5-2010

Acceleration of Biomolecular Simulations using FPGA-based Reconfigurable Computing

Ananth Nallamuthu
Clemson University, anallam@clemson.edu

Acceleration of Biomolecular Simulations using Reconfigurable Computing

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Ananth Nallamuthu
May 2010

Accepted by:
Dr.Melissa C. Smith, Committee Chair
Dr.Steven J. Stuart
Dr.Walter B. Ligon III

Abstract

A paradigm shift is occurring in the way compute-intensive scientific applications are developed. Thanks to advancements in commercially viable hybrid architectures for High-Performance Computing (HPC), the focus has shifted from improving performance by merely scaling algorithms on von Neumann computing nodes to fully exploiting additional computational capabilities provided by accelerators such as FPGAs (Field Programmable Gate Arrays) and GPGPUs (General Purpose Graphical Processing Units).

Computational chemists use Molecular Dynamics (MD) simulations like LAMMPS (Large Scale Atomic Molecular Massively Parallel Systems) and NAMD (NAnoscale Molecular Dynamics) to simulate biomolecular behaviour such as protein folding and small molecule docking to proteins. MD simulations are computationally complex n-body problems, which are time consuming to simulate in biologically relevant scales. Executing such simulations in best available HPC environments is critical for scientific advancements in the field. Thus, as HPC technology evolves, there is a need to update classical biomolecular simulation applications like LAMMPS to better suit the architecture. In this work we modify LAMMPS (a classical molecular dynamics simulation program developed for CPU-only clusters) to execute on a reconfigurable computer system, SRC-7 H MAP. The SRC-7 H MAP consists of two Altera FPGA logic chips interfaced to a dual-core Intel Xeon processor. Users can

benefit by offloading most compute-intensive tasks of the application to the FPGA logic. This work explores the challenges involved in effectively adapting a production level application code optimized for von Neumann architecture, to an FPGA-based hybrid architecture.

We have successfully accelerated the non-bonded force computations, the most compute-intensive module in LAMMPS for biomolecular simulations, by 5.0x over a single 3.0 GHz Xeon processor. This performance includes the data transfer overheads and function calling overheads. Further, using the accelerated non-bonded force computations function, we achieve an overall application speed-up of 2.0x to 2.4x

Dedication

I dedicate this work to my parents for their numerous sacrifices.

Acknowledgments

I sincerely thank my advisor Dr.Melissa C. Smith for her guidance, support and patience during this research. She has continuously encouraged me and contributed to my professional growth in many ways. I thank Dr.Walter B. Ligon III and Dr.Steven J. Stuart for being on my committee and reviewing my work. I thank NIH for financially supporting this work (grant R21GM083946).

I thank our collaborators, Dr.Scott Hampton and Dr.Pratul K. Agarwal of Oak Ridge National Laboratory for their support during this research and helping me understand many aspects of molecular dynamics simulations. My special thanks to Dr.David Pointer of SRC Computers Inc. for his technical support and interest in this research. I am grateful to him for teaching me some useful Carte C programming techniques. I thank Jacksonville State University for allowing access to their SRC-7 H MAP cluster and SRC Computers Inc. for providing us early access to SRC Carte v3.2 and allowing us to test our implementation on their systems.

I appreciate Randy Gelhausen and other colleagues in the Future Computing Technologies Lab for the several useful discussions we have had about this work. I thank Sudha Thiagarajan for her valuable feedback on my manuscripts. Finally, I am grateful to my friend Anusha Shankar for inspiring and encouraging me to pursue masters abroad and my family for their support and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"More than 1000 mega float point operations per second (megaFlops) are expected in future supercomputers" - Kai Hwang and Faye A. Briggs in 1984 [1].

Thanks to Moore's law and the advances in parallel computing techniques, High Performance Computing (HPC) has leapt into the petaflops era. However, there is a continuing need for even more computing performance in several domains. Specifically, in the biomolecular field, a large gap exists between the desired and achievable simulation capabilities.

Computational molecular biophysicists prefer smaller simulation systems (20,000 to 100,000 atoms with explicit solvent) and longer time-scales. On traditional parallel computers, this leads to a low computation/communication ratio. This ratio leads to a performance gap for parallel Molecular Dynamics (MD) simulations since a synchronization is required after every simulated time-step. An individual time-step is typically 1 femtosecond (e-15 seconds). Therefore, performing a microsecond MD simulation per day would typically require about a million time-steps per day or about 100 microseconds to complete one simulation time-step! Therefore, the network latency will highly impact the scalability of codes since the simulation must

synchronize after each time-step. Simulating and synchronizing at this rate is currently unachievable and in turn significantly impacts the outcome of science in this area, by limiting the scale, type, and/or number of simulations.

Fewer processing nodes with increased clock speeds, could counter the scalability problem of MD simulations, however as the silicon technology approaches atomic size, Moore's law will not hold and it will be impossible to increase clock speeds any further. The approaching end of Moore's law has prompted manufacturers to adopt multi-core technology and alternate computer architectures such as accelerator-based approaches. The first HPC system to cross the petaflop barrier, Roadrunner [2], uses an accelerator-based computing approach and more such hybrid HPC systems have emerged.

To take advantage of hybrid architectures in the HPC, many legacy applications that were originally developed for General Purpose Processors (GPPs) are being ported to hybrid architectures. Specifically, MD simulators that do not scale well beyond a few thousand nodes on conventional HPC systems are being accelerated with Graphical Processing Units (GPUs) and Field Programmable Gate arrays (FPGAs) to improve performance. For example AMBER [3] has been accelerated with FPGA-based reconfigurable system - SRC-6 H MAP [4] and NAMD [5] has been accelerated with GPUs [6]. In this thesis, we use an FPGA-based reconfigurable system to accelerate LAMMPS for biomolecular simulations.

1.1 FPGAs as accelerators

The fastest way to execute any algorithm is to use custom hard-wired technology such as an Application Specific Integrated Circuit (ASIC). Since ASICs are designed for specific computations and the parallelism in the algorithm need not be

altered to suit the hardware, typically the fastest and most power-efficient solution is provided with ASIC devices. The high capital costs involved in ASIC design and manufacturing and the inability to re-use the hardware for other applications or changes to the original application, make them unsuitable for HPC systems. In contrast, GPPs provide high-flexibility in terms of programmability, however the inherent sequential nature of processing instructions in von Neumann architectures make them inefficient in terms of work-rate, if not the clock rate. Reconfigurable Hardware (RH) such as FPGAs are mid way between the ASICs and GPPs in terms of programming flexibility and performance. FPGAs are comprised of Look Up Tables (LUTs), memory, DSP blocks and multipliers and can be programmed and re-programmed to implement any functionality. The FPGA architecture facilitates coarse grain task-level concurrency as well as concurrency via pipelining. In spite of the lower clock-rate, FPGAs are capable of providing high-speedups to compute-intensive algorithms when the available spatial and temporal parallelism are exploited.

1.2 Related Work

FPGA implementations of MD computations have been studied by several researchers. One of the initial MD implementation on FPGA studies is [7], in which, N.Azizi et al. implement the Lennard Jones (LJ) forces computation along with the Verlet integration on a Transmogrifier3 (TM3) [8] system. The TM3 consisted of four interconnected Xilinx Virtex 2000E FPGAs with external SRAMs. Their implementation uses fixed-point numbers at different scales and reports a performance of 0.29x slower than the original software implementation on a 2.4 GHz CPU.

In [9], Scrofano et al. demonstrate that the LJ force and potential calculation kernels can be implemented on an FPGA and take advantage of the parallelism

available from deep pipelining. Two LJ force calculation pipelines are implemented in VHDL for the Xilinx Virtex-II Pro XC2VP7 FPGA on the Xilinx ML300 board [10]. The implementation operates at 122 MHz and has a reported throughput of 3.9 GFLOPS for two pipelines, but the results do not include the overall performance when the LJ kernels are integrated with a complete MD simulator, which would reduce the overall speed-up

In [11], Yongfeng Gu et al. report an implementation of an MD simulator on a WildstarII-Pro [12] board consisting of two Xilinx Virtex-II Pro XC2VP70-5 FPGAs. The implementation uses fixed-point arithmetic of varying precisions. The authors show a performance gain of 31x to 88x compared to the SW implementation. However the SW implementation is not production level and is about 10x slower than production grade applications as pointed out by Scrofano et al in [13].

In [14], the authors accelerate Protomol [15] and report a speed-up of 5x to 10x for a 77k particle system, when compared to the execution time of GPP version of NAMD. Protomol is an experimental MD code, while NAMD is a mature popular parallel MD simulator. The authors use a fixed-point arithmetic approach for efficient usage of the FPGA resources and also demonstrate, that if the simulation accuracy could be relaxed, better speed-ups could be achieved with FPGA acceleration.

In [13], Scrofano et al. use an SRC-6 MAPstation [16] to accelerate the non-bonded force calculations. The authors follow design and evaluation method similar to the one presented in this thesis, however, the application used in [13] is not considered production-level code and uses a *cut-off* based method for the long-range coulombic force calculations. The authors implement a force pipeline consisting of two parts connected by a FIFO. Updating the force arrays involves storing the updated force values temporarily on the FPGA block RAMs until the end of an iteration of the outer loop and then updating the OBM banks. The authors do not implement the *newton*

off method implemented in this thesis, due to the limitations in earlier versions of the SRC Carte suite [17] for performing floating-point summations on sets of varying sizes. In Chapter 4, we discuss the design presented in [13] further and make direct comparisons with our implementation.

The major draw back in [14] and [13] is that both papers focus on acceleration of an MD simulator developed by the authors or experimental MD packages such as Protomol. Though useful in demonstrating the performance gains possible with FPGAs, these works cannot be used by application scientists unless integrated into mature production level simulators such as NAMD and LAMMPS. Such an integration is not trivial. Our approach differs from these research projects since we focus on accelerating a production-level application (LAMMPS), directly benefiting the application scientists. Further, to closely represent the end-user problem, we also select a benchmark that uses the *Particle Particle Particle Mesh* (PPPM) method (discussed in chapter 2) for long range electrostatic simulations, instead of cut-off based method as used in [13].

Authors in [18] and [4] report studies on accelerating the production-level applications NAMD and AMBER [3] respectively, and describe the steps involved in porting a large-scale scientific application to reconfigurable systems.

In [18], Kindratenko et al. describe the steps involved in porting NAMD to an SRC-6 MAPstation and report a speed-up of 3x. In [4] Alam et al. use two FPGA devices on the SRC-6 Mapstation to port the PME computations of AMBER and show a speed-up of about 3x. LAMMPS is different from NAMD and AMBER in design and the computation methodology followed. For example LAMMPS uses spatial-decomposition technique for parallel implementation while AMBER uses atom-decomposition technique. Thus our FPGA implementation is significantly different from those in [18] and [4]. Further, in this research we take advantage of the

new functionalities provided in the SRC Carte programming suite (version 3.2), which were not available during the earlier research studies with the SRC-6 MAPstations.

We believe the only other FPGA-based acceleration of LAMMPS is [19], where the author attempts to port the non-bonded force computations of LAMMPS to an XtremeData XD1000 [20] machine using the Impulse-C [21] programming tools. However the author reports errors in the XtremeData/Altera floating-point cores libraries used. The FPGA implementation is therefore not fully functional and only simulation-based performance is reported.

1.3 Our Approach

Our goal in this research is to accelerate a production-level MD simulator (LAMMPS) using the FPGA-based reconfigurable system - SRC-7 H MAP [22]. We choose a biomolecular simulation benchmark, *rhodo*, as a test case to closely represent the compute profile of a real-world biomolecular simulation problem.

We follow a hardware-software (HW-SW) partitioned approach, where the most compute-intensive tasks are offloaded to the RH while the remaining tasks (that do not benefit by porting to the RH) are executed on the GPP. Such HW-SW partitioning allows the limited FPGA resources to be used for the tasks that contribute the most to the acceleration of the overall application.

Profiling of LAMMPS for the *rhodo* benchmark helps identify the computational hot-spots, then we estimate the data transfer overheads. This information is then used to estimate the computational time on the FPGA and finally arrive at the theoretical performance. We use the Carte-C SDK, which includes a C-to-circuit translator, to develop the kernel that executes on the FPGA. We modify LAMMPS host code to invoke the RH to perform the compute-intensive calculations.

The remaining chapters are organized as follows: Chapter 2 discusses the concept of molecular dynamics simulations and specific characteristics of the LAMMPS simulator. Chapter 3 provides an overview of reconfigurable computing (RC) platforms in general and specifically about the SRC-7 H MAP system used in this work. In Chapter 4 we discuss the implementation techniques and the optimizations performed to obtain speed-up using RC. Chapter 5 presents the experimental results and discussion on performance gained. Finally in Chapter 6 we present the conclusions and proposed future work.

Chapter 2

Background

In this chapter we review Molecular Dynamics basics and the computational complexity of the MD simulations. Further we discuss the characteristics of LAMMPS that make it attractive for acceleration.

2.1 Molecular Dynamics

Fundamental physics-based simulation of Biomolecular behaviour complements experimental analysis allowing scientists to understand the biomolecular systems better. Critical information, that is difficult to study from experiments, such as molecular movements, molecular interactions and change in structures over short time scales, can be obtained from simulation results. Biomolecular simulators use classical Molecular Dynamics (MD) methods to simulate such biomolecular system behavior typically in picosecond to nanosecond timescales.

In a typical MD simulation, atoms are allowed to interact with each other over a certain number of discrete time-steps and the trajectory of motion of atoms are calculated based on Newton's second law of motion: $F = ma$ where F is the

force acting on an atom, m the mass and a the acceleration of the atom. During each time-step, knowing the mass, and the force acting on the atoms, the acceleration can be calculated. The new position and velocity are then calculated from the previous position, velocity and acceleration.

A basic MD simulation can be summarized in the following steps:

1. Read initial states of the atoms
2. Calculate the forces acting on each atom
3. Compute the acceleration of each atom
4. Determine the new positions and velocity of the atoms after a time-step
5. Repeat steps 2 through 4 for the required number of time-steps

The forces in a MD simulation are calculated from the potential energy E of the system, which is a combination of bonded energy due to interaction of atoms that are chemically bonded to each other and non-bonded energy due to interaction with all other atoms. The bonded energy contribution consists of only a few thousand interactions and thus is less complex to compute than the non-bonded energies that consist of millions of interactions. The potential energy of a system of N atoms is represented as shown in equation 2.1.

$$\begin{aligned} Energy_{Potential} = & \underbrace{\sum_{bonds} K_a(r-r_0)^2 + \sum_{angles} K_\Theta(\theta-\theta_0)^2 + \sum_{dihedrals} K_{\phi_p}[1+d_p cos(n_p\phi)]}_{bonded\ energy} \\ & \underbrace{+\sum_{i=1}^{N}\sum_{j>i}^{N}\epsilon_{ij}[(\frac{\sigma_{ij}}{r})^{12}-(\frac{\sigma_{ij}}{r})^{6}] + \sum_{i=1}^{N}\sum_{j>i}^{N}\frac{q_i q_j}{r}}_{non\ bonded\ energy} \end{aligned} \tag{2.1}$$

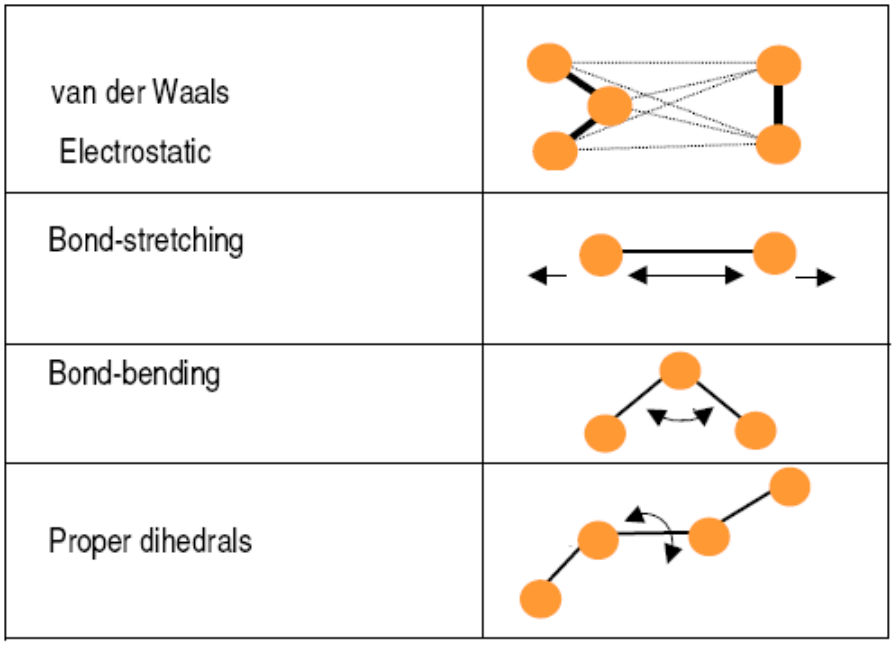

Figure 2.1: Types of atomic interactions that contribute to energy potential [23]

The first three terms in Equation 2.1 represent the harmonic two-body, three-body, and four-body interactions within the molecule and together represent the bonded energy. Computation of these bonded terms involves a single summation and is of complexity O(N). The fourth and fifth terms in equation 2.1 represent the van der Waals and electrostaic interactions respectively, and contribute to the non-bonded energy of the system. Both terms involve a double summation over N atoms, making the non-bonded energy calculation O(N^2), where N is the number of atoms in the system. Thus for large values of N, the simulation becomes computationally very demanding. Figure 2.1 shows the different types of atomic interactions.

To reduce the computational complexity of non-bonded energy computations, approximation techniques such as the cut-off based method is used in many MD simulations. For any atom, the non-bonded energy contributions of the neighbor atoms at long distances are much smaller when compared to those of the nearest neighbor

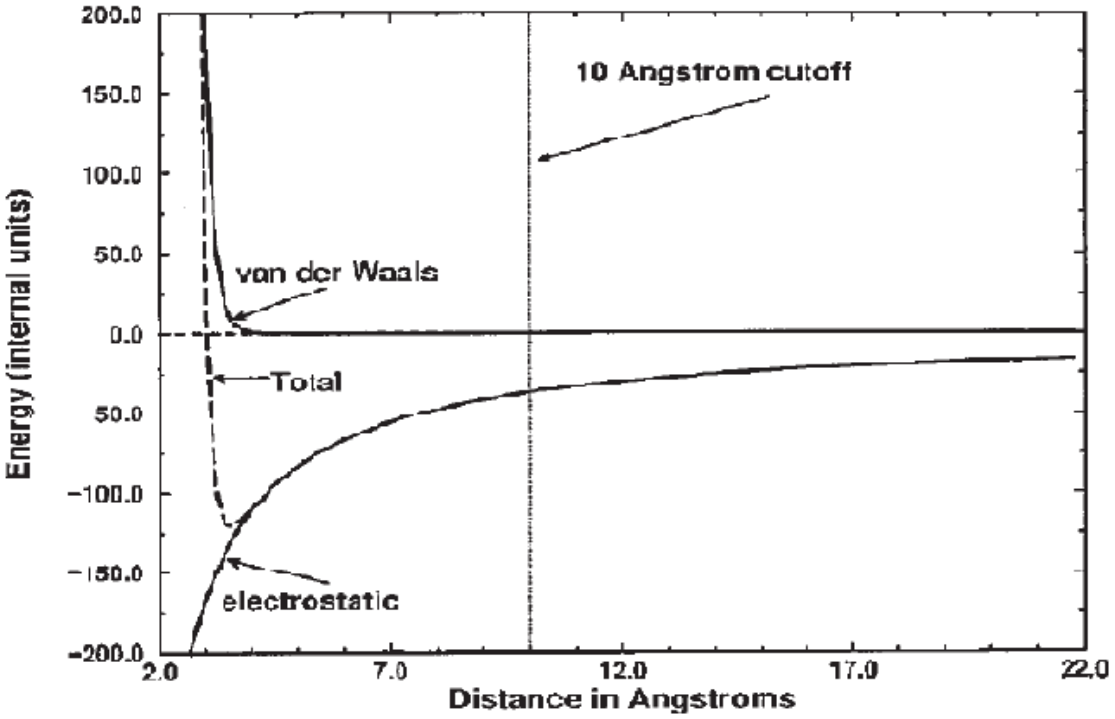

Figure 2.2: van der Waals and electrostatic forces between atom pairs as a function of inter-atomic distance [24]

interactions. Therefore, considering only neighbor atoms within a cut-off radius, the number of calculations can be drastically reduced from $N * (N-1)$ to N^*(Number of atoms within the cut-off radius). The approximation error introduced by the cut-off based method is within acceptable limits for the van der Waals interactions, but is large for electrostatic interactions. Figure 2.2 shows the decrease in van der Waals forces and electrostatic forces between atom pairs, with an increase in inter-atomic distance. While van der Waals contribution quickly drops to zero long before the cut-off distance (10 Å), the electrostatic contribution is significant at the cut-off distance and ignoring them would result in large errors in the energy computations. To avoid large approximation errors in electrostatic energy summations, transform based techniques such as Ewald Summation, Particle Mesh Ewald (PME) and Particle-Particle

Particle-Mesh (PPPM) methods have been developed. The computational complexity and scalability of the MD simulation depends greatly on the approximation technique used.

2.2 LAMMPS

There are several MD simulators such as NAMD [5], LAMMPS [25], and AMBER [26] available to the community either as open source or commercial software. The mathematical models used by these simulators and their resulting performance in terms of speed and accuracy of the simulation may vary slightly. Based on the application domain or the purpose of the simulation, one simulator may be preferred over another. In this work, we use LAMMPS as the MD simulator for acceleration with Reconfigurable Computing (RC). LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) is a parallel implementation of MD simulation distributed as an open source code by Sandia National Laboratories. LAMMPS employs the spatial decomposition technique to divide the physical geometry to be simulated into small boxes, one per processor. Each processor primarily works on the atoms within its box, referred to as owned atoms, and may use the information of atoms owned by other processors (known as other atoms) to compute neighbor atom interactions [27].

Important features of LAMMPS that make it an attractive application for accelerated biomolecular simulation:

1. Biomolecular simulation capabilities - LAMMPS is one of the few MD applications that can model both the CHARMM and AMBER force fields (potential function), an important feature for biomolecular simulations.

2. Optimized parallel as well as single node implementation - Single node ac-

celeration with reconfigurable computing could be expanded in the future to multi-node optimizations in a cluster.

3. Direct impact to end user community - LAMMPS is an open-source code, distributed under the terms of the GNU Public License (GPL). Which impacts the availability of the optimized software to the wide user community. In keeping with the terms of the GPL, all of the code described in this thesis is also freely available.

LAMMPS is capable of modelling systems with a few to billions of particles using a variety of force fields and boundary conditions for applications in chemistry, biology, and material science. LAMMPS provides several user configurable options to select the approximation technique to be used for Coulombic interactions, such as PPPM and Ewald summations, the time integrator for the simulation such as rRespa and velocity Verlet [27]. These simulation configurations are provided in the form of input script commands.

The LAMMPS codebase contains sample benchmarks to compare the performance of LAMMPS on different architectures. Since we are interested in accelerating LAMMPS for biomolecular simulations, we have selected the Rhodopsin protein benchmark *rhodo*, shown in Figure 2.3, as our test case. The *rhodo* benchmark consists of 32000 atoms of Bovine Rhodopsin protein contained in a solvated lipid bilayer with water as the solvent surrounding the top and bottom of the lipid layer. the *rhodo* benchmark uses the velocity Verlet method for time integration and the PPPM method to solve long-range Coulombic interactions. As mentioned earlier, the scalability of the transform method greatly influences the scalability of the MD simulator, hence it is important to note that PPPM scales as $N\sqrt{logN}$ [27] .

LAMMPS uses a neighborlist technique to keep track of neighboring atoms

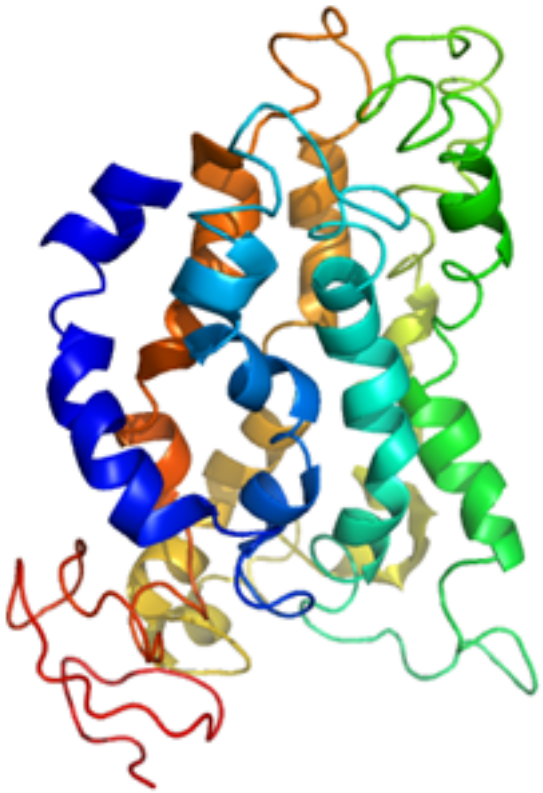

Figure 2.3: Rhodopsin protein [25]

that interact with each other. The size of the neghborlist is determined by the cut-off distance (configured in the input script file). LAMMPS uses a switching function to smoothly ramp down the non-bonded energies to zero. The inner and outer cut-offs used by the switching function are defined in the *pair_style* input command and are 8 and 10 Å for the *rhodo* benchmark. Further, to avoid a neighborlist re-build at every time-step, a skin distance (configurable in the input script) is added to the outer cut-off of the switching function when calculating the neighborlist. The *rhodo* benchmark uses the default skin distance of 2 Å. These settings result in an average of 375 neighbors per atom. Since an additional skin distance is used during neighbour list generation, approximately 60% of the atoms in the neighborlist participate in

the pair-wise force computations in any given time-step. However, the use of a skin distance results in the neighborlist generation only once in every 8 time-steps.

The neighborlist size in *rhodo* is approximately 12 million, making it the most compute-intensive benchmark. Table 2.1 shows the performance comparison of the five benchmarks available in LAMMPS codebase. Due to the complexity involved in the non-bonded pair interactions described earlier, it is more than 18x slower than the LJ benchmark. Accelerating LAMMPS for the Rhodopsin benchmark will be significant due its relevance to bimolecular field and its relatively poor performance on CPU machines.

Problem	***LJ***	***Chain***	***EAM***	***Chute***	***Rhodopsin***
CPU/atom/step	1.35E-6 S	6.25E-7 S	3.62E-6 S	5.91E-7 S	2.47E-5 S
Ratio to LJ	1.0	0.46	2.69	0.44	18.4

Table 2.1: Performance comparison of benchmarks available in LAMMPS codebase [25]

In this chapter, we discussed the application and the computational complexity of MD simulations. In Chapter 3 we will discuss the reconfigurable computing platform used in our implementation to accelerate the compute-intensive non-bonded force calculations of LAMMPS.

Chapter 3

Reconfigurable Computers

The concept of reconfigurable computing was first proposed by Estrin et al.[28], when they introduced the idea of re-using computational structures to perform independent computations and using multiplexers to route connections between the components. While the idea of re-using components is common place in modern microprocessor architectures, today the term *reconfigurable computing* refers to hardware-level reconfiguration with a programmable logic device such as an FPGA. In this chapter we provide an overview of the FPGA architecture and discuss the various ways to couple the FPGA with a microprocessor to accelerate applications. We also discuss the SRC-7 H MAP architecture, that is used in this research.

3.1 FPGA Overview

The computational units of an FPGA device are the Logic Elements (LEs), composed of a group of Look Up Tables(LUTs) and Flip-Flops as shown in Figure 3.1. The LUT allows any functionality (with up to N-inputs) to be implemented, where N is the number of inputs to the LUT. Today's devices have 6 input LUTS

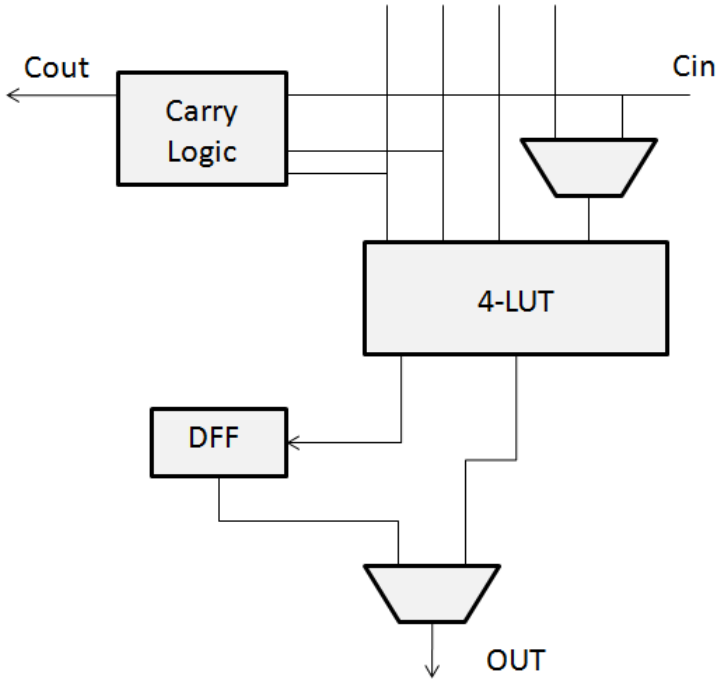

Figure 3.1: Basic logic block of an FPGA [29]

allowing them to implement complex logic functions. The flip-flop is used to hold the intermediate outputs for pipelining, or serve as registers [29]. The logic blocks are placed in two dimensional arrays and are connected to a routing fabric via connection blocks. The connection blocks can be programmed to select signals that connect to the logic blocks. Thus hundreds or thousands of LUTs can be routed together to perform meaningful high-level tasks. The first commercially viable FPGA XC2064, was made available by Xilinx in the year 1985. The XC2064 had 64 configurable logic elements with two 3-input LUTs each. Such early FPGAs, could hardly be used for any compute-intensive tasks, they were rather used for light-weight embedded applications. The current FPGA technology has advanced to a point where even HPC kernels can be accomodated on FPGA devices. Taking advantage of the task-level parallelism and pipelining that FPGAs offer, HPC kernels on FPGAs can execute

faster than their GPP versions. To compare the raw computing power of a modern FPGAs with a microprocessor, the Xilinx Virtex-5 SX240T with 149,760 LUTs and 2 outputs per LUT (149,760 LUTs * 2 bit operators per LUT * 250 MHz * 1/64) is capable of 1.17 trillion 64-bit op/s, while the quad-core Opteron (4 cores * 4 ops per clk * 2500 MHz) is capable of 40 billion 64-bit op/s, which amounts to (1170/40) 29x more raw computing performance. Further, the operating frequency of FPGAs is limited to approximately 250 MHz, which is an order of magnitude less than GPPs. However, the FPGA is able to provide 29x more computing power when compared to a quad-core Opteron, thereby providing more computational power per watt.

3.2 Reconfigurable Computers

A typical reconfigurable computing system consists of one or more microprocessors also known as General Purpose Processors (GPPs) coupled to RH such as an FPGA. The compute-intensive tasks or custom instructions are offloaded to the FPGA for acceleration.

The FPGA device can be coupled to a GPP to form a reconfigurable system in one of the following ways [29].

FPGA integrated within the processor to process custom instructions that may change over time. Communication latency is low, however the FPGA size is limited and thus not all tasks can be ported to take advantage of the parallelism in FPGA.

FPGA as an On-Chip embedded processor provides low-latency communication. In this architecture both task-level and instruction-level parallelism are achievable.

FPGA as a Coprocessor where the RH is typically coupled to the GPP via a memory interface or a peripheral interface such as PCIe. This configuration allows the RH to compute independently over large chunks of computation and thus is well suited for task-level acceleration of data-intensive applications.

Our goal in this research is to use FPGAs to accelerate the compute-intensive tasks of a biomolecular simulation with the LAMMPS framework. Since we intend to work on task-level granularity, it is advantageous to use a reconfigurable system with the following characteristics:

1. Fixed architecture CPU closely coupled with FPGA devices: enables the application partitioning where I/O operations and other tasks that do not benefit from FPGA acceleration to execute on the CPU while the most compute-intensive tasks are ported to the FPGA. Closely coupled systems minimize the data transfer overheads.

2. High communication bandwidth between CPU and FPGA: decreases the communication overheads and sustains the speed-up achieved by accelerating individual tasks.

3. A high-level programming environment: provides automatic translation from C/Fortran to VHDL/Verilog which is used to generate the FPGA bit stream.

Some of the popular high-performance reconfigurable systems available today that meet these specifications include the XtremeData XD1000/2000, DRC DS1000/2000, and SRC MAPstations.

3.3 SRC-7 H MAP

The SRC-7 H MAP is the latest from the SRC Computers's MAPstation series. The SRC MAPstation consists of a MAP processor (RH) coupled to the GPP via the memory interface (DIMM slots). The MAPstations can be scaled to a cluster using ethernet interconnect and can share common memory across nodes using SRC's Hi-Bar switch as shown in Figure 3.2.

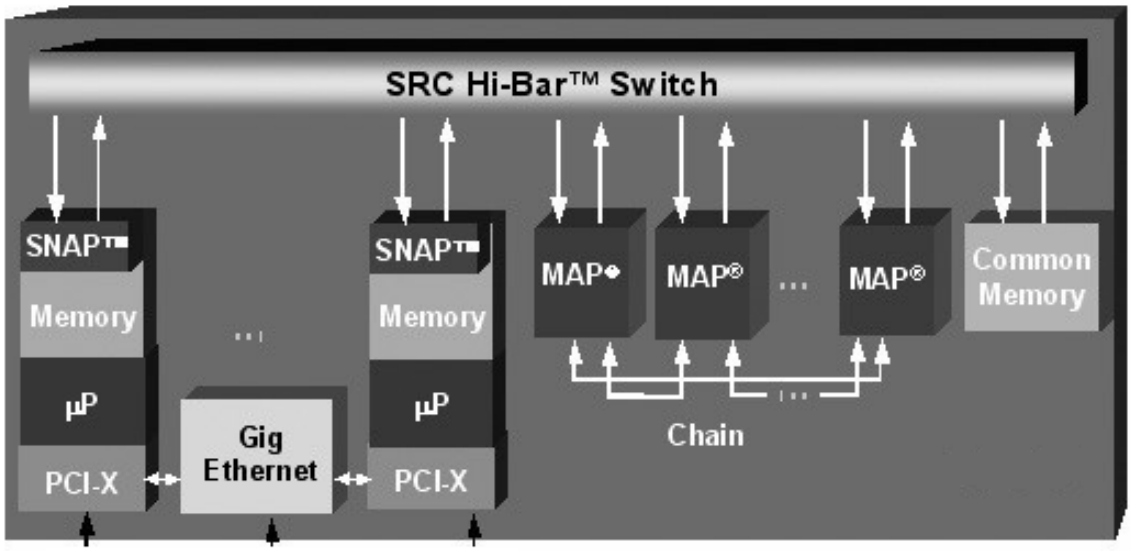

Figure 3.2: SRC MAP cluster with Hi-Bar switch and common memory [16]

In this research, we focus on accelerating LAMMPS on a single SRC-7 H MAP node, which consists of an Intel Xeon 3.0 GHz dual-core processor coupled with reconfigurable MAP processor. The MAP processor consists of two 150 MHz Altera Stratix II EP2S180 FPGAs, connected to each other by a 128-bit data channel. The block diagram of the Altera Stratix II EP2S180 FPGA is shown in Figure 3.3. Each EP2S180 device on the MAP consists of 179,400 Logical Elements (LEs), 97 embedded DSP blocks, and about 9 Mbits of embedded memory.

Apart from the memory embedded within the FPGA, the devices have access

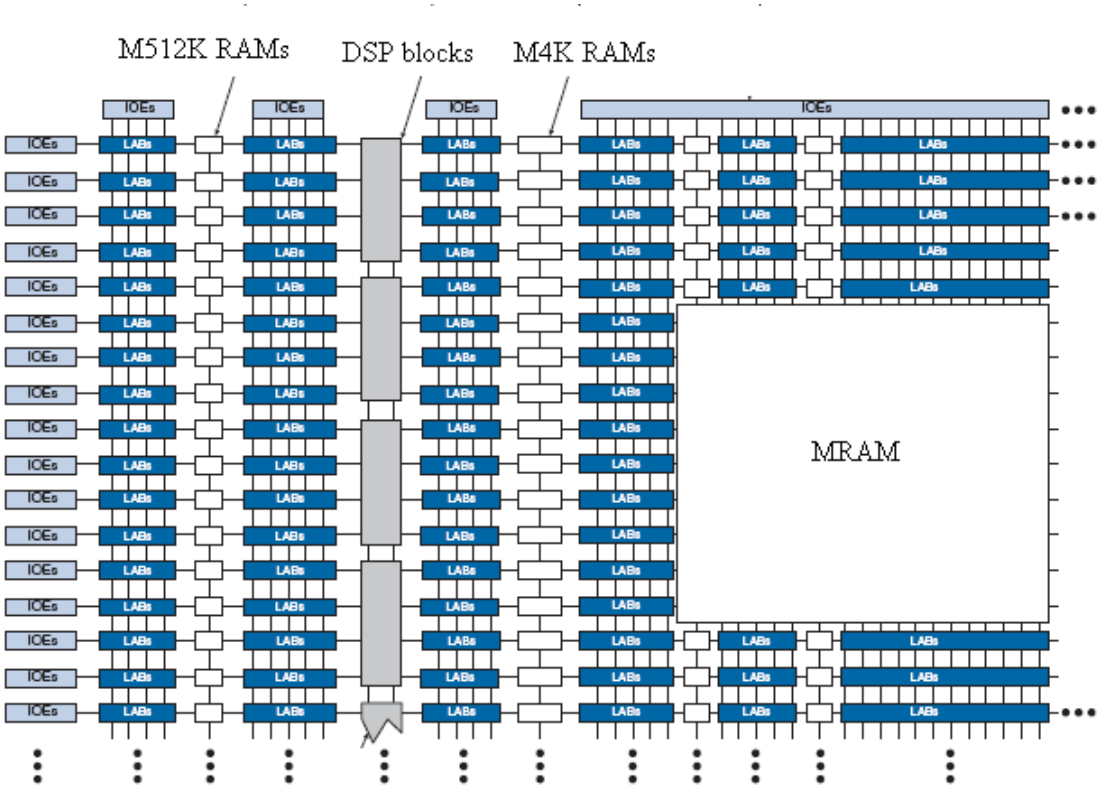

Figure 3.3: Altera Stratix II FPGA block diagram [30]

to 64 MB of SRAM, distributed across 16 On Board Memory (OBM) banks. The OBM banks are interfaced to the FPGA with 64-bit ports, thus a 64-bit independent memory reference can be made from each of the 16 OBM banks, adding up to a maximum memory bandwidth of 19.2 GB/s. Large data sets that cannot be accommodated on the FPGA Block RAMs or the OBMs can be stored in a third level of memory known as the Global Common Memory (GCM). The SRC-7 H MAP that we use in this research, has 1 GB of GCM. The GCM is accessible both from the microprocessor as well as the MAP processor, thus, could be used as an intermediate storage location for large data sets that need to be transferred between the processors.

The GPP and the MAP processor are connected via a SNAP interface that allows the GPP and the FPGA to share the system memory as peers, thus the commu-

nication bandwidth between the GPP and the FPGA is only limited by the memory characteristics of the system. The SNAP interface in the SRC-7 H MAP has seperate input and output ports and interfaces with the GPP motherboard's memory interface (DIMM). The SRC-7 H MAP system uses a DDR2 memory based SNAP interface, which provides a sustained bandwidth of 7.2 GB/s.

3.3.1 SRC Carte Programming suite

The SRC carte programming suite enables users to program in a high-level language (Fortran or C) and abstracts away the underlying architectural complexities of the system hardware. The SRC Carte suite consists of MAP-Fortran and C compilers for the FPGA hardware, Intel Fortran and C compilers for the host processor(s), Carte MAP Macro libraries for frequently used and optimized functions, and Quartus FPGA Place and Route to generate the bitstreams for the FPGAs. Figure 3.4 shows the steps involved in generating an unified executable for a C application. The functions that execute on the CPU are compiled with the appropriate Intel compiler, while the MAP function to be executed on the FPGA is compiled with the SRC Carte C compiler. As a part of the C to bit-stream translation process, the MAP C compiler invokes Altera Quartus Place and Route (P&R) to convert the intermediate EDIF files to an FPGA bit stream. The P&R process could take anywhere from several minutes to a couple of hours to complete, depending on the complexity of the MAP code. Upon completion of P&R, the FPGA bit stream is included in the object file created by the MAP C compiler. Finally the object files generated by the Intel compiler and the MAP C compiler are linked together with Intel linker to form a unified executable [17]. The SRC Carte progamming suite also provides the developer with debug mode and simulation mode compile options, which can avoid

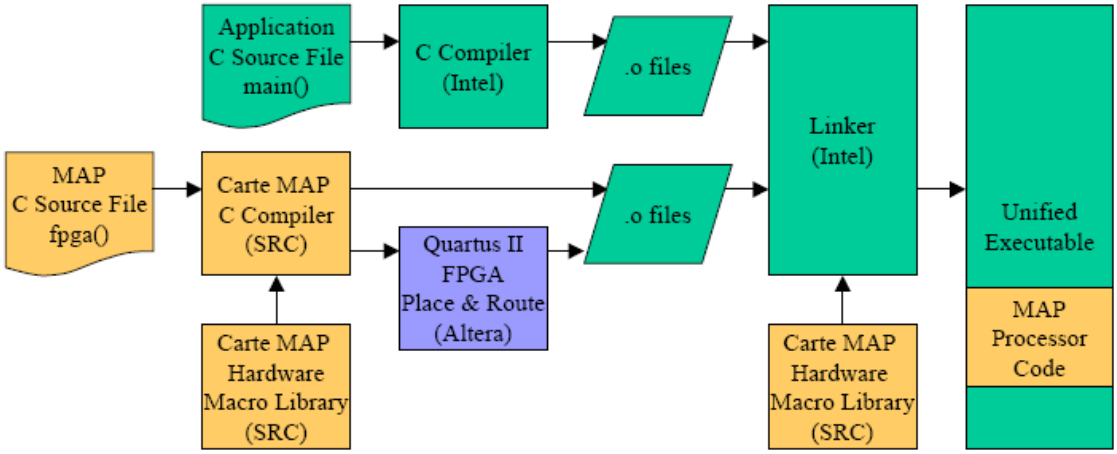

Figure 3.4: Carte unified execution compilation process [22]

the time consuming P&R process during functional verification.

This chapter presented an overview of the FPGA architecture and how it can be coupled to GPP to achieve acceleration. We also discussed the architecture of the SRC-7 H MAP reconfigurable system used in this research. In the next chapter we will discuss the design and implementation of the LAMMPS application on the SRC-7 H MAP.

Chapter 4

Design and Implementation

Gaining application performance on reconfigurable systems involves strategic partitioning of the application based on profile data from the GPP execution, careful planning of data movements, and rigorously designing the FPGA code around any hardware limitations of the system. This chapter discusses the various steps followed in the design and implementation stages, the design options considered, and the reasons behind the choices made in the implemented design. Further, it provides insight into the performance results expected from the implementation.

4.1 HW-SW Partitioning

The first step in accelerating an application using a hybrid architecture is to identify the compute-intensive functions in the application through profiling. The *Gprof* tool provides flat profile data useful in identifying the most compute-intensive tasks of an application, and call graph information indicating the calling order of the functions being called and the parent or child functions of the given function. We use the flat profile information to identify the functions to consider for acceleration

by RC. The call graph is used to determine if any of the less compute-intensive tasks should be agglomerated with their parent function or child function that is being ported to reconfigurable hardware. Such agglomeration of functions helps to reduce calling overheads and avoid unnecessary data movements between the reconfigurable hardware and the GPP.

Profiling LAMMPS for the Rhodopsin benchmark (*rhodo*), revealed that approximately 70% of the total execution time is spent in the `PairLJCharmmCoulLong::compute` function, making it a natural candidate for acceleration. Analyzing the call graph, we understand that `PairLJCharmmCoulLong::compute` is called once per time-step by `Verlet::run` and is a parent of the functions `Pair::ev_setup`, `Pair::ev_tally`, and `Pair::virial_compute`. While only a trivial amount of time is spent in the three child functions, the number of calls made to these functions varies. For the Rhodopsin benchmark where n_step=100, `Pair::ev_tally` is called 20858799 times while `Pair::virial_compute` and `Pair::ev_setup` are each called 101 times (once per time-step). We in-line the child functions along with their parent `PairLJCharmmCoulLong::compute` to avoid function calling overheads and data transfer overheads.

4.2 Accelerating non-bonded force computations

The `PairLJCharmmCoulLong::compute` function computes the forces due to the pair-wise non-bonded interactions - the Coulombic (or electrostatic) forces and the Lennard-Jones (or van der Waals) forces. The compute function has two loops, an outer loop of `nlocal` atoms (32000 for Rhodopsin benchmark on a single processor) and inner loop that iterates over the neighborlist of each atom. The size of the neigh-

borlist thus determines the cumulative number of iterations in the compute function. The inner and outer cut-off distances used by the Switching function for pair-wise force computations in Rhodopsin benchmark are 8 and 10 Ångstroms respectively resulting in an average of 375 neighbors per atom. Thus the total number of iterations in the `PairLJCharmmCoulLong::compute` function per time-step for the Rhodopsin benchmark is approximately 12 million. During each of these iterations, the distance between the pair of atoms is checked. If the distance is within the cut-off distance for Lennard-Jones (LJ) interaction, the LJ force for the pair is computed. Likewise for the pairs that satisfy the coulombic cut-off, the coulombic forces are computed. For the pairs that do not satisfy either of the cut-off distances, the force is simply assigned as zero, avoiding a significant number of floating-point operations on the CPU.

As discussed in Chapter 3, FPGAs offer both spatial parallelism (concurrent execution of independent tasks) and temporal parallelism (pipelined execution). We pipeline the loop (henceforth referred to as the *compute loop*) that iterates over the pairs of atoms to compute the non-bonded forces. In an ideal situation, without any stalls for memory access or scalar dependencies, a fully pipelined compute loop on the FPGA would produce one iteration per clock cycle once the pipeline was filled, irrespective of whether the force needs to be computed for the pair or not. Thus, on an FPGA operating at 150 MHz, computation time alone for an optimized, maximum pipelined loop of 12 million iterations would be 12 million iterations x $6.67\text{x}10^{-9}$ (seconds per clock cycle), which equals 80 ms. This loop performance will be deterministic irrespective of the number of pairs that fall under the cut-off distance. On a Xeon processor the compute loop consumes 990 ms when 60% of the atom pairs satisfy the cut-off distance (as in the case of Rhodopsin benchmark) and this time will increase when the percentage of atoms that satisfy the cut-off distance

increases. Thus we estimate an ideal speed-up of 990/80 = 12x for the non-bonded force computations for a single pipeline implementation on the FPGA.

To obtain a speed-up close to the ideal speed-up, our design goals were:

1. Create a non-bonded force calculation pipeline of maximum throughput that computes the forces between one pair of atoms per cycle,

2. Minimize the data transfer overheads and overlap the data transfers with computation.

4.2.1 The design

4.2.1.1 Maximum throughput pipeline

The Carte C compiler automatically pipelines the innermost loops in the MAP function, however to avoid stalls in the pipeline the user must ensure that there are no loop-carry memory dependencies or loop-carry scalar dependencies. In both cases, the pipeline will stall until the values generated in the previous iterations are updated. Further, the OBM banks on the MAP are single ported, which means multiple access to the same OBM bank is not possible in one clock cycle. Also since the ports are shared for both reading and writing data, there is a penalty of two cycles associated with switching between read and write modes. An extra clock cycle per iteration translates to a MAP execution time that would be 2x slower, thus to outperform a microprocessor operating at 3.0 GHz with a MAP processor operating at a frequency of 150 MHz it is critical to avoid such loop delays in the design.

The basic psuedo code for the CPU implementation is as shown in Figure 4.1. The outer loop iterates over all real atoms, loading the current `i` atom details such as position, charge, and neighborlist. The innerloop iterates over the neighborlist of `i`, loads the atom details of the current neighbor atom `j` to proceed with the force

```
1 for each atom i
2     read i atom position,charge,type
3     for each j in neighborlist[i]
4         read j atom position,charge,type
5         compute distance rsq between i & j
6         if rsq<cut_bothsq
7             if rsq<cut_coulsq
8                 compute f_coul
9             else
10                f_coul=0.0f
11            endif
12            if rsq<cut_ljsq
13                compute f_lj
14            else
15                f_lj=0.0f
16            endif
17            fpair=f_coul+f_lj
18        else
19            fpair=0.0f
20        endif
21        force[i] = force[i] + fpair
22        force[j] = force[j] - fpair
23    endfor
24 endfor
```

Figure 4.1: Pseudo code of non-bonded force computations in LAMMPS

computation for the atom pair. The Carte C compiler will automatically generate stalls in the pipeline to avoid read access conflicts between atom `i` and atom `j`. Thus to avoid stalls in the pipeline, we replicate the atom data across multiple OBM banks, one copy to be used by `i` atom and the other to be used by the `j` atoms.

As shown in lines 21 and 22 of Figure 4.1, the calculated force is updated on the current atom `i` and using Newtons third law, an equal and opposite force is added to the `j` atom. Updating the force on the OBM banks at the end of each iteration requires a read followed by a write access to the same bank. Updating forces on the same bank implies multiple accesses, as well as a switch between read and write modes to the bank, both of which take an extra clock cycle due to the hardware limitations

discussed earlier in this section. Therefore, we must avoid updating the forces on the same bank. The forces for `i` atoms can be summed without stalls using the stream-based floating-point accumulator (`stream_fp_accum_32_rr_term`) available in Carte C. Summing the forces for the `j` (neighbor) atoms is however a problem due to the random nature of the memory indexes.

We consider three design options to avoid pipeline stalls due to updates to the `j` atom's forces on the MAP processor. We now will estimate the computation time, communication times, and the speed-up achievable with each of the design options.

Design 1 - Perform the force summations on the host: This option requires transferring the forces computed for each of the 12 million pairs and updating the forces array on the host iterating over `all_pairs`. Transferring the forces back to host does not involve additional overhead since we perform a streamed DMA transfer overlapping with the force computation. However since the host starts the forces update loop only after the MAP completes its execution, updating the forces on the host will take additional time. The timing estimation for Design 1 is shown in Figure 4.2, t_{DMA_in} is the time required to transfer the atom position data every time-step, and t_{NL_copy} is the time required to copy the cumulative neighborlist to the GCM bank. With a communication bandwidth of 3.6 Gbps, t_{DMA_in} will be 0.3 ms for the *rhodo* benchmark containing 32K atoms; and to copy a neighborlist of size 12 million, t_{NL_copy} is estimated to be 13 ms and since t_{nl_copy} occurs only during a neighborlist build (typically, once in eight time-steps), the per step contribution of this data transfer is 1.7 ms. As estimated in the previous section, the force calculation time *t_calc* is about 80 ms in a stall-free pipeline. The time taken to transfer the forces for 12 million atom pairs to the host, t_{DMA_out}, is estimated to be 53 ms ((12 million x 4 x 4 bytes)/3.6 Gbps). Since the forces are transferred as they are produced by the compute loop, the time t_{DMA_out} is hidden by the the computation time. Simple

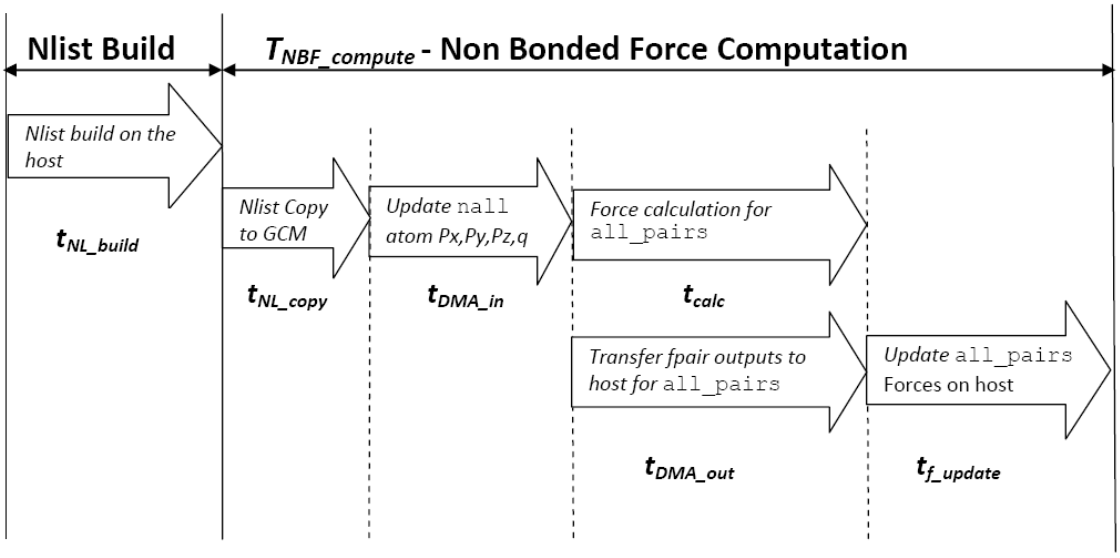

Figure 4.2: Estimated time in Design 1

experiments show that performing 12 million iterations and updating force arrays on the host would take 250 ms (t_{f_update}). Thus, the total time taken per step to compute the non-bonded forces will be 332 ms, which will be a speed-up of about 3.6x, when compared to the un-accelerated non-bonded force computation time of 990 ms/time-step. Combined with the full application this acceleration translates to 1.88x overall speed-up of the LAMMPS simulation time for `rhodo`.

Design 2 - Turn off Newton's third law: The force summations for `j` atoms can be entirely avoided by turning off the Newton's third law flag in LAMMPS and using a full neighborlist. The timing estimation for Design 2 is shown in Figure 4.3. With a full neighborlist, the number of non-bonded atom pairs increases by a factor of two, thus the computation time t_{calc} increases to 160 ms, twice the value of Design 1. Further, the time required to copy the neighborlist to the GCM (t_{NL_copy}) increases to 27 ms (24 million x 4 bytes/3.6 Gbps) per transfer, which is equivalent to

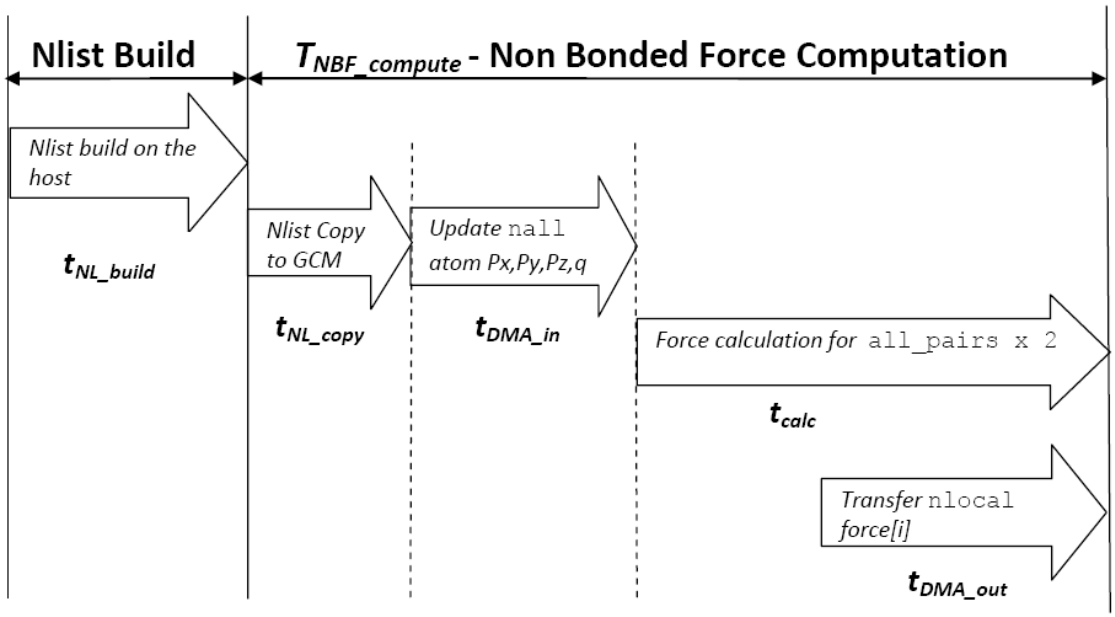

Figure 4.3: Estimated time in Design 2

3.3 ms per time-step. However in contrast to Design 1, we avoid the loop of 12 million iterations on the host required to perform the force updates. The value of t_{DMA_in} remains the same as Design 1, thus the estimated time/step spent in non-bonded force calculation, $t_{NBF\_compute}$, is 163 ms, which is an acceleration of about 990/163 = 6x over the compute time for the pair-wise calculations on the host. The main disadvantage of this design is the increased time spent building the neighborlist on the host. The time spent building the neighborlist would typically double for a full neighborlist generation. Profiling *rhodo* shows that the time spent in half neighborlist generations during a 100 time-step execution is 23 s (16 % of total execution time), we expect this time to increase to 46 s for a full neighborlist creation. Thus the overall speed-up for the application will be 1.7x.

Design 3 - Buffered copy of forces: In [13], the authors propose a method for reading the neighbor atom forces from the OBM bank, updating and storing them temporarily on an FPGA block RAM array, and at the end of iterations over all

neighbor atoms of the current `i` atom, copying the forces back to the OBM. Pipeline stalls due to memory dependencies could be avoided with this method, however the compute pipeline must be drained at the end of every `i` iteration. The authors estimate the time consumed for this method as $t_{calc} = n(N+p) + n(N+1) + n + sn$, where n is the total number of real atoms, p is the pipeline depth, N is the average number of neighbors per real atom and s is the sum of switching delays of the OBM between read mode to write mode and write mode to read mode. For the Rhodopsin benchmark of 32000 atoms and 375 neighbors on an average implemented on a pipeline of depth of 495 stages (estimated based a basic design implementation), the time $t_{calc} = 270ms$, and $t_{NBF\_compute}$=272ms ($t_{step} = t_{NL\_copy} + t_{DMA\_in} + t_{calc}$), which translates to a speed-up of 3.6x for the compute function and an overall speed-up of about 2x for LAMMPS.

Estimation of neighborlist build time on the FPGA: The total number of atoms in the system, which includes the real atoms and the ghost atoms, is represented as `nall`. The number of atoms that are "owned" by a parallel task is represented as `nlocal`. The neighborlist build function on the FPGA iterates over every possible atom in the system to check if the atom falls within the cut-off distance. Our initial design for the neighborlist build function, consists of four pipelines and each pipeline computes the nieghborlist for `nlocal`/4 atoms. For every `nlocal` atom `i`, an inner loop iterates over `nall j` atoms in the system. If the `j` atom is not in the special atoms (bonded atoms) list of atom `i` and if the distance between the current atom and the neighbor atom `j` is less than the cut-off distance, which is maximum of the LJ cut-off and Coulombic cut-off, the index `j` is added to the neighborlist of atom `i`. Since all four pipelines iterate over the same list of `nall j` atoms in the system, the coordinates of the `j` atom can be read once and streamed to all four pipelines. However, each pipeline has a different set of `i` atoms, and thus needs a

separate copy of the atom coordinates data to avoid multiple access to OBM banks (and therefore avoid stalls in pipeline). Two OBM banks are required to store the `i` atom coordinates for each pipeline and two OBM banks are required to store the `j` atom coordinates to be streamed to all four pipelines. Thus the proposed design with four pipelines requires 10 OBM banks to store the atom coordinates data. Further the special atoms list is stored in two OBM banks and one additional OBM bank is used to store the number of special atoms for each atom `i`. Thus a total of 13 of the 16 available OBM banks will be required, limiting any further increases in number of pipelines.

The neighborlist build time on the FPGA can be calculated as *Average number of iterations per pipeline / FPGA clock frequency.* Number of iterations per pipeline = (`nlocal`/number of pipelines) x `nall` x avg. no. of special atoms per `i` atom + number of stages in the pipeline. For *rhodo* `nlocal` = 32K (`i` atoms) and `nall` = 80K and each `i` atom has about 4 special atoms on average. From our initial implementation, we estimate the pipeline depth to be about 80 stages.Thus, based on the FPGA operating frequency of 150 MHz, the estimated time per neighborlist build is about 17 seconds on the FPGA (versus 2.4 seconds on the host). Clearly this design is about 6x slower than a half neighborlist build on the host processor. In spite of the reduction in neighborlist copy time (t_{NL_copy}), the current design of the neighborlist build function does not improve the overall performance of LAMMPS. The inefficiency in the FPGA implementation is due to the increased number of iterations (full neighborlist build compared to a half neighborlist on the host processor), lack of optimizations such as the binning technique used on the host processor that allows a bin of atoms to be tested or not tested based on whether the bin physically lies within the cut-off distance of the current i atom. Thus, there is scope for further design optimization of the neighborlist build function on the FPGA.

Description	*Design 1*	*Design 2*	*Design 3*
Individual speed-up of Compute function	3.0x	6.0x	3.6x
Overall speed-up with nlist build on host	1.9x	1.7x	2.0x

Table 4.1: Comparison of projected speed-ups for the three designs

Table 4.1 summarizes the speed-up projections for the three proposed designs of non-bonded force computations. Though the overall speed-up of LAMMPS execution is marginally less when compared to Design 1 and Design 3, we chose Design 2 for our implementation since there is better acceleration provided on non-bonded forces compute function. Further, Design 2 will provide maximum speed-up among the three designs if the neighborlist build function on the FPGA could be optimized further and executed concurrently with the non-bonded force calculations in the future.

4.2.1.2 Optimized Data Transfers

For an application executing on a hybrid architecture, the data movement between the GPP and the RH, if not handled appropriately, may off-set any acceleration acheived by the RH [31]. In the following section we analyze the data transfers and the methodology employed to minimize the overheads.

The data required for the force computations are: the position coordinates (`Px`, `Py` and `Pz`), charge `q`, atom `type` of all atoms in the system, and the neighborlist for all the real atoms in the system.

Also we need the constant array `lj` and the `PPPM` tables. The data stored in the OBM banks, FPGA block RAMs, and the GCM memory are persistent over multiple calls to the MAP function (unless otherwise reset by a `MAP_allocate()` or `MAP_free()` call). Thus to avoid redundant data transfers, we characterize the data

update and access patterns during runtime and only transfer modified data to the MAP processor. Table 4.2 lists the data to be transferred and the corresponding update frequency. The atom positions change during each time-step, so `nall` x 3 coordinates x 4 bytes of position data are transferred during each time-step. The atom type and charge on the atoms do not vary every time-step. However, each time the neighborlist is rebuilt, the ghost atoms in the system change, hence we transfer the atom charges and types to the MAP processor on every neighborlist build. As mentioned in the first chapter, LAMMPS uses neighborlists to track pairs of atoms that interact with each other and the size of the neighborlist varies with the cut-off distance defined by the user. For the *rhodo* benchmark, an average of 375 neighbors are present for each of the 32000 local atoms, which adds up to approximately 12 million elements in a half neighborlist and 24 million in a full neighborlist (used with *newton* on and *newton* off setting respectively). We pack the neighborlists together in a single integer array `cum_neighlist`, which is of size more than 90 MB for a full neighborlist. The neighborlist is copied to the GCM memory during each neighborlist build and the `MAP_computeForces` function streams this list from the GCM into its compute loop.

4.2.1.3 Avoiding CPU programming idiosyncrasies

Theoretically we can port the LAMMPS C++ function `PairLJCharmmCoulLong::Compute` 'as is' to Carte C (with a few modifications necessary for data transfers and OBM accesses) and execute it on the MAP. However, this would result in inefficient logic mapping that would likely occupy more FPGA resources than are available and/or experience severe loop slow downs as discussed earlier. Applications such as LAMMPS that have been highly optimized for the microprocessors require

S.No	***Array***	***Frequency***	***size***
1	Px	each tStep	nall
2	Py	each tStep	nall
3	Pz	each tStep	nall
4	q	with NList build	nall
5	type	with NList build	nall
6	num_neigh	with NList build	nreal
7	cum_neighlist	with NList build	all_pairs
8	lj1,lj2,lj3,lj4	first tStep	4 x n1 x n1
9	PPPM tables(r,dr, c,dc,e,de,f,df)	first tStep	8 x n2 x n2

Table 4.2: Data transferred from host to the MAP.For Rhodopsin benchmark all_pairs=24 million and nall=80k approximately,n1=69 and nreal=32k.

code modifications and optimizations to better suit the FPGA programming model. We have discussed some design-level changes made to avoid stalls in the compute pipeline, in this section we discuss the code modifications made to enable efficient resource utilization.

Branch controls in microprocessors imply that different sets of instructions must be loaded, decoded and executed depending on which path of the branch is taken. However, in an FPGA, the function as a whole is translated to hardware. Irrespective of the branch outcome, data flows through both branches of the circuitry and a selection process occurs on the resulting value based on the branch control. The `PairLJCharmmCoulLong::compute` function uses `PPPM` table-based calculations to avoid more complex calculations for pairs of atoms that satisfy the cut-off distance for table-based approximation. This method is a good optimization for microprocessors, however on an FPGA, this approximation adds an additional overhead. Apart from introducing additional control logic, which is not ideally suitable for an FPGA architecture, the table-based calculations consume resources in addition to the already existing logic for the non-approximated calculation. We thus modify the code

to better suit the MAP processor architecture by avoiding the table-based approximations and branching conditions wherever possible. For example, we do not check the condition `rsq<cut_bothsq`, to check if a particular pair of atoms needs to pass through the calculations or not, we rather test the condition at the stream source for forces to determine whether to place the force calculated for the pair in the stream or not.

4.3 Implementation

Taking advantage of the High-Level language (HLL) to circuit translator available in the SRC Carte Programming environment, we develop the `MAP_computeForces` function in SRC Carte C (version 3.2). SRC Carte C is very similar to the ANSI C programming language, except for specialized macros for optimization and communication tasks. In this section, we discuss the steps taken to develop a Carte C version of the pairwise non-bonded force computations function. The `pair_style` setting in the LAMMPS input script is used to identify the functions that must be invoked for the pair-wise force computations. For example The Rhodopsin benchmark uses the pair-style *lj/charmm/coul/long*, which invokes `PairLJCharmmCoulLong::compute`, the function that consumes 70% of execution time during profiling. New pair-styles could be easily added to LAMMPS by modifying the `user_style` header file.

We create a new pair style *lj/charmm/coul/long/fpga* identical to *lj/charmm/coul/long* except that the `PairLJCharmmCoulLongFPGA::compute` function used by the former, off-loads the pair-wise force computations to the `MAP_computeForces` function that executes on the MAP processor. We create two Makefiles for LAMMPS Make.intel_MAP_debug and Make.intel_MAP_hw. While

both compile LAMMPS functions to be executed on the GPP with the Intel compiler, the former invokes the MAP debug mode compilation process to compile the `MAP_Computeforces` function and the latter invokes the MAP HW mode compilation, which includes the P&R procedure for creating the FPGA bitstream for the MAP processor. We use the MAP debug mode during initial stages of development to verify functionality and results. LAMMPS data structures are in the form of C++ classes, which cannot be directly used in the Carte C function `MAP_ComputeForces`. Further, all data used by `MAP_ComputeForces` function must be accessible by the MAP processor, in other words, available in one of the three memory locations: FPGA RAM blocks, OBM banks, or the GCM. Thus data in the form of LAMMPS C++ classes are copied to one-dimensional arrays and transferred to one of the three memory locations accessible to the MAP processor. Since the `MAP_ComputeForces` function operates on single precision data, all data transferred to the MAP processor is down-casted to single precision. Though `MAP_ComputeForces` operates on 32-bit data, all data acccessed from the OBM banks will be in the form of 64-bit words, since the OBM banks are 64-bit wide. Thus for efficient data accesses from the OBM banks and to save communication bandwidth between the host and the MAP, we pack pairs of 32-bit words into 64-bit words on the host before they are transferred. We pair the 32-bit words from two different arrays, such that when a 64-bit word is accessed by the MAP processor, both the even and odd words are used. Each of the 64-bit arrays are stored in a separate OBM bank as shown in Figure 4.4. The array consisting of `Px` and `Py` are stored in AL, `Pz` and `q` are stored in AH, the atom types `tp` and the neighborlist size of each of the real atoms, `nn`, are stored in CL. The contents of banks AL and AH are replicated on banks BL and BH and the contents of CL are replicated on bank DL to avoid atoms `i` and `j` competing for the same ports. Atom `i` data is accessed from AL, AH and CL while atom `j` data is accessed from BL, BH

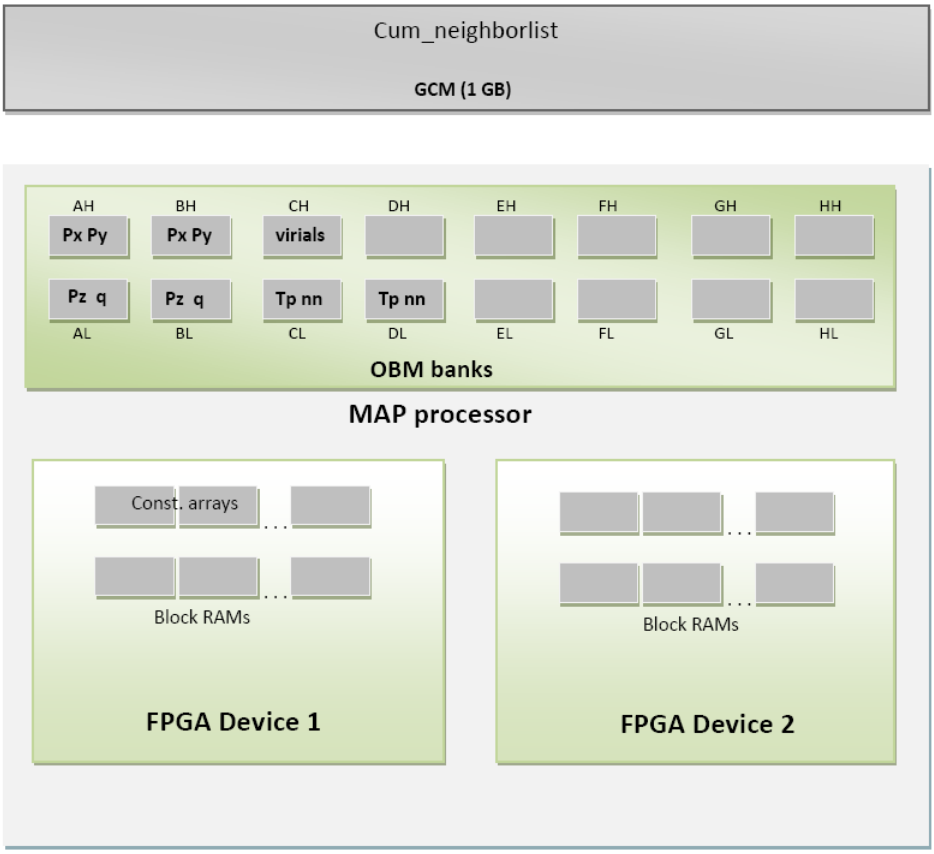

Figure 4.4: Data stored on the OBM, GCM and block RAM

and DL.

To overlap computation with communication, the *compute loop* and the streamed_data transfers are in separate parallel sections as shown in Figure 4.5. There is a parallel section that receives a 64-bit wide incoming stream of cumulative neighborlists and creates a 32-bit wide stream for the compute loop, which will consume one 32-bit integer per clock cycle. The compute loop, which executes in parallel to the data streaming, uses the index from the neighborlist stream to access the OBM bank for the `j` atom data. Since the data is packed together as 64-bit words, we split the data accessed from the OBM banks into two 32-bit floats. The compute loop computes the LJ and Coulombic forces and produces `fpair_x`, `fpair_y`,

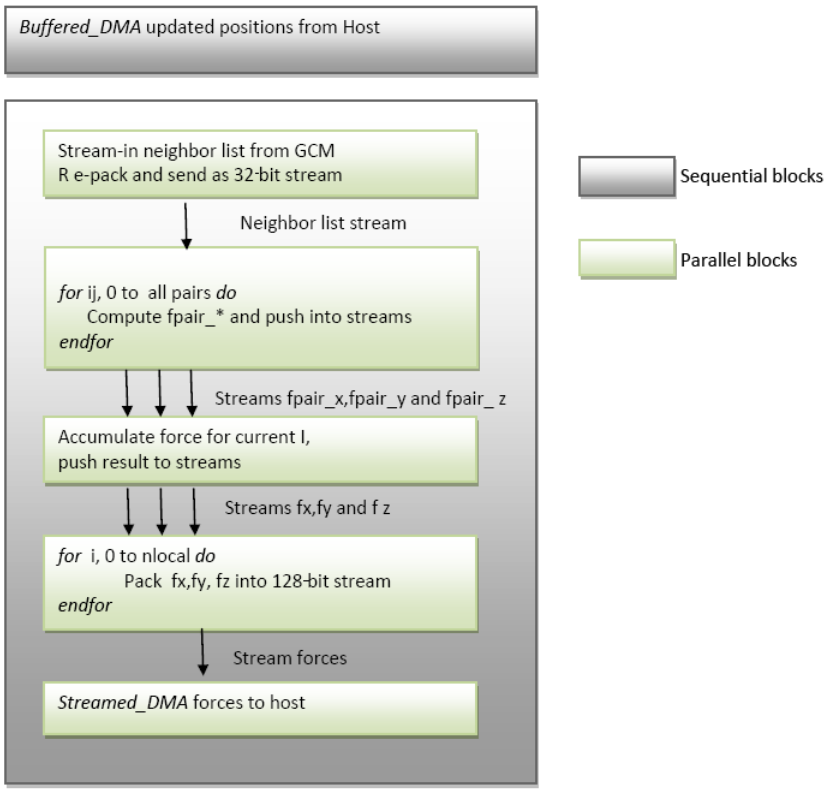

Figure 4.5: The `MAP_computeforces` function

and `fpair_z` for the interacting pair of atoms in each iteration. To avoid stalls in the pipeline caused by the force update on the `i` atoms, as discussed in the design section, we use the stream-based floating-point accumulators available in the Carte macro: `stream_fp_accum_strm_counts_32_rr_term`. This macro internally implements two floating-point adders and thus is capable of taking in a new float input value every clock cycle. Use of this macro allows us to avoid loop delays on the accumulations for `i` atoms.

The result streams from the macros contain the floating-point sums, force_*[i], produced at the end of every neighborlist for a real atom `i`. `force_x`, `force_y`, and `force_z` are received from these streams and packed into a 128-bit wide stream in another parallel section. The 128-bit streams are received in the final parallel section

that streams the data back to the host. Packing the data into 128-bit wide streams is done to save bandwidth during the data transfers between the MAP processor and the host. The `Pair::ev_tally` function is in-lined in the compute loop to avoid function calling overheads and reduce data transfers between the host and the MAP processor as explained in Section 4.1.

Single precision floating-point accumulators were initially used to perform the virial summation over the 12 million iterations. However, the energy values vary by more than 10^{-2} Kcal right from time-step 0, when compared to double precision execution on the Xeon processor. Analyzing the `virial` values being accumulated, we identified the source of the discrepancy as the lower precision used for the `virial` summation. Thus to improve accuracy, we use 64-bit floating-point macros for the `virial` term summations. The resultant implementation has an acceptable deviation (less than 1%) from the host-only execution results. The accuracy of the implementation will be discussed in further detail in Section 5.1

The accumulated results for `virial`[0 to 6] and variables `eng_vdwl` and `eng_coul` are passed back to the host function. All of the results, forces and the `virial`s are assigned to the LAMMPS variables in the host code.

Thus LAMMPS was successfully partitioned to off-load the compute-intensive non-bonded force computations to the reconfigurable hardware, and an optimal design was chosen and implemented. In Chapter 5 we discuss the results in terms of performance and simulation outputs when compared to that of GPP version of LAMMPS.

Chapter 5

Results

In this chapter, we present the experimental results that validate our implementation and discuss the performance improvements achieved by the MAP acceleration of non-bonded force calculations. We also analyze the time consumed for different modules of the MD simulation and their individual speed-ups. Further, based on the experimental results, we review the theoretical performance projections made in Chapter 5.

5.1 Validation

We validate our implementation by comparing the energies computed in mixed-precision by the MAP-accelerated code (SW+HW) with the energies computed in double-precision by the host-only (SW) execution of *rhodo* for 1000 time-steps. The graph in Figure 5.1 shows the total energy of the system plotted for every 50th time-step. As seen in the graph, the total energy in mixed-precision and double-precision executions diverge slightly (less than 1%) after 500 time-steps. This divergence is a result of the approximation errors in the single-precision computation of non-bonded

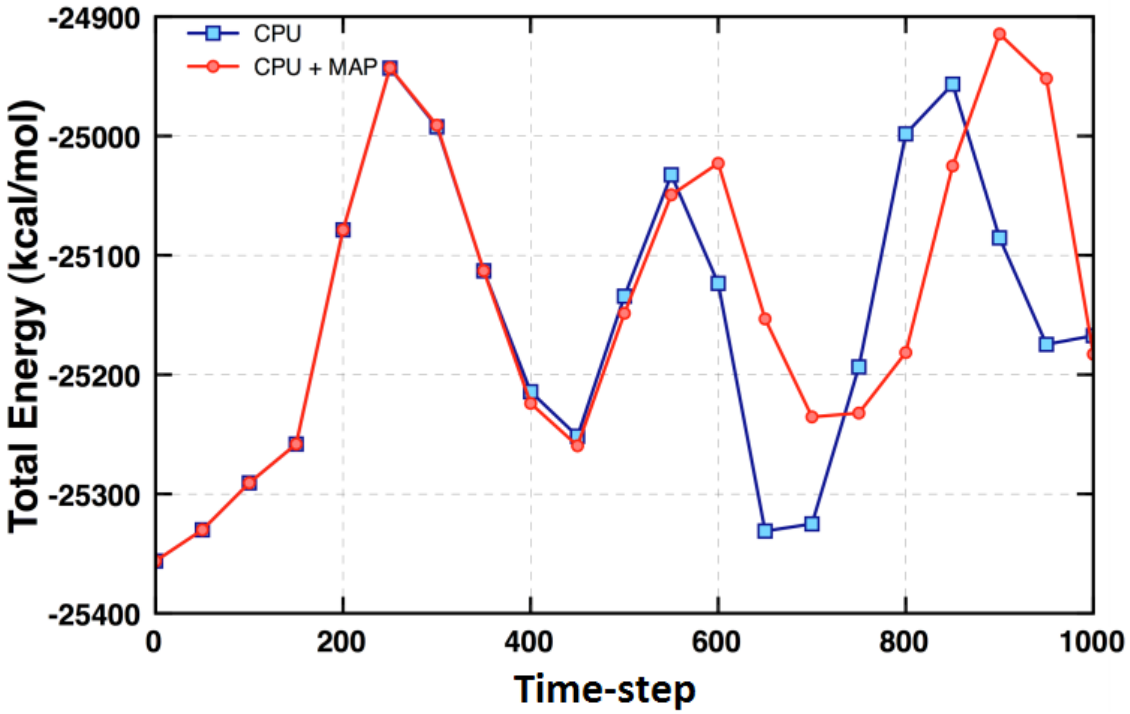

Figure 5.1: The total energy of the system plotted for every 50th step for *rhodo*

forces on the MAP processor. Further, the order of the floating-point operations in the non-bonded force computations differ in the MAP and host implementations, also contributing to the differences in the results.

5.2 Performance

LAMMPS simulation configurations such as the duration of the time-steps, number of time-steps, *newton* (third law) on/off flag, the skin distance used for neighborlist rebuilds, pair-style to be used for force calculations, etc., are set in the LAMMPS input-scripts. We compare our results using three different pair-styles with 1 fs time-step and 2 fs time-step execution of *rhodo* for 100 time-steps. The three pair-styles used are:

- *lj/charmm/coul/long* - SW-only execution with *newton* on and half neighborlist
- *lj/charmm/coul/long/full* - SW-only execution with *newton* off and full neighborlist
- *lj/charmm/coul/long/fpga/full* - SW+HW execution with *newton* off and full neighborlist

The time-to-solution (*overall* time) of *rhodo* and the division in terms of the time spent in non-bonded force computations (*pair* time), neighborlist build, and the remaining modules (*neigh+others* time) for the pair-styles listed above are presented in Table 5.1. Using the pair-styles listed earlier in combination with the *newton* on/off flag settings, full neighborlist and half neighborlist simulations were performed for 1 fs and 2 fs time-steps. Further, the skin distance setting was varied to control the time spent in neighborlist rebuilds. The *pair* time in the SW+HW execution, represents the time spent in the MAP processor computing the non-bonded forces plus the data transfers between the MAP and the host.

Comparing an equal number of computations (i.e. simulations with full neighborlist) on the Xeon (*lj/charmm/coul/long/full*) and the MAP (*lj/charmm/coul/-long/fpga/full*) for 2 fs time-steps and 2 Å skin distance, we are able to accelerate the non-bonded force calculations alone by a factor of 8 while the overall application speed-up is 2.4x. The full neighborlist versions require twice the number of computations to compute the pair-wise forces as the half neighborlist versions. On the FPGA, the force computations with the full neighborlist version performs better than the half neighborlist implementation on the FPGA as explained in Section 4.2.1. However on the host, the SW implementation of non-bonded force computations with a full neighborlist takes twice the time as the non-bonded force computations with a

Simulation settings		Execution time (s)					
		SW			SW+HW		
newton	Skin	*pair*	*neigh +others*	Overall	*pair*	*neigh +others*	Overall
Time-step = 2 fs							
ON	2.0 Å	99.48	44.52	**142.96**	-	-	-
OFF	2.0 Å	160.07	50.85	210.92	19.97	69.48	89.45
OFF	3.0 Å	-	-	-	23.28	48.95	72.23
OFF	4.0 Å	-	-	-	27.95	41.41	**69.36**
Time-step = 1 fs							
ON	2.0 Å	101.25	30.89	**132.14**	-	-	-
OFF	2.0 Å	335.78	42.4	378.18	18.59	41.48	60.17
OFF	3.0 Å	-	-	-	22.70	36.61	59.31
OFF	4.0 Å	-	-	-	27.26	27.64	**54.90**

Table 5.1: Performance comparisons of 100 time-step execution of *rhodo* (1 fs and 2 fs time-steps)

half neighborlist. Thus a more realistic performance comparison uses a full neighborlist execution in the SW+HW mode (*lj/charmm/coul/long/full/fpga*) and a half neighborlist execution in the SW-only mode (*lj/charmm/coul/long*). The *pair* time of 19.97 s in the SW+HW mode is about a 5x speed-up when compared to the 99.48 s in the SW mode. The performance improvement achievable through deep pipelining the force computations in the FPGA is evident from the fact that the time taken by the MAP processor to compute non-bonded forces for about 24 million pairs is 1/5th the time taken by the Xeon processor to perform non-bonded force computations for about 12 million pairs. However, in spite of the 5x speed-up of the non-bonded force computations, the overall time-to-solution has improved by only 1.6x for the *rhodo* benchmark with 2 fs time-steps.

In Section 4.2.1, the estimated overall speed-up of 1.7x over the SW execution with *newton* on and 2 Å skin distance, was based on an estimated speed-up of 6.0x over *pair* time. We estimated an average of 173 ms/time-step for the non-bonded

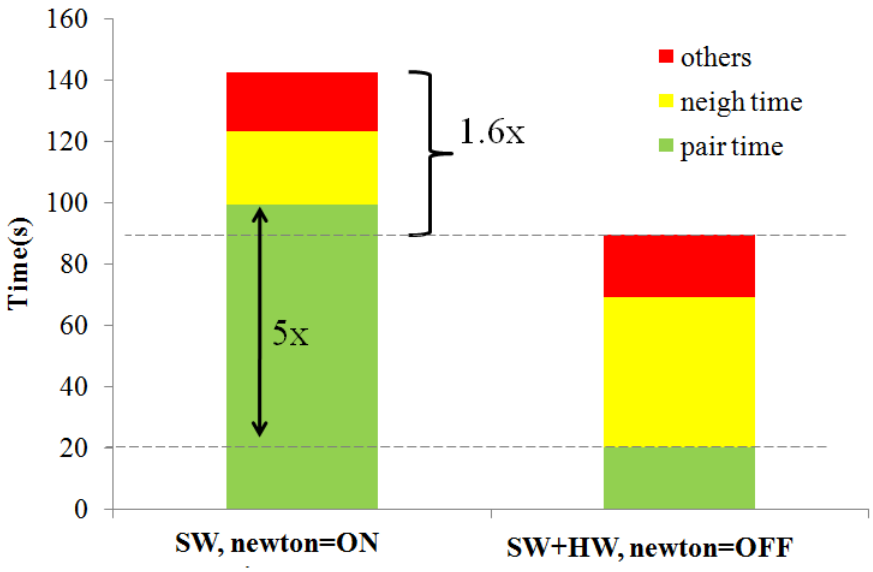

Figure 5.2: Performance of *rhodo* in SW and SW+HW modes with 2 fs time-steps and 2 Å skin distance

force computations on the MAP processor, but the actual measurements show 200 ms/time-step spent in the non-bonded force computations. The difference is due to the memory allocation and data copy overheads that were not included in the estimations and the MAP function calling latency, which is about 10 ms per call. Thus, with the experimental result of 5x speed-up of *pair* time, we achieved an overall speed-up of 1.6x over SW-only execution of *rhodo* with 2 fs time-step and *newton* on setting.

Figure 5.2 shows the performance comparison of *rhodo* execution with 2 fs time-steps and 2 Å skin distance in SW mode and SW+HW mode. The graph clearly shows that the overall performance gain is limited by the increased time spent in the full neighborlist build, which typically increases by a factor of 2 over a half neighborlist generation. LAMMPS performs a neighborlist rebuild only when some neighbor atom has moved more than half the skin distance. Thus increasing the skin distance from the default value of 2.0 Å, decreases the number of neighborlist builds

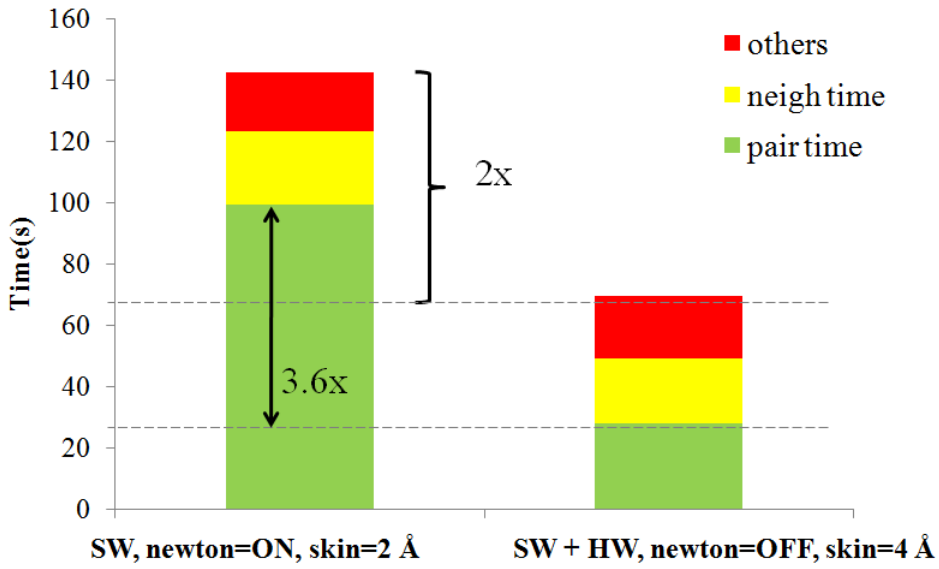

Figure 5.3: Performance comparison in terms of best execution times in SW mode and SW+HW modes for *rhodo* (2 fs time-steps)

during the simulation and therefore the time spent in the neighborlist builds (*neigh* time). For example, for 100 time-steps simulation of *rhodo* with time-step = 2 fs and skin distance = 2 Å, 11 neighborlist rebuilds occur. However, when the skin distance is increased to 3 Å and 4 Å, the number of neighborlist rebuilds decreases to 5 and 3 respectively. As shown in Table 5.1, with fewer number of neighborlist rebuilds the *neigh+others* time in the SW+HW mode decreases to 48.95 s and 41.41 s for the skin distances of 3 and 4 Å respectively and therefore the overall execution time decreases to 72.23 s and 69.36 s. The moderate increase in the *pair* time in the SW+HW execution with the increase in skin distance is due to the increased size of the neighborlist with an increase in skin distance. The increase in neighborlist size increases the number of iterations in the non-bonded forces *compute loop* and therefore increasing the *pair* time.

To make a fair performance comparison, we compare the execution time of the SW simulation with optimal settings for the SW code (*newton on*, 2 Å skin

distance) to the execution time of the SW+HW simulation with optimal settings for the SW+HW code (*newton off*, 4 Å skin distance). The best SW execution time and the best SW+HW execution time are 142.96 s and 69.36 s respectively (shown in bold face in Table 5.1). Thus, for 2 fs time-step simulations of *rhodo*, the performance gain using SW+HW mode is 2x (142.96 s/69.36 s) as shown in Figure 5.3.

The fewer neighborlist rebuilds required for the 1 fs time-step simulations results in a moderate performance gain, when compared to the 2 fs time-step simulation. For example, during 100 time-step simulations with 2 Å, 3 Å, and 4 Å skin distances, the 1 fs time-step simulation requires only 5, 3, and 1 neighborlist rebuilds respectively, whereas, the 2 fs time-step simulation requires 11, 5, and 3 neighborlist rebuilds respectively. Thus, the 1 fs time-step simulation has a smaller *neigh+others* time, resulting in a better overall performance gain of 2.4x when compared to the 2.0x performance gain with the 2 fs time-step simulation.

5.3 FPGA Resource Utilization

Initial implementation prior to the modifications discussed in Section 4.2.1.3, exceeded the logic capacity available on a single Altera Stratix II device by about 20%. After performing the modifications to remove unnecessary code (branching conditions and the table-based interpolation), we were able to map the `MAP_computeforces` function on a single FPGA. We also had room to use double-precision accumulator macros instead of single-precision accumulators, for the summation of `Virial` terms to improve accuracy (discussed in Section 4.3). The final resource utilization is as shown in Table 5.2. The logic utilization is 99% and the total registers utilized is 91%, leaving minimal room for any further computations on the first FPGA. The resource utilization could be reduced with more rigorous optimization steps such as

using fixed-point arithmetic rather than floating-point. We plan such optimizations for future implementations.

Resource	***Used***	***Available***	***Utilization***
Logic Utilization (ALUTs & Register pairs)	142,019	143,520	99%
Total registers	136,496	150,386	91%
M512s	327	930	35%
M4Ks	441	768	57%
M-RAMs	0	9	0%
Total block memory bits	1,634,584	9,383,040	17%
DSP block 9-bit elements	546	768	71%

Table 5.2: FPGA resource utilization

In this chapter we presented the results of our implementation. We compared the energy outputs of the SW+HW implementation with that of the SW version, and showed that the accuracy of the mixed-precision implementation is within acceptable deviation. Further, we reviewed the performance gain on the non-bonded force computations and its impact on the overall application performance. The profile results for the accelerated LAMMPS and the FPGA resource utilization data presented in this chapter are useful to plan the future directions of this research. In the final chapter we will summarize the conclusions of this research and future directions we have planned.

Chapter 6

Conclusions and Future Work

This research focussed on accelerating biomolecular simulations with an FPGA-based reconfigurable computer. We identified the bottleneck in the simulation as the non-bonded force computations and by porting these computations to the FPGA hardware, we successfully accelerated them by about 5x for *rhodo*. The 2x increase in the neighborlist build time, due to the full neighborlist required for the FPGA-accelerated version of LAMMPS, decreased the overall performance gain to 1.6x for 2 fs time-steps and 2.2x for 1 fs time-steps *rhodo* simulations with the default skin distance of 2Å. Increasing the skin distance for the SW+HW executions, decreased the number of neighborlist rebuilds and improved the overall application speed-up to 2.0x for 2 fs time-step simulation and 2.4x for 1 fs time-step simulation. These results and the effect on performance due to simulation configuration changes in the LAMMPS input-script were discussed in Chapter 5.

Accelerating an application with an FPGA-based reconfigurable system involves significant effort in terms of development, testing and code-optimization. Further, the development process may be time-consuming due to the lengthy compilation process. For example, for compute-intensive codes such as the non-bonded force

computations, the P&R process takes 6 to 8 hours to complete during hardware compilation. Thus, to avoid wasting effort in sub-optimal designs and repeating the entire cycle, it is useful to perform a thorough design-space exploration prior to the implementation stage. In Chapter 4, we provided theoretical performance estimations for the various design-options considered and a discussion on the choice of design for implementation. Thus, this thesis is an example of an efficient design and development procedure for FPGA-based acceleration.

To our knowledge this is the first fully integrated and fully functional FPGA-based acceleration of LAMMPS simulator. Thus, the detailed description in Chapter 4 regarding the design options and implementation methodology will be useful for future LAMMPS implementations on similar FPGA-based reconfigurable systems. There are several challenges involved in successfully accelerating an application using an FPGA: the limited FPGA resources for the logic implementation, the limited number of On Board Memory banks, and the limited number of parallel sections/pipelines that can be implemented on the FPGA. Further, due to the single-ported nature of the OBM banks, the data storage and accesses must be carefully planned to avoid extra clock cycle penalties when accessing data from the same bank. This hazard is particularly challenging when porting a data-intensive function such as the non-bonded forces computations function in MD. In Chapter 4, we also discussed optimization techniques used to reduce the FPGA resource utilization while retaining the required computational accuracy. In any accelerator based implementation, the data transfer overheads between the host and the accelerator must be minimized and overlapped with the computations. Our discussion of the designs and the implementation used in this thesis provide an overview of the methodology necessary to minimize data transfer overheads.

6.1 Future work

We have successfully accelerated the non-bonded force computations to an extent that it is no longer the dominating function. There are a number of future directions for this research including accelerating the currently dominant neighborlist build function on the second FPGA. We showed the theoretical estimation of an un-optimized design of FPGA-based neighborlist build function in Section 4.2.1. The design had four pipelines and was estimated to be 6x slower than the half neighborlist build on the host (i.e. about 3x slower than a full neighborlist build function on the host). The design could be optimized in the future to reduce the number of iterations in the pipelines. The SRC-7 H MAP used in this research consists of 16 logical OBM banks, thus allowing 16 concurrent OBM accesses without any stalls in the pipeline. With an FPGA-based system that would allow more concurrent data accesses (i.e. more OBM banks), it will be possible to implement more concurrent neighborlist computation (pipelines) and thus achieve better performance. Another potential research direction is to use fixed-point arithmetic instead of floating-point and reduce the FPGA resource utilization, which may allow us to implement additional force computation pipelines in the Stratix II FPGA device.

Further, only one of the two cores available on the host is used in the current implementation. In the future, we plan to implement a threaded version of the FPGA-accelerated LAMMPS that will make use of both the cores available on the dual-core Xeon host and both the FPGAs available on the MAP processor. Such an implementation will be the most resource-efficient implementation on a single SRC-7 H MAP reconfigurable system. We then plan to extend this work to a cluster of MAPstations and study the effect of FPGA acceleration on the parallel execution of LAMMPS.

Bibliography

[1] H. Kai and B. A. Faye, *Computer Architecture and parallel processing*. McGraw-Hill Book Company, 1984.

[2] C. H. Crawford, P. Henning, M. Kistler, and C. Wright, "Accelerating computing with the cell broadband engine processor," in *CF '08: Proceedings of the 5th conference on Computing frontiers*, (New York, NY, USA), pp. 3–12, ACM, 2008.

[3] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, and R. J. Woods, "The AMBER biomolecular simulation programs.," *Journal of computational chemistry*, vol. 26, pp. 1668–1688, December 2005.

[4] S. R. Alam, P. K. Agarwal, M. C. Smith, J. S. Vetter, and D. Caliga, "Using FPGA devices to accelerate biomolecular simulations," *Computer*, vol. 40, pp. 66–73, March 2007.

[5] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kale, "NAMD: Biomolecular simulation on thousands of processors," in *Proceedings of SC 2002*, pp. 1–18, 2002.

[6] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, pp. 2618–2640, September 2007.

[7] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow, "Reconfigurable molecular dynamics simulator," in *Proceedings of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004.*, vol. 0, (Los Alamitos, CA, USA), pp. 197–206, IEEE Computer Society, April 2004.

[8] "The Transmogrifier-3," [Online]. http://www.eecg.utoronto.ca/~tm3/. [Accessed:March 2010].

[9] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones potentials and forces with reconfigurable hardware," in *International Conference on Engineering of Reconfigurable Systems and Algorithm*, 2004.

[10] "Xilinx ML300," [Online]. http://www.xilinx.com/products/boards/ml300/," [Accessed:March 2010]

[11] Y. Gu, T. VanCourt, and M. C. Herbordt, "Accelerating molecular dynamics simulations with configurable circuits," in *International Conference on Field Programmable Logic and Applications, 2005.*, vol. 0, (Los Alamitos, CA, USA), pp. 475–480, IEEE Computer Society, August 2005.

[12] "Wildstar II pro," [Online]. http://www.annapmicro.com/wsiipace.html," March 2010.

[13] R. Scrofano, M. B. Gokhale, F. Trouw, and V. K. Prasanna, "Accelerating molecular dynamics simulations with reconfigurable computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 6, pp. 764–778, 2008.

[14] Y. Gu and M. C. Herbordt, "High performance molecular dynamics simulationswith FPGA coprocessors," in *RSSI '07: Proceedings of the Reconfigurable Systems Summer Institute*, 2007.

[15] T. Matthey, T. Cickovski, S. Hampton, A. Ko, Q. Ma, M. Nyerges, T. Raeder, T. Slabach, and J. A. Izaguirre, "Protomol, an object-oriented framework for prototyping novel algorithms for molecular dynamics," *ACM Trans. Math. Softw.*, vol. 30, pp. 237–265, September 2004.

[16] D. S. Poznanovic, "Application development on the src computers, inc. systems," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, (Washington, DC, USA), IEEE Computer Society, 2005.

[17] *SRC-7 Carte v3.2 C Programming Environment Guide*, June 2009.

[18] V. Kindratenko and D. Pointer, "A case study in porting a production scientific supercomputing application to a reconfigurable computer," in *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), pp. 13–22, IEEE Computer Society, April 2006.

[19] P. M. Martin, "Acceleration methodology for the implementation of scientific application on reconfigurable hardware," Master's thesis, Clemson University, May 2009.

[20] "Xtremedata Inc," [Online]. http://www.xtremedata.com, [Accessed:March 2010].

[21] "Impulse-C," [Online]. http://www.impulseaccelerated.com, [Accessed:March 2010].

[22] "Introduction to the SRC-7 mapstation," November 2007.

[23] J. Grotendorst and D. Marx, *Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms, Lecture Notes*, vol. 10. John von Neumann Institute for Computing, 2002.

[24] M. Crowley, T. Darden, T. Cheatham, and D. Deerfield, "Adventures in improving the scaling and accuracy of a parallel molecular dynamics program," *Journal of Supercomputing*, vol. 11, pp. 255–278, November 1997.

[25] "LAMMPS home page," [Online]. http://lammps.sandia.gov, [Accessed:March 2010].

[26] "The AMBER molecular dynamics package," [Online]. http://amber.scripps.edu, [Accessed:March 2010].

[27] S. Plimpton, R. Pollock, and M. Stevens, "Particle-mesh ewald and rRespa for parallel molecular dynamics simulations," in *In Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[28] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel processing in a restructurable computer system," *Electronic Computers, IEEE Transactions on*, pp. 747–755, December 2006.

[29] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, pp. 171–210, June 2002.

[30] "Stratix II device handbook, volume 1.," [Online]. http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf, [Accessed:April 2010].

[31] S. R. Alam, J. S. Vetter, and M. C. Smith, "An application specific memory characterization technique for co-processor accelerators," pp. 353–358, March 2008.