5-2009

Acceleration Methodology for the Implementation of Scientific Applications on Reconfigurable Hardware

Phillip Martin
Clemson University, pmmarti@clemson.edu

ACCELERATION METHODOLOGY FOR THE IMPLEMENTATION OF SCIENTIFIC APPLICATION ON RECONFIGURABLE HARDWARE

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
Phillip Murray Martin
May 2009

Accepted by:
Dr. Melissa Smith, Committee Chair
Dr. Richard Brooks
Dr. Walter Ligon

ABSTRACT

The role of heterogeneous multi-core architectures in the industrial and scientific computing community is expanding. For researchers to increase the performance of complex applications, a multifaceted approach is needed to utilize emerging reconfigurable computing (RC) architectures. First, the method for accelerating applications must provide flexible solutions for fully utilizing key architecture traits across platforms. Secondly, the approach needs to be readily accessible to application scientists. A recent trend toward emerging disruptive architectures is an important signal that fundamental limitations in traditional high performance computing (HPC) are limiting break through research. To respond to these challenges, scientists are under pressure to identify new programming methodologies and elements in platform architectures that will translate into enhanced program efficacy.

Reconfigurable computing (RC) allows the implementation of almost any computer architecture trait, but identifying which traits work best for numerous scientific problem domains is difficult. However, by leveraging the existing underlying framework available in field programmable gate arrays (FPGAs), it is possible to build a method for utilizing RC traits for accelerating scientific applications. By contrasting both hardware and software changes, RC platforms afford developers the ability to examine various architecture characteristics to find those best suited for production-level scientific applications. The flexibility afforded by FPGAs allow these characteristics to then be extrapolated to heterogeneous, multi-core and general-purpose computing on graphics processing units (GP-GPU) HPC platforms. Additionally by coupling high-level

languages (HLL) with reconfigurable hardware, relevance to a wider industrial and scientific population is achieved.

To provide these advancements to the scientific community we examine the acceleration of a scientific application on a RC platform. By leveraging the flexibility provided by FPGAs we develop a methodology that removes computational loads from host systems and internalizes portions of communication with the aim of reducing fiscal costs through the reduction of physical compute nodes required to achieve the same runtime performance. Using this methodology an improvement in application performance is shown to be possible without requiring hand implementation of HLL code in a hardware description language (HDL)

A review of recent literature demonstrates the challenge of developing a platform-independent flexible solution that allows access to cutting edge RC hardware for application scientists. To address this challenge we propose a structured methodology that begins with examination of the application's profile, computations, and communications and utilizes tools to assist the developer in making partitioning and optimization decisions. Through experimental results, we will analyze the computational requirements, describe the simulated and actual accelerated application implementation, and finally describe problems encountered during development. Using this proposed method, a 3x speedup is possible over the entire accelerated target application. Lastly we discuss possible future work including further potential optimizations of the application to improve this process and project the anticipated benefits.

DEDICATION

I dedicate this to my mom Murray Martin and to everyone who helped along the way.

ACKNOWLEDGMENTS

Special thanks to: XtremeData for donating the development system to Clemson University under their university partners program, the Computational Sciences and Mathematics division at Oak Ridge National Laboratory and the University of Tennessee at Knoxville for sponsoring the summer research at Oak Ridge National Laboratory that lead to this paper, Pratul Agarwal, Sadaf Alam, and Melissa Smith for their involvement with the research.

TABLE OF CONTENTS

Table of Contents (Continued)

Page

LIST OF TABLES

LIST OF EQUATIONS

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

Computer simulations are used extensively to accurately reproduce the process of interest for the purpose of quantifying costs and benefits. Through the analysis of different parameters and their effect on the recreated process, real world problems can be explored. Weather, chemical, atomic, and biological processes are all areas that make extensive use of computer simulations to develop new findings. The results from these fields are, however, bound by two universal factors of computer simulation: effort expended to create an efficient vs. accurate simulation model and the computational power available to execute the simulation.

Historically, traditional computing solutions have aimed to leverage large-scale distributed environments to boost computational power. This technique has in turn led to the development of more complex and accurate models. As the model's complexity grows, the communication time needed in these distributed systems typically multiplies. The inability to scale problems on these large-scale distributed platforms becomes a critical impediment for new discoveries. To overcome this barrier, many industry vendors are introducing heterogeneous platforms which pair traditional HPC hardware with emerging non-RC architectures such as the Cell Broadband Engine™ and general-purpose graphics processing units (GP-GPU) computing with Nvidia's Tesla™ products. Cell and GP-GPU architectures provide the a path to performance through on the use of many-core. While the many-core approach does provide increased compute power and internalized communication, a many-core approach is not an application specific solution.

The additional computational power may be underutilized since the underlying architecture cannot be modified to specifically match the application. When the right applications are matched to these architectures, they provided a very powerful computing platform as demonstrated by Roadrunner, the world's number one supercomputer as of November 2008 is a heterogeneous platform combining AMD Opteron™ processors with CellBE processors (Top500, Nov. 2008).

Another class of hybrid computing platforms that are both general purpose (can be used on a wide variety of applications) and application specific (can be tailored specifically for an application to achieve the best performance) is heterogeneous reconfigurable computing. Over forty years since reconfigurable hardware was first proposed, (Estrin and Turn, 1963), advancements in logic density and the availability of hardware floating-point macros for reconfigurable platforms have garnered attention from the scientific community. RC platforms with FPGAs are essentially an extreme form of heterogeneous computing. The main difference between fixed multi-core (FMC) or traditional homogeneous computing and FPGA implementations is that the underlying architecture is not fixed. FPGAs allow the user to define the application-specific architecture for solving problems in the hardware. Allowing the problem to guide the underlying architecture is extremely efficient in terms of utilization and computational density as only elements pertinent to the processing of the problem are included in the design. The affect is a reduction in energy usage, space use, and often improved communication versus a general-purpose processor.

The abilities of an Application Specific Integrated Circuit (ASIC) parallel that of a FPGA. While an ASIC has similar efficiency as an FPGA, it is usually cheaper in large quantities and slightly faster than a field programmable device since it does not have the extra routing overhead present in FPGA devices. However, at the time of manufacture an ASIC's design is fixed which restricts its use requiring the user to change the design, develop and manufacture a new ASIC for new features or computations. For example, a custom ASIC for assisting in simulating supernova most likely will not be useful to a simulation involving weather forecasting. Thus the reconfigurable nature of a FPGA more then makes up for the slight performance tradeoff. Further, currently available FPGAs provide capacities that are necessary for the computationally dense and complex simulations currently conducted in many fields of research.

Biomolecular simulation is one area that is leading the advancements in computational biology. The fundamental approach for most biomolecular simulators is the use of Molecular Dynamics (MD). MD is a method for treating atoms as points with both mass and charge thereby allowing the use of classical mechanics (IBM Corp., 2006) to simulate the process. The forces on a single atom are split into two categories: bonded and non-bonded interactions. The bonded interactions refer to the forces resulting from the chemical bonds between the atoms in question. Non-bonded forces consist of the electrostatic and Lennard-Jones potentials of the atoms. The charge and mass along with the force of any bonds, which includes bond angles and bond torsions, are feed into the equation of motion to solve for the trajectory of each atom over an extremely small unit of time (Alam, et al, 2007; IBM Corp., 2006). Predicting the behavior of these atoms

requires a large number of force calculations that can be summarized as shown in the overall potential energy function shown in equation 1.1:

$$E(potential) = \sum_{bonds} f(bond) + \sum_{angles} f(angle) + \sum_{torsions} f(torsion) + \sum_{i=1}^{N}\sum_{j<i}^{N}\left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}}\right) + \sum_{i=1}^{N}\sum_{j<i}^{N}\left(\frac{q_i q_j}{\varepsilon r_{ij}}\right)$$

Equation 1.1: Potential Energy function used in computing particle trajectories (Alam, et. al, 2007)

The first three chemical bond terms are constant throughout the simulation as the number of bonds is kept constant (Alam, et. al, 2007). The latter two terms are the summations of the van der Waals and electrostatic forces. These non-bonded terms constitute a more significant portion of the computations than the bonded terms since the number of atoms increases because the non-bonded terms are calculated between all other atoms. This results in an $O[N^2]$computations for a simulation with N atoms. Since all atoms must communicate their current position to each other for the calculation of these non-bonded interactions, scaling becomes a significant problem for large sets of atoms.

To overcome such challenges MD software packages typically include a ‘cutoff’ distance for non-bonded interactions allowing the users to control the complexity and to improve algorithm parallelization (or performance) in traditional large-scale HPC environments. This cutoff value is chosen at the discretion of the investigating scientist to balance execution time with simulation accuracy. The accuracy achieved through the selection of the cutoff value is problem dependent. A larger cutoff value results in a longer but more accurate simulation since an infinite cutoff would result in the ideal electrostatic force calculation from (Alam, et. al, 2007). Further, the cutoff value not only

determines the number of non-bonded computations, it also establishes the amount of required communications for a parallel implementation since an atom must exchange the distance and position of all other atoms within the cutoff distance.

Several custom computing projects, such as Blue Gene/L, Folding@Home, MD-GRAPE, and others (Bader, 2004), were developed with the aim of improving the performance of comprehensive MD simulations. However, MD-Grape and Folding@Home are more application specific solutions and are not versatile enough to be used in different problem domains. Blue Gene/L, on the other hand is more versatile but weakly scales for problems that are not easily segmented into smaller sub-problems. While achievements for MD simulations have been significant, all the platforms still suffer from the basic substantial communication requirements of particle interactions (Sandia National Laboratory, 2006; IBM Corp., 2006; Reid and Smith, 2005). These requirements for numerous particle interactions, which are dominated by global communication, have previously made MD simulation a difficult candidate for application acceleration. Early studies of MD simulations on reconfigurable computing platforms however, have demonstrated the performance potential of this class of systems.

NAMD, a MD simulator similar to LAMMPS, was ported to the SRC-6 platform by Kindratenko and Pointer (Kindratenko and Pointer, 2006). In this paper the authors use profiling to perform an analysis on the NAMD code and identify a specific function that is appropriate for hardware acceleration. The function is then ported using SRC's MAP C development tool to perform assisted C to HDL translation. These implementation steps are similar to the methods and research presented here, however,

the disadvantage of using the MAP C development tool is that it locks the user to a particular platform, the SRC-MAPstations.

Scrofano also presents the acceleration of a MD simulation on a SRC MAPstation (Scrofano, et. al, 2006). The focus here is on partitioning the application between hardware and software. By correctly mapping certain tasks to the software and FPGA hardware a 2x speedup is achievable. In choosing to keep at least some calculations in software Scrofano is able to preserve the ability to flexibly add and remove tasks. The main drawback of this work in comparison to the work presented here, is the choice to develop and use a custom MD kernel that may not be amenable to applications in widespread use by the scientific community.

Herbordt and Vancourt present a more focused view on the use of specialized MD techniques that can be implemented to extract higher performance from FPGAs (Herbordt and VanCourt, 2007). The twelve methods presented in the paper underscore the need for development of hardware code that is portable across platforms while maintaining acceleration for a family of software instead of more targeted, specialized approaches. These key points were an inspiration for implementing the two large communication buffers used in this research for shared memory to help hide signaling overhead.

To address these limitations a flexible methodology is proposed for leveraging recent advances in RC platforms and software development environments to accelerate scientific applications. By using FPGAs to remove computational loads from the host systems, we propose to redirect large portions of communication currently on the

network to internal buses such as the AMD's HyperTransport™ bus. The additional computational power per node will also result in a reduced number of physical compute nodes required to achieve the same runtime performance, which leads to other cost and power savings. Furthermore, the use of HLL languages for development is emphasized as a means to allow application scientists to utilize the performance of cutting-edge RC platforms.

We have shown that there is a need for studying and developing a method for flexible implementation of a scientific application that maintains platform independence. This methodology should address the characteristics (computation and communication profiles) of the targeted application and utilize appropriate tools for producing a hardware accelerated program that is portable. The next chapter will discuss the LAMMPS software, our chosen hardware platform and the HLL-to-HDL development environment that allows scientists easier access to RC hardware.

CHAPTER TWO

RESEARCH DESIGN AND METHODS

To harness the increased computational power provided by reconfigurable computing (RC) hardware an innovative technique is essential for overcoming the challenge of porting application code written in a high-level language to a hardware description language (HDL). Further, traditional methods such as hand porting required complex modifications to application codes for each potential target platform. These modifications have been a significant hindrance to the adoption of reconfigurable computing architectures. Even preliminary questions such as 'what algorithm would benefit most from porting to an RC platform' and 'how to accurately estimate the performance gain without an actual implementation in hardware' seem daunting when combined with the user-defined nature of FPGAs.

Using a production-level molecular dynamics software package, LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) developed by Sandia National Laboratory (Sandia National Laboratory, 2006) we seek to develop and demonstrate a framework for accelerating scientific applications in RC environments. LAMMPS's prevalence in the computational biology field, well defined mathematical computations, and implementation in the C++ language make it a desirable candidate application for demonstrating the methods used to accelerate this and similar classes of scientific applications.

To measure the performance gain against multiple systems we intend to use the Rhodopsin protein benchmark. In detail the Rhodopsin protein benchmark comprises a simulation of the interactions of 32,000 atoms contained in the Bovine Rhodopsin protein in a solvated lipid bilayer (Sandia National Laboratory, 2007). In simple terms the protein is trapped within a layer of lipid (fat) with water as the solvent surrounding both the top and bottom of the lipid layer. Figure 2.1 shows a ribbon view of the protein. The Rhodopsin protein benchmark is an inbuilt simulation provided with the LAMMPS software as a means for a standard measure of system performance. This benchmark is the most complex of the inbuilt LAMMPS simulations and a more detailed comparison is given in chapter three. Additionally the development team has compiled a list, available at http://lammps.sandia.gov/bench.html, of other traditional HPC platforms in which performance data was collected for comparison.

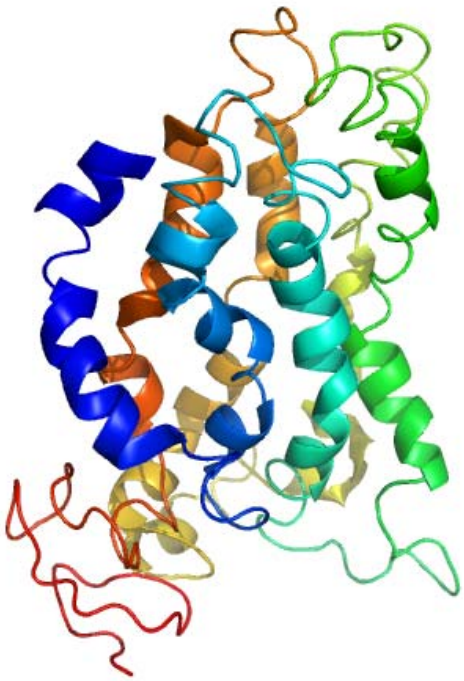

Figure 2.1: Bovine Rhodopsin protein shown in ribbon form with random coloring to better show the alpha helices, the protein does not contain any beta sheets.

In a performance test on the IBM Blue Gene/L, LAMMPS was shown to be the most parallelizable algorithm - scaling relatively efficiently to 4096 processors (IBM Corp., 2006). As figure 2.2 shows, scaling beyond 4096 processors results in the overall communication overhead outweighing the computational benefits – diminishing returns. Overcoming this scaling limitation, present in many of the currently available high-performance computing platforms, is the long-term goal of this research.

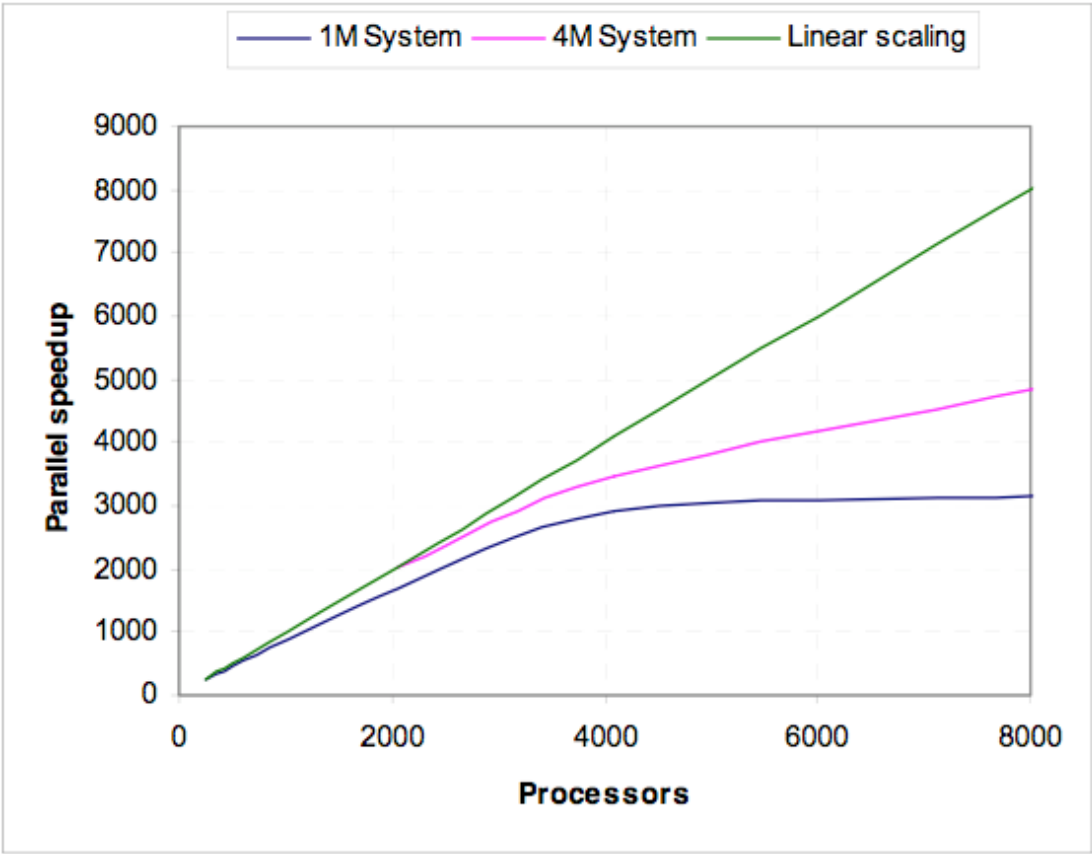

Figure 2.2: Parallel scaling of LAMMPS on Blue Gene (1M System: 1-million atom scaled Rhodopsin protein, 4M System: 4-million atom scaled Rhodopsin protein) (IBM Corp., 2006)

As in the early days of computing, application porting to early RC environments required the entire program functionality to be hand-coded in HDL. This costly development method is still in use today due to the ability to produce the most computationally efficient result with any other available development method. The result is dependent, however, on several factors: how familiar the developer is with the intricacies of both the hardware platform and software to be ported and the developer's proficiency with HDL. Hardware vendors have responded to this challenge with intellectual property (IP) libraries that implement certain specific and sometimes limited functionalities, such as floating-point libraries. These IP libraries however are often black-boxes, their implementation is completely hidden to the application developer. Additionally the IP library is almost always tied to that vendor's hardware making cross platform support difficult at best. These limitations have driven a recent push toward complete tool suites that build upon the IP libraries of each hardware vendor to form a universal SDK for programming RC platforms through the use of HLL abstraction. Of these HLL-to-HDL suites, ImpulseC was chosen for this research due to its support for a number of RC platforms of interest, namely the XtremeData XD1000, DRC DS1000 and Nallatech H101 PCI-X board.

ImpulseC's CoDeveloper tool suite (ImpulseC Corp., 2008) allows programmers to conduct application development in a familiar language, C, without requiring an extensive hardware background or familiarity with obtuse HDL languages. Further, programmers can optionally cross-develop for multiple platforms with minimal changes.

Various project settings control which platform the CoDeveloper tool suite targets through specific generation macros. Fig. 2.3 displays an overview of the development flow within the ImpulseC toolset.

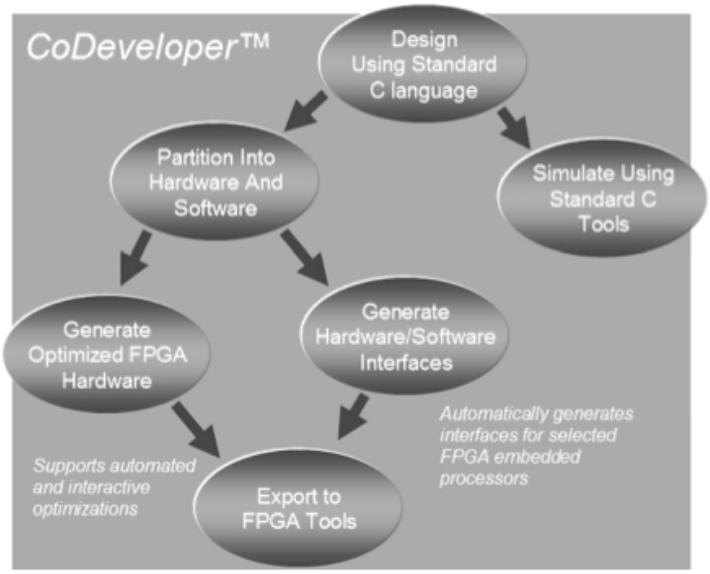

Figure 2.3: ImpulseC Codeveloper tool flow (ImpulseC Corp., 2008)

In the RC development for LAMMPS, which is implemented in C++, we make use of the ImpulseC development environment for easy integration between RC code and existing software portions of the application. After modifying select portions of the original LAMMPS source code with ImpulseC to target the reconfigurable hardware, it is possible to port these portions of the algorithm to multiple hardware platforms. One of our objectives is to examine and document the capabilities of the XD1000 with LAMMPS as a potential platform of study for the scientific community. Later studies will take advantage of the portability of code developed in ImpulseC to target other RC platforms including the DRC DS1000 (DRC Computer Corp., 2008).

The advantage of using a C-to-HDL development method, as (Kilts, 2007) mentions, is that these applications have the ability to compile and run against other C models. More importantly Kilts states that, “One of the primary benefits of C-level design is the ability to simulate hardware and software in the same environment.” In this implementation we extensively use both capabilities to reduce complexity and fast-track the development on new platforms.

The ImpluseC CoDeveloper tool suite includes a C-to-VHDL (or Verilog) compiler and development environment. This compiler permits the creation of communication channels, buffers, and signals through simple function calls from the high-level language (HLL) environment (Pellerin and Thibault, 2005). Effectually, the abstraction gained from using HLL interfaces enables two things. Most importantly the developer is not required to have specific hardware design knowledge to generate results. An additional benefit is the user’s code is now portable since any platform specific code is now hidden below these universal function calls making the functionality transparent to the developer.

The development environment in the tool suite also assists the programmer with system integration and includes several options for debugging and simulating application codes in software for a variety of reconfigurable computing platforms. The built-in simulator’s capabilities include simulating the buffers, communication channels, FPGA hardware, and host program during run-time as well as logging options useful for debugging. In detail the CoDeveloper tool suite supports the integer math functions: addition, subtraction, multiplication, division, and number comparisons. Similar

operations in floating-point are additionally supported to an extent. Issues relating to the extent of implementation surrounding these floating-point operations are addressed in the discussion of the results.

There are two main methods for producing VHDL or Verilog from target code segments in the CoDeveloper tool suite: shared memory or a stream interface approach. A stream interface allows a direct software-to-hardware channel that can be uni- or bi-directional. The main benefit of a stream approach is the simplified signal interface to synchronize producer and consumer functions when accessing data exchanged between the host processor and FPGA. The more complex shared memory approach however usually allows for higher data transfer bandwidth. All reads and writes for shared memory are performed directly to the FPGA's internal BRAM. The drawback with this method is the need for the programmer to explicitly manage the synchronization of the memory accesses in C through the use of signals. While ImpulseC's development tools are able to provide transparent communication, the bandwidth and latency is still determined by the platform hardware.

The target platform is XtremeData Inc.'s XD1000 which has an Altera Stratix II FPGA module that is an AMD Opteron™ replacement (XtremeData Corp., 2007). The ability to place an FPGA module into any open Opteron socket allows the FPGA to leverage the existing cooling, power and communication infrastructure. Further, the ImpulseC SDK is able to take advantage of AMD's HyperTransport™ bus present in the XD1000 system to provide the tightly-coupled communication interface necessary to

improve the scaling of scientific applications. The communication layout of the XD1000 development system is shown in figure 2.4.

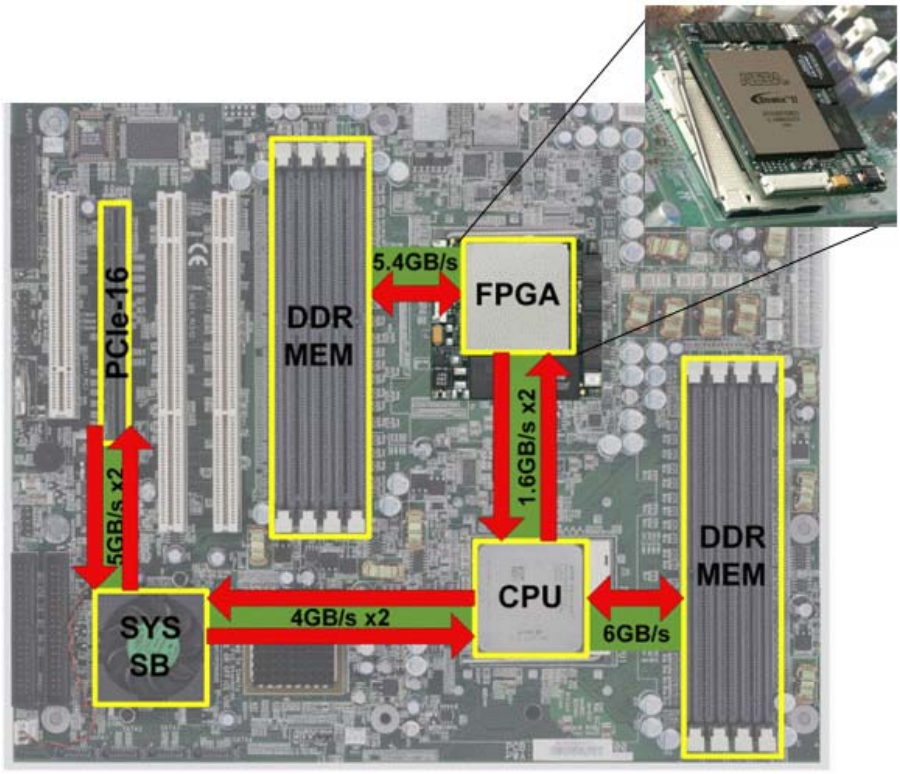

Figure 2.4: XD1000 Development System Communication Interface Bandwidths (XtremeData Corp., 2007)

As shown in figure 2.4, the XD1000 platform allows a developer great flexibility in application porting through the close integration of the FPGA with the memory and host CPU. With the knowledge of the underlying architecture, we can further explore the requirements involved with porting an application. The most challenging part of porting applications to a hardware platform such as the XD1000, is partitioning the problem such that it fits into the logic and communication resources of the given FPGA and platform. The *Stage Master Explorer* tool in the CoDeveloper tool suite can give the developer a rough estimate of the potential hardware speedup before conducting the time consuming

tasks of synthesis and place and route that are required to implement the application in hardware. The Stage Master Explorer tool graphically shows the computations that are preformed in a flow chart layout. From this graphical view, bottlenecks within the code can be easily identified allowing the developer to modify the code and minimize the space and communication costs when porting algorithms to hardware.

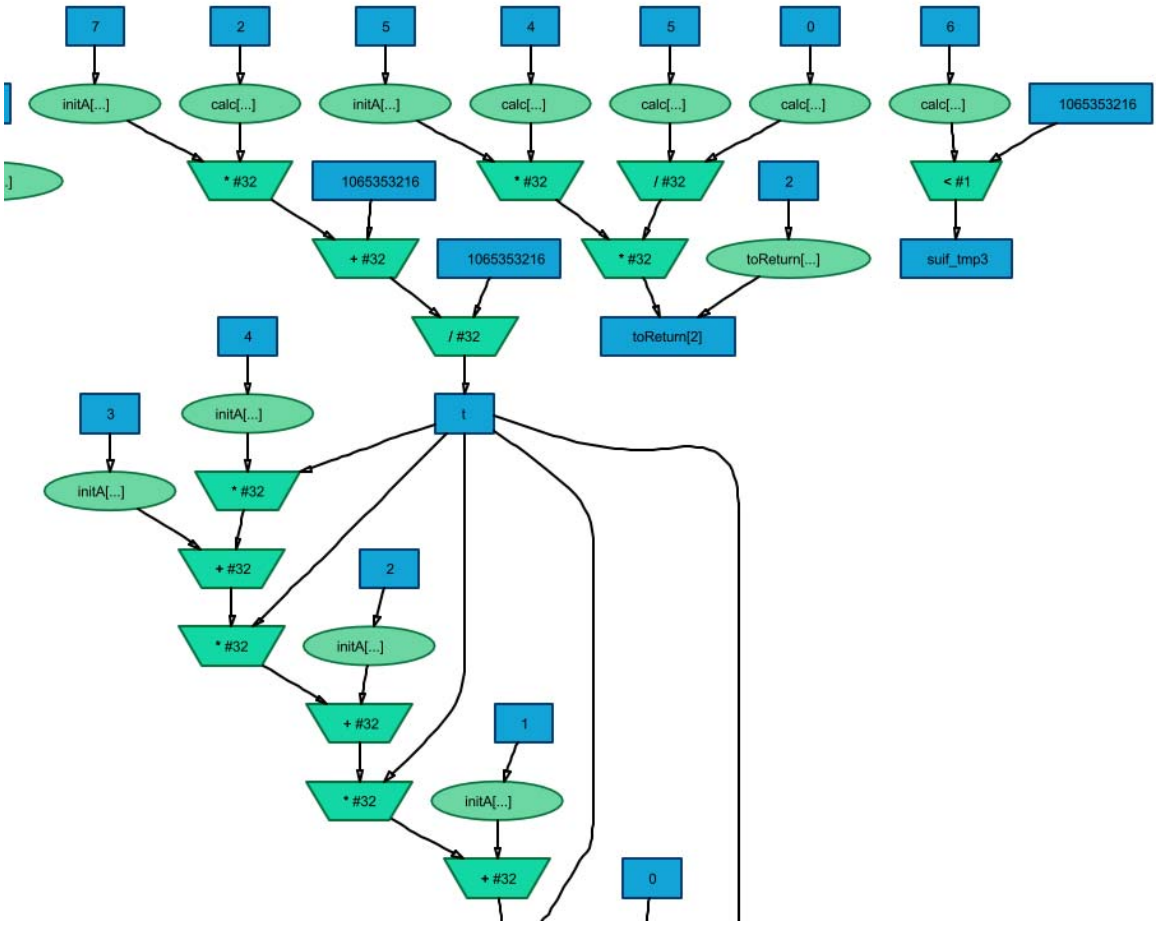

Figure 2.5: A screenshot of an Excerpt of the LAMMPS algorithm in Stage Master Explorer. Square boxes are communication variables, ovals are memory arrays and trapezoids are execution blocks.

Another feature of the Stage Master Explorer is that the number of stages or combinatorial cycles is automatically counted. From this number a developer can determine how many clock cycles that an algorithm may roughly take to complete. There is one caveat however, the stage count neglects memory and communication overheads so these must also be taken into consideration. Figure 2.5 is an excerpt of the main LAMMPS algorithm that was ported. Square boxes represent variables received over the shared memory stored at the index value 0,1,2,3, etc. or are constants. Ovals are BRAM memory locations on the FPGA and trapezoids are execution operations. For example '+#32' denotes a 32bit addition. Stage Master Explorer helps a developer to characterize the datapath of an algorithm and will be used in the next chapter to help characterize the FPGA communication requirements.

With the background knowledge of LAMMPS, ImpulseC, and our choice of the XtremeData XD1000 platform we have laid out the tools we will use to demonstrate the hardware acceleration of a scientific application in the next chapter. Using this knowledge we will inspect the requirements of the application to better match the task to the RC hardware. The experimental results will tie together this knowledge and display a methodology of profiling, porting and the resulting speedup that a general user can achieve.

CHAPTER THREE

EXPERIMENTAL RESULTS

When considering code for application acceleration on reconfigurable computing platforms, it is critical to locate and characterize all communication and memory utilization related to the target code segments. This analysis is key to minimizing data transfer overheads and maximizing performance (Smith, et. al, 2006). Analysis of the LAMMPS application code with profiling tools revealed that the function `pair_lj_charmm_coul_long:compute` consumes approximately 70% of the total execution time when running the Rhodopsin benchmark.

A Comparison of the complexity of each benchmark provided in the LAMMPS code base is given in table 3.1.

Problem:	LJ	Chain	EAM	Chute	Rhodopsin
CPU/atom/step	1.35E-6	6.25E-7	3.62E-6	5.91E-7	2.47E-5
Ratio to LJ:	1.00	0.46	2.69	0.44	18.40

Table 3.1: A summary of single-processor LAMMPS performance in CPU secs per atom per timestep for the 5 benchmark problems (Sandia National Laboratory, 2006)

The Rhodopsin Protein benchmark is the most difficult simulation to run of the group at more than 18 times slower than LJ. As described in the previous section, the expense in computing a large number of pairwise interactions accounts for the significant increase of complexity found in the Rhodopsin benchmark. This time-consuming calculation makes improving the computation of pairwise atomic interactions a desirable candidate for hardware acceleration.

An ideal target for hardware acceleration would have no child functions, repetitive intense computations and a minimal amount of communication. The selected function, `pair_lj_charmm_coul_long:compute` conforms closely to the two characteristics: a relatively small amount of communication versus computations and only one child function. For communication, 16 double-precision floating-point values are passed to the function and consumed by over a hundred 100 floating-point operations, consisting of division, multiplication, and addition/subtraction. This task is then repeated for each atom.

In a traditional parallel implementation of LAMMPS, atoms are divided among the various processors within a computing system. For each atom of the 32,000 present in the Rhodopsin protein benchmark, `pair_lj_charmm_coul_long:compute` must compute the electrostatic and van der waals forces on each atom resulting from all neighboring atoms within a given cutoff distance. This cutoff, chosen at the discretion of the investigating scientist, is used to balance execution time with accuracy and for the purposes of these experiments a cutoff of 10 angstroms will be used. This cutoff is a universal value defined in the benchmark itself and is set for the purpose of allowing comparison between other benchmarked systems. Increasing the cutoff will result in a decline in parallel efficiency (IBM Corp., 2006; Reid and Smith, 2005).

In light of the effect a cutoff has when using multiple processors or multiple nodes in a system, LAMMPS was profiled on a single processor to ascertain a more accurate overview of the structure and computational intensity within the program. This single-node analysis provided a clear picture of the memory requirements necessary in

the RC system implementation as well as where to target hardware implementation. The LAMMPS code was profiled running the Rhodopsin benchmark on a single 1.3 Ghz Power4 processor using Xprofiler. These tests were conducted without exclusive access to the entire machine, thus the background load is present in the results. Statistical runs were therefore conducted and the mean runtime was measured to be 194 seconds for the 32,000-atom benchmark. Within the 194 seconds, a total of 132 seconds (68%) were consumed in the `pair_lj_charmm_coul_long:compute` function. Since this function consumed the largest amount of execution time compared to all other functions it is the prime candidate for hardware implementation. Dividing the total time (132 seconds) by the total number of timesteps (100) yields 1.32 seconds per timestep, which is the time required to compute 32,000 atoms.

Implementation of LAMMPS running the Rhodopsin protein benchmark on the XD1000 development system consisted of decoupling the `pair_lj_charmm_coul_long:compute` function from the original application code and building the interfaces to marshal data between the host code and the FPGA module. The host code running on the Opteron™ processor of the XD1000 consists of the original LAMMPS code minus the `pair_lj_charmm_coul_long:compute` function, plus the software interfaces to the ported function running on the FPGA module. The `pair_lj_charmm_coul_long:compute` function itself was split into an initialization section and a computation section. The initialization section receives the data that is used across the entire timestep through a shared memory interface coded in ImpulseC. The computation section receives each atom's unique data from a second shared memory

interface, calculates it with almost no changes to the ordering and structure of the calculations in the function. The preservation of the structure and order of the function allow easy reference to the original software code as well as reducing the number of modifications needed to port the algorithm. Once the computation is complete, the values are written to the FPGA's internal BRAM where they can be accessed by the host through another shared memory interface. Most of the changes to the function to port it to the FPGA module were communication and memory related, the rest of the structure due to the ability of ImpulseC's HLL development environment to automatically parse and compile the C code into a selected HDL, remains functionally the same.

Stepping through the operation of the ported hardware function, each timestep starts with the receipt of new initial values. These initial values do not change during a given timestep and can be buffered before calculations commence, eliminating repetitive communication. The calculation mode is then initiated on the FPGA as normal execution progresses concurrently on the host side. Currently the hardware implementation loops 64,000 times processing the same data repeatedly that is given at runtime for the purposes of gathering implementation timing. The FPGA does not communicate any results to the host during this loop but does write to internal BRAM after each atom calculation. The host program receives a signal after the completion of the entire loop and collects the results of the computation from the FPGA module.

Timing is measured from the time the host program signals the FPGA to enter the loop of 64,000 atom calculations to the time the host receives a signal from the FPGA indicating completion of all atom calculations. It includes not only the computational

time but also two communication delays, one when sending a message to the FPGA to commence operation and another at the end of the run when the FPGA signals the host computations are complete. The latency of the bus is obscured when using the generated HLL interface provided by the ImpulseC toolset, but it is assumed to be almost negligible.

Taking advantage of the ImpulseC toolset, the ported `pair_lj_charmm_coul_long:compute` function described above was simulated first within the ImpulseC development environment to verify the functionality and estimate the performance. The simulated design has a maximum combinational path of 364 clock cycles; meaning, to computing one atom on the FPGA takes 364 clock cycles. This result is obtained purely through the automated translation of the HLL-to-HDL in CoDeveloper leaving the potential for further improvements, which will be discussed later. At the clock rate of 100Mhz, limited by the floating-point core design, 32,000 atoms (one time step) can be computed in 114ms based on the number of numerical operations the FPGA must perform internally. The simulated compute time does not include communication signals and data transfers overheads to and from the FPGA. Using equation 3.1, the effective speedup of the estimated function's computations 11.5x, over an order of magnitude, for this specific function.

$$Speedup = \frac{Runtime_{Microprocessor}}{Runtime_{FPGA-acceleration}}$$

Equation 3.1: Speedup (Alam, et. al, 2007)

This acceleration translates to a total overall runtime of only 70 seconds and a speedup for the entire application of 2.7x, neglecting all communication overheads.

In addition to the simulated computational requirements, the communication performance is another important consideration for the viability of the system. An analysis of the communication overhead is needed to give an estimation of the bandwidth requirements for this implementation. In the ported code, a timestep is computed every 100ms or ten timesteps per second. For the Rhodopsin protein benchmark this equates to transmitting 40.96MB of data or 32,000 atoms with 16 double precision floating-point numbers per atom to the FPGA. As shown in the previous section, figure 2.3, the theoretical bandwidth to the FPGA device is 1.6GB/s or 800MB/s bidirectional when leveraging the HyperTransport™ bus. The HyperTransport™ link provides more than 18 times the required bandwidth for the application, leaving a wide margin for actual implementation requirements.

The current implementation of the fully-accelerated application is not fully functional. The execution of the algorithm on the target FPGA results in erroneous values. The software simulation values are given in table 3.2 and the hardware implementation values in table 3.3 below. The hardware implementation values are largely affected by a bug in the handling of over and under flow situations that arise in the floating-point operations. To counteract the errors several methods were attempted. First, additional memory was allocated to include every variable in each step of the computation and variables were interspaced within memory with 64 bit blank blocks. This extra memory functions as a register, which allowed computations to be observed

with finer granularity. Further, the goal of using extra memory interspaced between each variable was to allow the capture of any overflow. The additional memory read from the device was blank indicating overflow from the floating-point operations was not being addressed. The numerical results were also unchanged.

Columbic Force	0.063911
$(1/distance^2)$	0.900109
Prefactor	-4.358398
$(1/distance^2)^3$	0.729266
Lenard Jones force	736856.875000

Table 3.2: Software simulation results. 'Distance' is the distance between the given atom pair being computed.

Columbic Force	-0.000000
$(1/distance^2)$	3.660845
Prefactor	0.000000
$(1/distance^2)^3$	49.061855
Lenard Jones force	-66195928.000000

Table 3.3: Hardware implementation results. Note the negative zero value, which indicates an underflow problem in the floating point core.

Floating-point libraries were switched from XtremeData's own implementation to Altera's. Each floating-point IP library supports different rounding methods and operator implementations. The shift in libraries was expected to improve the results to within a

reasonable approximation of the software results. There was no change in the value calculated on the FPGA, which lead to an exploration of the timing and utilization of the implementation on the FPGA. The implementation uses approximately 35% of the total logic and all clock tolerances are met. If the utilization of the logic space were high, incorrect timing and placement of the design on the device might have developed causing calculation errors.

For this implementation we use 2 blocks of 1MB bram to send and buffer values. The size of this buffer may be limited by the resources on the FPGA as the Stratix II 180 is cited by Altera as having a maximum of 1.17MB of memory capacity.. The ImpulseC Codeveloper may also be limiting the size of buffers arbitrarily to ease HLL-to-HDL translation. Each block of atom values sent to the FPGA must also generate a signal to confirm that memory values are currently readable. The FPGA must then read a block of values and then generate a signal back to the host allowing the host to start rewriting that block of values. While the FPGA is still reading the values, the host is writing to the second block of values. The two blocks allow the FPGA and host to overlap reading and writing. Since the benchmark requires 40.96MB of data a second, a minimum of 41 synchronizations are required. These synchronizations over all the transfers become a significant source of latency. The run time of the hardware implementation with communication overhead is almost 64 times slower than the original software run time. To get a better picture of just the computation performance of the hardware-ported algorithm, the original algorithm was modified to load just one atom’s values and then repeatedly perform the calculations 64,000 times (computationally equivalent to two

timesteps). The hardware performance figures are taken from this implementation in order to measure only the core performance of the algorithm's calculations.

While the results are numerically incorrect, the FPGA must still perform the all the operations. For example a multiplier will take N number of clock cycles regardless of it multiplying an erroneous or correct value, allowing timing to be somewhat independent of the values computed. The meantime of the hardware implementation for performing 64,000 atom calculations, is 163ms and 168ms is the median. This is almost a 16x speedup due to the fact the run calculates twice the number of atoms, 64,000 atoms, versus 32,000 used in the software version of the Rhodopsin protein. This measured result is better than the estimates made with Stage Master Explorer. Results from Stage Master Explorer and timing runs are based on the core runtime of the algorithm, meaning they do not include any significant communication overhead which will be discussed later.

The communication channels between the FPGA and host, as discussed earlier, are shown to be theoretically sufficient for the amount of data transferred. A previous implementation attempted to stream values to and from the device. The measured throughput when using these streaming interfaces was significantly smaller than what was needed for the ported algorithm. A move to shared memory interfaces did improved the bandwidth, but due to the synchronization required at every memory update between the host and FPGA for shared memory interfaces, the latency of the bus deteriorated performance.

The understanding gained through the method presented of analysis of the targeted application, simulated implementation, and hardware experimentation is universally applicable across RC and heterogeneous platforms. Results show significant possible performance gains if implementation details are suitably addressed in the continued development of HLL-to-HDL technologies. The acceleration methodology, flexibility and advancements in the field of FPGAs, and HLL support allow scientific disciplines to develop application specific hardware that are both potentially powerful and portable. As we will discuss in the next chapter, FPGAs serve as an increasingly universal solution to scientist's needs for application acceleration across a number of specific problem domains.

CHAPTER FOUR

CONCLUSIONS

The implementation methodology and analysis presented for the targeted application including profiling and analysis, hardware implementation, simulations, performance prediction and analysis, and hardware experimentation are universally applicable across many RC and heterogeneous platforms. The acceleration methodology, flexibility and advancements in the field of FPGAs and HLL support combine to allow scientific disciplines to develop application specific hardware that is portable and not permanently fixed to a specific problem domain. Leveraging these advancements in reconfigurable computing (RC) hardware and software development has enabled scientific applications to utilize RC platforms to improve application performance and circumvent some of the limitations plaguing traditional high-performance computing platforms.

Using LAMMPS as a representative scientific application this thesis presented an approach that is targeted at exploring how an application scientist could achieve application acceleration on RC hardware using a few key techniques. First profiling was used to characterize the application's appropriateness for FPGA acceleration and identify where the majority of the compute time was spent. Next, these specific compute intense portions of the code were studied in detail to characterize computational and communication loads. To achieve the most performance, only a few 'hot spots' (compute intense functions) were exploited in ImpulseC for acceleration. Use of the ImpulseC

development environment allowed the estimation of the performance and verification of functionality in a HLL before deciding on targeting a specific platform.

The specific platform chosen to demonstrate the implementation of the accelerated LAMMPS application was the XD1000. The XD1000 demonstrated potential to support HPC applications through its distinctive architecture. However, ImpulseC's automated HLL-to-HDL was not able to fully utilize this architecture's potential, leading to a cycle of identify and resolve issues on that platform. These issues while currently limiting should not detract from the focus of the performance gains of a hardware implementation. Neglecting the communication, the application acceleration is in line with what was estimated by the ImpulseC toolset.

To further clarify, there are two main issues in the hardware implementation preventing a fully functional implementation. First, the double-precision floating point suffers from an underflow that causes a cascade effect down to other values in the calculation. The results from the hardware are thus numerically inconsistent from the software-only observations. Second, the interface between the host and FPGA on the XD1000 platform does function using a shared memory approach; however it is a poor choice for this type of application. For this reason this work has mainly focused on only the runtime of the core algorithm that was measured in software-only, in hardware simulation, and with the hardware implementation

The demands of such an intensive HPC scientific application may necessitate VHDL hand-coding of a few crucial areas of communication. While developing a custom interface may be out of the scope of an application scientist, any other portions of the

algorithm can still use the automation and flexibility provided by the ImpulseC toolset. This leaves a scientist with the ability to update the target hardware to new versions of software given the hand-coded interface is robustly designed. It is expected that as the HLL-to-HDL software evolves, issues with platform and floating-point support will also be resolved.

With minimum optimization and user effort, an appreciable speedup of 3x over the entire application is achievable. The results shown do neglect most or all of the communication between the FPGA and host, but sufficient communication present in the XD1000 platform to allow for implementation overheads. The analysis of the algorithm and system indicates a data bandwidth available that is substantially greater than required. However, desired implementation of improved communication techniques to fully utilize the XD1000 platform outstrips the ImpulseC CoDeveloper's current abilities provided by HLL-to-HDL translation. Room for performance optimization in the areas of pipelining and parallel processing on the FPGA are also plausible given the abundant bandwidth and current small logic utilization of the implementation. These optimizations are likely candidates for future work discussed in the next chapter and are projected to further improve the performance of LAMMPS.

CHAPTER FIVE

FUTURE WORK

The acceleration of the LAMMPS software places several complex demands on current HLL-to-HDL software. The architecture of the XD1000 is challenging due to the HyperTransport™ bus and dedicated SRAM that must be controlled and interfaced with the FPGA logic fabric or user's design. Additionally the demand of fully functional double-precision and later single precision floating-point operations utilize libraries that have to integrate with these relatively unique communication interfaces. Problems such as timing and bandwidth within the FPGA module itself along with correct floating-point library implementations must all work properly for a successful hardware implementation. Future work will examine in more detail the implementation difficulties and attempt to develop additional solutions to the present problems.

Shared memory interfaces are one such difficulty. This interface type was used due to the significantly limited performance of the alternative streaming interface. The result was that for every update to a memory block, a signal had to be generated to allow the host or FPGA respectively to know that the memory block was now valid for reading. This signal handshaking required for streaming interfaces introduced a large amount of latency. In the future, streaming interfaces will be implemented to allow the buffering of incoming and outgoing data thus eliminating the need for signaling handshaking and is expected to increase the performance of the communication.

Another future performance enhancement is the implementation of pipelining techniques for computing the forces on each atom. Initial attempts revealed insufficient

logic in the FPGA device to support a full pipeline of the function. With a revisal of communication interfaces and hand optimization, it is expected that this pipelined implementation is an easily achievable goal. The benefits would provide a higher throughput but a longer latency when observing the computations for an individual atom.

The final goal on the agenda is to also include performance comparison research between the XtremeData XD1000 platform and the DS1000 system by DRC. These two systems are very similar in specifications. The main difference is the FPGA device: DRC DS1000 utilizes a Xilinx Virtex 4 FPGA as opposed to the Altera Stratix II FPGA in the XtremeData XD1000 platform. It is anticipated that the investigation of these two platforms will reveal advantages in FPGA devices and RC platforms as well as strategies in hardware and software that best meet with the needs of the scientific community.

APPENDIX

Selected portions of LAMMPS Xprofiler Report

Flat profile: Abbreviated results

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ks/call	total Ks/call	name
74.33	2813.63	2813.63	101	0.03	0.03	PairLJCharmmCoulLong::compute(int, int)
13.78	3335.14	521.51	12	0.04	0.05	Neighbor::pair_bin_newton()
3.20	3456.33	121.19	101	0.00	0.00	PPPM::fieldforce()
1.69	3520.48	64.15	144365708	0.00	0.00	Neighbor::find_special(int, int)
1.57	3579.89	59.41	101	0.00	0.00	PPPM::make_rho()
0.99	3617.41	37.52	101	0.00	0.00	DihedralCharmm::compute(int, int)
0.83	3648.64	31.24	6464000	0.00	0.00	PPPM::compute_rho1d(double, double, double)
0.48	3666.64	18.00	101	0.00	0.00	AngleCharmm::compute(int, int)
0.34	3679.51	12.87	101	0.00	0.00	PPPM::setup()
0.28	3690.18	10.66	1373	0.00	0.00	pack_3d(double*, double*, pack_plan_3d*)
0.25	3699.81	9.63	40211534	0.00	0.00	Domain::minimum_image(double*, double*, double*)
0.21	3707.72	7.91	1272	0.00	0.00	unpack_3d_permute1_2(double*, double*, pack_plan_3d*)
0.17	3714.26	6.54	101	0.00	0.00	Pair::virial_compute()
0.16	3720.46	6.20	101	0.00	0.00	PPPM::poisson(int, int)
0.14	3725.66	5.20	427533	0.00	0.00	FixShake::shake3angle(int)
0.13	3730.66	5.00	101	0.00	0.00	Verlet::force_clear(int)
0.10	3734.45	3.79	606	0.00	0.00	AtomFull::unpack_reverse(int, int*, double*)

---

call graph profile: Abriviated results

The sum of self and descendents is the major sort for this listing.

function entries:

index — the index of the function in the call graph listing, as an aid to locating it (see below).

%time — the percentage of the total time of the program accounted for by this function and its descendents.

self — the number of seconds spent in this function

itself.

descendents
: the number of seconds spent in the descendents of this function on behalf of this function.

called
: the number of times this function is called (other than recursive calls).

self
: the number of times this function calls itself recursively.

name
: the name of the function, with an indication of its membership in a cycle, if any.

index
: the index of the function in the call graph listing, as an aid to locating it.

parent listings:

self*
: the number of seconds of this function's self time which is due to calls from this parent.

descendents*
: the number of seconds of this function's descendent time which is due to calls from this parent.

called**
: the number of times this function is called by this parent. This is the numerator of the fraction which divides up the function's time to its parents.

total*
: the number of times this function was called by all of its parents. This is the denominator of the propagation fraction.

parents the name of this parent, with an indication of the parent's membership in a cycle, if any.

index the index of this parent in the call graph listing, as an aid in locating it.

children listings:

self* the number of seconds of this child's self time which is due to being called by this function.

descendent*
the number of seconds of this child's descendent's time which is due to being called by this function.

called** the number of times this child is called by this function. This is the numerator of the propagation fraction for this child.

total* the number of times this child is called by all functions. This is the denominator of the propagation fraction.

children the name of this child, and an indication of its membership in a cycle, if any.

index the index of this child in the call graph listing, as an aid to locating it.

* these fields are omitted for parents (or children) in the same cycle as the function. If the function (or child) is a member of a cycle,

```
          the propagated times and propagation denominator
          represent the self time and descendent time of the
          cycle as a whole.

          ** static-only parents and children are indicated
          by a call count of 0.



          cycle listings:
          the cycle as a whole is listed with the same
          fields as a function entry.  Below it are listed
          the members of the cycle, and their contributions
          to the time and call counts of the cycle.
granularity: Each sample hit covers 4 bytes.

                                  called/total       parents
index  %time    self descendents  called+self    name         index
                                  called/total       children

                0.00      194.32       1/1           .__start [2]
[1]     94.0    0.00      194.32       1         .main [1]
                0.00       99.22       1/1           .Run::command(int,char**) [5]
                0.00       94.95       1/1           .System::destroy() [7]
                0.00        0.13       1/1           .ReadData::command(int,char**) [61]
                0.00        0.02       3/3           .Input::next() [116]
                0.00        0.00       1/1           .System::create() [179]
                0.00        0.00       1/1           .System::open(int*,char***) [450]
                0.00        0.00       1/1           .ReadData::ReadData() [435]
                0.00        0.00       1/1           .ReadData::~ReadData() [443]
                0.00        0.00       1/1           .System::close() [449]

-----------------------------------------------
                                                <spontaneous>
[2]     94.0    0.00      194.32               .__start [2]
                0.00      194.32       1/1           .main [1]
                0.00        0.00       1/1           .__C_runtime_startup [476]

-----------------------------------------------
                0.00       94.95       1/2           .Update::~Update() [8]
                0.00       94.95       1/2           .Verlet::run() [6]
[3]     91.9    0.00      189.90       2         .Verlet::iterate(int) [3]
```

```
                132.58        0.79     100/101         .PairLJCharmmCoulLong::compute(int,int) [4]
                  0.02       28.09      11/12          .Neighbor::build() [9]
                  0.00       12.87     100/101         .PPPM::compute(int,int) [11]
                  0.00        7.43     100/100         .Modify::initial_integrate() [15]
                  3.13        0.49     100/101         .DihedralCharmm::compute(int,int) [21]
                  1.50        0.48     100/101         .AngleCharmm::compute(int,int) [23]
                  0.00        1.04     100/100         .Modify::post_force(int) [26]
                  0.35        0.00     100/101         .Verlet::force_clear(int) [41]
                  0.00        0.32     100/100         .Modify::final_integrate() [44]
                  0.00        0.25     100/101         .Comm::reverse_communicate() [49]
                  0.12        0.01     100/101         .BondHarmonic::compute(int,int) [59]
                  0.06        0.07      11/12          .Comm::borders() [58]
                  0.01        0.11      89/89          .Comm::communicate() [65]
                  0.00        0.08     100/100         .Neighbor::decide() [77]
                  0.04        0.01     100/101         .ImproperHarmonic::compute(int,int) [88]
                  0.00        0.04      11/11          .Modify::pre_neighbor() [103]
                  0.02        0.00      11/12          .Comm::exchange() [123]
                  0.00        0.00       2/2           .Output::write(int) [165]
                  0.00        0.00     513/513         .Timer::stamp(int) [216]
                  0.00        0.00     202/202         .Timer::stamp() [230]
                  0.00        0.00      11/12          .Domain::pbc() [269]
                  0.00        0.00      11/12          .Domain::reset_box() [270]
                  0.00        0.00      11/12          .Comm::setup() [268]
                  0.00        0.00      11/12          .Neighbor::setup_bins() [271]

-----------------------------------------------
                  1.33        0.01       1/101         .Verlet::setup() [20]
                132.58        0.79     100/101         .Verlet::iterate(int) [3]
[4]     65.2  133.91        0.80     101          .PairLJCharmmCoulLong::compute(int,int) [4]
                  0.57        0.00     101/101         .Pair::virial_compute() [32]
                  0.23        0.00 2161526/9150610     .exp [27]
```

REFERENCES

Alam, S.R., P.K. Agarwal, J.S. Vetter, and M.C. Smith, "Throughput Improvement of Molecular Dynamics Simulations Using Reconfigurable Computing," *Scalable Computing: Practice and Experience - Scientific International Journal for Parallel and Distributed Computing*, **8/4**, 395-410, (2007).

Alam, S. R., P. K. Agarwal, M. C. Smith, J. S. Vetter, and D. Caliga. "Using FPGA Devices to Accelerate Biomolecular Simulations." *Computer* vol. 40, no. 3. March 2007. pp. 66-73.

Bader, D. A. "Computational biology and high-performance computing." *Comm. ACM.* vol. 47 no. 11, June 2004. pp. 34-41.

DRC Computer Corp. "DRC DS1000 Dev System." DRC Computer Corp. product brief. 2008; http://www.drccomputer.com/pdfs/DRC_DS1000_fall07.pdf

Estrin, G. and R. Turn. "Automatic Assignment of Computations in a Variable Structure Computer System." *IEEE Transactions on Electronic Computers.* Vol. EC12 No. 5, December 1963.

Herbordt, M. C., T. VanCourt, Y. F. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. "Achieving high performance with FPGA-based computing." *Computer* vol. 40, 2007. pp. 50-57.

IBM Corp. "Life Sciences Molecular Dynamics Applications on the IBM System Blue Gene Solution: Performance Overview," IBM Corp. white paper, 2006.

ImpulseC Corp. "ImpulseC CoDeveloper C-to-FPGA tools." ImpulseC Codeveloper product website, 2008; http://www.impulsec.com/products_universal.htm

Kilts, S. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007.

Kindratenko, V., and D. Pointer. "A case study in porting a production scientific supercomputing application to a reconfigurable computer." in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

Pellerin, D. and S. Thibault, *Pratical FPGA Programming in C*. Upper Saddle River, NJ: Prentice-Hall, 2005.

Reid, F. and L. Smith, "Performance and Profiling of the LAMMPS Code on HPCx." tech. report HPCxTR0508, HPCx Consortium, 2005.

Sandia National Laboratory. LAMMPS Molecular Dynamics Simulator, Release April 2006; http://lammps.sandia.gov.

Scrofano, R., M. Gokhale, F. Trouw, and V. Prasanna, "A Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers." in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

Smith, M.C., S.R. Alam, P. Agarwal, and J.S. Vetter, "A Task-based Development Model for Accelerating Large-Scale Scientific Applications on FPGA-based Reconfigurable Computing Platforms." *Reconfigurable Systems Summer Institute, RSSI'06*, Champaign-Urbana, IL: July 10-14, 2006.

Top500 "Top 500 SuperComputers November 2008." Top 500 Supercomputer website, 2008; http://www.top500.org/lists/2008/11

XtremeData Corp. "XD1000 Development System Product Brief." XtremeData Corp. product brief. 2007;8 http://www.xtremedatainc.com/pdf/Dev_Sys_XD1000_Brief.pdf