Boston University
Electrical & Computer Engineering
EC464 Capstone Senior Design Project

User's Manual

High Performance Molecular Dynamics on Various Parallel Architectures

Submitted to

Martin Herbordt
8 St. Mary's St
Boston, MA 02215
(617) 35309850
herbordt@bu.edu

by

Team 17

Team Members

Vance Raiti vraiti@bu.edu
Austin Jamias ajamias@bu.edu
Emika Hammond eth@bu.edu
Sora Kakigi skakigi@bu.edu
Fadi Kidess fadik@bu.edu

Submitted: (April 18, 2025)

User Manual

Table of Contents

Executive Summary

Molecular dynamics is a technique in computational biology that simulates physical systems of interest at the particle level: integrating intermolecular forces using simple Newtonian mechanics over a series of discrete timesteps. Providing a rich understanding of the structure and dynamics of subject systems, these simulations are of great interest to biologists, and their extreme computational demand makes their optimization of great interest to computer engineers. While MD has some characteristics that lend themselves well to the highly parallel architectures of today's accelerators, the dense matrix of interactions when computing the short-range non-bonded Lenard Jones potential makes optimization of MD a nontrivial task. In this project, we develop optimizations for MD on three accelerators: CPU, GPU, and FPGA.

1. Introduction

Molecular dynamics (MD) is a technique in computational biology that simulates physical systems at the particle level: integrating pairwise intermolecular forces using simple Newtonian mechanics over a series of discrete timesteps. Providing a rich understanding of the structure and dynamics of subject systems, these simulations are of great interest to biologists. Given their extreme computational demand, their optimization is of great interest to computer engineers.

At a high level, MD involves computation in two phases for each timestep: a force computation phase and a motion integration phase. The motion integration phase includes only a small amount of arithmetic per particle as the velocity and position for each particle is updated given the particle's computed force. The force computation phase is also largely innocuous, with most forces being computed from chemical bonds that require relatively simple bookkeeping to do correctly. However, the non-bonded Lenard-Jones potential, with its all-particle pairwise interaction, is the main culprit in the steep cost of the large-scale MD simulations we are interested in optimizing.

While MD has some characteristics that lend themselves well to the highly parallel architectures of today's accelerators, the dynamic and relatively sparse nature of the short-range non-bonded Lenard Jones potential makes optimization of MD a nontrivial task. In this project, we investigate optimizations for MD on three accelerators: CPU, GPU, and FPGA.

We find GPU to be the most performant, with its massive SIMD and rich ecosystem of development tools. However, all architectures are able to achieve performance within an order of magnitude of each other with proper optimizations.

System Overview and Installation

2.1 CPU

2.1.1 Program Description

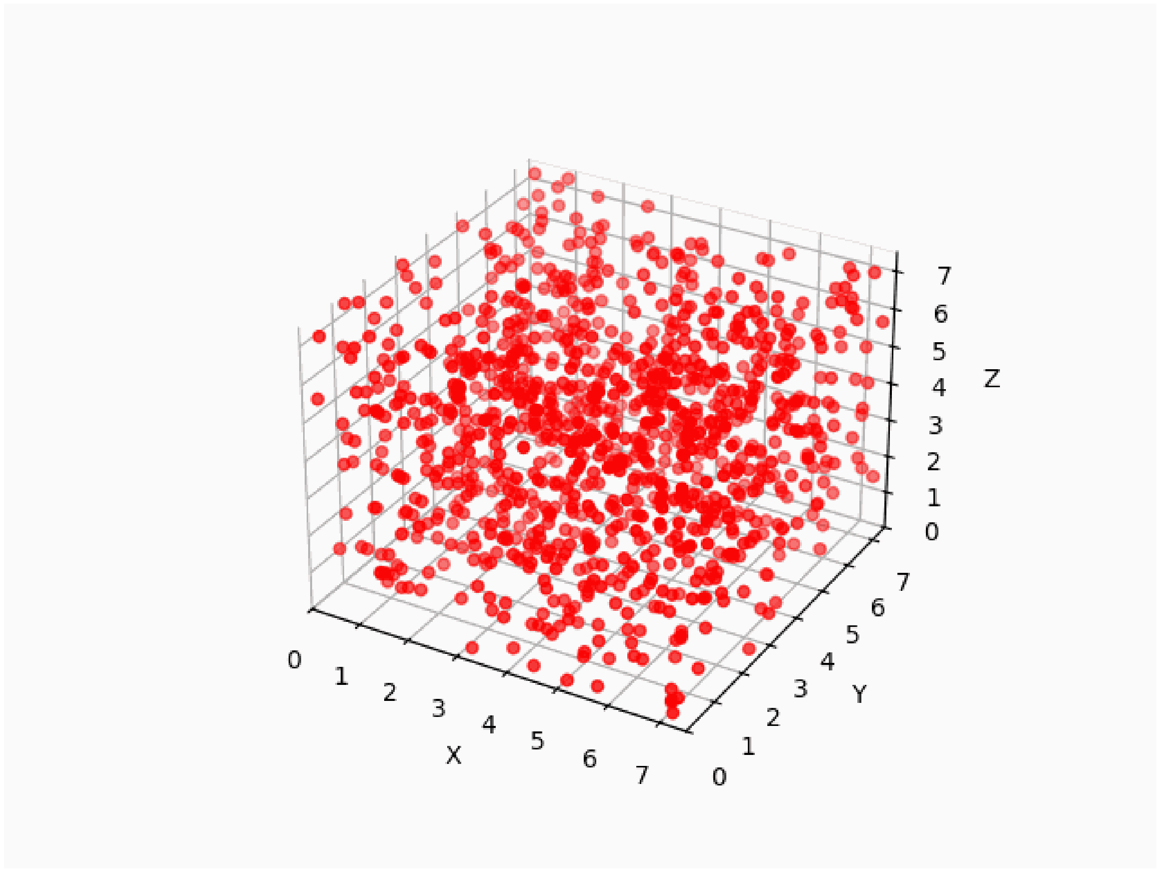
The CPU implementation is a Linux binary using Pthreads multithreading and AVX2 SIMD. It takes as input command-line configurations and an optional PDB and produces a table of particle positions at regular intervals during the simulation.

2.1.2 Usage

The command line interface allows configuration of various physical parameters (e.g. those used to compute the LJ interaction between particles or the attributes of the periodic simulation box), simulation parameters (attributes controlling simulation length, granularity, and initial conditions), and computer system parameters (threads, input and output paths).

2.1.3 Sample Output

If the program is configured to produce output, it will produce a plain-text table that can be passed to a python script that will render an animation of the simulation:



2.1.4 Installation and Usage

To install:

1. Download the github repository: https://github.com/md-hpc/cpu-1
2. Invoke the Makefile with `make`
3. (for visualization) install required packages with `pip install -r requirements.txt`

2.2 FPGA

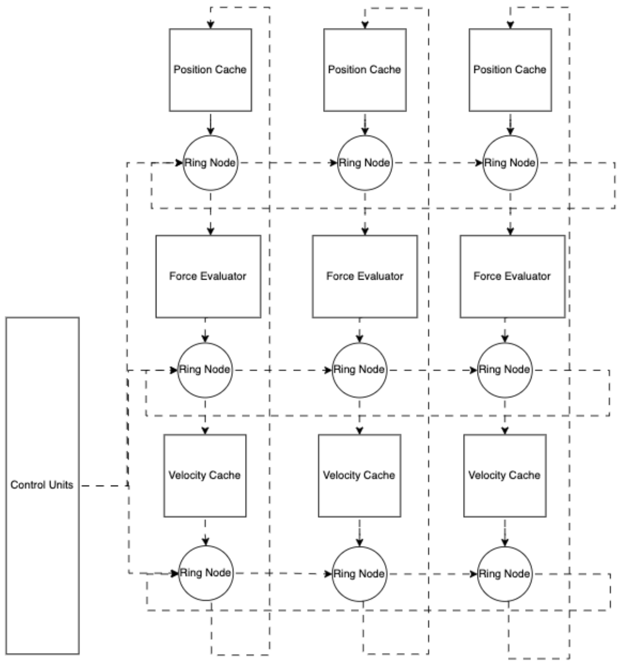
2.2.1 Program description

A host program running on a PC connected to an Alveo U280 FPGA is executed. The program first initializes the memory on the FPGA from a file called BRAM_INIT.txt, containing input in hexadecimal format.The PC then receives the output from the FPGA after an MD iteration is completed and writes the updated particle positions in a new text file for each iteration in the dump/ directory.

- Host setup: Run config.sh to setup the host environment and download the required libraries
- Execution: The user runs the host program run.py which executes the MD simulation, running it on the FPGA.
- Timing analysis: The host program will output the total runtime as well as average time per MD iteration.
- Visualization: Results are plotted using viz.py, which produces animations from the output csv file.

2.2.2 Physical Description

The figure below shows a clear representation of our FPGA design. The design consists of position and velocity caches connected in a ring node topology. The position data is read by the force evaluator which produces the updated velocities using the Lennard-Jones equation and writes to the velocity caches. Then, the next position is calculated based on the current position and velocity data.



2.2.3 Installation, setup, and support

To install:

1. Download the github repository: https://github.com/md-fpga
2. Invoke the Makefile with `make`
3. Copy the bitstream file to the OCT host using ‘scp’
4. Run ./config.sh to setup the OCT environment
5. Run run.py to run the host program and execute the MD simulation

2.3 GPU

2.3.1 Program Description

The GPU implementation consists of several CUDA C/C++ executables compiled with NVIDIA CUDA Compiler (nvcc), each representing a different optimization strategy for simulating particle interactions using the Lennard-Jones (LJ) potential.

Each executable runs on NVIDIA GPUs with the CUDA Toolkit installed and outputs particle position data at fixed simulation intervals or times the execution of the kernel.

The simulation reads input particle configurations from .pdb files, simulates interactions across time steps, and writes results as a .csv file suitable for visualization.

The attributes of the pdb file are used to compile by using the -D flag.

2.3.2 Usage

GPU simulations are controlled through shell scripts and command-line parameters. The workflow is designed for reproducibility and batch execution.
Configurations:
Input

- PDB files generated by vis.py

Execution

- Can compile files with Makefile and run on GPU directly
- Or submit jobs via SLURM. Be sure to change the following variables as needed: TIMESTEPS, TIMESTEP_DURATION, TYPE
    - scripts generated by script_generator.sh
    - jobs submitted by batch_job.sh
    - outputs obtained by get_results.sh

2.3.3 Installation, setup, and support

Import pdb files or create randomly generated input files using generate_pdb.py. Assuming that a user has provided a file that contains particle data called input.pdb, then the user can compile with nvcc or the Makefile provided. When compiling an nsquared related program, it is important to have the -D UNIVERSE_LENGTH=# flag with the number of Angstroms in each dimension replacing the "#". For cell list related programs, ensure the following flags are defined: -D CELL_CUTOFF_RADIUS_ANGST=# -D CELL_LENGTH_X=# -D CELL_LENGTH_Y=# -D CELL_LENGTH_Z=#

To create visualization of every implementation and example particle counts, pdb files can be generated using the python script mentioned above, and can be put into an input file like so: input/random_particles-1024.pdb.

Scripts with different parameters (ie. simulation dimension lengths, implementations, input files) can be generated so implementations can be run as batch jobs.

3. Operation of the Project

3.1 CPU

3.1.1 Normal Operation: Command Line Arguments

The CPU program is a Linux executable accessible by the command-line with the following options. <*name*: *type* = *default*> indicates that a value must be passed for that command-line argument where *name* is the variable that receives the value, *type* is the data type that the value is cast to, and *default* is the default setting for the variable. For float types, either decimal (e.g. 12.3) or floating point (e.g. 1.23e1) values may be provided. For options that act as a switch (e.g. --save), the default value is always 0 or false.

- --sigma <SIGMA: float = 1>: sets the value used for σ when computing LJ force
- --epsilon <EPSILON: float = 1>: set the value used for ε when computing LJ force
- --cutoff <CUTOFF: float = 2.5>: set the cutoff radius to apply when computing LJ force
- --universe-size <UNIVERSE_SIZE: int = 3>: set the length of the periodic simulation box to UNIVERSE_SIZE*CUTOFF
- --particles <PARTICLES: int = -1>: (ignored when --pdb is used) set the number of particles to generate for simulation. If PARTICLES == -1, the program will generate 80 * UNIVERSE_SIZE ^ 3 particles (so that the average density is 80 particles per CUTOFF ^ 3). Particles will be generated with a uniform random distribution within the periodic simulation box using random seed SEED
- --timesteps <TIMESTEPS: int = 10>: set the number of timesteps to simulate. Total simulated time will be DT*TIMESTEPS
- --dt <DT: float = 1e-7>: set the time-granularity of the simulation. For a given particle i at timestep t with velocity v(t) and position r(t), r(t+1) = r(t) + v(t) * dt. Total simulated time will be DT * TIMESTEPS
- --seed <SEED: int = 0>: (ignored when --pdb is used) set the random seed with which initial particle positions are generated
- --resolution <RESOLUTION: int = 100>: (ignored when --save is not used) set the number of timesteps between each saved frame of simulation.
- --threads <THREADS: int = 16>: set the number threads across which to partition the computational load. This value is capped at the number of hardware threads as determined by the return value of sysconf(_SC_NPROCESSORS_ONLN).

- ❖ --path <PATH: string = “particles”>: (ignored when --save is not used) set the filesystem path used for output.
- ❖ --save: if specified, write current particle positions to PATH every RESOLUTION timesteps
- ❖ --pdb <PDB: string = “”>: set the filesystem path from which to read initial particle positions. If PDB == “”, PARTICLES particles are randomly generated using seed SEED and a uniform distribution.

3.1.2 Abnormal Operation: Special Inputs and Outputs

As this is a computer research application intended to explore acceleration of MD on CPU, we do not try to be particularly useful to scientists wishing to use our application for real simulation purposes. However, if you would like to use this software to simulate specific systems of particles and analyze their behavior, we provide functionality to do so.

As specified in 3.1.1, --pdb <PDB> will point the program to a path that it will interpret as a PDB file, using particle positions as initial positions for particles in the simulation. All particles have an initial velocity of 0 (as is convention for MD simulations) and are assumed to be homogenous (as is not convention for MD simulations). To simulate the properties of a particular element or compound, the appropriate SIGMA, EPSILON, and CUTOFF should be set.

As specified in 3.1.1, --save --path <PATH> will point the program to a path that it will use to record the simulation. The file at this path can then be passed to the script `viz/viz.py` to produce a GIF of the animation such as the one seen in section 2.1.3

3.1.3 Security Concerns

This program has not been tested or analyzed for use in a secure environment. Please do not run this program with administrative privileges. All of the hardware capabilities it needs to function are granted to all regular user processes. Please do not feed arbitrary untrusted input to this program (e.g. by allowing users to upload arbitrary PDB files to pass to this program), as no checks are in place to secure the program against malicious input.

3.2 FPGA

3.2.1 Normal Operation

The FPGA system consists of a python host program that loads the FPGA bitstream (analogous to a computer executable) to our target FPGA. Since the hardware design is fixed, variables including dT and epsilon are also fixed. The host program can control whether or not the FPGA continues onto the next iteration after completing the current iteration, allowing it to receive all of the update particle positions before overriding their values.

3.2.2 Abnormal Operation: Special Inputs and Outputs

To achieve maximum performance, the user can choose to step through a specific number of operations without reading the particle positions until the last iteration is complete, reducing time spent on output transmission. This is done by simply removing the transmission algorithm only from the host python program.

3.2.3 Security Concerns

This program has not been tested or analyzed for use in a secure environment. Please do not run this program with administrative privileges. All of the hardware capabilities it needs to function are granted to all regular user processes. Please do not feed arbitrary untrusted input to this program (e.g. by allowing users to upload arbitrary TXT files to pass to this program), as no checks are in place to secure the program against malicious input.

3.3 GPU

3.3.1 Normal Operation

The simulation supports several compiled CUDA modes for evaluating molecular dynamics using different optimization strategies.

The available simulations are:

nsquared: Baseline brute-force N^2 interaction calculation. Each thread calculates pairwise force interactions naively.

nsquared_shared: Shared memory optimization of the N^2 version.

nsquared_n3l: Optimization using shared memory and reducing calculations by half by using Newton’s 3rd law.

cell_list: Uses spatial partitioning into cells to reduce pairwise checks

cell_list_n3l: Cell_list but with Newton’s 3rd Law optimization.

These simulations can be run individually on a GPU or scheduled as a batch job on a shared computing platform with `batch_job.sh`.

3.3.2 Abnormal Operation: Special Inputs and Outputs

Having a non-pdb file as input has undefined behavior. Out-of-Bounds particle placement may occur if universe size or cutoff radius is misconfigured. This can be corrected by either adjusting particle generation or the spatial parameters. Memory access violations can also occur if too many particles are placed into a single cell. This is put in place to prevent particles being too close to each other creating infinite velocity, and to help with performance having static sized cells. Hanging jobs can also be checked through logs. Also, NaN and inf values can occur due to floating point arithmetic by either dividing by a small number to get NaN or multiplying two big numbers in magnitude to get inf or -inf. If this is the case, generally this can be solved by modifying the TIMESTEP_DURATION parameter.

3.3.3 Security Concerns

This program has not been tested or analyzed for use in a secure environment. Please do not run this program with administrative privileges. All of the hardware capabilities it needs to function are granted to all regular user processes. Please do not feed arbitrary untrusted input to this program (e.g. by allowing users to upload arbitrary TXT files to pass to this program), as no checks are in place to secure the program against malicious input. It is possible that excessive usage may throttle shared computing environments.

4. Technical Background

4.1 Theory

4.1.1 Lennard-Jones Potential

The high level sequence of an MD simulation involves two phases per-timestep: force computation and motion integration. The mathematics of motion integration is important for good convergence, but as the actual arithmetic is simple, it's not considered further here.

Within force computation, most force terms also involve only a small amount of arithmetic. Most compute is spent, instead, computing the non-bonded Lenard-Jones potential that models the weak attraction and strong repulsion all pairs of atoms experience. It is parameterized with two physical constants, sigma and epsilon, as described by the following equation:

$$V_{\mathrm{LJ}}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

Because the LJ potential is pairwise, a naive computation would grow quadratically in complexity with problem size.

4.1.2 Periodic Boundary Conditions

Correct modeling of boundary conditions is essential for producing realistic results. In MD, a periodic boundary condition is traditionally used. i.e. our mathematical model assumes that our universe is some periodic simulation box that is replicated infinitely in each direction. Practically, this means that if a particle moves outside one face of the periodic box, we simulate it “coming out” of the opposite face with an unchanged velocity. This also means that the interaction of a particle A with a particle B happens between A and the periodic replica of B that is closest to A. To do so, we use the following distance function:

```
float dist(float a, float b) {
        float dists[3] = {a - (b - L), a - b, a - (b + L)};
        float abs_dists[3] = {abs(opts[0]), abs(opts[1]), abs(opts[2])};
        int i = 0;
        int min = abs_dists[0];
        i = abs_dists[1] < abs_dists[i] ? 1 : i;
        i = abs_dists[2] < abs_dists[i] ? 2 : i;
        return opts[i];
}
```

Applied to each coordinate of the interacting pair.

4.1.3 Cutoff Radius

As stated in section 4.1.1, the Lenard-Jones potential is pairwise and naively of quadratic complexity to compute. This would make computation of almost any system of interest prohibitively expensive. However, since the potential converges rapidly to 0 as distance increases, we are able to apply a cutoff radius (henceforth referred to as R) to the force. Assuming uniform density of particles, this means that an approximately constant amount of work must be done to compute the LJ force per-particle.

4.1.4 Cell Lists

Simply applying a cutoff radius to the LJ computation is not enough to achieve scalability. If we perform a pairwise distance check between all particles and only compute the LJ force between particles that are < R away from each other, we have saved computation, but O(n^2) arithmetic operations are still being performed. In order to make the filtering efficient as well, we can use cell lists, which bin particles by their x, y, and z coordinates into “cells” of side length R. Conceptually, these cells tile the simulation box in a regular 3D grid.

This is useful because for a given “reference” particle in a given “home” cell, we know that we must consider only particles that reside within cells sharing a face, edge, or vertex with the home cell, allowing us to finally achieve our O(n) LJ computation.

4.1.5 Neighbor Lists

A further optimization of range-limited force computation with cutoff involves saving, for each particle in the simulation, a much more compact list of “neighbors” that that particle needs to interact with for several timesteps.

Using cell lists, each “reference” particle will interact with all particles residing in a box of side length 3R centered on that particle’s cell. Given that the only “neighbor” particles that the reference particle actually must interact with are contained within a sphere of radius R centered on that particle, this means that the LJ potential (or at least the distance comparison) is done on nearly 6.5x more particles than is necessary given our cutoff radius.

Neighbor lists addresses this problem by constructing a neighbor list periodically (typically once every 20 timesteps or so) that specifies the exact “neighborhood” of particles that each particle should be interacting with.

While this method promises very efficient numerical computation, the fine-grained memory access patterns and large memory overhead can make this method difficult to implement on hardware platforms with very large SIMD (e.g. GPU) or small on-chip memory (e.g. FPGA), so this algorithm is commonly only seen in CPU MD applications.

4.2 CPU

4.2.1 Parallelism

Parallelism in a CPU is broken down into three main parts: instruction-level, data-level, and task-level (There are others, but they are unused in this project)

Instruction-level parallelism is the use of multiple instructions at a time. This support is built into the hardware through superscalar-pipelining, as well as out of order execution in modern processors. Superscalar-pipelining is the usage of multiple compute pathways depending on the instruction, as well as the issue of multiple instructions at once. The pipelines are stages are small parts of a full instruction that allow for multiple instructions to be on the same compute path, without having to wait for each instruction to fully complete. However, some data dependencies create hazards in the pipeline, so modern processors rely on out-of-order execution to remedy these issues. To support this functionality, techniques such as loop unrolling are useful to expose more instructions at a time to the CPU to schedule such that there are minimal data dependencies (this is done implicitly using compiler flags e.g., -O3)

Data-level parallelism is the process of performing the same operation on multiple data elements simultaneously. This is also called single instruction, multiple data (SIMD), and is commonly implemented using vectorization. The technique of vectorization in Intel is done using SSE or AVX instructions, the latter of which is used in this implementation. These allow for bundles of eight floating-point values in our case (but can be extended to 4 double precision), being computed at once.

Task-level parallelism is the distribution of the larger problem into smaller components so they run on different threads of a CPU or are distributed onto multiple CPUs. This is done using POSIX threads for multi-threading and OpenMPI for passing messages through multiple CPUs.

4.2.2 Memory

Memory in a CPU is a hierarchical structure where there are multiple levels of memory to balance the performance-to-size ratio. In this project, the levels considered are L1 Cache, L2 Cache, L3 Cache, and Main Memory, with the latencies of 1-3, 4-10, 10-40, and 60-100 cycles, respectively. The way memory is accessed is key to the performance of the program, which is why algorithmic optimizations such as cell/neighbor-lists are used to minimize the memory overhead of each element.

One special case is when vectorizing. To vectorize optimally, the usage of data structures that allow for access of the same portion of multiple data points (or particles in this case) is important to access consecutive and contiguous data. Accessing data in this way allows for better locality as we fetch more useful data out of each cache block. This is implemented through the use of separate data structures for x, y, z position and velocity coordinates.

5. Relevant Engineering Standards

5.1 Protein Data Bank format

The protein data bank (PDB) format is a file format used to specify the structure of a molecular system of interest for simulation by an MD engine. It's a plaintext document, but it encodes some sophisticated hierarchical structures. As we are only interested in simulating our simple, workhorse example of a homogenous gas, we interpret all entries in the PDB file as simple particles.

5.2 CPU Programming APIs

5.2.1 POSIX Threads

POSIX Threads (or pthreads) is a standard interface for multithreaded applications on CPU. It's implemented in GNU C and supported by the Linux operating system. It defines interfaces for spawning and synchronizing threads in a fast, portable manner suitable for high-performance applications

To spawn a thread, invoke pthread_create with an initial routine. For high performance applications, thread affinity should be configured with pthread_setaffinity_np: the Linux scheduler will migrate threads freely if this is not done, causing all sorts of problems with cache.

Once the threads are running, synchronization APIs can be used to prevent race conditions. We use pthread_barrier_wait, which causes all threads to wait for all other threads before proceeding. This prevents threads from proceeding with motion update before all forces have been computed, computing different timesteps, etc.

5.2.2 GCC Vector Extensions

To access the SIMD capabilities of modern CPUs within a standard coding environment, we use a mixture of GCC vector extensions and Intel Intrinsics. Vector extensions allow arrays of data to be treated as vector data types in such a way that they are automatically mapped to vector registers and instructions. To enable these features, one only needs to define their desired variables with __attribute__ ((vector_size(n))) and take care to write code with the proper alignment considerations.

5.2.3 Intel Intrinsics

Intel Intrinsics are a standard C API supported by GCC. The standard defines a large library of functions that operate on special vector data types (__m256, __m256i, etc.). They function as thin wrappers around AVX assembly instructions. While more difficult to use and less portable than the GCC vector extensions, they allow for element-wise conditional moves and are thus useful for functions that require comparisons.

5.3 GPU Programming APIs

5.3.1 NVIDIA CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It allows developers to write programs that executes across many threads for high-performance computing workloads.

CUDA extends C/C++ with a few keywords and APIs that expose GPU memory hierarchies, thread management, and data transfer. A CUDA program defines kernels which are functions that run in parallel across many threads.

Threads are organized into warps, blocks, and grids. Thread hierarchies are exploited in CUDA code to optimize computations.The memory hierarchy consists of global, shared, local, and constant memory which can be exploited as well.

5.3.2 CUDA Shared Memory

Shared memory is low-latency memory shared among threads in a thread block. It is managed by the programmer, and it allows for reuse of data and minimizes access to global memory. Threads need to be synchronized via the CUDA intrinsic __syncthreads() to ensure data is visible across all threads. Improper synchronization leads to race conditions and incorrect results.

6. Cost Breakdown

Project Costs for Production of Beta Version (Next Unit after Prototype)				
Item	Quantity	Description	Unit Cost	Extended Cost
NVIDIA A40	1	The GPU used for testing, provided on SCC	0	0
Intel Xeon Gold 6242	2	The CPU used for testing, provided on SCC (multiple nodes for MPI usage)	0	0
Alevo U280	1	FPGA used for testing, provided on the NERC through the Open Cloud Testbed	0	0
VTune	1	Intel Performance Profiler Tool	0	0
Caliper	1	Performance Instrumentation / Measurement Tool	0	0
			Beta Version-Total Cost	0

Budget Narrative

Since our project is done completely in software (hence has no cost), listed above is instead the hardware used and some performance tools that would be useful in future iterations of this project.

It is important to note that the CPU and GPU can be interchanged for others; however, maintaining consistency is important to evaluate how optimizations to the code change performance properly. The FPGA's resource characteristics (such as number of BRAMs and Registers) are the deciding factors when determining parameters (e.g, universe size, force pipeline count), so changing the FPGA used would heavily affect design decisions.

VTune is Intel's performance-profiler software that provides algorithm optimization, microarchitecture/memory bottlenecks, and accelerator bottlenecks. This may be useful when understanding limitations in performance due to memory access or time-consuming portions of code.

Cailper is an open-source performance analysis toolbox that provides a suite of tools for measuring performance, including timers, data capture, and collections to popular third-party tools such as the aforementioned VTune.

7. Appendices

7.1 Appendix A - Specifications

Requirement	CPU	GPU (cell list n3l)	FPGA
Maximum error per particle (from serial baseline)	< 1e-4	< 1e-3	< 1e-4
Time per particle per timestep	500 ns	100 ns	150 ns
Maximum particles per simulation	1e+8	128K	2k

7.2 Appendix B – Team Information

Emika Hammond is a graduating Computer Engineering student with a passion for high performance computing, infrastructure engineering, and operating systems. After graduation, she will be joining Red Hat as a Site Reliability Engineer.

Sora Kakigi is a Computer Engineering student interested in computer architecture and high-performance computing. He is attending NYU for his master's in computer engineering after graduation.

Vance Raiti is a Computer Engineering student interested in high-performance computing, operating systems, and cloud computing. He will be working in the Emerging Technologies department at Red Hat after graduation.

Austin Jamias is a computer engineering student interested in cloud computing, high-performance computing, and operating systems. Following graduation, he will obtain his masters in electrical and computer engineering at BU while working full time at Red Hat in the research department.

Fadi Kidess is a computer engineering student interested in embedded systems, logic design, and high performance computing. He will be continuing his education at BU after graduation to pursue his PhD in computer engineering under the supervision of professor Rabia Yazicigil.