University of Arkansas, Fayetteville
ScholarWorks@UARK

Theses and Dissertations

5-2017

Molecular Dynamics Simulations of DNA-Functionalized Nanoparticle Building Blocks on GPUs

Tyler Landon Fochtman
University of Arkansas, Fayetteville

Follow this and additional works at: http://scholarworks.uark.edu/etd

Part of the Nanoscience and Nanotechnology Commons, and the Numerical Analysis and Scientific Computing Commons

Recommended Citation
Fochtman, Tyler Landon, "Molecular Dynamics Simulations of DNA-Functionalized Nanoparticle Building Blocks on GPUs" (2017). *Theses and Dissertations*. 1926.
http://scholarworks.uark.edu/etd/1926

This Thesis is brought to you for free and open access by ScholarWorks@UARK. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu, ccmiddle@uark.edu.

Molecular Dynamics Simulations of DNA-Functionalized
Nanoparticle Building Blocks on GPUs

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

by

Tyler Fochtman
University of Arkansas
Bachelor of Arts in English Creative Writing, 2010

May 2017
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

Dr. Matthew Patitz
Thesis Director

Dr. Gordon Beavers
Committee Member

Dr. Jacob Hendricks
Ex-Officio

Professor Jin-woo Kim
Committee Member

Dr. Qinghua Li
Committee Member

Abstract

This thesis discusses massively parallel molecular dynamics simulations of nBLOCKs using graphical processing units. nBLOCKs are nanoscale building blocks composed of gold nanoparticles functionalized with single-stranded DNA molecules. To explore greater simulation time scales we implement our nBLOCK computational model as an extension to the coarse grain molecular simulator oxDNA. oxDNA is parameterized to match the thermodynamics of DNA strand hybridization as well as the mechanics of single stranded DNA and double stranded DNA. In addition to an in-depth review of our implementation details we also provide results of the model validation and performance tests. These validation and performance tests are comprised of over a hundred separate simulations spanning in simulation length from one thousand to ten million times steps and with simulation sizes ranging from 16 to 27832 particles. Together these tests show the ability of our implementation to handle the full range of basic nBLOCK topologies in a diverse set of conditions.

A selection of the utilities developed during the course of this thesis are also discussed. We provide descriptions of the scripting utilities which support nBLOCK assembly generation, simulation, and analysis.

Acknowledgments

I would like to acknowledge my mother and father for the encouragement and support they provided me during my studies. I would also like to acknowledge Dr. Jacob Hendricks for being a mentor and a friend. Finally, I would like to acknowledge my adviser Dr. Matthew Patitz for his patience and guidance and for the countless suggestions and improvements he made to this thesis.

Contents

1 Introduction **1**
1.1 Outline . . . 2

2 Background **3**
2.1 The nBLOCK Model . . . 3
2.1.1 Predecessors . . . 3
2.1.2 nBLOCK Synthesis . . . 3
2.1.3 nBLOCK Simulation . . . 5
2.2 oxDNA . . . 6
2.2.1 Coarse Grain Simulation . . . 6
2.2.2 Simulation Parameters . . . 8
2.2.3 Description of oxDNA Code Base . . . 8
2.3 NVIDIA GPU Programming Model and Architecture . . . 11
2.3.1 Programming Model . . . 11
2.3.2 Memory Types on the Kepler Architecture . . . 12
2.3.3 Details of Memory Access for Addressable Memory . . . 12

3 Implementation **14**
3.1 Implementation Constraints . . . 15
3.2 Implementation Strategy . . . 16
3.2.1 Simulation Step . . . 19
3.2.2 Accounting for Particle Mass . . . 21
3.2.3 oxDNA-nBLOCK Support Scripts . . . 23
3.2.4 Mixed Precision . . . 25

4 Results **26**
4.1 Simulation Techniques . . . 26
4.2 Validation Tests . . . 27
4.2.1 Comparing Kinetic Energy . . . 27
4.2.2 Probability Plot Results Analysis . . . 28
4.2.3 Comparing Potential Energy . . . 32
4.3 Performance Tests . . . 35
4.3.1 Dataset Preparation . . . 35
4.3.2 Performance Tests Results Analysis . . . 37

5 Conclusion **40**

References **42**

List of Figures

2.1 **Complementary DNA base pairs.** On the left Adenine (A) and Thymine (T). On the right Guanine (G) and Cytosine (C). The dotted lines represent hydrogen bonds and the solid lines represent covalent bonds. 4

2.2 **nBLOCK Visualization.** We extended Python scripts within oxDNA utilities for our nBLOCK visualization within UCSF Chimera [14]. 5

2.3 **Potentials and components of an nBLOCK in oxDNA.** 6

2.4 **CUDA code sample.** 13

3.1 **Host to device translation of particle data.** Particle data is stored in objects on the host side. On the device, particle data is split into arrays by member value label. 16

3.2 **Stably reorder particle data nanoparticle first.** **(a)** Shows pointers between bonded neighbors. These points define the NP-DNA topology. **(b)** Depiction of particle data as a collection of nanoparticle and nucleotide objects before and after reordering. **(c)** This sorting must preserve the order of nucleotides within a strand. . 17

3.3 **Operations on Bonded Neighbor Pointers.** **(a)** Depiction of pointers immediately after ordering particle data nanoparticle first. **(b)** Correction of pointers such that original topology is preserved. **(c)** Invalidate bonds pointing from nucleotides to nanoparticles while preserving nanoparticle to nucleotide bonded neighbor pointers. **(d)** Decrement bonded nucleotide bonds by the number of nanoparticles before passing data to CUDA DNA interaction class. 21

4.1 **Sparese CPU Maxwell probability plots.** Each of these probability plots represents random samples of 10^2 points taken from the full KE distribution gathered during the simulation of a 200 particle nBLOCK duplex assembly. The complete plot of all 10^3 KE values is shown in Figure 4.3 **(a)**. 29

4.2 **Sparese GPU Maxwell probability plots.** Each of these probability plots represents random samples of 10^2 points taken from the full *KE* distribution gathered during the simulation of a 200 particle nBLOCK duplex assembly. The complete plot of all 10^3 *KE* values is shown in Figure 4.3 **(b)**. 30

4.3 **Maxwell probability plots. Figure Explanation.** The Y-axis is scaled to match the distribution of the gathered *KE* measurements. The X-axis shows the values corresponding to quantiles of the Maxwell distribution. The blue dots in **(a)** and **(b)** represent one thousand *KE* samples taken at an interval of one thousand time steps from the simulation of a two-hundred particle diatomic nBLOCK assembly. The straight red line is plotted through the center of mass for this points. Because there are many overlapping points in these plots we further quantify the result using the square root of the coefficient of determination *r* which is shown in each figure. An *r* value of 1.0 would indicate that the variance in the *KE* samples is completely predictable by the Maxwell distribution. As such the reported *r* values demonstrate the *KE* distribution of our nBLOCK simulations are a good fit the Maxwell distribution. 33

4.4 **Potential energy plots. Figure explanation.** The solid black line represents CPU *PE* over time. The green line represents the GPU *PE* over time. The three horizontal dashed lines represent the CPU mean *PE* and $\pm$ 5 standard errors of the mean, while the solid blue line is the GPU mean *PE*. **Column one. NP-PolyA-15** Plots the results from ten *PE* samples taken at 10^2 time steps intervals. **Column two. Diatomic-nBLOCK** Same simulation duration and sampling interval as column one but taken from a simulation of a diatomic nBLOCK assembly with fifty total particles. **Column three, upper-half.** The same nBLOCK topology as column two. One hundred samples with interval of 10^3. **Column three, lower-half.** Diatomic-nBLOCK assembly with one hundred particles. One thousand samples with interval of 10^3. 36

4.5 **Example initial configurations along 3D grid.** **a** 126 nt and 1 NP **b** 1008 nt and 8 NP **c** 3402 nt and 27 NP **d** 8064 nt and 64 NP 37

4.6 **Performance Tests. Figure explanation.** All X-axes represent the number of particles N. All Y-axes represent the average time step length, t in milliseconds. **Figure (a)** the left Y-axis represents t scaled to the CPU timings. The right Y-axis represents t scaled to the GPU timings. Floating precision and mixed precision calculations share the same scale, although pure floating point precision simulations enjoy a considerable speed-up over the mixed point precision. Notice that the spikes are mirrored across both GPU implementations and to a lesser extent the CPU implementation. **Figures (b) and (c)** display log-log plots for the average CPU and GPU timings respectively. In these plots average milliseconds per time step, the number of particles in the simulation, and the nBLOCK topology are accounted for. We can also see that the intermittent spikes present in the plots of **(a)** can occur as we move along the X-axis from a simulation with a greater number of arms to a simulation with relatively fewer arms. **Figure (d)** displays a log-log plot of CPU and GPU timings. The CPU timings are represented by the marks which have no fill. Otherwise the markings follow the same legends of plots **(b, c)**. We can see the GPU begins to outperform the CPU at approximately eighty-five particles. 39

Chapter 1

Introduction

Molecular simulation is a way to study the physical interactions of atoms or molecules over time. This thesis discusses techniques for simulating the molecular dynamics of deoxyribonucleic acid (DNA) functionalized nanoparticles using software which utilizes graphical processing units (GPU). In particular, with our molecular simulations we wish to model the dynamics of DNA functionalized nanoparticles that are topologically consistent with the nBLOCK model [7]. The experimental results obtained by the nBLOCK protocol have shown the ability to specify the spatial configuration of gold nanoparticles by first strategically designing and placing DNA strands on them which can be used to control their connectivity via DNA base pair complementarity.

To simulate the dynamics of nBLOCKs we implemented an extension to a coarse grain simulation package called oxDNA [3]. The simulation techniques discussed within this thesis can be used as a guide for adding additional functionality to the base oxDNA-nBLOCK extension or for modifying it to fit the constraints of similar nanoparticle-ssDNA models.

We can use our oxDNA-nBLOCK extension to perform experiments that are difficult or impossible to perform in the wet lab. After a simulation we can analyze particle trajectories for statistics on target properties of our assembly. In order to trust these statistics it is necessary to embed the characteristic properties of the chosen material into the simulator so that they can be observed. For example, the base oxDNA DNA model captures the mechanical properties of DNA that allow for ssDNA to be floppy and dsDNA to be rigid. The base model also captures the thermodynamic properties of hybridization at certain temperatures and strand disassociation or melting at a certain temperature. As is explained in [3] both of these targets were met by the oxDNA model by rigorously parameterizing the potentials in the model to experimental data. By extending the oxDNA model we are able to leverage this existing functionality.

This thesis also presents tests and their results for validating that the dynamics simulated by the GPU implementation match the CPU. In addition we also provide the results of performance tests.

Our performance tests cover simulations of nBLOCKs having from one to six anisotropic strand attachments. Each of the six nBLOCK strand topologies were grouped into simulations ranging in size from 22 to $27,432$ particles. We then show the efficiency of our implementation through comparisons between CPU and GPU simulation run times.

1.1 Outline

The remainder of this thesis is organized as follows. Background information is given in Chapter 2. Subsections of Chapter 2 include a more detailed review of the nBLOCK model, the oxDNA molecular simulator, and the NVIDIA CUDA programming model. Details of our nBLOCK GPU implementation can be found in Chapter 3. In particular we discuss how we are able to implement our massively parallel nBLOCK model while without editing the highly optimized existing CUDA DNA interaction class. We explain our implementation in terms of maintaining a data dependency graph between bound and unbound particles in simulation. We show how our CUDA nanoparticle backend, CUDA nBLOCK interaction class, and an adaptation of the standard Verlet-cell list maintain this graph. Results of validation and performance tests are reported in Chapter 4. Chapter 4 also includes a more detailed explanation of the Brownian thermostat and how it dictates test design and interpretation of simulation results. Concluding remarks can be found in Chapter 5.

Chapter 2

Background

2.1 The nBLOCK Model

The nBLOCK experimental protocol demonstrates a refinement to preceeding models by outlining an experimental design which results in the anisotropic functionalization of gold nanoparticles with ssDNA. This anisotropic functionalization allows for previously unprecedented levels of control of the placement of gold nanoparticles by first strategically designing and placing strands on them which can be used to control their combinations via DNA base pair complementarity.

2.1.1 Predecessors

In 1996 two papers were published in a single issue of Nature Letters that detailed separate experimental methodologies for attaching single stranded DNA to gold nanoparticles. The first of these papers was by Alivisatos et al. [1] and demonstrated an experimental protocol for the mono-functionalization of gold nanoparticles with DNA. They further demonstrated that mixing complimentary batches of these nanostructures resulted in the expected arrangement of the nanoparticles into one dimensional structures with spacing between nanoparticles determined by strand length. The second paper was by Mirkin et al. [8] demonstrated a separate process by which many strands of DNA could be anchored to a single gold nanoparticle. They showed that non-complimentary batches of these isotropically functionalized gold nanoparticles would self-assemble into colloidal aggregates once complimentary dsDNA linkers were added to the mixture.

2.1.2 nBLOCK Synthesis

An nBLOCK [7] is a gold nanoparticle (Au-NP) that has been functionalized with strands of deoxyribonucleic acid (DNA). DNA is a molecule that defines the development, functionality, and replication of all known living organisms. DNA is composed of four bases, adenine (A), guanine

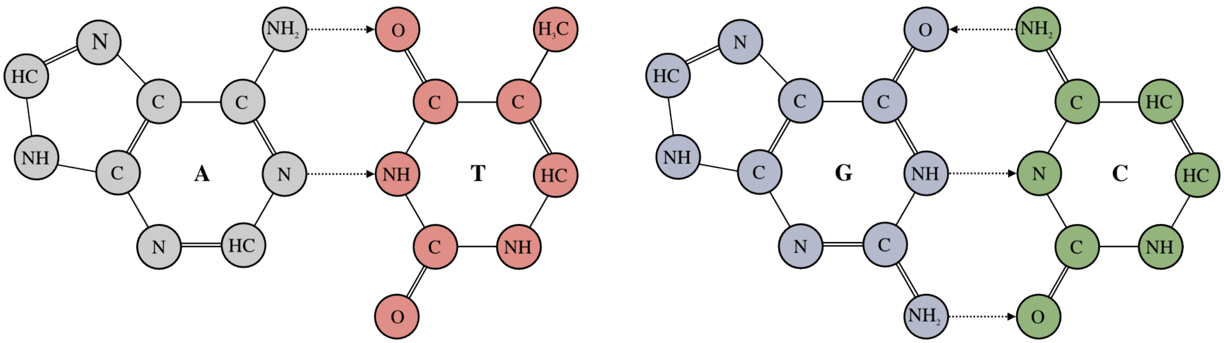


Figure 2.1: **Complementary DNA base pairs.** On the left Adenine (A) and Thymine (T). On the right Guanine (G) and Cytosine (C). The dotted lines represent hydrogen bonds and the solid lines represent covalent bonds.

(G), cytosine (C), and thiamine (T), a sugar (deoxyribose), and a phosphate group. A and T share a symmetric attraction as do G and C. Figure 2.1 gives a graphical representation of the nucleotides bound with their Watson-Crick compliment. The bond that forms between these pairs is a hydrogen bond that is about one hundred times weaker [2] than the covalent bonding that occurs between stacked bases in the backbone of single stranded DNA (ssDNA). Double stranded DNA (dsDNA) is considered anisotropic due to the weak hydrogen bonding between the strands at the core of the helix and the strong covalent bonds between the stacked bases. DNA is also anisotropic in the sense that hybridization of two strands only occurs in an anti-parallel manner.

The average diameter of the nanoparticles used in [7] was 2.83 ± 0.47 nanometers (nm). Using the nBLOCK experimental protocol the first step in the synthesis of NP-DNA is to prepare the nanoparticle surface with capping ligands. The nanoparticles are modified by two different ligand types which cause the AU-NP to have a net negative charge. The $5'$ ends of ssDNA are functionalized (chemically modified) so that they can bind to the ligands on the surface of the NP. These functionalized-ssDNA can then bound to the surface of the nanoparticle one at a time up to a maximum of six strands as shown in Figure 2.2.

After the initial DNA strand (S_0) has bound the remaining five strands bind sequentially with the repulsive electrostatic forces between strand backbones as well as the negatively charged NP surface guiding the placement of the strands towards the area on the NP surface with the minimum repulsion. As a consequence the odd indexed strands (S_1, S_3, S_5) will seek the antipodal point to the

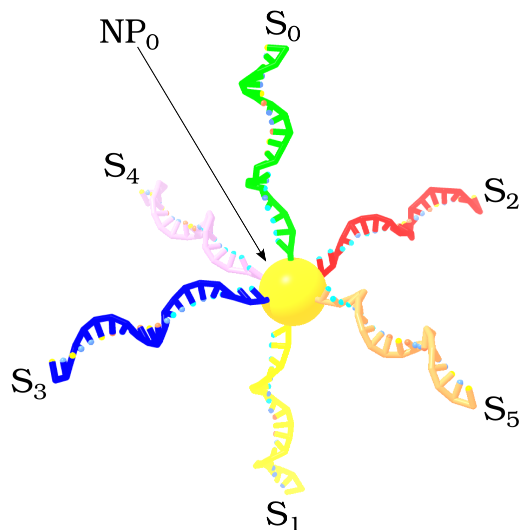


Figure 2.2: **nBLOCK Visualization.** We extended Python scripts within oxDNA utilities for our nBLOCK visualization within UCSF Chimera [14].

most recently bound strand. While the even index strands (S_2, S_4) bind to any point on the sphere that is equidistant from the previously attached strands.

2.1.3 nBLOCK Simulation

We chose to extend the oxDNA molecular simulator with a coarse grain description of the nBLOCK model. The goal of coarse grain molecular dynamics is to reduce the detail of the simulated components from the atomistic scale to a polyatomic scale. Our nBLOCK extension to oxDNA models gold nanoparticles as a single unit. In reality gold nanoparticles have many atomic components. In addition we do not explicitly model the capping ligands covering the gold nanoparticles or the chemically modified ends on the five prime ends of the ssDNA that are bound to these capping ligands. Instead we implicitly account for the covalent bond between nanoparticle and ssDNA with a modified version of the finitely extensible nonlinear elastic (FENE) spring potential. The isotropic nature of this FENE Spring potential allows nanoparticle and nucleotides to rotate independently of one another while maintaining an average distance interval.

By implementing the nBLOCK model as an extension to oxDNA we hope to capture all the work that has gone into modeling the dynamics of DNA interaction ([13], [20], [19]) for our own

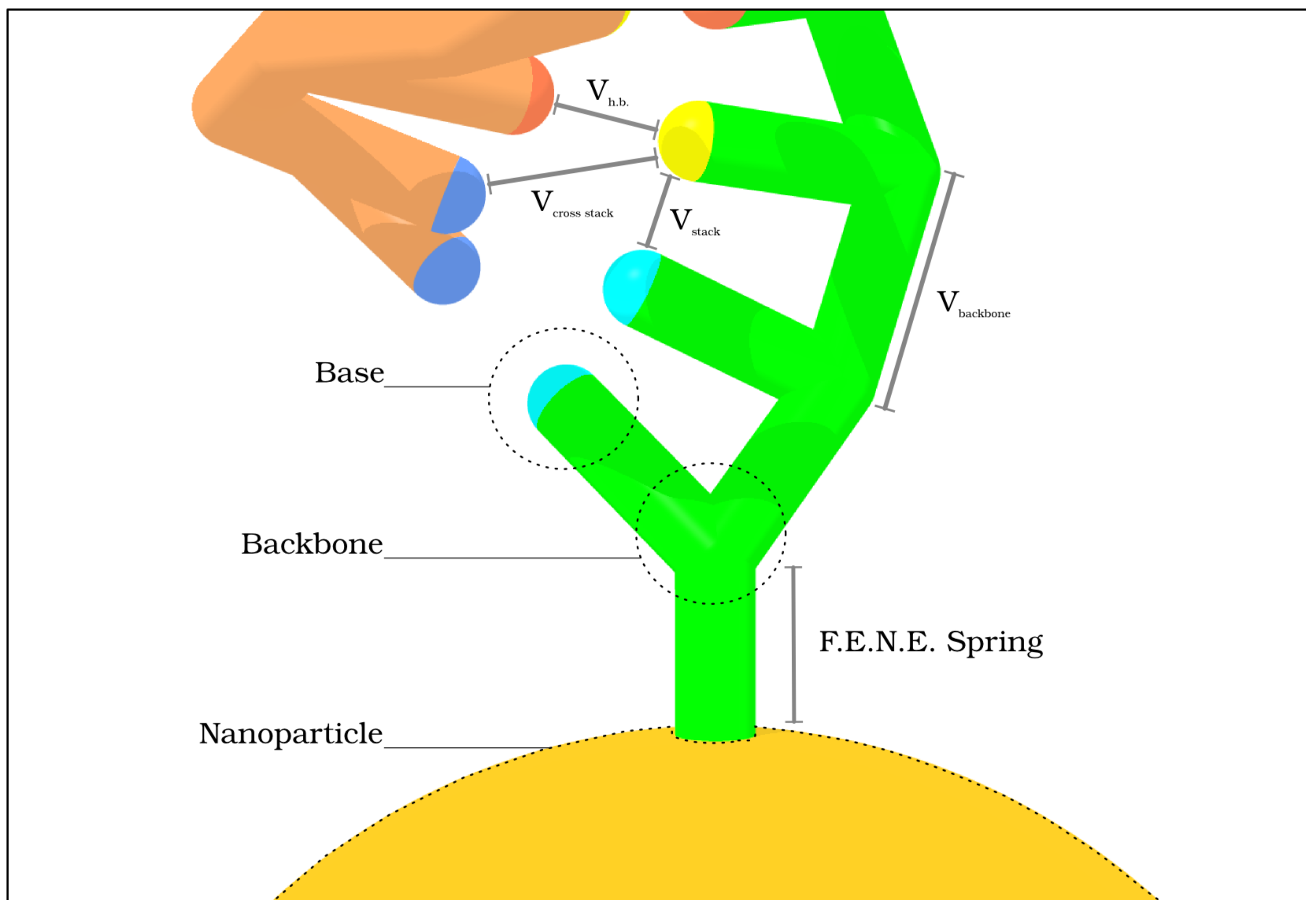


Figure 2.3: **Potentials and components of an nBLOCK in oxDNA.**

investigations into the dynamics of self-assembling nBLOCKs.

2.2 oxDNA

oxDNA [11] is a molecular dynamics simulator built around a unique coarse-grain model of DNA. This coarse grain description encorporates the geometric, thermodynamic, and mechanical properties of DNA which are considered relevant to DNA nanotechnology.

2.2.1 Coarse Grain Simulation

One of the principle differences between molecular dynamics simulators is the scale at which simulation occurs. An atomistic resolution simulation computes interatomic forces for the atoms in simulation. An atomistic simulation of an nBLOCK might include the ligands bound to the

surface of the NP and or model the nanoparticles as packed spheres of gold atoms. In addition it would also model an atomistic description of DNA molecules. As is noted in [15], the standard molecular dynamics time step scales linearly with the number of particles (N) in the simulation. As a result, the computational effort to perform 10^3 time steps for a system with 10^9 particles is the same as performing 10^9 time steps for a system with 10^3 particles. Each atom or molecule which doesn't need to be explicitly modeled to capture the dynamics of the target system increases the timescales accessible by that system. This is particularly important for many applications of DNA nanotechnology which rely on the rigidity of a dsDNA and the flexibility of a ssDNA as well as capturing the transition between them. For our purposes modeling the transition between these states for ssDNA anchored to gold nanoparticles is one of the principal motivations behind this work. The mechanical and thermodynamic model of oxDNA allows for these dual states to be simulated as well as the transitional states between them. oxDNA is able to simulate these transitions in a tractable amount of computation time due to a unique coarse-grained DNA model and accompanied force potentials that describe these dynamics.

In general coarse-grained models reduce the number of atoms in simulation by combining them into functional units. The atomic components of a molecule that have an effect on the properties we wish to measure are simulated at a higher resolution, while the atoms or groups of atoms that do not effect the simulation can be abstracted over. The coarse grain description of oxDNA models DNA strands at the scale of nucleotides. That is, although there many atoms in a nucleotide, oxDNA groups them into a single point and emulates their bulk description through a set of potentials. Figure 2.3 shows a graphical representation of these potentials between nucleotides [11] as well highlighting the main components of our coarse grain nBLOCK description as modeled in oxDNA. Within Figure 2.3 we have labeled the finitely extensible nonlinear elastic (FENE) spring potential between consecutive backbone sites as V_{backbone}, and the modified FENE spring between nucleotide backbones and the nanoparticle as FENE Spring. Additionally in Figure 2.3 the hydrogen bonding between Watson-Crick compliments is labeled as $V_{\text{h.b.}}$, the intra-strand stacking potential is labeled as V_{stack}, and the cross stacking potential is labeled as $V_{\text{cross stack}}$. The Lennard

Jones excluded volume potential between nucleotides and between nucleotides and nanoparticles is not shown.

2.2.2 Simulation Parameters

The oxDNA executable takes as input a single file which describes the scope of simulation. Informally we can group these parameters into four main categories: backend, interaction class, list, and IO. The backend parameters include flags for hardware initialization such as the specification of CPU or GPU, initialization seed, and the computational precision. In addition the backend also takes parameters that modulate the macroscopic state of the simulation environment. These parameters include a designation of temperature T, the integration time step δt, choice of thermostat, the interval at which thermalization occurs $N_{Newtonian}$, and multiple ways to designate the diffusion coefficient. The interaction class is in general specified by a single flag which is just the name of the interaction type. For nBLOCK simulation the value for this flag should be set to `NBLOCK`. Verlet-cell lists are the standard choice for particle non-bonded neighbor calculations. To increase the area around each particle that is checked for non-bonded neighbors users may designate a floating point value for the Verlet skin input flag. IO parameters in the input file include the relative or absolute file system paths for the initial configuration topology and trajectory files. Other IO parameters include specifying the interval (in simulation steps) that system energy profiles and particle trajectories are saved.

2.2.3 Description of oxDNA Code Base

Currently the oxDNA code base consists of over fifty thousand lines of code split between classes for particle data, particle lists, particle interactions, thermostats, logging, analysis, simulation backends, and a simulation manager. Other than utilities written in the Python programming language and CUDA code written for GPU enabled simulation, the standard programming language of oxDNA is C++. C++ is a typed language that offers an abstraction for grouping related data and functions called a `class`. In the following overview we use terminology associated with C++ and

call the data values associated with a class its "members."

The highest level component of the oxDNA code base is the simulation manager. This manager has a polymorphic member for data management that also has members that define the extent of simulation. For example part of the data management responsibilities of a backend is to load particle configuration files into particle objects. The extent of simulation is defined by backend members for particle interaction, particle lists, and a thermostat.

A model for representing nBLOCKs has been implemented for molecular dynamics (MD) simulations. The molecular dynamics backend is responsible for loading the particle configuration files into particle objects. The molecular dynamics backend also has a member for characterizing the interaction between particles during a simulation. These are the interaction classes.

The MD simulation backend also contains a member for a data structure which is constructed for each particle and designates a neighboring set of particles which this particle may interact with during a simulation step.

The final member designates a system thermostat. The thermostat may take on different roles depending on which ensemble the system is modeled under. In the NVT ensemble, where N stands for the total number of particles, V stands for the volume of the simulation space, and T stands for temperature, the system thermostat is called the Brownian thermostat. This thermostat introduces Brownian motion to the dynamics of the system through random reinitialization of particle velocity and particle angular momentum.

Each of the simulation backend members parses the input file and relevant parameters are extracted. Some of these parameters may be shared between these classes.

The simulation manager has a member function `run()` which is called from the standard C++ global `main()` function. The simulation manager then iterates for the designated number of simulation steps, calling the `simulation_step()` function that is implemented as part of the simulation backend. The CPU molecular dynamics backend `simulation_step()` function is composed of five functions. Three of these five functions calculate the Newtonian mechanics of the model. That is, they calculate forces between the interacting particles and use these forces to update a particle's

position and orientation. There is also a function for updating non-bonded neighbor lists and a function which calls the system thermostat.

```
void Molecular_Dynamics_CPU_Backend::sim_step(llint curr_step)
{
    first_step(curr_step);

    if (!lists->is_updated())
        lists->global_update();

    compute_forces();

    second_step();

    if (curr_step % newtonian_steps == 0)
    thermostat->apply(particles, curr_step);
}
```

The first function that is executed in a simulation step is called `first_step()`. This function updates particle positions and orientations based on the sum of the forces acting on the particles. The second mandatory function is `compute_forces()`. This function takes as input the output of the first step and calculates new forces in a pairwise manner between each particle and its bound and unbound neighbors. However, some of the particles in a neighbor list may not be included in these calculations if they are greater than r_{cut} distance away from the list owner. The final function that is always called is `second_step()`. This function updates the velocity and angular momentum of each particle based on new values for forces and torques calculated in `compute_forces()`.

Between the `first_step()` and `compute_forces()` functions particle neighborhood lists may be updated if the update to any particle's position during `first_step()` exceeds a threshold distance as measured from the initial position that the particle occupied when the current particle lists were constructed. This threshold, known as the Verlet-skin, is a parameter that is specified by the user in the input file. The final function that is called in `simulation_step()` is `thermalize()`. This function checks the number of simulation steps that have been performed and if this number modulo $N_{Newtonian}$ equals zero applies the system thermostat to the particle data. In the NVT ensemble the system thermostat is constructed to maintain an average system temperature by making adjustments to particle velocities and angular momentums.

2.3 NVIDIA GPU Programming Model and Architecture

The graphics processing unit (GPU) is made for problems which can be formulated in such a way that the same set of instructions are executed on many data elements in parallel. CUDA was released on November 2006 as a heterogeneous hardware and software platform for general purpose parallel computing. Since release there have been several major hardware architecture updates and corresponding API updates. In this thesis we will run simulations on a GTX780 from the Kepler [10] compute architecture. The NVIDIA CUDA software platform [9] exposes an API which can be executed natively in C++ and other languages such as C and Fortran.

2.3.1 Programming Model

Using Flynn's taxonomy [4] of hardware architectures NVIDIA describes the type of data parallelism achieved by the model as *single instruction multiple threads* or (SIMT). In contrast, the classic CPU architecture is described as *single instruction single data* under Flynn's taxonomy. The SIMT programming model exposes data parallelism by organizing threads into nested addressable groups, granting threads access to a number of memory types, and providing high-level API function calls that can make threads share data and or synchronize execution as needed. The GPU thread hierarchy is an abstraction for the underlying GPU hardware to which it maps. A CUDA thread corresponds to a CUDA arithmetic-logic unit (ALU) which executes the thread instructions. Ordered groups of these threads are organized in what are called thread blocks which which map to contiguous cores on a streaming multiprocessor (SMX on the Kepler Architecture [10].) Each SMX may have one or more thread blocks under its control but the SMX as a whole is addressable to a kernel grid. Figure 2.4 displays equivalent functions for both the host (CPU) and device (GPU) as well as initialization details for host to device transfer.

A function which is executed on an NVIDIA GPU is called a kernel. Kernels can be invoked from either the host (CPU) or device (GPU) but all of the execution occurs on the device. The kernel function signatures on the host side have a separate parameter space which can be used to

define the dimensions of the kernel grid. Sections of the kernel grid are mapped to SMX, and each SMX distributes the kernel instructions among the CUDA cores by block. And again, these cores execute in parallel and are addressable within the kernel function as threads.

2.3.2 Memory Types on the Kepler Architecture

Each CUDA thread executing on a GPU that is from the Kepler Architecture has access to six memory types: register memory, local memory, shared memory (L1 Cache), global memory, constant memory, and texture memory. Register memory is alloted on a per thread basis and in general is managed by the compiler. The register file is 256 kilobytes implemented as 64k 32-bit registers. These registers are partitioned among threads in an SMX. Registers are spilled to local memory as needed during execution. Local memory has the same latency as global memory. It is possible to share data among threads in a thread block (either loaded from global memory or as the product of intermittent thread calculations) in the L1 Cache or 'Shared Memory'. The L1 Cache is located on the chip along with the register file and has the second lowest latency. The lifetime of the data in the L1 Cache is tied to the lifetime of the thread block and is addressable based on the components of the thread id. Global memory is persistent between kernel invocations and can be read from and written to by both the host and the device. Global memory has the highest latency of all the memory types. The size of the global memory is typically several gigabytes and is addressable based on the components of the thread id. Constant and texture memory are both read only memory types and similar in principal. The cache for each resides on chip. Constant memory has an access pattern similar to global memory whereas the texture cache is optimized for 2D spatial locality.

2.3.3 Details of Memory Access for Addressable Memory

The threads in a thread block are bundled into groups called warps. Each warp holds exactly 32 threads. As a result a kernel grid should have a block size that is a multiple of 32. Memory operations are issued by the warp. When the memory operation is a read or write from global memory all threads within the warp will perform the memory transaction simultaneously. Global

```
void host_func(Particle* particles, const int N) {
  for (int i = 0; i < N; i++) {
    Particle* p = particles[i];
    float3 a    = p->pos;
    ...
  }
}

__global__ void device_func(float3* poss, const int N) {
  int BID = gridDim.x  * blockIdx.y  + blockIdx.x;  // block  id
  int TID = blockDim.x * threadIdx.y + threadIdx.x; // thread index in block
  int IND = N          * BID         + TID;         // thread index in poss
  if (IND >= N) return;
  float3 a   = poss[IND];
  ...
}

int main() {
  const int N = 32;
  Particle* particles[N]; // Host side particle objects

  /* Read particle object data from configuration files */

  if (RUN_CUDA) {
    float3* h_poss; // Host side particle positions
    float3* d_poss; // Device side particle positions

    /* Initialize h_poss and d_poss. */
    /* Copy position data from particles array into h_poss. */
    /* Perform a host to device memory copy (h_poss to d_poss) using CUDA API. */

    int tib = 8;                   // threads in a block
    int big = (N + tib - 1) / tib; // blocks in a grid
    device_func<<<big, tib>>>(d_poss, N);
  } else {
    host_func(particles, N);
  }

  return 0;
}
```

Figure 2.4: **CUDA code sample.**

memory is off chip in an area called device memory. Device memory is accessed via 32-, 64-, or 128-byte memory transactions. To reduce the total number of fetches required it is best if the per thread calculated addresses are aligned and the sum of the consecutive *n*-byte words requested is 32-, 64-, or 128-bytes.

Chapter 3

Implementation

Our nBLOCK extension to the oxDNA CUDA code involved translating the host side implementation of the nBLOCK interaction class from C++ to CUDA C++. However, to accommodate the nBLOCK model without making changes to the current CUDA DNA interaction class it was necessary to implement a new backend, and make modifications to the Verlet-cell list, and Brownian thermostat. As was discussed in the background chapter, a simulation backend links particle and list objects to the interaction class. Lists are maintained for bonded and non-bonded neighbors. Non-bonded neighbor lists are constructed according to the widely used Verlet-cell list algorithm. An interaction class uses these lists to apply potentials between particles. Finally, the Brownian Thermostat occasionally perturbs the Newtonian mechanics of the interaction class by introducing non-physical movements to the system via adjustments to the velocity and angular momentum of randomly selected particles.

All of the various forces acting on a nanoparticle or nucleotide can be calculated in isolation and in any order, and then reduced into a single force or torque vector. Calculations are split up in our nBLOCK GPU interaction class to exploit this fact.

Our CUDA nBLOCK interaction class first calls the CUDA DNA interaction class `compute_forces` function using the offset device arrays. Within the CUDA DNA `compute_forces` forces and torques are calculated pairwise between the bonded and the non-bonded neighbors of each nucleotide. When the CUDA DNA interaction functions are complete our CUDA nBLOCK `compute_forces` then computes the pairwise forces and torques between nucleotides bound to each nanoparticle as well as its non-bonded neighbors. The nature of these calculations which allows them to be carried out in a non-deterministic order and concurrently is a natural fit to the computation model offered by the GPU hardware and CUDA API

It is also worth noting that there are special backends written for GPU simulation in oxDNA. These backends inherit from the base backend but also implement functions for data transfer be-

tween the CPU (host) and GPU (device).

3.1 Implementation Constraints

A particle object on the host implementation contains members for its type, position, orientation, velocity, angular momentum, as well as the index into the backend particle list of that particle's five prime and three prime bonded-neighbors. These indexes are referred to as *pointers* throughout this discussion. For example, in Figure 3.2 **(a)** each box represents a particle and the black arrows represent neighbor pointers.

As is shown in Figure 3.1 the host side list of particle objects is converted into a position array, an orientation array, a velocity array, an angular momentum array, and an array for bonded neighbor pointers. To access the data a GPU thread is spawned for every particle and given a unique index into these device data arrays. The following code snippet depicts how each GPU thread calculates its index value.

```
// block  id
int BID = gridDim.x * blockIdx.y + blockIdx.x;

// thread index in block
int TID = blockDim.x * threadIdx.y + threadIdx.x;

// thread index in poss
int IND = N * BID + TID;
```

Representing particle data like this allows the device side force functions to spawn a thread for each particle and calculate forces between them in parallel. This is in contrast to the host-side method of iterating over the particle list and calling force functions between pairs of associated particles. For the host nBLOCK interaction class the pair-wise iterative method allows force functions between pairs of nucleotides to be calculated by its DNA interaction class member variable as they arise. The assumption with the device side technique is that all of the particle data passed to these functions is of the same type. While we could differentiate between nucleotide and nanoparticle data on a per thread basis within the force functions of the GPU DNA interaction class, doing

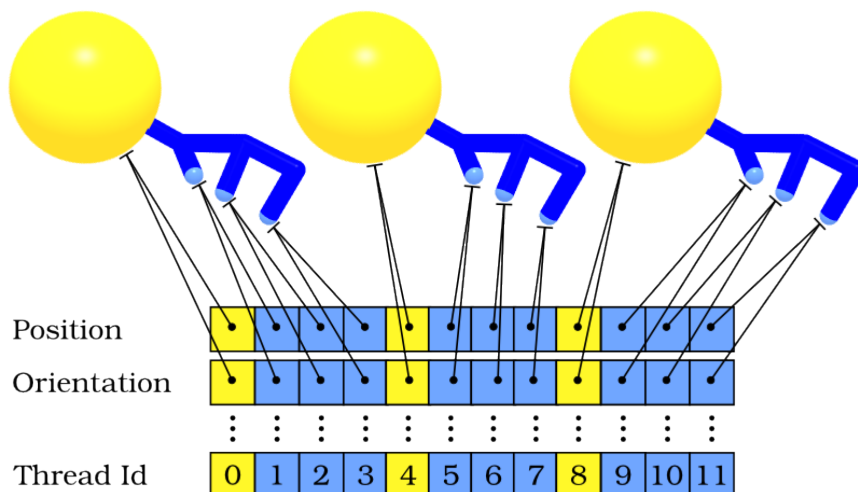


Figure 3.1: **Host to device translation of particle data.** Particle data is stored in objects on the host side. On the device, particle data is split into arrays by member value label.

so would require editing these functions.

3.2 Implementation Strategy

Given the difference in particle data layout between host and device as well as the preexisting CUDA DNA interaction class assumptions there were two identified strategies for implementing the nBLOCK model in CUDA within oxDNA. The first and most obvious strategy would be to copy the force functions from the CUDA DNA interaction class into our CUDA nBLOCK interaction class and then edit them such that they branch based on particle type. The second strategy would be to make no changes or copies of the CUDA DNA interaction class and instead make changes to the backends and neighbor lists. The first choice wouldn't scale well in terms of code complexity and overall maintenance. As a result we choose the second option. This option allows our implementation to automatically benefit from any future updates to the DNA interaction class by simply updating the files in the repository.

Instead of altering the CUDA DNA interaction class we edit nucleotide bonded neighbor information, and filter nanoparticles from nucleotide non-bonded neighbor lists. This information is instead maintained in the nanoparticle data. The way we implemented this is to stably reorder the

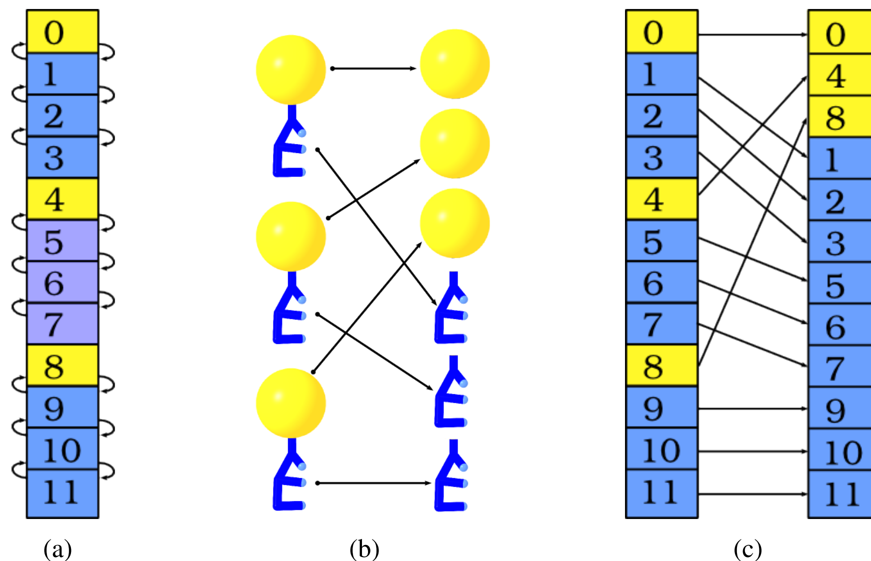


Figure 3.2: **Stably reorder particle data nanoparticle first. (a)** Shows pointers between bonded neighbors. These points define the NP-DNA topology. **(b)** Depiction of particle data as a collection of nanoparticle and nucleotide objects before and after reordering. **(c)** This sorting must preserve the order of nucleotides within a strand.

five particle data arrays such that all nanoparticle data comes first as in Figure 3.2.

Before we can perform our particle reordering we must generate maps between a particle's current index and its future index once the reordering occurs. This process is initiated by the following code:

```
void NP_MD_CUDABackend::set_index_data()
{
  np_first_indices = new int[NUMB_N];
  original_indices = new int[NUMB_N];
  int np_count     = 0;
  int nucl_count   = 0;
  for (int i = 0; i < NUMB_N; i++) {
    if (is_NP(this->_particles[i])) {
      np_first_indices[i]         = np_count;
      original_indices[np_count] = i;
      np_count++;
    } else {
      const int index          = NUMB_NP + nucl_count;
      np_first_indices[i]      = index;
      original_indices[index] = i;
      nucl_count++;
    }
  }
  for (int i = 0; i < NUMB_N; i++) {
    inv_np_first[np_first_indices[i]] = i;
```

```
        inv_original[original_indices[i]] = i;
      }
    }
```

Before calling `set_index_data()` we precalculate the total number of nanoparticles using the configuration files. `np_first_indices` matches the index of the current particle with the index to which it will be swapped to when we reorder the particles. To perform the inverse of this index swapping operation we also build an array `original_indices`. Notice that the nucleotide particles are also shifted right by `NUMB_NP`, which equals the total number of nanoparticles. The final for-loop in this function uses the results from the first to build maps that shift bonded neighbor pointers according to the rearrangement that that particle. Graphically this step is depicted in Figure 3.2 **(b)**.

After sorting the data in this way it is then necessary to adjust bonded neighbor pointers between nucleotides and nanoparticles such that they point to the new positions of their original neighbors. Next, the nucleotides which are bound to nanoparticles must have their three prime neighbor index assigned the value of -1. When a bonded-neighbor list pointer is assigned this value it indicates that there is no bonded neighbor in that direction. In other words, -1 act as a global sink node for nucleotides and nanoparticles for all unbound neighbor pointers. And finally, all of the bonded neighbor nucleotide indexes are decremented by the number of nanoparticles, `NUMB_NP`. By editing neighbor data in this way we can then offset the data arrays passed to the DNA interaction class force-function by the total number of nanoparticles using pointer arithmetic. The CUDA DNA interaction class can then calculate bonded and non-bonded forces between nucleotides without modification.

These maps are used as part of the functions in the nanoparticle-backend which perform host to device memory transfers. The inverse of the routine is correspondingly a part of the nanoparticle-backend device to host memory transfer functions.

3.2.1 Simulation Step

Once the particle data is organized nanoparticle first we may begin performing simulation steps. Our `sim_step` function is slightly different than what was seen on the host side molecular dynamics backend. Because we update the non-bonded neighbor lists for all particles in parallel, the decremented nucleotide bonded neighbors pointers must be temporarily fixed to accommodate for the fact that nucleotide GPU thread index values with the update list functions aren't offset by the total number of nanoparticles, as is the case with the compute forces function. Note that the `increment_LR_bonds()` and `decrement_LR_bonds()` are implemented on the device to increase efficiency.

```
void NP_MD_CUDABackend::sim_step(llint curr_step)
{
  first_step();

  if (are_lists_old)
  {
    increment_LR_bonds();

    cuda_lists->update(d_poss, d_list_poss, d_bonds);

    decrement_LR_bonds();

    decrement_edge_list_bonds();

    are_lists_old = false;
  }

  forces_second_step();

  thermalize(curr_step);
}
```

Within `forces_second_step()` the `compute_forces` function is called. Code for this function is seen below and a description follows.

```
void CUDANBLOCKInteraction::compute_forces(d_data)
{
  DNA_interaction->compute_forces(
    lists,
    (d_data->poss         + NUMB_NP),
    (d_data->orientations + NUMB_NP),
```

```
    (d_data->forces       + NUMB_NP),
    (d_data->torques      + NUMB_NP),
    (d_data->bonds        + NUMB_NP)
  );

  const dim3 blocks    = launch_cfg.blocks;
  const dim3 threads   = launch_cfg.threads_per_block;
  const int np_blocks  = (NUMB_NP - 1) / threads.x + 1;
  const int np_threads = threads.x;

  np_nucl_bonded_part<<<np_blocks, np_threads>>>(
    d_poss, d_orientations, d_forces,
    d_torques, d_np_bonds
  );

  np_nucl_excluded_volume_lists<<<np_blocks, np_threads>>>(
    d_poss, d_orientations, d_forces, d_torques,
    _v_lists->_d_number_neighs, _v_lists->_d_matrix_neighs, d_np_bonds
  );
}
```

By the time this function is called all particle data has been sorted according to our scheme. We utilize the particle reordering so that we can call `DNA_interaction->compute_forces` function on nucleotide only data. Because all of the nanoparticle data elements are at the front of each data array, offsetting these arrays by the number of nanoparticles causes the *i*th GPU thread to access the *i*th nucleotide past the last nanoparticle. In other words, when we adjust the device side pointers in this way before invoking a kernel the thread index of each particle is effectively reduced by the offset `NUMB_NP`. The final operation **(d)** in Figure 3.3 adjusts the bonded neighbor pointers to match the adjusted nucleotide indexes.

For the calculation of nanoparticle potentials that occur in the GPU functions `np_nucl_bonded_part` and `np_nucl_excluded_volume_lists` there is no need to utilize pointer arithmetic. Instead we know that if a thread's calculated index falls within a certain range then we have a particle of a particular type. Importantly, all of this can be accomplished without altering any of the native CUDA DNA interaction class code itself.

As of the current implementation of oxDNA the base simulation back-end only requests data transfer between the CPU and GPU to write trajectory or energy data to a file, perform analysis, or adjust particle position and orientation to maintain periodic boundary effects. The frequency of

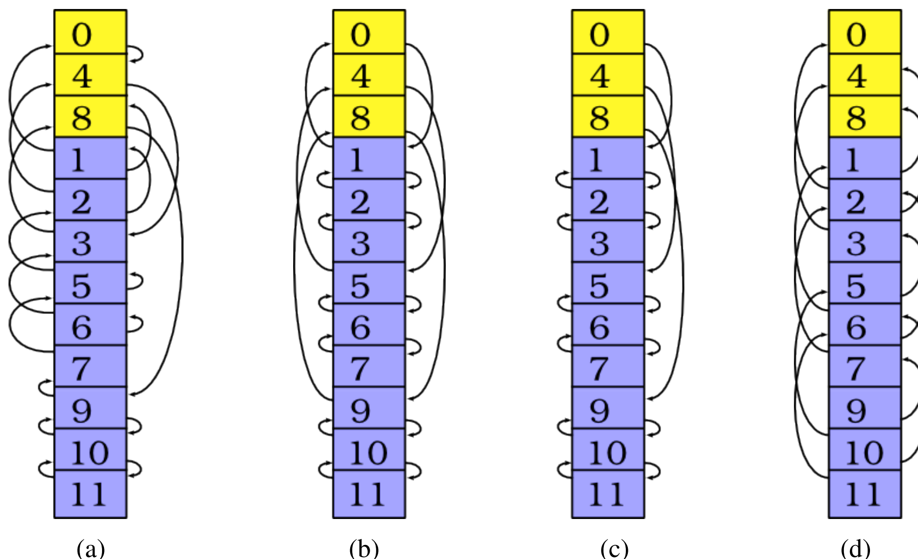


Figure 3.3: **Operations on Bonded Neighbor Pointers.** **(a)** Depiction of pointers immediately after ordering particle data nanoparticle first. **(b)** Correction of pointers such that original topology is preserved. **(c)** Invalidate bonds pointing from nucleotides to nanoparticles while preserving nanoparticle to nucleotide bonded neighbor pointers. **(d)** Decrement bonded nucleotide bonds by the number of nanoparticles before passing data to CUDA DNA interaction class.

all three cases is determined by setting flags in the input file. The more host-to-device and device-to-host transfers over the course of a simulation run the greater the sorting tax. However, as will be seen in the tests of the results chapter, these operations have a negligible impact on the overall performance of our implementation.

3.2.2 Accounting for Particle Mass

Originally the thermostat, first step, and second step functions assumed that simulations will consist of homogeneous particle type and uniform mass. We modified the CUDA Brownian thermostat and the step functions in the backend to accommodate the separate mass values for nanoparticles and nucleotides. We differentiate mass on a per particle basis based by the index of that particle, or correspondingly the index of the thread that performs calculations for that particle. If a thread has an index value less than `NUMB_NP` than we know it is a thread aligned with nanoparticle data and use the corresponding mass constant that is set for all nanoparticles. If the thread index is greater than `NUMB_NP` than we know it is a thread aligned with nucleotide data and scale the calculations

by the appropriate mass accordingly.

Currently the mass of all nanoparticles in a simulation share the same value. This value is assigned to each nanoparticle within the C++ NBLOCK interaction class function `read_topology` using a hard coded value. So to change nanoparticle masses between simulations would currently require adjusting this value by hand within the code. Note that changing the nanoparticle mass would require a corresponding change in the nanoparticle's diameter if you wish to keep the same particle density. However, assuming that all nanoparticles of a particle elemental type (e.g. Au) share the same density, the density value could be hard coded and the new diameter and radius could be found by algebraic manipulation of the formula Mass $=$ Density $\times$ Volume. To change both nanoparticle mass and diameter between simulations a variable for nanoparticle mass could then be added to the input file.

It is worth noting here that if we wanted to perform simulations of nBLOCKs using a range of different mass values this could be accomplished without any major degradation to performance of the device side implementation in terms of the first step, second step, and thermostat functions. To do this we could construct an array of particle mass values during the initialization functions of the nanoparticle backend at the same time that we construct the other particle data arrays such as position, orientation, etc. It would be convenient to do so because particle objects already maintain a member value for mass. We would also need to add the particle mass device array to the routines which sort the particle arrays nanoparticle first so that data for a particle is aligned appropriately in the device side memory.

Changes to diameter due to differing mass values would also have to be considered in the nBLOCK interaction classes which offset the nanoparticle-bonded nucleotide positions in proportion to the radius of the nanoparticle. We would then need to construct a particle radius device data array in the same way we would construct the particle mass device data array. Sorting of this array with all other device data would also be necessary. Visualization of nanoparticles with differing radii would also need to be accounted for.

3.2.3 oxDNA-nBLOCK Support Scripts

oxDNA offers a number of convenient tools for the generation of configurations and the analysis of simulations. Some of these tools come in the form of Python scripts. In some cases the nBLOCK Python scripts build on the functionality that was implemented previously for the oxDNA DNA model. In many ways the utility scripts of a molecular simulator are as important as the simulation model itself. The main output of a simulation is trajectory data or energy profiles and translating those values into a more useful form is a great benefit when developing and analyzing the unerlying model.

- `nb_fileio.py`

  Takes as a parameters the absolute or realteive paths to a a pair of complimentary configuration files. One of these files defines the topology of each nBLOCK and the other file defines the position, orientation, normals, velocity, and angular momentum of each particle. These files are used to create an nBLOCK data container object. The nBLOCK data object is then used in other scrips supporting our nBLOCK extension to oxDNA.

  In addition this script calculates the orientation matrix of each particle from data in the configuration file. That is, given the orientation vector $v1$ and the backbone normal $v3$ we caclulate their cross product as the value $v2$. $v1, v2, v3$ then become the first, second, and third rows of the particle orientaiton matrix.

- `nb_simcube.py`

  This script takes as input paths to configuartions files as well as an integer that designates the number of lattice points along the edges of a cubic lattice, n. This script then reads in the input configuration files and translates each partcile among the n^3 lattice points. For example if the input is a monofunctionalized nBLOCK assembly and $n = 5$ this script would output a new set of initial configurations with 125 monofunctionalized nBLOCKs. The nBLOCK lattice configurations can then be thermalized as desired to create inputs for tests and experiments. Reference [5] has a good description of this general technique.

- `test_fileio_simdata_obj.py`

  This script added regression testing to our `fileio` and `simdata` Python scripts. Regression tests are designed to determine if any code modifcations have inadvertently changed some aspect of the underlying script. Before this script can be used it must be initialized. To perform the regression tests a large number of pregenerated configurations are fed into the program. These configurations are turned into simulation data objects using `nb_fileio.py` and `nb_simdata.py`. Once the simulation data objects are initialized we recursively *XOR* the SHA-1 hexidecimal digest of their member values together. The result is a 160 bit long hash value that (probably) uniquely represents the instantiated object. SHA-1 hashes are generated like this for all test files and then persisted with the Python `pickle` library. Then when we make changes to our simulation data scripts we can run this script which recalculates new hashes from the test data and compares them to the persisted values.

- `nb_distances.py`

  Calculates the distances between each nanoparticle in a given set of configuration files. Due to the use of periodic boundary conditions within the standard oxDNA simulation distances bewteen NP are reported based on the minimum images of each. This distance is expressed in the following Python code snippet:

```
def distance(np_i, np_j, box_size):
  n = box_size / 2.0
  xs = np_i - np_j
  u = 0.8518
  for i, k in enumerate(xs):
    if k > n:
      xs[i] -= box_size
    elif k < -n:
      xs[i] += box_size
  return u * l2_norm(xs)
```

3.2.4 Mixed Precision

It is considerably more expensive in terms of memory bandwidth to read a double floating point value from the device main memory than it is to read a single floating point value. It may be tempting to use floating point values due to their considerable performance gain. However, when your intent is to simulate a physical process as accurately as possible the accumulation of round off error due to the use of floating point precision causes errors in simulation. To work around this oxDNA has implemented a mixed-precision backend. The techniques from this backend use single precision to calculate forces and use double precision during the integration of positions and momenta in the first and second step. We have implemented mixed precision for our model and show comparisons between run times for single and mixed-precision simulations in our result section.

Chapter 4

Results

Our results section describes the tests used to validate and quantify the performance of our GPU-based implementation of the nBLOCK extension to oxDNA. These tests fall into two categories: validation tests, and performance tests. Through our validation tests we show that the device and host implementations simulate the dynamics of nBLOCKs equivalently. This process is more difficult than it may first appear because GPU simulations within oxDNA are not reproducible [16]. Despite this we expect the distribution of kinetic energy values on the host or device to follow the Maxwell distribution. To validate potential energy logs between the host and device we borrow a technique from the oxDNA testing utilities that bounds the allowable difference between the mean potential energy of separate simulations of the same configuration. The performance tests were designed to measure the relative run times of the host versus device implementations, and the results presented in this section show comparisons between the run times of thirty six unique initial configurations. As expected, the GPU implementation of the nBLOCK model is significantly faster than the CPU implementation.

4.1 Simulation Techniques

Validation and performance test statistics were sampled from simulations using an Andersen-like (aka Brownian) thermostat in the NVT ensemble. Under these macro parameters random collisions with an implicit solvent are introduced to the simulation dynamics. The sequence and magnitude of particle trajectory perturbations is not shared between host and device due to independent random number generation. As a result we cannot test the validity of the device implementation by direct comparison of particle trajectories. Instead we must make comparisons of the average behavior of separately seeded simulations with matching initial configurations. For the validation tests we gather statistics on the kinetic energy (KE) and potential energy (PE) across a range of carefully designed simulations.

The thermostat in an NVT ensemble is designed to keep the average temperature of the system at a specified value. To accomplish this, every $N_{Newtonian}$ time steps the simulation backend passes all of the particles to the Brownian thermostat. The thermostat then draws two random values from a uniform distribution for each particle. If the first of these random values is below the threshold value p_{V} the velocity of that particle is reassigned to a vector drawn from a Gaussian distribution. If the second random value is below a separate threshold p_{L} the angular momentum of the particle is reassigned to a vector drawn from a Gaussian distribution.

In other words, the thermostat intentionally breaks the time reversibility of the Newtonian mechanics governing particle interactions by moving the particles in non-physically justified ways. One of the purposes of these random adjustments to each particle is to model the interaction of the particle with an implicit solvent. The global values p_{V} and p_{L} are a function of T, δt, $N_{Newtonian}$, and the diffusion coefficient. All of these parameters can be specified by the user in the input file of a simulation.

4.2 Validation Tests

The process of generating validation datasets and the methods used to compare them must be done with respect to how the Brownian thermostat effects the simulation dynamics. That is, without consideration of the random non-physical moves introduced by the thermostat, direct comparison of host and device trajectories for a given initial configuration will indicate that the device and host implementation are not simulating the same dynamics. To make meaningful comparisons between the host and device we characterize the average behavior of each by analyzing statistics for the continuous random variables KE and PE.

4.2.1 Comparing Kinetic Energy

Under the NVT ensemble the distribution of kinetic energy values, $KE \sim \mathcal{P}(\mu, \sigma^2)$, should closely approximate the Maxwell distribution with location μ_{KE} and scale $\sigma^2{}_{\mathrm{KE}}$. The distribution of KE values should fit the Maxwell distribution regardless of the current micro-state of the simulation.

In other words not only is the comparison device independent but it is also conformationally independent. That is, any stable simulation with the nBLOCK interaction class should produce a distribution of KE energy values that closely match the Maxwell distribution.

To gather statistics for this test we simulated a two hundred particle nBLOCK duplex assembly [7] for 10^6 simulation steps using both the device and host implementation. Both the device and host simulations used the following initialization parameters: temperature $T = 293K$, a step size of $\delta t = 0.005$, a Verlet-skin of length 0.5, the thermostat was called every 103 Newtonian steps, the diffusion coefficient set to 2.5, and a $[NA+]$-concentration of 0.5. The KE of these simulations was sampled and logged every 10^3 simulation steps. Using all 10^3 sample KE values in a single plot (as is done in Figure 4.3) can be visually deceiving because outliers are overrepresented in comparison to the hundreds of overlapping points that make up the majority of points. To give intuition to this notion we generated Figures 4.1 and 4.2 which show Maxwell probability plots for the host and device that were plotted using only 10^2 KE values randomly drawn from the full set of 10^3 samples.

4.2.2 Probability Plot Results Analysis

Probability plots can be used to determine if a sample distribution may have been drawn from some other target distribution. In this case we want to determine if the KE values taken intermittently from nBLOCK simulations match the Maxwell distribution.

Let KE represent the kinetic energy values from a single simulation and let n represent the number of values in KE. First we sort KE in ascending order. We then pair a value with each ith index in KE using Filliben's estimate represented here as the function F. Filliben's estimate takes the form of the following piecewise function:

$$\begin{cases} 1 - 0.5^{\frac{1}{n}} & i = 1 \\ \frac{(i-0.3175)}{(n+0.365)} & 1 < i < n \\ 0.5^{\frac{1}{n}} & i = n \end{cases}$$

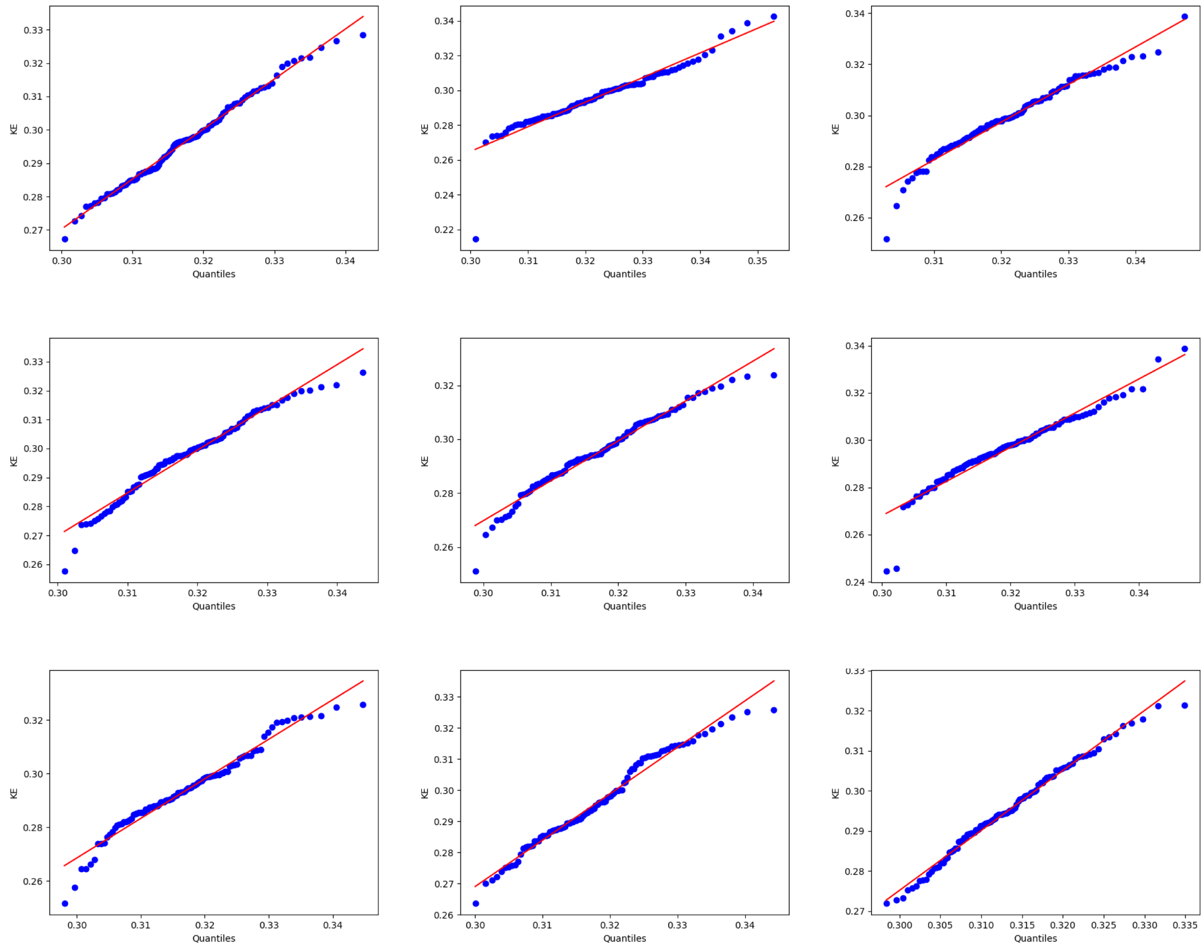


Figure 4.1: **Sparese CPU Maxwell probability plots.** Each of these probability plots represents random samples of 10^2 points taken from the full KE distribution gathered during the simulation of a 200 particle nBLOCK duplex assembly. The complete plot of all 10^3 KE values is shown in Figure 4.3 **(a)**.

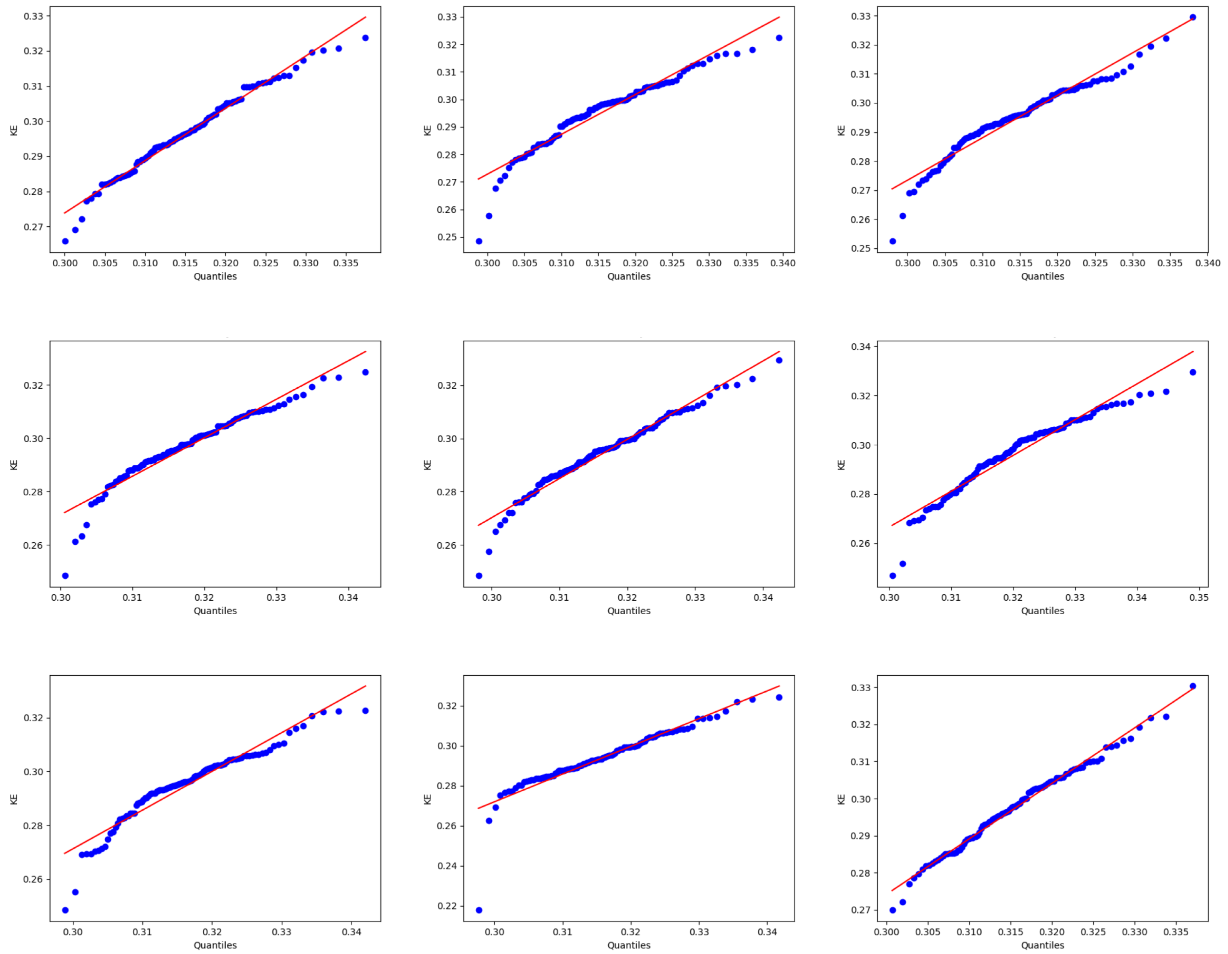


Figure 4.2: **Sparese GPU Maxwell probability plots.** Each of these probability plots represents random samples of 10^2 points taken from the full KE distribution gathered during the simulation of a 200 particle nBLOCK duplex assembly. The complete plot of all 10^3 KE values is shown in Figure 4.3 **(b)**.

We plot these points $(KE_i, F(i))$ against the cumulative distribution function of the Maxwell distribution that is centered on μ_{KE} and has the spread $\sigma^2{}_{\text{KE}}$. The x-axis values of the points along the Maxwell cumulative distribution function which are horizontal to each $(KE_i, F(i))$ point are designated by M_i. Our probability plot is composed of the points (KE_i, M_i). If every i^{th} point in KE is approximately equal to the paired point in M then when we plot these points they should lie along a straight line.

When building a probability plot from a large set of samples, as we have done in Figure 4.3, the outliers at the distribution tails are visually overrepresented so it is no longer reliable to visually inspect the plots for goodness of fit. To quantify goodness of fit for our Maxwell probability plots we calculate the coefficient of determination for the all of the 10^3 KE samples. The coefficient of determination (r) quantifies the straightness of our line. The closer that r is to 1.0 the stronger the relationship is between the generated KE values and the Maxwell distribution.

The first step in calculating this coefficient is to draw a line through the center of mass of all (KE_i, M_i) points. This line is also known as the least squares line. Next we calculate the mean value of M which we designate as $\bar{y}$. If all we knew about M was that it had a mean $\bar{y}$ and we were asked to make a prediction for the value of a random variable in M we would choose $\bar{y}$. If we did this for all the values in M we could say that equation 4.1 is the sum of our squared prediction errors.

$$\sum_{i=1}^{n}(y_i - \bar{y})^2 \tag{4.1}$$

We repeat the calculation for the sum of our prediction errors again, but this time we guess the value $\hat{y}$ from the y-axis of the least squares line.

$$\sum_{i=1}^{n}(y_i - \hat{y})^2 \tag{4.2}$$

The strength of the relationship between our plotted points and the least squares line is measured by

$$\sum_{i=1}^{n}(y_i - \bar{y})^2 - \sum_{i=1}^{n}(y_i - \hat{y})^2 \tag{4.3}$$

Ideally the (KE_i, M_i) points should be close to the least squares line so the value of the second summation (4.2) should be relatively small in value compared to the first (4.1), and the larger the value produced by this subtraction the greater the strength of the relationship.

Recall that $\sum_{i=1}^{n}(y_i - \bar{y})^2$ is a component in the measurement of variance. These subtractions represent the reduction in variance that occurs by using the least squares line to predict values in M. We divide the result of 4.3 by $\sum_{i=1}^{n}(y_i - \bar{y})^2$ to get our coefficient r^2.

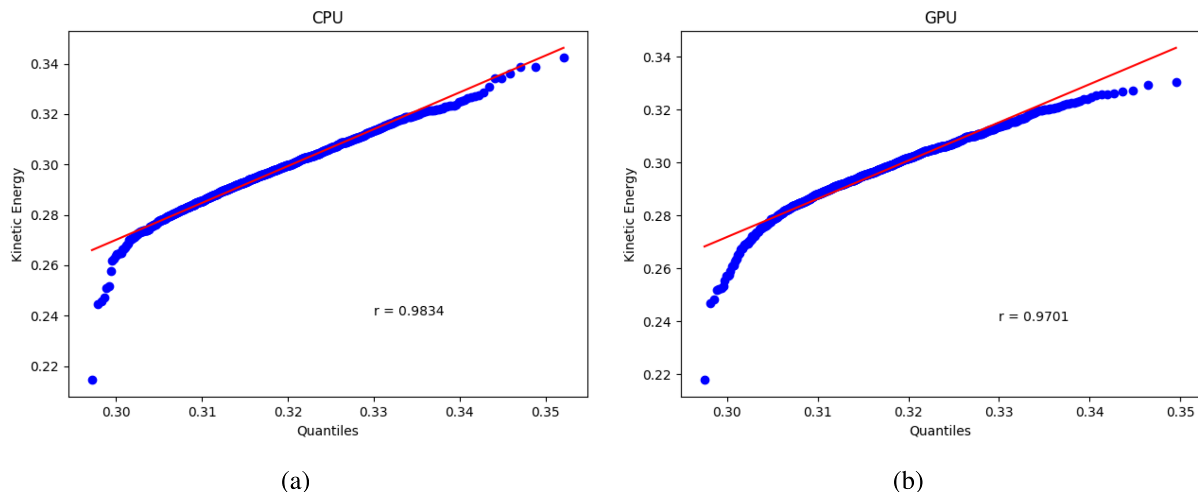
$$r^2 = \frac{\sum_{i=1}^{n}(y_i - \bar{y})^2 - \sum_{i=1}^{n}(y_i - \hat{y})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{4.4}$$

The square root of which is the coefficient of determination. The r values shown in each plot indicates that in spite of the outliers a large proportion of our points can be predicted by the least squares line, thus validating our implementation in terms of the kinetic energies of the simulated particles.

4.2.3 Comparing Potential Energy

A potential energy measurement can be described as a summary statistic for the relative position and orientation of each particle in a simulation. As particle positions and orientations are updated in each simulation step the potential energy of the simulation will change. In an nBLOCK simulation potential energy is a function of the FENE spring potential between nucleotide backbone sites and nucleotides bound to nanoparticles, hydrogen bonds between base pairs, the intra-strand stacking potential, the cross stacking potential, the Lennard Jones excluded volume potential, and a modulating potential that encourages helicity of dsDNA [11, 13]. Each of these potentials, for all particles in the simulation, contributes to the total PE of the system.

Our methods of comparison were borrowed from the testing utilities of oxDNA. We also mimicked the strand topologies from these tests. The original strand topologies used to compare the



(a) (b)

Figure 4.3: **Maxwell probability plots. Figure Explanation.** The Y-axis is scaled to match the distribution of the gathered *KE* measurements. The X-axis shows the values corresponding to quantiles of the Maxwell distribution. The blue dots in **(a)** and **(b)** represent one thousand *KE* samples taken at an interval of one thousand time steps from the simulation of a two-hundred particle diatomic nBLOCK assembly. The straight red line is plotted through the center of mass for this points. Because there are many overlapping points in these plots we further quantify the result using the square root of the coefficient of determination *r* which is shown in each figure. An *r* value of 1.0 would indicate that the variance in the *KE* samples is completely predictable by the Maxwell distribution. As such the reported *r* values demonstrate the *KE* distribution of our nBLOCK simulations are a good fit the Maxwell distribution.

mean potential energy of DNA simulations include a fifteen base pair ssDNA composed of pure Adenine (Poly-A-15) and a fully hybridized hairpin composed of eighteen base pairs. To closely approximate these tests in the nBLOCK model we generated a Poly-A nBLOCK with fifteen nucleotides and a fully hybridized diatomic nBLOCK assembly. These tests allowed for the comparison of *PE* between separately seeded simulations by removing the contribution of hydrogen bonding to the potential energy of the system. As might be expected the distribution of *PE* values for the hairpin loop has a smaller scale factor than what is sampled from the Poly-A ssDNA. This is because the range of motion for the nucleotides within the hybridized hairpin is restricted. The Poly-A topology has a wider range of conformations it can sample from but cannot get stuck in any intermediary states due to hydrogen bonding between Watson-Crick complements.

To gather statistics for PE validation we ran a series simulations using identical initial configurations of nBLOCK assemblies on both the host and device. These simulations were parameterized

with a temperature $T = 293K$, a step size of $\delta t = 0.005$, a Verlet-skin of length 0.5, 103 Newtonian steps, the diffusion coefficient set to 2.5, and a $[NA+]$-concentration of 0.5. During these simulations potential energy values were logged at fixed intervals. For each simulation we calculated the mean of the potential energy values reported by the host and by the device. A confidence interval $\pm 5\frac{\sigma}{\sqrt{n}}$ around the mean potential energy of the host simulation was then constructed. Our validation tests then checked whether or not the mean potential energy of the GPU simulation fell within this interval.

While we have translated the Poly-A and hairpin topologies of the original tests to similar topologies for the nBLOCK computational model there are two important differences in the parameterization of these tests that should be considered. First, for nBLOCK simulations on the host and device we used a Verlet skin with a value of 0.5. This is in contrast to the Verlet skin of the original DNA simulations which used a value of 0.05. We used this increased Verlet skin size to compensate for the large diameter of the nanoparticles. Because we used a larger Verlet skin for our nBLOCK simulations the area that was included in each nucleotide's non-bonded neighbor list was greater than the area around each nucleotide in the original tests. Due to this larger volume the potential energy for an nBLOCK simulation would have had on average a greater number of pairwise particle interactions. Second, the original tests were designed to test the *PE* between separate simulations on the CPU, whereas our tests were used to compare the results between a CPU nBLOCK simulation and a GPU nBLOCK simulation.

Additionally, while we can use probability plots to compare the distribution of logged *KE* values against the Maxwell distribution there are in general no known standard statistical methods for the comparison of *PE* distributions [18]. That is, while we can collect samples of the *PE* distribution from our simulations the population distribution of *PE* is unknown.

The differences in parameterization between the original tests and our own along with having no distribution to make strict statistical comparisons against prevent us from determining whether our results for *PE* validation are statistically significant. Nonetheless, we can state that the mean *PE* of the device side implementation is within the designated confidence interval in eleven of the

twelve test cases presented in Figure 4.4, which provides strong evidence that our GPU implementation is accurately matching the CPU implementation. Additionally, the results for the nBLOCK duplex assembly also demonstrate that our nBLOCK GPU implementation methodologies allow hybridization to occur and to be maintained as expected.

4.3 Performance Tests

Performance tests were taken across thirty-six unique initial configurations on a CPU and a GPU. As is to be expected the GPU implementation has a greatly reduced average milliseconds per time step compared to the CPU. In addition the performance tests demonstrate the robustness of the device implementation to a wide range of particle counts and the full range of nBLOCK topologies. All of our GPU simulations were run on an NVIDIA GTX780 which has the Kepler computer architecture [10]. The CPU performance test simulations were run on an Intel Xeon X5670 processors [6].

4.3.1 Dataset Preparation

Generation of the dataset for the performance tests proceeded as follows. We started with six sets of nanoparticles. Each set had a unique size (1, 8, 27, 64, 125, 216). The nanoparticles from each set were then evenly spaced on a cubic lattice in proportion to the eventual volume that each would occupy once functionalized with strands. We made six copies of each lattice and then functionalized the nanoparticles in each copy with between one and six strands. Two complimentary strands were used for all thirty six initial configurations and each nanoparticle was functionalized with one strand type. The strand topologies are shown as S, S' below.

S: $5'$ ACA CAC ACA CAC ACA CAC ACA $3'$

S': $5'$ TGT GTG TGT GTG TGT GTG TGT $3'$

In simulations with multiple nBLOCKs the strands were alternated between neighbors along the lattice points. Finally, before performance statistics were gathered each of the thirty six configurations were thermalized using the host implementation for 10^5 time steps using double precision,

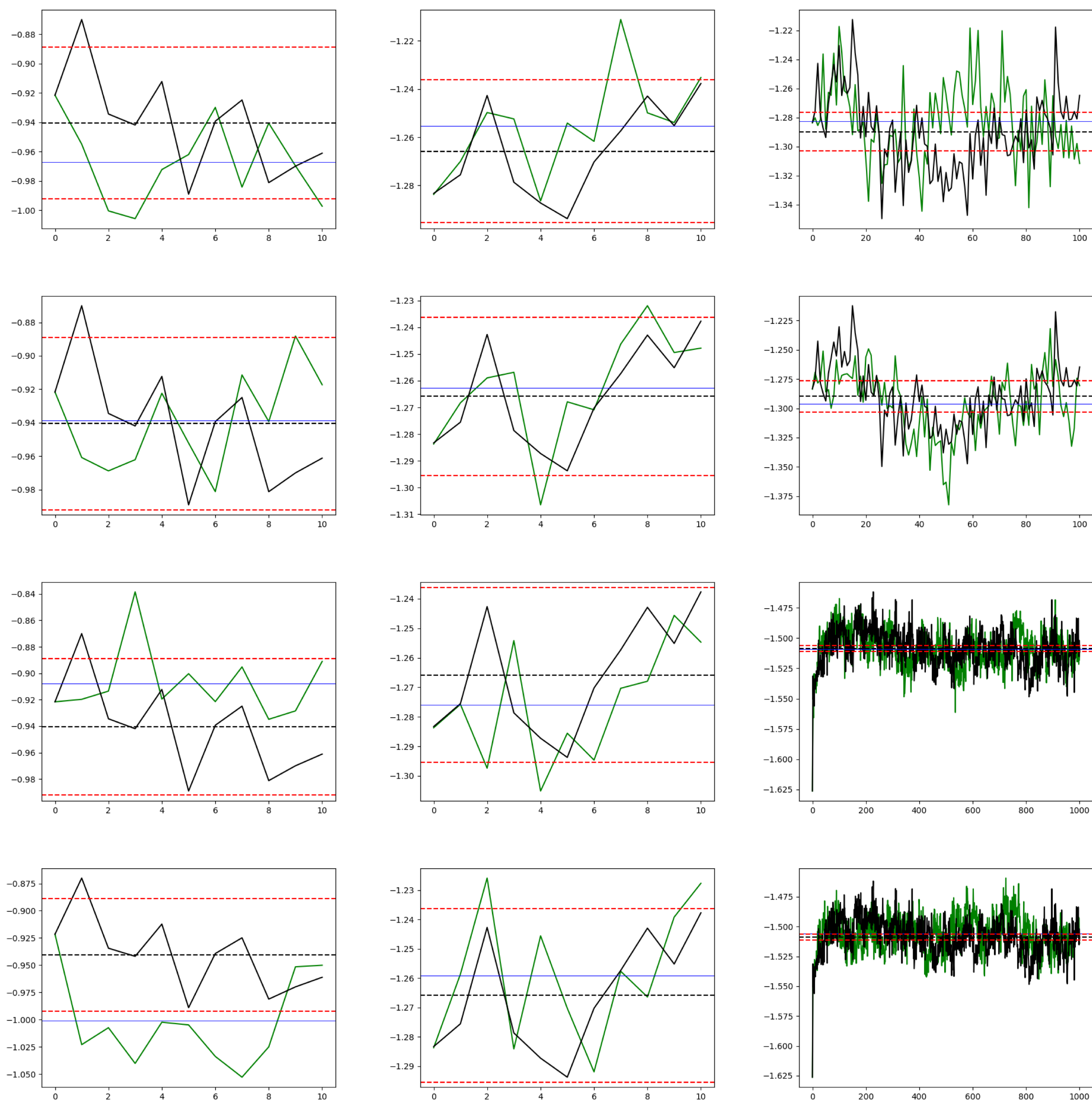


Figure 4.4: **Potential energy plots. Figure explanation.** The solid black line represents CPU *PE* over time. The green line represents the GPU *PE* over time. The three horizontal dashed lines represent the CPU mean *PE* and $\pm$ 5 standard errors of the mean, while the solid blue line is the GPU mean *PE*. **Column one. NP-PolyA-15** Plots the results from ten *PE* samples taken at 10^2 time steps intervals. **Column two. Diatomic-nBLOCK** Same simulation duration and sampling interval as column one but taken from a simulation of a diatomic nBLOCK assembly with fifty total particles. **Column three, upper-half.** The same nBLOCK topology as column two. One hundred samples with interval of 10^3. **Column three, lower-half.** Diatomic-nBLOCK assembly with one hundred particles. One thousand samples with interval of 10^3.

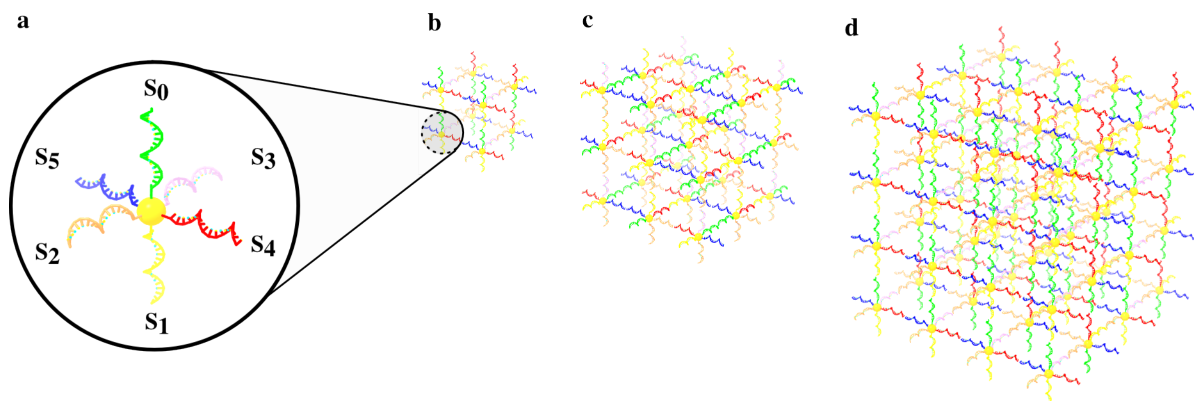


Figure 4.5: **Example initial configurations along 3D grid. a** 126 nt and 1 NP **b** 1008 nt and 8 NP **c** 3402 nt and 27 NP **d** 8064 nt and 64 NP

$T = 293K$, $\delta t = 0.005$, Verlet-skin of length 0.5, 103 Newtonian steps, the diffusion coefficient set to 2.5, and a $[NA+]$-concentration of 0.5.

4.3.2 Performance Tests Results Analysis

To gather performance statistics each of the thermalized initial configurations were run for five million simulation steps on the host and device implementations with the parameterization used for thermalization. This process was repeated a total of four times for each configuration. The average total run time and the average milliseconds per time step were then tabulated from the simulation logs. Timings were tabulated at the end of each simulation using the preexisting oxDNA logging utilities. In addition to logging simulation step timings we also set flags in our input files for these tests that caused each simulation to record energy profiles every five hundred thousand time steps. Before energy profiles can be calculated the device must transfer the data back to the host causing our backend to return the particle data to its original order. As such the results in Figure 4.6 also indicate that the additional processing required by our nanoparticle backend during host-to-device and device-to-host transfers of particle data is greatly outweighed by the overall increase in computation efficiency offered by the GPU. The reported timings measure the average number of milliseconds per time step for a given number of particles and nBLOCK topology.

The plots in Figure 4.6 show that considerable speed-up was achieved by our GPU nBLOCK

implementation over the CPU version. But there are some interesting aspects of the plots that are worth discussing. To give some background on our proceeding analysis we note that in [16] it is stated that the simulations of DNA in oxDNA are weakly dependent on the density of the particles in a simulation. This weak dependency is due to the highly anisotropic interactions between nucleotides. That is, the orientation of the nucleotide screens out other nucleotides which are not as closely aligned.

However, the spikes in the graphs of Figure 4.6 reveal some that nBLOCK simulations are to some extent dependent on the local density of each nBLOCK assembly. Generally we define the idea of local density to be a function of the number of arms on a particular nBLOCK. Occasionally it can be seen that as we move along the x-axis the runtime decreases despite the increase in the total number of particles. Note also that these spikes occur on both the GPU and CPU. The log-log plots for CPU timings in subplot **b** and **d** smooth these spikes for the CPU timings to some extent. But in the non-scaled axis of subplot **a** they are present. A closer inspection of the subplot **c** which is the log-log scaled timings for the GPU tests more clearly demonstrates a few of these points. In these circumstances the spikes are caused by sequential simulation timings that go from simulations with proportionally fewer nanoparticles compared to nucleotides to simulations with a proportionately greater number of nanoparticles with respect to the number of nucleotides. Because all strands in our performance tests are of length 21 we can also state this shift in proportions as going from nanoparticles with more arms to nanoparticles with fewer arms. In either case a downward spike in the timings plot indicates a decrease in the local density of the nanoparticles. It is reasonable to assume that the greater the local density around an nBLOCK the more the nucleotides between each strand will interact. And it is certain that the nanoparticle-nucleotide interaction computations will be greater.

Although the spikes occur on both the host and device implementations the severity of the spikes on the device are greater. This is because we spawn a thread for each nanoparticle and the workload of this thread increases in proportion to the number of arms and the relatively simple arithmetic logic unit behind each thread is not as robust to this increase in workload as is the CPU.

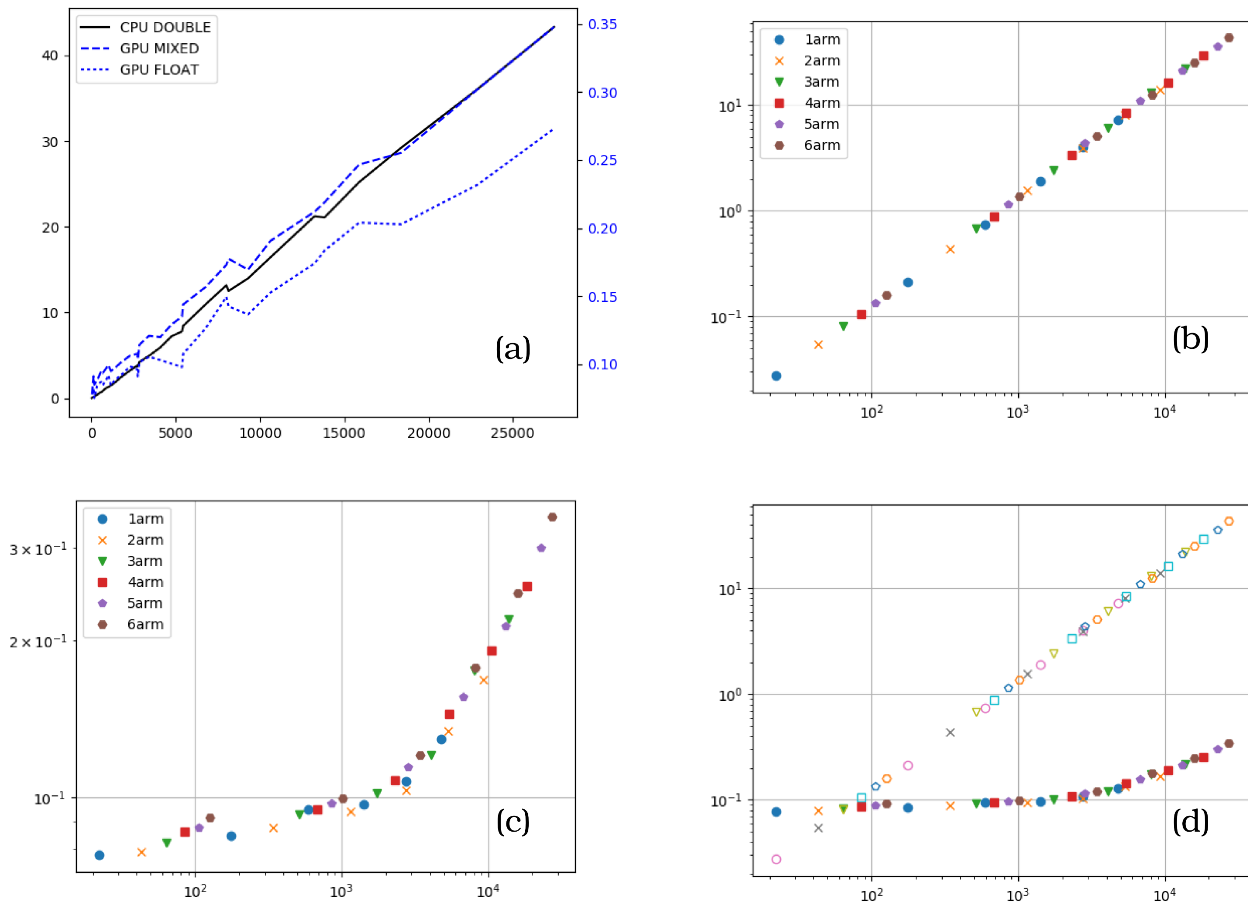


Figure 4.6: **Performance Tests. Figure explanation.** All X-axes represent the number of particles N. All Y-axes represent the average time step length, t in milliseconds. **Figure (a)** the left Y-axis represents t scaled to the CPU timings. The right Y-axis represents t scaled to the GPU timings. Floating precision and mixed precision calculations share the same scale, although pure floating point precision simulations enjoy a considerable speed-up over the mixed point precision. Notice that the spikes are mirrored across both GPU implementations and to a lesser extent the CPU implementation. **Figures (b) and (c)** display log-log plots for the average CPU and GPU timings respectively. In these plots average milliseconds per time step, the number of particles in the simulation, and the nBLOCK topology are accounted for. We can also see that the intermittent spikes present in the plots of **(a)** can occur as we move along the X-axis from a simulation with a greater number of arms to a simulation with relatively fewer arms. **Figure (d)** displays a log-log plot of CPU and GPU timings. The CPU timings are represented by the marks which have no fill. Otherwise the markings follow the same legends of plots **(b, c)**. We can see the GPU begins to outperform the CPU at approximately eighty-five particles.

Chapter 5

Conclusion

In conclusion this thesis details the implementation of a coarse grained nBLOCK computational model for graphical processing units (GPUs) using the NVT ensemble and a Brownian thermostat [17] for the modeling of kinetics. This implementation is an extension to the oxDNA molecular simulator. The main contribution of these techniques is a design technique for the integration of forces between heterogeneous particle types with separately implemented interaction classes using the NVIDIA CUDA programming model. We have also demonstrated validation and performance testing methodologies for our implementation. In particular we show that these nanoparticle-nucleotide simulations maintain an appropriate KE distribution as dictated by the Brownian thermostat in an NVT ensemble. We further validate our device implementation by borrowing metrics from oxDNA's testing utilities to make comparisons between the mean PE profiles of separate simulations. Our performance tests showed 100-fold speedup for the simulation of configurations which have a total number of particles greater than 10^4.

Ideas for future work may be taken from [12] in which T.E. Ouldridge outlines a list of computational methods used for the study of DNA in oxDNA. We will now list the techniques from that list which have yet to be implemented for the oxDNA nBLOCK extension.

- The use of *Virtual Move Monte Carlo* for modeling thermodynamic properties.
- The implementation of umbrella sampling for the collection of statistics under thermodynamic models.
- The use of a Langevin thermostat for additional modeling of dynamical properties beyond what is provided by the Brownian thermostat.
- The implementation of forward flux sampling for the collection of statistics under dynamical models. These dynamic models include our current implementation of the Brownian thermostat as well as the proposed implementation of the Langevin thermostat.

Other future work may include the extension of our current nBLOCK computational model to include nanoparticles with differing masses and diameters. Other research groups which study DNA-functionalized nanoparticles with different patterns of strand attachment to the nanoparticle surface may find our nBLOCK extension to oxDNA as a good starting place for the implementation of their own models.

References

[1] A Paul Alivisatos, Kai P Johnsson, Xiaogang Peng, Troy E Wilson, et al. Organization of'nanocrystal molecules' using dna. *Nature*, 382(6592):609, 1996.

[2] George Church and Ed Regis. *Regenesis*. Basic Books, 2012.

[3] Jonathan PK Doye, Thomas E Ouldridge, Ard A Louis, Flavio Romano, Petr Šulc, Christian Matek, Benedict EK Snodin, Lorenzo Rovigatti, John S Schreck, Ryan M Harrison, et al. Coarse-graining dna for simulations of dna nanotechnology. *Physical Chemistry Chemical Physics*, 15(47):20395–20414, 2013.

[4] M. Flynn. Very high-speed computing systems. *IEEE Proc.*, 54:1901–1909, 1966.

[5] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*, volume 1. Academic press, 2001.

[6] INTEL. Intel xeon processor x5670.

[7] Jin-Woo Kim, Jeong-Hwan Kim, and Russell Deaton. Dna-linked nanoparticle building blocks for programmable matter. *Angewandte Chemie International Edition*, 50(39):9185–9190, 2011.

[8] Chad A Mirkin, Robert L Letsinger, Robert C Mucic, and James J Storhoff. A dna-based method for rationally assembling nanoparticles into macroscopic materials. *Nature*, 382(6592):607, 1996.

[9] NVIDIA. Cuda toolkit documentation v7.5. Accessed: 2016-04-06.

[10] NVIDIA. Nvidias next generation cuda compute architecture: Kepler gk110. 2012. Accessed: 2016-04-20.

[11] Thomas Ouldridge. *Coarse-grained modelling of DNA and DNA self-assembly*. PhD thesis, University of Oxford, 2011.

[12] Thomas E Ouldridge. Dna nanotechnology: understanding and optimisation through simulation. *Molecular Physics*, 113(1):1–15, 2015.

[13] Thomas E Ouldridge, Petr Šulc, Flavio Romano, Jonathan PK Doye, and Ard A Louis. Dna hybridization kinetics: zippering, internal displacement and sequence dependence. *Nucleic acids research*, 41(19):8886–8895, 2013.

[14] Eric F. Pettersen, Thomas D. Goddard, Conrad C. Huang, Gregory S. Couch, Daniel M. Greenblatt, Elaine C. Meng, and Thomas E. Ferrin. Ucsf chimera - a visualization system for exploratory research and analysis. *Journal of Computational Chemistry*, 25(13):1605–1612, 2004.

[15] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[16] Lorenzo Rovigatti, Petr Šulc, István Z Reguly, and Flavio Romano. A comparison between parallelization approaches in molecular dynamics simulations on gpus. *Journal of computational chemistry*, 36(1):1–8, 2015.

[17] J. Russo, J. M. Tavares, P. I. C. Teixeira, M. M. Telo da Gama, and F. Sciortino. Re-entrant phase behaviour of network fluids: A patchy particle model with temperature-dependent valence. *The Journal of Chemical Physics*, 135(3):034501, 2011.

[18] Michael R Shirts. Simple quantitative tests to validate sampling from thermodynamic ensembles. *arXiv preprint arXiv:1208.0910*, 2012.

[19] Niranjan Srinivas, Thomas E Ouldridge, Petr Šulc, Joseph M Schaeffer, Bernard Yurke, Ard A Louis, Jonathan PK Doye, and Erik Winfree. On the biophysics and kinetics of toehold-mediated dna strand displacement. *Nucleic acids research*, 41(22):10641–10658, 2013.

[20] Petr Šulc, Flavio Romano, Thomas E. Ouldridge, Lorenzo Rovigatti, Jonathan P. K. Doye, and Ard A. Louis. Sequence-dependent thermodynamics of a coarse-grained dna model. *The Journal of Chemical Physics*, 137(13):135101, 2012.