Hardware Implementation of non-bonded Forces in Molecular Dynamics Simulations

Álvaro José Caicedo Beltrán

Thesis submitted as partial fulfillment of the requirements for the degree of Bachelor in Electronics Engineering

Universidad del Valle
College of Engineering
School of Electrical and Electronics Engineering
Santiago de Cali, Colombia
December, 2011

Approval Sheet

NOTES

Approved by:

Committee member:
Jaime Andrés Arteaga, M.Sc.

Advisor:
Jaime Velasco Medina, Ph.D.

Committee member:
José Ferney Rivera, M.Sc.

Advisor:
John Michael Espinosa, M.Sc.

Abstract

Molecular Dynamics is a computational method based on classical mechanics to describe the behavior of a molecular system. This method is used in biomolecular simulations, which are intended to contribute to the study and advance of nanotechnology, medicine, chemistry and biology. Software implementations of Molecular Dynamics simulations can spend most of time computing the non-bonded interactions.

This work presents the design and implementation of an FPGA-based coprocessor that accelerates MD simulations by computing in parallel the non-bonded interactions, specifically, the van der Waals and the electrostatic interactions. These interactions are modeled as the Lennard-Jones 6-12 potential and the direct-space Ewald summation, respectively. In addition, this work introduces a novel variable transformation of the potential energy functions, and a novel interpolation method with pseudo-floating-point representation to compute the short-range forces. Also, it uses a combination of fixed-point and floating-point arithmetic to obtain the best of both representations.

The FPGA coprocessor is a memory-mapped system connected to a host by PCI Express, and is provided with interruption capabilities to improve parallelization. Its main block is based on a single functional pipeline, and is connected via Avalon Bus to other peripherals such as the PCIe Hard-IP and the SG-DMA. It is implemented on an Altera's EP2AGX125EF35C4 device, can process 16k particles, and is configured to store up to 16 different types of particles. Simulations in a custom C-application for MD that only computes non-bonded forces become up to 12.5x faster using the FPGA coprocessor when considering 12500 atoms.

Acknowledgements

I would like to thank primarily my advisors for their support during the development of this thesis, and because in their lectures I have learned a lot of their long experience and wide knowledge in advanced digital system design. I thank Prof. Velasco for his vision towards research and motivation to do things better and also because he accepted me in the Bionanoelectronics Research Group. I thank Prof. Espinosa for his suggestions related to Molecular Dynamics and his strong recommendations about optimizations of hardware architectures.

I also would like to thank my committee for their interest in the evaluation of this work. Thanks to Altera Corporation and its University Program for providing our research group with software and hardware tools for academic purposes. I also thank Texas Instruments Germany, because during my internship I learned a lot about testbenches and hardware emulation.

In a very especial way, I want to thank the guys in the Bionanoelectronics Research Group for sharing with me their valuable knowledge and tools in many helpful aspects: Adolfo "Fito" for introducing me to ModelSim-Altera, Claudia for her introduction to SignalTap II, Jorge Guerrero for providing us with LaTeX templates for the documentation and presentation, and Juan for his enthusiasm in many discussions about hardware and science. Some of the guys were reviewers of this thesis, and their suggestions were totally welcome.

Beyond the technical aspects, I feel very proud of this thesis, since it has been the result of perseverance, dedication, and love to the work. I also thank God for giving me life, health, strength and hope. I would like to thank my parents, my sisters and my closest family for their love, care, support and company. I thank my lovely, sweet girlfriend Aleksandra for her company in the good and in the bad times, for her patience, for her warm love, and for her charming smile. Finally, I thank my friends for their invaluable friendship along my life and for all those unforgettable nice times, and I thank my lecturers of my universities for their guidance along my study.

This work is dedicated to my parents and to my dear Aleksandra.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 2

1.3 Thesis organization 2

2 Molecular Dynamics **5**

2.1 The Molecular Dynamics method 5

2.1.1 Discrete time integration 6

2.1.2 Force field 8

2.1.3 Boundary conditions 10

2.2 Non-bonded interactions 11

2.2.1 Van der Waals interactions 11

2.2.1.1 Lennard-Jones 6-12 potential 12

2.2.1.2 Exponential-6 potential 13

2.2.2 Electrostatic interactions 13

2.2.2.1 Ewald summation for electrostatic potential 14

2.2.3 Truncation of the potential energy function 15

2.3 Software packages for Molecular Dynamics simulations 16

3 Hardware Design of the FPGA coprocessor **19**

3.1 Design methodology 19

3.2 Variable transformation of the potential and force functions 21

3.2.1 Transformation of Lennard-Jones 6-12 22

3.2.2 Transformation of the direct-space Ewald summation 22

3.3 Polynomial interpolation using pseudo-floating-point representation 23

3.4 Datapath of the LJEwDir core 27

3.4.1 Interpolation engines 29

3.4.2 Storage elements 31

3.4.3 Computation of the squared distance . . . 33
3.4.3.1 Minimum Image Convention - MIC . . . 33
3.4.3.2 Squared Distance . . . 34
3.4.4 Floating-point units . . . 34
3.4.4.1 Single-precision floating-point multiplier and adder . . . 34
3.4.4.2 Single-precision floating-point accumulator . . . 34
3.4.5 Floating point/Fixed point conversion . . . 35
3.4.5.1 Fixed point to Floating point converters . . . 36
3.4.5.2 Floating point to Fixed point converters . . . 36
3.5 Controller of the LJEwDir core . . . 37
3.6 Memory-Mapped LJEwDir Core . . . 40
3.7 Top Level and the Avalon Memory-Mapped System . . . 41

4 Hardware Verification and Synthesis of the FPGA coprocessor 45
4.1 Simulation of the LJEwDir core in ModelSim and Matlab . . . 45
4.1.1 Simulation of the functional building units . . . 45
4.1.1.1 Minimum Image Convention - MIC . . . 45
4.1.1.2 Squared Distance . . . 46
4.1.1.3 Interpolation Engine . . . 47
4.1.1.4 Floating-point units . . . 47
4.1.1.5 Pair Controller . . . 49
4.1.2 Simulation of the Datapath . . . 49
4.1.3 LJEwDir Core . . . 52
4.2 In-System hardware verification of the FPGA coprocessor . . . 52
4.3 Resource usage and Timing analysis . . . 53
4.3.1 Resource usage . . . 54
4.3.2 Timing analysis . . . 54

5 Software Application for Molecular Dynamics Simulations 57
5.1 Description of the software application . . . 57
5.1.1 Molecular Dynamics algorithm . . . 57
5.1.2 Computation of non-bonded interactions . . . 59
5.1.3 Configuration options . . . 61
5.2 Accuracy and Speed up . . . 62
5.2.1 Accuracy . . . 63
5.2.2 Speedup . . . 64

6 Comparison with Previous Works **67**
6.1 Previous works . . . 67
6.2 Characteristics of this work . . . 69

7 Conclusions and Future Work **71**
7.1 Conclusions . . . 71
7.2 Future work . . . 72

A Lennard-Jones 6-12 Engine with Floating-Point Arithmetic **75**
A.1 Implementation with floating-point units . . . 75
A.1.1 FP_LJ Model 1 . . . 76
A.1.2 FP_LJ Model 2 . . . 76
A.2 Comparison between the FP_LJ Model 1 and 2 . . . 77
A.3 Conclusions . . . 78

B Tables of Execution Time **81**

C Arria II GX Development Board **85**

Bibliography **87**

List of Figures

2.1 Evolution and resource usage of the modified Velocity Störmer-Verlet. [17] . . . 8
2.2 Potential energy models: a) Bonds, b) Angles, c) Torsions, d) Non-bonded.[29] 9
2.3 Particles in a 2-D periodic system. a) Moving and wrapping. b) The minimum image convention. 11
2.4 Van der Waals interactions: Lennard-Jones 6-12 and Exponential-6 potentials with normalized parameters ($\epsilon_{ij} = 1$ and $\sigma_{ij} = 1$). 12

3.1 HW/SW task division of the MD application. 20
3.2 Example of logarithmic and inner linear partitions in range $[2^{-3}, 2^{1})$. 25
3.3 Relative error for the interpolation of $f^{LJ}\left(w^{LJ}\right)$ with respect to the interpolation order and the word length for 8 linear partitions and convergent rounding. a) Mean, b) Variance. 26
3.4 Polynomial coefficients and exponents for the interpolation of $f^{LJ}\left(w^{LJ}\right)$. a) Coefficients, b) Exponents. 27
3.5 Interpolation of the normalized function $f^{LJ}\left(w^{LJ}\right)$. a) Normalized output, b) Normalized output for each polynomial, c) Normalized output w.r.t. the normalized input, d) Histogram of errors. 28
3.6 Overview of the functional pipeline. 28
3.7 Block diagram of the Interpolation Engine connected to Interpolation Input decoder. 30
3.8 Block diagram of the memories. a) 2-port memory, b) 3-port memory. 31
3.9 Block diagram of the MIC unit. 33
3.10 Block diagram of the Squared Distance unit. 34
3.11 Block diagram of the floating-point units for multiplication and addition. . . . 35
3.12 Block diagram of the FPSingleAcc unit. 35
3.13 Generic architecture of the fix2single converter for 32-bit fixed-point data. . . . 37
3.14 Generic architecture of the single2fix converters. 37
3.15 Block diagram of the Pair Controller. 38

3.16 State of the signals controlled by the Pair Controller for $N = 5$. a) Particle counters, b) Fload, c) Fsave, d) Uen, e) Uload, f) Usave. 39
3.17 State diagram of the Pair FSM. 39
3.18 State diagram of the Interruption FSM. 40
3.19 Overview of the memory-mapped LJEwDir core. 40
3.20 System overview. 42

4.1 Constrained-random verification of the MIC. a) Inputs, b) Software and hardware outputs, c) Error bits. 46
4.2 Constrained-random verification of the SquaredDistance unit. a) Coordinates of the input position, b) SW and HW outputs, c) Relative error. 46
4.3 Verification of the Interpolation Engine together with the Interpolation Input for the function $u^{LJ}\left(w^{LJ}\right)$. a) SW and HW outputs, b) Error, c) Relative error. 47
4.4 Random verification of the FPSingleMult unit. a) Inputs, b) SW and HW outputs, c) Error. 48
4.5 Random verification of the FPSingleAdder unit. a) Inputs, b) SW and HW outputs, c) Error, d) Relative error. 48
4.6 Random verification of the FPSingleAcc unit. a) Inputs, b) SW and HW outputs, c) Error, d) Relative error. 49
4.7 Simulation results of the Pair Controller for 15 atoms. 50
4.8 Verification results of the distance between one particle and the rest of particles. a) SW and HW outputs, b) Error, c) Relative error. 50
4.9 Verification results of $U_{ij}^{LJ}\left(r_{ij}\right)$ between one particle and the rest with $\sigma_{ij} = 3.4050$. a) SW and HW outputs, b) Error, c) Relative error. 51
4.10 Verification results of $U_{ij}^{d}\left(r_{ij}\right)$ between one particle and the rest with$\alpha = 0.349\,\text{Å}^{-1}$. a) SW and HW outputs, b) Error, c) Relative error. 51
4.11 Verification results of the accumulation of $\mathbf{F}[2]$ for all 100 particles. a) SW and HW outputs, b) Error, c) Relative error. 52
4.12 Verification results of $\mathbf{F}[3]$ for 100 particles in the core. a) SW and HW outputs, b) Error, c) Relative error. 53

5.1 Snapshots of the testcases. a) 100 atoms b) 800 atoms c) 2700 atoms d) 6400 atoms e) 12500 atoms. This images were generated with VMD [23]. 62
5.2 Energy profile for a simulation of 100 Argon atoms during 1000 steps at 1 fs/step. a) Kinetic, potential and total energy, b) Potential energies. 64
5.3 Energy drift for a simulation of 100 Argon atoms for 1000 steps at 1 fs/step. a) Only software in double precision, b) With hardware. 64

5.4 Simulation time with $r_{cut} = L/2$ for 100, 800, 2700, 6400 and 12500 particles. a) Total simulation time, c) Average time in direct space, c) Average time in reciprocal space. Lower plots show time in logarithmic scale. 65
5.5 Simulation time varying r_{cut} between 18 Å and 45 Å for 12500 atoms. a) Total simulation time, c) Average time in direct space, c) Average time in reciprocal space. Lower plots show time in logarithmic scale. 66

A.1 Block diagram of the CalculateDistance unit. 76
A.2 Block diagram of the FP_LJ1 core. 77
A.3 Block diagram of the FP_LJ2 core. 78

C.1 Arria II GX development board. [2] 85

List of Tables

3.1 Final interpolation parameters. 27
3.2 32-bit address space of the LJEwDir core for up to 16k particles and 16 elements. 41
3.3 CSR of the LJEwDir core. 42

4.1 Post-fitting results of the FPGA Coprocessor on the EP2AGX125EF35C4. . . . 55

5.1 Options of the MD application. 61
5.2 Preset values of some parameters for each testbench. 61
5.3 Parameter values of the elements in the topology file for the simulation. 63

A.1 Compilation results for the FP_LJ1 and FP_LJ2 cores on the Stratix III EP3SE50F780C2. 80

B.1 Execution time with $N = 100$, $L = 18.0$ Å, $nsteps = 1000$. 81
B.2 Execution time for different cutoff radii with $N = 800$, $L = 36.0$ Å, $nsteps = 1000$. 82
B.3 Execution time for different cutoff radii with $N = 2700$, $L = 54.0$ Å, $nsteps = 100$. 82
B.4 Execution time for different cutoff radii with $N = 6400$, $L = 72.0$ Å, $nsteps = 100$. 83
B.5 Execution time for different cutoff radii with $N = 12500$, $L = 90.0$ Å, $nsteps = 10$. 83

C.1 Characteristics of the EP2AGX125EF35 FPGA. 85
C.2 Characteristics of the Arria II GX board. 86

Glossary

ASIC Application-Specific Integrated Circuit.

BAR Base Address Register.

CPU Central Processing Unit.

CSR Control and Status Register.

DMA Direct Memory Access.

DNA Deoxyribonucleic Acid.

DSP Digital Signal Processor.

erfc Complementary Error Function.

FFT Fast Fourier Transform.

FIFO First In, First Out.

FLOPS Floating-point Operations per Second.

FP Floating Point.

FPGA Field Programmable Gate Array.

FSM Finite State Machine.

GPGPUs General Purpose Graphical Processing Unit.

GPU Graphical Processing Unit.

HW Hardware.

IBC Isolated Boundary Conditions.

IE Interpolation Engine.

II Interpolation Input.

IP Intellectual Property.

JTAG Joint Test Action Group. IEEE standard 1149.1.

LJ Lennard-Jones.

LSB Least Significant Bit.

MAD Multiply-Addition unit. Same structure than the MAC (Multiplier-Accumulator unit) without feedback to accumulate..

MD Molecular Dynamics.

MIC Minimum Image Convention.

MM Memory-Mapped.

MSB Most Significant Bit.

MSF Mean Square Fluctuation.

NoC Network on Chip.

ODE Ordinary Differential Equations.

PBC Periodic Boundary Conditions.

PC Pair Controller.

PCIe Peripheral Component Interconnect Express.

PDB Protein Data Bank file.

PIO Parallel Input/Output.

PLL Phase-Locked Loop.

PME Particle Mesh Ewald.

RAM Random-Access Memory.

RMS Root Mean Square.

ROM Read-Only Memory.

RTL Register-Transfer Level.

SG-DMA Scatter/Gather Direct Memory Access.

SOPC System on Programmable Chip.

SPME Smooth Particle Mesh Ewald.

ST Streaming.

SW Software.

TPF Topology File.

VHDL VHSIC Hardware Description Language.

VMD Visual Molecular Dynamics.

1. Introduction

1.1 Motivation

Molecular Dynamics (MD) is a computational method used in biomolecular simulations that is based on classical mechanics to describe the behavior of a set of particles. Such biomolecular simulations are intended to contribute to the study and advance of nanotechnology, medicine, chemistry, biology, and materials science.

MD simulations run for long time performing discrete integration of the Newtonian equations of motion and evaluation of force fields at a very fine time step. Software implementations of Molecular Dynamics simulations can spend the most of the time computing the non-bonded interactions. The computational complexity of non-optimized algorithms that compute these interactions is at least $O\left(N^{2+}\right)$. This does not represent a problem for small systems but becomes tedious for large systems such as proteins, DNA strands and viruses. For instance, the complete simulation of the Satellite Tobacco Mosaic virus took one month in a 256-node supercomputer [16]. Hence, accelerating the computation of such interactions leads to a significant reduce of time when performing long simulations.

Some approaches purpose new algorithms that reduce computational complexity, while other techniques center their attention in technological implementations to parallelize algorithms, to increase the number of processing units, to optimize memory accesses, etc. Most of the acceleration attempts have been implemented on PC-clusters, supercomputers, GPUs, and powerful ASICs for MD. However, these solutions are very expensive and power consuming.

As an alternative, FPGAs become very suitable for this problem of the High-Performance Computing due to their fast reprogrammability, high density, high speed, distributed embedded set of resources like memories and DSPs, as well as their high-speed IO links. Furthermore, there are hundreds of free IPs (Intellectual Properties) and libraries that reduce the design time. Besides standard hardware architectures, applications in FPGAs involve new programming concepts such as streaming, associative computing and massive parallelism, which can offer better solutions than software implementations for some specific problems. [21]

Taking into account the several advantages that come with the rapid prototyping in FPGAs, and the importance of reducing the runtime of MD simulations, this work presents the

design and implementation of an FPGA-based coprocessor that computes the short-range part of non-bonded interactions. This coprocessor pretends accelerate the computation of the Lennard-Jones 6-12 potential and the direct-space Ewald summation. These functions are mapped in pipeline architecture to carry out functional parallelism. The design methodology explores an implementation with fixed-point and floating-point representations to achieve the desired accuracy. This hybrid arithmetic brings the best of both representations: accuracy, low area, low latency, high throughput, and flexibility.

In order to validate performance and accuracy of the FPGA coprocessor, a C-application to run MD simulations was also developed during this thesis. This software application implements a simple force field that performs direct evaluation of the Lennard-Jones 6-12 potential and uses the Ewald summation method to compute the electrostatic potential in periodic systems. The FPGA coprocessor is connected to the host via PCI Express bus to reduce the impact of communication overhead, and is acknowledged by the host as a memory-mapped device.

1.2 Contribution

The contribution of this work is primarily focused on the acceleration of MD applications, which was achieved considering the fact that simulations with the FPGA coprocessor can run up to 12.5x faster than the original software implementation, and still showing an acceptable accuracy and stability. Other contribution is about the viability of FPGAs when dealing with High-Performance Computing. This was demonstrated in several ways such as the implementation of functional parallelism in high-throughput pipeline architecture, the implementation of custom arithmetic in fixed-point and floating-point representation, and high-speed communication with the host via PCI Express, also driven by the FPGA.

The third contribution is found in the attempt to implement the short-range part of non-bonded forces interactions in FPGA, because the novel variable transformation purposed and implemented in this work together with the pseudo-floating-point interpolation method offer a better mapping of those potential functions into an FPGA in comparison to others in the literature. Finally, this works contributes to other works that may not be directly related to MD, but that can still take advantage of the design methodology and architecture presented here.

1.3 Thesis organization

After this brief introduction, the rest of this thesis is organized as follows. Chapter 2 presents the theoretical background of Molecular Dynamics and the non-bonded interactions that

were implemented in hardware, as well as some of the available software packages for MD simulations. Chapter 3 provides the reader the hardware design methodology of the FPGA coprocessor, its functional characteristics, and its architecture from the functional building units until the top level description. Chapter 4 presents the verification methods that were used to ensure the correct functionality of the system, as well as synthesis and timing results of the implementation. Chapter 5 presents a C-application used for performance validation that runs simple MD simulations assisted by the FPGA coprocessor. Chapter 6 presents previous works mainly related to the acceleration of MD simulations using FPGAs, and summarizes the features and characteristics of this work. Finally, chapter 7 draws some conclusions and purposes the further work.

2. Molecular Dynamics

This chapter presents an introduction to Molecular Dynamics (MD), and explains deeper the time integration, force field and boundary conditions. It also presents the non-bonded interactions that are implemented in hardware. Finally, software packages for MD simulations are presented.

2.1 The Molecular Dynamics method

Molecular and biomolecular simulations contribute to the study and advance of nanotechnology, medicine, chemistry and biology. These simulations are carried out by two main approaches: one stochastic method known as Monte Carlo and one deterministic method called Molecular Dynamics. The Molecular Dynamics method is used in an intermediate time-space scale between the Quantum Chemistry (e.g. solving Schrödinger's equations) and the Monte Carlo method. MD simulations of (bio)molecular systems are used in order to simulate motion at an atomic level by using classical mechanics, i.e. solving the Newtonian equations of movement. Calculation of the motion in molecules is required for many reasons such as prediction of structures, understanding of interactions and properties, learning about normal modes of vibration, design of bio-nano materials, experimentation on what cannot be studied experimentally, and obtaining a movie of the interacting molecules. [29]

MD simulations provide detailed information about fluctuations and conformational changes in proteins and in nucleic acids. This method is currently used to investigate about the structure, dynamics and thermodynamics of complex biological molecules. Some applications are protein folding and ion transport.

MD assumes that the force $\mathbf{F} = -\partial V/\partial \mathbf{x}$ (due to the potential energy V) that acts upon the N particles (atoms and molecules) of a molecular system, is the same force in the Newton's second law $\mathbf{F} = m\mathbf{a}$. The trajectory of a particle is found by solving the second-order ordinary differential equation (ODE) in (2.1), where $V(\mathbf{x}(t)) = V(\mathbf{x}_1(t), ..., \mathbf{x}_N(t))$ is the potential energy function. This scalar function represents a $3N$-dimensional system, and depends on the position of the particles.[1]

[1] The potential energy $E_{pot} = V(\mathbf{x}(t))$ depends on the particle positions, while the kinetic energy $E_{kin}(t) =$

$$m_i \frac{d^2 \mathbf{x}_i(t)}{dt^2} = \mathbf{F}_i \equiv -\frac{\partial V}{\partial \mathbf{x}_i} \quad i = 1, \ldots, N \tag{2.1}$$

The analytical solution of this system becomes a very hard task, since biomolecular systems have thousands and even millions of particles. Therefore, the solution requires numerical methods and the discretization of the ODEs, as well as a force field that defines the potential energy in the system.

In a general description, MD is an iterative process of finding the potential energy between particles, and then moves them to the next state. MD simulations run for long time at very fine steps to describe trajectories. However, MD is mathematically ill-conditioned for long-time simulations, and generates cumulative errors due to the numerical integration that cannot be avoided even with infinite precision. This limits the time scale of this method.

An MD simulation can be treated as an experiment. This can be described as follows:

1. **System configuration:** Sample selection. (number of atoms, initial conditions, force field, boundary conditions, etc)

2. **Equilibrium:** Prepare the sample to reach some predefined pressure and temperature.

3. **Run simulation:** Execute for a certain number of steps: forces computation, integration, average of the properties to be studied.

4. **Analysis of output data:** Compute properties and report them.[2]

2.1.1 Discrete time integration

The MD method requires a discretization of the differential equations to find their solution in some specific points through time. This is applied to the computation of the new positions and velocities of particles from the old positions, old velocities, and corresponding forces.

Critical properties of integration methods in MD are efficiency, accuracy, and energy conservation. Accuracy specifies how much the numerically computed trajectory deviates from the exact trajectory after one time step. The error is usually given in powers of the time step δt. The energy is conserved along the trajectory of the particles for Hamiltonians $\mathcal{H}$ that do not explicitly depend on time. The numerical trajectory can deviate from the exact trajectory and thereby causes a small drift in the energy. Here, it is important to distinguish between errors caused by the finite accuracy of computer arithmetic and errors caused by the integration method itself even if infinitely accurate arithmetic is assumed. [17]

$\frac{1}{2}\sum_{i=1}^{N} m_i \left|\mathbf{v}_i(t)\right|^2$ depends on the velocity of all particles. The internal energy E is defined as $E \equiv E_{kin} + E_{pot}$ and should be conservative if the system is closed and isolated.

[2]The reports can be simple information on screen, storage in text files, graphs or even animations; i.e. anything that can be useful for an appropriate analysis of simulation results.

Closely connected with these issues is the question whether the integration method has the properties of time reversibility and symplecticity. First, time reversibility guarantees that, if the sign of the velocity is changed in the differential equation, the computed trajectory is followed exactly in inverse direction and the initial configuration is finally reached in the absence of numerical rounding errors.

On the other hand, an integration method can be interpreted as a mapping in phase space. If the integration method is applied to a measurable set of points in the phase space, this set is mapped to another measurable set in the phase space. The integration method is called symplectic if the measure of both of those sets is equal. Symplectic methods exhibit excellent behavior with respect to energy conservation. For 1-D systems, symplecticity is even equivalent to energy conservation. This is, however, not the case for higher-dimensional systems. Numerical approximations computed by symplectic methods can be viewed as exact solutions of slightly-perturbed Hamiltonian systems. The computed trajectory is valid if the difference between the hamiltonian $\mathcal{H}$ and the slightly-perturbed hamiltonian $\tilde{\mathcal{H}}$ is considerably small.

Most of the usual numerical methods, like the primitive Euler scheme and the classical Runge-Kutta scheme, are not symplectic integrators. The Störmer-Verlet method is a time-reversible and symplectic integration method that is used for MD.

The Störmer-Verlet integration method The standard form of the Störmer-Verlet method for the integration of Newton's equations is described by (2.2), where δt is the sample time (or time step), n is the current simulation step at time $t_n = n\,\delta t$, and $\mathbf{x}_i^n := \mathbf{x}_i\,(t_n)$[3].

$$\mathbf{x}_i^{n+1} = 2\mathbf{x}_i^n - \mathbf{x}_i^{n-1} + \frac{\mathbf{F}_i^n}{m_i}\,(\delta t)^2 \tag{2.2}$$

This method has two disadvantages. First of all, large rounding errors can be produced during the addition of the small $(\delta t)^2\,\mathbf{F}_i^n/m_i$ with the large terms $2\mathbf{x}_i^n$ and $\mathbf{x}_i^{n-1}$. Second, this method does not provide the velocity. Then, the velocity has to be approximated using $\mathbf{v}_i^n = \left(\mathbf{x}_i^n - \mathbf{x}_i^{n-1}\right)/2\delta t$. [17]

There are two variants of the Störmer-Verlet that allow to obtain the velocity: the leapfrog scheme, and the so-called Velocity-Störmer-Verlet method (a.k.a Velocity-Verlet). Both methods reduce the effect of rounding errors, but the leapfrog provides the position and velocity at different times. The standard form of the Velocity-Verlet method is described by (2.3) and (2.4).

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \mathbf{v}_i^n\delta t + \frac{1}{2}\frac{\mathbf{F}_i^n}{m_i}\,(\delta t)^2 \tag{2.3}$$

[3] Analogously for $\mathbf{v}_i$ and $\mathbf{F}_i$.

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \frac{1}{2}\frac{\mathbf{F}_i^n + \mathbf{F}_i^{n+1}}{m_i}\delta t \tag{2.4}$$

Implementations of the Velocity-Verlet can save memory resources by reusing the vector $\mathbf{F}$. This implies that the velocity has to be computed in two different steps. Let $\mathbf{v}_i^{n+1/2}$ be the intermediate computation of the velocity, then the velocity $\mathbf{v}_i^n$ is computed as:

$$\mathbf{v}_i^{n+1/2} = \mathbf{v}_i^n + \frac{1}{2}\frac{\mathbf{F}_i^n}{m_i}\delta t \tag{2.5}$$

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^{n+1/2} + \frac{1}{2}\frac{\mathbf{F}_i^{n+1}}{m_i}\delta t, \tag{2.6}$$

where $\mathbf{F}_i^{n+1}$ is computed in between by evaluating the force field with the particles already moved to the position $\mathbf{x}^n$. After velocities and positions have been updated, the algorithm can continue computing energies and other derivative quantities. Figure 2.1 shows the sequence and resources used by the modified Velocity-Verlet integration scheme.

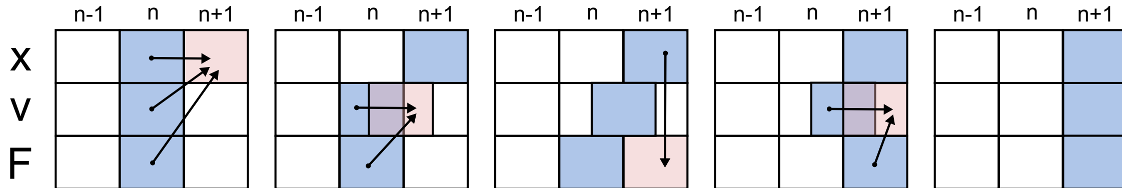

Figure 2.1: Evolution and resource usage of the modified Velocity Störmer-Verlet. [17]

2.1.2 Force field

MD simulations use a force field to compute the potential energy of a system of particles. The force field is a specific set of functions and parameters that have been obtained not only from experimental measurements but also from theoretical calculations in quantum mechanics. This approximation makes feasible the simulation of biological systems, since such systems involve a big set of atoms and would demand high computational resources to solve them by using principles of quantum mechanics. Among the most commonly used potential energy functions are the AMBER, CHARMM, GROMOS and OPLS/AMBER force fields.

The total potential energy in an additive force field is defined by the contribution of the bonded and non-bonded interactions. The bonded interactions are intermolecular interactions restricted to less than three bonds between particles. Bonded interactions have three terms: bond, angle and torsion. Some force fields may also include the hydrogen bonds. On the other hand, non-bonded interactions are long range interactions that include many interactions per atom, so it is more computational intensive. Simulations are commonly limited

to pairwise energies, since the computation of many-body energies are even more resource and time demanding. Examples of pairwise energies are the van der Waals interaction and the electrostatic interaction. Figure 2.2 shows some of the models that are used to compute the stretching (bonds), bending (angles), dihedral (torsion), and non-bonded terms of the potential energy.

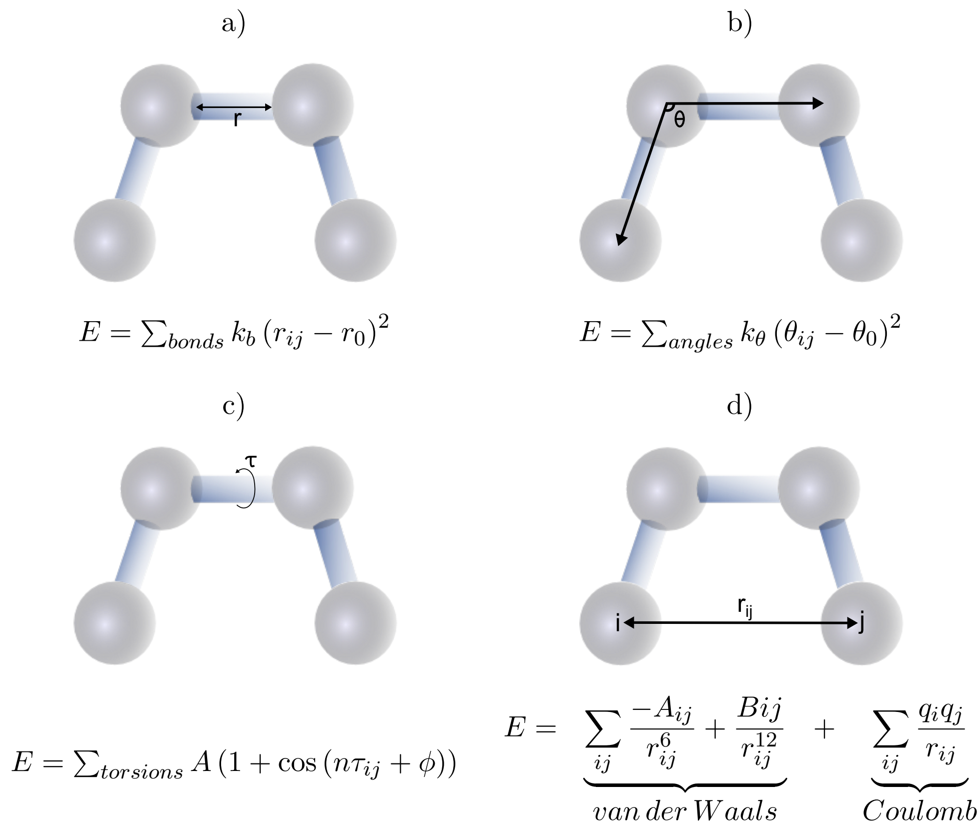

Figure 2.2: Potential energy models: a) Bonds, b) Angles, c) Torsions, d) Non-bonded.[29]

Equation 2.7 describes the total potential $V(\mathbf{x})$ due to only pairwise potentials $U_{ij}(r_{ij})$ in a set of N particles. In this case, the potential $U_{ij}(r_{ij})$ is a function depending on the distance r_{ij} between particle i and particle j. Repulsive forces between particles result from positive potential energies, while attraction results from negative potentials. The total force that acts over particle i is, by definition, the negative gradient of the potential energy with respect to its position $\mathbf{x}_i$, as described by (2.8).

$$V(\mathbf{x}) = V(\mathbf{x}_1, ..., \mathbf{x}_N) = \sum_{i=1}^{N} \sum_{j>i}^{N} U_{ij}(r_{ij}) \tag{2.7}$$

$$\mathbf{F}_i(\mathbf{x}) = -\nabla_{\mathbf{x_i}} V(\mathbf{x}) = \sum_{j \neq i}^{N} \left(\frac{\partial}{\partial r_{ij}} U_{ij}(r_{ij}) \mathbf{r}_{ij} \right) \tag{2.8}$$

The total force can be obtained from the contribution of single pairwise forces excluding interactions with the same particle. The pairwise force is described by (2.9).

$$\mathbf{F}_{ij} = \frac{\partial}{\partial r_{ij}} U_{ij}(r_{ij}) \mathbf{r}_{ij} \tag{2.9}$$

The distance vector between particle i and particle j is defined as $\mathbf{r}_{ij} := \|\mathbf{x}_i - \mathbf{x}_j\|$, and the squared magnitude of the distance $r_{ij}^2 = |\mathbf{r}_{ij}|^2$ can be computed using (2.10), where $\mathbf{x}_i[d]$ denotes the dth component of the position of particle i.

$$r_{ij}^2 = \sum_{d=1,2,3} (\mathbf{x}_i[d] - \mathbf{x}_j[d])^2 \tag{2.10}$$

$$r_{ij}^2 = (\mathbf{x}_i[1] - \mathbf{x}_j[1])^2 + (\mathbf{x}_i[2] - \mathbf{x}_j[2])^2 + (\mathbf{x}_i[3] - \mathbf{x}_j[3])^2 \tag{2.11}$$

2.1.3 Boundary conditions

Boundary conditions define the behavior of the system in the border of the simulation box. There are two main groups of boundary conditions:

- **Isolated Boundary Conditions (IBC)** are suitable for the study of clusters and molecules. The system with N particles is in vacuum, where particles only interact with each other, and it is assumed that they are completely isolated in the universe, but they only interact with an external force.

- **Periodic Boundary Conditions (PBC)** is normally used to study liquids and solids. The set of particles are grouped in a super cell that is surrounded by the images of the same super cell. It means that particles inside the super cell not only interact with others in the super cell but also with those in the neighbor cells. A particle leaving the super cell will appear in the opposite side of the simulation box.

The simulation domain is often assumed rectangular $\Omega = [0, \mathbf{L}[1]] \times [0, \mathbf{L}[2]] \times [0, \mathbf{L}[3]]$.

Minimum Image Convention in periodic systems In periodic systems, the use of replicated images leads to increase the complexity, because the new system is theoretically infinite, and the particles in the simulation box has to interact not only with those in its box, but also with all others in the images. To simplify the search of interacting pairs, the minimum

image convention assumes that interactions at a distance greater than half the simulation box are neglected.

Equation 2.12 describes the condition used to compute the components of the distance vector $\mathbf{r}_{ij}$. Figure 2.3 shows a set of particles moving in a 2-D periodic system with PBC, and a subset of particles that are selected by the minimum image convention. Note that the particles wrap in the simulation box when going outside.

$$\mathbf{r}_{ij}[d] = \begin{cases} \mathbf{x}_i[d] - \mathbf{x}_j[d] - L & if \ (\mathbf{x}_i[d] - \mathbf{x}_j[d]) > L/2 \\ \mathbf{x}_i[d] - \mathbf{x}_j[d] + L & if \ (\mathbf{x}_i[d] - \mathbf{x}_j[d]) < -L/2 \\ \mathbf{x}_i[d] - \mathbf{x}_j[d] & otherwise \end{cases} \tag{2.12}$$

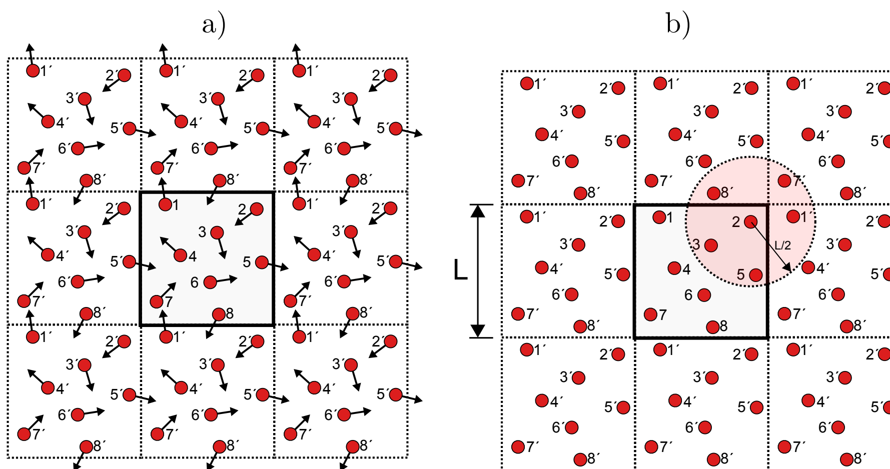

Figure 2.3: Particles in a 2-D periodic system. a) Moving and wrapping. b) The minimum image convention.

2.2 Non-bonded interactions

2.2.1 Van der Waals interactions

The van der Waals interaction describes the effects of polarization by defining the attractive and repulsive forces due to instantaneous induced dipoles between atoms, molecules and surfaces. These instantaneous dipoles are caused by a shortly redistribution of electrons in the electronic cloud of non-polar molecules. The redistribution is a consequence of quantum mechanics, which makes these interactions probabilistic. The short-range attraction and repulsion between non-polar molecules are not as strong as the chemical bonds, but play a fundamental role in fields as diverse as supramolecular chemistry, structural biology, polymer science, nanotechnology, surface science, and condensed matter physics. This phenomenon is

the only intermolecular force present between non-polar species such as helium, nitrogen, or methane as a few examples. Without the van der Waals force, there would be no attractive force between these molecules and they could not be obtained in liquid form.

The van der Waals interactions are mainly modeled by the Lennard-Jones potential, but there also exists the Exponential-6 potential. Figure 2.4 shows the plot of these potentials with normalized parameters.

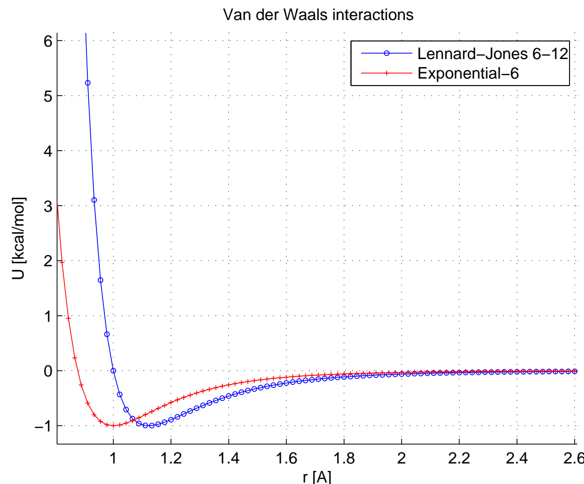

Figure 2.4: Van der Waals interactions: Lennard-Jones 6-12 and Exponential-6 potentials with normalized parameters ($\epsilon_{ij} = 1$ and $\sigma_{ij} = 1$).

2.2.1.1 Lennard-Jones 6-12 potential

The Lennard-Jones (LJ) potential energy is used to model van der Waals interactions. The 6-12 configuration is mostly used for intermolecular interactions, whereas others like the LJ 1-4 is mostly used for intramolecular interactions. The potential and force functions are described by (2.13) and (2.14), respectively.

$$U_{ij}^{LJ} = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] \tag{2.13}$$

$$\mathbf{F}_{ij}^{LJ} = 24\epsilon_{ij}\left[2\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right]\frac{1}{r_{ij}^2}\mathbf{r}_{ij} \tag{2.14}$$

The parameter ϵ_{ij} is the minimum of the potential or the depth of the potential well, and σ_{ij} is the zero-crossing of the potential function. Both parameters are determined by the type of each particle and can be calculated from the interatomic parameters ϵ_{ii} and σ_{ii} applying the Lorentz-Berthelot mixing rule, which states that $\sigma_{ij} = \frac{1}{2}(\sigma_{ii} + \sigma_{jj})$ and $\epsilon_{ij} = \sqrt{\epsilon_{ii}\epsilon_{jj}}$.

The Lennard-Jones potential has a strongly attractive term r^{-12} that decays quickly at short distances, and a smoother repulsive term with less magnitude $-r^{-6}$ that decays slower and dominates the middle and long range.

2.2.1.2 Exponential-6 potential

The modified-Buckingham or Exponential-6 (Exp-6) potential energy function has the advantage over Lennard-Jones and others like the Kihara potential, in that the exponential repulsive potential seems more realistic than a simple power law [10]. However, it is not usually implemented to describe van der Waals interactions because of its higher computational cost. Besides the parameters ϵ_{ij} and σ_{ij}, the Exp-6 function also has an extra parameter ς, which is the repulsive-wall steepness parameter. For $\varsigma = 12$, the Exp-6 potential approximates the LJ 6-12 potential at long distances. The Exp-6 potential is described by (2.15).

$$U_{ij}^{Exp6} = \epsilon_{ij}\left[\frac{6}{\varsigma - 6}\exp\left(\varsigma\left(1 - \frac{r_{ij}}{\sigma_{ij}}\right)\right) - \frac{\varsigma}{\varsigma - 6}\left(\frac{\sigma_{ij}}{r_{ij}}\right)^6\right] \tag{2.15}$$

2.2.2 Electrostatic interactions

Polar molecules are permanent dipoles with positive and negative charges on their ends, which produce a strong intermolecular interaction called electrostatic interaction. The charged particles (ions) are modeled in MD as punctual charges centered in a spherical rigid body. The repulsion or attraction is determined if the sign of the charges is the same or opposite, respectively. This phenomenon is mathematically represented by the Coulomb potential in (2.16) and the Coulomb force in (2.17), where $k_e = {}^1\!/_{4\pi\epsilon_o} = 8.987551 \cdot 10^9\, N \cdot m^2/C^2$ is the Coulomb constant and $\epsilon_o \approx 8.854187 \cdot 10^{-12}\, F \cdot m^{-1}$ is the vacuum permittivity.

$$U_{ij}^{C} = k_e \frac{q_i q_j}{r_{ij}} \tag{2.16}$$

$$\mathbf{F}_{ij}^{C} = \frac{1}{4\pi\epsilon_o}\frac{q_i q_j}{r_{ij}^3}\mathbf{r}_{ij} \tag{2.17}$$

In contrast to the Lennard-Jones 6-12 potential that decays at a rate of $-r^{-6}$, the r^{-1} term indicates that the electrostatic potential is a long-range intermolecular potential. This potential has a computational complexity of $O\left(N^2\right)$ for isolated systems, but is much higher for periodic systems because of its slow convergence. The complexity of long-range potentials can be reduced using mesh-based methods, tree methods, or the classical Ewald summation.

2.2.2.1 Ewald summation for electrostatic potential

The charge distributions are described by delta functions in the electric field, but the Ewald summation method adds and subtracts diffuse charge distributions in the form of Gaussians around those punctual charges in order to decompose the Coulomb potential into summations that converge not only rapidly but also absolutely [25]. The general idea of Ewald summation is: sum screened particle interactions in direct space for the short-range part, and subtract compensating potential in reciprocal space for the long-range part. This method only applies to periodic systems that are electrically neutral. Equation 2.18 describes the three terms involved in the Ewald summation

$$V^{Ew} = V^o + V^d + V^r, \tag{2.18}$$

where V^o is the self potential, V^d is the direct-space potential, and V^r the reciprocal-space potential.

Self potential The self potential V^o described by (2.19) is computed only once during the simulation if and only if the number of particles, the charge of each particle, and the parameter α remains constant. If V^o remains constant, then the force $\mathbf{F}^o_{ij}$ is null.

$$V^o = \sum_{j=1}^{N} \frac{-\alpha}{\sqrt{\pi}} q_j^2 \tag{2.19}$$

The parameter $\alpha \equiv 1/\sqrt{2}\sigma$, where σ is the standard deviation of the Gaussian distribution, can be chosen to reduce the computational complexity to $O\left(N^{1.5}\right)$ by balancing the load between the short-range and long-range parts.

Direct-space potential The direct-space potential presented in (2.20) represents the short-range part of the Coulomb potential, and converges quickly in the direct space. The pair force is given by (2.21).

$$V^d = \sum_{i=1}^{N} \sum_{j>i}^{N} U^d_{ij}\left(r_{ij}\right) = \sum_{i=1}^{N} \sum_{j>i}^{N} \frac{q_i q_j}{r_{ij}} erfc\left(\alpha r_{ij}\right) \tag{2.20}$$

$$\mathbf{F}^d_{ij} = \frac{q_i q_j}{r_{ij}^3} \left(\frac{2\alpha}{\sqrt{\pi}} r_{ij} \exp\left(-\alpha^2 r_{ij}^2\right) + erfc\left(\alpha r_{ij}\right) \right) \mathbf{r}_{ij} \tag{2.21}$$

The function $erfc\left(x\right)$ is the complementary error function defined as $erfc\left(x\right) := 1 - erf\left(x\right)$, and $erf\left(x\right) = \frac{2}{\sqrt{\pi}} \int e^{-u^2} du$ is the error function.

Reciprocal-space potential The reciprocal-space potential is the long-range part of the Coulomb potential. The long-range potential and force converge quickly in the reciprocal space and are described by (2.22) and (2.23), respectively.

$$V^r = \frac{1}{2\pi Vol} \sum_{m \neq 0} \frac{\exp\left(-\pi^2 m^2 \alpha^{-2}\right)}{m^2} \left|S_j\left(m\right)\right|^2 \tag{2.22}$$

$$\mathbf{F}_{ij}^r = \left(\frac{1}{2\pi Vol} \sum_{m \neq 0} \frac{\exp\left(-\pi^2 m^2 / \alpha^2\right)}{m^2} \right) \mathbf{m} \tag{2.23}$$

The variable Vol is the volume of the simulation box, the vector $\mathbf{m}$ is the reciprocal-space vector, and the vectors $S_j\left(\mathbf{m}\right)$ are defined as

$$S_j\left(\mathbf{m}\right) = \sum_{j=1}^{N} q_j \exp\left(2\pi i\, \mathbf{m} \cdot \mathbf{r}_j\right) = \sum_{j=1}^{N} q_j \exp\left(2\pi i \left(\mathbf{m}\left[1\right] \mathbf{r}_j\left[1\right] + \mathbf{m}\left[2\right] \mathbf{r}_j\left[2\right] + \mathbf{m}\left[3\right] \mathbf{r}_j\left[3\right]\right)\right).$$

The Particle Mesh Ewald (PME) is a grid method that reduces the complexity of the Ewald summation to $O\left(N \log\left(N\right)\right)$ using a 3-D Fast Fourier Transform (FFT) over a regular grid in the reciprocal space. The charge of each particle is interpolated to its surrounding grid points using splines to build the potential surface. The resulting force over each point is interpolated to each particle. The Smooth Particle Mesh Ewald (SPME) method is a variant of the PME that uses B-splines for the interpolation. The PME method is more efficient for systems with *"smooth"* variations in density, or continuous potential functions. Localized systems or those with large fluctuations in density may be treated more efficiently with the fast multipole method of Greengard and Rokhlin. [17]

2.2.3 Truncation of the potential energy function

The cutoff radius r_{cut} is a very simple method to reduce computational complexity by skipping particles located outside the cutoff sphere centered at the current particle, i.e. only compute forces for those particles with a separation distance less than the cutoff radius. The computational complexity is $O\left(N^2\right)$ for isolated systems, but it tends to infinite for periodic systems. This method results especially useful for the short-range pair interactions like the Lennard-Jones potential and the direct-space Ewald summation because they converge quickly in direct space. The cutoff radius is the base of other successful methods like the Linked Cell method for spacial decomposition.

Equation 2.24 describes the redefinition of the pairwise Lennard-Jones 6-12 potential, which has been truncated using the cutoff radius. Likewise, the total Lennard-Jones potential

energy and force are redefined by (2.25) and (2.26), respectively.

$$U(r_{ij}) \approx \begin{cases} 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] & if\, 0 < r_{ij} \leq r_{cut} \\ 0 & otherwise \end{cases} \tag{2.24}$$

$$V^{LJ}(\mathbf{x}_1, ..., \mathbf{x}_N) \approx \sum_{i=1}^{N} \sum_{\substack{j = i+1 \\ r_{ij} < r_{cut}}}^{N} 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] \tag{2.25}$$

$$\mathbf{F}_i^{LJ} \approx \sum_{\substack{j = 1,\, j \neq i \\ r_{ij} < r_{cut}}}^{N} 24\epsilon_{ij}\left[2\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right]\frac{1}{r_{ij}^2}\mathbf{r}_{ij}. \tag{2.26}$$

The value of parameters like σ and α determine how quickly these functions converge. A simulation using only the Lennard-Jones potential can define $r_{cut} \geqslant 2.5\sigma_{max}$,[4] but this could not be appropriate for other potentials. However, it is still valid for the direct-space Ewald summation if the value ofα makes it converge more quickly than the Lennard-Jones potential.

The truncation of the potential function has the disadvantage that it introduces noise into the system. To reduce this undesirable effect, a shift[5] function $S(r_{ij})$ can be defined to softly decrease the function from some specific distance until reaching zero at distance r_{cut}.

2.3 Software packages for Molecular Dynamics simulations

There are several software packages in the market for molecular simulations using the MD method. Most of them are written in high-level programming languages, are optimized to perform parallelization in clusters and supercomputers, and provide interfaces for quantum-mechanics packages and for visualization tools. Free software for visualization of the resulting dynamics can be downloaded for free from the Internet. Software such as VMD (Visual Molecular Dynamics [23]), which is fully compatible with NAMD. Other software for visualization and modeling can be found in the Linux repository. Here there is a short list and description of some of the most used software:

NAMD The *Not (just) Another Molecular Dynamics* program was developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign. It can be

[4] Assuming that values below $U_{ij}^{LJ}(2.5\sigma_{ij}) \approx -\epsilon/6$ can be neglected

[5] The shift function is the addition to the original function. A switch function multiplies the original function, is a special case of the shift function. [39]

integrated with VMD [23]. It uses a combination of spatial decomposition and force decomposition techniques to generate a high degree of parallelism. (See [34])

GROMACS The *Groningen Machine for Chemical Simulation* is written in ANSI C and can be easily recompiled to work with single or double precision. (See [40])

LAMMPS The *Large-scale Atomic/Molecular Massively Parallel Simulator* is a classical MD code and uses spatial decomposition techniques by partitioning the simulation domain into small 3-D subdomains.

PROTOMOL is a high-performance object-oriented framework written in C++ for the rapid prototyping of novel algorithms for MD and related algorithms. (See [27])

MMTK The Molecular Modeling Toolkit is a Python library that implements common molecular simulation techniques, with an emphasis on biomolecular simulations. (See [22])

3. Hardware Design of the FPGA coprocessor

This chapter describes the methodology followed to design the FPGA coprocessor that computes the Lennard-Jones 6-12 and direct-space Ewald forces and potentials.

3.1 Design methodology

FPGAs have become a very powerful platform for rapid prototyping due mainly to their reprogrammability. They also have increased considerably the speed and the density of resources with the time. Current FPGAs dispose of embedded, distributed, resizable, true dual-port memories; embedded multipliers and DSP blocks with rounding and saturation blocks; high-speed links for Rapid-IO, Ethernet and PCI Express; PLLs, and so on. They also result less power consuming than clusters and supercomputers.

Furthermore, new programming models are presented for FPGAs in [21] to implement massive parallelism and distributed processing. For this and more reasons, FPGAs are suitable to face many of the problems in the High Performance Computing that used to be dominated by PC-clusters, multi-core ASIC microprocessors, GPUs, mainframes, custom ASICs, and so on. For instance, recent reports about suitable floating-point computation in FPGAs ([31, 30, 32, 6]) demonstrate that they can achieve Tera FLOPS.

According to the general methodology for designs with FPGAs [5], the design starts with an RTL (Register-Transfer Level) description for a functional solution of the problem. The hardware design is then simulated to verify its correct functionality. Next, it is synthesized and fitted in an FPGA to know how many resources it demands. Timing analysis is also performed to identify bottlenecks and to analyze performance. Finally, the whole design is validated in the application. These steps are executed in such a way that design meets every time better the requirements of the application.

The RTL descriptions are written in VHDL and Verilog, and are simulated using ModelSim-Altera. Matlab generates stimuli files that are read by the testbench in ModelSim to stimulate the design. Matlab also reads simulation results, carries out comparisons with mathematical models, and generates text and graphical reports. Altera's Quartus II software is used for synthesis, fitting and timing analysis. Validation of the application uses MD simulations that

were run in a C-application also developed during this work. This application uses a simple force field that only computes the Lennard-Jones 6-12 potential for the van der Waals interactions, and the Ewald summation for the Coulomb interactions, i.e. it does not compute bonded interactions. Figure 3.1 shows how tasks are divided between software (SW) and hardware (HW).

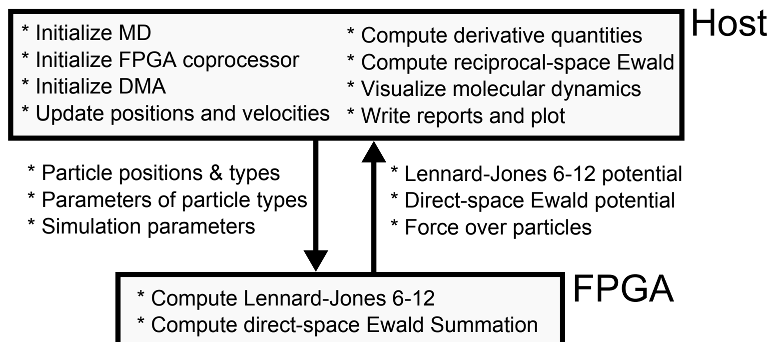

Figure 3.1: HW/SW task division of the MD application.

After a study of the problem and of previous hardware implementations, the design started with the following requirements:

- Computation of van der Waals interactions using the Lennard-Jones 6-12 force and potential.
- Computation of the short-range part of electrostatic interactions using the direct-space Ewald summation.
- Use of the basic $O\left(N^2\right)$ algorithm.
- 3-D simulation box with maximum size of 256 Å, where each coordinate can be in range ±128 Å.
- Capacity to store and process more than 1000 particles and more than 10 elements.
- Periodic refolding using the minimum image convention.
- Store separately each potential V^{LJ} and V^{d}.
- The system should be connected to a computer through a high-speed IO link (e.g. USB, PCI, PCI Express) to reduce communication overhead.
- Accuracy comparable with software simulations using single-precision data.
- Operating frequency greater than 100 MHz.

- Acceleration of MD simulations is desired, but optional.

Following sections present the bottom-up design flow starting from the functional building units and finishing with a system level description. The first idea is to design an IP core called *LJEwDir core* that computes the short-range part of non-bonded interactions in direct space. This core consists of a datapath, a controller and interfaces for communication with master devices. The top level description is the final implementation of the design on an FPGA that, in this case, integrates an Avalon memory-mapped system that instantiates the LJEwDir core. Before the explicit hardware design, some mathematical treatments are presented in order to obtain an optimal implementation of the potential energy functions.

3.2 Variable transformation of the potential and force functions

Direct evaluation of the potential and force functions in hardware is found very difficult because of their complexity. Moreover, each operation in an FPGA represents area, and the problem grows when some functions like the power function and trigonometric functions use recurrence in software to bring results. For this reason, the functions should be reorganized in something more convenient that lets use polynomial interpolation to simplify the hardware implementation while still keeping accuracy.

This work purposes a novel method to simplify the computation in hardware of the short-range part of non-bonded interactions. The method consists in applying variable transformations to reduce the number of variables in the function, in such a way that a family of curves depending of two variables is reduced to a single curve.

Sections 3.2.1 and 3.2.2 present the transformations applied to the potential function $U(r)$ and the force function $F(r)$ (scalar part of $\mathbf{F}(\mathbf{r})$). In general, the transforming variable w is computed operating the squared distance between pairs of particles with some other parameter. It is followed by the definition of the functions $u(w)$ and $f(w)$ that are part of the functions $U(r)$ and $F(r)$, respectively. The new $U(w)$ and $F(w)$ are left as simple multiplications of $u(w)$ and $f(w)$ with other parameters.

This method brings several advantages:

- The remaining expression for $U(w)$ and $F(w)$ requires only multiplications between some constants and the interpolated functions $u(w)$ and $f(w)$. It is especially convenient for floating-point arithmetic, since the multiplication is not so expensive as the addition/subtraction is, only if the FPGA disposes of available embedded multipliers or DSP blocks.

- It uses only one interpolant per function, which is different to other reported methods that interpolate several powers of r^{-1} for further use in the force or the potential functions.
- It uses a simple multiplication with r^2, and so avoiding inverses, squared roots and other complex functions.
- The main advantage comes with the fact that the evaluation $u(w)$ and $f(w)$ does not depend anymore on r^2. It lets trace a curve that can be bounded in the relevant interval, which can be split in several subintervals to optimize interpolations.

3.2.1 Transformation of Lennard-Jones 6-12

The LJ functions are transformed using $w_{ij}^{LJ} = \sigma_{ij}^{-2} r_{ij}^2$, leading to the definition of the functions $u_{ij}^{LJ}\left(w_{ij}^{LJ}\right)$ and $f_{ij}^{LJ}\left(w_{ij}^{LJ}\right)$ as described by (3.1) and (3.2), respectively.

$$u_{ij}^{LJ}\left(w_{ij}^{LJ}\right) = 4\left[\left(w_{ij}^{LJ}\right)^{-6} - \left(w_{ij}^{LJ}\right)^{-3}\right] \tag{3.1}$$

$$f_{ij}^{LJ}\left(w_{ij}^{LJ}\right) = 24\left[2\left(w_{ij}^{LJ}\right)^{-7} - \left(w_{ij}^{LJ}\right)^{-4}\right] \tag{3.2}$$

Replacing (3.1) and (3.2) within the potential and force function results in (3.3) and (3.4), respectively.

$$U_{ij}^{LJ}\left(r_{ij}\right) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] \to U_{ij}^{LJ}\left(w_{ij}^{LJ}\right) = \epsilon_{ij}\, u_{ij}^{LJ}\left(w_{ij}^{LJ}\right) \tag{3.3}$$

$$F_{ij}^{LJ}\left(r_{ij}\right) = 24\epsilon_{ij}\left[2\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right]\frac{1}{r_{ij}^2} \to F_{ij}^{LJ}\left(w_{ij}^{LJ}\right) = \epsilon_{ij}\sigma_{ij}^{-2}\, f_{ij}^{LJ}\left(w_{ij}^{LJ}\right) \tag{3.4}$$

Hence, the system will only require the values ϵ_{ij} and σ_{ij}^{-2} to compute the functions performing multiplications with r_{ij}^2 and with the interpolated functions u_{ij}^{LJ} and f_{ij}^{LJ}. These parameters are particle type dependent and can be easily computed by the host at the beginning of the simulation, and then loaded to system's memory.

3.2.2 Transformation of the direct-space Ewald summation

Similarly to the transformation of the Lennard-Jones 6-12 potential, the direct-space Ewald functions are transformed using $w_{ij}^{d} = \alpha^2 r_{ij}^2$. Equations 3.5 and 3.6 describe the functions $u_{ij}^{d}\left(w_{ij}^{d}\right)$ and $f_{ij}^{d}\left(w_{ij}^{d}\right)$.

$$u_{ij}^{d}\left(w_{ij}^{d}\right)=\frac{erfc\left(\sqrt{w_{ij}^{d}}\right)}{\sqrt{w_{ij}^{d}}} \tag{3.5}$$

$$f_{ij}^{d}\left(w_{ij}^{d}\right)=\left(\frac{2}{\sqrt{\pi}}\sqrt{w_{ij}^{d}}\exp\left(-w_{ij}^{d}\right)+erfc\left(\sqrt{w_{ij}^{d}}\right)\right)\left(w_{ij}^{d}\right)^{-3/2} \tag{3.6}$$

Replacing (3.5) and (3.6) within the potential and force function results in (3.7) and (3.8), respectively.

$$U_{ij}^{d}\left(r_{ij}\right)=\frac{q_iq_j}{r_{ij}}erfc\left(\alpha r_{ij}\right)\rightarrow U_{ij}^{d}\left(w_{ij}^{d}\right)=q_iq_j\alpha\, u_{ij}^{d}\left(w_{ij}^{d}\right) \tag{3.7}$$

$$F_{ij}^{d}\left(r_{ij}\right)=\frac{q_iq_j}{r_{ij}^{3}}\left(\frac{2\alpha}{\sqrt{\pi}}r_{ij}e^{-\alpha^2 r_{ij}^2}+erfc\left(\alpha r_{ij}\right)\right)\rightarrow F_{ij}^{d}\left(w_{ij}^{d}\right)=q_iq_j\alpha^3\, f_{ij}^{d}\left(w_{ij}^{d}\right) \tag{3.8}$$

In this case, the system will only require the value of $q_iq_j\alpha$ and α^2. In some simulations, the value of α may change, so it will require that the host recomputes the values of $q_iq_j\alpha$ and loads them again into system's memory.

3.3 Polynomial interpolation using pseudo-floating-point representation

The computation of some functions such as $erfc\left(x\right)$, x^{-12}, or $exp\left(-x^2\right)/x$ is very expensive in hardware. In order to save hardware and increase performance and accuracy, it is purposed the use of polynomial interpolations using pseudo-floating-point representation[1]. This representation is used to perform fixed-point arithmetic with normalized variables that keep their values in a tight range while keeping apart a reference exponent that denormalizes the variables for the respective range.

The idea behind this method is to fit intervals of a function $y\left(x\right)$ by using polynomials $p^n\left(x\right)$, where n is the order. The polynomial $p^n\left(x\right)$ has $n+1$ coefficients c_k with $k\in[0,\, n]$. Evaluating the polynomial using the factorization presented for the third-order polynomial in (3.9) reduces rounding errors that can be produced due to high-order powers.

$$p\left(x\right)=\left(\left(c_3x+c_2\right)x+c_1\right)x+c_0 \tag{3.9}$$

The following methodology was used to fit the target functions $u\left(w\right)$ and $f\left(w\right)$ for the LJ and the direct-space Ewald summation:

[1] Other interpolation methods for these energy functions are found in [15, 18].

1. Evaluate the target function $y(x)$.

2. Define logarithmic intervals[2] in powers of two[3], e.g. setting the logarithmic break points at $[2^{-3}, 2^{-2}, 2^{-1}, 2^{0}, 2^{1}]$. If the range is defined as $[2^{expmin}, 2^{expmax})$ then the total number of logarithmic intervals P_{log} is given by $P_{log} = expmax - expmin$.

3. Define linear intervals inside each logarithmic interval. The number of inner linear intervals P_{lin} should be a power of two to simplify the search in hardware; thus, $P_{lin} = 2^{l_{lin}}$ corresponds to the number of linear intervals using l_{lin} bits. The total number of partitions P is then given by

$$P = P_{log} * P_{lin} = (expmax - expmin) * 2^{l_{lin}}.$$

4. Select a set of training points $y_t(x_t)$ per partition, whicht will be used by the algorithm for polynomial fitting.

5. Normalize the target function at the training points in such a way that the normalized values $\overline{y_t}(x_t)$ span the range $[-2, 2)$. This requires that all points in the range are divided by the next power of two 2^{expy} with respect to the maximum absolute value of $y(x)$ in the current partition. The exponent $expy$ of the normalizing power of two is stored separately for further denormalization.

6. To improve numerical properties of both the polynomial and the fitting algorithm, the input points x_t are transformed into $\bar{x}_t$ by centering and scaling from the range $[2^a, 2^{a+1})$ in the current partition to the range $[0, 2)$ as described by (3.10). This process produces well-conditioned polynomials that require less order than the ill-conditioned, and are more accurate.

$$\overline{x_t} = (x_t - 2^a) * 2^{1-a},\ x_t \in \left[2^a, 2^{a+1}\right) \tag{3.10}$$

7. Select the order for all polynomials, and perform polynomial fitting of $\overline{y_t}(\overline{x_t})$ in all partitions.

8. Finally, implement the fixed-point model and evaluate precision. The fixed-point model is built converting the coefficients c_k of all polynomials and the transformed input points $\bar{x}_t$ to a signed fixed-point representation. The conversion requires a rounding method and a word length corresponding to the hardware capabilities. The model should operate like the target hardware does.

[2] Logarithmic intervals are used here because all target functions decay exponentially. Other functions such are $\sin(x)$ can be interpolated using only linear intervals.

[3] The powers of two are friendly with binary arithmetic.

Figure 3.2 shows an example of how a function evaluated in the range $[2^{-3}, 2^{1})$ is divided into logarithmic partitions, and likewise each logarithmic partition is divided into four inner linear intervals.

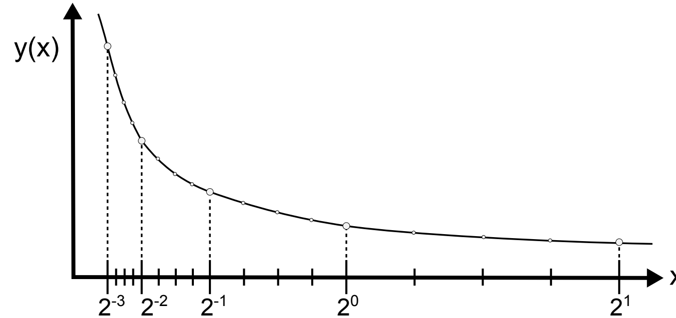

Figure 3.2: Example of logarithmic and inner linear partitions in range $[2^{-3}, 2^{1})$.

The above steps where performed in Matlab using the function *polyfit* for polynomial fitting, which uses an algorithm based on least squares. Matlab also has an embedded toolbox for fixed-point modeling called Fixed Point Toolbox. This toolbox has all the necessary functions and data types to play with the fixed-point representation, and even with the floating-point representation. Here some features of the toolbox:

- Data types support up to 65536 bits.
- Rounding and overflow methods. For rounding: ceil, floor, fix, round, nearest and convergent.
- Binary-point and bias-scale representations.
- Built-in single- and double-precision data types.
- Conversion between binary, hex, double and built-in integer representations.
- Relational, logical and bit-wise operators.
- Logging of minimum, maximums, overflows and underflows.
- Matrix and statistical functions. Many other Matlab functions support the fixed-point data type (e.g. plot).
- Compatible with the Matlab's Simulink tool.

In order to avoid overdimensioning of the interpolating system while still keeping accuracy, a parametric analysis was performed evaluating accuracy for all the $u(w)$ and $f(w)$ target functions with respect to the interpolation order, word length, number of linear partitions and

rounding method. Each parameter represents some cost in hardware: *"the main cost for finer intervals is in block RAMs, while the main cost for higher order interpolation is in hardware multipliers and registers"*. [20]

The simplest rounding method in hardware is truncation, which is equivalent to the method *floor* in Matlab. Other methods require extra hardware. It is very important to mention that the FPGA used in this work contains 18-bit DSP blocks that implement the rounding-to-nearest (equivalent to *nearest*) and rounding-to-even (equivalent to *convergent*) methods for words of maximum 18 bits.

Figure 3.3 shows results of one of these parametric analysis using 1000 training points per partition to interpolate $f^{LJ}\left(w^{LJ}\right)$ using the *convergent* rounding method and 8 linear partitions. Here, the mean and variance of the relative error are plotted with respect to the polynomial order and the word length. The acceptable region corresponds to all combinations with mean below 10^{-4} and variance below 10^{-5}. These values were chosen arbitrarily considering error propagation.

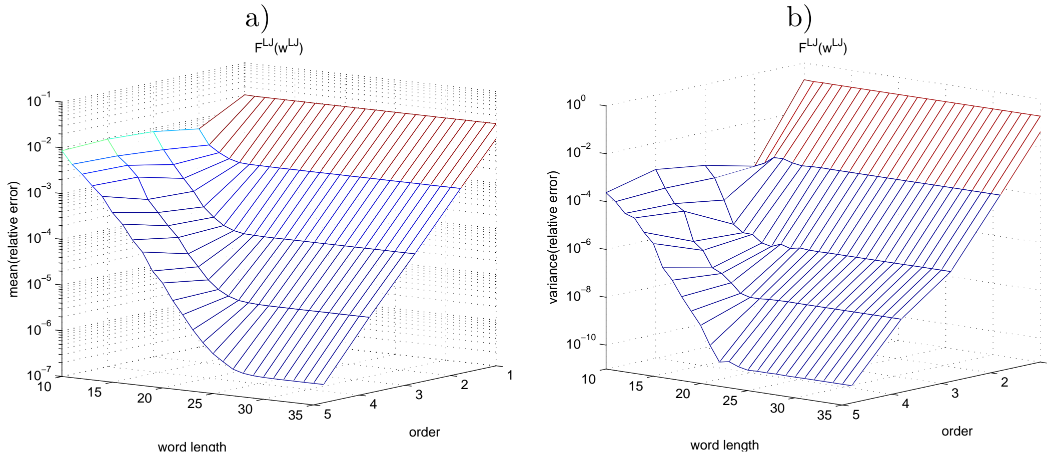

Figure 3.3: Relative error for the interpolation of $f^{LJ}\left(w^{LJ}\right)$ with respect to the interpolation order and the word length for 8 linear partitions and convergent rounding. a) Mean, b) Variance.

Note that increasing the word length after certain point does not improve precision for a given polynomial order. Note also that increasing the polynomial order tends to improve precision exponentially, but it is not totally valid for short words. During the polynomial evaluation, all partial values stayed in the range $[-4, 4)$, which means that a 3-bit integer part is enough to represent such partial variables.

After this exhaustive task, the parameters presented in table 3.1 were chosen to continue with the design. The selected range was $\left[2^{-7}, 2^{4}\right)$, i.e. $expmin = -7$, $expmax = 4$ and 11 logarithmic partitions. The lower limit of this interval satisfies simulations with e.g. $\sigma_{max} = 2.5$ and $r_{min} = 0.25$. The upper limit is chosen where all functions tend to be less than the

error bound $\varepsilon = 10^{-3}$. For a lower error bound, *expmax* has to be increased.

Table 3.1: Final interpolation parameters.

Parameter	Value	Parameter	Value
Word length	18	Order	3
Fractional length	15	Range	$2^{[-7,4)}$
Exponent length	7	Logarithmic partitions	11
Signedness	Signed	Inner linear partitions	8 (3 bits)
Rounding	Convergent	Total number of partitions	88

Figure 3.4 shows the denormalizing exponents and the coefficients of each of the 88 polynomials used to interpolate the function $f^{LJ}\left(w^{LJ}\right)$. Note that the value of all coefficients are constraint to $[-2, 2)$, and that the coefficient c_0 is mostly the biggest. Figure 3.5 shows how looks the target function after normalization, as well as how those values are moved to the range of the normalized input $\overline{x_t}$. An histogram of errors shows that most errors are less than 10^{-4}.

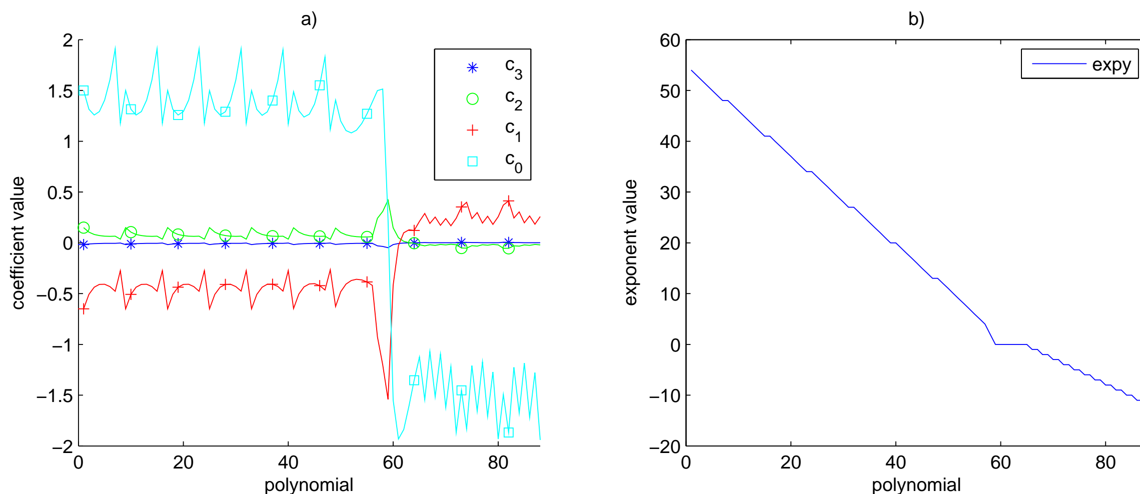

Figure 3.4: Polynomial coefficients and exponents for the interpolation of $f^{LJ}\left(w^{LJ}\right)$. a) Coefficients, b) Exponents.

3.4 Datapath of the LJEwDir core

This section presents the datapath of the LJEwDir core and its functional building units. The datapath shown in figure 3.6 is basically a deep functional pipeline with distributed memory that computes in parallel forces and potentials of Lennard-Jones 6-12 and direct-space Ewald summation taking into account the variable transformation method presented in section 3.2, and implementing the interpolation method seen in section 3.3.

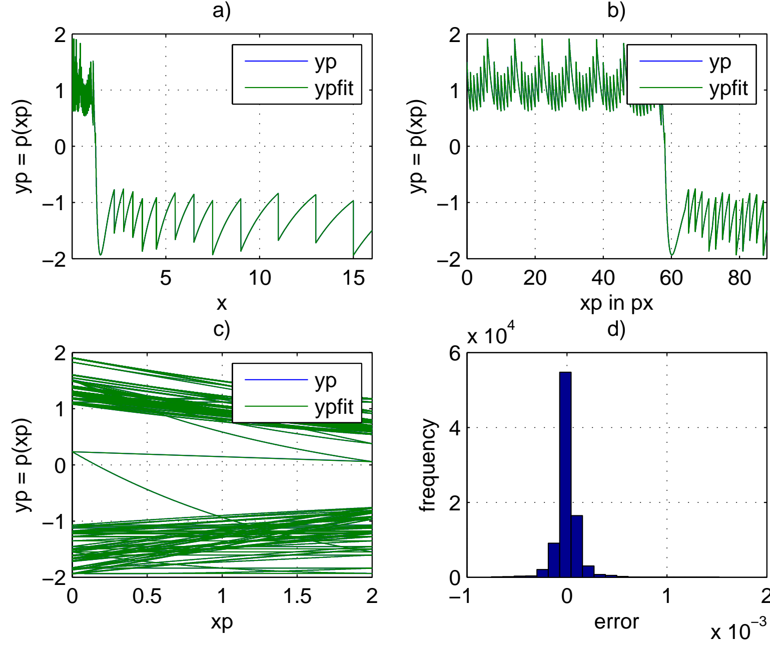

Figure 3.5: Interpolation of the normalized function $f^{LJ}\left(w^{LJ}\right)$. a) Normalized output, b) Normalized output for each polynomial, c) Normalized output w.r.t. the normalized input, d) Histogram of errors.

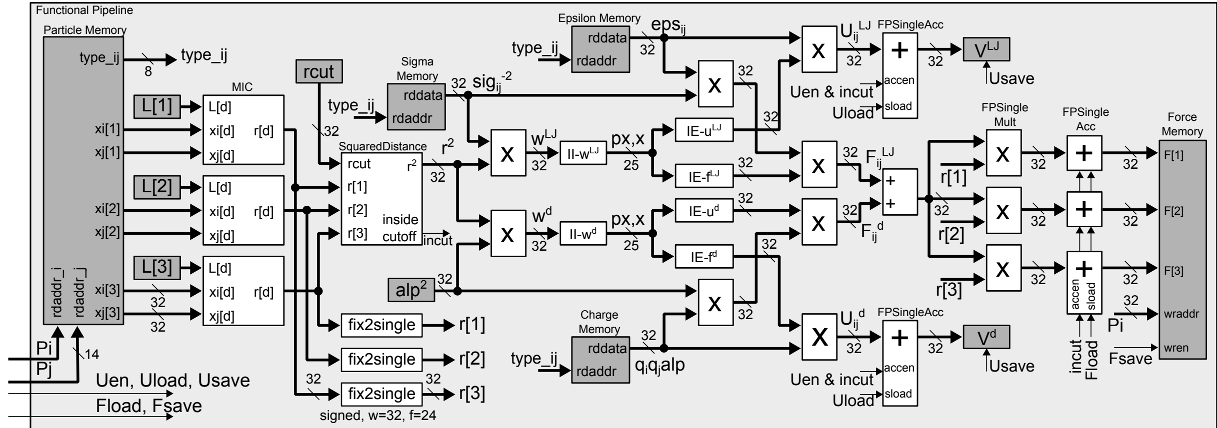

Figure 3.6: Overview of the functional pipeline.

One of the main advantages that pipeline architectures bring is their very high throughput. On the other hand, a little disadvantage is its latency[4]. The efficiency of the pipeline is given by (3.11). The more the number of cycles of computation, the higher the efficiency.

[4]However, the latency has in this case a very low impact in the efficiency because this system is made to compute many more cycles than the fixed latency. For instance, a pipeline with 100 cycles of latency that needs to perform 10000 computations has an efficiency of 99.0099%.

$$E = \frac{\#computations}{latency + \#computations} * 100\% \tag{3.11}$$

The datapath is divided into three stages:

1. The first stage computes the squared distance after applying the minimum image convention to the coordinates of the current pair of particles. The Particle Memory stores the position and the type of all particles in the simulation.

2. The interpolation engines are located in the heart of the datapath. Around the interpolation engines, several floating-point multipliers and Particle Memories are connected to perform the variable transformations and to complete the computation of the pair potentials and forces from the interpolated functions.

3. The last stage is the accumulation of the pair potentials and the accumulation of the resulting force over each particle. This stage has a floating-point adder to reduce the scalar forces F_{ij}^{LJ} and F_{ij}^{d}, and then multiplies the result by the components of $\mathbf{r}_{ij}$ to get the components of $\mathbf{F}_{ij}$. The resulting force $\mathbf{F}_i$ is finally stored in the Force Memory, while the potentials are stored in their respective registers.

The datapath uses hybrid arithmetic, i.e. it uses not only floating-point arithmetic, but also different fixed-point representations. From now on, when talking about floating-point data representation it refers to the 32-bit single-precision representation in the IEEE-754 standard, which is compatible with most computers.

The datapath has two different clock domains: an internal clock for the Functional Pipeline, and an external clock to read/write from/to the dual-port memories and registers for variables and parameters. This technique pretends to improve timing by reducing the load on the internal clock, and so the pipeline achieves higher operating frequencies.

3.4.1 Interpolation engines

The interpolation engines are the hardware responsible for the interpolation of the potential and force functions. They are implemented considering the values presented in table 3.1. The Interpolation Engine (IE) is a pipelined arrangement of Multiplier-Adders (MAD) to evaluate the polynomial $p(x)$ according to the coefficients c_k of each polynomial. These coefficients are stored in separate ROMs called CkROMs, while the denormalizing exponent *expy* is stored in the ExpROM. At the end of the pipeline, the Interpolation Engine converts from fixed point to floating point taking the value of $p(x)$ and *expy*. This fixed-point processing with a separate exponent is denominated in this work as pseudo-floating-point representation. This representation avoids the use of shifting operations to align data, while still keeping

high accuracy. Figure 3.7 shows the block diagram of the third-order Interpolation Engine connected to the Interpolation Input decoder.

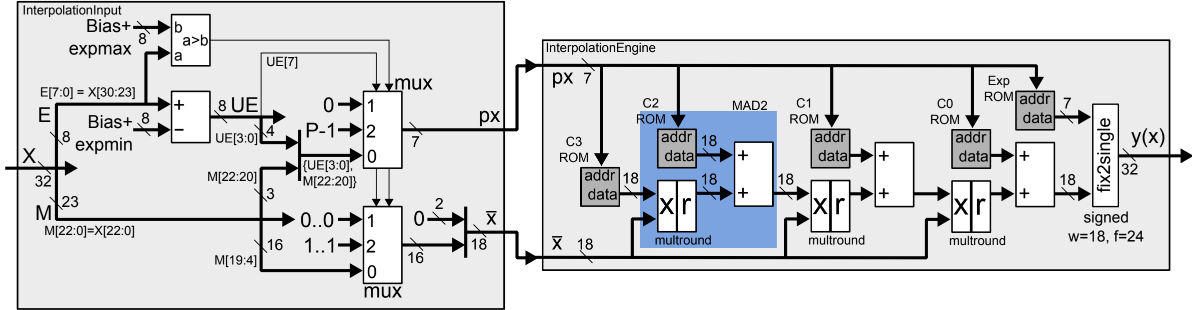

Figure 3.7: Block diagram of the Interpolation Engine connected to Interpolation Input decoder.

The size of the memories is given by the polynomial order, number of partitions, and word length of the coefficients and exponents. Let P be the number of partitions, n the order of the polynomial, l_{coeff} the number of bits of each coefficient, and l_{exp} the number of bits of the exponent, then the total number of bits that the Interpolation Engine implements in its ROM memories is $P * ((n+1) * l_{coeff} + l_{exp})$. The total number of MADs in each Interpolation Engine is n. The fix_multround units in the MADs are implemented using slices of 18-bit DSPs, which contain integer multipliers and rounding units. Increasing P and n improves accuracy but has an impact in memory and DSPs.

Interpolation Input decoder The Interpolation Engine uses the integer variable px to select the polynomial that matches the partition containing the real input x. However, the IE evaluates the centered and scaled fixed-point $\overline{x}$. To perform this transformation, the Interpolation Input (II) unit in figure 3.7 is located before the Interpolation Engine to provide it with the values of px and $\overline{x}$ from a given input x. The Interpolation Input is not an integral part of the Interpolation Engine, because px and $\overline{x}$ can be used by other Interpolation Engines. In this work, there are two Interpolation Input units that process w^{LJ} and w^{d}, and four Interpolation Engines for $u^{LJ}\left(w^{LJ}\right)$, $f^{LJ}\left(w^{LJ}\right)$, $u^{d}\left(w^{d}\right)$ and $f^{d}\left(w^{d}\right)$.

Since the input x is in floating-point representation with a biased exponent $E\left[7:0\right]$ and a normalized mantissa $M\left[22:0\right]$, the Interpolation Input takes advantage of this normalized representation to simplify the decoding of the logarithmic partitions, and does not require shift operations. It also performs saturation of out-of-range values by assigning the first and last polynomial to px, as well as the minimum and maximum value of $\overline{x}$ (zero and approx. two, respectively). Equations 3.12 and 3.13 describe the operations performed by the Interpolation Input.

$$px\,[7:0]\begin{cases}0 & UE < 0\\ P-1 & E > expmax + Bias\\ \{UE\,[3:0]\,,\, M\,[22:20]\} & otherwise\end{cases} \tag{3.12}$$

$$\overline{x}\,[17:0]\begin{cases}000.000000000000000 & UE < 0\\ 001.111111111111111 & E > ExpMax + Bias\\ \{00,\, M\,[19:4]\} & otherwise\end{cases} \tag{3.13}$$

Here, $UE = E - (expmin + Bias)$ is the unbiased exponent, $Bias$[5] is the bias of the exponent field in the IEEE-754 standard, and $expmin$ is the lower exponent in the input range. Note that the number of MSBs taken from the mantissa M to build px corresponds to the number of bits l_{lin} assigned to the linear partitions.

3.4.2 Storage elements

The system needs to access simulation parameters and input variables to start computing the forces and energies. The low-latency nature of this information requires that its storage is preferable in on-chip RAM or in an external SRAM. Hence, this system uses dedicated logic registers and on-chip 2- and 3-port memories (see figure 3.8) to store such information.

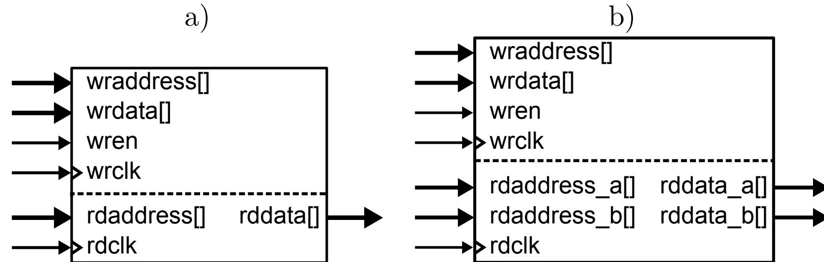

Figure 3.8: Block diagram of the memories. a) 2-port memory, b) 3-port memory.

These memories are not only dual-port, but also dual-clock memories. This allows to improve the overall performance of the datapath by establishing two asynchronous clock domains. Information coming from and going to the Functional Pipeline works with the denominated internal clock, while the external clock is the one used by the core's IO bus to write inputs and read results[6]. A short description of the datapath memories is presented in the following.

[5] The bias for single-precision IEEE-754 data is 127.
[6] IO interfacing will be shown later in section 3.6

Particle Memory The Particle Memory stores information about the position $\mathbf{x}$ and the type of each particle using two internal memories called Type Memory and Coordinate Memory. This memory has three ports, because the memory is written from the external IO bus, and the Functional Pipeline requires a pair of particles at the time. There are two different ways to implement in hardware this kind of memory.

The first implementation utilizes two instances of the same dual-port memory, which have a common write port, and use separate read ports. The second implementation is using just one dual-port memory, but the read access is done using a clock 2x faster than the pipeline clock. This work implements the first method, but with the disadvantage that it uses two times more memory bits.

The depth of this memory depends on the maximum number of particles. The width depends on the word length of each coordinate, and the maximum number of elements (number of bits of the field *type*). The coordinates are stored in 32-bit fixed-point format with 8-bit integer part. The total number of bits is $max\,(N_{atoms}) * (length\,(type) + 3 * length\,(coordinate))$. The $length\,(type)$ was configured to 4, i.e. the system supports up to 16 different elements or particle types.

Force Memory The Force Memory is a two-port memory located at the end of the Functional Pipeline. The depth of this memory matches the depth of the particle memory to support the same number of particles, but with the difference that the width only depends on the format chosen for the components $\mathbf{F}\,[d]$ of the force. The size of this memory in number of bits is $max\,(N) * 3 * length\,(force)$, where $length\,(force) = 32$ because of the floating-point representation.

Parameter Memories There are three instances of the Parameter Memory spread along the Functional Pipeline to store the values of epsilon (ϵ_{ij}), sigma (σ_{ij}^{-2}) and charge ($q_i q_j \alpha$). These values are particle type dependent representing a squared 2-D memory. In order to simplify addressing in a 1-D implementation, the maximum number of elements was chosen to be a power of two.

The read address is driven by the signal $type_{ij}$, which is no more than the concatenation of the signals representing the type of the particle i and j. The size of this memory is $(length\,(type))^2 * length(parameter)$, where $length(parameter) = 32$ because each parameter is stored in floating-point representation.

Registers The following parameters and variables are stored in registers: the box size ($\mathbf{L}\,[1]$, $\mathbf{L}\,[2]$, $\mathbf{L}\,[3]$), the cutoff radius (r_{cut}), alpha (α^2), the number of atoms (N), and the resulting potentials (V^{LJ}, V^{d}).

3.4.3 Computation of the squared distance

The computation of the Euclidian distance between pairs of particles is performed by the Squared Distance unit. This is preceeded by the MIC unit to implement the minimum image convention for periodic molecular systems. These units are explained in detail as follows.

3.4.3.1 Minimum Image Convention - MIC

Based on the Minimum Image Convention method presented in section 2.1.3, and recalling (2.12):

$$\mathbf{r}_{ij}[d] = \begin{cases} \mathbf{x}_i[d] - \mathbf{x}_j[d] - L & if\ (\mathbf{x}_i[d] - \mathbf{x}_j[d]) > {}^{L}\!/_{2} \\ \mathbf{x}_i[d] - \mathbf{x}_j[d] + L & if\ (\mathbf{x}_i[d] - \mathbf{x}_j[d]) < -{}^{L}\!/_{2} \\ \mathbf{x}_i[d] - \mathbf{x}_j[d] & otherwise, \end{cases}$$

the MIC unit computes a single component of the distance vector $\mathbf{r}_{ij}$. There are three MICs, one for each dimension. Each MIC requires the coordinates of each particle and the box size in the corresponding axis. According to the requirements presented in section 3.1, the effective range of these variables allows the computation to be perfectly done in fixed-point representation without significant loss of precision, which results very convenient because additions, subtractions and comparisons in floating point are very hardware expensive. The format for this representation is 8-bit integer part due to the range of ±128 Å, and 24-bit fractional part that offers the same resolution than single-point representation for numbers out of the range ±0.5 Å.

Figure 3.9 shows the block diagram of the MIC unit. It is built from a subtractor to compute the difference between the coordinates, a comparator to compare with the half length of the box, a mux to select the compensation, and an adder to apply the possible compensation by $\pm L$. The components of the distance are given in fixed-point representation to the Squared Distance unit, and are further converted to floating point to compute the components of the pair force.

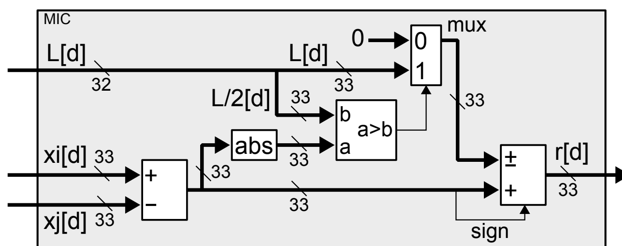

Figure 3.9: Block diagram of the MIC unit.

3.4.3.2 Squared Distance

The Squared Distance unit computes the squared magnitude r_{ij}^2 of the distance vector $\mathbf{r}_{ij}$ based on its components and indicates if this is inside the cutoff radius with the flag $inside_cutoff$. The inputs $\mathbf{r}[d]$ coming from the MIC block are squared using DSP blocks and then are summed up using a parallel adder.

The cutoff radius r_{cut} (also in fixed point) is also squared using DSPs and then is compared with the squared distance r_{ij}^2 by an unsigned comparator. The signal $inside_cutoff$ comes from this comparator, and is high for pairs inside the cutoff radius. This flag enables the computation and accumulation of potentials and forces further in the Functional Pipeline. Finally, the squared distance is converted to floating point from its full-resolution fixed-point representation, because this variable is going to be multiplied with floating-point data. Figure 3.10 shows the block diagram of the Squared Distance unit.

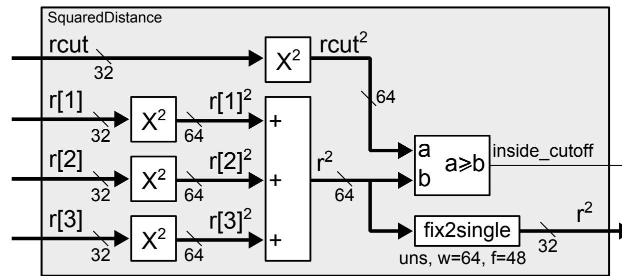

Figure 3.10: Block diagram of the Squared Distance unit.

3.4.4 Floating-point units

The floating-point arithmetic used in the FPGA coprocessor requires only three basic operations: multiplications, additions and accumulations. This are implemented using the FPSingleMult, FPSingleAdder and FPSingleAcc units.

3.4.4.1 Single-precision floating-point multiplier and adder

Figure 3.11 shows the block diagram of the FPSingleMult unit and the FPSingleAdder unit. They are based on Altera's IPs for floating-point arithmetic (altfp_mult and altfp_addsub), which are optimized to offer high performance, implement exception handling, and can be parameterized to fit the design requirements in terms of latency and precision.

3.4.4.2 Single-precision floating-point accumulator

Although Altera has a wide catalog of IPs for floating-point, it does not have one for accumulation. Therefore, it has been built by using the components shown in figure 3.12, which

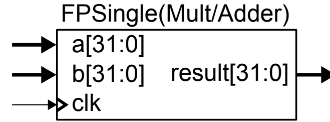

Figure 3.11: Block diagram of the floating-point units for multiplication and addition.

are two format converters (single2fix and fix2single) and the Altera's altaccumulate for the pipelined accumulation. The internal fixed-point representation uses 128-bit words with 64-bit fractional part. This allows to accumulate full-resolution single-precision data in the range $\pm(2^{-41}, 2^{63})$.

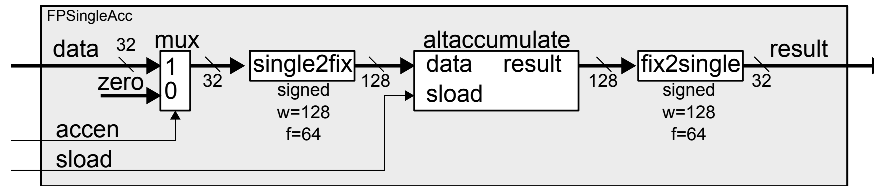

Figure 3.12: Block diagram of the FPSingleAcc unit.

This unit is relative big in comparison to others in the design, because of its internal 128-bit accumulator. There are five accumulator in the Functional Pipeline: two of them accumulate pair potentials U_{ij} to compute the total potential V; and other three accumulate pair forces $\mathbf{F}_{ij}[d]$ to obtain the resulting force $\mathbf{F}_i[d]$.

The block has two control signals: *sload* loads data in the internal accumulator to restart the accumulation; and *accen* enables accumulation of new data by selecting between the input data and zero.

According to the dataflow in the Functional Pipeline, data are initially loaded using the *sload* signal, and then new data are accumulated continuously in fixed-point representation. The floating-point value in the accumulator can be stored at any moment in a separate register/memory. The accumulation process by adding floating-point data leads to loss of precision when small values are added between big values. That does not happen in this accumulator and becomes its strongest advantage, because data are aligned only before their final storage. Other advantage is that the system does not have to bear with the latency produced by FP adders when data are brought back from memory.

3.4.5 Floating point/Fixed point conversion

Since the system performs computations using floating-point data and different fixed-point formats, it is necessary to carry out conversions between both formats along the datapath. Although Altera provides IP cores for these conversions that simplify the design and that

offer exception handling, they were not implemented here because they use many resources and do not support all the required conversions[7].

These custom converters designed in this work are not fully parameterized, and were written specifically to solve the requested conversions to/from single-precision IEEE-754 data. By now, only the ZERO exception is handled. Further work will be oriented to optimization, full parameterization, and extension of exception handling. The following subsections present the set of converters used in the LJEwDir core and their architecture.

3.4.5.1 Fixed point to Floating point converters

Four converters are used to change from two's complement fixed-point representation to floating point. The notation of the name of these converters uses sig/uns for signed/unsigned data, w# for the word length, f# for the length of the fractional part.

fix2single__sig__w32__f24 converts $\mathbf{r}_{ij}[d]$ between the MIC units and the FP multipliers that operate with F_{ij} to obtain $\mathbf{F}_{ij}[d]$.

fix2single__uns__w64__f48 converts r_{ij}^2 for further FP multiplication with σ_{ij}^{-2} and α^2 to obtain the transforming variables w^{LJ} and w^d, respectively.

fix2single__sig__w18__f15 converts the resulting $p(\overline{x})$ together with *expy* in the Interpolation Engines to obtain $y(x)$ in floating point.

fix2single__sig__w128__f64 converts accumulated data in the FPSingleAcc.

Figure 3.13 shows the block diagram of the generic architecture of a 32-bit fix2single converters. The idea is to get the absolute value of the input data and normalized it by aligning it using a series of barrel shifters-to-the-left. The number of places to shift is given from priority encoders that identify the most significant '1'. At the end, the aligned data is part of the mantissa; and the sign was already extracted at the beginning. The output exponent is the result of summing the bias in the IEEE-754 standard, the input exponent *expin*, the total number of shifts, and a number that depends on the relationship between the integer and fractional part.

3.4.5.2 Floating point to Fixed point converters

Two converters from floating point to fixed point are used in the system. The name of each converter follows the notation presented above for fix2single converters.

[7] 128-bit conversion is not supported by Altera floating-point IPs

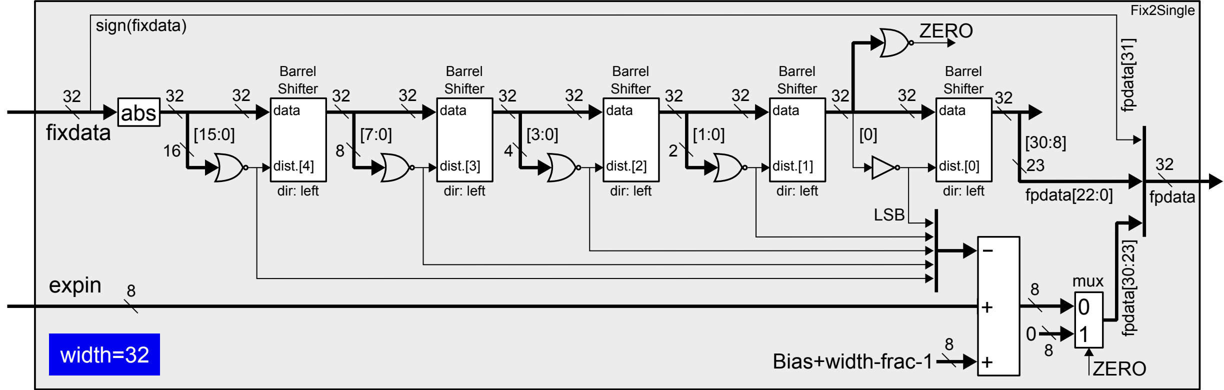

Figure 3.13: Generic architecture of the fix2single converter for 32-bit fixed-point data.

single2fix__sig__w33__f24 converts data coming from the IO bus that represent the cutoff radius r_{cut}, and the box size $\mathbf{L}[d]$ and the components $\mathbf{x}[d]$ of the position. Only $\mathbf{L}$ is unsigned data with 8-bit integer part, while the others are signed data. For this reason it converts to 33 bits instead of 32 bits, i.e. to treat $\mathbf{L}$ as signed data, but ignoring the extra bit of sign.

single2fix__sig__w128__f64 converts incoming data in the FPSingleAcc that go to the internal fixed-point accumulator.

Figure 3.14 shows the block diagram of the generic architecture of the single2fix converters. Initially, the converter checks for the ZERO exception to addition '1' or not to the mantissa. Next, it unbiases the exponent and uses the sign to represent the mantissa in two's complement. It is then followed by a barrel shifter-to-the-right that uses arithmetic shifting to keep the integrity of the two's complement representation.

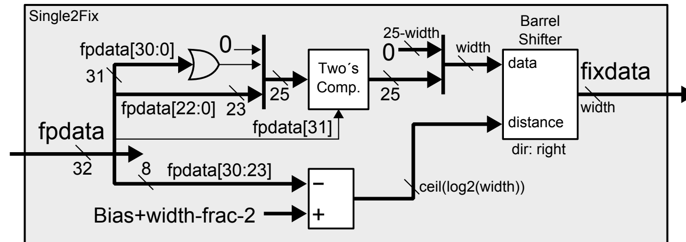

Figure 3.14: Generic architecture of the single2fix converters.

3.5 Controller of the LJEwDir core

The control of the datapath in the LJEwDir core is done by the Pair Controller (PC). It controls the flow in the Functional Pipeline, and addresses the particle memory to extract all

permitted pairs according to the given number of particles. For N particles it needs $N*(N-1)$ cycles to read all pairs skipping those pairs between one particle and itself. Figure 3.15 shows the block diagram of the Pair Controller.

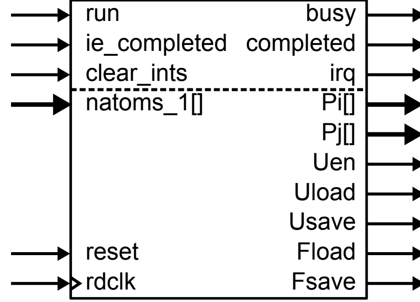

Figure 3.15: Block diagram of the Pair Controller.

The Pair Controller receives the signals *reset*, *run*, *ie_completed* and *clear_ints* from a master controller. Outputs to the master are the signals *busy*, *completed*, and *irq*. This controller can generate interruptions at the *irq* port when *completed* is high and the interruptions are enabled by setting *ie_completed*. The PC drives several control signals of the Functional Pipeline. Two of them are the addresses P_i and P_j of the current pair of particles. Other five signals are single-bit signals that control accumulation and storage of the potentials and forces. The state of the control signals at cycle n is described by the following conditions:

$$U_{en}^{n} \leftarrow P_i^n < P_j^n \tag{3.14}$$

$$U_{load}^{n} \leftarrow n = 0 \tag{3.15}$$

$$U_{save}^{n} \leftarrow (P_i^n = N-2) \wedge \left(P_j^n = N-1\right) \tag{3.16}$$

$$F_{load}^{n} \leftarrow (n = 0) \vee \left(P_j^n = P_i^n - 1\right) \tag{3.17}$$

$$F_{save}^{n} \leftarrow \left(P_j^n = P_i + 1\right) \wedge \left(P_j^n = N - 1 \wedge P_i^n = 0\right) \tag{3.18}$$

The PC simplifies the implementation of these expressions by reusing shared and delayed signals according to the sequence that it performs to search for pairs. This sequence starts with $P_i = 0$ and $P_j = N-1$ and finishes with $P_i = N-1$ and $P_j = 0$. Figure 3.16 shows an example of this sequence for $N = 5$ particles. The accumulators of pair potentials controlled by U_{en} have a maximum efficiency of 50% in normal operation. This efficiency decreases when

more pairs are out of the cutoff radius.

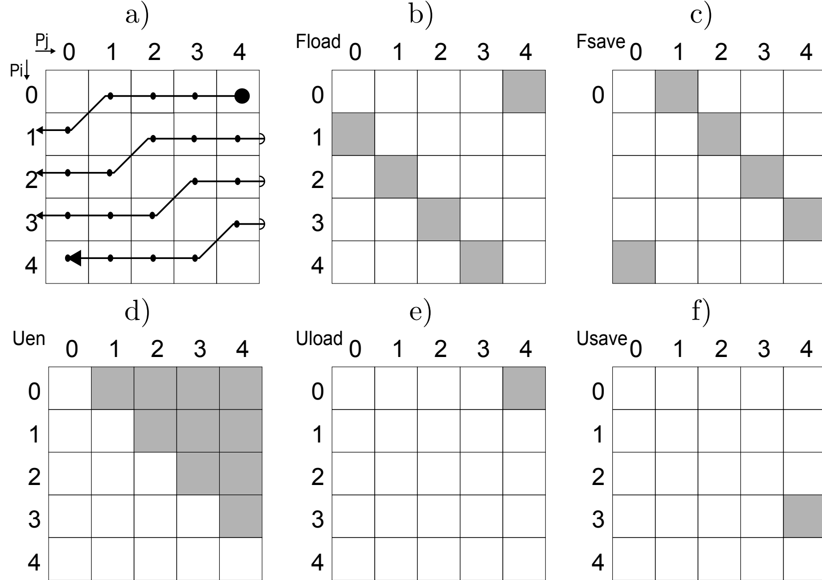

Figure 3.16: State of the signals controlled by the Pair Controller for $N = 5$. a) Particle counters, b) Fload, c) Fsave, d) Uen, e) Uload, f) Usave.

The Pair Controller uses the same clock than the Functional Pipeline, i.e. it works with the so-called internal clock. Once the Pair Controller starts, the number of cycles τ that are necessary for completion is given by $\tau{=}N\,(N-1) + maxlatency$, where *maxlatency* is the maximum latency of the Functional Pipeline. Two separate Moore FSMs (Finite State Machines) control the behavior of the Pair Controller: The Pair FSM and the Interruption FSM.

Pair FSM The Pair FSM has five states to control the search of pairs as shown in the state diagram of figure 3.17. It basically waits for *run* to start the computation, then generates all $N\,(N-1)$ pairs, and finally uses the *endcomputing* signal to know when all data have already been processed by the Functional Pipeline.

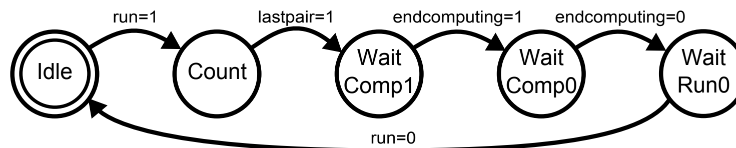

Figure 3.17: State diagram of the Pair FSM.

Interruption FSM This simple two-state FSM is used to hold the *irq* signal when an interruption is generated until the interruption is properly acknowledged by setting *clear_ints*. The FSM enters in IDLE state after reset, and changes to INTERRUPTED when both signals *completed* and *ie_completed* are active in the same cycle. The FSM returns to IDLE when *clear_ints* is 1. Figure 3.18 shows the state diagram of the Interruption FSM.

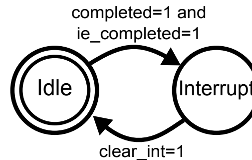

Figure 3.18: State diagram of the Interruption FSM.

3.6 Memory-Mapped LJEwDir Core

The LJEwDir core is the memory-mapped IP core designed in this work that integrates not only the datapath and the Pair Controller, but also an IO Bus, an address decoder, and a Control & Status Register (CSR). The core can be part of a bigger system with other MD accelerators, and work as a 32-bit slave capable to generate interruptions. Figure 3.19 shows the block diagram of the core and indicates the two asynchronous clock domains: one for the Pair Controller and the Datapath, and other for the IO Bus.

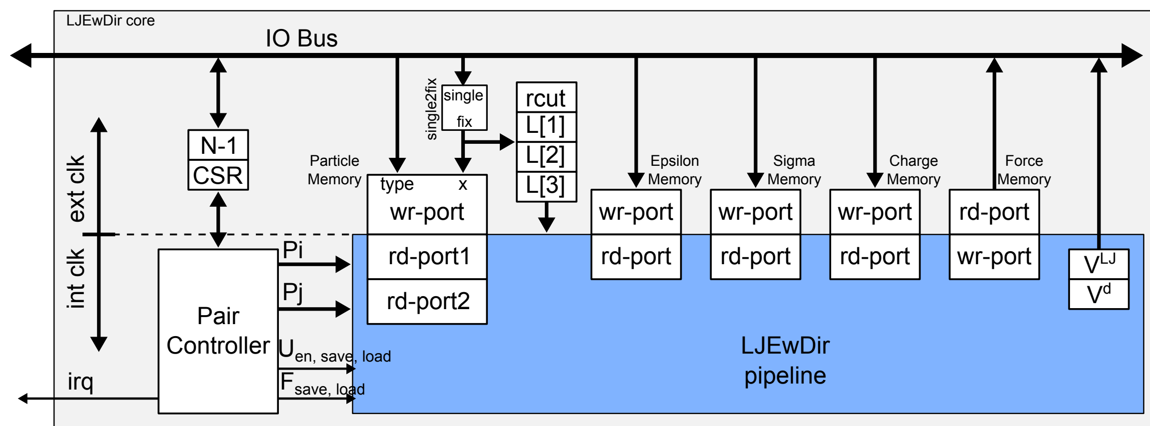

Figure 3.19: Overview of the memory-mapped LJEwDir core.

The address decoder enables write operations to registers and memories according to the address space presented in table 3.2. This 32-bit address space is for maximum 16k particles and 16 elements. The IO Bus is connected to all registers and memories that store simulation parameters and variables. There is a single2fix converter that converts data from the IO Bus to the registers $\mathbf{L}[d]$ and r_{cut}, and to the Coordinate Memories that are part of the Particle

Memory. The *wren* signals associated to the these data are properly synchronized with the single2fix converter.

Table 3.2: 32-bit address space of the LJEwDir core for up to 16k particles and 16 elements.

Base Address	[17]	[16]	[15]	[14]	[13:8]	[7:0]	Name	Access
0x0000	0	0	0	0	——		$\mathbf{x}[1]$	W
					——		$\mathbf{F}[1]$	R
0x0400	0	0	0	1	——		$\mathbf{x}[2]$	W
					——		$\mathbf{F}[2]$	R
0x0800	0	0	1	0	——		$\mathbf{x}[3]$	W
					——		$\mathbf{F}[3]$	R
0xC00	0	0	1	1	——		$type$	W
0x1000	0	1	0	0	xxxx	—	ϵ_{ij}	W
0x1400	0	1	0	1	xxxx	—	σ_{ij}^{-2}	W
0x1800	0	1	1	0	xxxx	—	$q_i q_j \alpha$	W
0x1C00	0	1	1	1	xxxx	xxxx	α^2	W
0x2000	1	0	0	0	xxxx	xxxx	$\mathbf{L}\,[1]$	W
0x2400	1	0	0	1	xxxx	xxxx	$\mathbf{L}\,[2]$	W
0x2800	1	0	1	0	xxxx	xxxx	$\mathbf{L}\,[3]$	W
0x2C00	1	0	1	1	xxxx	xxxx	r_{cut}	W
0x3000	1	1	0	0	xxxx	xxxx	V^{LJ}	R
0x3400	1	1	0	1	xxxx	xxxx	V^d	R
0x3800	1	1	1	0	xxxx	xxxx	$N_{atoms} - 1$	RW
0x3C00	1	1	1	1	xxxx	xxxx	CSR	RW

The CSR can be addressed by an external master to control the Pair Controller and to check its status. A short description of the fields in the CSR is presented in table 3.3.

3.7 Top Level and the Avalon Memory-Mapped System

The memory-mapped system integrates the memory-mapped LJEwDir core presented in the last section with other peripherals, such as the PCI Express (PCIe) core. This system is finally instantiated in the top level entity, which includes other modules such as PLLs, IPs for the high-speed links, and connection to pins of the FPGA for clock sourcing and IO. Figure 3.20 shows an overview of the whole system in the FPGA and development board.

The design of the memory-mapped system was carried out using QSys, an embedded tool in the Quartus II 11.0 software that is very useful to automatically generate systems based on memory-mapped (MM) and streaming (ST) interfaces. The Avalon Interconnect Fabric manages transfers between the masters and slaves in the system, i.e. it does all arbitration, address alignment, data alignment, and more. This interconnection fabric is based on a Network on Chip (NoC) architecture that can improve performance up to 2x with respect to

Table 3.3: CSR of the LJEwDir core.

Bit	Name	Access	Description
[0]	RUN	RW	Set this bit to start the force and potential computation. It must be cleared before starting a new computation.
[1]	IE_COMPLETED	RW	Enables interruption requests (IRQ). When this bit is 1, the core generates an IRQ when the status bit COMPLETED is 1.
[13:2]	*reserved*		
[14]	SWRESET	RW	Set this bit to reset the core.
[15]	CLEAR_INTS	RW	Writing 1 to this bit clear all pending interruptions.
[16]	BUSY	R	This bit is 1 when a computation of non-bonded interaction is in progress.
[17]	COMPLETED	R	This bit is 1 when the core has completed a computation. An interruption is generated if the bit IE_COMPLETED is set.
[31:18]	*reserved*		

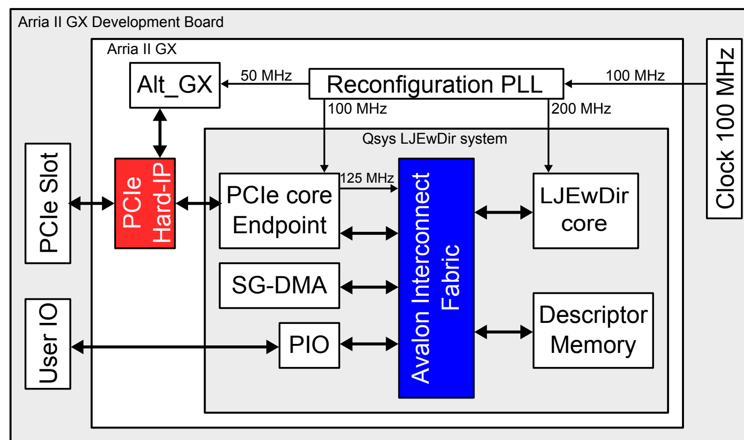

Figure 3.20: System overview.

the older Altera's SOPC Builder.

The embedded peripherals that were integrated in the memory-mapped system are presented in the following.[8]

PCI Express Endpoint The PCIe endpoint allows high-speed transfers with the host to minimize loss of performance due to communication overhead. Using the PCIe IP Compiler [7], the PCIe Hard-IP in the Arria II GX was configured as following:

[8]More information about the Avalon Interface and the embedded peripherals are found in [4, 3].

- Generation 1 Link with four lanes (Gen1 x4) that works @ 2.5 Gbps. It achieves a theoretical maximum performance of approx. 850 MB/s. This result is computed from the number of lanes, the operating frequency, the maximum payload, and the overhead produced by the encoding 8/10 in the Data-Link Layer and the additional frame bits in the PHY layer.

- 32-bit non-bursting CRA Avalon-MM slave to access internal control and status registers. Using this slave, the host is able to modify the address translation table and to enable/check PCIe interruptions.

- 64-bit bursting TX Avalon-MM slave to execute upstream requests.

- Dynamic Address Translation table with two entries of 20 bits. Each entry allows to address 1 MB of physical memory in the host.

- Two Base Address Registers (BAR) were configured to access the Avalon system from the host. The BAR[1:0] is a 64-bit prefetchable master, which is only connected to the LJEwDir core. The BAR[2] is a 32-bit non-prefetchable master connected to the CRA of the PCIe core, to the CSR of the SG-DMA core, to the descriptor memory, and to the PIO core.

- PCIe interruptions to the host coming from the Avalon bus, which are generated by the LJEwDir core or by the SG-DMA core.

- Maximum payload of 256 Bytes.

Scatter-Gather Direct Memory Access (SG-DMA) This core implements high-speed data transfer between two components. The transfer mode can be memory-to-memory, stream-to-memory, and memory-to-stream. It is provided with four Avalon interfaces:

- The descriptor master is connected to the Descriptor Processor. This processor reads a descriptor chain stored in a memory, and starts a single DMA transfer according to the parameters given by each descriptor. The processor pushes commands into an internal command FIFO that can be accessed by the read and write blocks.

- The read and write masters are connected to the Read and Write blocks, whose implementation depends on the transfer mode, i.e. the interface can be either an Avalon-MM or an Avalon-ST interface. The read and write blocks are connected to each other by an internal data FIFO.

- The CSR is a slave used to control the SG-DMA core.

The SG-DMA core is included in the system to perform memory-to-memory transfers between the on-chip memories in the LJEwDir core and the host's main memory. Both master interfaces create a bridge between the TX interface of the PCIe core and the slave port of the LJEwDire core. Bursting transfers for up to 256 Bytes are configured to match the maximum payload of the PCIe core. The SG-DMA sends an interruption to the PCIe core when the last descriptor is processed.

Descriptor Memory The descriptor memory is a small on-chip RAM to allocate the descriptor chain that is processed by the SG-DMA core when transferring data to/from the host's main memory. In this work, this memory can store up to 32 descriptors. Each descriptor uses 1024 bits that are organized as a set of 32 registers of 4 Bytes. These registers contain the source address, the destination address, the number of bytes to be transferred, the pointer to the next descriptor, and an embedded control and status register.

PIO The PIO (Parallel Input/Output) core is used by the system to interact with user IOs such as leds, dip switches and push buttons for configuration and status.

4. Hardware Verification and Synthesis of the FPGA coprocessor

This chapter describes the verification methods used for the functional verification of the RTL design, as well as the resource usage and timing analysis of the design when it was implemented on a real FPGA.

4.1 Simulation of the LJEwDir core in ModelSim and Matlab

Functional simulations were run in ModelSim-Altera for the LJEwDir core and most of its functional building units. Moreover, by using Matlab and its embedded Fixed Point Toolbox (see section 3.3), several units were studied more deeply when comparing with their fixed-point and floating-point models in software. The use of this mathematical software enhances the verification, because it allows to create more complex stimuli and generate more complex reports, such as 1-D plots, 2-D plots, statistical analysis, etc. The testbenches written in VHDL to be run in Modelsim read the stimuli generated as text files in Matlab. During simulation, the testbenches write the results of interest also in text files that are further read by Matlab. Most of these units have a latency associated due to their pipeline architecture, so it is very important to keep in mind this time shift when reading the time series. These functional simulations do not give any information about resource usage and timing, but are useful to detect error sources, loss of precision, and connectivity problems.

4.1.1 Simulation of the functional building units

4.1.1.1 Minimum Image Convention - MIC

The MIC unit receives three signals corresponding to the same axis: two coordinates and the box size. These signals and the output are all fixed-point data. To verify that this unit works correctly, it was necessary to perform a constrained-random verification, because a formal verification is very difficult taking into account that generating all possible combinations results very exhaustive and unnecessary. This method generates random values for the input

coordinates distributed in a random range, which starts at a random starting point and that spans according to the random box size. The values are converted to fixed point considering the convergent rounding method.

Figure 4.1 shows the simulation results of the MIC unit. First, it shows the input parameters of the simulation that are distributed in several ranges. Then, the software ouput of Matlab (*ref*) and the hardware output Modelsim (*out*) are plotted. The binary logarithm of the difference shows that only the -24th bit is different, and it corresponds to the LSB in the fixed-point representation of these variables, which is 8-bit integer part and 24-bit fractional part.

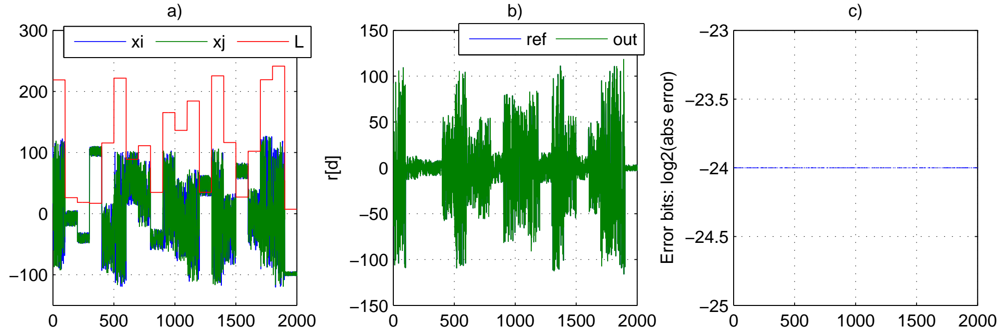

Figure 4.1: Constrained-random verification of the MIC. a) Inputs, b) Software and hardware outputs, c) Error bits.

4.1.1.2 Squared Distance

The Squared Distance unit also uses constrained-random verification to generate a set of inputs limited by a random range. Figure 4.2 shows the verification results of the Squared Distance unit.

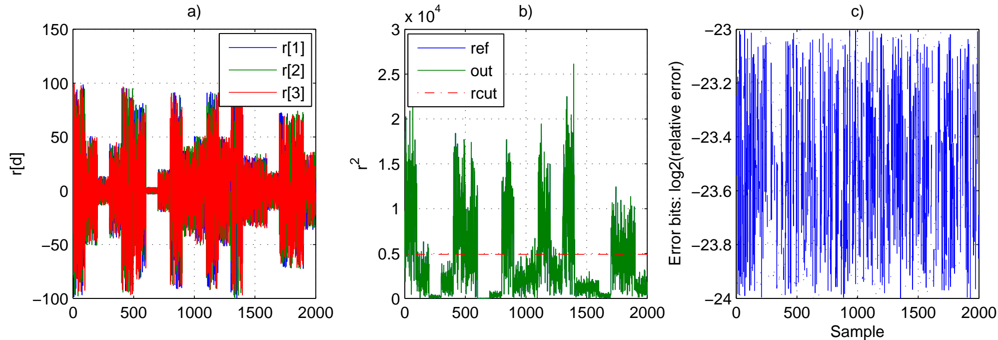

Figure 4.2: Constrained-random verification of the SquaredDistance unit. a) Coordinates of the input position, b) SW and HW outputs, c) Relative error.

Values of the three components of $\mathbf{r}_{ij}$ are shown first. The cutoff radius r_{cut} does not

change during simulation. The floating-point outputs of the software and hardware models differ between the -23th and the -24th bit. This is due to the conversion to single-precision representation that may lose precision, since the rounding method of the conversion in hardware is simple truncation. The flag $inside_cutoff$ was also verified but is not shown in the figure.

4.1.1.3 Interpolation Engine

Although section 3.3 presented the methodology oriented to ensure that interpolation method suits the design requirements, it does not mean that its hardware implementation works fine. The verification of the Interpolation Engine and the Interpolation Input was done separately, and then these units were tested together.

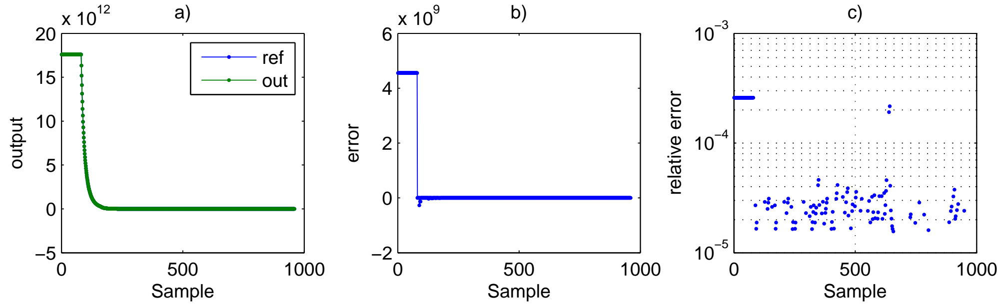

Figure 4.3: Verification of the Interpolation Engine together with the Interpolation Input for the function $u^{LJ}\left(w^{LJ}\right)$. a) SW and HW outputs, b) Error, c) Relative error.

The verification process starts generating several points x_t in each partition and uses direct evaluation to obtain $y_t\left(x_t\right)$. The values x_t are read by the Interpolation Input unit and are decoded to px and $\overline{x}$ for the use of the Interpolation Engine. The floating-point outputs of the hardware implementation are compared with the values $y_t\left(x_t\right)$. Simulation results for the function $u^{LJ}\left(w^{LJ}\right)$ when using both II and IE units are shown in figure 4.3. Here, values of $x_t = w^{LJ}$ out of the considered range $\left[2^{expmin}, 2^{expmax}\right)$ were also generated to test the saturation. The values in the range of interest present a relative error below 10^{-4}, except for some points where $u^{LJ}\left(w^{LJ}\right)$ approximates zero[1].

4.1.1.4 Floating-point units

The verification of the floating-point units uses the Matlab's built-in data type *single* to compare results. All units were tested with random values, also including the exception ZERO.

[1]The Lennard-Jones potential and force presents high relative errors around the zero-crossing in comparison to the rest of values. This does not happen for the direct-space Ewald potential and force, since they do not cross zero.

According to the simulation results in figure 4.4, the FPSingleUnit works perfectly and throws the same results than Matlab. On the other hand, results of the FPSingleAdder in figure 4.5 present a relative error of 10^{-7}, i.e. in the LSB, which suggests that is an error due to rounding.

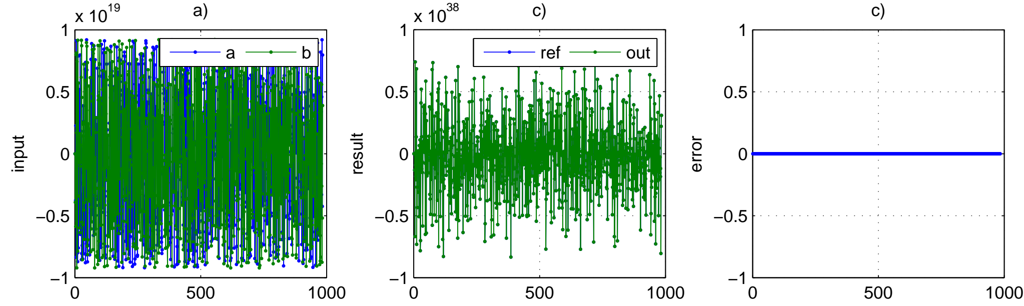

Figure 4.4: Random verification of the FPSingleMult unit. a) Inputs, b) SW and HW outputs, c) Error.

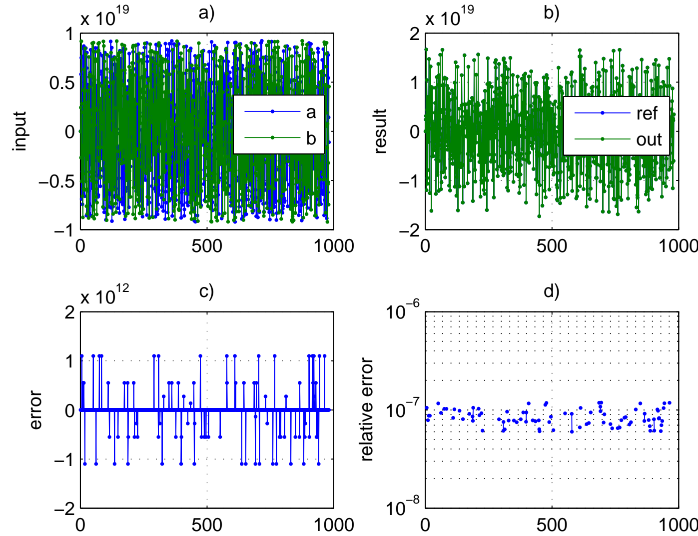

Figure 4.5: Random verification of the FPSingleAdder unit. a) Inputs, b) SW and HW outputs, c) Error, d) Relative error.

Simulation results for the FPSingleAcc unit are shown in figure 4.6. Results show that most of the relative errors are below 10^{-4} and above 10^{-7} (again, the LSB)[2]. For some simulations with input values near to 10^{19} ($\approx 2^{63}$), the accumulator wraps due to overflow of the internal fixed-point accumulator.

[2]However, since the hardware accumulator does not lose information with continuous rounding as software does, these errors are considered errors of the software implementation.

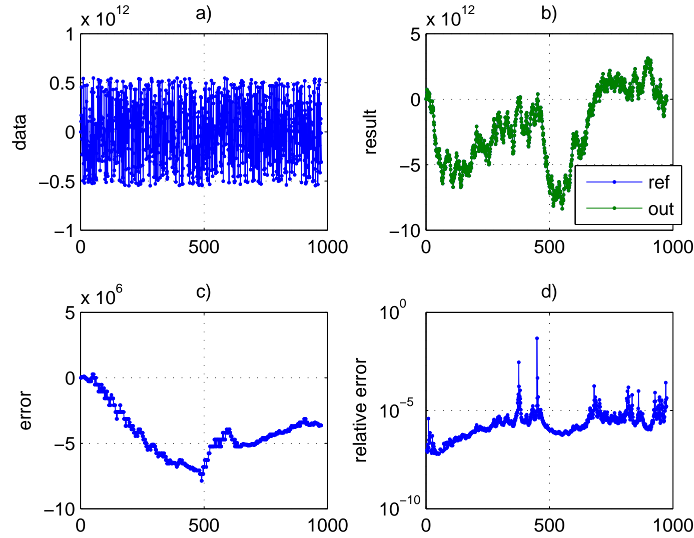

Figure 4.6: Random verification of the FPSingleAcc unit. a) Inputs, b) SW and HW outputs, c) Error, d) Relative error.

4.1.1.5 Pair Controller

Simulations of the Pair Controller verify the integrity of the sequence performed by the controller, as well as the correct state of the control signals. Simulations were run for a low number of atoms (5 atoms to 100 atoms), reflecting the importance of the WAIT_COMP* states to ensure that the Functional Pipeline has really completed its task. Figure 4.7 shows a simulation for 15 atoms, where the maximum latency of the Functional Pipeline is 83 cycles[3]. This simulation does not show the signals related to interruption generation.

4.1.2 Simulation of the Datapath

Simulation of the Datapath required scripts in Matlab to perform some tasks of the MD software, such as read positions and parameters from configuration files, and compute forces and potentials from this information. Matlab generates all the data and address for the memories and registers in the core, and performs conversions when required. On the other hand, the testbench stores the data in the corresponding memory space.

The verification of the Datapath was divided into three stages. The following verification results are presented considering a set of 100 atoms.

Stage 1 The first stage includes the Particle Memory, the MICs, and the SquaredDistance. This stage only processes the positions of the particles. The particle type is not relevant for this computation. Figure 4.8 shows verification results of the first stage of the datapath for

[3]This is the maximum latency of the synthesized design.

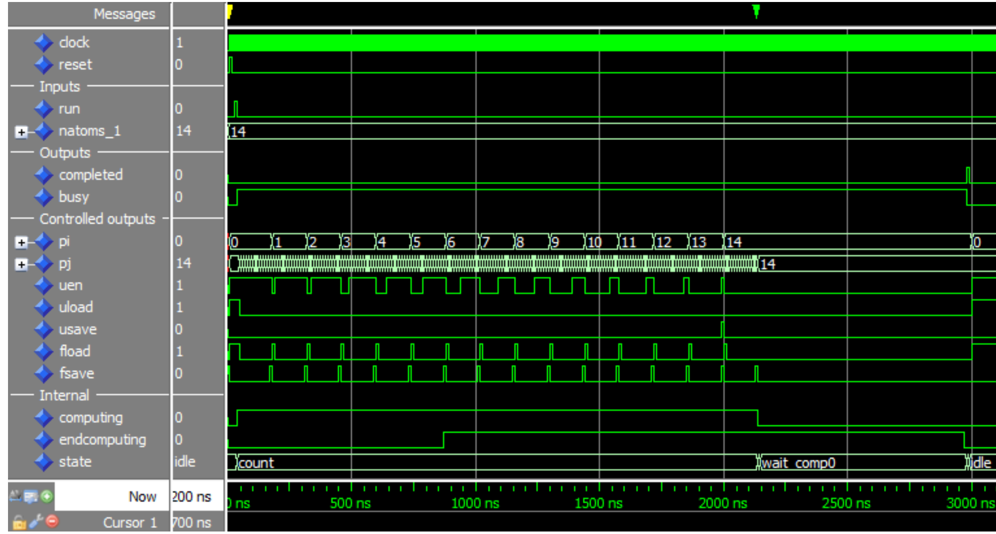

Figure 4.7: Simulation results of the Pair Controller for 15 atoms.

the computation of the squared distance between one particle and the rest in the simulation box. Again, the relative error is around 10^{-7} due to LSB rounding.

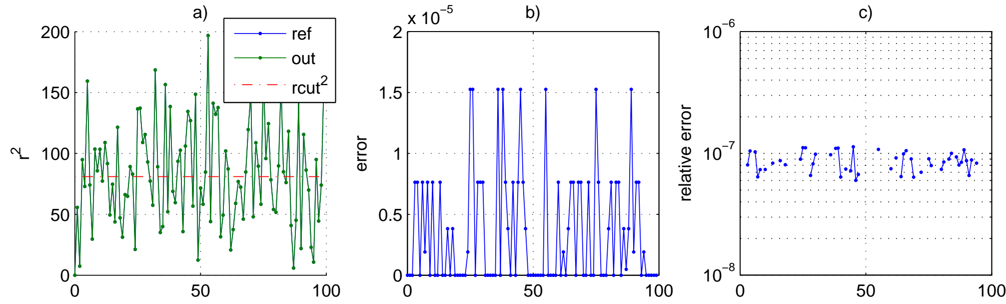

Figure 4.8: Verification results of the distance between one particle and the rest of particles. a) SW and HW outputs, b) Error, c) Relative error.

Stage 2 The second stage is more complex than the first stage. It includes the FPSingleMults and the Interpolation Engines. The squared distance r_{ij}^2 and the parameters ϵ_{ij}, σ_{ij}^{-2}, $q_i q_j \alpha$ and α^2 are computed in double precision in Matlab, and are converted to single precision for storage. The pair potentials and functions are computed by direct evaluation in Matlab and are compared with the hardware results.

Figure 4.9 and 4.10 show verification results of the second stage of the datapath. Figure 4.9 shows that the potential $U_{ij}^{LJ}(r_{ij})$ has a relative error between 10^{-4} and 10^{-5} that increases for long distances. This phenomenon is better appreciated for $U_{ij}^{d}(r_{ij})$ in figure 4.10. This happens when the transforming variables $w^{LJ} = \sigma_{ij}^{-2} r_{ij}^2$ and $w^d = \alpha^2 r_{ij}^2$ exceed the upper

limit of the range for interpolations given by *expmax*. However, values of these functions are still very small in comparison to the others in the range of interest.

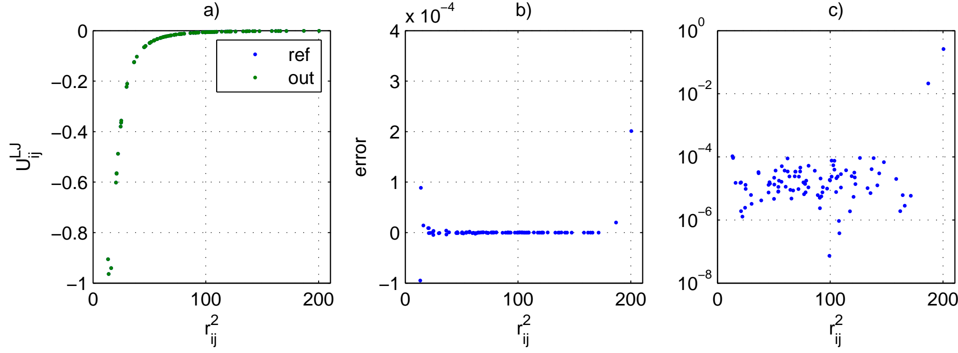

Figure 4.9: Verification results of $U_{ij}^{LJ}(r_{ij})$ between one particle and the rest with $\sigma_{ij} = 3.4050$. a) SW and HW outputs, b) Error, c) Relative error.

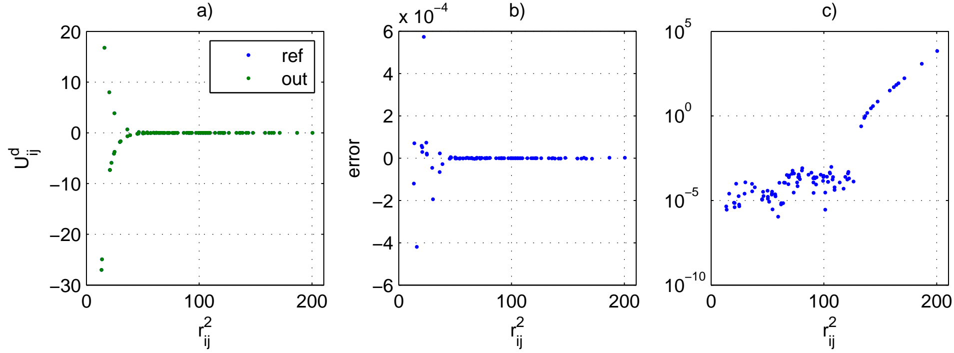

Figure 4.10: Verification results of $U_{ij}^{d}(r_{ij})$ between one particle and the rest with$\alpha = 0.349$ Å^{-1}. a) SW and HW outputs, b) Error, c) Relative error.

Stage 3 The third and last stage of the Datapath is conformed by one FPSingleAdder and several instances of FPSingleMult and FPSingleAcc. There is also a Force Memory and a couple of registers for the total potentials. The components $\mathbf{r}_{ij}[d]$ of the distance are introduced in floating-point representation. Matlab evaluates all the pair potentials and forces for the given atomistic configuration and creates stimuli for the testbench. Matlab also writes the control signals, which are easily computed from the matrix representation of $(P_i,\, P_j)$ like in figure 3.16. Figure 4.11 shows simulation results for the second component of the resulting force over each particle ($\mathbf{F}_i[2]$), that presents a relative error below 10^{-5}. In this case, relative errors of the total potentials V^{LJ} and V^d are $2.78 * 10^{-2}$ and $3.14 * 10^{-5}$, respectively.

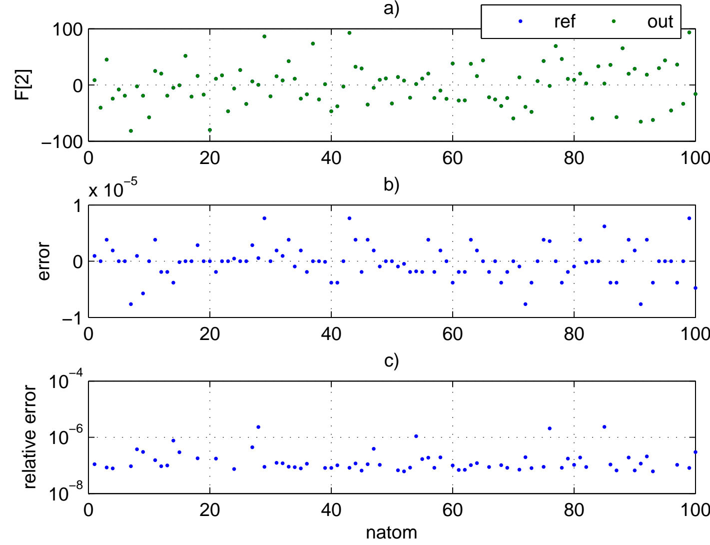

Figure 4.11: Verification results of the accumulation of **F** [2] for all 100 particles. a) SW and HW outputs, b) Error, c) Relative error.

4.1.3 LJEwDir Core

Simulation of the core requires that all data are passed through the IO Bus. Matlab generates the proper address for each parameter/value, and the finishes writing the CSR to starts computation. The testbench running in Modelsim reads these stimuli and applies them to the LJEwDir core. After the data streaming finishes, the testbench polls the CSR waiting for completion. Then it writes the results in a separate text file.

Figure 4.12 shows simulation results for a complete execution of the core. In this case, the relative error of **F** [3] is mainly found below 10^{-4}. The relative errors of the total potentials V^{LJ} and V^{d} are $5.60 * 10^{-6}$ and $1.17 * 10^{-4}$, respectively.

4.2 In-System hardware verification of the FPGA coprocessor

Not only functional simulations were run to verify the design. The use of in-system hardware verification methods was also necessary during the first attempts to integrate the FPGA coprocessor with the software application. The FPGA can be debugged via JTAG using the SignalTap II embedded logic analyzer and the In-system Memory Content Editor. The Quartus II software presents snapshots of waveforms obtained from the target signals. The in-system verification is very appropriate for this stage of the design, where the design is very dense and starts to depend on many things from the real world.

This verification method helped specifically to find that the system was not correctly performing DMA because the software in the host had an wrong DMA configuration. It

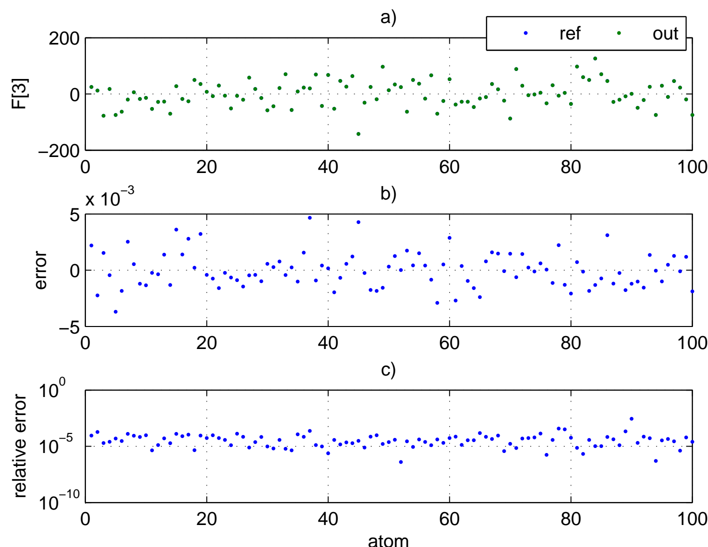

Figure 4.12: Verification results of **F** [3] for 100 particles in the core. a) SW and HW outputs, b) Error, c) Relative error.

also helped to find that the address of the particle-dependent parameters was not properly generated by the software. After correcting these two problems, the FPGA coprocessor was ready to compute non-bonded forces and potentials with the MD application.

4.3 Resource usage and Timing analysis

This section presents implementation results about synthesis and timing of the FPGA coprocessor on the Arria II GX EP2AGX125EF35C4 device.

Using the Altera's Quartus II 11.0, a normal compilation flow performs the following steps: Analysis & Synthesis, Fitter (Place & Route), Timing Analysis, and Assembler. Additionally, incremental compilations using LogicLock regions and Design Partitions were also performed to reduce compilation time by executing an extra step in the compilation flow called Partition Merge. This kind of compilation is especially useful when the whole system is completely routed and will be tested using an embedded logic analyzer like the SignalTap II, because it keeps untouched the final routing of design without affecting the timing, and also because it reduces substantially the total compilation time.

Following with the design flow, units were separately compiled when they passed their functional verification. At the same time, a parametrical analysis was performed to estimate resource usage and maximum operating frequency with respect to latency and other parameters, and also to explore advantages of other RTL descriptions. The objective was to facilitate optimization when the units are instantiated inside their parent entities.

4.3.1 Resource usage

Table 4.1 presents the resource usage of some units in the hierarchy according to compilation reports generated by the Quartus II software. It shows that the critical resource is the on-chip memory, where 76% of the total memory bits are in use. Currently, the three Coordinate Memories represent 46.76% of the available memory and 61.52% of the system memory. Purposed optimization of these memories is intended to save 50% memory bits per Coordinate Memory. With this change, the system uses only 52.62% of the available memory. However, it is probably not enough to duplicate the maximum number of particles supported.

Other resources in table 4.1 such as adaptive lookup tables (ALUTs), dedicated logic registers, and DSPs are not in critical state. The LJEwDir core currently uses 9.6% of ALUTs, 13.8% of registers, and 14.6% of DSPs, which suggests that there is still space for more Functional Pipelines, or for parts of it. The system also uses one of the four PLLs, 4 of the 12 GXBs, and uses the only one PCIe Hard-IP in the Arria II GX. Implementations in more powerful FPGAs like the Stratix IV GX in the DE4 Development Board offer a technological improvement that may lead to a faster system with more capabilities.

4.3.2 Timing analysis

The designed was properly constrained by declaring the penalty operating frequency, the setup/hold times and the uncertainty for all base clocks and known derived clocks, as well as by specifying the asynchronous clock domains and signals. This description of timing constrains is used during fitting to increase compilation effort with the purpose of meeting timing requirements. The timing analysis is run by the TimeQuest Timing Analyzer, which was also used to identify potential bottlenecks. [8]

All RAM and ROM memories have a read latency of two cycles with an operating frequency over 750 MHz. The Pair Controller showed a maximum operating frequency of 315 MHz. In general, most of the functional building units can run over 300 MHz. The three stages of the Datapath can operate at 300 MHz, 270 MHZ, and 230 MHz, respectively. The total latency of the Functional Pipeline is 83 cycles with a maximum operating frequency of 230 MHz for the internal clock and 320 MHz for the external clock. However, the whole system presents lower operating frequencies, with 137 MHz for the external clock[4] and 215 MHz for the internal clock.

[4]Note that this clock is now the Avalon clock and is driven by the PCI Express core (See figure 3.20 on page 42).

Table 4.1: Post-fitting results of the FPGA Coprocessor on the EP2AGX125EF35C4.

Entity	Comb. ALUTs	Logic Regs	Reg ALUT*	Mem. bits	M9Ks	9-bit DSPs	DSP 18x18	DSP 36x36
Available resources	99280	99280		6727680		576		
Percentage used	16%	23%		76%		15%		
LJEwDir_top	15524	22770	9662	5113134	706	84	12	15
altgx_reconfig	485	300	225	0	0	0	0	0
reconfig_pll	0	0	0	0	0	0	0	0
LJEwDir_sys	15036	22470	9437	5113134	706	84	12	15
pcie_hard_ip	2223	2335	1196	57104	26	0	0	0
pio_in	6	6	6	0	0	0	0	0
pio_out	5	4	4	0	0	0	0	0
descriptors_ram	0	0	0	8192	1	0	0	0
sgdma	804	1168	458	71844	11	0	0	0
LJEwDir_avaloncore	9515	13712	6376	4910458	660	84	12	15
LJEwDir_PairController	82	42	41	81	1	0	0	0
LJEwDir_Datapath	9296	13600	6288	4910377	659	84	12	15
ParticleMemory	208	0	6	3276800	400	0	0	0
TypeMemory	10	0	0	131072	16	0	0	0
CoordinateMemory	66	0	2	1048576	128	0	0	0
ForceMemory	2	2	0	1572864	192	0	0	0
ParameterMemory	0	0	0	8192	1	0	0	0
FPSingleAdder	510	478	281	0	0	0	0	0
FPSingleMult	119	132	93	0	0	4	0	1
FPSingleAcc	908	1664	630	6	1	0	0	0
fix_acc	140	518	131	0	0	0	0	0
single2fix_sig_w128_f64	249	203	171	0	0	0	0	0
fix2single_sig_w128_f64	487	933	336	6	1	0	0	0
MinimumImageConvention	241	202	167	0	0	0	0	0
SquaredDistance	346	695	247	0	0	16	0	4
fix2single_uns_w64_f48	209	354	149	0	0	0	0	0
InterpolationInput	27	25	24	0	0	0	0	0
InterpolationEngine	209	221	133	7071	8	6	3	0
InterpolationMAD	28	37	37	1584	1	2	1	0
CkRom	0	0	0	1584	0	0	0	0
fix2single_sig_w18_f15	91	105	56	19	2	0	0	0
ExpRom	0	0	0	616	0	0	0	0
fix2single_sig_w32_f24	154	163	68	3	1	0	0	0
single2fix_sig_w33_f24	123	133	109	0	0	0	0	0

* Combinational with a register ALUT/register pair

5. Software Application for Molecular Dynamics Simulations

In order to validate performance and accuracy of the FPGA coprocessor, a C-application to run simple MD simulations was also developed during this work. This software application implements the basic MD algorithm and uses a reduced force field. This force field performs direct evaluation of the Lennard-Jones 6-12 potential for van der Waals interactions, and uses the Ewald summation method to compute the electrostatic interactions in a periodic system. This chapter presents more details of this application, and also presents accuracy and performance analysis of the application when it is assisted by the FPGA coprocessor.

5.1 Description of the software application

5.1.1 Molecular Dynamics algorithm

The C-code developed for the MD algorithm is based on the code found in [41] that is used for visualization and parallelization of MD simulations. This code was chosen for this work because it implements part of the functions that are computed in hardware, and also because of its simplicity and programming language. The code has been strongly modified for better processing by creating new data structures and optimizing some operations. Moreover, it has been adapted to run with the FPGA coprocessor, which required the development of a Linux driver for the PCIe bus and for DMA. This driver was written using the Jungo WinDriver library. Algorithm 1 presents an overview of the MD application.

The application starts receiving simulation options from command console (see section 5.1.3). The TPF file is the topology file that contains the values of mass, sigma, epsilon and charge of each particle type. The PDB file contains the configuration of the set of particles, and specifies the number of particles, their positions (and possibly initial velocities), and the name of the particle type. Next, it computes some derived parameters such as α and uses the Lorentz-Berthelot mixing rule (see section 2.2.1.1) for ϵ_{ij} and σ_{ij}. Then, it reassigns initial velocities using a Maxwell-Boltzmann distribution[1] according to the initial temperature T_0.

[1] All simulations run with the same seed for the sake of simplicity in the comparisons.

Algorithm 1 Overview of the MD algorithm considering the FPGA coprocessor.

```
1  function mdsim()
2     (dt, rcut, L, T0, nsteps, ...) := assignOptions();
3     (sigma, epsilon, charge, mass) := readTpfFile();
4     (x, v) := readPdbFile();
5     (alpha, ...) := computeParameters();
6     v := v + initialVelocities(T0);
7     (x, v) := removeCenterOfMassAndRotation(x, v);
8     initializeHW();
9     toFPGA(sigma, epsilon, charge, alpha, type, L, rcut);
10    Vo := computeSelfEwaldPotential();
11    kvec := construct_mVectors();
12    computeEnergiesAndForces();
13    for n := 1 to nsteps do
14       integrateVelocityVerlet();
15       computeProperties();
16    end for
17    closeHW();
18    summary();
19 end function
```

After that, it removes the center of mass and rotation of the system[2].

Initialization of the FPGA coprocessor involves the following actions: search and open the FPGA board, open DMA, create DMA descriptors and write them into the FPGA, and enable interruptions. After that, the coprocessor is ready to receive and process information. The simulation parameters are the first information that is downloaded into the FPGA.

If the hardware is successfully initialized, then the system continues computing the self Ewald potential V^o, and is followed by the construction of the m-vectors in reciprocal space, which are used to compute the reciprocal-space Ewald summation. These vectors are selected according to the value of the cutoff $\overline{r_{cut}}$ in reciprocal space.

Before entering in the integration loop, the total energy is computed and reported for the initial state of the particles. Integration is done using the Velocity-Verlet scheme presented in section 2.1.1 on page 6. After completing each integration step, physical properties of the materials under simulation and other derivative quantities are computed and reported. Examples of such variables are energies, volume, temperature, pressure, diffusion, elasticity, conductivity, hardness, and so on. Besides these values, the positions and velocities are written in text files for further processing and analysis of the MD simulation.

Once the integration loop is finished, the FPGA coprocessor is properly closed by disabling interruptions, releasing DMA resources, and closing the connection to the board. Finally, the application presents the summary of the simulation, plot results using GNU Plot and show

[2]This is a recommended practice in MD simulations.

atomic trajectories using VMD.

Algorithm 2 Integration method of Velocity-Störmer-Verlet.

```
1 function integrate_VelocityVerlet()
2     x := updatePositions(x, v, F);
3     v := updateVelocities(v, F);
4     (Vsr, Fsr) := computeShortRangeForce(x);
5     (Vlr, Flr) := computeLongRangeForce(x);
6     F := Fsr + Flr;
7     v := updateVelocities(v, F);
8 end function
```

Algorithm 2 presents the sequence of statements of the Velocity-Verlet integrator. The function *updatePositions()* moves the particles from their current position using the current velocity and resulting force of each particle. This function also applies the boundary conditions that, for this case, are periodic. The function *updateVelocities()* appears two times in the algorithm in order to save memory by reusing the force vector (see figure 2.1). The computation of the force and potential energy only considers non-bonded interactions, which are divided into short and long range. The long range is always computed by the software through the reciprocal-space Ewald summation, while the short range can be computed either by software or by hardware. The parallel execution of these interactions in this MD application is only available when the hardware is enabled[3].

5.1.2 Computation of non-bonded interactions

In this MD application, the user can configure if the short-range interactions are computed either by software or by hardware. Likewise, short- and long-range interactions can be computed either sequentially or in parallel. Computation of long-range interactions[4] is always done in software. Software computation of short-range interactions is presented in the second part of algorithm 3. It uses an $O\left(0.5\,N^2\right)$ algorithm to compute the Lennard-Jones 6-12 potential and force, as well as the direct-space Ewald summation. It basically computes the separation distance between one particle and the rest in the simulation box, and continues with the force computation of those particles if and only if the distance is less than the cutoff radius. On the other hand, hardware computation of short-range interactions carries out an $O\left(N\left(N-1\right)\right)$ algorithm using the method described in chapter 3.

There are two different ways to interact with the FPGA coprocessor, and both differ in the CPU usage. The first method (see first part of algorithm 3) uses the CPU to send data of

[3]Only-software force computations can be run in parallel as child processes using thread programming and multiprocessor parallelization. Nevertheless, this MD application does not make use of these techniques.

[4]Specifically the long-range part of electrostatic interactions.

Algorithm 3 Sequential compuation of the short-range potential and force with the $O\left(0.5N^2\right)$ algorithm in software.

```
1  function shortRangeForce_sequential()
2     if useHW then
3        toFPGA(x);
4        toFPGA(start_PairController);
5        while fromFPGA(completed) != true do
6           wait;
7        end while
8        (Vlj, Vd, Fsr) := fromFPGA(read_VF);
9     else
10       Vlj := 0.0; Vd := 0.0;
11       for i := 0 to NATOM-1 do
12          for j := i+1 to NATOM do
13             (r[1], r[2], r[3], r2) := mic(x[i], x[j], L)
14             if r2 < rcut2 then
15                Vlj := Vlj + computeUlj();
16                Flj := computeFlj();
17                Vd := Vd + computeUlj();
18                Fd := computeFd();
19                for d := 1 to 3 do
20                   Fsr[i][d] := Fsr[i][d] + r[d]*(Flj + Fd);
21                end for
22             end if
23          end for
24       end for
25    end if
26    Vsr := Vlj + Vd;
27 end function
```

each particle to the hardware. Then, it starts the Pair Controller and waits until completion by polling the CSR register. Finally, this method uses the CPU to send downstream read requests to the hardware in order to obtain forces and potentials.

The second method uses DMA and PCIe interruptions to reduce CPU overhead. An FSM in software runs in a separate thread for the management of interruptions generated by the SG-DMA and the Pair Controller. The flow in this FSM starts when the CPU starts transferring particle positions with a Host-to-Device SG-DMA transfer via PCIe. When the interruption by SG-DMA completion is generated, the FSM starts the Pair Controller. Once the Pair Controller has sent its interruption due to completion, the FSM starts the Device-to-Host SG-DMA transfer to store forces in main memory. When this transfer finishes and generates an interruption, the FSM reads the two potentials stored in hardware with simple downstream read requests.

5.1.3 Configuration options

The user can select among several options when the application starts from command console. These options are listed and described in table 5.1. The option *-argon* selects predefined simulation sets (see table 5.2) that were used as testbenches for validation. In these testbenches, the value of α is always given by $\alpha = \pi/r_{cut}$, and the default initial temperature T_0 is 120 K.

Table 5.1: Options of the MD application.

Option	Argument	Description	Default
-dt	[real]	Time step in seconds.	1 fs
-nstep	[integer]	Number of integration steps.	10
-tpf	[string]	Topology file (.tpf file)	argon.tpf
-pdb	[string]	Initial configuration (.pdb files).	orig100.pdb
-rcut	[real]	Cutoff radius in Å.	100
-T0	[real]	Initial temperature in Kelvin.	120
-fs	[integer]	Subsampling interval to write output files.	1
-argon	[integer 1:5]	Execute predefined simulations of Argon.	1
-plot		Execute GNUplot script after simulation to plot energies, temperature and energy drift.	
-vmd		Open VMD animation after simulation to observe trajectories.	
-printvel		Include velocities in the XYZ output file.	
-hw		Enable the use of the hardware coprocessor.	
-dma		Enable DMA transfers to/from the FPGA coprocessor.	
-par		Enable parallel computation of forces. This option is only available if the option -hw is present.	

Table 5.2: Preset values of some parameters for each testbench.

	Argument of option *-argon*				
	1	2	3	4	5
PDB	argon100	argon800	argon2700	argon6400	argon12500
N	100	800	2700	6400	12500
L	18.0	36.0	72.0	90.0	108.0
r_{cut}	9.0	18.0	36.0	45.0	54.0
δt	1e-15	0.5e-15	0.5e-15	0.1e-15	0.1e-15
$nsteps$	10000	1000	100	100	10

5.2 Accuracy and Speed up

The following results presented in this section correspond to the final implementation of the FPGA coprocessor on the Altera's Arria II GX development kit, using the EP2AGX125EF35C4 device (see appendix C). The board was inserted in one of the PCIe slots of a Dell Optiplex 780 (Intel Core i7 @ 3.4 GHz, 16 GB RAM) running the custom C-application. Some simulations were also run on a Toshiba Satellite A305 (Intel Core2Duo T6400 @ 2.00 GHz, 3GB RAM). However, the Toshiba only runs pure software simulations because the board cannot be inserted in that laptop. Both computers have the 64-bit Ubuntu 10.4 distribution as operating system.

The testcases are a small set of preconfigured simulations for 100, 800, 2700, 6400 and 12500 Argon atoms distributed in cubic boxes of lattice 18.0, 36.0, 54.0, 72.0 and 90.0 Å, respectively. Figure 5.1 shows snapshots in VMD of each simulated set of particles. The list of elements is build from two different variants of Argon. Table 5.3 presents the parameters of the two variants of Argon simulated in the testcases.

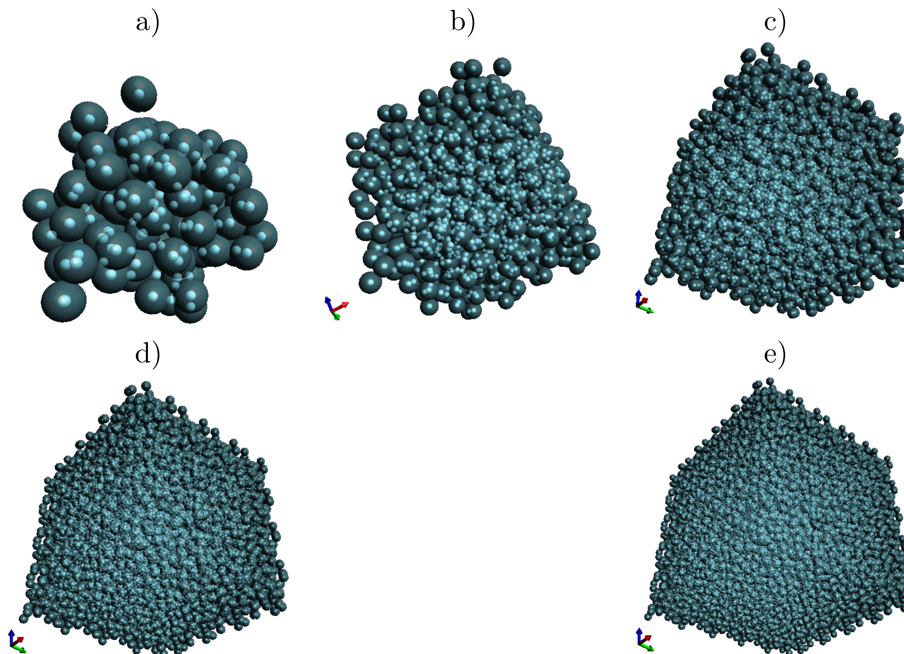

Figure 5.1: Snapshots of the testcases. a) 100 atoms b) 800 atoms c) 2700 atoms d) 6400 atoms e) 12500 atoms. This images were generated with VMD [23].

Table 5.3: Parameter values of the elements in the topology file for the simulation.

	Argon 1	Argon 2	Unit
ϵ_{ii}	0.9980	0.9980	kJ/mol
σ_{ii}	3.4050	3.4050	Å
q_i	+3.7274e-4	-3.7274e-4	$\sqrt{kJ/mol * m}$
m_i	39.9480	39.9480	a.m.u.

5.2.1 Accuracy

There are two quantities that are commonly used to measure the accuracy of an MD simulation. The first one is the Mean Squared Fluctuation (MSF) or RMS fluctuation that gives time-average information about the position of a particle. Equation 5.1 describes the MSF, where T is the integration time, $x_i\left(t_j\right)$ is the position of particle i at time t_j, and $\tilde{x}_i$ is a reference position that is normally the mean position $\overline{x}_i$. Well-conditioned MD simulations have an MSF below 10^{-5}.

$$MSF = \frac{1}{T}\sum_{t_j=1}^{T}\left(x_i\left(t_j\right) - \tilde{x}_i\right)^2 \tag{5.1}$$

The second quantity is the energy drift. According to the principle of energy conservation, the total energy of a system should be constant. Nevertheless, it is not fulfilled due to errors coming from rounding and numerical integration. The energy drift indicates the relative change of the total energy, which should be zero. The energy drift is calculated using (5.2).

$$drift = \frac{1}{T}\sum_{n=1}^{T}\left|\frac{E^n - E^{n-1}}{E^n}\right| \tag{5.2}$$

This work uses the energy drift as a measure of simulation accuracy. The testbenches were run using double- and single-precision data. Differences between the single-precision and the double-precision simulations resulted acceptable, since both show trajectories with similar energy drifts and with similar energy profiles. In the same way, the trajectories using the FPGA coprocessor resulted comparable to those using double precision in software.

Figure 5.2 shows the kinetic, potential and total energy for a simulation considering 100 atoms during 1000 steps with an integration step of 1 fs. Figure 5.3 shows the energy drift of a simulation that computes the short-range forces in software using double precision, and the energy drift of the same simulation using the FPGA coprocessor. Both simulations present similar energy drifts with values below 10^{-4}. All simulations with the FPGA coprocessor keep stability even for long simulations[5]. Such stability together with the energy drift and

[5] The longest simulation run for 10000 steps.

the similar trajectories validate the accuracy of the coprocessor.

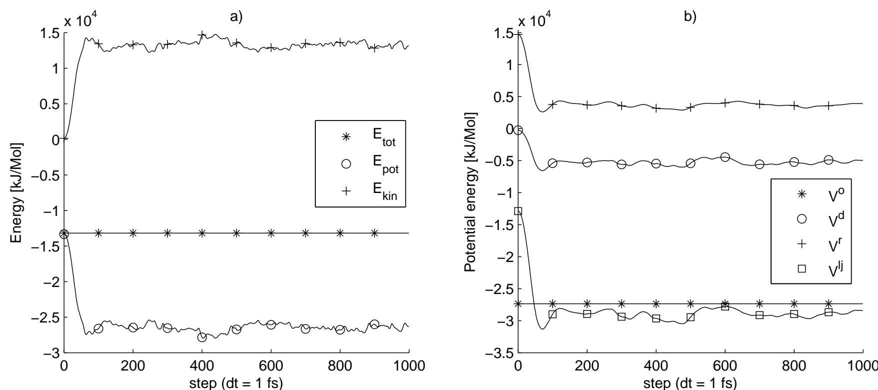

Figure 5.2: Energy profile for a simulation of 100 Argon atoms during 1000 steps at 1 fs/step. a) Kinetic, potential and total energy, b) Potential energies.

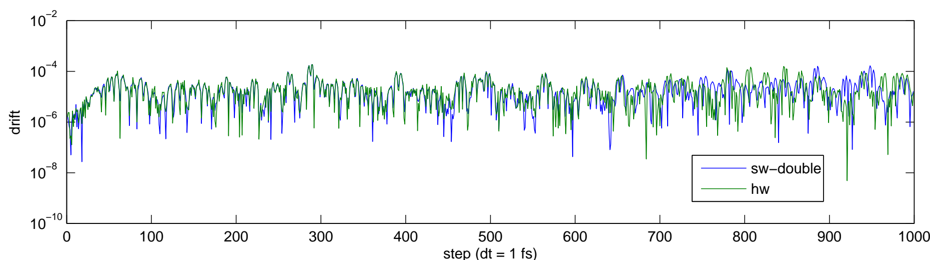

Figure 5.3: Energy drift for a simulation of 100 Argon atoms for 1000 steps at 1 fs/step. a) Only software in double precision, b) With hardware.

5.2.2 Speedup

The main purpose of this work is the acceleration of MD simulations; however, acceleration of this C-application in particular is open for discussion, since this application is neither fully optimized nor parallelized in software. Nevertheless, this platform can be used for more than accuracy analysis, and can also be used for performance analysis.

Speedup was evaluated for two different kind of execution times: the total simulation time, and the average execution time of short-range interactions. The last one is expected to be directly accelerated by the FPGA coprocessor. The total simulation time, however, has an upper bound given by that part of the process that must be executed sequentially; no matter how short is the parallelizable part of the process. This upper bound is calculated using Amdahls's law, which is described by

$$S(A) = \frac{t_T}{T(A)} = \frac{t_T}{\beta \cdot t_T + \gamma \cdot t_T / A}, \tag{5.3}$$

where β is the portion of the total execution time t_T in the algorithm that must be sequential, and the remainder $\gamma = 1 - \beta$ can be perfectly parallelized and accelerated in a factor A. Therefore, the total execution time of the parallelized process is $T(A) = \beta \cdot t_T + \gamma \cdot t_T/A$. Thus, the maximum speedup occurs when $A \to \infty$, and is bounded by the portion β of the algorithm that cannot be parallelized[6].

Figures 5.4 and 5.5 show the total execution time and the average time in direct (t_d) and reciprocal (t_r) space for different configurations. Here, times are also presented for different options: *sw* refers to the Dell Optiplex 780, *sw2* refers to the Toshiba Satellite A305, *hw* refers to the FPGA coprocessor using CPU for transfers, *hw+dma* refers to the FPGA coprocessor with DMA and interruptions, and *par* refers to parallel execution[7].

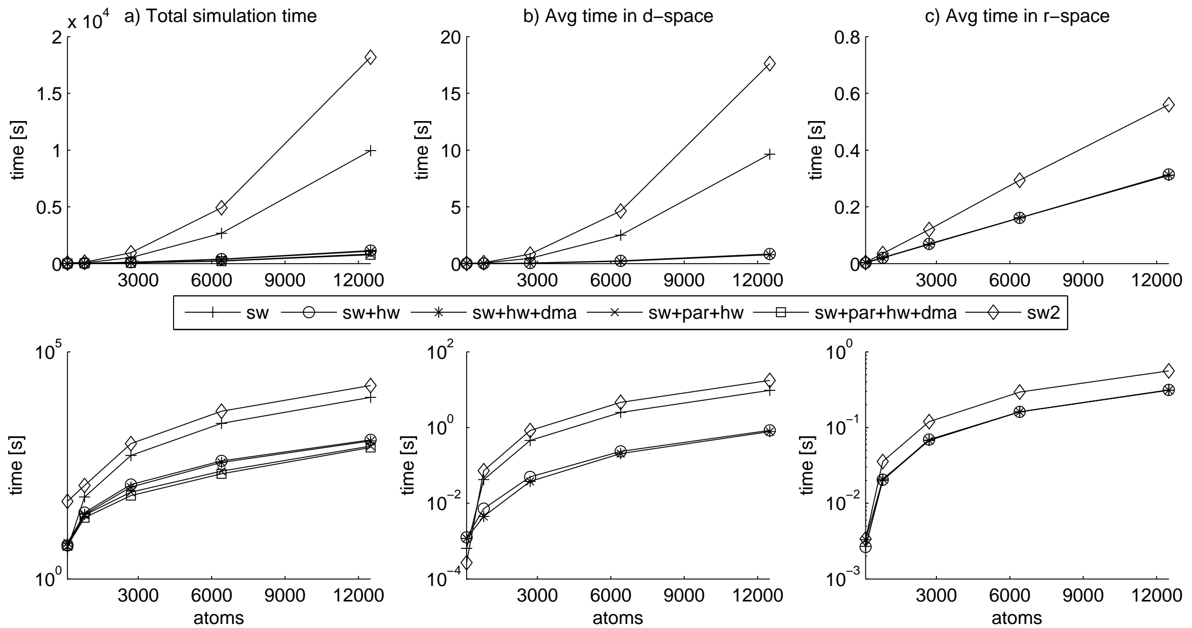

Figure 5.4: Simulation time with $r_{cut} = L/2$ for 100, 800, 2700, 6400 and 12500 particles. a) Total simulation time, c) Average time in direct space, c) Average time in reciprocal space. Lower plots show time in logarithmic scale.

Figure 5.4 shows the simulation times for different number of atoms using $r_{cut} = L/2$. Figure 5.5 shows how the simulation time of 12500 atoms is influenced by the cutoff radius. Note that the cutoff radius moves the computational load between the short-range and long-range interactions. This can be used to minimize total runtime by proper load balance.

In general, *sw2* is approx. 1.7 times slower than *sw*, and there is no remarkable difference

[6]This serious limitation has long been used as an argument against massively parallelization that may represent an extra effort. However, for large problems is the parallelization the only way to achieve successful simulations [17]. For instance, if only 1% of the algorithm cannot be parallelized, then only a speedup of 100x can be obtained.

[7]Detailed values of the simulation time for each configuration can be found in appendix B.

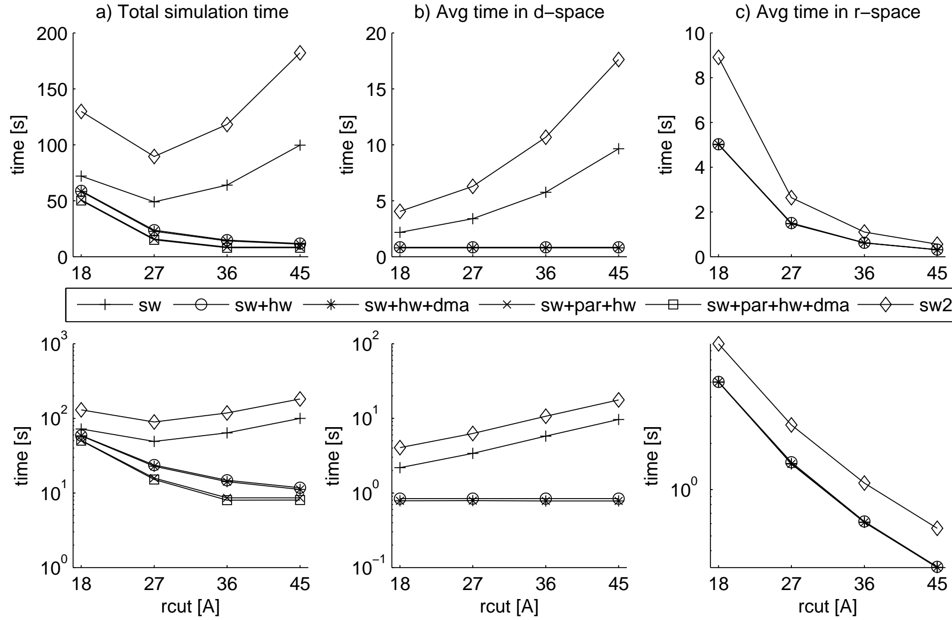

Figure 5.5: Simulation time varying r_{cut} between 18 Å and 45 Å for 12500 atoms. a) Total simulation time, c) Average time in direct space, c) Average time in reciprocal space. Lower plots show time in logarithmic scale.

between *hw* and *hw+dma*. Furthermore, *hw* takes the same time, no matter the value of r_{cut}, while the simulation time in reciprocal space shows a quadratic growth when r_{cut} decreases linearly. The maximum acceleration of the total runtime achieved with the FPGA coprocessor is 12.5x and that happened for a simulation considering 12500 atoms. Results show that the bigger the system, the higher the acceleration.

6. Comparison with Previous Works

This chapter presents previous works related to the implementation of MD algorithms in hardware, as well as some of the characteristics of this work that are comparable to other works.

6.1 Previous works

One of the most notable advances about the acceleration of MD simulations is the successful architecture created by the company PetaChem [33, 38]. Here, simulations are run over an arrangement of four NVIDIA's GPGPUs. They have reported accelerations up to 650x, reducing so a simulation time from 3.4 hours to 19.1 seconds. So far, GPUs are the best solution due to their programmability, performance and relative low cost. Nevertheless, GPUs are specific-purpose multiprocessors for video processing that could still be optimized for MD simulations.

Example of such optimization is the successful special-purpose machine for MD called Anton [36], which is built from 512 identical MD-specific ASICs that interact with each other using a specialized high-speed communication network. Anton uses novel parallel algorithms and special-purpose logic, and runs millisecond-scale MD simulations. Comparison of execution time of a long-range time step for a 512-node Anton machine and a single Xeon processor running GROMACS shows an overall acceleration of 9000x with respect to the entire long-range time step, and 50000x over the calculation of range-limited forces. [24]

From this sight, the development of solutions based on high-performance reconfigurable architectures (e.g. FPGAs) could offer acceptable results, reducing the cost and improving the performance. In [1], biomolecular simulations are accelerated by using FPGAs by just using high-level programming languages. They reported a speedup of 3x over an Intel 2.8 GHz Xeon and ensure that the bigger the problem, the more the speedup, overcoming cluster-based supercomputing platforms. A state-of-art until 2008 of the acceleration of MD simulations with reconfigurable computers is presented in [19].

Since 2003, the University of Toronto has published several works about MD simulations on FPGAs. In [9] an MD simulator is completely implemented on the Transmogrifier 3 (TM3),

a multi-FPGA system. The design is scalable and parallelizable. It performs calculations on an 8192 particle system in 37 s @ 26 MHz. This long processing time is mainly due to communication overhead, because the connection is done via RS-232.

A second work is [15], where two separate computational engines compute the Lennard-Jones 6-12 potential and the direct-space Ewald summation. The communication is also via RS-232. The design runs on a Virtex-II XC-2V2000 with computational engines running at approx. 80 MHz. One of the disadvantages of this design is that the interpolation method uses 1st-order polynomials, which demands high memory resources to store the polynomials for the fine partitioning of the interpolation range.

A third work is [26], which presents a computational engine to compute the reciprocal-space Ewald summation using the Smooth Particle Mesh Ewald (SPME) method. This method requires 3-D FFTs, as well as B-Spline interpolations to the grid points. The design runs at 40 MHz on a Xilinx Multimedia Board, and is integrated with NAMD2 to run simulations considering 66 particles.

Since 2005, the CAAD laboratory of the Boston University has published several works about MD acceleration using FPGAs. In [28], Discrete MD simulations are accelerated by using FPGAs. In [18], a novel interpolation method using the so-called semi-floating-point arithmetic[1] simplifies the interpolation of the Lennard-Jones 6-12 and short-range part of the Coulomb potential. The long-range part of the Coulomb potential is computed using the Multigrid method, which fits better on FPGAs than FFTs. This work implements the cell-list method for the short-range part of non-bonded interactions. It reports 5x and 10x accelerations of MD simulations, and can run large models of up to 256k particles using off-chip memory. The target hardware is a generic PC connected to a PCI plug-in board with two Xilinx VP70s. The target software is ProtoMol and NAMD.

The work in [37] presents the acceleration of molecular docking and binding using FPGAs and GPUs, with speedup between 6x and 42x depending of the algorithm and device. In [13, 12], there is a comparison between direct evaluation using floating-point IPs running at 250 MHz, and the interpolation method with table lookup, both for the computation of short-range pair forces and potentials. Also, it considers a smoothing function to reduce noise caused by the cutoff radius. Additionally in [11, 14], several methods for pair filtering are purposed to accelerate MD simulations based on neighbor lists. These methods are implemented as filter banks that queue particles to the force pipelines. The system is implemented on a Stratix III EP3SE260, and is tested with a NAMD benchmark containing 92224 particles, computing short-range forces in less than 22 ms.

[1]This concept was previously published in [20] by the same author, and was tested reporting a total speedup of 5.5x for a 8192-atom simulation with 26 atom types.

The number of atoms in that simulation was limited to 11200 because all storage was done on chip.

In [35], the Lennard-Jones 6-12 potential is directly evaluated by double-precision floating-point units in a deep functional pipeline with 119 stages running at 122 MHz . It reports an overall throughput of 3.9 GFLOPS with two parallel pipelines. However, this pipeline is integrated with no MD simulation, and only computes the scalar part of the pairwise force, not its components.

Finally, appendix A presents prior work to the one presented in this thesis. In that work, direct evaluation of the Lennard-Jones 6-12 force and potential is executed only by single-precision floating-point IPs distributed in a functional pipeline running at 300 MHz. Although the design of that system required shorter time and showed high throughput, it is still limited by conventional operators (e.g. addition, multiplication, division, inverse, etc.). The high cost in hardware resources of these blocks motivated to a migration towards a hybrid arithmetic that uses the best of the floating- and fixed-point arithmetic in hardware architectures.

6.2 Characteristics of this work

This work presents an FPGA coprocessor that computes the potential and forces of the Lennard-Jones 6-12 potential and the direct-space Ewald summation. The coprocessor was implemented on an Arria II GX Development Board, and is connected via PCI Express to a custom application for MD simulations. It takes less than 800 ms to compute the short-range part of the non-bonded interactions in a system with 12500 particles, achieving a maximum speedup of 12.5x over the single-core of an Intel Core i7 @ 3.4 GHz. The system processes up to 16 particle types and up to 16538 particles. The number of particles is limited by on-chip memory.

This works introduces and implements a novel variable transformation of the potential energy functions. This transformation is supported by a novel interpolation method that implements pseudo-floating-point arithmetic, and that uses scaling/centering transformation to improve polynomial fitting. It implements a single force/potential pipeline that runs at 200 MHz, and that uses floating-point and fixed-point arithmetic to obtain the best of both representations.

7. Conclusions and Future Work

7.1 Conclusions

The design, development, verification and validation of an FPGA coprocessor and its integration with an MD software has been presented in this thesis. This FPGA-based coprocessor has been designed to compute the short-range part of two non-bonded interactions modeled as the Lennard-Jones 6-12 potential and direct-space Ewald summation.

The RTL description of the core was written in VHDL and is highly parameterized. System-level design was done using the Qsys tool that comes with the Quartus II software. Analysis of the tradeoff between maximum operating frequency, latency, resource usage and accuracy, was based on resource usage analysis, timing analysis, and functional simulations. The design runs in an Arria II GX Development Board and fits in the EP2AGX125EF35C4, a high-end Altera's FPGA.

Several verification methods were used along the design to ensure its correct operation. VHDL testbenches in ModelSim-Altera run functional simulations of the RTL hardware descriptions. The Matlab's Fixed Point Toolbox was used to create fixed-point models of the system. Matlab was also used to create complex stimuli for the testbenches in Modelsim, and to carry out comparisons of results between software and hardware implementations. In-system hardware debugging using SignalTap II and the In-system Memory Content Editor were used for logic analyzing of the system when it was running in the FPGA.

The system implements interruption generation and a SG-DMA to reduce CPU overhead and to improve parallelization. The communication overhead has been reduced by using a PCI Express Gen1 x4 link between the host and the FPGA. This link under this configuration has a theoretical throughput of 850 MB/s, and makes use of the embedded PCIe Hard IP of the Arria II GX to save hardware resources.

The design has been successfully integrated within MD simulations. The coprocessor can process up to 16k particles that are stored in on-chip memory. It currently supports up to 16 elements, but can easily be configured to support more. The simulations with/without FPGA coprocessor considering from 100 until 12500 Argon atoms showed similar trajectories with energy drifts less than 10^{-5}, indicating that there is no problem about precision. Simulations

assisted by the FPGA can compute in parallel the long-range part of the non-bonded interactions in the host and the short-range part in the FPGA, achieving a maximum speedup of 12.5x in comparison to those simulations with a sequential computation of the forces that runs purely in software.

This work purposed and implemented a novel variable transformation of the potential and force functions that has still not been published in the literature. This transformation makes easier and more accurate the interpolation of those functions in hardware in comparison to previous reports. The interpolation method presented in this work uses pseudo-floating-point representation and has been carefully studied by considering the effect that the polynomial order, the partitioning of the range of interest, the rounding method, and the word length, has over accuracy and hardware resources. The force and potential computation has been mapped into a functional pipeline that runs at 200 MHz. Floating-point and fixed-point arithmetic were used along the functional pipeline to obtain the best of both representations.

7.2 Future work

This work has been finished leaving satisfactory results. However, there are still some optimizations and additional features that can be applied to the current design. The following list presents some of them:

1. Reduce the size of the replicated Particle Memory by reading it using a clock two-times faster than the so-called internal clock. This is possible because the synthesized memory reported a maximum operating frequency around 750 MHz, which is at least three times faster than the current operating frequency of the internal clock. This method also allows the use of more pipelines in parallel, since more pairs can be generated in the same cycle without increasing the size of the memory. However, more pipelines only reduce linearly a problem based on an $O\left(N^2\right)$ algorithm.

2. Use two or more external SRAM memories to increase the maximum number of particles supported.

3. In this moment, the pipeline is not 100% efficient for certain cutoff radii because some computed potentials and forces are discarded when the distance overcomes the cutoff radius. To solve this, a FIFO can be located between the Squared Distance unit and the multipliers for variable transformation. This is justified because the first stage of the Datapath (from Particle Memory to Squared Distance) can run faster than the further stages. This increases the probability that the slower stages are computing only the necessary pairs. This improvement requires a deep study about the probability distribution of the particles in order to not overdimension the FIFO.

4. Integrate the FPGA coprocessor with free software in the market such as GROMACS, LAMMPS or NAMD, in order to compare with software used in real-world scientific projects.

5. Adapt the system (especially the Pair Controller) to work with linked cells for simulations using the Linked-Cell method for the short-range part of non-bonded interactions. The use of this method reduces dramatically the computational complexity in software and in hardware. The reason is that the system does not have to look for pairs in the whole simulation box, but only in a smaller area defined by the cutoff radius.

6. Add a flag to indicate if the boundary condition is IBC or PBC. This prevents the MIC unit about performing the periodic refolding.[1]

7. If the software application does not require the separate value of each potential, then they can be summed up into a single register in hardware. It saves one of the expensive FPSingleAcc units but still requires an extra FPSingleAdder for reduction of potentials.

8. Add flags in the core's CSR to indicate possible error sources, e.g. overflow in the converters and accumulators. These flags can be used to break the current computation and generate interruptions to inform the host about such inconsistencies.

9. Optimize and parameterize the fix2single and single2fix units.

10. Migrate to a more powerful FPGA like the Stratix IV GX in the Altera's DE4 Development Board. The design in the Stratix IV GX can be forwarded to the HardCopy IV GX for power saving and for a possible enhancement of performance. This technological improvement can make the design run faster and give it more capacity, but much better are the solutions oriented to improve the algorithm and the architecture.

Beyond the scope of this thesis, there are still many more things to do in this field, and they can now be considered thanks to the experience and knowledge gained during this work.

[1] Keep in mind that the Ewald summation is only for periodic systems.

A. Lennard-Jones 6-12 Engine with Floating-Point Arithmetic

This chapter presents prior work to the final design presented in this document. This first attempt was inspired in the fact that Altera's floating-point IPs were presented in [6, 32, 30] as very suitable devices for High-Performance Reconfigurable Computing. Furthermore, a previous work about the floating-point implementation of a Lennard-Jones 6-12 engine was presented in [35]. Part of the knowledge collected in this experience was used to better understand pipelined architectures, how much area is demanded by the floating-point units, and how efficient these units result.

A.1 Implementation with floating-point units

The following sections present the implementation of two cores to compute the Lennard-Jones 6-12 potential and forces for a single pair of particles using only floating-point arithmetic. Both cores basically differ in how the potential

$$U_{ij}^{LJ} = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right]$$

and the dth component of the force

$$\mathbf{F}_{ij}^{LJ}[d] = 24\epsilon_{ij}\left[2\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right]\frac{1}{r_{ij}^{2}}\mathbf{r}_{ij}[d]$$

are factorized in order to save resources by reusing intermediate variables.

For a 3-D simulation box without applying the minimum image convention, the squared distance is calculated as the Euclidian distance given by $r_{ij}^2 = \sum_{d=1,2,3}(\mathbf{x}_i[d] - \mathbf{x}_j[d])^2$. Figure A.1 shows the block diagram of the Calculate Distance block based on the floating-point IPs for squaring and addition/subtraction. This unit provides the squared magnitude r_{ij}^2 and the components $\mathbf{r}_{ij}[d]$ of the distance vector. Its inputs are the positions $\mathbf{x}_i$ and $\mathbf{x}_j$ of the current

pair of particles described by their components. This figure also shows the possible range of latencies (in clock cycles) in that every IP can be parameterized, as well as the resulting range of the total latency. Black rectangles represent synchronization registers.

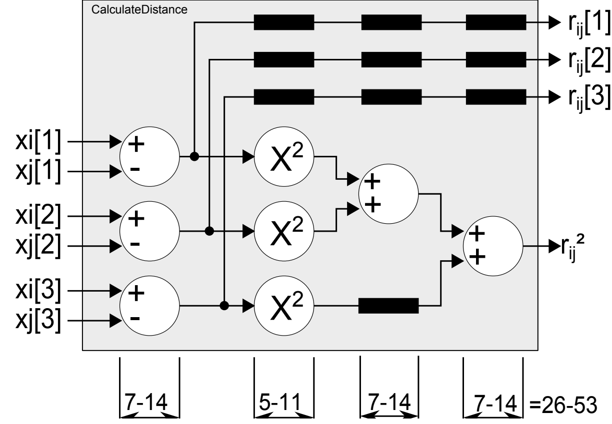

Figure A.1: Block diagram of the CalculateDistance unit.

A.1.1 FP_LJ Model 1

The FP_LJ Model 1 reorganizes the potential and force functions in terms of powers of r^{-1} as

$$\frac{1}{4}U_{ij}^{LJ} = \left(\epsilon_{ij}\sigma_{ij}^{12}\right)\left(r_{ij}^{-12}\right) - \left(\epsilon_{ij}\sigma_{ij}^{6}\right)\left(r_{ij}^{-6}\right)$$

and

$$\frac{1}{24}\mathbf{F}_{ij}^{LJ}[d] = \left(\left(2\epsilon_{ij}\sigma_{ij}^{12}\right)\left(r_{ij}^{-14}\right) - \left(\epsilon_{ij}\sigma_{ij}^{6}\right)\left(r_{ij}^{-8}\right)\right)\mathbf{r}_{ij}[d].$$

To compute them, this model requires three constants that depend on the particle type: $\epsilon_{ij}\sigma_{ij}^{12}$, $2\epsilon_{ij}\sigma_{ij}^{12}$, and $\epsilon_{ij}\sigma_{ij}^{6}$. These constants are supposed to be already stored in RAM. The target is now to compute four powers of r^{-1} (r^{-6}, r^{-8}, r^{-12} and r^{-14}) by direct evaluation from r^2, i.e. without interpolations.

Figure A.2 shows the block diagram of the FP_LJ1. The first stage computes the powers of r^{-1} starting with an inverse and followed by squaring and multiplications. The second stage computes in parallel the potential and the force using multiplications and additions. The total latency of this model is minimum 42 cycles and maximum 89 cycles.

A.1.2 FP_LJ Model 2

The FP_LJ Model 2 reorganizes the potential and force functions as

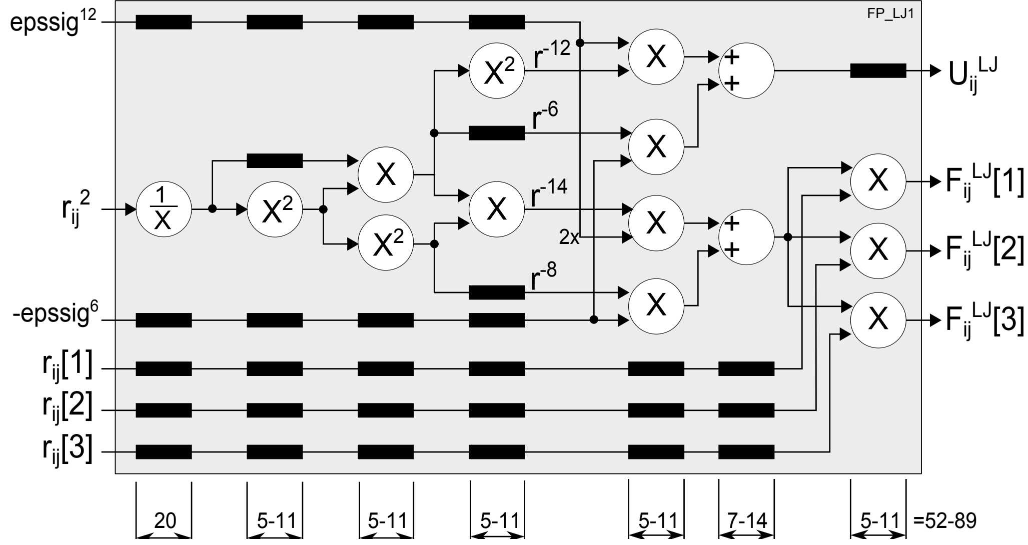

Figure A.2: Block diagram of the FP_LJ1 core.

$$\frac{1}{4}U_{ij}^{LJ} = (\epsilon_{ij}A_{ij})(A_{ij} - 1)$$

and

$$\frac{1}{48}\mathbf{F}_{ij}^{LJ}[d] = ((A_{ij}B_{ij})(A_{ij} - 0.5))\,\mathbf{r}_{ij}[d],$$

where $A_{ij} = \left(\sigma_{ij}^2/r_{ij}^2\right)^3$ and $B_{ij} = \left(\epsilon_{ij}/\sigma_{ij}^2\right)\left(\sigma_{ij}^2/r_{ij}^2\right)$. This time, the target is to compute A_{ij} and B_{ij}, which are half the number of target variables needed in FP_LJ1.

Figure A.3 shows the block diagram of the FP_LJ2. This block uses fewer units in the first stage than the FP_LJ1, and computes A_{ij} and B_{ij} starting with a divider and followed by multiplications and squaring operations. The second stage is similar in structure to FP_LJ1, but swapping a couple of adders with multipliers.

A.2 Comparison between the FP_LJ Model 1 and 2

Simulation results of these cores show that they have the same accuracy than single-precision results in software. Synthesis and timing results are presented in this section for the top level entity called LJ_Engine, which contains the following building units: CalculateDistance, GenerateRPowers, CalculatePotentialLJ and CalculateForceLJ. The last three units represent the FP_LJ1 and FP_LJ2 cores presented in the last sections. Additionally, four instances of the LJ_Engine were grouped in the LJ_Engine_x4 entity that includes 12 floating-point adders to reduce the resulting vector of potentials and forces.

Table A.1 presents compilation results for both LJ_Engine models that were implemented

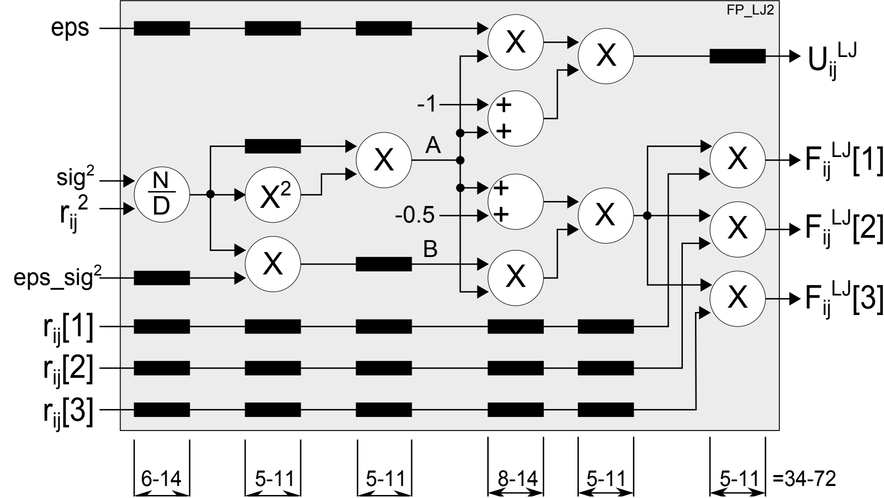

Figure A.3: Block diagram of the FP_LJ2 core.

on the Stratix III EP3SE50F780C2. The table presents the area as a function of the number of ALUTs, dedicated logic registers, block memory bits, and 18-bit DSPs. The maximum operating frequency of the clock is presented by the TimeQuest Timing Analyzer for the slow model of the FPGA at 85°C. Results for each unit of each model are presented with respect to the minimum, intermediate, and maximum allowed latency. The intermediate latency is not the optimal value, but offers a satisfactory tradeoff between area and throughput.

In general, floating-point additions/subtractions consume more ALUTs than multiplications, but multiplications require four DSP blocks. The division and the inverse are more expensive in hardware that additions and multiplications. Both use 16 DSPs, but the division uses approx. 5000 memory bits, while the inverse uses approx. 400 of them.

The inverse has a fixed latency of 20 cycles and is the highest in comparison to the others, which have from 5 to 14 cycles. When these floating-point IPs are compiled separately, their operating frequencies are between 142 MHz and 440 MHz.

Both FP_LJ1 and FP_LJ2 can operate over 250 MHz, but the FP_LJ2 demands less resources and has lower latency. This demonstrates that FP_LJ2 benefits from the reorganization of the force and potential functions. Using four of these pipelines in LJ_Engine_x4 reduces the operating frequency in approx. 20%, but increases throughput to more than 300%.

A.3 Conclusions

The direct evaluation of the Lennard-Jones 6-12 potential and force using floating-point units works satisfactorily in terms of speed and accuracy; however, the core presented in this chapter

does not solve the big problem. Moreover, this core includes neither the implementation of the direct-space Ewald summation, nor the accumulation, storage and management, and it still consumes high resources in comparison to the synthesis results presented in section 4.3.1 for the whole FPGA coprocessor.

This experience left the following advantages and disadvantages when working with Altera's floating-point IPs to compute the LJ potential:

Pros

- Short design time due to the easy mapping of classical arithmetic operators.[1]
- High precision.
- High throughput.
- Exception handling.

Cons

- Big size.
- High latency.
- Simple additions and subtractions become very hardware consuming. The same happens for simple logical operations.
- It does not allowed the implementation of functions like $erfc\,(x)$.

[1] These models were designed and tested in just a couple of weeks.

Table A.1: Compilation results for the FP_LJ1 and FP_LJ2 cores on the Stratix III EP3SE50F780C2.

Unit	Model	Latency	ALUTs (38000)	Ded. registers (38000)	Mem. bits (5455872)	DSPs (384)	Freq. @ 85°C
CalculateDistance	1, 2	26	3094	1756	526	12	202.63
		42	2905	3238	1172		260.28
		53	3311	3317	2630		327.65
GenerateRPowers	1	35	1011	1050	860	36	248.69
		38	1031	1061	1022		306.28
		53	1082	1618	2121		375.09
	2	16	626	640	5021	28	147.47
		24	733	685	5768		324.68
		36	796	979	6526		326.80
CalculatePotentialLJ	1	12	823	632	0	8	210.57
		18	772	912	144		325.20
		25	919	1069	636		414.25
	2	13	778	678	96	8	250.75
		18	796	833	242		298.42
		25	881	1049	605		386.25
CalculateForceLJ	1	17	1156	1055	0	20	214.73
		24	1125	1335	189		320.72
		36	1392	1740	1158		363.24
	2	18	1130	1102	96	20	245.52
		24	1167	1228	323		303.40
		36	1358	1705	1149		337.04
LJ_Engine	1	78	6075	4252	15486	76	207.56
		104	5725	6362	20917		290.19
		142	6666	7459	31467		315.96
	2	60	5498	3933	13277	68	147.67
		92	5366	5812	19227		292.48
		125	6159	6841	26590		294.38
LJ_Engine x4	1	92	28177	19514	62389	304	178.95
		124	27056	28430	84606		235.90
		170	30233	32777	127265		271.67
	2	74	26078	18420	53481	272	143.25
		112	25640	26972	77914		215.10
		153	28519	29850	108887		264.14

B. Tables of Execution Time

Tables B.1, B.2, B.3, B.4 and B.5 contain execution times of the simulation for 100, 800, 2700, 6400 and 12500 atoms respectively. The total simulation time t_T also includes tasks such as file I/O, computation of properties, initialization, etc. The average execution times avg-t_d and avg-t_r are calculated from the total execution time total-t_d and total-t_r spent to compute all forces and potentials in the direct and reciprocal space, respectively, divided by the number of steps to simulate. Each table specifies the number of atoms N, the size of the cubic simulation box L, and the number of integration steps *nsteps*. Different cutoff radii have been selected in the range $[2.5\sigma_{max},\, L/2]$, where $\sigma_{max} = 3.4050\,\text{Å}$.

The notation *sw1* represents the Dell Optiplex 768 (Intel Core i7 3.4 GHz, 16 GB RAM), while *sw2* states for the Toshiba Satellite A-305 (Intel Core2 Duo 2.0 GHz, 3 GB RAM). Only *sw1* is assisted by the FPGA coprocessor (*hw*). *dma* represents the DMA capability with interruption handling. *par* indicates simulations that compute in parallel the short- and long-range part of the non-bonded interactions.

All speedup factors *SU* are calculated with respect to times in *sw1*.

Table B.1: Execution time with $N = 100$, $L = 18.0\,\text{Å}$, $nsteps = 1000$.

r_{cut}	Mode	t_T	total-t_d	total-t_r	avg-t_d	avg-t_r	$SU\ (t_T)$	$SU\ (t_d)$
9.0	sw1	34.70	6.40	27.00	0.00064	0.00270	1.0000	1.0000
	sw1+hw	40.56	12.60	26.40	0.00126	0.00264	0.8555	0.5079
	sw1+hw+dma	46.63	11.80	33.10	0.00118	0.00331	0.7442	0.5424
	sw1+par+hw	39.77	38.40		0.00384		0.8725	
	sw1+par+hw+dma	38.79	37.10		0.00371		0.8946	
	sw2	83.77	2.65	33.86	0.00027	0.00339	0.4142	2.4151

Table B.2: Execution time for different cutoff radii with $N = 800$, $L = 36.0\,\text{Å}$, $nsteps = 1000$.

r_{cut}	Mode	t_T	total-t_d	total-t_r	avg-t_d	avg-t_r	$SU\ (t_T)$	$SU\ (t_d)$
18.0	sw1	63.60	42.19	20.11	0.04219	0.02011	1.0000	1.0000
	sw1+hw	28.87	7.21	20.46	0.00721	0.02046	2.2030	5.8516
	sw1+hw+dma	26.55	4.49	20.86	0.00449	0.02086	2.3955	9.3964
	sw1+par+hw	25.60	24.50		0.02450		2.4844	
	sw1+par+hw+dma	22.33	21.16		0.02116		2.8482	
	sw2	114.19	72.70	35.62	0.07270	0.03562	0.5570	0.5803
9.0	sw1	177.62	13.74	162.59	0.01374	0.16259	1.0000	1.0000
	sw1+hw	172.38	6.98	164.01	0.00698	0.16401	1.0304	1.9685
	sw1+hw+dma	169.98	7.48	161.28	0.00748	0.16128	1.0449	1.8369
	sw1+par+hw	168.85	167.66		0.16766		1.0519	
	sw1+par+hw+dma	165.91	164.46		0.16446		1.0706	
	sw2	316.55	20.51	289.48	0.02051	0.28948	0.5611	0.6699

Table B.3: Execution time for different cutoff radii with $N = 2700$, $L = 54.0\,\text{Å}$, $nsteps = 100$.

r_{cut}	Mode	t_T	total-t_d	total-t_r	avg-t_d	avg-t_r	$SU\ (t_T)$	$SU\ (t_d)$
27.0	sw1	52.67	45.40	6.81	0.45400	0.06810	1.0000	1.0000
	sw1+hw	12.30	4.96	6.94	0.04960	0.06940	4.2821	9.1532
	sw1+hw+dma	10.89	3.78	6.75	0.03780	0.06750	4.8365	12.0106
	sw1+par+hw	8.52	8.09		0.08090		6.1819	
	sw1+par+hw+dma	7.26	6.79		0.06790		7.2548	
	sw2	96.73	83.64	11.97	0.83640	0.11970	0.5445	0.5428
18.0	sw1	43.23	20.09	22.88	0.20090	0.22880	1.0000	1.0000
	sw1+hw	28.29	4.94	22.93	0.04940	0.22930	1.5281	4.0668
	sw1+hw+dma	26.99	3.77	22.84	0.03770	0.22840	1.6017	5.3289
	sw1+par+hw	24.75	24.27		0.24270		1.7467	
	sw1+par+hw+dma	23.21	22.73		0.22730		1.8626	
	sw2	78.63	37.29	40.30	0.37290	0.40300	0.5498	0.5388
9.0	sw1	198.95	10.26	188.15	0.10260	1.88150	1.0000	1.0000
	sw2	350.06	17.92	331.00	0.17920	3.31000	0.5683	0.5725

Table B.4: Execution time for different cutoff radii with $N = 6400$, $L = 72.0$ Å, $nsteps = 100$.

r_{cut}	Mode	t_T	total-t_d	total-t_r	avg-t_d	avg-t_r	SU (t_T)	SU (t_d)
36.0	sw1	268.12	250.87	16.13	2.50870	0.16130	1.0000	1.0000
	sw1+hw	40.69	23.45	16.12	0.23450	0.16120	6.5893	10.6981
	sw1+hw+dma	37.73	20.48	16.13	0.20480	0.16130	7.1063	12.2495
	sw1+par+hw	24.52	23.46		0.23460		10.9347	
	sw1+par+hw+dma	21.61	20.53		0.20530		12.4072	
	sw2	494.12	462.60	29.37	4.62600	0.29370	0.5426	0.5423
27.0	sw1	174.32	133.42	39.79	1.33420	0.39790	1.0000	1.0000
	sw1+hw	64.29	23.48	39.72	0.23480	0.39720	2.7115	5.6823
	sw1+hw+dma	61.51	20.73	39.73	0.20730	0.39730	2.8340	6.4361
	sw1+par+hw	43.71	42.65		0.42650		3.9881	
	sw1+par+hw+dma	40.74	39.67		0.39670		4.2788	
	sw2	319.63	244.75	72.76	2.44750	0.72760	0.5454	0.5451
18.0	sw1	203.31	71.50	130.75	0.71500	1.30750	1.0000	1.0000
	sw1+hw	155.19	23.43	130.62	0.23430	1.30620	1.3101	3.0516
	sw1+hw+dma	152.05	20.48	130.44	0.20480	1.30440	1.3371	3.4912
	sw1+par+hw	134.82	133.76		1.33760		1.5080	
	sw1+par+hw+dma	131.47	130.34		1.30340		1.5464	
	sw2	372.43	130.77	239.49	1.30770	2.39490	0.5459	0.5468

Table B.5: Execution time for different cutoff radii with $N = 12500$, $L = 90.0$ Å, $nsteps = 10$.

r_{cut}	Mode	t_T	total-t_d	total-t_r	avg-t_d	avg-t_r	SU (t_T)	SU (t_d)
45.0	sw1	99.71	96.40	3.11	9.64000	0.31100	1.0000	1.0000
	sw1+hw	11.75	8.38	3.14	0.83800	0.31400	8.4860	11.5036
	sw1+hw+dma	11.15	7.80	3.12	0.78000	0.31200	8.9426	12.3590
	sw1+par+hw	8.59	8.38		0.83800		11.6077	
	sw1+par+hw+dma	8.02	7.81		0.78100		12.4327	
	sw2	182.11	176.20	5.60	17.62000	0.56000	0.5475	0.5471
36.0	sw1	63.96	57.60	6.15	5.76000	0.61500	1.0000	1.0000
	sw1+hw	14.81	8.38	6.20	0.83800	0.62000	4.3187	6.8735
	sw1+hw+dma	14.14	7.82	6.12	0.78200	0.61200	4.5233	7.3657
	sw1+par+hw	8.59	8.40		0.84000		7.4459	
	sw1+par+hw+dma	7.99	7.80		0.78000		8.0050	
	sw2	118.12	106.73	11.02	10.67300	1.10200	0.5415	0.5397
27.0	sw1	48.95	33.90	14.81	3.39000	1.48100	1.0000	1.0000
	sw1+hw	23.67	8.39	15.06	0.83900	1.50600	2.0680	4.0405
	sw1+hw+dma	22.77	7.87	14.69	0.78700	1.46900	2.1498	4.3075
	sw1+par+hw	15.81	15.61		1.56100		3.0961	
	sw1+par+hw+dma	15.12	14.89		1.48900		3.2374	
	sw2	89.53	62.79	26.37	6.27900	2.63700	0.5467	0.5399
18.0	sw1	72.04	21.79	50.02	2.17900	5.00200	1.0000	1.0000
	sw1+hw	58.82	8.39	50.23	0.83900	5.02300	1.2248	2.5971
	sw1+hw+dma	58.29	7.85	50.23	0.78500	5.02300	1.2359	2.7758
	sw1+par+hw	50.72	50.50		5.05000		1.4203	
	sw1+par+hw+dma	50.15	49.92		4.99200		1.4365	
	sw2	129.82	40.43	89.03	4.04300	8.90300	0.5549	0.5390

C. Arria II GX Development Board

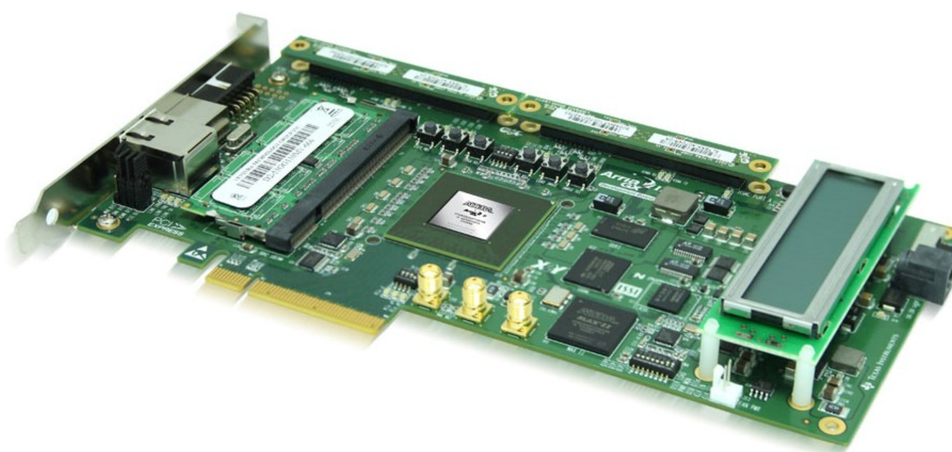

Figure C.1: Arria II GX development board. [2]

Table C.1: Characteristics of the EP2AGX125EF35 FPGA.

Characteristic	Value
Package	1152-pin fine pitch BGA
Logic elements (LEs)	124100
Adaptive logic modules (ALMs)	49640
On-chip memory	8121 kb
High-speed transceivers	12
Phase-locked loops (PLLs)	6
18x18 DSP blocks	288
Core power	0.9V

Table C.2: Characteristics of the Arria II GX board.

FPGA	EP2AGX125EF35
On-board ports	1 Gigabit Ethernet ports
	1 HSMC expansion ports
On-board memory	128-MB 16-bit DDR3
	1-GB 64-bit DDR2 SODIMM
	2-MB SSRAM
	64-MB flash
On-board clocking circuitry	4 On-board oscilators
	100 MHz
	155.52 MHz
	Programmable oscillator (100 MHz)
	Programmable oscillator (125 MHz)
Mechanical	SMA connector for external LVPECL clock input
	SMA connector for clock output
	PCI Express full-length standard-height
	PCI Express chassis or bench-top operation

Bibliography

[1] S.R. Alam, P.K. Agarwal, M.C. Smith, J.S. Vetter, and D. Caliga. Using FPGA Devices to Accelerate Biomolecular Simulations. *Computer*, 40(3):66 –73, march 2007.

[2] Altera. *Arria II GX Development Board - User's Guide.* Altera Corporation, 2009.

[3] Altera. Embedded Peripherals IP - User Guide. Technical report, Altera Corporation, 2010.

[4] Altera. Avalon Interface Specifications. Technical report, Altera Corporation, 2011.

[5] Altera. Design Planning with the Quartus II Software. Technical report, Altera Corporation, 2011.

[6] Altera. Floating-Point Megafunctions User Guide. Technical report, Altera Corporation, January 2011.

[7] Altera. IP Compiler for PCI Express User Guide. Technical report, Altera Corporation, May 2011.

[8] Altera. Section II Timing Analysis Guide. Technical report, Altera Corporation, 2011.

[9] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow. Reconfigurable Molecular Dynamics Simulator. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 197 – 206, april 2004.

[10] David Robert Bates. *Advances in Atomic and Molecular Physics*, volume 2. Elsevier.

[11] M. Chiu and M.C. Herbordt. Efficient particle-pair filtering for acceleration of molecular dynamics simulation. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 345 –352, 31 2009-sept. 2 2009.

[12] M. Chiu and M.C. Herbordt. Towards production FPGA-accelerated molecular dynamics: Progress and challenges. In *High-Performance Reconfigurable Computing Technology*

and Applications (HPRCTA), 2010 Fourth International Workshop on, pages 1 –8, nov. 2010.

[13] Matt Chiu, , Martin C. Herbordt, and Martin Langhammer. Performance potential of molecular dynamics simulations on high performance reconfigurable computing systems. In *High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on*, pages 1 –10, nov. 2008.

[14] Matt Chiu and Martin C. Herbordt. Molecular Dynamics Simulations on High-Performance Reconfigurable Computing Systems. *ACMTransactions on Reconfigurable Technology and Systems*, 3(4), November 2010.

[15] David Chui. An FPGA Implementation of the Ewald Direct Space and Lennard-Jones Compute Engines. Master's thesis, University of Toronto, 2005.

[16] P. Freddolino, A. Arkhipov, S. Larson, A. McPherson, and K. Schulten. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure, , .*, 14:437–449, 2006.

[17] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications (Texts in Computational Science and Engineering)*. Springer, 2010.

[18] Yongfeng Gu. *FPGA Acceleration of Molecular Dynamics Simulations*. PhD thesis, Boston University, 2008.

[19] Yongfeng Gu, T. VanCourt, and M.C. Herbordt. Accelerating Molecular Dynamics Simulations with Configurable Circuits. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 475 – 480, aug. 2005.

[20] Yongfeng Gu, T. VanCourt, and M.C. Herbordt. Integrating FPGA Acceleration into the Protomol Molecular Dynamics Code: Preliminary Report. In *Field-Programmable Custom Computing Machines, 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 315 –316, april 2006.

[21] Martin C. Herbordt, Yongfeng Gu, Tom VanCourt, Josh Model, Bharat Sukhwani, and Matt Chiu. Computing Models for FPGA-Based Accelerators. *Computing in Science & Engineering*, 10(6):35–45, November 2008.

[22] Konrad Hinsen. The Molecular Modeling Toolkit: A new approach to molecular simulations. *Journal of Computational Chemistry*, 21:79–85, 2000.

[23] William Humphrey, Andrew Dalke, and Klaus Schulten. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics*, 14:33–38, 1996.

[24] J.S. Kuskin, C. Young, J.P. Grossman, B. Batson, M.M. Deneroff, R.O. Dror, and D.E. Shaw. Incorporating Flexibility in Anton, a Specialized Machine for Molecular Dynamics Simulation. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 343 –354, feb. 2008.

[25] Hark Lee and Wei Cai. Ewald Summation for Coulomb Interactions in a Periodic Supercell. Technical report, Department of Mechanical Engineering, Stanford University, 2009.

[26] Sam Lee. An FPGA Implementation of the Smooth Particle Mesh Ewald Reciprocal Sum Compute Engine (RSCE). Master's thesis, University of Toronto, 2005.

[27] Thierry Matthey, Trevor Cickovski, Scott Hampton, Alice Ko, Qun Ma, Matthew Nyerges, Troy Raeder, Thomas Slabach, and Jesús A. Izaguirre. ProtoMol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Trans. Math. Softw.*, 30(3):237–265, September 2004.

[28] J. Model. FPGA Acceleration of Discrete Molecular Dynamics Simulation. Master's thesis, Boston University, 2007.

[29] F. Naomi. Molecular Dynamics. In *ECE 697S: Topics in Computational Biology*, 2006.

[30] Michael Parker. How to achieve 1 trillion floating-point operations per second in an FPGA. `http://www.eetimes.com/design/programmable-logic/4207687/How-to-achieve-1-trillion-floating-point-operations-per-second-in-an-FPGA?pageNumber=0`.

[31] Michael Parker. High-Performance Floating-Point Implementation using FPGAs. 2009.

[32] Michael Parker. Taking Advantage of Advances in FPGA Floating-Point IP Cores - Altera White Paper. Technical report, Altera Corporation, 2009.

[33] PetaChem. http://www.petachem.com/performance.html.

[34] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant V. Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.

[35] Ronald Scrofano and Viktor K. Prasanna. Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware.

[36] D.E. Shaw. Anton, a special-purpose machine for molecular dynamics simulation. *Proceedings of the International Symposium on Computer Architecture*, pages 1–12, 2007.

[37] Bharat Sukhwani. *Accelerating Molecular Docking and Binding Site Mapping using FPGAs and GPUs*. PhD thesis, Boston University, 2011.

[38] I.S. Ufimtsev and T.J. Martínez. Graphical Processing Units for Quantum Chemistry. *Computing in Science & Engineering*, pages 26–34, 2008.

[39] D. van der Spoel, E. Lindahl, B. Hess, A. R. van Buuren, E. Apol, P. J. Meulenhoff, D. P. Tieleman, A. L. T. M. Sijbers, K. A. Feenstra, R. van Drunen, and H. J. C. Berendsen. Gromacs User Manual version 4.5.4. 2010.

[40] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E. Mark, and Herman J. Berendsen. GROMACS: fast, flexible, and free. *Journal of computational chemistry*, 26(16):1701–1718, December 2005.

[41] Lee Wonpyo. Visualization of Parallel Molecular Dynamics Simulations. Master's thesis, University of Houston.