Merge in level \( h \) and \( h + 1 \) using notation
CHAPTER 3. METHODOLOGY AND DESIGN

Figure 3.10: Transformation operation storing process

Figure 3.11: Pattern used to accumulate transformation operation and calculate global Cartesian coordinates for each residue
Chapter 4

Experiments and Results

In this chapter, we present our implementation for both coordinate conversions. We use several different input examples and evaluate their accuracy and acceleration. In both directions, parallel implementation on a GPU results in considerable acceleration.

4.1 Cartesian to Internal Coordinates

The coordinate conversion is implemented in MATLAB, MATLAB PCT, C, C with OpenMP constructs, and CUDA-C. We evaluate our implementation on two types of architecture: (i) CPU - Intel Xeon E2620 Sandy Bridge processor with 6 cores and two way hyperthreading, and (ii) GPU - NVIDIA Tesla C2075, with 448 cores and 14 streaming processors. The Tesla GPU has a maximum thread block size of \(1024 \times 1024 \times 64\) and grid size \(65535 \times 65535\) \([28]\). Our implementation is tested with two different protein structures and trajectory files. In both cases, additional atoms from water are included because the proteins are in solution. The first one is a tripeptide (3 amino acids) with 2443 atoms; its trajectories are simulated in 1000 time frames. The second file is the lysozyme protein \([29]\) with 17566 atoms and its trajectories are given in 2210 time frames. The program has four outputs, each representing one internal coordinate of the input protein in a time frame. The dimension of the output is the size of internal coordinates multiplied by the number of time frames. The exact number of bonds, angles and improper and proper dihedrals for each file and their output size are given in Table 4.1. Note that all results are for end-to-end processing and include data transfer times. Because the trajectories are used by all the kernels the transfer time of that data is included in the total computation time.
As shown in Tables 4.2 and 4.3, the MATLAB implementation is the slowest. Using MATLAB PCT as a multithreaded version with pool size 12 improves the runtime by a factor of 1.4. The single threaded C version is considerably faster (200x) than MATLAB. The multithreaded C version using OpenMP with 12 threads demonstrates around 3x to 5x speedup compared to serial C; its run time is similar to CUDA-C without streaming. The advantage of OpenMP is its ease of use. However, since the target architecture, the CPU, has a limited number of threads, as the data gets larger its performance falls behind compared to GPUs which have many more cores and are designed for large problem size and high throughput. The fastest implementation is using CUDA-C with data streaming. Because the result is calculated while data is being transferred, we see an additional 3x speedup over CUDA-C without data streaming, and a total speed-up of 13x-20x (depending on the size of the protein) compared with sequential C. We also compiled CUDA kernels and called the .ptx file from our MATLAB implementation. The CUDA-MATLAB implementation is not as efficient as the CUDA-C version but it is a good choice for those who want to take advantage of GPU processing while programming in MATLAB. Fig. 4.1 and Fig. 4.2 summarize the run time results for both files. Note that the y axis is a logarithmic scale; speedups from the original version are substantial.
Figure 4.2: Total time (ms) for lysozyme

Table 4.1: Size of input and output for lysozyme and tripeptide

<table>
<thead>
<tr>
<th></th>
<th>Tripeptide</th>
<th>Lysozyme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of atoms</td>
<td>2443</td>
<td>17566</td>
</tr>
<tr>
<td>Number of frames</td>
<td>1000</td>
<td>2210</td>
</tr>
<tr>
<td>Number of bonds</td>
<td>1635</td>
<td>12185</td>
</tr>
<tr>
<td>Number of angles</td>
<td>843</td>
<td>7702</td>
</tr>
<tr>
<td>Number of proper dihedrals</td>
<td>41</td>
<td>3293</td>
</tr>
<tr>
<td>Number of improper dihedrals</td>
<td>4</td>
<td>204</td>
</tr>
<tr>
<td>Size of bond distance output</td>
<td>$1635 \times 1000$</td>
<td>$12185 \times 2210$</td>
</tr>
<tr>
<td>Size of angle output</td>
<td>$843 \times 1000$</td>
<td>$7702 \times 2210$</td>
</tr>
<tr>
<td>Size of dihedral output(proper)</td>
<td>$41 \times 1000$</td>
<td>$3293 \times 2210$</td>
</tr>
<tr>
<td>Size of dihedral output(improper)</td>
<td>$4 \times 1000$</td>
<td>$204 \times 2210$</td>
</tr>
</tbody>
</table>
### CHAPTER 4. EXPERIMENTS AND RESULTS

#### Table 4.2: Forward conversion timing (ms) results for tripeptide

<table>
<thead>
<tr>
<th></th>
<th>MATLAB</th>
<th>MPCT</th>
<th>C</th>
<th>OpenMP</th>
<th>CUDA-C</th>
<th>CUDA-C streaming</th>
<th>CUDA-MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetBond distance</td>
<td>4344.1</td>
<td>3325.3</td>
<td>128.0</td>
<td>52.0</td>
<td>17.1</td>
<td></td>
<td>42.0</td>
</tr>
<tr>
<td>GetAngle</td>
<td>30820.0</td>
<td>22493.5</td>
<td>242.0</td>
<td>76.0</td>
<td>24.0</td>
<td></td>
<td>44.6</td>
</tr>
<tr>
<td>GetDihedral (proper)</td>
<td>10984.0</td>
<td>7885.8</td>
<td>10.0</td>
<td>3.0</td>
<td>0.2</td>
<td></td>
<td>18.8</td>
</tr>
<tr>
<td>GetDihedral (improper)</td>
<td>1390.0</td>
<td>1067.9</td>
<td>1.0</td>
<td>0.4</td>
<td>0.2</td>
<td></td>
<td>12.7</td>
</tr>
<tr>
<td>Total computation</td>
<td>66812.0</td>
<td>48449.0</td>
<td>370.0</td>
<td>134.0</td>
<td>66.4</td>
<td>29.9</td>
<td>114.4</td>
</tr>
</tbody>
</table>

#### Table 4.3: Forward conversion timing (ms) results for lysozyme

<table>
<thead>
<tr>
<th></th>
<th>MATLAB</th>
<th>MPCT</th>
<th>C</th>
<th>OpenMP</th>
<th>CUDA-C</th>
<th>CUDA-C streaming</th>
<th>CUDA-MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetBond distance</td>
<td>358009.5</td>
<td>255693.7</td>
<td>2097.0</td>
<td>458.0</td>
<td>450.6</td>
<td></td>
<td>437.2</td>
</tr>
<tr>
<td>GetAngle</td>
<td>757829.3</td>
<td>549829.6</td>
<td>4825.0</td>
<td>734.0</td>
<td>653.1</td>
<td></td>
<td>383.7</td>
</tr>
<tr>
<td>GetDihedral (proper)</td>
<td>728004.0</td>
<td>530758.2</td>
<td>1459.0</td>
<td>261.0</td>
<td>76.7</td>
<td></td>
<td>149.7</td>
</tr>
<tr>
<td>GetDihedral (improper)</td>
<td>43080.7</td>
<td>27280.3</td>
<td>103.0</td>
<td>22.0</td>
<td>49.7</td>
<td></td>
<td>20.4</td>
</tr>
<tr>
<td>Total computation</td>
<td>1879300.0</td>
<td>1547926.8</td>
<td>8484.0</td>
<td>1574.0</td>
<td>1643.6</td>
<td>439.1</td>
<td>1431.0</td>
</tr>
</tbody>
</table>
4.2 Internal to Cartesian Coordinate Conversion

4.2.1 Experimental Setup

For the reverse conversion, the GPU and CPU architectures used are the same as for the forward conversion: (i) CPU - Intel Xeon E2620 Sandy Bridge processor, and (ii) GPU - NVIDIA Tesla C2075. The regular internal to Cartesian coordinate conversion uses dihedral angles to walk over the protein chain serially. This computes the Cartesian coordinates using those bonds and angles that form the dihedral. We call this version sequential-by-dihedral conversion. Our method uses the reduction technique (see chapter 3) to exploit parallelism. We compare sequential-by-dihedral conversion to our new design. The dihedral reverse version serial code and the reduction serial code are both run on the CPU; the reduction code is run using multithreading on the GPU.

4.2.2 Input data

We assume that the protein backbone has no improper dihedral angles, and another assumption is that there are no loops inside the backbone chain. This assumption has 2 exceptions in real proteins: (I) proteins containing a proline amino acid (see Fig.4.3) in their structures and (II) proteins with disulfide bond \(^\text{30}\) (see Fig.4.4) in their backbone. Thus, lactate dehydrogenase (LDH) \(^\text{31}\) is a good candidate because it has neither of these. LDH is an enzyme found in animals, plants, and is of medical significance because it is found extensively in body tissue, such as blood cells and heart muscle. It is also released during tissue damage, so it is a marker of common injuries and disease.\(^\text{32}\) This protein has 8 chains (A to H). We preprocess the PDB file downloaded from the protein data bank \(^\text{33}\) to split its chains. For our experiments we use the first chain, A, which has 331 residues. The input to the reverse translation program is protein internal coordinates in different time frames. Since right now we do not have real experimental data, we generate internal coordinates using Visual Molecular Dynamics (VMD). To test our implementation, we produce the internal coordinates only for one time frame. The output of the reverse conversion is the Cartesian coordinates which can be represented in a text or PDB file. To produce the PDB output file we use MATLAB read/write PDB file.\(^\text{34}\) The PDB file rounds the Cartesian coordinate to two decimal places. Table 4.4 shows the size of input for one time frame. The output data includes the \((x, y, z)\) coordinates of all the atoms on the backbone structure (LDH chain A has 1324 atoms).
CHAPTER 4. EXPERIMENTS AND RESULTS

Figure 4.3: Proline amino acid [6]

Figure 4.4: Disulfide bond [1]

Table 4.4: Size of LDH for reverse conversion

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of atoms</td>
<td>5224</td>
</tr>
<tr>
<td>Number of residues</td>
<td>331</td>
</tr>
<tr>
<td>Number of backbone atoms</td>
<td>1324</td>
</tr>
<tr>
<td>Number of backbone bonds</td>
<td>1323</td>
</tr>
<tr>
<td>Number of backbone angle</td>
<td>1322</td>
</tr>
<tr>
<td>Number of backbone proper dihedral</td>
<td>1321</td>
</tr>
</tbody>
</table>
CHAPTER 4. EXPERIMENTS AND RESULTS

Table 4.5: Reverse conversion timing (ms) results

<table>
<thead>
<tr>
<th>Reduction reverse</th>
<th>CUDA-C</th>
<th>C-Serial</th>
<th>Sequential-by-dihedral</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.37</td>
<td>6.78</td>
<td>3.34</td>
</tr>
</tbody>
</table>

Table 4.6: Reverse conversion CUDA-C kernel and memory copy time (ms)

<table>
<thead>
<tr>
<th>Name of kernel</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Cartesian coordinate</td>
<td>0.056</td>
</tr>
<tr>
<td>Merge reduction</td>
<td>0.160</td>
</tr>
<tr>
<td>Cartesian coordinates rebuild</td>
<td>0.077</td>
</tr>
<tr>
<td>Copy data to device</td>
<td>0.066</td>
</tr>
<tr>
<td>Copy data to host</td>
<td>0.053</td>
</tr>
</tbody>
</table>

4.2.3 Timing Results

We use the LDH protein with 331 residues as an input to our new method of internal-to-Cartesian coordinate conversion. It is known to be more efficient to launch threads with the same number as a power of two. Since $2^8 < 331 < 2^9$, we can launch either 256 threads and calculate the rest of chain on the CPU, or launch 512 threads and pad the extra threads to zeros. In this experiment we chose the first approach. We compare the output of our method with a dihedral reverse conversion and both outputs were the same. Table 4.5 shows our method accelerates LDH coordinate conversion by approximately 10x compared to the dihedral serial implementation, and by 18x compared to the serial implementation of our reduction algorithm. We think because the reduction technique has a different looping property, its serial implementation is slower than the dihedral serial implementation. Table 4.6 shows the break down of CUDA-C total time into data copying time between host and device and the time consumed by each kernel.

To evaluate the performance of our parallel method, we designed a simple structure containing different number of alanine residue (see Fig. 4.5), and use them as an input to the reverse program. Table 4.7 and Fig. 4.6 present the timing results. As the size of protein grows our parallel method indicates more speedup compared to the dihedral reverse serial implementation (from 5x to 12x speedup). Also note that for the serial C implementation, the timing difference between the reduction C-serial and the dihedral reverse C-serial implementation reduces as protein size increases.
CHAPTER 4. EXPERIMENTS AND RESULTS

Table 4.7: Reverse conversion timing (ms) for a simple alanine chain

<table>
<thead>
<tr>
<th>Number of alanine residues</th>
<th>Reduction reverse</th>
<th>Sequential-by-dihedral</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CUDA-C</td>
<td>C-Serial</td>
</tr>
<tr>
<td>100</td>
<td>0.29</td>
<td>2.63</td>
</tr>
<tr>
<td>500</td>
<td>0.6</td>
<td>9.59</td>
</tr>
<tr>
<td>1000</td>
<td>0.84</td>
<td>12.21</td>
</tr>
</tbody>
</table>

Figure 4.5: Alanine amino acid

Figure 4.6: Total time (ms) for reverse conversion
CHAPTER 4. EXPERIMENTS AND RESULTS

4.3 Summary

In both directions of forward and reverse translation, GPU parallelization helps us to implement a rapid, accurate, and efficient algorithm. The process of Cartesian-to-internal coordinates conversion, we accomplish 13x to 20x speedup, depending on protein size and the number of time frames. In the reverse conversion, the parallel version runs 18x faster compared to the similar serial-C version for LDH protein and around 10x-12x faster than the dihedral reverse C implementation. Right now we only perform backbone reverse conversion. In the next chapter, we discuss our future plans to handle the side chains of the protein.
Chapter 5

Conclusion and Future Work

In this chapter we focus on the future work and an overview of the presented results.

5.1 Future Work

Our current implementation in the forward direction produces significant acceleration. In the future we would like to be able to handle much larger proteins with thousands of atoms in more time frames. To handle these, we need to investigate using multiple GPUs. In addition to forward translation we investigated the reverse direction: internal coordinates to Cartesian. The dependency caused by relative coordinates makes the parallelization of this conversion more challenging.

Currently we perform this conversion for backbone atoms only but, once the rigid body position of the molecular structure is specified, the next step is to add side chains to the protein backbone. In this step, we can assume each side chain is independent of the other side chains and depends only on the position of its bonded atom in the backbone. Therefore, we can compute their positions in parallel.

We have not yet demonstrated the reverse conversion of backbone structures in which bonds between atoms form a loop, like proteins having proline amino acids or disulfide bonds. Thus, another improvement to our design is to ensure that the algorithm can compute atoms’ position of these structures correctly.

Another interesting approach to the reverse conversion problem is to combine the naive merge design (chapter 3 algorithm 3) with the reduction merge and transformation matrix design (chapter 3 algorithm 4). In other words, for the first levels of the tree, we use the naive implementation and at level $h_i$ we switch to the other method. This combined approached might work better for
CHAPTER 5. CONCLUSION AND FUTURE WORK

larger structures. Finding the best $h_i$ to optimize the timing is also important. We plan to investigate this in the future.

5.2 Conclusions

We have presented the conversion of Cartesian to internal coordinates to represent large proteins. Our CUDA-C implementation, using data streaming and overlapping computation outperforms other parallel versions. The results show that the CUDA code takes approximately 30 milliseconds for a protein model with 2,443 atoms, and 440 milliseconds for a protein with 17,566 atoms, which is approximately 20 times faster than the single threaded C implementation and around 5 times faster than the multithreaded C with OpenMP version. In internal to Cartesian coordinate conversion, we investigate how the design of an algorithm can have a direct effect on its parallelization potential. The common implementation of this conversion is inherently serial because of all the dependencies along the molecular chain, but our new method implemented with CUDA-C accelerates the computation considerably.

Our ultimate goal is to accelerate conversion between these two representations so a scientist can choose either representation based on the tool they wish to use, and not be concerned with computational speed. Being able to handle larger proteins and adding side chains for reverse conversion will pave the way for us to accomplish our goal. The speedup of our implementation in both directions using CUDA-C shows the problem is suitable for parallelizing on GPU hardware.
Bibliography


[29] Protein Data Bank, “Lysozyme(1HEL),” http://www.rcsb.org/-pdb/explore/explore.do?structureId=1HEL


[31] Protein Data Bank, “Lactate dehydrogenase (1i10),” http://www.rcsb.org/pdb/explore/explore.do?structureId=1i10


[33] “Protein Data Bank,” http://www.rcsb.org/