BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

SCALABLE MOLECULAR DYNAMICS SIMULATION USING FPGAS AND MULTICORE PROCESSORS

by

MD. ASHFAQUZZAMAN KHAN

B. Eng., Tohoku University, 2004
M. Eng., Tohoku University, 2006

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2013

Approved by

First Reader

Martin C. Herbordt, PhD
Associate Professor of Electrical and Computer Engineering

Second Reader

Ayse K. Coskun, PhD
Assistant Professor of Electrical and Computer Engineering

Third Reader

Ajay Joshi, PhD
Assistant Professor of Electrical and Computer Engineering

Fourth Reader

Gerald Paul, PhD
Research Scientist of Physics

Acknowledgments

First of all, I would like to thank my advisor, Professor Martin Herbordt, for his guidance throughout the entire course of my PhD. He not only taught me how to look at the big picture, but also provided me with opportunities to take part in preparation of research proposals and classroom teaching. His patience and enthusiasm, together with his intellectual, moral, and financial support, played a crucial role in the successful completion of my studies.

I would also like to thank the members of my PhD defense committee, Professor Ayse Coskun, Professor Ajay Joshi, and Dr. Gerald Paul, for taking the time to carefully review my dissertation. Their encouragement and insightful comments greatly improved the quality of this dissertation. My thanks to Professor Alan Pisano for kindly presiding over my final defense session, and also for his feedback.

I want to thank the staff of the ECE department for being so friendly and helpful all the time. I also thank Professor Janusz Konrad for providing the LaTex template for the dissertation through Boston University website. My thanks to all my friends and colleagues in the department for making my days in Boston a pleasant memory. Special thanks to Matt (Shihchin) Chiu for the numerous technical and casual discussions we had, as well as for his help and suggestions.

My heartfelt thanks to my lovely wife Satia Ahmed, who did so well at one of the toughest tasks in the world, being married to an international PhD student. She and my little daughter Sora provided me with the perfect amount of inspiration, emotional support, and much-needed break. Last but not least, I thank my parents and all other family members for their constant support and love, without which the Almighty would most certainly not bless me with this accomplishment.

I dedicate this dissertation to my mother. It is her inspiration during the early days of my academic life that ultimately guided me to this point.

SCALABLE MOLECULAR DYNAMICS SIMULATION USING FPGAS AND MULTICORE PROCESSORS

(Order No.)

MD. ASHFAQUZZAMAN KHAN

Boston University, College of Engineering, 2013

Major Professor: Martin C. Herbordt, PhD, Associate Professor of Electrical and Computer Engineering

ABSTRACT

While Molecular Dynamics Simulation (MD) uses a large fraction of the world's High Performance Compute cycles, the modeling of many physical phenomena remains far out of reach. Improving the cost-effectiveness of MD has therefore received much attention, especially in using accelerators or modifying the computation itself. While both approaches have demonstrated great potential, scalability has emerged as a critical common challenge. The goal of this research is to study this issue and develop MD solutions that not only achieve substantial acceleration but also remain scalable.

In the first part of this research, we focus on Discrete Molecular Dynamics Simulation (DMD), which achieves high performance by simplifying the underlying computation by converting it into a Discrete Event Simulation (DES). In addition to the inherent serial nature of DES, causality issues make DMD a notorious target for parallelization. We propose a parallel version of DMD that, unlike any previous work, uses task decomposition and efficient synchronization and achieves more than 8.5x speed-up for 3D physical systems on a 12 core processor, with potential for further strong scaling.

The second part of this research focuses on FPGA acceleration of timestep-driven MD. We first enhance an existing FPGA kernel to take advantage of the Block RAM architecture of FPGAs. This results in a 50% improvement in speed-up, without sacrificing simulation quality. We then parallelize the design targeting multiple on-board FPGA cores. We combine this with software pipelining and careful load distribution at the application level to achieve a 3.37x speedup over its CPU counterpart.

In the third part we create a framework that integrates the FPGA accelerator into a prominent MD package called NAMD. This framework allows users to switch between the actual accelerator and a simulated version, and provides a means to study different characteristics, such as the communication pattern, of such an accelerated system. Using this framework, we identify the drawbacks of the current FPGA kernel and provide guidelines for future designs. In addition, the integrated design achieves 2.22x speed-up over a quad-core CPU, making it the first ever FPGA-accelerated full-parallel MD package to achieve a positive end-to-end speed-up.

Contents

List of Tables

List of Figures

List of Abbreviations

1D	.....	One Dimensional
2D	.....	Two Dimensional
3D	.....	Three Dimensional
ALM	.....	Adaptive Logic Module
AMBER	.....	Assisted Model Building with Energy Refinement
AMD	.....	Advanced Micro Devices
ASIC	.....	Application Specific Integrated Circuit
BRAM	.....	Block Random Access Memory
CAAD	.....	Computer Architecture and Automated Design
CHARMM	.....	Chemistry at HARvard Molecular Mechanics
CPU	.....	Central Processing Unit
CUDA	.....	Compute Unified Device Architecture
D-Cache	.....	Data Cache
DC	.....	Direct Computation
DDR	.....	Double Data Rate
DeDup	.....	Deduplication
DES	.....	Discrete Event Simulation
DMA	.....	Direct Memory Access
DMD	.....	Discrete Molecular Dynamics (Simulation)
DNA	.....	DeoxyriboNucleic Acid
DRAM	.....	Dynamic Random Access Memory
DSP	.....	Digital Signal Processor
EDA	.....	Electronic Design Automation
EPCC	.....	Edinburgh Parallel Computing Centre
FFT	.....	Fast Fourier Transformation
FHPCA	.....	FPGA High Performance Computing Alliance
FIFO	.....	First In First Out
FPGA	.....	Field-Programmable Gate Array
fs	.....	Femtosecond (10^{-15} second)
GB	.....	Giga (2^{30} or 10^{9}) Bytes
Gbit	.....	Giga (2^{30} or 10^{9}) Bit
Gbps	.....	Giga (2^{30} or 10^{9}) Bit Per Second

GC	.....	Geometry Core
GFLOPS	.....	Giga (2^{30} or 10^9) FLoating-point Operations Per Second
GHz	.....	Giga (2^{30} or 10^9) Hertz (cycles per second)
GPU	.....	Graphics Processing Unit
GRAPE	.....	GRAvity PipE
GROMACS	.....	GROningen MAchine for Chemical Simulation
HOOMD	.....	Highly Optimized Object-oriented Many-particle Dynamics
HPC	.....	High-Performance Computing
HTIS	.....	High-Throughput Interaction System
I-Cache	.....	Instruction Cache
IBM	.....	International Business Machines
IEEE	.....	Institute of Electrical and Electronics Engineers
I/O	.....	Input Output
IP	.....	Intellectual Property
ISE	.....	(Xilinx) Integrated Synthesis Environment
K	.....	Kilo (2^{10} or 10^3)
KB	.....	Kilo (2^{10} or 10^3) Bytes
L1	.....	Level One
L2	.....	Level Two
L3	.....	Level Three
LAMMPS	.....	Large-scale Atomic/Molecular Massively Parallel Simulation
LJ	.....	Lennard-Jones
LSI	.....	Large Scale Integration
LUP	.....	Look-Up
LUT	.....	Look-Up Table
MB	.....	Mega (2^{20} or 10^6) Bytes
MD	.....	Molecular Dynamics (Simulation)
MDGRAPE	.....	Molecular Dynamics GRAvity PipE
MHz	.....	Mega (2^{20} or 10^6) Hertz (cycles per second)
MODEL	.....	MOlecular Dynamics processing ELement
ms	.....	Millisecond (10^{-3} second)
NAMD	.....	NAnoscale Molecular Dynamics
NCSA	.....	National Center for Supercomputing Applications
nm	.....	Nanometer (10^{-9} meter)
ns	.....	Nanosecond (10^{-9} second)
OpenGL	.....	Open Graphics Library

PARSEC	.....	The Princeton Application Repository for Shared-Memory Computers
PaulQ	.....	Paul's Event Queue
PBC	.....	Periodic Boundary Condition
PC	.....	Personal Computer
PCI	.....	Peripheral Component Interconnect
PCIe	.....	Peripheral Component Interconnect express
PDMD	.....	Parallel Discrete Molecular Dynamics (Simulation)
petaflops	.....	Peta (2^{50} or 10^{15}) FLoating-point Operations Per Second
PME	.....	Particle Mesh Ewald
PPME	.....	Particle Particle Mesh Ewald
PPPM	.....	Particle-Particle Particle-Mesh
PQ	.....	Paul's Event Queue
RAM	.....	Random Access Memory
rFFT	.....	Reverse Fast Fourier Transformation
RIKEN	.....	RIkagaku KENkyujo (Institute of Physical and Chemical Research, Japan)
RMS	.....	Root-Mean-Square
SDRAM	.....	Synchronous Dynamic Random Access Memory
SIMD	.....	Single Instruction Multiple Data
SM	.....	Streaming Multiprocessor
SODIMM	.....	Small Outline Dual Inline Memory Module
SPME	.....	Smooth Particle Mesh Ewald
SRAM	.....	Static Random Access Memory
STMV	.....	Satellite Tobacco Mosaic Virus
TM	.....	Transmogrifier
UIUC	.....	University of Illinois at Urbana-Champaign
USC	.....	University of Southern California
VdW	.....	Van der Waals

Chapter 1

Introduction

1.1 The Problem

Decades of sustained growth of the computer industry have fundamentally revolutionized our way of approaching scientific problems. Computer simulations have evolved as the third pillar of science, bridging the gap between the traditional two, theory and experiment [135]. By combining the right computational model of a physical system with appropriate software and hardware, it is now possible to study the behavior of that system without actually performing physical experiments. Molecular dynamics simulation (MD) is one such tool that models an actual system at the atomic level and uses classical mechanics to study its behavior [3, 139]. MD acts as a virtual experiment and provides a projection of the laboratory experiment with greater details. It is one of the most widely used tools in computational science and engineering. In biomedical science it has so far provided many important insights in understanding the functionality of biological systems [2, 25, 45, 54, 83, 85, 87, 116].

A recent example of such findings can be found in a study by Severin, et al. [150] where the authors studied DNA methylation, a process where the hydrogen atom at the 5-position of a cytosine base is replaced by a methyl group (CH3). Methylation plays an important role in gene expression, which in turn dictates how a living organism adapts to its habitat and life experience, e.g., dietary habits. The pattern of methylation controls protein binding to target sites on DNA, often silencing

genes, which may pathologically lead to cancer. Until recently, scientists were aware of only two indirect methods how methylation dictates gene expression. This work found, using MD, that there might be a third way how methylation can regulate gene expression more directly, through changing mechanical properties of DNA. Such a finding can be crucial in understanding methylation-based epigenetics, including understanding how our body adapts to our life style.

MD is an iterative process where simulation advances by timesteps. Each timestep has two phases; force computation and motion update. Forces working on each particle in the system are computed first, using classical mechanics, and then the state of each particle is updated. One timestep typically corresponds to one or a few femtoseconds (fs) of real time; MD runs thus require millions of timesteps to simulate just a few nanoseconds (ns) of real time.

Among all computations in MD, evaluation of forces consumes the majority of runtime. Although the exact computation depends on the physical system, the force model and other simulation parameters, it generally involves computing the forces due to various bonds (e.g., hydrogen bonds) and the non-bonded forces, namely the van der Waals (VdW) and the electrostatic (or Coulomb force):

$$F_{total} = F_{bond} + F_{angle} + F_{dihedral} + F_{hydrogen} + F_{vanderWaals} + F_{electrostatic} \tag{1.1}$$

Since bonds only affect a few neighboring particles, evaluation of bonded forces has computational complexity of $O(N)$ where N is the total number of particles in the system. The motion update phase of a timestep is also relatively straightforward and only takes $O(N)$ runtime. Computational complexity of the non-bonded forces, on the other hand, is initially $O(N^2)$, because they require evaluating interactions between all possible pairs of particles. Although several techniques exist to reduce this complexity significantly, overall, MD remains compute intensive.

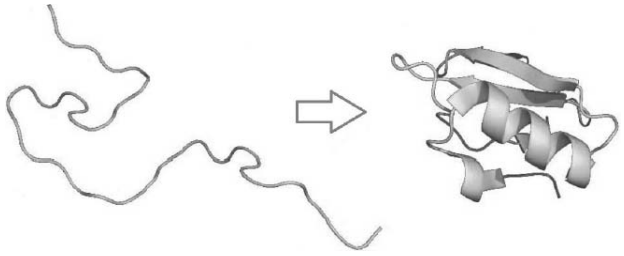


Figure 1·1: Protein folding - the process by which a linear chain of amino acids folds into a three dimensional functional protein [174]

Fortunately MD lends itself well to parallelism through spatial decomposition of the physical system. Widely used MD packages can take full advantage of this and as a consequence, MD scales well on CPU-based systems [94]. Heroic runs of MD easily involves hundreds of CPU cores and months of runtime [55, 87]. Yet the size and timescale achieved by even the most heroic MD run still fall short, by many orders of magnitude, compared to the size of the systems of interest and the biologically relevant timescales of potential events. For example, it is estimated to take about 70 years on a single high-end CPU core to simulate only a 100 ns of a million-atom biological system, such as the Satellite Tobacco Mosaic Virus (STMV) [55]. In contrast, many important biological phenomena, e.g., protein folding (see Figure 1·1), are believed to occur in microsecond to millisecond range [49, 72]. Thus, faster execution of MD is a crucial goal for the scientific community.

Acceleration of MD has therefore received much attention. One way to achieve acceleration is to use specialized hardware, such as Field Programmable Gate Arrays (FPGAs). A typical hardware-accelerated MD will offload some of the most time-consuming computations, e.g., the evaluation of non-bonded forces between particle-pairs, to the hardware to achieve speed-up over the CPU-only version. While an order of magnitude speed-up seems to be achievable in this manner [33, 162], the gain diminishes quickly if the accelerated system does not scale. For example, a 10x speed-up over a single CPU core can be easily matched just by adding another high-

end multicore CPU. Scalability has, in fact, emerged as one of the greatest challenges in realizing the full potential of hardware-accelerated MD [131].

Another way to achieve faster simulation is to use simpler computational model. For MD, converting the computation into a Discrete Event Simulation (DES) promises great speed-up, at some cost in simulation quality. In this method, called Discrete MD (DMD), step-wise potential is used to approximate the continuous force fields of a real system. Instead of updating every particle in every timestep, the state of a particle is only updated when it crosses a point of discontinuity in its step-wise potential. For the rest of the time, particles maintain constant velocity. Predictions are made regarding when a particle will cross such a point of discontinuity and they are recorded as events in a priority queue. Events are extracted from the priority queue in temporal order and processed. New predictions are made and queued, while old predictions are invalidated as needed. Thus, timestep-driven MD is converted into event-driven DMD. The simpler force model and the ability to avoid frequent system-wide update result in faster simulation. Studies have shown that this approach can provide significant insight about biological processes with far less hardware resources [46, 136, 156, 179]. The simplicity of the models can be substantially compensated for by the capability of researchers to refine simulation models interactively [169, 170]. The issue here is, again, scalability. DES is inherently serial and its application in MD has especially been proven to be a very difficult target of parallelization [41, 56, 103, 109, 158].

In this dissertation, we address the problem of scalability in both discrete MD (DMD) and hardware-accelerated MD. In particular, we investigate the issues in parallelizing DMD on multicore CPUs and the challenges in achieving a full-parallel production-level FPGA-accelerated MD package.

1.2 Summary of Previous Work

1.2.1 Hardware Acceleration of MD

MD is a central method in High Performance Computing (HPC) with applications throughout engineering and natural science. Its impact in biology and medicine can be seen PUBMED [137], which shows more than 3000 related publications in the year 2011 alone (returns 3355 hits for the query "molecular dynamics"[Title/Abstract] AND ("2011/01/01"[PDAT] : "2011/12/31"[PDAT])). Computational complexity of MD, along with its importance, made it a target of clusters and supercomputers from the very beginning. The relatively longer lifetime of MD packages, due to the stability of the underlying algorithms, also made the effort of porting MD to co-processors or even to MD-specific hardware, worthwhile. This was further fueled by the saturation of increase in raw CPU performance; first the saturation of instruction-level parallelism in around the year 2000 [76], followed by the saturation of CPU frequency due to the power density issue in around the year 2005 [164, 165].

ASICs (Application Specific Integrated Circuits): ASIC-based acceleration of MD goes back to the 1990's, when a group of researchers in Japan developed MD Engine, a collection of custom developed processors that worked in parallel to simulate a system [8, 168]. This was followed by the MDGRAPE-3, also known as the "Protein Explorer", a special-purpose petaflops computer system with a hardware accelerator for MD [117, 141, 166, 167] developed in RIKEN, Japan in 2006. The most recent work on ASIC implementation of MD was done by D. E. Shaw Research in New York, where they developed Anton, a supercomputer specialized for bio-medical application of MD [48, 153]. Each of these systems is reported to have achieved significant speed-up over corresponding CPU-based systems. Among these, the 512-node version of Anton has achieved millisecond range simulation, two orders of magnitude beyond

the previous state of the art [154]. While ASIC-based systems are, by definition, expected to provide the optimal performance, in practice finite on-chip resources and data communication issues often restrict the size of the problem that can be simulated efficiently, thus limiting the advantages. In addition, the non-recurring cost of ASIC-based systems still remains prohibitive for most users.

Cell Processor: Cell, primarily developed for gaming, combines a general purpose Power PC processor with multiple SIMD vector coprocessors on a single die. It was used to accelerate the non-bonded force computation kernel of MD [155]. Although a speed-up of around 7x - 9x was reported, the baseline kernel itself was modified, making a direct comparison difficult. Absolute runtime results indicate a few times slowdown with respect to the corresponding production code.

FPGAs (Field-Programmable Gate Arrays): FPGAs have traditionally been known for their programmability and low power consumption. In recent years, they are equipped with dedicated multipliers and individually accessible Block RAMs (BRAMs) of different granularity [5, 177]. There have been numerous studies on how to implement different portions of MD on FPGAs, the earliest one dating back to 2003 [12, 33, 36, 38, 65, 66, 67, 70, 71, 86, 91, 95, 100, 147, 148, 176]. While many of these studies reported significant speedup for the individual pieces they worked on, none of the widely used MD packages have an FPGA-accelerated version yet.

The most notable recent work on FPGA-accelerated MD was done at CAAD Lab at Boston University, where a non-bonded force kernel was implemented on an Altera Stratix-III FPGA [33]. The kernel ran 26x faster than the total serial runtime of the corresponding production software code. While this work demonstrates that FPGAs can be a viable option for acceleration of MD, absence of an integrated full-parallel production version remains an issue. In fact, this design does not consider the scaling

issues of hardware-accelerated MD. For example, it requires signification amount of additional data conversion and communication, which will very likely diminish the advantages of acceleration eventually.

Most of the FPGA work on MD is done by hardware engineers without sufficient consideration of production MD packages, which tend to be heavily optimized for performance. Often these are bottom-up approaches where a portion of the computation is accelerated and then integrated into (typically simplified) software, and only for proof-of-concept. This, along with the lack of generalized FPGA platforms, is one of the main reasons why FPGAs are not yet the primary choice of acceleration of MD, despite promising performance and a great advantage in energy efficiency. Integration of FPGA kernels into production MD packages is, therefore, one of the most important steps towards realizing the full potential of FPGA-accelerated MD.

GPUs (Graphics Processing Units): Although GPUs were initially developed for fast rendering of computer graphics, the tremendous computational power of modern GPUs, together with their abundance and low cost, mostly due to the worldwide gaming market, has made them a viable candidate for scientific computing [123, 124, 126]. The introduction of high-level programming languages for GPUs, such as CUDA and OpenGL, has allowed rapid development of molecular dynamics applications using GPUs. Almost all publicly available MD packages now have GPU-accelerated versions [125]. While some of them have achieved significant speed-ups in some restricted conditions, most struggled to achieve reasonable speed-up for complex and practical cases [62, 162]. Scaling to multiple GPU nodes has become a particularly tough issue [131]. The most notable work on GPU acceleration of MD was done by the NAMD group at the University of Illinois [131, 162] where a 6x-7x speed-up was achieved over CPUs. Scaling to multiple

GPU nodes was identified to be a critical issue, limiting the speed-up to as low as 3.4x for a 60 CPU/GPU cluster. It is worth noting that this work only implemented a non-bonded force computation kernel on the GPUs, leaving the amount of data communication unchanged. Since reasonable speed-up was achieved in computation, communication became a bottleneck in scaling to multiple nodes. FPGA-based accelerators, in this regard, can play an important role, since FPGAs have the potential to provide direct data communication among accelerators. FPGAs are routinely used in network routers, making them a natural fit for this purpose [28].

1.2.2 Parallelization of DMD

DMD, an instance of DES, has been successfully used to study bio-medical systems, making it an emerging alternative to traditional timestep-driven MD [27, 46, 136, 169, 170, 179, 180]. In addition to the inherent serial nature of DES, causality concerns make DMD an especially difficult target of parallelization [56, 102, 103, 109]. Most previous work on parallelization of DMD used spatial decomposition, the same way a timestep-driven MD is parallelized, and often studied only two dimensional (2D) systems (e.g. disks) [103, 109, 113, 158]. Such approaches resulted in modest speed-up using a huge amount of resources and are likely to be impractical for three dimensional (3D) systems. Moreover, much of this work was done a decade ago. Since then, event processing speed has increased dramatically, through advances in both processors and algorithms, especially when contrasted with interprocessor communication latency. This means that parallelizing DMD through spatial decomposition is likely to be even less efficient now. These are the reasons why, in fact, all previously reported work in bio-medical work that used DMD, used a serial version of it. They can end up taking month-long runtimes, although requiring far less hardware resources than timestep-driven MD. We are aware of no existing production parallel DMD (PDMD) codes that predates the publication of our work [78, 90].

1.3 Summary of Contributions

The contributions of this dissertation can be categorized into, 1) parallelization of DMD on shared memory multicore processors and 2) integration of FPGA kernel into production MD package. In the the first category, we propose a new parallelization method for DMD that achieves close to 6x and 9x speed-up on an 8-core and a 12-core processor respectively, with potential for further strong scaling. In the second category, we first improve the performance of an existing FPGA kernel by 50%, just by exploiting of the Block RAM architecture of the FPGAs. We then parallelize the design to utilize multiple on-board FPGAs and integrate it into a full-parallel production MD code. The integrated design achieves 2.22x speed-up over the highly optimized production code and provides a framework for studying design issues of FPGA-accelerated MD, such as the data communication pattern. Using this framework, we identify the drawbacks of the current FPGA kernel and provide guidelines for future designs. We also study the plausibility of using FPGA-centric clusters for MD, in particular by determining the communication requirements of such a cluster.

1.3.1 Efficient Parallelization of DMD

Studying Issues in Parallelizing DMD: Although parallelization of DES is a well-studied field, there has been no systematic description of the issues that needs to be dealt with while parallelizing DMD. In this work, we define various hazards, similar to those found in hardware pipeline design, that may occur in parallel DMD; and describe how to deal with them efficiently. The specific hazards we define are causality hazard, coherence hazard and a combination of these two.

Parallelizing DMD through Event-based Decomposition: We propose a method that, unlike any previous work, uses event-based decomposition or task

decomposition to parallelize DMD. Our method is micro-architecture inspired where multiple simultaneous threads work on different events from the head of the event queue in parallel, but only one thread commits at one time. In other words, we process the events in parallel, but maintain in-order commitment. Using the shared memory architecture of multicore CPUs, in combination with efficient synchronization and a recently proposed data structure, we limit the serial commitment time such that it does not become a bottleneck. Our method achieves close to 6x and 9x speed-up on an 8-core processor and a 12-core processor respectively, with potential for further strong scaling. In achieving these results, we study various implementations of the proposed task-decomposed DMD, with particular focus on synchronization methods. We find that having one helper and multiple worker threads, and separate distributed locks for each worker thread results in the most efficient implementation, as the number of threads increases.

Analyzing Architectural Limitations: Although our method achieves significant speed-up, a perfect linear speed-up is not achieved. We study the issues that limit the speed-up and find that, individual event processing time increases as the number of threads goes up, inhibiting a perfect speed-up. A major portion of this runtime overhead is due to the requirement of fine-grained locks among threads, while the remaining is mostly due to architectural limitations, such as increased bus utilization.

1.3.2 Integration Framework for FPGA-accelerated MD

Enhancing Performance using Block RAM (BRAM) Architecture: In MD, table interpolation method is ofter used to compute inter-particle forces, replacing some expensive computations (e.g. square root computation) with simple table look-ups. Accuracy of table interpolation method depends on the granularity or size of the table and the order of interpolation. CPU implementations typically use

smaller table such that it fits easily in L1 cache. They compensate this by using relatively higher order of interpolation. In our study, we find that we can reverse this for an FPGA implementation. We can use fine-grained, hence larger, table, and lower order of interpolation, without sacrificing the quality of the simulation. This is possible because of the relative abundance of Block RAMs (distributed configurable memory) in recent FPGAs. This saves us significant amount of logic cells, which would otherwise be used to implement higher order of interpolation. This saving can be used in doing more force computation in parallel, ultimately resulting in a 50% improvement in performance.

Parallelizing FPGA-accelerated MD for Multi-FPGA Board: Recent FPGA boards are often equipped with more than one FPGAs, making it essential to make the accelerated design work on multiple on-board FPGAs. We study the issues in doing so, namely data conversion, partitioning, and software pipelining. We specifically study two methods of partitioning, and analyze the shortcomings of the current kernel design. The quad-FPGA version of the FPGA-accelerated MD achieves a speed-up of 3.37x over a single CPU core.

Integrating FPGA Kernel into a Full-parallel Production MD Package: We integrate the existing FPGA kernel into a prominent full-parallel production MD package, called NAMD [130]. This integrated version allows switching between the actual FPGA kernel and a simulated version of it, and provides a framework for studying various design issues of an FPGA-acclerated MD. Using this framework, we identify the limiting factors of the current kernel and provide guidelines for future designs. We also instrument the code to study various characteristics, e.g. communication pattern, of possible future designs. In addition to providing these features, the integrated quad-FPGA design achieves 2.22x speed-up over the

quad-core CPU (8.39x over a single CPU core), making it the first ever FPGA-accelerated full-parallel MD package to achieve a positive end-to-end speed-up.

Determining Communication Requirements for FPGA-centric MD: Using the integrated framework, we study the plausibility of using FPGA-centric clusters for MD. We derive time budget for data communication in such a system and quantify the data communication characteristics in two ways: analytically and by instrumenting the framework. We apply this information to clusters of various sizes and node complexity and conclude that a cluster with 256 FPGAs distributed in 64 nodes is appropriately provisioned, even for modest simulations, with a bidirectional 3D torus where each link consists of 1-2 of an FPGA's serial ports.

1.4 Organization of the Dissertation

This chapter of the dissertation introduces the problem we address in this work, reviews related previous work in brief, and lists our key contributions.

Chapter 2 provides computational description of MD, including advanced algorithms and techniques for improving performance. It provides a detailed review of previous work on hardware acceleration of MD. In course of doing so, it also introduces the most widely used MD packages. This is followed by an introduction of DMD and a discussion on previous work on parallelization of DMD.

Chapter 3 describes our work on parallelization of DMD. We first discuss the issues in parallelizing DMD and possible approaches towards it. We then establish our baseline serial code and describe simulation models and conventions. Then we present our method and implementation issues: three possible scheduling mechanisms, and other software refinements. Finally, we present the scalability results, followed by various analytical models fleshed out with system-level measurements.

Chapter 4 describes our work on enhancing the baseline FPGA kernel. We first introduce FPGAs in general and then our target hardware system. This is followed by a brief description of the FPGA kernel. We then present the results of our exploration with the design space of table interpolation method and how we can take advantage of Block RAM architecture of the FPGAs. We conclude the chapter with performance improvement results.

In Chapter 5, we first describe the issues in intra-node partitioning. We then present data conversion, software pipelining issues and two partitioning methods. We conclude by presenting speed-up results and contrasting them with their theoretical limits.

In Chapter 6, we discuss the challenges in achieving a full-parallel production-level FPGA-accelerated MD package. We describe the target software and present our integration methods. We describe the features of the integrated framework and also present speed-up results. We analyze the performance of the current FPGA kernel and provide guidelines for future designs.

In Chapter 7, we study the plausibility of using FPGA-centric clusters for MD. We review MD on a single FPGA-based node and use the estimated performance of an optimized implementation to determine the time budget for the communication. We then quantify the data communication characteristics and use this information to determine the communication requirements for such a cluster.

Finally, Chapter 8 summarizes the dissertation and provides possible future directions of this work.

Chapter 2

Molecular Dynamics Simulation (MD)

This chapter provides necessary computational background for this work, and a survey of related previous work. We first introduce various computations in molecular dynamics simulation (MD) and some of the important optimization techniques. Then we review previous work on hardware acceleration of MD. This is followed by a description of discrete MD (DMD) and a discussion on previous efforts to parallelize it.

2.1 Description of MD

MD is an iterative process that models dynamics of molecular particles by applying simple laws of classical mechanics [3, 139]. Simulation advances by timesteps where each timestep is comprised of two phases; force computation and motion update. Figure 2·1 shows a simple MD timestep where forces working on each particle is computed first and then the state of each particle is updated accordingly. The upper

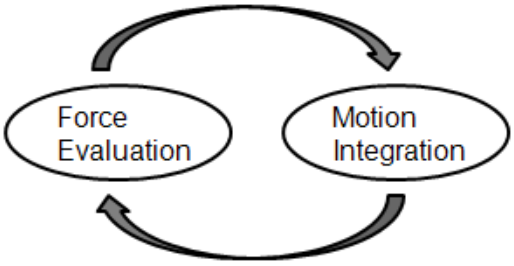


Figure 2·1: MD timestep

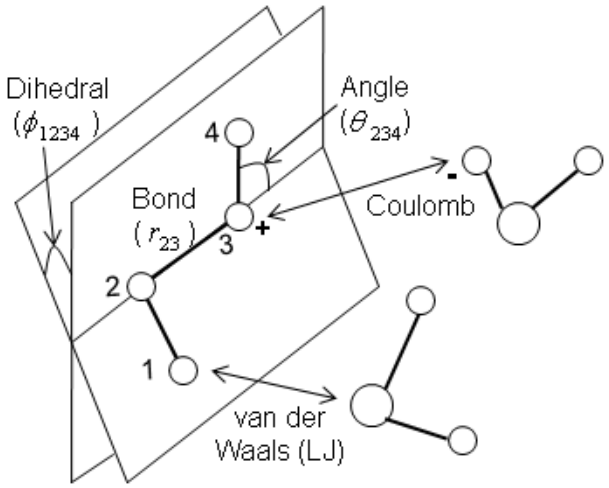


Figure 2·2: MD forces [3, 166]

bound of Δt, the interval between two consecutive timesteps, is determined by the vibration of particles and in a typical simulation, one timestep corresponds to one or a few femtoseconds (fs) of real time. This means millions of timesteps need to be simulated to study a reasonable timescale of the target system, resulting in the long runtime of MD. The user provides the initial state (position, velocity etc.), the force model and other properties of the system and some simulation parameters (simulation type, output frequency etc.) and then MD simulates the dynamics of the system.

At the time of this writing, there are many MD packages (e.g. AMBER [30], Desmond [20], GROMACS [79], LAMMPS [97], NAMD [130] etc.) that are publicly available and widely used. They support various types of force fields (e.g. AMBER [134], CHARMM [104] etc.) and simulation types. But regardless to the specific package or force field model, force computation in MD generally involves computing force contribution of van der Waals, electrostatic (Coulomb) and various bonded terms, as shown in Figure 2·2 and Equation 2.1.

$$F_{total} = F_{bond} + F_{angle} + F_{dihedral} + F_{hydrogen} + F_{vanderWaals} + F_{electrostatic} \qquad (2.1)$$

Van der Waals and electrostatic forces are classified as **non-bonded** forces and the rest are classified as **bonded** forces. As we will see later in Section 2.1.2 and Section 2.1.3, non-bonded forces can be further divided into two types. **Range-limited non-bonded** force that comprises of van der Waals force and short-range portion of electrostatic force and **long-range non-bonded** force that comprises of the long-range portion of electrostatic force.

Since bonded forces only affect a few neighboring atoms, they can all be computed in $O(N)$ time, where N is the simulation size, which is the total number of particles in the system. Non-bonded terms originally have complexity of $O(N^2)$, but several algorithms and techniques exist to reduce their complexity. Some of these algorithms and techniques will be described in later subsections. In practice, the complexity of van der Waals force computation is reduced to $O(N)$ and, that of electrostatic force computation is reduced to $Nlog(N)$. Motion update and other simulation management tasks take $O(N)$ runtime. In a typical MD run on a single CPU core, most of the time is spent computing non-bonded forces. For parallel runs of MD, inter-node data communication becomes a dominant factor as the number of processors increases.

2.1.1 Periodic Boundary Condition (PBC)

Boundary conditions play an important role in MD simulation since they define the surroundings of the simulation system. The number of particles in a typical MD simulation is far smaller than that of a bulk system. A bulk system has particles in the order of 10^{23}, whereas even the largest MD simulation size contains only a few million particles [55]. A common practice in MD is to use periodic boundary condition (PBC) [105, 139]. Instead of having a hard wall at the boundary, this condition assumes that the same simulation box is replicated at the boundaries in every dimension. A particle in the system not only interacts with the particles within

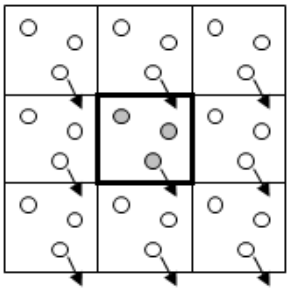


Figure 2·3: 2D representation of particles crossing simulation boundary under PBC

the defined simulation box, but also with the particles in the replicated images. As a particle moves in the original system, all of its replicated images also move in a consistent manner. When one particle leaves the original simulation box, another particle, its replicated image, enters the original simulation box. Figure 2·3 shows a simplified 2D representation of such boundary crossings. Since all the images are merely replications of the original particle, keeping the data of the original particle only is sufficient to run the simulation. Thus, PBC allows an MD simulation to be performed using a relatively small number of particles in such a way that the particles experience forces as if they were in a bulk solution.

2.1.2 Van der Waals (VdW) or Lennard-Jones (LJ) Force

Van der Waals (VdW) force is the attraction (or repulsion) force that works between a pair of atoms that are not bonded [139]. It is approximated by the Lenard-Jones (LJ) potential as shown in Figure 2·4. Equation 2.2 gives LJ force acting on particle i.

$$\overrightarrow{F}_i(LJ) = \sum_{i \neq j} \frac{\epsilon_{ab}}{\sigma_{ab}^2} \left\{ 12 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{14} - 6 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{8} \right\} \overrightarrow{r_{ji}} \tag{2.2}$$

Here, ϵ_{ab} and σ_{ab} are parameters related to particle types and r_{ji} is the distance between particle i and particle j.

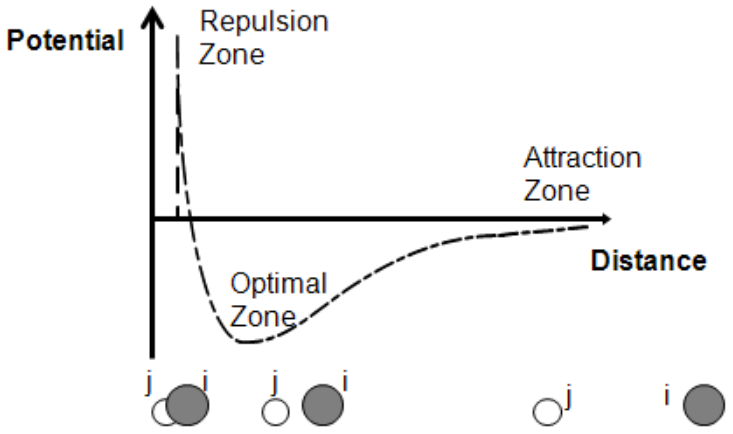


Figure 2·4: Lennard-Jones (LJ) potential [37]

A complete evaluation of VdW or LJ force requires evaluation of interactions between all particle-pairs in the system. The computational complexity is, therefore, $O(N^2)$, where N is the number of particles in the system. A common way to reduce this complexity is applying a cut-off. LJ force vanishes quickly with the distance of a particle-pair and is usually ignored when two particles are separated beyond some cut-off distance. A typical cut-off distance in MD of biological systems lies between 8-16 Angstroms. To ensure a smooth transition at cut-off, an additional switching function is often used too. Using a cut-off distance alone does not reduce the complexity of LJ force computation, because all particle pairs must still be checked to see if they are within the cut-off distance. The complexity is reduced to $O(N)$ by using techniques like cell-list method or neighbor-list method, which will be described in Section 2.1.6.

2.1.3 Electrostatic or Coulomb Force

The electrostatic or Coulomb force works between two charged particles and is given by Equation 2.3 [139].

$$\overrightarrow{F}_i(CL) = q_i \sum_{i \neq j} \left(\frac{q_j}{|r_{ji}|^3} \right) \overrightarrow{r_{ji}} \tag{2.3}$$

Here q_i and q_j are the particle charges and r_{ji} is the distance between particle i and j.

The evaluation of Equation 2.3 is computationally expensive ($O(N^2)$), because it requires evaluation of all possible pairs in the system. Several efficient algorithms have been developed to account for this force without actually interacting with all pairs [19, 44, 52, 129, 159]. Some of these will be introduced here with particular focus on PME (Particle Mesh Ewald) method, since this is the one widely used in MD.

Ewald Summation: Ewald Summation (or Ewald Sums) is an algorithm used to compute electrostatic forces in a system with periodic boundary conditions [3]. It was originally developed in 1921 to evaluate the electrostatic energy of ionic crystals [53]. Compared to direct calculation, it reduces the computational complexity from $O(N^2)$ to $O(N^{3/2})$.

For an N-particle periodic system, the Coulomb contribution to potential energy can be expressed using the Equation 2.4.

$$E_i = \frac{1}{2} \sum_{n}^{'} \sum_{(i,j)=1}^{N} \frac{q_i q_j}{|r_{ji} + nL|} \tag{2.4}$$

Here the prime on the summation means that the sum is over all periodic images and over all particles except $i = j$ if $n = 0$, meaning a particle does not interact with itself but interacts with all of its periodic images; L is the dimension of the unit cell.

The idea of Ewald Sums is to add and subtract a carefully selected Gaussian charge distribution to the system such that the subtracted Gaussian distribution cancels out the point charges of the system. This does not change the electrostatic potential of the system, but conveniently divides it into two major portions, as shown in Figure 2·5. A range-limited portion that can be computed using a cut-off in real

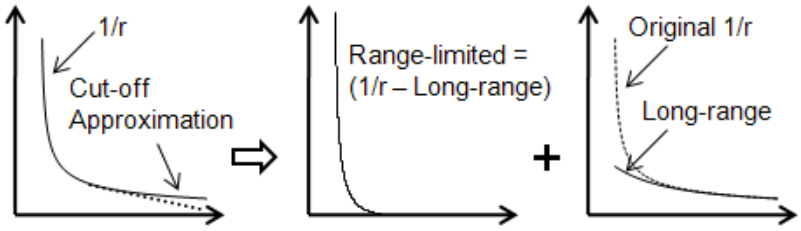


Figure 2·5: Splitting of electrostatic or Coulomb force into range-limited and long-range portions

space with complexity $O(N)$, and a long-range portion that can be computed using a cut-off in reciprocal or Fourier space with complexity $O(N^{3/2})$. In addition to these two, an inexpensive correction term is also required. While the exact derivation of the equation can be found in literature (e.g. [139]), here we provide the final form of Coulomb potential energy in Ewald Sums in Equation 2.5.

$$E_i = E^{(r)} + E^{(k)} + E^{(s)} \tag{2.5}$$

Here, $E^{(r)}$, $E^{(k)}$, $E^{(s)}$ are the real, reciprocal and self-correction terms respectively and they are given by Equation 2.6, Equation 2.7 and Equation 2.8, . The Ewald parameter α needs to be chosen appropriately to achieve $O(N^{3/2})$ complexity for the reciprocal term.

$$E^{(r)} = \frac{1}{2}\sum_{n}{}' \sum_{(i,j)=1}^{N} q_i q_j \frac{erfc(\alpha|r_{ji}+nL|)}{|r_{ji}+nL|} \tag{2.6}$$

$$E^{(k)} = \frac{1}{2L^3} \sum_{k\neq(0,0,0)} \frac{4\pi}{k^2} e^{-\frac{k^2}{4\alpha^2}} |\rho(\overrightarrow{k})|^2, \rho(\overrightarrow{k}) = \sum_{j=1}^{N} q_j e^{-i\overrightarrow{k}\cdot\overrightarrow{r_j}} \tag{2.7}$$

$$E^{(s)} = -\frac{\alpha}{\sqrt{\pi}} \sum_i {q_i}^2 \tag{2.8}$$

Particle Mesh Ewald (PME) Method: PME is an efficient technique that has

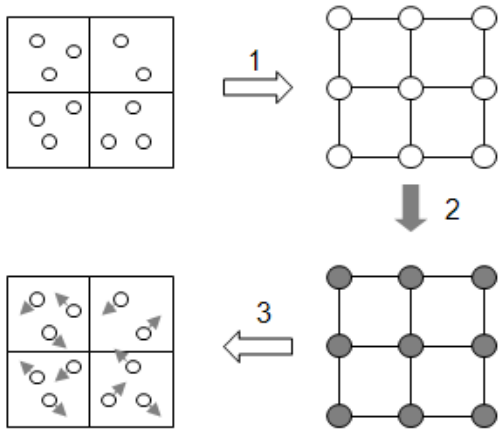


Figure 2·6: PME computation steps [63]

been widely used to evaluate the standard Ewald Sums due to its computational efficiency [44, 129]. It approximates the reciprocal or long-range portion of the Ewald Sums by a discrete convolution on an interpolation grid, using discrete 3D Fast Fourier Transformation (FFT), reducing the computational complexity from $O(N^{3/2})$ to $O(Nlog(N))$. Careful evaluation of the interpolation scheme and mesh size is required to achieve high simulation accuracy and speed. The basic procedure of PME is illustrated in Figure 2·6 and consists of three steps as follows [63].

1. Assign charge of particles to mesh points

2. Compute force with FFT and reverse FFT (rFFT)

3. Interpolate forces back to particles

The complexity of the first and third steps is $O(N)$ while that of the second step is $O(Nlog(N))$ which dominates the overall computation. It is worth mentioning that sometimes, to save computation time, the long-range portion of PME is only computed every few timesteps, e.g. every 4 timesteps. An improved scheme of PME, called Smooth PME (SPME) was reported in [52].

Multigrid Method: First introduced in the 1970's by Brandt [23], the multigrid method was originally used to solve partial differential equations and now has become an efficient technique to evaluate the electrostatic force [159]. For a system containing N particles, the computational cost of multigrid method is $O(N)$, whereas the direct calculation, the Ewald Sums, and PME are of order $O(N^2)$, $O(N^{3/2})$ and $O(Nlog(N))$ respectively. The general steps of the multigrid method are described as follows.

1. Interpolate and assign particles charge on a grid
2. Apply multigrid method to solve Poissons equation on the grid
3. Interpolate forces and energy from the grid domain back to particle space

Compared with the standard Ewald Sums method and its variants such as PME and SPME, the multigrid method not only reduces the cost of force computations but also offers several advantages. These include no PBC requirement, ease of parallelization, and no FFT. Therefore, the large communication overhead associated with the 3D FFT computation can be avoided.

Although the multigrid method helps reduce the computational complexity of force evaluation, to achieve a high-quality in energy conservation, it consumes more time compared to Fourier-based schemes such as Ewald Sums or PME. For a given accuracy, it was reported that the multigrid method was 1.85 times as expensive as PME method on a single processor although it could become competitive with, and eventually faster than, the PME method for a parallel system [143] . Most of the highly tuned MD software packages employ PME method or its variants to compute electrostatic forces.

2.1.4 Other Forces

In addition to the non-bonded forces, bonded interactions (e.g. bond, angle, dihedral in Figure 2·2) must also be computed every timestep. They have $O(N)$ complexity

and take relatively small portion of the total timestep. Bonded pairs are generally excluded from non-bonded force computation, but if for any reason, e.g. to avoid a branch instruction in an inner loop, non-bonded force computation includes bonded pairs, then those forces need to be subtracted accordingly.

2.1.5 Motion Update

Motion update in MD takes $O(N)$ runtime. Although there are various ways to update the motion of the system (e.g. predictor-corrector method etc.), most MD packages use a variant of leapfrog style method [139]. In their simplest forms, these methods yield coordinates that are accurate to third order in Δt, the interval between two consecutive timesteps. The formula can be derived from Taylor expansion of the coordinate variable $x(t)$.

$$x(t+h) = x(t) + h\dot{x}(t) + (h^2/2)\ddot{x}(t) + O(h^3) \tag{2.9}$$

Here t is the current time and $h = \Delta t$ between timesteps. $\dot{x}(t)$ is the velocity component and $\ddot{x}(t)$ is the acceleration. $O(h^3)$ represents the higher order terms of Δt. We can re-write Equation 2.9 as

$$x(t+h) = x(t) + h[\dot{x}(t) + (h/2)\ddot{x}(t)] + O(h^3) \tag{2.10}$$

The term multiplying h is actually just $\dot{x}(t+h/2)$. So Equation 2.10 now becomes

$$x(t+h) = x(t) + h\dot{x}(t+h/2) \tag{2.11}$$

Velocity can be obtained by subtracting from $\dot{x}(t+h/2)$ the corresponding expression for $\dot{x}(t-h/2)$.

$$\dot{x}(t+h/2) = \dot{x}(t-h/2) + h\ddot{x}(t) \tag{2.12}$$

The fact that coordinates and velocities are evaluated at different times is not a

real problem and can be avoided by employing these methods in a two-step form. For example, by applying Equation 2.13, followed by Equation 2.14.

$$\dot{x}(t + h/2) = \dot{x}(t) + (h/2)\ddot{x}(t)$$
$$x(t + h) = x(t) + h\dot{x}(t + h/2) \tag{2.13}$$

$$\dot{x}(t + h) = \dot{x}(t + h/2) + (h/2)\ddot{x}(t + h) \tag{2.14}$$

2.1.6 Cell-list vs. Neighbor-list

Every particle needs to interact with some of its neighboring particles to evaluate range-limited non-bonded forces. This requires a method of traversing through the neighboring particles of each particle. A naive approach will iterate through all particles in the system to find the neighbors of each particle. This will result in an inefficient implementation of $O(N^2)$ complexity. Here we present two efficient methods to accomplish this.

Cell-list: In cell-list method [80, 139, 144], a simulation box is first partitioned into several cells, generally cubic in shape. Each dimension is typically chosen to be slightly larger than the interaction cut-off distance. This means, for a 3D (2D) system, traversing through the particles of the home cell and 26 (8) adjacent cells will

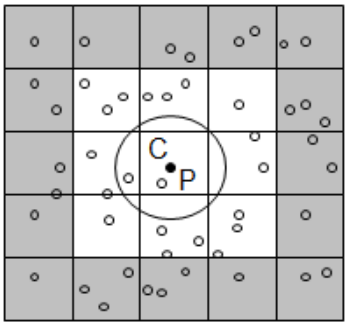


Figure 2·7: 2D illustration of cell-list for particle 'P' in cell 'C'

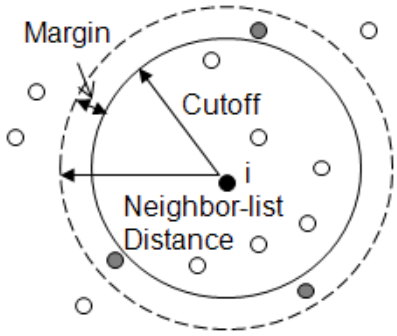


Figure 2·8: Neighbor-list sphere

suffice. If Newton's third law is used, then only half of the neighboring cells need to be checked. If cell dimension is less than the cut-off distance, then more number of cells will have to be checked.

The cost of constructing cell list scales linearly with the number of particles. Scanning each particle in the system and assigning it to a corresponding cell, once before evaluating the forces, is sufficient. The additional effort in constructing the cell list pays well, because the complexity of the force evaluation now becomes $O(N)$.

Neighbor-list: Although cell-list reduces the complexity of range-limited force computation, it still results in checking many more particles than necessary. For a particle we only need to check its surrounding volume of $(4/3) \times 3.14 \times R_c^3$, where R_c is the cut-off radius. But in the cell-list method we end up checking a volume of $27 \times R_c^3$, which is roughly 6 times larger than needed. This can be improved using neighbor-list method [139, 172]. In this method, a list of possible neighboring particles is maintained for each particle and only this list is checked for force evaluation. A particle is included in the neighbor-list of another particle if the distance between them is less than $R_c + R_m$, where R_m is a small margin. R_m is chosen such that, the neighbor-list also contains the particles which are not yet within the cut-off range but might enter the cut-off range before the list is updated

next. In every timestep, the validity of each pair in a neighbor-list is checked before it is actually used in force evaluation. Neighbor-list is usually updated periodically in a fixed time interval or when displacements of particles exceed a pre-determined value.

A neighbor-list can be constructed for all particles in $O(N)$ time using cell-list. As long as the neighbor-list is not updated too frequently, which is generally easy to ensure, this method reduces the range-limited force evaluation time significantly. The savings in runtime comes at the cost of extra storage required to save the neighbor-list of each particle. For most high-end CPUs, this is not a major issue.

As shown in Figure 2·8, for particle i, all particles within its neighbor-list distance, except the particle itself, are included in its neighbor-list and only particles residing between cut-off and neighbor-list distance are overhead. Although reducing the frequency of neighbor-list updates (by increasing the margin) could reduce the computational expense of constructing neighbor-lists, this would result in lower efficiency since more particles than actually needed will be added into the list.

2.1.7 Direct Computation vs. Table Interpolation

The most time consuming part of an MD simulation is typically the evaluation of range-limited non-bonded forces. That is why, a great deal of optimization takes place here. One of the major optimizations is the use of table look-up, instead of computing forces directly. This allows avoiding expensive direct computations like square root computation or $erfc$ function evaluation. This method not only saves the amount of computation time, but is also robust in incorporating small changes, e.g. applying switching function in VdW etc.

Typically, square of inter-particle distance (r^2) is used as an index of table look-up. The possible range of r^2 is divided into several segments and each segment is further divided into bins (intervals) as shown in Figure 2·9. For an M order interpolation,

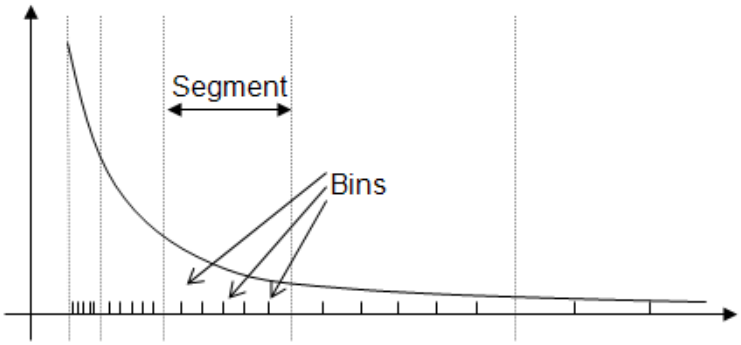


Figure 2·9: Interpolation using table lookup

each bin needs $M + 1$ coefficients and each segment needs $N \times (M + 1)$ coefficients, where N is the number of bins in the segment.

$$F(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \tag{2.15}$$

Accuracy increases with both the number of bins per segment and the interpolation order. Generally the rapidly changing regions are assigned relatively higher number of bins, and relatively stable regions are assigned fewer bins. Equation 2.15 shows a third order interpolation.

2.1.8 Parallelization of MD

Parallelization of MD requires careful consideration of a couple of things. First, how to distribute the responsibility of updating particle data among the processors. Next, how to distribute the computation of various forces among the processors such that the amount of data communication among processors is minimal. While the former may seem simple, meaning spatially decomposing the particles among processors seems sufficient, its interaction with the latter complicates the scenario. Further, different types of force computation requires different patterns of data communication. Range-limited non-bonded force computation requires data from

many neighboring particles, while bonded force computation requires data of only a few neighboring particles. On the other hand, 3D FFT-based computation of long-range portion of electrostatic forces requires all-to-all communication. Parallelization of MD is well-studied and most widely used MD packages achieve good scaling on CPU-based systems. A detailed discussion on this topic can be found in [20, 21, 83].

Here we first provide a brief discussion on parallelization of MD for range-limited force computation. This can be achieved in at least four different ways. These are Replicated Data (RD) method, Atom Decomposition (AD) method, Force Decomposition (FD) method, and Spatial Decomposition (SD) method. These basic methods can be used in combination to have more efficient hybrid methods, which are often the choices of production codes [20, 130].

The RD method was used in the earlier days where only a few processors were used. In this method, all data are replicated on each processor. While this allows computation of any force on any processor, the results need to be communicated to all processors. The amount of computation increases drastically with the number of processors, making it impractical for large number of processors.

The AD method is a simple alternative of RD where the array containing particle (atom) data is partitioned among the processors. Since this method does not consider the spatial location of the particles, it may incur large amount of data communication, limiting its practical applicability.

The FD method is a technique where pairwise force computation matrix is distributed among the processors in a block-wise fashion. For an $N \times N$ force matrix and P processors, the communication to computation ratio becomes $\sqrt{P}$. While this method performs better than the two previously described methods, load-imbalance may arise if the spatial locality of particles is not taken into

consideration.

In the SD method, the simulation space is divided into some sections and these sections are distributed among processors. Each processor is responsible for computing the forces of the particles in its assigned sections and also for updating those particle data. While the simulation space can be directly partitioned into P sections, P being the number of processors, a more common practice is to divide it into some fixed sized boxes, typically with a dimension larger than the cut-off distance, and distribute those boxes among processors.

Although SD method leads to a constant communication to computation ratio, in practice load-imbalance may arise if the number of boxes are not evenly distributed among processors. Production-level MD packages therefore often employs a combination of SD and FD methods to yield better performance scaling [20, 130]. A first level of partitioning is achieved using spatial decomposition - by assigning partitioned boxes to processors. This is followed by a second level of parallelization, where force computations (e.g., interactions of particles in two neighboring boxes) are further divided among the processors. There are many different ways that such a hybrid method can be implemented, leading to some number of variant methods [21, 83]. For example, force between two particles can be computed on a processor that owns at least one of the particles, or it can be computed on a processor that owns none of the particles. Overall performance of these methods depends on the target of simulation as well as the architecture of the simulation platform.

Now we briefly discuss parallelization of other force computations in MD. Bonded force computation typically scales well in SD method, since it only involves a few neighboring particles. Computing 3D FFT-based long-range portion of electrostatic force, on the other hand, requires all-to-all communication. It should be noted that,

for many practical simulations (e.g., less than 100K particles), it is not the net amount of computation but the fact that data are already distributed among processors, that motivates parallelization of 3D FFT. Parallelization of 3D FFT is primarily based on slab-decomposition or pencil-decomposition. The former decomposes the computation in one dimension only, allowing lower level of parallelism but less amount of data communication. The latter decomposes the computation in two dimensions, therefore allowing higher level of parallelism but more amount of data communication.

2.2 Hardware Acceleration of MD

2.2.1 ASIC Acceleration

ASIC acceleration of MD goes back to the 1990's, when a group of researchers in Japan developed MD Engine, a collection of custom developed processors that worked in parallel to simulate a system [8, 168]. Each processor (called MODEL: MOlecular Dynamics processing ELement) had an embedded pipeline to calculate the non-bonded interactions. Forces and virials were calculated with sufficient accuracy for practical MD simulations and a speed-up of about 50x was achieved compared to an UltraSPARC-I processor Sun Ultra-2 (200 MHz), for an MD simulation of a Ras p21 protein molecule immersed in a water sphere (13,258 particles). Below we discuss two other works on ASIC-acceleration of MD, that are more recent.

MD-GRAPE: The MDGRAPE-3 system, also known as "Protein Explorer", is a special-purpose petaflops computer system with hardware accelerator for classical molecular dynamics simulation [117, 141, 166, 167]. It was developed in RIKEN (RIkagaku KENkyujo: Institute of Physical and Chemical Research), Japan in 2006, in collaboration with SGI Japan and Intel Japan. Its architecture is similar to its predecessors, the GRAPE (GRAvity PipE) system, which was originally developed

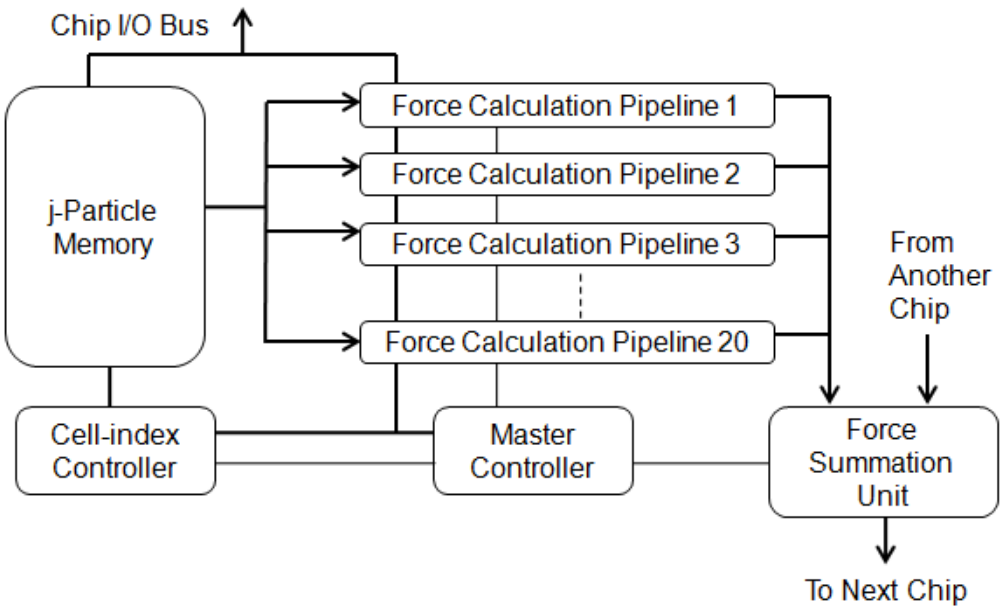


Figure 2·10: Block diagram of MDGRAPE-3 ASIC [166]

to solve gravitational N-body simulation and then extended to accelerate classical molecular dynamics simulations [50, 57, 81, 92, 118, 119, 120, 163].

MDGRAPE-3 system consists of 201 units of 24 MDGRAPE-3 chips, 64 parallel servers of Intel Xeon 5000-series processors (codename Dempsey), and 37 parallel servers having Intel Xeon 3.2 GHz processors with 2MB L2 caches. Each MDGRAPE-3 board consists of 12 MDGRAPE-3 chips, each of which has 20 force pipelines that are responsible for non-bonded force evaluation of MD only. The rest of the computation is left for the host computers. Each chip can fit up to 32,768 particles and deliver 165 GFlops @ 250MHz (230 GFlops @ 350MHz). One of the key innovations in the MDGRAPE-3 chip, as shown in Figure 2·10, is the capability of broadcasting particle data to the force pipelines, which reduces memory bandwidth requirement significantly.

At the time of development, the 130 nm generation MDGRAPE-3 chip was reported to be the fastest LSI for molecular dynamics simulation and the entire

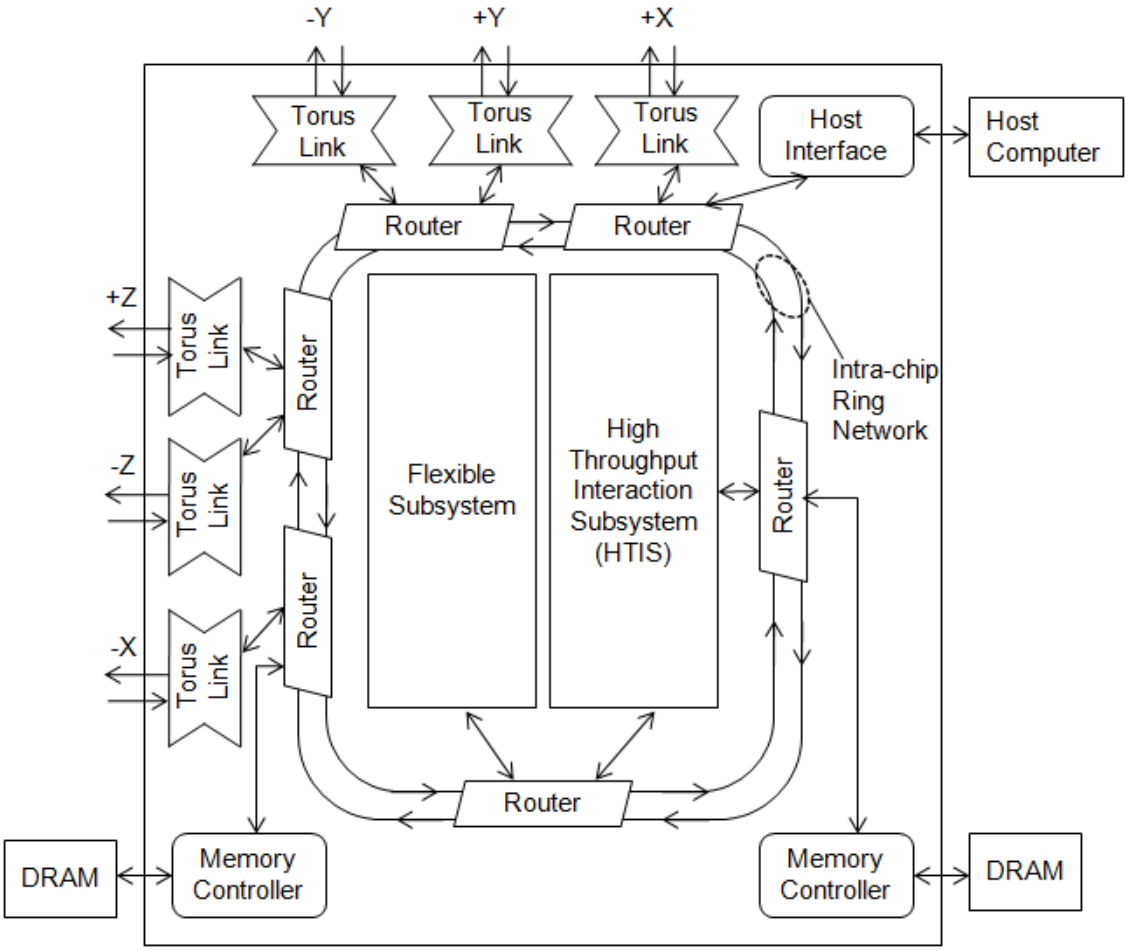


Figure 2·11: Block diagram of an Anton processing node [153]

system had a power consumption of 200 kilowatts per hour (19 Watt @ 350 MHz per chip).

Anton: Anton, developed at D. E. Shaw Research in New York in 2008, is a special-purpose supercomputer designed for molecular dynamics simulation of biomolecular systems [48, 153]. The 512-node version of it has reportedly achieved millisecond range simulation, two orders of magnitude beyond the previous state of the art [154].

As shown in Figure 2·11, each node of Anton includes an ASIC with two major computational subsystems, the high-throughput interaction subsystem (HTIS) that

computes range-limited pairwise interactions using thirty two 26-stage pipelines, and the flexible subsystem that takes care of the remaining computations. The flexible subsystem contains eight custom-designed geometry cores (GCs) for numerical computations, four Tensilica LX processors to control the overall data flow, and four data transfer engines. The Anton ASIC also has a pair of DDR2-800 DRAM controllers and a host interface that communicates with external host computer for input, output, and general control of the system. Anton nodes are connected in a toroidal topology and neighboring node-pairs support 50.6 Gbits/sec data communication in each direction, giving a total of 300 Gbits/sec of bandwidth per node. Anton ASICs are implemented in 90 nm technology and clocked at 485 MHz, with the exception of the force pipelines in HTIS, which are clocked at 970 MHz.

Some of the key innovations in Anton include a neutral territory partitioning method for range-limited non-bonded force computation, reordering communication data on-the-fly, and counted remote writing to reduce synchronization overhead. Anton employs a fixed-point arithmetic throughout the system which makes the computation deterministic, parallel-invariant and reversible. Anton's low-latency communication allows very fast end-to-end inter-node software communication (in hundreds of nanoseconds), which proves to be very useful for 3D FFT-based long-range electrostatic force computation [22, 47, 178].

2.2.2 GPU Acceleration

The tremendous computational power of recent day Graphics Processing Units (GPUs), together with their abundance, primarily due to the huge worldwide gaming market, has made them a viable option for scientific computing [123, 124, 126]. The introduction of high-level programming languages for GPUs, e.g. CUDA, OpenGL, has allowed rapid development of molecular dynamics applications using GPUs. Almost all publicly available MD packages now have their

GPU-accelerated versions [125]. While some of them have achieved significant speed-ups in some restricted conditions, most struggled to achieve reasonable speed-up for complex and practical cases [62, 162]. Scaling to multiple GPU nodes has become a particularly tough issue [131]. Below we discuss the GPU-accelerated versions of some of the prominent MD packages, in no particular order. It should be noted here that, although most work report the speed-up achieved using GPUs, speed-up is a relative term and not always a fair way to evaluate such a work. For example, a highly optimized CPU-only version is likely to have less speed-up using GPUs than the packages that are not so optimized. It should also be noted that the exact speed-ups depend on the specific benchmark, simulation settings and hardware in use. Therefore, comparing various packages is not a straightforward task.

NAMD (NAnoscale Molecular Dynamics): NAMD is developed and maintained by the Theoretical and Computational Biophysics Group of UIUC [83, 121, 130]. It is a simulation package that is known for scaling well on various platforms [94, 112]. It has a GPU accelerated version that achieves 6x - 7x speed-up on quad-core CPU/GPU combination, for large and complex benchmarks [131, 162]. Unlike the GPU versions of other packages, it scales reasonably well and has achieved up to 5.5x speed-up on a 60 CPU/GPU core cluster. It maintains original structure of the CPU-only version of NAMD and only accelerates the range-limited non-bonded force computation on GPUs. While this helps achieving better scalability, the acceleration is limited by Amdahl's law. In addition, Newton's third law is not applied and a lot of GPU threads are wasted because their respective particle pairs do not pass cut-off check.

GROMACS (GROningen MAchine for Chemical Simulations): GROMACS was originally developed in the University of Groningen, Netherlands, and is now extended

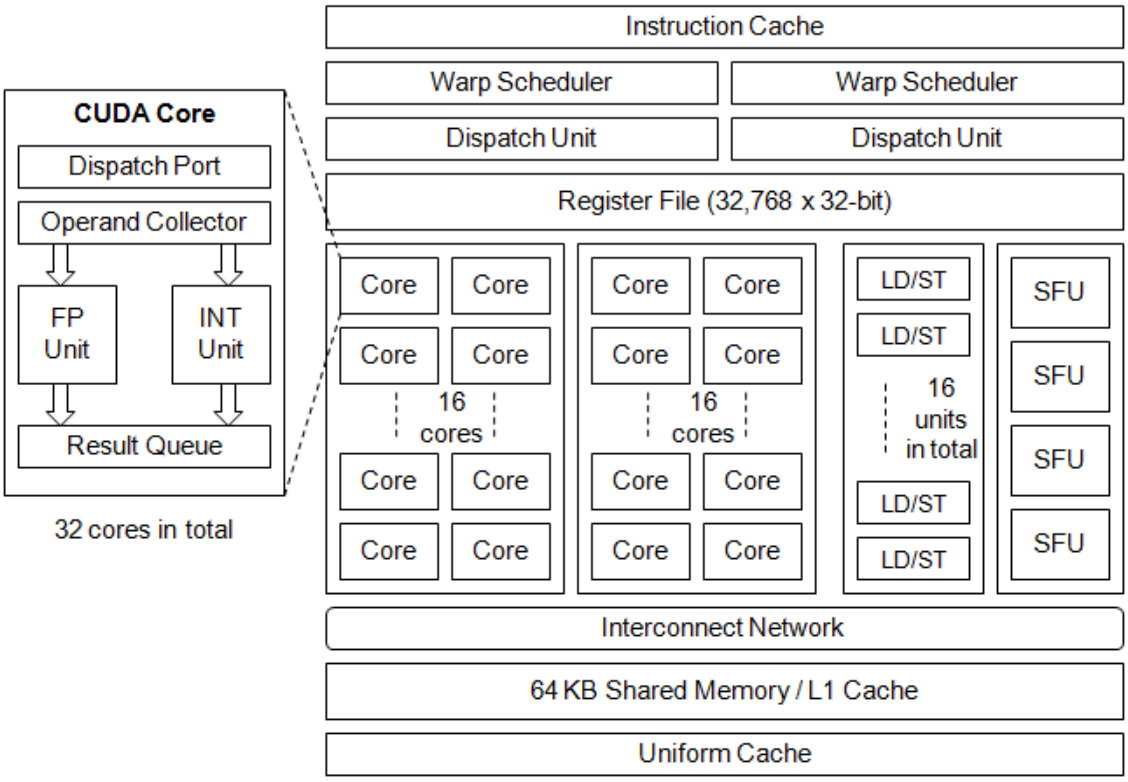


Figure 2·12: Streaming Multiprocessor (SM) of NVIDIA Fermi architecture. Fermi has 16 such SMs, a shared L2 cache and up to 6GB of DRAM [123]

and maintained in multiple places, including the University of Uppsala, University of Stockholm and the Max Planck Institute for Polymer Research [17, 79, 101, 171]. It is particularly optimized to achieve high utilization rate for every CPU core. Therefore, it does not scale as well as some of the other packages, although it may achieve the same performance using fewer nodes. At the time of this writing, GROMACS has a GPU-accelerated version that works with a single GPU [62]. Parallel runs do not work yet and getting actual speed-up using multiple nodes seems to be challenging. Using GTX280, 20x speed-up has been reported for small protein systems in implicit solvent using all-to-all kernels, where as for other setups involving cut-offs and PME, the acceleration is reported to be about 5x times relative to a 3 GHz CPU core.

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator): This package is distributed by Sandia National Laboratories, a US Department of Energy laboratory [97, 133]. Its GPU package and USER-CUDA package provide GPU support for neighbor-list build, particle-particle-particle mesh (PPPM) and range-limited non-bonded potential computation [26]. A modest speed-up of 4.47x is reported for simple atomic Lennard-Jones models, while for more complex model (e.g. Rhodopsin Benchmark) the speed-up is reported to range from 1.75x to 2.7x on 1-8 nodes on Yona, a 15 node cluster at Oak Ridge National Lab where each node has two six-core AMD Opteron 2435 processors running at 2.6 GHz and two Tesla C2050 GPUs [96].

AMBER (Assisted Model Building with Energy Refinement): AMBER MD package is distributed by the University of California, San Francisco [29, 30] and reports 2x to 3x speed-up over CPU version for complex benchmarks, e.g. DHFR [6, 7].

AceMD: AceMD is a high performance bio-molecular dynamics package specially optimized to run on NVIDIA GPUs and supported commercially by Acellera Ltd [1, 74, 75]. It achieves significant speed-up for reasonably complex systems, although scaling is limited to 3 GPUs at this point. It reports achieving performance equivalent to 256 CPU-core-NAMD using 3 CPUs and 3 GPUs, for DHFR benchmark. For a larger benchmark (ApoA1 with 92,224 atoms), the speed-up is close to 4x that of the GPU-accelerated NAMD.

HOOMD-Blue (Highly Optimized Object-oriented Many-particle Dynamics – Blue Edition): The HOOMD-blue development effort is led by the Glotzer group at the University of Michigan and it is probably the only MD package where all computational steps run on GPU [9]. It runs on single GPU and reports nearly 32x speed-up over LAMMPS CPU version for simple benchmarks. This work only

handles simple Lennard-Jones particles and may not be appropriate for bio-medical applications.

2.2.3 FPGA Acceleration

Early Work: The earliest reported work on FPGA-accelerated MD goes back to 2003, where the velocity and position calculations of the Velocity Verlet algorithm were mapped to an FPGA [176]. The calculations were done with IEEE 754 32-bit floating point arithmetic. Performance was presented for two types of FPGA platforms. The implementation on an Altera Stratix achieved 5.69 GFLOPS while the implementation on a Xilinx Virtex-II Pro achieved 4.47 GFLOPS.

In 2004, N. Azizi, et al. at Prof. Paul Chow's group at the University of Toronto implemented a preliminary MD system on Transmogrifier 3 (TM3) system [12, 95]. TM3 contained four Virtex-E 2000E devices that were connected to each other via 98-bit bi-directional buses. Each FPGA was connected to a dedicated 256 K x 64 bit external SRAM. The all-to-all LJ force calculation and Velocity Verlet algorithm were implemented on the FPGAs. Numerical computation was carried out by fixed-point arithmetic with various scaling factors and precisions. The LJ force was computed with table interpolation and the system was able to accommodate up to 8,192-particles. The system was validated with an academic C-based software MD simulation, MD3DLJ [14, 111], and on the order of 1% RMS error was reported for both force and energy evaluations. The details of error analysis about numerical precision and scaling factors, however, were not reported. Nor was the support of multiple particle types. The performance was reported to be 0.29x that of the original software code running on a PC with a 2.4 GHz Pentium 4, due to limited memory bandwidth and low clock speed of 26 MHz of the design. A 20x speed-up was projected if the implementation were scaled to more advanced FPGA devices with improved memory.

In 2008, a high performance FPGA-based MD system was designed and implemented in CAAD Lab at Boston University [63, 66, 67, 68, 69, 70]. The LJ and Coulomb forces were implemented on an Annapolis Microsystems WildstarII-Pro board, which had two Xilinx Virtex-II Pro XC2VP70-5 FPGAs [10, 11]. The design supported up to 32 atom types and 11,200 atoms and obtained 5.5x speedup over a single CPU (2.8 GHz Xeon) for a simulation of 8,000-particle system, using the MD package Protomol [110]. Up to 256K particles could be supported using off-chip/on-board memory [63]. One of the key achievements of this work was obtaining simulation accuracy comparable to that of the software-only version using a novel arithmetic mode, semi-floating point, and table lookup interpolation using the third-order orthogonal polynomial. Simulation accuracy was measured in terms of energy fluctuation and both software-only version and accelerated version had a value close to 0.014 [69].

USC: A few papers were presented by R. Scrofano et al. in Prof. Viktor Prasanna's group at the University of Southern California (USC) during 2004-2008 [146, 147, 148, 149]. In [148], an implementation of direct computation with double-precision floating point arithmetic was carried out to compute LJ force and potential. The design was synthesized using Synplicity Synplify Pro 7.2 and then placed-and-routed using Xilinx ISE 5.2 targeting the Xilinx Virtex-II Pro XC2VP125-7. Based on the placing-and-routing result, a throughput of 3.9 GFLOPS was reported using the two force pipelines that could fit on-chip. They compared their work to the CPU throughput for the same kernel and found it to be faster by a few times.

In [146], a similar force pipeline with single-precision floating point arithmetic was implemented on the SRC 6e MAPstation. The SRC 6e MAPstation has two 2.8 GHz Intel Xeon processors; a MAP processor, which has two Xilinx Virtex-II FPGAs available for custom designs; and a high-performance interface connecting them. The

force pipeline was implemented on one of the FPGAs and was responsible for LJ and Coulomb force evaluations. Neighbor-list was computed in host and was sent to the FPGA along with position data. A 2x speed-up was reported over a custom-written MD software.

A modification was made in 2006 to improve the accuracy of Coulomb force calculation [149]. Smooth Particle Mesh Ewald (SPME) method was implemented to replace the previous cut-off and shifted-force approximation for electrostatic force evaluation. Only the real space part of SPME was accelerated in hardware while the reciprocal space part was still executed in software on the host. Direct evaluations of non-bonded short-range forces were performed with single-precision floating point arithmetic, except $erfc(x)$ and e^{-x^2} which were approximated by table interpolation. Two test cases of 52K and 33K particles were reported and 2.7x - 2.9x speedup was achieved over the custom-written software code.

Maxwell: Maxwell is an FPGA-based computing cluster developed by the FHPCA (FPGA High Performance Computing Alliance) project at EPCC (Edinburgh Parallel Computing Centre) at the University of Edinburgh [15]. The architecture of Maxwell comprises 32 blades housed in an IBM Blade Center. Each blade comprises one Xeon processor and 2 Virtex-4 FX-100 FPGAs. The FPGAs are connected by a fast communication subsystem which enables the total of 64 FPGAs to be connected together in an 8 x 8 toroidal mesh. Each FPGA also has four 256 MB DDR2 SDRAMs connected to them. The FPGAs are connected with the host via a PCI bus.

In 2011, an FPGA-accelerated version of LAMMPS was reported to be implemented on Maxwell [86, 133]. Only range-limited non-bonded forces (including potential and virial) were computed on the FPGAs with 4 identical pipelines per FPGA. A speed-up of up to 14x was reported for the kernel (excluding data communication) on two or more nodes of the Maxwell machine, although the

end-to-end performance was worse than the software-only version.

This work essentially implemented the inner-loop of a neighbor-list-based force computation as the FPGA kernel. Every time a particle and its neighbor-list would be sent to the FPGAs from the host and then corresponding forces would be computed on the FPGAs. This incurred tremendous amount of data communication which ultimately resulted in the slowdown of the FPGA-accelerated version. They simulated a Rhodopsin protein in solvated lipid bilayer with LJ forces and PPPM method. The 32K system was replicated to simulate larger systems. This work, however, to the best of our knowledge, is the first to integrate an FPGA MD kernel into a full-parallel MD package. An 8-node performance figure was presented and the kernel (excluding data communication) retained its speed-up (end-to-end performance was still worse than the software-only version).

Boston University CAAD Lab: Several papers were published from CAAD (Computer Architecture and Automated Design) Lab at Boston University around 2010, describing a highly efficient FPGA kernel for range-limited force computation of MD [32, 33, 34, 35, 36]. The kernel was integrated into NAMD-lite [73], a serial MD package developed at UIUC to provide a simpler way to examine and validate new features before integrating them into NAMD [130]. The FPGA kernel itself was implemented on an Altera Stratix-III SE260 FPGA of Gidel ProcStar-III board [59, 60, 61]. As shown in Figure 2·13, the board consists of four such FPGAs, and is capable of running at system speed of up to 300 MHz. The FPGAs communicate with the host CPU via a PCIe bus interface. Each FPGA is individually equipped with a 256 MB on-board DDR II SDRAM (bank A) and 2x2GB DDR II memory (bank B and bank C) via SODIMM sockets.

The runtime of the kernel was 26x faster over the end-to-end runtime of NAMD, for ApoA1, a benchmark consisting of 92,224 atoms [37]. Electrostatic force was

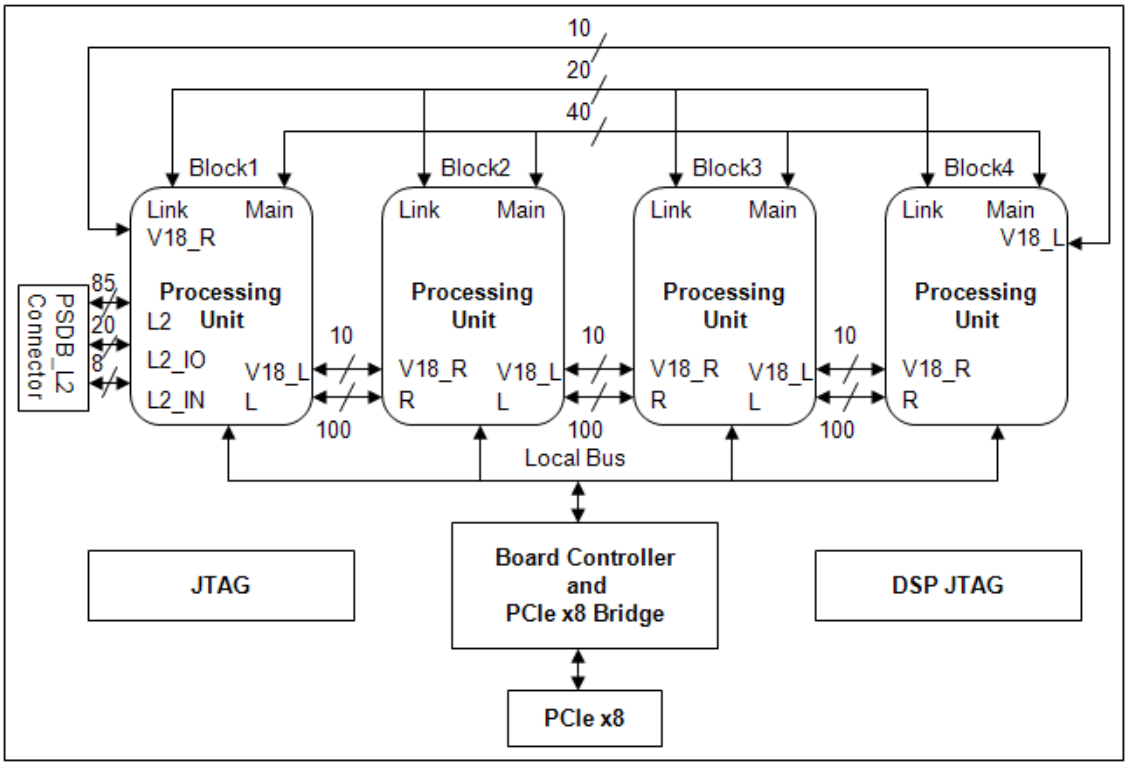


Figure 2·13: Gidel PROCStarIII system overview [59]

computed every cycle using the PME method and both LJ and range-limited portion of electrostatic force were computed on the FPGAs. Particle data, along with cell-list data and particle type data are sent to the FPGA every timestep, while force data is received from the FPGA and then integrated on the host. A direct end-to-end comparison with the software-only version was not done, since the software itself was not optimized for performance. Below, we discuss some of the key contributions of this work.

Multiple pipelines worked in parallel to compute forces of particles of a certain cell (home cell) as shown in Figure 2·14. Newton's 3rd law was used to avoid duplicate computations. Fixed precision was used in distance computation, while single-precision floating point was used for force computation. Force accumulation was done in fixed point, requiring conversion between fixed and floating point before

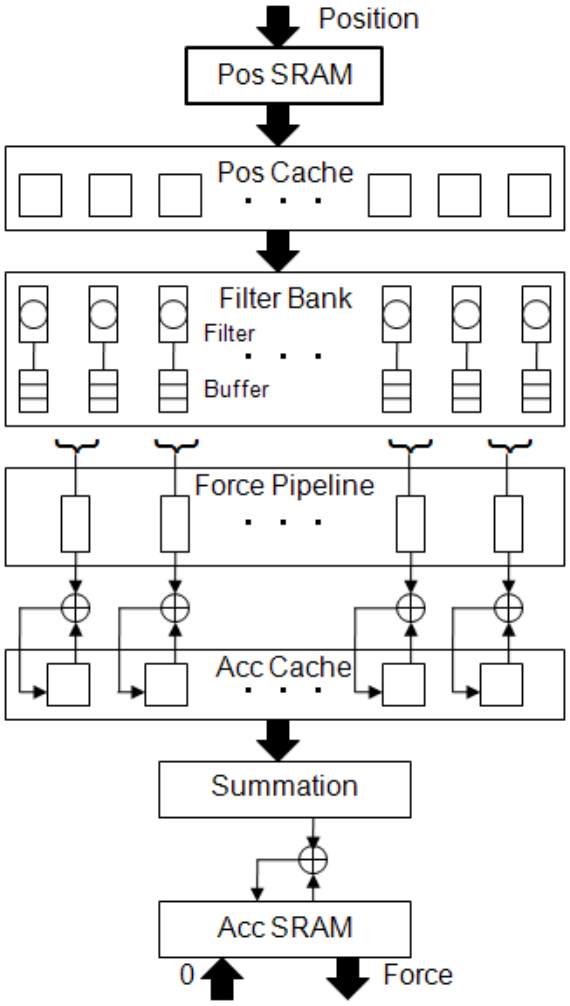


Figure 2·14: Schematic of the FPGA-kernel for range-limited non-bonded force computation, developed at CAAD Lab[33]

and after accumulation. Altera Floating-point Compiler was used to generate the floating-point force computation logic.

One of the key contributions of this work is a detailed study of filtering methods to improve the efficiency of cell-list method (recall discussion from Section 2.1.6 that, in cell-list method, many particle-pairs do not pass cut-off check). A number of filtering methods were compared and a reduced filtering method was used that required little amount of logic (especially DSP or multiplier blocks), yet was able to filter unwanted

particle-pairs efficiently, resulting in improved efficiency of the force pipelines.

Another contribution is a novel mapping scheme for load-balancing of the force computation pipelines. A "half-moon" scheme is proposed where a particle only computes forces for the particles on its right hand side. This method provided better load-balance at the cost of modest increase in data communication. In this method, each home cell is required to check 18 of its 27 neighboring cells in 3D (including itself) for range-limited non-bonded force computation.

Another contribution of this work is the use of Block RAM (BRAM) architecture of FPGAs to allow lower order table interpolation, saving resources for implementation of more force computation pipelines. This will be described later in Chapter 4, as this is a contribution of this dissertation work too.

Others: In 2005, Prof. Paul Chow's group at the University of Toronto made an effort to accelerate the reciprocal part of the SPME method on a Xilinx XC2V2000 FPGA [100]. The computation was performed with fixed-point arithmetic with various precisions to improve numerical accuracy. Due to the limited logic resources and slow speed grade, the performance was sacrificed by some design choices, such as the sequential executions of the reciprocal force calculation for x, y, and z directions; and slow radix-2 FFT implementation. The performance was projected to be a factor of 3x to 14x over the software implementation running in an Intel 2.4 GHz Pentium 4 processor.

In 2006, a simplified version of NAMD was accelerated on the SRC-6 MAPstation platform at NCSA, UIUC [91]. The modified NAMD code eliminated all bonded force computations. Only non-bonded range-limited forces were evaluated in hardware by table interpolation with single-precision floating point arithmetic. Several design choices were analyzed and implemented. It is reported that a 1.3x speedup can be achieved against the software by using both FPGAs on a single SRC-6 MAPstation

and 3x speedup can be obtained with a series-E MAPstation for a simulation of 92K particles.

In 2009, a design was proposed by a group from Dalian University of Technology, China, that used cell-list and filtering method to compute LJ force on the FPGA using table interpolation [71]. The primary target system consisted of a PC with a 2.66 GHz Pentium 4 CPU and Xilinx Virtex-II-Pro XC2VP70-5 FPGA. The software was Protomol, running on Windows XP. Although the paper reported about 11x to 12x speed-up for short runs of small simulations, it is not clear how precise the accuracy of the accelerated simulation was.

There were some work on FPGA implementation of the multigrid method for electrostatic force computation too [38, 39, 64, 65]. While 5x to 7x speed-up was reported in [64, 65], the rest did not have any implementation result.

2.3 Discrete Molecular Dynamics Simulation (DMD)

Discrete, or Discontinuous, Molecular Dynamics Simulation (DMD) uses simplified models; for example atoms are modeled as hard spheres, covalent bonds as infinite barriers, and van der Waals forces as a series of one or more square wells. This discretization enables simulation to be advanced by event, rather than timestep. Events occur when two particles cross a discontinuity in inter-particle potential. The result is simulations that are typically faster than timestep-driven molecular dynamics [46, 128, 136, 156, 179]. The simplicity of the models can be substantially compensated for by the capability of researchers to refine simulation models interactively [169, 170].

2.3.1 Overview of Discrete Event Simulation (DES) and DMD

MD is the iterative application of Newton's laws to ensembles of particles. It is transformed into DMD by simplifying the force models: all interactions are folded

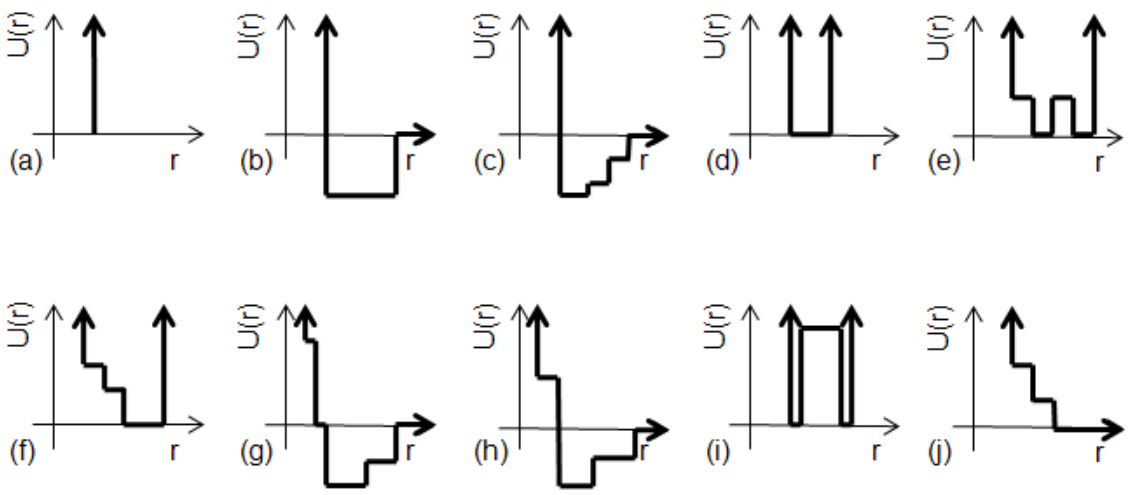


Figure 2·15: A collection of DMD potential models used in different studies (from [27, 136, 169]). (a) Simple hard sphere characterized by infinite repulsion at the sphere diameter. (b) Hard spheres with an attractive potential square well, zero interaction after a given cut-off radius. (c) A square well potential with multiple levels. (d) Single-infinite square well used for covalent bonds, angular constraints, and base-stacking interactions. (e) Dihedral constraint potential. (f) Hydrogen-bonding auxiliary distance potential function. g) Discretized van der Waals and solvation non-bonded interactions potential. h) Lysine-arginine-phosphate interaction potential in DNA-histone nucleosome complex. (i) Two-state bond used to create auxiliary bonds between backbone beads if they are also linked by a covalent bond. (j) Repulsive ramp with two steps for auxiliary interactions in hydrogen bond and with multiple steps to model liquids with negative thermal expansion coefficient.

into spherically symmetric step-wise potential models. Figure 2·15 shows a selection of the potentials described in the literature (see, e.g [27, 136, 169]). It is through this simplification of forces that the computation mode shifts from timestep-driven to event-driven.

Overview of DMD can be found in many standard MD references (e.g. Rapaport's book [139]) and DMD surveys [27, 136, 169]. A DMD system follows the standard DES configuration (Figure 2·16) and consists of the

- **System State,** which contains the particle characteristics: velocity, position,

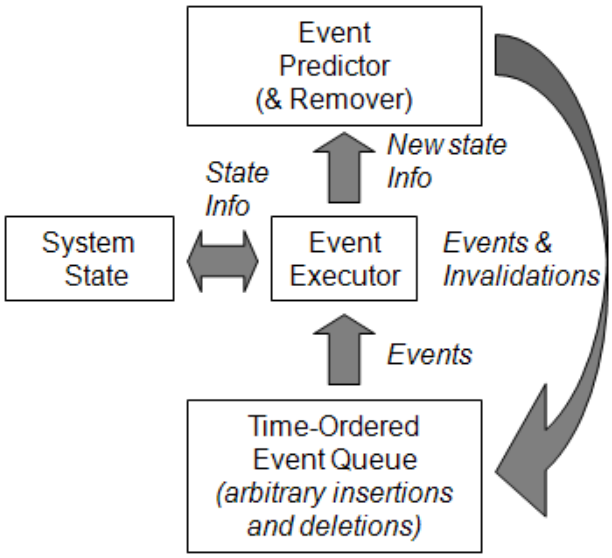


Figure 2·16: DES/DMD block diagram.

time of last update, and type;

- **Event Predictor (& Remover),** which transforms the particle characteristics into events, e.g. pairwise interactions;
- **Event Executor,** which turns the events back into particle characteristics; and
- **Event (Priority) Queue,** which holds events waiting to be processed, ordered by time-stamp.

Simulation proceeds as follows. After initialization, the event with the highest priority (involving, say, particles a and b) is popped off the queue and executed. Then, all other previously predicted events involving a and b, if any, are removed from the queue, since they are no longer valid. Finally, new events involving a and b are predicted and inserted into the queue. Then the next event, the current highest-priority event, is dequeued and processed. This loop continues until a user-defined end condition of the simulation is reached.

To bound the complexity of event prediction, the simulated space is subdivided into cells, similar to MD, as shown in Figure 2·7. Since both the number of particles per cell and the number of cells in a neighborhood (typically 27 in 3D) are fixed, the number of predictions per event is also bounded and independent of the total number of particles. One complication of using cells in DMD is that, since there is no system-wide clock advance (e.g. timestep in MD) during which cell lists can be updated, bookkeeping must be facilitated by treating cell crossings as events and processing them explicitly. Cell size is usually determined such that two particles have to be in the same or adjacent cells to interact with each other (cell dimension > particle interaction cut-off distance). Thus for any ***home*** cell, we ensure that checking only the 26 neighboring cells (in 3D) is always sufficient. Particles in other cells must enter these neighboring cells prior to interacting with home cell particles; such events are handled separately as ***cell-crossing*** events. It should be noted that, cell-dimension can be chosen to be smaller too, which will require slightly different bookkeeping.

2.3.2 Event Queuing Policy: Rapaport vs. Lubachevsky

One design issue that has received much attention is how many of the newly predicted events to insert into the event queue [93, 160]. The first algorithm by Rapaport [138] inserts all predicted events. The other, Lubachevsky's method [102], keeps only a single event, the earliest one, per particle. The reduced queue size, however, comes at a cost: whenever the sole event involving a particle is invalidated, the events for that particle must be re-predicted. This is done by converting the invalidated event into an ***advancement*** event of that particle; when the advancement event is processed, new predictions are made. There is thus a trade-off between the processing required to update the larger queue and that required for re-prediction.

2.3.3 Software Priority Queues

The basic operations for the priority queue are as follows: dequeue the event with the highest priority (smallest time stamp), insert newly predicted events, and delete events in the queue that have been invalidated. A fourth operation can also be necessary: advancing, or otherwise maintaining, the queue to enable the efficient execution of the other three operations.

The data structures typically are

- An array of particle records, indexed by particle ID;
- An array to save information about which particle belongs to which cell;
- An event pool;
- An event priority queue; and
- A series of linked lists, at least one per particle, with the elements of each list consisting of all the events in the queue associated with that particular particle [139].

Implementation of priority queues for DMD is discussed by Paul [128]; they have for the most part been based on various types of binary trees, and all share the property that determining the event in the queue with the smallest value requires $O(logN)$ time [108]. Using these structures, the basic operations are performed as follows.

Dequeue: The tree is often organized so that for any node the left-hand descendants are events scheduled to occur before the event at the current node, while the right-hand descendants are scheduled to occur after it. The event with highest priority is then the left-most leaf node. This dequeue operation is therefore either $O(1)$ or $O(logN)$ depending on bookkeeping. If the implementation is a binary search tree,

the worst case asymptotic bound is $O(logN)$, as long as the tree does not degenerate into a list.

Insert: Since the tree is ordered by time, insertion is $O(logN)$ (again, in the worst case and as long as the tree does not degenerate into a list).

Delete: For Rapaport-style queuing, when an event involving particles a and b is processed, all other events in the queue involving a and b must be invalidated and their records must be removed. This is done by traversing the particles' linked lists and removing events both from those lists and the priority queue. Deleting an event from the tree is $O(logN)$ (again, in worst case and as long as the tree does not degenerate into a list). A particular event generally invalidates $O(1)$ events, independent of simulation size, since cell subdivision method limits the maximum number of predicted events per particle.

Advance/Maintain: Binary trees are commonly adjusted to maintain their shape. This is to prevent their possible degeneration into a list and so a degradation of performance from $O(logN)$ to $O(N)$. With DMD, however, it has been shown empirically by Rapaport [138] and verified by us elsewhere, that event insertions are nearly randomly (and uniformly) distributed with respect to the events already in the queue. The tree shape is therefore maintained without rebalancing, although the average access depth can be slightly higher than the minimum.

2.3.4 Paul's Event Queue (PaulQ)

Much work has been done in optimizing the DMD event queue (see survey in [128]) with the design converging to what we call PaulQ. The event queue is based on work by G. Paul [128], which leads to a reduction in asymptotic complexity of priority queue operations from $O(logN)$ to $O(1)$, and a substantial benefit in realized performance.

The observation is that most of the $O(logN)$ complexity of the priority queue operations is derived from the continual accesses of events that are predicted to occur

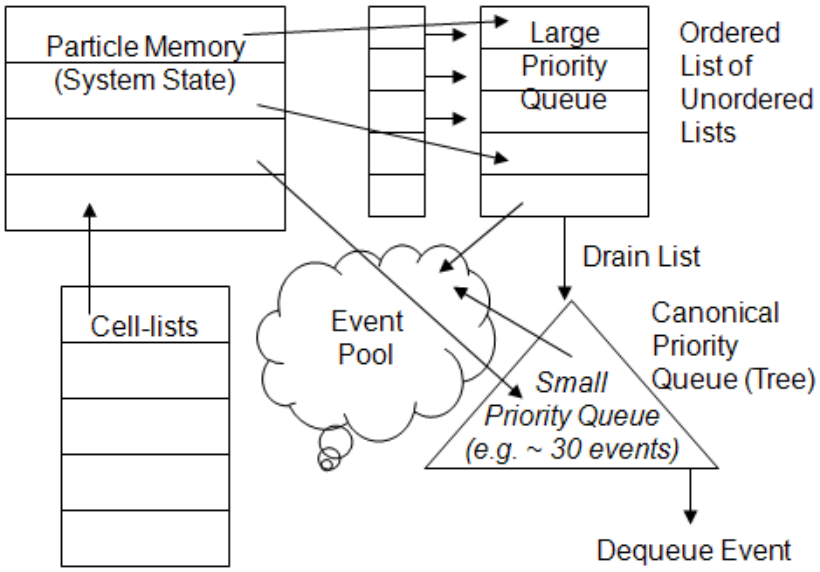


Figure 2·17: DMD data structures including Paul's two-level event queue (PaulQ)

far in the future. The idea is to partition the priority queue into two structures. This is shown in Figure 2·17, along with most of the other major data structures. A small number of events at the head of the queue, say 30, are stored in a fully ordered binary tree as before, while the rest of the events are stored in an ordered list of unordered lists. Also retained are the particle memory and the per-particle linked lists of events that are used for invalidations.

To facilitate further explanation, let T_{last} be the time of the last event removed from the queue and T be the time of the event to be added to the queue. Each of the unordered lists contains exactly those events predicted to occur within its own interval of $T_i ... T_i + \Delta t$, where Δt is fixed for all lists. That is, the ith list contains the events predicted to occur between $(T - T_{last}) = i \times \Delta t$ and $(T - T_{last}) = (i+1) \times \Delta t$. The interval Δt is chosen so that the tree never contains more than a small number of events.

Using these structures, the basic operations are performed as follows.

Dequeue: While the tree is not empty, operation is as before. If the tree is empty, a new ordered binary tree is created from the list at the head of the ordered list of unordered lists.

Insert: For $(T - T_{last}) < \Delta t$, the event is inserted into the tree as before. Otherwise, the event is appended to the ith list, where $i = \lfloor (T - T_{last})/\Delta t \rfloor$.

Delete: If the event is in binary tree, it is removed as before. If it is in the unordered list, it is simply removed from that list. It should be noted that, particle and event data are stored such that finding an event to delete takes $O(1)$ time.

Advance/Maintain: The ordered list of unordered lists is constructed as a circular array. Steady state is maintained by continuously draining the next list in the ordered list of lists whenever a tree is depleted.

For the number of lists to be finite there must exist a constant T_{max} such that for all $T, (T - T_{last}) < T_{max}$. In the rare case where this relation is violated, the event is put in a separate ***overflow list***, which is drained after all the lists have been drained once. Performance of this data structure (PaulQ) depends on tuning Δt. The smaller Δt, the smaller the tree at the head of the queue, but the more frequent the draining and the larger the number of lists.

2.3.5 Prior Work on Parallelization of DMD

What makes parallelization of DMD, or any DES, difficult, is the very inherent requirement of DES that all events must be processed in order. There are two ways to achieve parallelization without violating this requirement.

- **The conservative approach,** where an event is only processed when it is safe to do so, that is, when no causality violation will occur.
- **The optimistic approach,** where events are allowed to be processed without checking for safety first, but any error is later fixed by some sort of rollback

mechanism.

Previous DMD work were based on spatial decomposition of the simulation space, just like in regular timestep-driven MD; and used one of the two approaches mentioned above. An example of conservative approach, and also a discussion on difficulties in parallelizing DES in general, can be found in [56]. While this approach can be useful in cases where there is a safe window for executing events, e.g. in network simulation where a packet arrival time may have a lower bound, it is problematic in DMD, because event propagation time is not known here. Examples of optimistic approaches can be found in [103, 107, 113], where events are executed optimistically, and any error is recovered using a rollback to the last known correct state. While workable in one or two dimensions, spatial decomposition requires too much data communication (e.g. to handle events that are occurring at the boundary of partitions) for three-dimensional systems. For cubic decomposition, each thread has to exchange information with a large number of neighbors (26) for potential conflicts. Or, if decomposition is done by slices, then it must handle a drastic increase in the ratio of surface area to volume and so in the number of interactions per thread-pair. The best reported result was only about $\sqrt{P}$ scaling, where P is the number of processors working in parallel (less than 10x speed-up using 128 processors) [113].

Since the reporting of these works, event processing speed has increased dramatically, through advances in both processors and algorithms, especially when contrasted with inter-processor communication latency. This means that parallelizing DMD through spatial decomposition is likely to be even less efficient now. We believe this to be one of the main reasons why, prior to our work, all bio-medical work that used DMD, used a serial version of it.

2.4 Chapter Summary

In this chapter we provided necessary background on computations in MD and DMD, and reviewed related previous work. We note that there has not been any successful attempt to integrate an FPGA accelerator into a production-level MD package. We also note that prior work on parallelization of DMD had limited success only. We are not aware of any production-level parallel DMD application in use for 3D systems, at least for bio-medical studies, that predates our work.

Chapter 3

Parallel DMD (PDMD)

In this chapter we describe our work on parallelizing DMD. We begin with a discussion on the issues in parallelizing DMD. We then establish our baseline DMD serial code. This includes a description of the simulation models and experimental methods, as well as a discussion on selecting various parameters. Next we present our method of parallelization using event-based decomposition and three possible implementations of it, including synchronization techniques and other optimizations. Finally we provide scaling results and analyze them from both application and architectural point of view.

3.1 Issues in Parallelizing DMD

3.1.1 PDMD Hazards

Parallelizing DMD essentially means processing more events from the queue that just the highest priority one. This presents certain difficulties. Given three events E_{ex}, E_{pre}, and E_{can} where:

- E_{ex} is the event at the head of the queue being processed at time t,
- E_{pre} is an event predicted due to E_{ex}, and
- E_{can} is an event cancelled due to E_{ex}.

Then

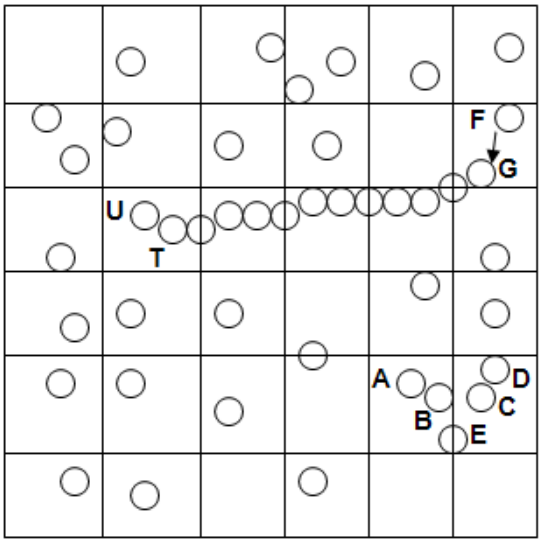


Figure 3·1: Events AB and CD cause BC and cancel BE. Event FG causes TU almost instantly and at long distance

- E_{pre} can be inserted at any position in the event queue, including the head,
- E_{can} can be at any position in the event queue, including the head, and
- another event E caused by E_{ex} (perhaps indirectly through a cascade of intermediate events) can occur at time $t + \epsilon$ after E_{ex} where ϵ is arbitrarily small and at a distance δ from E_{ex} in the simulation space where δ is arbitrarily large.

Examples of these occurrences are shown in Figure 3·1. In the lower part, events $E_{A,B}$ and $E_{C,D}$ occur at times t_0 and $t_{0+\epsilon}$. Previously predicted event $E_{B,E}$ gets cancelled, even though it is currently at the head of the queue. Newly predicted event $E_{B,C}$ will happen almost immediately and so it gets inserted at the head of the queue. The upper part of Figure 3·1 shows how causality can propagate over a long distance δ. After $E_{F,G}$, a cascade of events causes $E_{T,U}$ to happen almost instantly and on the other side of the simulation space. Although long-distance events such as in Figure 3·1 may appear to be rare, they are actually fundamental to polymer

simulations. The polymer forms a chain with rigid links. A force applied to one end – say, by an atomic force microscope that is unraveling a protein – creates exactly such a scenario.

These conditions introduce hazards into the concurrent processing of events. In each of the following cases, let E_1 and E_2 be the events in the processing queue with the lowest and next lowest time-stamps, respectively.

Causality Hazards occur when the processing of events out of order causes an event to occur incorrectly. For example, let event E_1 be such that its execution causes E_2 to be cancelled, either directly, or through a cascade of new events inserted into the event queue with time-stamps between those of E_1 and E_2. Then the sequence E_1, E_2 presents a causality hazard and should not be processed concurrently.

Coherence Hazards occur when predictions are made with stale state information. For example, let E_1 and E_2 be processed concurrently. Then even if there is no causality hazard, there may still be a coherence hazard. For example, a particle taking part in E_2 may be predicted to collide with a particle taking part in E_1, but only in the now stale system state prior to update due to the execution of E_1. Coherence hazards can exist only among events in the neighboring cells.

Combined Causality and Coherence Hazards occur as follows. Let a new event E_{new} caused by E_1 be inserted into the queue ahead of E_2 and not invalidate E_2, but still result in a coherence hazard. That is, E_{new} could change the state used in E_2's prediction phase, or vice versa.

Efficient detection and resolution of these hazards is a key to creating scalable parallel DMD codes.

3.1.2 Possible Approaches to PDMD

Parallelization of DMD can be achieved in at least three different ways.

Spatial Decomposition: The simulation space is partitioned into some number of sectors and one or more of these sectors are assigned to each thread. Events can be processed conservatively, letting no causality hazard occur ever; or optimistically, allowing some sort of rollback when causality hazard occurs. In any case, this approach becomes complex for 3D simulations. For cubic decomposition, each thread must exchange information with a large number of neighbors (26) for potential conflicts. Or, if partitioning is done by slices, then it must handle a drastic increase in the ratio of surface area to volume and so in the number of interactions per thread-pair. Spatial decomposition is likely to become ever more challenging as the latency ratio of inter-processor communication to event processing continues to increase.

Functional Decomposition: For any event, there is work that can be performed in parallel. In particular, there are likely to be predictions needed with respect to a number of nearby particles. The advantage of functional decomposition within events is that hazards are not an issue. The disadvantage is that the predictions can be executed in a few hundred nanoseconds and requiring extremely fine-grained invocation and synchronization of threads.

Event-based Decomposition or Task Decomposition: Some number of threads process events in parallel, dequeuing new events as the old ones are processed. This is the method we propose in this work. The advantage is that concurrency can be tuned to limit synchronization overhead (described in Section 3.3.3). The disadvantage is that some serialization cannot be avoided.

While previous PDMD work has been based on spatial decomposition

[56, 103, 107, 113], we are not aware of any such systems currently in use for 3D simulations, at least for bio-medical studies. We are not aware of any system based on functional decomposition. We believe our system to be the first to use event-based decomposition.

3.2 Establishing a DMD Serial Baseline

The primary purpose of this section is to describe the parameter selection of the serial baseline code and then present a profile of that code. In the process we update results of long-standing issues of cell size and queue insertion policy, as well as describe the interaction of the latter with the latest queue data structure. We also describe our simulation models, performance metric and hardware platforms.

3.2.1 Experimental Methods

The baseline code is by Rapaport and is described in Chapter 14 of [139]. This code is highly efficient being written in C in a "FORTRAN-like" style and including standard optimizations (such as those described in [31]). All modifications were also written in C and compiled using gcc (v4.2.4) with O3 optimization. We augmented the baseline code to support:

- The Lubachevsky insertion policy (in addition to Rapaport's),
- Paul's data structure, and
- Spherically symmetric potentials (square-well potential).

The event insertion policy and data structure modifications were validated against the original code and square-well potential was incorporated into the validated version. The new potential was verified through checks of internal consistency and of conservation of physical invariants.

Performance was measured on two platforms, an 8-core Intel machine and a 12-core AMD machine. The 8-core machine was used for all initial measurements, e.g. profiling of the serial code, comparing event-queuing scheme etc., and other analysis. The 12-core machine was only used to measure scalability. Below are the detailed configuration of the machines.

- 8-core Intel machine: A 64-bit, 2-processor, 8-core Dell Precision T-7400 Workstation with 4GB of RAM. Each processor is a quad-core Intel Xeon CPU E5420 (Harpertown) @2.50GHz. This was built with a 45nm process, has a Penryn microarchitecture, 32KB L1 I-Cache, 32KB L1 D-Cache, and two 6MB L2 caches, each shared by two cores. The operating system was Ubuntu Linux (version 8.04).

- 12-core AMD machine: A 64-bit, 2-processor, 12-core AMD Magny-Cours Server with 16 GB of RAM. Each processor is a 6-core AMD Opteron CPU 6172 (Istanbul) @2.10GHz. This was build with a 45 nm process, has a Bulldozer architecture, 64KB L1 I-Cache, 64 KB L1 D-Cache, 512KB L2 Cache, 6 MB of L3 cache shared by the 6 cores. The operating system was GNU/Linux (version 2.6).

DMD simulations are generally evaluated in terms of events computed per unit time. For clarity, we count only **Payload** events. These are pairwise events that involve two particles crossing discontinuities in potentials, including repulsive collision between two hard spheres (as shown in Figure 2·15). **Overhead** events are needed only to ensure correct simulation and maintain data. There are two such event types.

- **Cell-crossing:** When a particle crosses the boundary of a cell. This is present in all models.

- **Advancement:** This is required only if we implement Lubachevsky-style event queuing [102] where only the earliest event for each particle is queued. If that earliest event $E_{a,b}$ for a particle a is a pairwise payload event, but the other participant b is involved in another event $E_{b,c}$ before $E_{a,b}$ takes place, then the event $E_{a,b}$ is turned into an advancement event E_a for a. During the processing of E_a, the position of a is updated and new events are predicted.

In all experiments we simulated 10 million payload events, but in general, performance is independent of simulation time beyond a brief initialization phase. Following standard procedures (see, e.g., [139]), the particles were initially distributed uniformly in a 3D grid. The simulation box size was determined from the density and the number of particles. Particles were assigned velocities in random directions, but with a fixed magnitude depending on the temperature. Velocities were adjusted to make the center of mass stationary. Particles were then assigned to cells and events were predicted and scheduled for each particle. Runtime was measured after all initializations were done, and when actual event processing had begun.

3.2.2 Simulation Models and Conventions

Various models exist to accommodate molecular systems of differing complexity, flexibility, and desired resolution of the system of interest. They all have in common, however, the use of spherically symmetric step potentials, some of which are shown in Figure 2·15. Somewhat surprisingly, DMD simulator throughput (in events/second) is affected only marginally by model complexity. For example, the difference in throughput between simulations using a simple square-well, shown in Figure 2·15b, and complex square-wells, shown in Figure 2·15c and Figure 2·15g, is negligible (see Section 3.4.1). The reason is that the added model complexity is

processed using a switch/case statement to identify the correct discontinuity, which requires only a few instructions. A similar observation is made for per-particle differences in step functions, including particle radius. As a result, the simulation throughput also does not materially change as a function of number of particle types in the simulation, or whether some particles are covalently bonded or not. Some factors that do affect throughput are the number of particles, the radius of the furthest discontinuity from the particle center, and the simulation density.

As a consequence, for this study, instead of parallelizing any particular model in use, we chose, without loss of generality, a generic simulation framework that encompasses the properties that have an effect on event throughput. Note that adding complexity, such as processing reactions rather than simple discontinuities in potentials, necessarily adds to the work needed per event and so improves the scalability of most parallelizations. In that sense the performance improvements reported here are lower bounds.

We use MD units in our study. The simulations are of identically sized hard spheres of unit diameter and unit mass. Simple square-well in the square-well model, the main target of this work, is of radius 2.5 unit, unless stated otherwise. Simulated time is also presented in MD unit time. Conversion from MD units to real units is immediate and a description with specific examples can be found in [139]. Systems have periodic boundary with wrap-around effects considered as necessary.

Variations in density and temperature are tested. For density, a liquid-like density of 0.8 is used by default, but there is little effect on performance until the density falls below 0.4 (see Section 3.4.1). Temperature variation has virtually no effect on relative performance (see Section 3.4.1). Cell lists are used to bound the complexity of event prediction, with cell size fixed at slightly larger than the square-well radius. The selection of cell size is described in Section 3.2.4. In the experiments we report

results for three different simulation sizes: 2K, 16K and 128K particles. There is little relative change in performance beyond 128K particles. The chosen parameters are typical for liquid simulation [139] and are sufficiently general to represent most of the biomolecular DMD simulations reported in the literature.

3.2.3 Selecting PaulQ Parameters

Two parameters, the number of linear lists (n) and the scaling factor ($s = 1/\Delta t$) must be chosen to specify the implementation of PaulQ [128]. The method described in Paul's paper to determine these parameters ends up requiring large amount of memory, due to having too many lists (example: list size of $35\text{x}10^6$ for 70K particles). For our simulations, we determined in a slightly different way that is much simpler and requires less memory. It should be noted that, as also mentioned in Paul's paper, the performance of PaulQ is only marginally sensitive to the choice of s. For example, a choice of s which results in a doubling of the number of events in the binary tree results in only one additional level in the tree.

Step 1. Fix the # of lists, n, based on the simulation size (number of particles).

For Rapaport policy: n = Simulation size $\times$ 64

For Lubachevsky policy: n = Simulation size

Thus, n is always set to be the same as the size of the event pool, which is the maximum possible number of predicted events at any given time. In parallel implementations, since events are deleted in a lazy manner, sometimes we may need to have more events in event pool than the maximum possible number of predicted events. However, such case was not observed in the simulations we performed.

Step 2. T_{max} (the maximum difference between the time associated with a newly predicted event and the current time) is determined using cell-crossing events only.

Step 3. The scaling factor s is determined using the following equation: $n = s \times T_{max}$.

Step 4. A few other neighboring values are tried for scaling factor and the best value

Table 3.1: PaulQ parameters computed for various simulation sizes and queuing policies

Simulation size	2K		16K		128K	
	Number of Lists	Scaling Factor	Number of Lists	Scaling Factor	Number of Lists	Scaling Factor
Rappaport	131072	1195	1048576	63	8388608	72
Lubachevsky	2048	2304	16384	13696	131072	41984

is chosen.

Values of the parameters determined this way is presented in Table 3.1. Figure 3·2 shows how implementing the PaulQ improved performance for the square-well model simulation, the main target of this work. For reference, we also present the result for a simple hard sphere model in Figure 3·3. As shown in these figures, the speed-up due to PaulQ was more significant for Rapaport style, since it originally had a larger tree size and more frequent access to the tree. Lubachevsky style already had smaller sized tree and less frequent updates, hence the improvement was less too. The average tree size was about 60 for the Lubachevsky policy and about 200 for the Rapaport policy for the square-well model.

We also examined reducing the number of lists with the more aggressive use of the "overflow list" (see Section 2.3.4 in Chapter 2). We found, however, that unlike hardware implementation of this algorithm [77], this optimization has little benefit here.

3.2.4 Selecting Cell Sizes

Selecting the cell size involves determining the optimal trade-off between the number of predictions per event (more with a larger cell size) and the fraction of overhead cell-crossing events (decreases with larger cell size). Setting the cell size to slightly larger than the cut-off radius ensures that all relevant events can be found in the

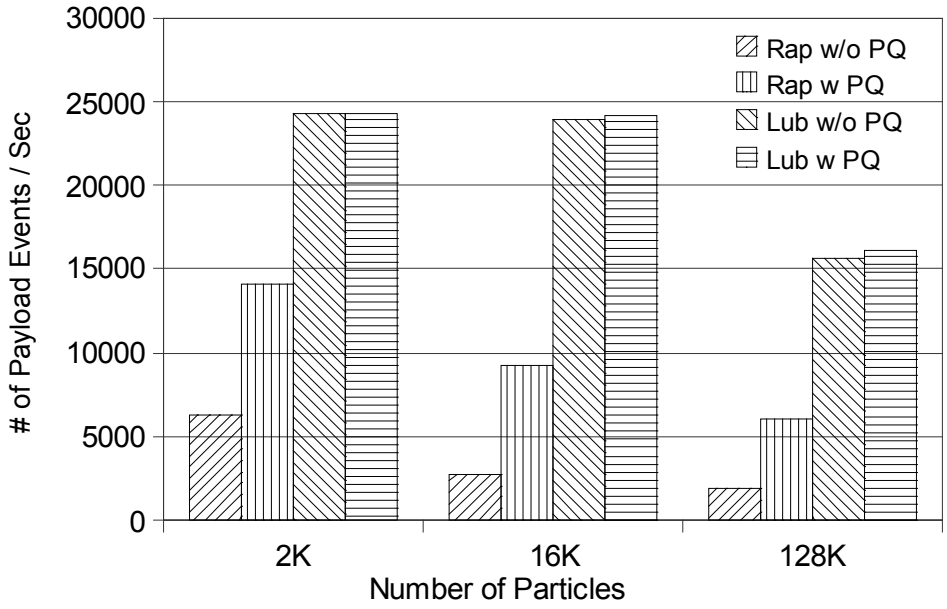


Figure 3·2: Performance of Rap Vs. Lub, with and without PaulQ (PQ), for square-well model of density 0.8

27 neighboring cells. For higher density systems, such as we assume here for liquid simulations, this is the cell size we use. The resulting proportion of cell-crossings to payload events is about 1:5 for the hard-sphere model, and significantly lower for the square-well model.

For low density systems, especially when they are simulating only hard spheres with no square-well potential, a substantially larger cell size is naturally optimal. We found, however, that, for such systems, a density somewhat lower than 1 particle per cell is preferred; and the cell size should be selected such that 3-6 particles fit in the 27-cell neighborhood.

3.2.5 Selecting Event Queuing Policy

There has been much discussion about the relative benefits of the two best-known queuing policies, those originated by Rapaport [138] and Lubachevsky [102],

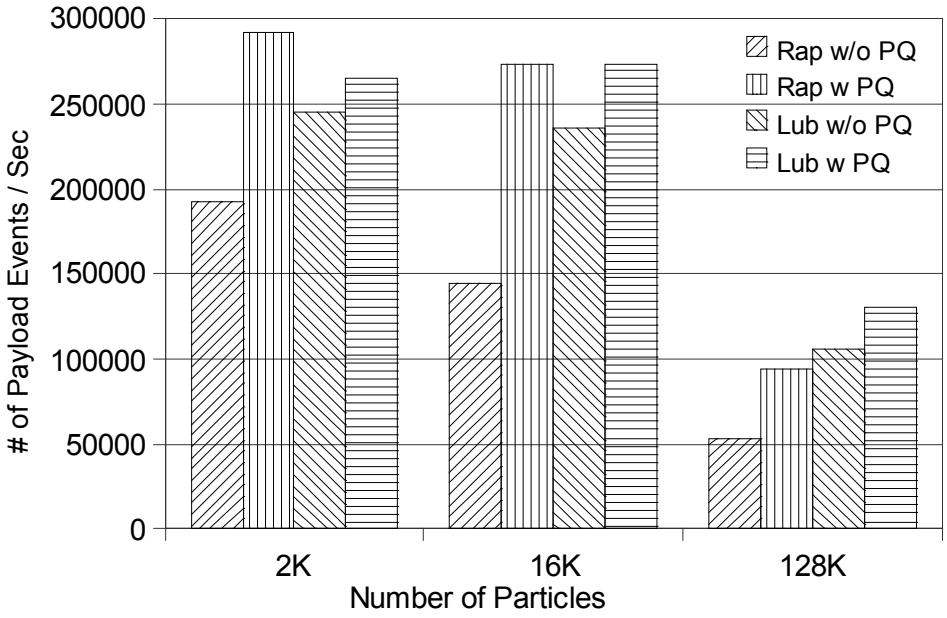


Figure 3·3: Performance of Rap Vs. Lub, with and without PaulQ (PQ), for simple hard sphere model of density 0.8

respectively, and reviewed here in Section 2.3.4 and Section 3.2.3. We find that the discussion is far from over and likely to continue as new algorithms, simulation models, and computer architectures are explored.

The Rapaport method queues all predicted events and also maintains a linked list of events for each particle to facilitate event invalidation. Since it saves all predicted events, cell-crossing events can be implemented efficiently. Unlike the Lubachevsky method, it does not require advancement events.

One advantage of the Lubachevsky method is that it has fewer events to queue, although with a small tree accessed with logarithmic complexity, the number of operations saved may not be large. There is some advantage, however, with respect to memory hierarchy performance in having a smaller working set size. Another advantage of the Lubachevsky method is that it avoids the linked lists in the Rapaport method.

There also exists a hybrid approach that saves all predicted events but queues only the earliest one [109]. This reduces the tree size, but still requires linked lists. The PaulQ data structure, however, diminishes the advantage of this method, and the linked list operations dominates. We therefore consider further only the Rapaport and Lubachevsky methods.

- Use of the PaulQ data structure favors Rapaport because the tree operation is no longer the most time consuming part. But Rapaport style queuing still requires linked lists to track all events of each particle. Figure 3·3 and Figure 3·2 show the improvement in both methods when the PaulQ is used.

- Simulation density matters. In low density simulations, particles travel farther between payload events, causing a higher proportion of cell-crossing events. This favors Rapaport because, in the Lubachevsky method, regardless of event type, all neighboring cells must be checked to predict new events. But in the Rapaport method, for cell-crossing events, only one-third of the neighboring cells need to be checked. That is, only the newly entered cell-neighborhood need to be checked.

- Models requiring a large number of predictions per particle, such as square-wells, favor Lubachevsky because it keeps only the earliest. Models requiring small numbers of predictions, such as simple hard-sphere, favor Rapaport because it does not have advancement events.

From our experiments, we have found that the Lubachevsky method performs better as the system becomes denser and larger, the Rapaport method for the converse. Since we are here more concerned with the former, we assume the Lubachevsky method for the remainder of this study.

3.2.6 Profiling the Serial Baseline Code

Here we provide a profiling of our final baseline code which uses PaulQ data structure and Lubachevsky-style event queuing method. Table 3.2 shows event statistics and serial runtimes. In all cases, the force model was the square-well and density was 0.8. Scaling results in Section 3.4 for the 8-core Intel machine are normalized to these serial runtimes (scaling results for the 12-core AMD machine are normalized to its respective serial runtimes).

In profiling the runtime of the serial baseline execution, we found the following breakdown.

- event execution, including state update, takes 1%;
- event commitment, including queuing operations, takes 3%; and
- event prediction takes 97%.

Table 3.2: Breakdown of event types for runs of 10M payload events using the serial baseline code

Number of Particles	Runtime (second)	Cell-crossings (%)	Advancements (%)	Payload Events (%)
2K	411	1.3	36.0	Repulsive collision: 13.9 Well entry: 17.6 Well exit: 15.6 Well bounce: 15.6
16K	414	1.3	36.2	Repulsive collision: 12.0 Well entry: 20.0 Well exit: 15.0 Well bounce: 15.5
128K	623	1.8	36.9	Repulsive collision: 6.1 Well entry: 28.3 Well exit: 13.9 Well bounce: 13.1

3.3 Parallelizing DMD through Event-based Decomposition

The main idea in our design is to process DMD in a single pipeline (as shown in Figure 3·4). That is, while a large number of events can be processed simultaneously, at most one event at a time is committed. As long the serial commitment time of an event is reasonably low, significant speed-up should be achievable. This expectation is motivated by the availability of shared memory in modern multicore processors and by the fact that a priority queue operation now only takes $O(1)$ runtime when PaulQ is used. Therefore, maintaining a centralized priority queue for events should not become a major bottleneck anymore.

Viewed another way, this design is of a microarchitecture that processes events rather than instructions: the logic is analogous to that used in modern high-end CPUs for speculative instruction execution. In this section we first describe how hazards and commitment are handled in a pipelined design (see [114] for details). Then we describe how this design translates conceptually into a multithreaded software version, and how we actually implement it. We end this section by describing some deeper software issues and how they can be addressed.

3.3.1 A Pipelined Event Processor

Commitment in a pipelined design (as shown in Figure 3·4) consists of the following steps: (i) updating the system state, (ii) processing all causal event cancellations, (iii) new event insertions, and (iv) advancing the event priority queue. As in a CPU, dependences—this time among events rather than instructions—combined with overlapped executions cause hazards. And as in a CPU, these hazards are compounded by speculation.

Causality Hazards: The problem is that a new event can be inserted anywhere in the pipeline, including the processing stages. But this cannot be allowed because then

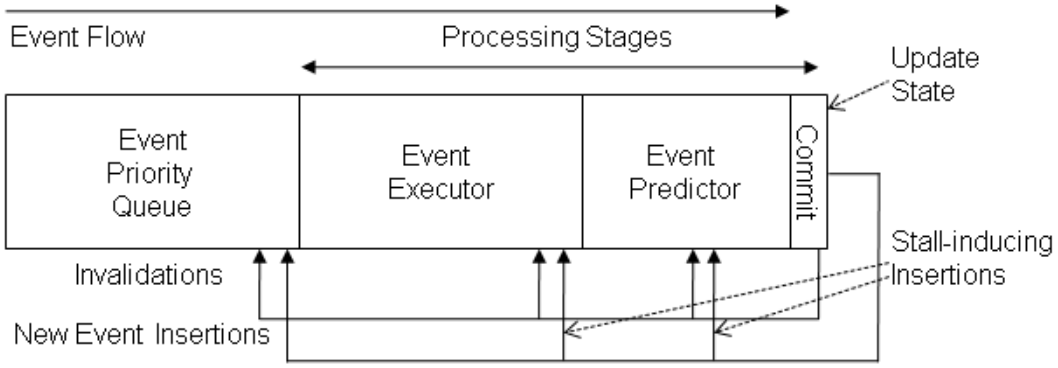


Figure 3·4: DMD with a dedicated pipelined event processor. The event queue is several orders of magnitude larger than the processing stages even for modest simulations

it will have skipped some of its required computation. Insertion at the beginning of the processing stages, however, results in out-of-order execution which leads to causality hazards. A solution is to insert the event at the beginning of the processing stages, but to pause the rest of the pipeline until the event finds the correct slot. This results in little performance loss for simulations of more than a few hundred particles.

Coherence Hazards: After an event E completes its execution, it begins prediction. The problem is that there will be several events ahead of E, however, none of which has yet committed, but which will change the state of the system when they do. This has the potential to make E's predictions incorrect because they may be made with respect to stale data (coherence hazard). One solution begins with the observation that E is predicting events only in its 27-cell neighborhood. It checks the positions of the events ahead of it in the predictor stages, an operation we call a neighborhood check, or *hood-check* for short. If the neighborhood is clear, i.e., it is *hood-safe*, then E proceeds, otherwise it waits. This check results in substantially more performance loss than that due to causality hazards, but it is still not large for large sized (e.g. 32^3 or more particles) simulation.

Combined Causality and Coherence Hazards: The problem is that an event E can be inserted ahead of events that have already begun prediction assuming they were hood-safe. The solution is as follows. As before, E must be inserted at the beginning of the processing stages. The added complication is that events in the predictor stages with time-stamps greater than E must restart their predictions. Since the probability of such insertions is small, however, this causes little additional overhead.

3.3.2 Conceptual Description of Software Implementation

The conceptual implementation of our method on software is shown in Figure 3·5.

- A FIFO is appended to the head of the event queue and contains the events that are currently being processed. This is analogous to the following processing components in Figure 3·4: the Event Executor, the Event Predictor, and Commit. While this FIFO is not necessary algorithmically, it is useful in visualizing how hazards and synchronization are handled.

- Each event in the FIFO is processed by an individual thread.

- The FIFO is ordered by time-stamp to facilitate handling of hazards, but processing is not otherwise constrained.

- Events are committed serially and in order. This allows the handling of all causality hazards.

- Events can be added to FIFO in two ways. They can be dequeued from the event queue and appended to the back of the FIFO. Or they can be inserted directly from the predictor.

- All coherence hazards are handled by checking whether any of the preceding events in the FIFO are in the same neighborhood. Since such hazards occur

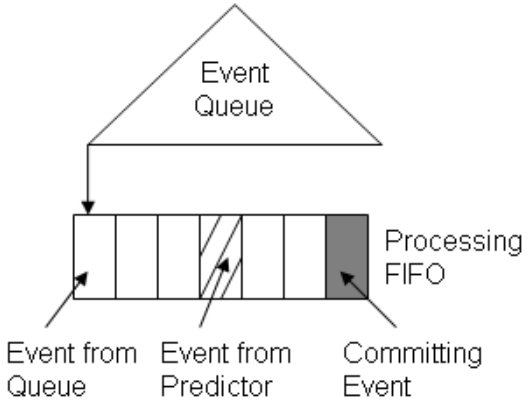


Figure 3·5: Parallel DMD implemented on software with an event FIFO

rarely (see Section 3.4.2), the hood-check is not done before a thread takes an event for processing, rather it is done right before committing. A small history of committed event is maintained for this purpose.

Event handling now contains the following tasks where commitment is explicitly separated from other processing tasks.

- Event Execution and Prediction: Calculate state updates, predict new events, and save these as temporary data.
- Synchronization: Wait until the event's turn to commit.
- Handling potential coherence hazards: Perform hood-checks; restart the event or update processing results as necessary.
- Committing and, potentially, discarding, the processing result.

3.3.3 Implementing PDMD through Event-based Decomposition

After translating these ideas into the standard DES framework (e.g. Figure 2·16), we obtain the design in Figure 3·6. There are several implementation issues which must be handled carefully if any speed-up is to be obtained: serial commitment, including

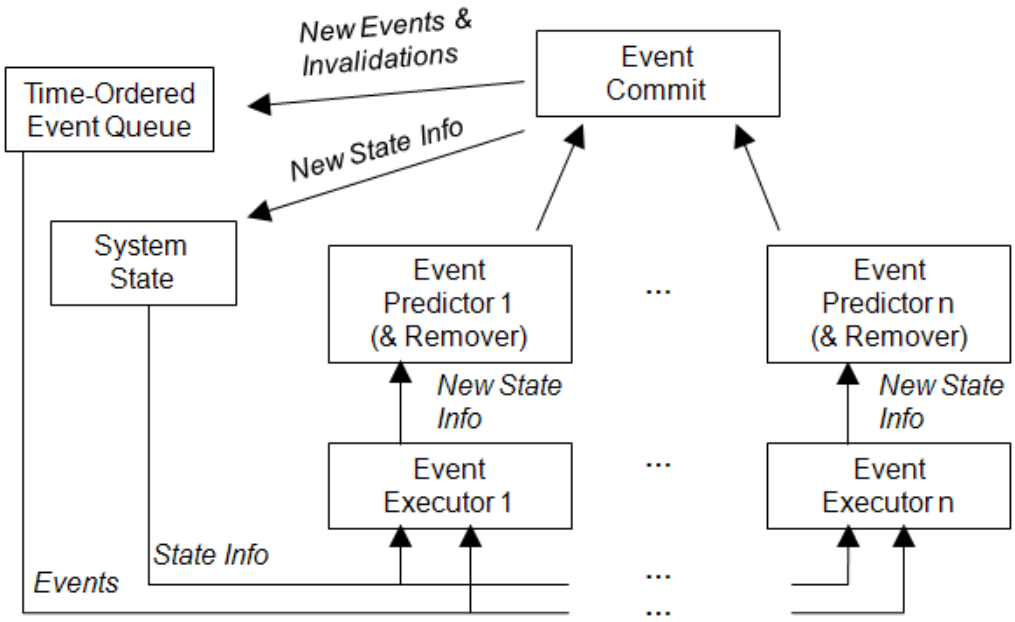


Figure 3·6: PDMD in the standard DES framework

updates of all of the data structures; locks on shared data structures, including the potential waiting time for threads to obtain new data; and contention in accessing a shared computing resource, e.g. shared memory.

PDMD through event-based decomposition can be implemented in various ways, depending on the synchronization scheme. Here we present three implementations where two implementations (the first and the third) proved to be more efficient than the other. For all implementations, number of available events at any time is assumed to be larger than the number of threads.

Implementation of Code 1: In Code 1 synchronization is done using a variable, EventToCommit, which holds the ID of the highest priority event, i.e., of the event at the head of the queue. Initially, Thread 0 is assigned the highest priority event and EventToCommit is set to the ID of that event. Since all threads will poll EventToCommit to check their turn, no other synchronization is necessary. A

thread updates the shared data structures and particle states only when it has processed the highest priority event. Thus only one thread commits at a time (updates shared data structures and particle states). A committing thread also dequeues the next highest priority event available from the event queue, before it updates the value of EventToCommit. This guarantees that the highest priority event is always assigned to a thread.

Once an event is assigned to a thread, the event will not be deleted and its turn to commit will eventually arrive. In the case where it has to be deleted, it is only marked as *canceled*; and at commit time it is discarded. This ensures that no thread ends up in an infinite loop. Hazards (and conversion to advancement events for the Lubachevsky method) are checked before committing the result of an event. If hazards (or conversions to advancement events) exist, then the event is reprocessed as necessary.

As a thread commits an event it notifies all other threads. This information is used by each thread to handle hazards and conversions. Every thread maintains a fixed-sized data structure (a list) for this information. If too many threads commit ahead of a particular thread causing an overflow in the list, then that thread simply reprocesses its event at commit time. Below is the pseudo-code for this implementation.

```
Main Thread{
    Initialize all data structures, including the event queue;
    EndCondition = False;
    EventToCommit = The very first event to be processed and committed;
    Invoke Worker Threads and assign them each the highest priority event available;
    //Only one thread will be assigned the 'EventToCommit' event
    //Main Thread will also continue as a Worker Thread
    Wait until all threads are done;
```

```
}

Worker Thread{
    While (Not EndCondition){
        Process assigned event;
        Wait until ( (EventToCommit = assigned event) or (EndCondition));
        //Only one thread will reach beyond this point at a time,
        //except when EndCondition is true
        If (EndCondition) return;
        If (Event has been cancelled){
            Discard event;
            Assign itself the next highest priority event that is available;
            EventToCommit = Next event to be processed and committed;
        }
        Else if (No ConversionToAdvancement and No hazard){
            Commit result;
            Assign itself the next highest priority event available;
            Update EndCondition;
            EventToCommit = Next event to be processed and committed;
        }
        Else {
            Update result or re-process the event as necessary;
            Commit result;
            Assign itself the next highest priority event available;
            Update EndCondition;
            EventToCommit = Next event to be processed and committed;
        }
```

```
    }
}
```

Implementation of Code 2: Code 1 has a simple synchronization method but requires threads to wait for their turn to commit. Due to load-imbalances (different types of events require different amount of processing time), unpredictable cache behavior, and time to update the common data structures, threads spend much time waiting. In Code 2, threads do not wait; rather, after processing their assigned events, they only mark the event as processed. They then acquire a centralized global lock and try to commit all available events that are already processed. Then, as before, they assign themselves the highest priority event that is not assigned yet, release the lock, and start processing the new event.

A centralized fixed-sized list of committed events is maintained. When an event is assigned to a thread, the current number of committed events is recorded. This number is used during commitment to determine hazards and conversions. As before, if too many events have been committed before a processed event can be committed, making it impossible to determine the hazards and conversions, then that event is simply restarted. Below is the pseudo-code for this implementation.

```
Main Thread{
    Initialize all data structures, including the event queue;
    EndCondition = False;
    Invoke Worker Threads and assign them each the highest priority event available;
    //Main Thread will also continue as a Worker Thread
    Wait until all threads are done;
}

Worker Thread{
```

```
    While (Not EndCondition){
        Process the assigned event and mark it as processed;
        Acquire Lock;
        //Only one thread will reach beyond this point at a time
        If (EndCondition) Release Lock and return;
        While (The highest priority event is marked as processed){
            If (Event has been cancelled){
                Discard event;
            }
            Else if (No ConversionToAdvancement and No hazard){
                Commit result;
                Update EndCondition;
            }
            Else {
                Mark the event as not-processed;
                It will be assigned to some thread again;
            }
        }
        Assign itself the next highest priority event that is available;
        Release Lock;
    }
}
```

Implementation of Code 3: Code 2 allows threads to continue immediately after they have finished processing their assigned event, but it requires continually acquiring a centralized lock. Since the processing time of an event is short (typically 3 - 60 microseconds), this requirement results in a substantial loss of performance. We

have found that instead of allowing all threads to commit and get new work, better performance can be achieved by assigning a Helper Thread for this purpose. This mechanism, however, requires synchronization between each helper-worker thread pair. But now, instead of using one centralized lock, we use separate locks for each helper-worker thread pair.

The implementation of the locks is done using flags and by allowing threads to spin on the values of their respective flags (somewhat similar to a ticket-lock). Two flags, threadGotWork and threadFinishedWork, are used for each helper-worker pair. The Helper Thread raises the flag threadGotWork for each thread once it has assigned an event to that thread. Meanwhile, each Worker Thread spins on its threadGotWork flag until it is raised. Once it is raised, the Worker Thread reads the event data and resets the flag. When the Worker Thread finishes processing the event, it raises its threadFinishedWork flag and again waits for its threadGotWork flag to be raised.

The Helper Thread checks the threadFinishedWork flags of all the threads. Once it is raised by any Worker Thread, the Helper Thread resets that flag, tries to commit the event, and assigns a new event to that thread. At this point, the Helper Thread raises that thread's threadGotWork flag and processing continues.

We declare these flags such that they reside in different cache blocks so that each thread can spin on their values independently without any false sharing. A centralized fixed-sized data structure of committed events is maintained; its processing is analogous to that in the previous codes. Below is the pseudo-code for this implementation.

```
Main Thread{
    Initialize all data structures, including the event queue;
    EndCondition = False;
    For (i = 0; i < threadCount; i++){
```

```
        threadGotWork[i] = 0;
        threadFinishedWork[i] = 1;
        threadEventID[i] = -1;
    }
    Invoke Worker Threads and assign them each the highest priority event available;
    //Main Thread will continue as the Helper Thread with threadNum = 0
    Wait until all threads are done;
}

Worker Thread (threadNum){
    While (True){
        While (threadGotWork[threadNum] = 0){} // wait for flag
        If (threadGotWork[threadNum] = -1) return;
        eventID = threadEventID[threadNum]; // get event
        threadGotWork[threadNum] = 0; // reset flag
        If (eventID != -1) Process Event;
        threadFinishedWork[threadNum] = 1; // raise flag
    }
}

Helper Thread (threadNum){
    WorkerCount = Number of worker threads;
    While (WorkerCount != 0){
        For (i = 1; i < threadCount; i++){ // exclude helper thread
            If ( threadFinishedWork[i] = 1){ // check flag
                threadFinishedWork[i] = 0; // reset flag
                If (threadEventID[i] != -1){ // mark this event as processed
```

```
                If (EndCondition){
                    threadGotWork[i] = -1; // signal end
                    WorkerCount = WorkerCount -1;
                }
                else{
                    threadEventID[i] = next highest priority event available;
                    //threadEventID[i] = -1 if none available
                    threadGotWork[i] = 1;
                }
            }
        }
    }//end of for loop
    Commit highest priority events that are processed;
    Handle hazards, conversions, and restarts;
    Update EndCondition;
} //end of while loop
}
```

3.3.4 Efficient Restart

Restarting an event every time there is a coherence hazard (alone or combined with a causality hazard) is inefficient. We optimize this by updating only the necessary portion of the prediction.

- In case a payload event has taken place in the neighborhood before the current event, it suffices to update the prediction for only those particles (one or both) that took part in that event and are in the same neighborhood.

- In case a cell-crossing event has taken place in the neighborhood before the

current event, if that new particle entered the neighborhood, then updating the prediction only for that particle suffices. If that new particle left the neighborhood, then (for our implementation) the event must be restarted. This is because, depending on the time of commitment of that cell-crossing event, it is possible that the current event may have used the incorrect cell-list (linked list of particles in the same cell) values.

- In case an advancement event has taken place in the neighborhood before the current event, theoretically, updating prediction result for that particle is not needed. This is because nothing about that particle has changed. But, to ensure compatibility with the serial output, we update the prediction for that particle. If hardware had infinite precision, this would not be necessary.

3.4 Results

This section is organized as follows. We begin by presenting the basic scalability results of the three implementations presented in Section 3.3.3, together with a qualitative analysis. In the following subsections we present more detailed analyses: determining the parallelism inherent in PDMD with event-based decomposition; basic modeling of the inherent architectural limitations; and a quantitative analysis of the most promising implementation, together with an experimentally validated analytical model that accounts for the details of the target architecture.

3.4.1 Scalability

The experimental setup and the baseline code are described in Section 3.2. The parallel versions have been created according to the description in Section 3.3. As described in Section 3.2.2, the energy model is of uniform hard spheres of radius 1 with simple square wells of radius 2.5. As discussed there, this model generalizes,

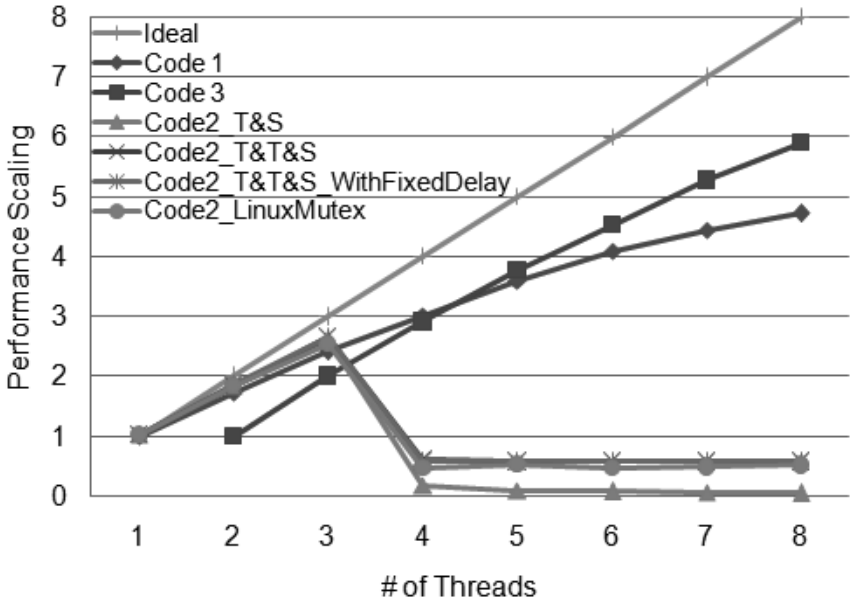


Figure 3·7: Performance scaling of the various Codes (simulation size = 128K and density = 0.8). For Code 2, performance with different locks is shown

with respect to relative performance, most models described in the literature.

All parallel versions were verified to have complete agreement with their respective serial versions. This consistency includes complete matches of all particle histories. The method used was as follows. All events were saved in order with participant information and time of occurrence. This was done for both serial and parallel versions which were checked to be exactly the same, including overhead events. Other physical parameters, e.g., energy, were also checked to have the same values. The codes are running and have been tested in both Windows and Linux environments (except some versions that were used to test system-specific implementation of locks).

The primary scaling results are shown in Figure 3·7, Figure 3·8, Figure 3·9, and Figure 3·10. As shown in Figure 3·7, the best speed-up is achieved by Code 3 with 5.9x for a 128K particle simulation using one helper and seven worker threads (on the 8-core Intel machine). Figure 3·10 shows scaling results for the 12-core AMD

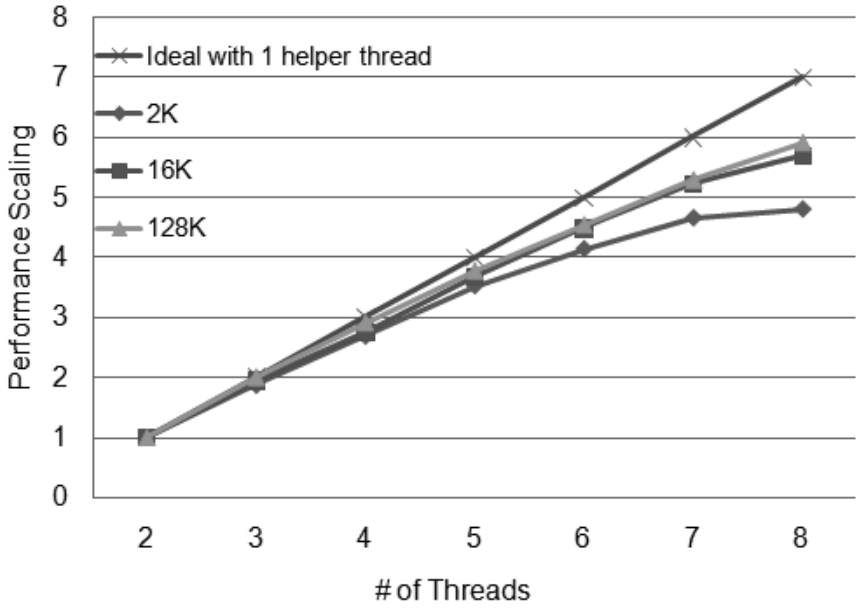


Figure 3·8: Performance scaling of Code 3 for different simulation sizes (density = 0.8)

machine where the best speed-up is 9.1x with one helper and 11 worker threads. From Figure 3·7 we can see that Code 2 is clearly not viable, Code 1 performs better for a smaller number of threads, and Code 3 performs better as the number of threads increases. This is because Code 3 uses a helper thread: initially the overhead is apparent, but it rapidly overtakes the other methods.

Figure 3·8 shows the scaling of Code 3 with respect to simulation size: not surprisingly, better scaling is achieved for larger simulations, and the benefit of parallelization diminishes somewhat with small sizes (2K). This is mostly because of an increase in coherence hazards (see Section 3.4.2). Since Code 3 appears to be the preferred method, we discuss its performance in detail in the next subsections.

Code 1 has two inefficiencies that result in threads waiting additional time to commit. One is uneven load-balance. Payload events spend more time in event-prediction than do in cell-crossing events or advancement events. This is because a

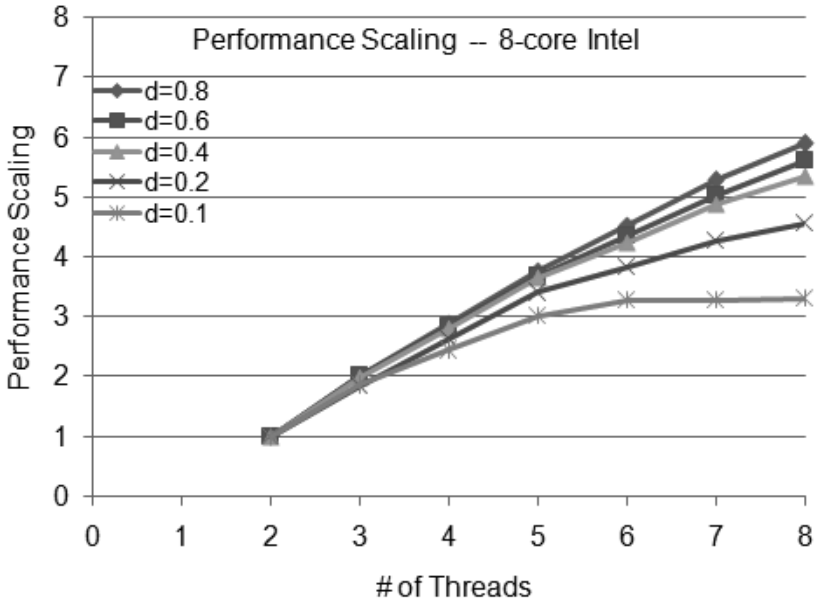


Figure 3·9: Performance scaling of Code 3 for different densities on the 8-core Intel machine (simulation size = 128K)

payload event predicts events for two particles, whereas the other types predict for only one particle. The other is increased cache misses and the random occurrence of those misses. Thus the order of start-of-processing does not guarantee the order of end-of-processing.

For Code 2, we note that the performance collapses suddenly with four or more threads. This is because of the bottleneck at the centralized lock and the small processing time per event. We performed extensive tuning of the lock, starting initially with the standard Linux function Mutex. We found that this Mutex has an undesirable system call and so replaced the function with various hand-tuned alternatives: Test&Set, Test&Test&Set, and Test&Test&Set_WithFixedDelay (see, e.g., Culler et al., [41]). As shown in Figure 3·7, none of these did significantly better than the original.

Figure 3·9 and Figure 3·10 show how relative performance varies with particle

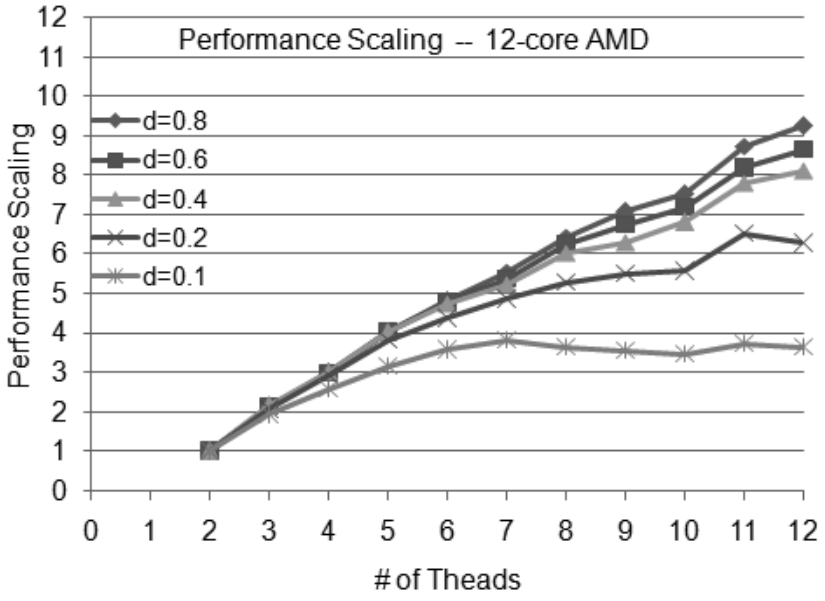


Figure 3·10: Performance scaling of Code 3 for different densities on the 12-core AMD machine (simulation size = 128K)

density. There is apparently little variation for densities higher than 0.4. The effect of density on scaling is that it changes the amount of work per event: As density increases, more particles must be checked for potential events, meaning more work to do in parallel.

A number of other parameters were tested but found not to affect relative performance:

- Various combinations of temperatures and particle densities with a temperature range from 0.4 to 1.6 and particle density from 0.1 to 0.8.
- Model complexity with number of steps in the square well ranging from 1 to 15.

3.4.2 Available Concurrency

In this subsection we measure the available concurrency in PDMD (with event-based decomposition) for the simulation models described. Event-based

decomposition enables all events to be executed concurrently as long as they are independent. Since this independence is hard to determine *a priori*—as the system state is changing continuously and unpredictably—all events but that at the head of the queue are essentially processed speculatively and so may result in work being wasted. There are two possible reasons for this: (i) The event may need to be invalidated due to a causality hazard (and possibly be converted into an advancement event), and (ii) the event prediction may need to be recomputed due to a coherence hazard.

Effect of Causality Hazards: Recall that causality hazards occur, causing the cancellation of a speculatively processed event E, when a particle P involved in E has been involved with a preceding event. In the Lubachevsky method this only happens if an event E_{new} involving P is inserted ahead of E after E has begun processing. We have examined the queue positions into which new events are inserted and have found that the positions are nearly uniformly randomly distributed. Moreover, the number of events in the queue is in the same order (more than half) of the number of particles being simulated. We find, therefore, that for likely numbers of threads T and particles N, the probability that an event will be part of a causality hazard $P_{causality} \simeq T/N$. This makes the loss of concurrency due to causality hazards negligible. A consequence is that few events are inserted into or deleted from the binary tree part of the PaulQ. Therefore, no additional FIFO-like data structure is needed. Instead, the binary tree is used directly to retrieve the highest priority events.

Effect of Coherence Hazards: Recall that coherence hazards occur when the predictions made during the processing of an event E may have been made using stale data. This occurs when an event preceding E is committed after E has begun processing and has occurred within the 27-cell neighborhood of E. We have examined

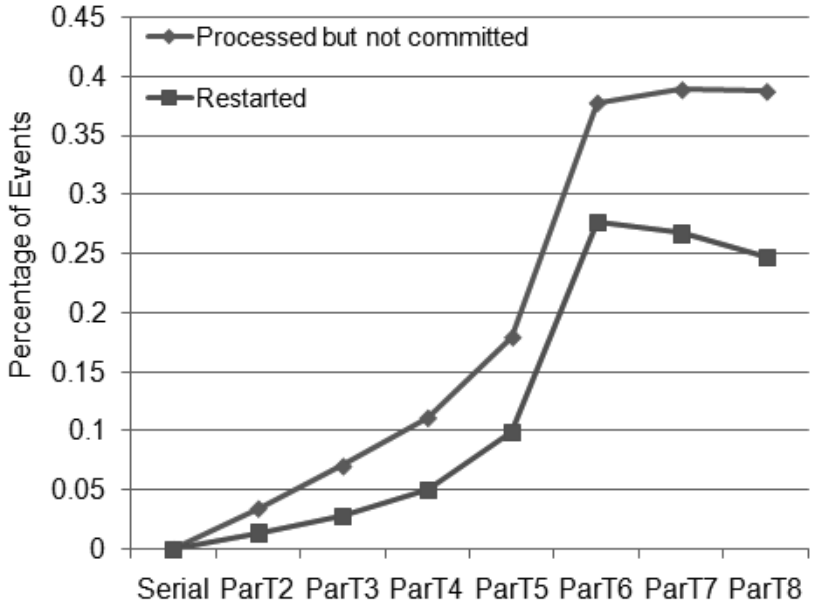


Figure 3·11: Events that are processed but not committed represent wasted effort only. Restarted events represent, in addition to wasted effort, the need for the payload effort to be serialized. Graphs are for Code 3 and 128K particles

the spatial distribution of committed events and found that their locations are nearly uniformly and randomly distributed. The probability that there will be a coherence hazard is therefore related to the number of threads T, the particle density ρ, and the ratio of the volume of the cell neighborhood to the overall simulation space. We use here 64 cells rather than 27 cells for cell neighborhood, because an event can involve particles two neighboring cells. Given a square-well radius of d_{sq} (cell dimension slightly larger than d_{sq}) and a number of particles N, then the probability of a coherence hazard is approximately $P_{coherence} \simeq 1 - (1 - \rho \times 64 \times d_{sq}^3/N)^{T-1}$. Plugging in typical values of $\rho = 0.8$, $T = 8$, $d_{sq} = 2.5$, and $N = 128K$, we obtain $P_{coherence} = 0.042$. This value of $P_{coherence}$, however, serves only as an upper bound on the number of restarts due to coherence hazard: most can be avoided by using the methods for efficient restart described in Section 3.3.4.

We now relate the theory to the actual implementation and effect on execution

time. Figure 3·11 shows the measured fraction of the events that are *processed but not committed* and the events that need to *restart*. The latter events are a subset of the former because every restarted event has been processed before it is restarted. Restarts are mostly due to coherence hazards, but a small fraction are also caused by causality hazard. The fraction that is *processed but not committed* includes both the events that were restarted and the events that were processed but canceled later due to a causality hazard.

Note that updating the prediction results and detecting the need to restart is handled by helper thread during commitment. If an event needs to be restarted, it is immediately processed and committed by the helper thread. This means that the restart latency is often hidden and worker threads can continue processing new events in parallel. These complex effects account for the non-linear behavior in Figure 3·11.

The most important conclusion from this subsection is that lack of available concurrency is likely to affect performance by only a fraction of a percent, and is not likely to affect scalability as much as architectural limitations (see Section 3.4.4).

3.4.3 Simple Model of Limitations on Scalability

In this subsection we present a simple analytical model for the limit on scalability. The two constraints are (i) serial commitment, and the associated synchronization overhead, and (ii) serialized memory access due to the shared bus. Each event-processing task has four components:

I_{cpu} = CPU portion of independent code (independent: can be done in parallel),

I_{mem} = Memory portion of independent code,

S_{cpu} = CPU portion of synchronization code (synchronization: cannot be done in parallel), and

S_{mem} = Memory portion of synchronization code.

Assuming that the application is not memory bound and that computation and

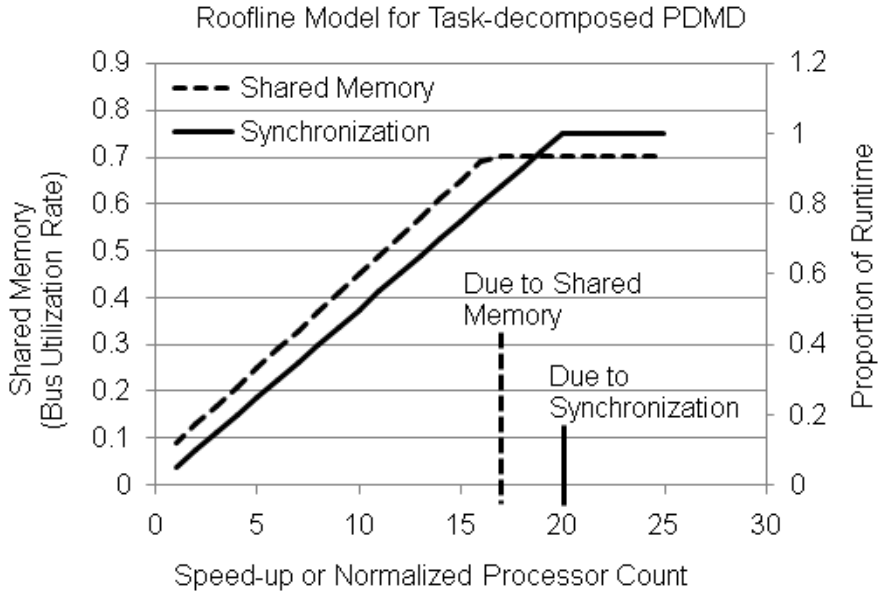


Figure 3·12: Roofline model for task-decomposed PDMD

memory access can be overlapped, processing time of an event by a single processor $= I_{cpu} + S_{cpu}$.

Constraint 1 – synchronization. For multiple processors P handling separate events, the I_{cpu} can be processed in parallel while the S_{cpu} must processed serially. Synchronization can be hidden as long as

$P * S_{cpu} \leq I_{cpu} + S_{cpu}$;

that is, $P_{SyncLimit} = (I_{cpu} + S_{cpu})/S_{cpu}$.

Constraint 2 – shared memory. Similarly, the memory components can be hidden until the memory system approaches saturation:

$P * (I_{mem} + S_{mem}) \leq I_{cpu} + S_{cpu}$;

that is, $P_{ShMemLimit} = (I_{cpu} + S_{cpu})/(I_{mem} + S_{mem})$.

If neither of the constraints is in effect, then the application scales linearly. Otherwise maximum scaling is $P_{SyncLimit}$ or $P_{ShMemLimit}$, depending on which is stronger.

From measurement, we find that S_{cpu} takes a little less than 5% of the total

time of event processing with a single processor. We note that the bottleneck due to $I_{mem} + S_{mem}$ can be represented by the increase in utilization rate of the shared bus. This was measured to be approximately 4% per thread. It should also be noted that bus utilization rate does not necessarily increase linearly with number of threads and a practical saturation point is well below 100%. Here we use 70% to be the saturation point of bus utilization. Therefore, for our target platform, when Constraint 1 dominates, the maximum linear scaling $P_{SyncLimit} = 20$; when Constraint 2 dominates, the maximum linear scaling $P_{ShMemLimit} = 17$. These limits are represented using the roofline model in Figure 3·12 [175].

3.4.4 Architectural Limitations on Scalability

To account for the difference between the achieved speed-up and a perfect scaling, we analyze in more detail the interaction between the application and the architecture of the target hardware. As shown in Figure 3·13, the processing time per payload event itself increases by about 25% as the number of threads is increased to 8. This increase in event processing time is the reason why a perfect scaling is not achieved. Of this increase, roughly 40% is due to synchronization overhead, including synchronization timing mismatch, while the rest 60% is due to the increase in cache misses and bus utilization.

Figure 3·14 shows total cache misses and rate of bus utilization for the entire simulation period for the different numbers of threads. The data were collected with VTune (vtune_linux_9.1) [106, 140] using separate simulations of size 128K. We observe that the count of total cache misses increases slightly with the number of threads. This indicates that there is neither significant data reuse by a single thread nor data sharing among threads; these would cause substantial increases and decreases in cache misses, respectively. More likely is that the modest increase is related to synchronization: more threads means more misses on explicitly shared

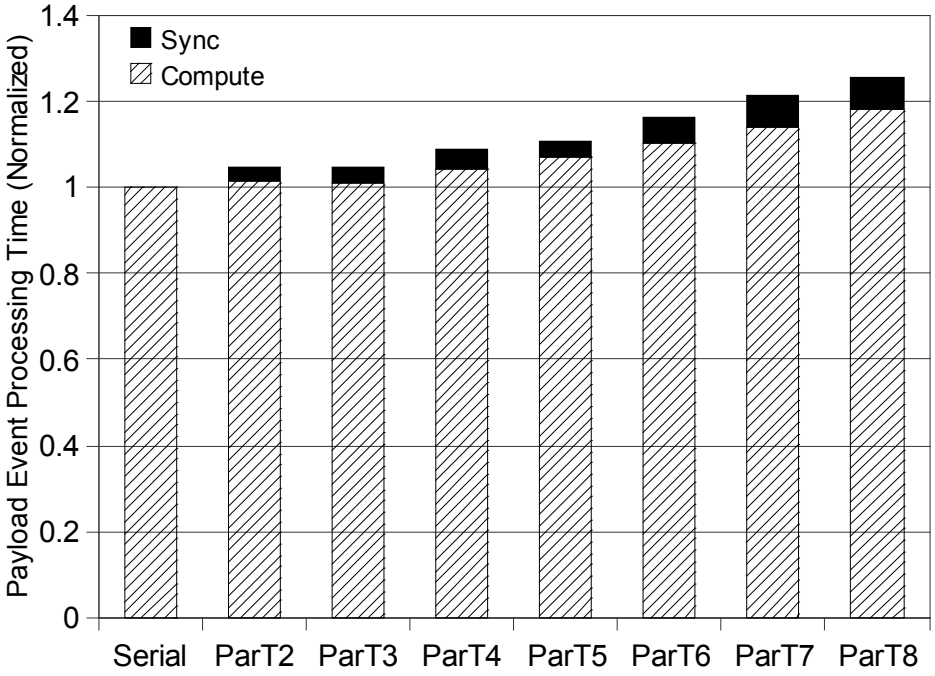


Figure 3·13: Event processing time as a function of number of threads (Code 3, size 128K)

data.

We observe also that bus utilization increases roughly linearly with the number of threads with exceptions between 1 and 2 and between 7 and 8. The first exception is because of the shift between serial and parallel code with the latter having helper and worker threads. The second exception is most likely due to system effects as no CPU cores remain free. We note that the increase in bus utilization increases memory latency through queuing and other delays. We have tested the memory hierarchy and found that, for random accesses (non-DMA), the performance saturates at 60% utilization rate of the bus.

These results match those of Section 3.4.2: that threads are almost always working on events in different neighborhoods and thus different data sets. And since every prediction requires accessing data of a new particle, there is little reuse or sharing of

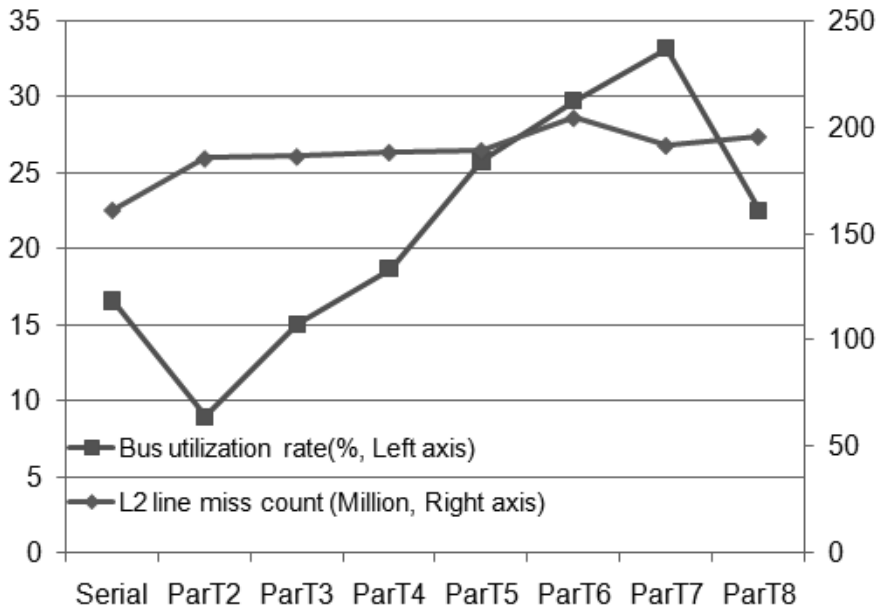


Figure 3·14: Rate of memory bus utilization and total number of L2 cache misses as a function of number of threads (Code 3, size 128K)

cached data.

To confirm our analysis on architectural limitation we use the following procedure. First, we project PDMD performance based on the fact that the addition of each thread increases individual event processing time. Second, we validate the projection against the observed scaling result. Third, we present a hypothesis that the majority of the increase is due to architectural limitation. And finally, we validate the hypothesis by using a microbenchmark that has data access pattern similar to those of the application.

The addition of each thread increases individual event processing time. Therefore, speed-up is limited by that increase rate. For example, if using 7 processors causes 30% increase in processing time, then speed-up = (Original Processing Time)/(New Processing Time/7) = 100/(130/7) = 7/(1 + 0.3) = 5.38. For PDMD, the addition of each thread causes roughly a 5% increase in event

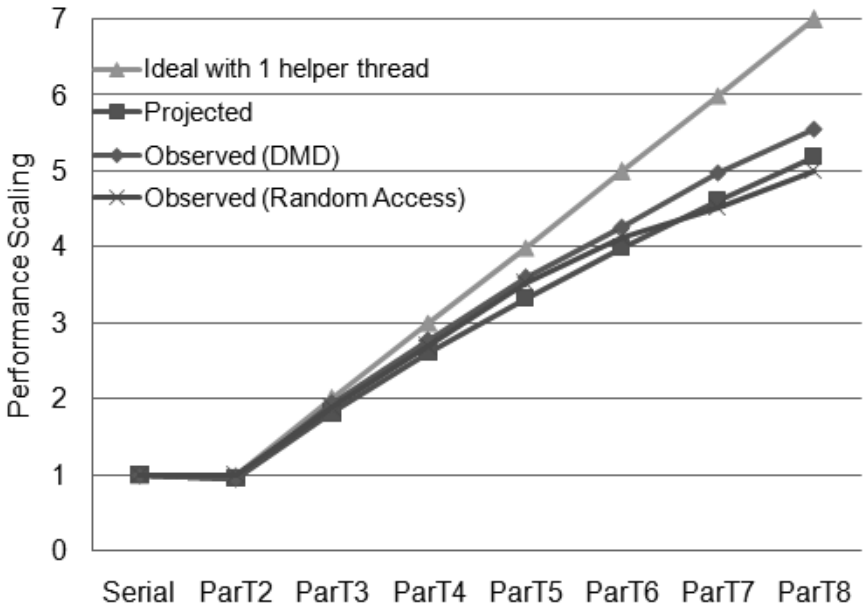


Figure 3·15: Overlap of scaling result for an analytical model with PDMD scaling result

processing time. Of this, about 2% is due to synchronization timing mismatch, confirmed from measurement in Figure 3·13, and the rest is due to the increased time in memory access. The latter portion is a hypothesis based on the increase in cache misses and bus utilization.

Based on the above discussion, the equation of projected speed up for implementation 3 is,

Speed-up = # of worker threads / (1 + increase rate × # of worker threads).

For example, for 8 processors (7 worker threads), the projected speed-up = $7/(1 + 0.05 \times 7) = 7 / 1.35 = 5.18$; the observed speed-up is 5.36. For 7 processors (6 worker threads), projected speed-up = $6/(1 + 0.05 \times 6) = 6 / 1.30 = 4.61$ and observed speed-up is 4.82. Thus the projection conforms well with the observed results, as shown in Figure 3·15.

To confirm our analysis on architectural limitation, we designed a microbenchmark

that has a data access pattern similar to our DMD application: Data of similar size is accessed in nearly random fashion. Figure 3·15 shows how the scalability result of that program overlaps with our DMD application, confirming the fact that the increase in data access time is a major obstacle to good scaling of PDMD. Therefore with a 5% increase in event processing time per thread, the projected runtime, with more than two threads T = (serial runtime × (1 + 0.05 × (# of threads -1))) / (# of threads - 1).

We validate our hypothesis that a 3% increase in event processing time per additional thread is caused by architectural limitation, i.e., the increase in cache miss count and bus utilization. We created a program that has a single loop and accesses different portions of a large data set randomly in each iteration. We kept the data size the same as our DMD application. If our hypothesis was correct, the scaling pattern of this program would resemble the scaling pattern of DMD. In fact, as shown in Figure 3·15, the scaling result of the random access program almost perfectly overlaps with the DMD result. DMD scales slightly better, since its data access is not entirely random. We conclude that we have accounted for the architectural limitations that limit the scalability of PDMD with event-based decomposition.

3.5 Chapter Summary

In this chapter we systematically defined the issues in parallelizing DMD, and presented our method of parallelization, where events are processed in parallel but committed in serial. We provided three possible implementations, including synchronization and other optimization techniques. Our task-decomposed method is micro-architecture inspired and motivated by the availability of fast shared memory in modern CPUs and by a recent advancement in DMD data structure (PaulQ),

which allowed us to achieve very low serialization overhead. Our final implementation achieved over 5.5x (8.5x) speed-up on an 8 (12) core CPU, with potential for further strong scaling. Our analysis shows that lack of concurrency in the application and contention for shared resources in the hardware are the major factors that prevent perfect scaling.

Chapter 4

FPGA Kernel for Acceleration of MD

The kernel we use in this study was introduced in Section 2.2.3. Here we provide a system-level overview, followed by a board-level description of the design. Then we discuss some of the key features of the design that will be important in a full-parallel integration. Finally, we describe how we improved the performance of the kernel by exploring the design space of the table interpolation method and utilizing the Block RAM (BRAM) architecture of the FPGAs. We begin with a description of FPGAs in general, followed by a description of the specific target platform.

4.1 FPGA Architecture

FPGAs have traditionally been known for their programmability and energy efficiency. In recent years, they have been equipped with dedicated multipliers and individually accessible BRAMs of different granularity [5, 177]. EDA tools support efficient floating point compilation [98]. All these have made FPGAs candidates for accelerating scientific applications such as MD.

Configurable Logic: Configurable logic blocks of FPGAs have evolved from simple 4-input Look-up table (LUT) and register combination to combinations of multiple 4-6 input LUTs, registers, adders and shift-registers. They provide bit-level control, fine-grained parallelism, deep pipelining, and data broadcast facilities. An example of such a module is shown in Figure 4·1.

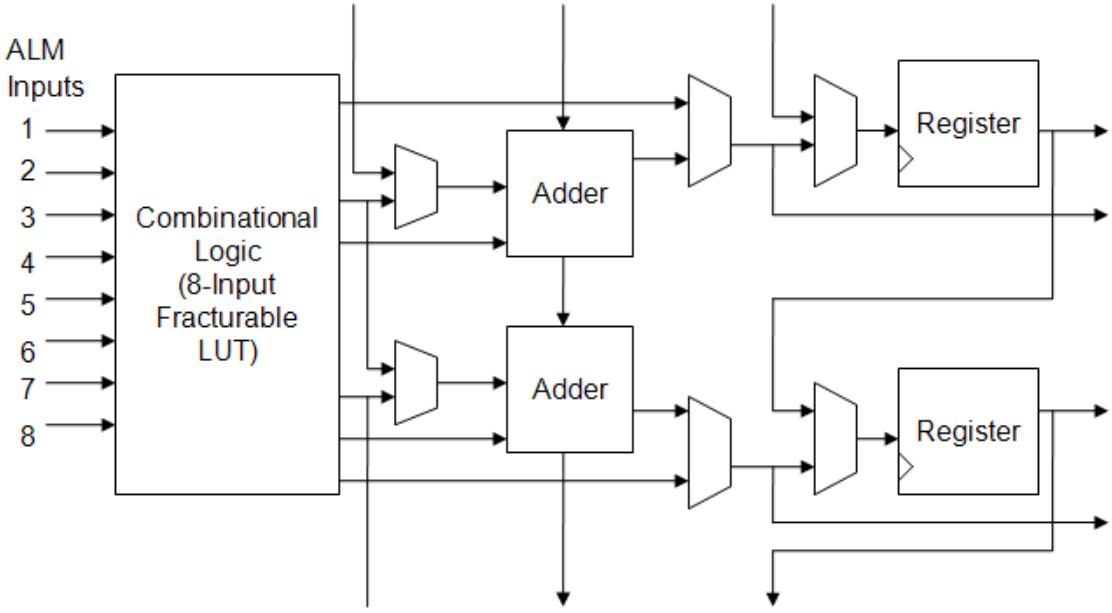


Figure 4·1: Block diagram of Adaptive Logic Module (ALM) of an Altera FPGA [4]

Configurable Memory Interface: Unlike CPUs or GPUs, FPGAs provide on-chip memory, called Block RAM (BRAM), that can be configured for various different granularity. A typical high-end FPGA has thousands of 32-bit BRAM ports for an aggregate bandwidth in the terabytes per second. FPGA boards also come with on-board DRAMs with 16 GB - 64 GB being common. Together, they give designers tremendous control over data storage and transfer, which is a key issue in many applications.

DSP Blocks and Vendor IPs: High-end FPGAs come with dedicated multiplier blocks and other vendor IP, e.g., shift-registers and FFT blocks. These make development of scientific applications easier and more efficient.

Floaring-point Compiler: FPGAs have typically been stronger in fixed point arithmetic. But most scientific computations require floating point operation.

Altera provides a floating point compiler that generates efficient FPGA code for floating point operation from C code [98].

Data Transceiver: High-end FPGAs are equipped with very fast transceivers for data communication [5, 177]. These are essential in the FPGAs' core market: communication switching. This resource is only beginning to be tapped for scientific computing and promises to distinguish FPGA-based systems from their alternatives.

4.2 Target Platform and Simulation Benchmark

The target platform for the FPGA kernel of this work is a Gidel PROCStar III board [59, 60, 61]. It is a PCI-based system with 8-lane PCI Express (PCIe x 8) host interface. The block diagram of the system is shown in Figure 2·13.

The system consists of four FPGA units, Altera Stratix III SE260 FPGAs, and is capable of running at system speeds of up to 300 MHz. The FPGAs communicate with a host via a PCIe bus interface. Each processing unit contains the following components:

- Altera Stratix III SE260 FPGAs
- 256 MB on-board DDR II SDRAM (bank A)
- 2 x 2 GB DDR II memory (bank B and bank C) via SODIMM sockets
- 2 PROCStar III Daughterboard connections

The system is able to deliver high-performance FPGA solution with massive capability and throughput memory. Its memory performance is summarized in Table 4.1.

Table 4.1: Gidel PROCStarIII memory performance [59]

	Bank A (On-board)	Bank B (SODIMM)	Bank C (SODIMM)
Capability	256 MB x 4	2 GB x 4	2 GB x 4
Performance (DDR)	667 MHz	667 MHz	360 MHz
Throughput	16 GB/s	16 GB/s	8.5 GB/s

Throughout this dissertation, we use the ApoA1 benchmark of NAMD [130] to evaluate our work on FPGA-accelerated MD. This benchmark has 92,224 particles of 47 types, and uses PBC with an original simulation box of $108\mathring{A} \times 108\mathring{A} \times 78\mathring{A}$. It uses a cut-off radius of $12\mathring{A}$ for range-limited force computation, and a switching function is applied to smooth the force when the inter-particle distance is between $10\mathring{A}$ and $12\mathring{A}$. The Coulomb force is evaluated using the PME method. We compute the long-range portion of PME every tiemstep.

4.3 Description of the Kernel

4.3.1 System-level Control Flow

The kernel is implemented on a Stratix-III FPGA on the Gidel ProcStar-III which is integrated into a host PC through a high-speed PCIe bus. Figure 4·2 shows the control flow of the FPGA-accelerated MD, where range-limited non-bonded force computation is off-loaded to the FPGA kernel. The left hand side shows the steps executed on the software and the right hand side shows the steps executed on the FPGA.

Before invoking the FPGA kernel, the software first initializes it by clearing on-board memory and on-chip caches; and by setting up necessary simulation parameters, e.g. numeric precision, number of cells etc. Particle data are prepared and sent to the on-board memory banks via PCIe bus using DMA. Such preparation includes any

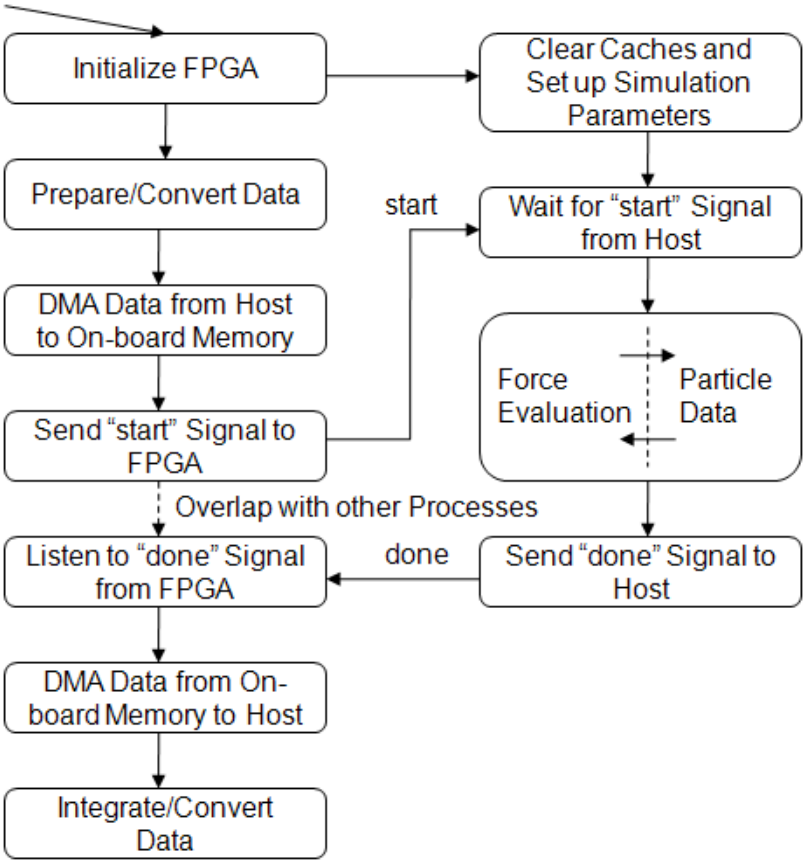


Figure 4·2: Control flow of the FPGA-accelerated MD [37]

conversion of particle position data from double-precision to single-precision floating point, as well as packing them together according to their cell-list ID. Cell-list data also needs to be created if it is not already available. Cell-list data is sent in a simple form of a single-item list where each item corresponds to the number of particles in that cell. In the current implementation of the design, particle type and cell-list data is stored in Bank C, where as position and charge data are saved in Bank B. Results of the force computation are stored in Bank A. This allows seamless access to on-board data during force computation, as well as independent DMA between the host and the FPGA board.

After DMA operations are completed, the host issues a "start" signal to the

FPGA to indicate that data are all updated now and it is okay to begin force computation. Once "start" signal is received by the FPGA, the controller on the user design initializes the force pipelines, loads data from the off-chip memory to on-chip caches and begins force computation.

After forces of all particles are computed, a "done" signal is sent back to the host, which then reads the force data using DMA. These forces are then, after possibly being converting to double-precision floating point, added to other forces, e.g. bonded terms, that were computed on the host. Evaluation of other forces on the host can be overlapped with the computation on the FPGA kernel.

4.3.2 Board-level Integration

The system-level diagram of the FPGA-accelerated MD design is shown in Figure 4·3. The host runs the MD software and communicates with the accelerator board through PCIe bus. The accelerator board consists of a high-end FPGA, memory blocks, and a bus interface. Besides configurable logic, the FPGA has dedicated components such as independently accessible multiport memories (e.g., 1000 x 1KB) and a similar number of multipliers.

FPGA itself is divided into two main components, user design and vendor logic. Vendor logic is dedicated to system (non-application) functions such as memory controllers and occupies about 10%-15% of the FPGA's logic in the implementation. User design contains the computational engine of the MD accelerator, including control logic, filter banks, and force pipelines. In additional to the central computational logic, it also includes BRAMs to store particle data and forces, and simulation parameters.

On the accelerator board, three memory blocks, including one 256MB on-board memory (bank A) and two 2GB DDR2 SODIMMs (bank B and C), are used for data storage (coordinates, charges, types and forces). Coordinates and charges are

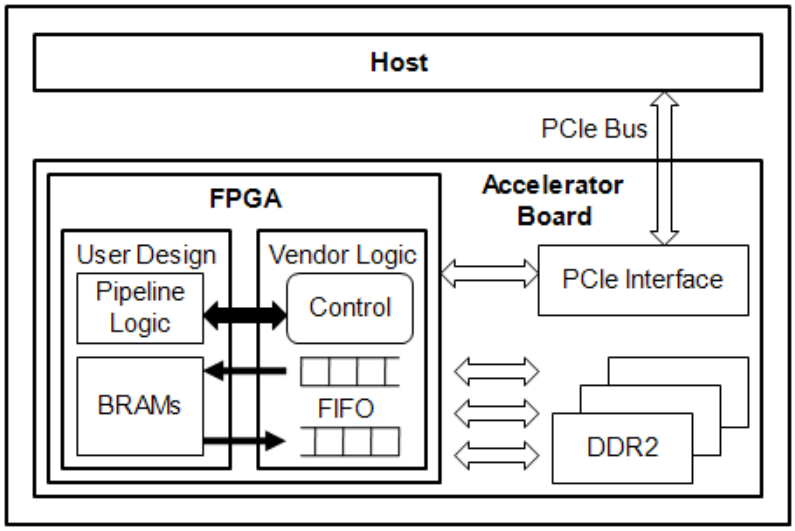


Figure 4·3: System architecture of the FPGA-accelerated MD design [37]

grouped together and stored in bank B and particle types are stored in bank C. Bank A is used to collect computed forces for each particle. Coordinates, charges and forces are 32-bit single precision floating point numbers. Particle types are represented in reduced precision integers and the required precision depends on how many particles are supported in simulations. One limitation of bank C is that it only runs at half frequency (167 MHz) of bank A and B (333 MHz). Therefore, compact-sized data (e.g., particle types) are stored in Bank C, to prevent their access from becoming a critical path. PCIe interface is responsible of the management of communication between the host and accelerator.

4.3.3 Cell-list and Filtering

Cell-list and Neighbor-list methods were described in Section 2.1.6 as ways to reduce the complexity of the range-limited non-bonded force computation. Cell-list method is simple but only about 15% of the checked particle-pairs fall within the cut-off distance. This inefficiency is improved in neighbor-list method at the cost of additional memory. While neighbor-list method is proved to be efficient on CPUs, the requirement of

additional memory makes it prohibitive for FPGA implementation. If neighbor-lists are computed on the host, it will require a large amount of data communication between the host and the FPGA. Even if they are computed on the FPGA, they will require too much storage. That is why, in this work cell-list method is chosen, but low-cost filtering is used to improve its efficiency. A number of low-precision filters are used in parallel to filter out the particle-pairs that are certainly not within the cut-off distance. Only the remaining particle-pairs are sent to the force pipeline for final evaluation and force computation. This method can also be seen as a neighbor-list method where instead of saving and re-using the lists, they are computed on-the-fly every timestep.

While this method is efficient on the FPGA, it requires creation of cell-lists on the host every timestep.

4.3.4 Half-moon Mapping Scheme

The efficiency of the FPGA kernel is directly related to the efficiency of the force computation pipelines. In the kernel we use, multiple pipelines work in parallel to compute forces. At one time, each pipeline is responsible for computing forces of a certain particle, called the reference particle, in the current reference cell, called the home cell. Since the registers for all the reference particles are set in batch, and their results are also written back to memory in batch, it is important to make sure that each of these particles have similar number of interacting particles. Otherwise, some of the force pipelines will waste time waiting for others to finish. This preference becomes critical when there are a large number of force pipelines and a much larger number of filter pipelines.

Standard methods of mapping scheme give rise to a load-balance issue here. In order to use Newton's third law, so that a force between a particle-pair is only computed once, only a "half-shell" of the surrounding cells is examined. Figure 4·4a

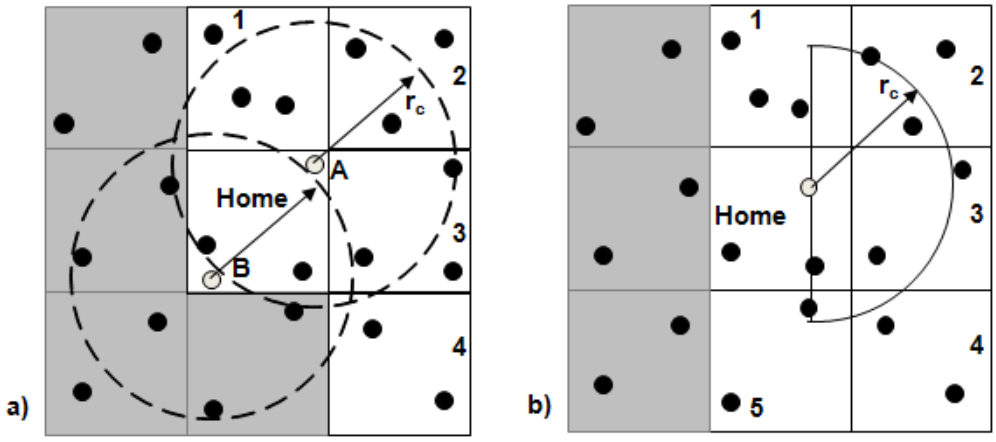


Figure 4·4: 2D illustration of two partitioning schemes that use Newton's 3rd Law. a) 1-4 plus home are examined with a full sphere b) Half-moon scheme where 1-5 plus home are examined, but with a hemisphere [33]

shows a 2D example, where cells 1-4 and home cell is examined. As illustrated, particle B has a much smaller number of particles to interact than does particle A.

This imbalance of load is addressed in the design by introducing a novel mapping scheme, called "half-moon" scheme, where only particles at the right side of a reference particles are considered for force computation. Although this increases the number of cells to check, it ensures a better load-balance for the force pipelines. Figure 4·4b shows an example in 2D, where cells 1-5 and home cell are examined in the new scheme.

4.3.5 Particle Exclusion

While combining various forces before computing acceleration is a straightforward process of linear summation, careful consideration is required for bonded pairs when using hardware accelerators. As described in Section 2.1.4, covalently bonded pairs need to be excluded from non-bonded force computation. One way to ensure this is to check whether two particles are bonded before evaluating their non-bonded forces. This is expensive because it requires on-the-fly check for bonds. Another

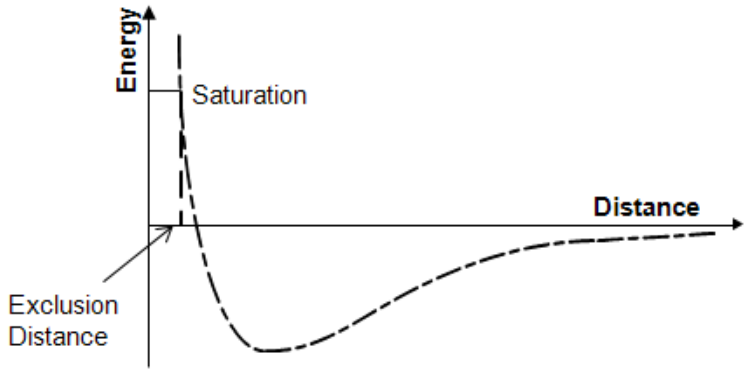


Figure 4·5: Graph shows van der Waals interaction with cut-off check with saturation force

way is to use separate neighbor-lists for bonded and non-bonded neighbors. Both of these methods are problematic for hardware acceleration: one requires implementing a branch instruction while the other forces the use of neighbor-lists, which is impractical for hardware implementation.

A way that is often preferred for accelerators is to compute non-bonded forces for all particle-pairs within the cut-off distance, but later subtract those for bonded pairs in a separate stage. This method avoids both on-the-fly bond checking and neighbor-lists. There is a different problem here though. The r^{14} term of the LJ force (Equation 2.2) can be very large for bonded particles because they tend to be much closer than non-bonded pairs. Adding and subtracting such large scale values can overwhelm real but small force values. Therefore, care needs to be taken so that the actual force values are not saturated.

This kernel applies a short cut-off to the non-bonded force calculations based on the fact that two particles that are not bonded generally cannot be too close to each other. Therefore, two particles within a certain short distance are likely to be bonded. The short cut-off distance can be calculated by solving the inequality $F_{short} < range$, where $range$ is the dynamic range with a reasonable force value. The left term of

the inequality is dominated by the 14 term of Equation 2.2. Multiple short cut-off values are required as this depends on the particle type. A simple graph is shown in Figure 4·5 to demonstrate this concept.

If the exclusion cut-off is chosen conservatively, two particles will be bonded as long as their intra-distance is smaller than the exclusion distance. For bonded particle-pairs whose intra-distance is larger than the exclusion cut-off, the non-bonded force is subtracted in the host. Since the exclusion distance check in FPGA is performed in integer arithmetic while it is done in double-precision floating point in the host, an inconsistency may occur when the distance between two particles is very close to the exclusion cut-off. In order to minimize the impact of this inconsistency, a saturation force is applied if the intra-distance between two particles is smaller than the exclusion cut-off, as shown by the saturation line in Figure 4·5.

4.4 Improving Performance using Block RAM (BRAM) Architecture

4.4.1 Exploring Design Space of Table Interpolation

Table interpolation method was introduced in Section 2.1.7 as an alternative to direct computation of forces. This method is often used to compute the pairwise range-limited non-bonded forces, as shown in Equation 4.1.

$$\frac{\mathbf{F}_{ji}^{short}}{\mathbf{r_{ji}}} = A_{ab} r_{ji}^{-14} + B_{ab} r_{ji}^{-8} + QQ_{ab}(r_{ji}^{-3} + \frac{g'_a(r)}{r}) \tag{4.1}$$

Here, the first two terms compute the van der Waals force and the third the Coulomb force. A_{ab}, B_{ab}, and QQ_{ab} are distance (r) independent coefficients that depend on atom types of a and b, and the g term is a correction for integration with the long-range force. While computing these terms directly is certainly a viable option, it often becomes computationally expensive. For example, evaluating the Coulomb

term requires computing r, which in turn requires square-root computation of r^2. It also requires evaluating expensive functions like $erfc$. Computing otherwise simple van der Waals force can be complicated too, if switching function is used.

To avoid these direct computations, Equation 4.1 is rewritten as

$$\frac{\mathbf{F}_{ji}^{short}(|r_{ji}|^2(a,b))}{\mathbf{r_{ji}}} = A_{ab}R_{14}(|r_{ji}|^2) + B_{ab}R_8(|r_{ji}|^2) + QQ_{ab}R_3(|r_{ji}|^2) \tag{4.2}$$

where R_{14}, R_8, and R_3 are looked-up from tables indexed with $|r_{ji}|^2$ (rather than $|r_{ji}|$ to avoid the square-root operation).

As mentioned in Section 2.1.7, accuracy of computation increases both with the number of bins (intervals) per segment and with the order of interpolation. While increasing bins per segment requires more memory for storing parameters, increasing order of interpolation requires more logic for implementation of actual computation. Thus it may be possible to trade one with another, storage vs. logic, as long as the quality of computation is acceptable.

We first provide a survey of table interpolation methods used in some of the widely used MD software packages.

- NAMD (CPU) – Ref: [130] and Source code of NAMD2.7

  Order = 2 bins/segment = 64 Index: r^2

  Segments: 12 – segment size increases exponentially, starting from 0.0625

- NAMD (GPU) – Ref: [162] and Source code of NAMD2.7

  Order = 0 bins/segment = 64 Index: $1/\sqrt{r^2}$

  Segments: 12 – segment size increases exponentially, starting from 0.0625

- CHARMM – Ref: [24]

  Order = 2 bins/segment = 10-25 Index: r^2

  Segments: Uniform segment size of $1\mathring{A}^2$ is used which results in relatively more

precise values near cut-off

- ANTON – Ref: [99]

  Force Table Order = Says 3 but that may be for energy only. Value for force may be smaller.

  # of bins = 256 Index: r^2

  Segments: Segments are of different widths, but values not available, nor whether the number of bins is the total or per segment.

- GROMACS – Ref: [79] and GROMACS Manual 4.5.3, page 148

  Order = 2 bins = 500 (2000) per nm for single (double) precision

  Segments: 1 Index: r^2

  Comment: Allows user-defined tables.

Clearly there are a wide variety of parameter settings. These have been chosen with regard to cache size (CPU), routing and chip area (Anton), and the availability of special features (GPU texture memory). These parameters also have an effect on simulation quality, which will be discussed later.

What we focus on here is the fact that there is no need to blindly follow the parameter settings of the reference software code. Rather, we can choose parameters that suit the resources available on the accelerator. In particular, FPGAs are equipped with BRAMs, which may let us to use more bins per segment allowing lower order of interpolation. This will save logic resources, which can then be used to implement more force computation pipelines. Depending on the availability of such complementary resources, this approach may improve overall performance.

4.4.2 Studying Simulation Quality

Since MD is chaotic, simulation quality must be validated. This is especially true when we chose parameters that are different from those in the reference software.

Here, we study the quality of simulation for different parameter settings of the table interpolation method. Quality measures can be classified as follows (see, e.g., [51, 122, 153]).

- Arithmetic error here is the deviation from the ideal (direct) computation done at high precision (e.g. double-precision floating point). A frequently used measure is the relative RMS force error, which is defined as follows [152]:

$$\Delta F = \sqrt{\left(\frac{\sum_i \sum_{\alpha \in x,y,z} [F_{i,\alpha} - F^*_{i,\alpha}]^2}{\sum_i \sum_{\alpha \in x,y,z} [F^*_{i,\alpha}]^2} \right)} \tag{4.3}$$

- Physical invariants should remain so in simulation. Energy can be monitored through fluctuation (e.g., in the relative RMS value) and drift. This work used following expression (suggested by Shan et al. [152]):

$$\Delta E = \frac{1}{N_t} \sum_{i=1}^{N_t} \left| \frac{E_0 - E_i}{E_0} \right| \tag{4.4}$$

  where E_0 is the initial value, N_i is the total number of time steps in time t, and E_i is the total energy at step i. Acceptable numerical accuracy is achieved when $\Delta E \leq 0.003$.

We use these metrics to validate the quality of FPGA-accelerated simulation. Results are presented for the ApoA1 benchmark described in Section 4.2. NAMD-Lite, the reference software, was modified to support the quality measurements [73].

To determine the force error, NAMD-lite was modified to support the various functions. The simulation was first run for 1000 timesteps using direct computation. Then in the next timestep, both direct computation and table interpolation were used to find the relative RMS force error for table interpolation. Only the range-limited non-bonded forces (switched VdW and range-limited portion of PME) were considered. All computations were done in double-precision floating point and

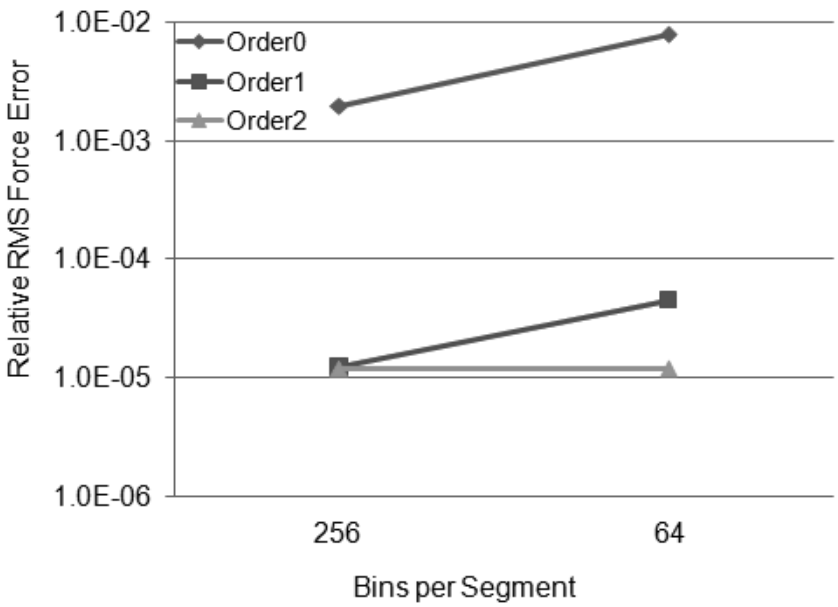


Figure 4·6: Relative RMS Force Error versus bin density for interpolation orders 0, 1, and 2

Equation 4.3 was used to compute the relative RMS. Results are shown in Figure 4·6. We note that 1st and 2nd order interpolation have two orders of magnitude less error than 0th order. We also note that with 256 bins per segment (and 12 segments) 1st and 2nd order are virtually identical.

Preliminary results with respect to energy fluctuation and drift are shown in Figure 4·7. A number of design alternatives were examined, including the original code and all combinations of the following parameters: bin density (64 and 256 per segment), interpolation order (0th, 1st, and 2nd), and single and double-precision floating point. We note that all of the 0th order simulations are unacceptable, but that the others are all indistinguishable (in both energy fluctuation and drift) from the serial reference code running direct computation in double-precision floating point.

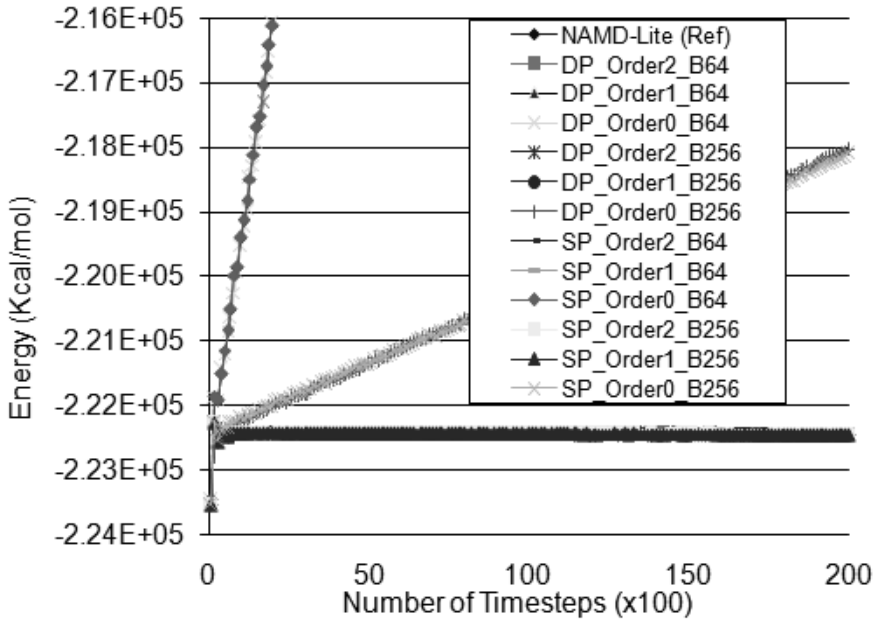


Figure 4·7: Energy for 20,000 timesteps for various designs. Except for 0-order, plots are indistinguishable from the reference code

Energy plot for longer runs of 100,000 timesteps is provided in Figure 4·8, which shows good energy conservation in the FPGA-accelerated versions. Only a small divergence of 0.02% was observed compared to the software-only version. ΔE value, using Equation 4.4, for all accelerated versions were found to be much smaller than 0.003. For example, after 70,000 timesteps, the value of ΔE were all less than 1.5E-07.

4.5 Results

Performance is directly related to resources consumed, as shown in Table 4.2. All of these designs have been implemented and run on the Gidel board. Time shown here is the runtime of the kernel in a timestep. We note that the number of pipelines increases from 4 of direct computation (DC) to 5 to 6 to 7 with interpolation order 2, 1, and 0, respectively. According to the quality results, the six pipeline design with 1st

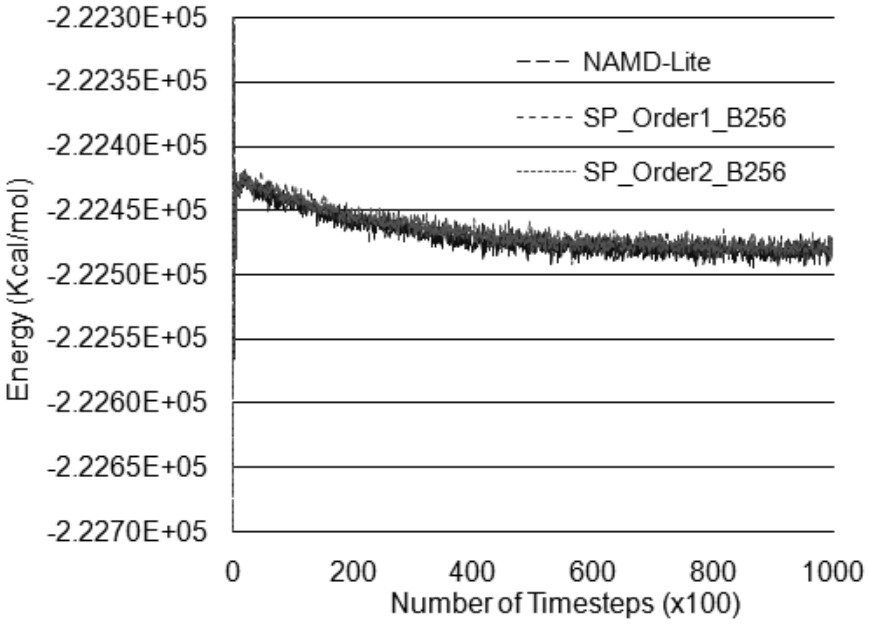


Figure 4·8: Energy for 100,000 timesteps for selected designs

order interpolation is likely to be preferred. This design takes advantage of the BRAM architecture and increases performance by almost 50% over direct computation. The resource utilization results indicate that the limiting factor is the logic resources, mostly registers. An interesting observation is that number of bins is not a major concern and could be doubled if needed to achieve better simulation quality.

Thus, by using 1st order interpolation, rather than 2nd order interpolation which is the choice of the reference software, we maintain the simulation quality while achieving better performance. This is largely a consequence of the number of interpolation intervals (bins) that we were able to use: over 3,000 for each of the three terms in Equation 4.2. Our ability to do this is a direct result of the availability of BRAMs in high-end FPGAs. By comparison, the 72 KB storage needed for these tables would swamp the L1 data cache of a modern CPU core and would likely reduce performance substantially.

We have also synthesized the designs with respect to the Stratix IV EP4SE530

Table 4.2: Resource utilization and performance of various pipeline configurations on the Stratix III EP3SE260 (bins/segment = 256)

	LUP Order 0	LUP Order 1	LUP Order 2	DC
Multipliers	67%	63%	66%	68%
Logic	87%	88%	85%	94%
BRAM (M9K)	89%	86%	89%	62%
BRAM (M144K)	87.5%	75%	62.5%	50%
Number of Pipeline	7	6	5	4
Timing (ms) @ 200 MHz	NA	45	56	67

Table 4.3: Resource utilization and performance of various pipeline configurations on the Stratix IV EP4SE530 (bins/segment = 256)

	LUP Order 0	LUP Order 1	LUP Order 2	DC
Multipliers	76%	87%	98%	100%
Logic	69%	75%	78%	86%
BRAM (M9K)	98%	98%	95%	67%
BRAM (M144K)	100%	100%	94%	75%
Number of Pipeline	12	11	10	8

(post place-and-route) with the results shown in Table 4.3. After optimization we anticipate achieving an operating frequency similar to that for the Stratix III. This indicates a nearly proportional increase in performance.

4.6 Chapter Summary

In this chapter we provided a description of the FPGA kernel that we use in this design, and how we enhanced its performance by utilizing BRAM architecture of the FPGAs. The availability of BRAMs allowed us to use finer-grained table for table interpolation method without sacrificing simulation quality, saving logic resources, which were then used to implement more force computation pipelines. The net result was a 50% improvement in the performance of the kernel.

Chapter 5

Intra-node Parallelization of FPGA-accelerated MD

The FPGA kernel enhanced and used in this work was originally integrated into NAMD-Lite [73], a serial MD package developed at UIUC to provide a simpler way to test new features before integrating them into NAMD, a widely used full-parallel production MD package [130]. In this work, we integrate the kernel into NAMD (version 2.8), the production package; and study various scaling issues. Although NAMD is a full-parallel software, in this chapter we restrict the discussion to its serial run and rather focus on intra-node parallelization of the kernel itself, including relevant integration issues. Full-parallel integration will be described in the next chapter. The work described in this chapter is motivated by the fact that most recent FPGA boards come with multiple high-end FPGAs. Intra-node parallelization of the kernel, that is, being able to use all on-board FPGAs in parallel, thus, becomes absolutely crucial in realizing the full potential of such FPGA-based systems.

5.1 Challenges and Opportunities

As described in Chapter 4, data need to be sent back and forth between the host processor and the FPGA every timestep in the FPGA-acclerated MD. Several conversions need to take place before and after each data communication. In order to take advantage of the availability of multiple FPGAs on a board, we now also need to partition the workload accordingly. We study these issues, without loss of

generality, using the ApoA1 benchmark which is described in Section 4.2. Our target software, NAMD [130], uses neighbor-list method for the range-limited non-bonded force computation. Parallelization is achieved by spatially decomposing the simulation box into patches, where a patch is typically a cubic box with each dimension much larger than the cut-off distance. The reason of choosing such large dimensions is to make sure that all interacting particle-pairs remain in neighboring patches for a substantial number of timesteps. Particles are allowed to be marginally outside the border of their patches, but migration is performed when it becomes necessary, e.g. when a particle moves too far outside the border of its patch. Particle data are also managed in a per-patch basis. That is, each patch, a C++ object in actual implementation, contains and owns data of all particles that belong to it; and is responsible for updating the data every timestep. It should be noted that, just like in cell-list method, smaller patch-dimensions can also be used; it will only require checking more neighboring patches to find interacting particle-pairs.

5.1.1 Data Conversion and Communication

We recall from the discussion in Chapter 4 that the FPGA kernel uses cell-list method in combination with Newton's third law and half-moon mapping scheme. This scheme assumes that each of the cell in the simulation box will become the reference cell at one time during the computation when its 18 neighboring cells (in a 3D system with cell-dimension slightly larger than the cut-off distance) will be checked for range-limited non-bonded force computation.

Since our reference software NAMD does not use cell-list method, cell-list data structure needs to be computed every timestep, before invoking the FPGA kernel. And since the FPGA returns the result of the computation grouped by cells as they appear in the cell-list, the result also needs to be transfered to patch data structure. Other data conversions, e.g. conversion from double-precision to single-precision

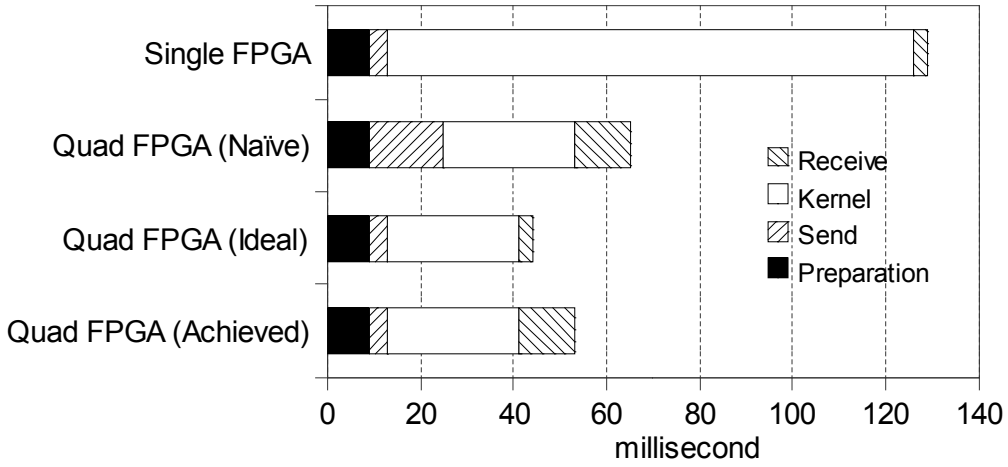


Figure 5·1: Profiling of kernel-related runtime of a timestep in the FPGA-accelerated MD

floating point etc., also need to done. We note that the conversion to cell-list can be avoided if we just use patches as cells. But this would require checking substantially large number of particle-pairs for potential interaction. The volume of a patch is roughly 3.375x larger than that of a cell (9x9x6 cells vs. 6x6x4 patches in ApoA1), resulting in a 60.75 (18×3.375) fold increase. Besides, fitting particle data of 18 patches in on-chip cache of our target platform is also challenging.

Although data communication between the host and the FPGA is an overhead, it also provides some opportunities for optimization when we have multiple FPGAs on a board. To understand this we first provide a profiling of the kernel related operations of a timestep in a single FPGA run.

Using a host and a single FPGA on the Gidel board (details of the hardware will be provided in Section 5.4), it takes about 9 milliseconds (ms) to prepare cell-list data, including data structure to map the results back to patch data. About 4 ms is spent in sending all data to the FPGA with multiple DMA transactions. The computation on kernel itself takes 113 ms to compute range-limited non-bonded forces of all particles.

Then 3 ms is spent bringing the data back to the host CPU and converting back to patch data. Figure 5·1 shows this runtime profiling and contrasts a naive and ideal parallelization scenario using 4 FPGAs. A naive implementation will simply spend 4x more time in sending data to and receiving data from the FPGA. Assuming a perfect scaling for the kernel computation time, this will result in only a little less than 2x speed-up. An ideal implementation will hide successive DMAs behind computation on CPU and FPGAs, and achieve close to 3x speed-up (we assume sending the same data to all FPGAs, as will be described in Section 5.3.2). A perfect speed-up (4x using 4 FPGAs) is not feasible due to Amdahl's law (because the other portions, especially the data conversion time, do not scale). Actual speed-up will depend on partitioning of the workload and other implementation issues.

5.1.2 Partitioning

Partitioning of the workload plays an important role in achieving speed-up using multiple on-board FPGAs. First, partitioning of the workload determines the size of the data that needs to be sent to and received from each of the FPGAs. For example, we can send the same data to all FPGAs and set parameters separately such that they compute different portions of the workload. Or we can send to each FPGA only the data that are necessary for computing its designated workload. Partitioning scheme also determines the runtime of the kernel of each individual FPGA. We can assign larger workload to the FPGAs that are invoked first and smaller workload to those that are invoked later. Partitioning also affects how we integrate the results from multiple FPGAs on the host. For example, we can partition the workload such that each FPGA will compute the total range-limited non-bonded force acting on the particles that it is responsible for. This will mean, we can use these values directly in the software. On the other hand, we can have a strategy where each FPGA will be responsible for partial computation of certain cells only (e.g. interaction with only 18

neighboring cells). We will have to combine forces from all FPGAs to get the total force in this case.

5.2 Data Communication with Software Pipelining

5.2.1 Data Conversion

Creating cell-list data structure and copying all required data to the DMA buffer is done by traversing through all particles in the patches twice. In the first traversal we compute the number of particles in each cell and create data structure necessary to map particles from cell data structure back to patch data structure. In the second traversal, particle data are copied in the DMA buffer for transfer. The first traversal is necessary because particle data need to be prepared for DMA in a contiguous memory buffer, grouped by cell ID. That is, all particles in cell 0 need to appear first, followed by all particles in cell 1 and so on. This is a requirement from the FPGA kernel. Therefore, it becomes essential to know the number of particles in the cell before actually copying data to the buffer.

Another way of of accomplishing this will be to create separate arrays of particle data for each cell first (as shown in Figure 5·2) and then copying them again to the DMA buffer in order. This is not implemented because of the significant increase in memory requirement. Separating the creation of cell-list data structure also allows us to keep this process simple, which will be helpful when we use software pipelining (see Section 5.2.2).

Particle data are mapped from cell-list data structure back to patch data structure using a patch ID and a particle ID that is local to the owner patch. This is illustrated in Figure 5·2. While creating cell-list data from the coordinates of the particles, only these two IDs are saved. Particle data are shown in Figure 5·2 only to illustrate the concept, they are not actually saved. They are rather copied to the DMA buffer

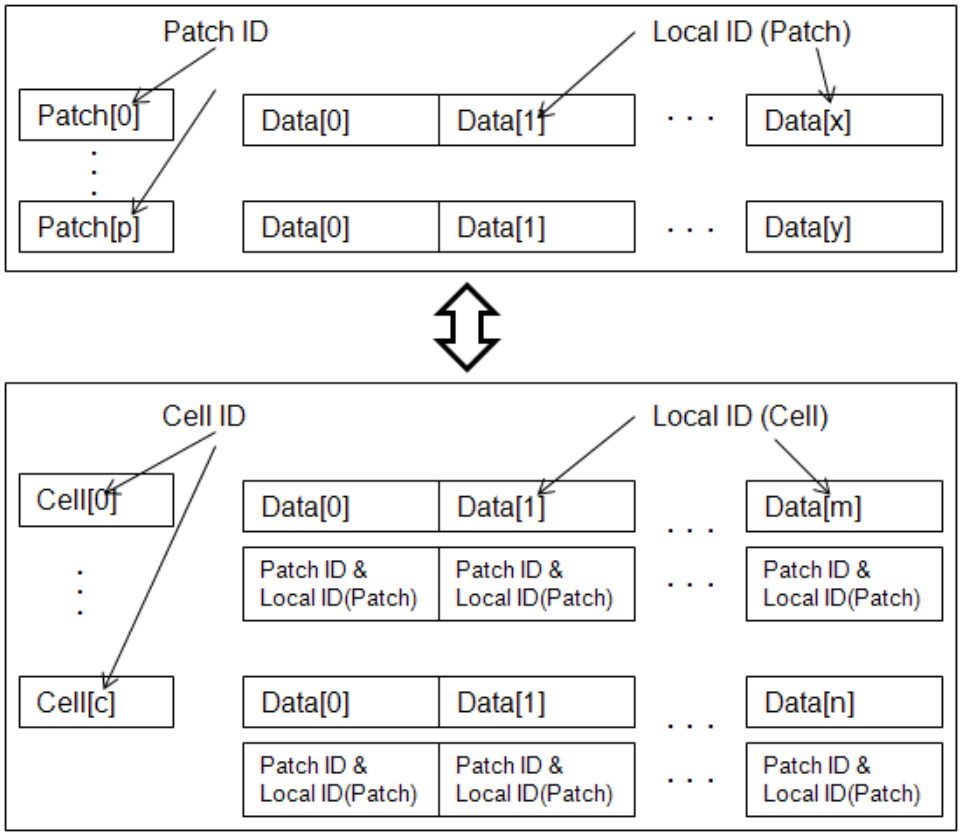


Figure 5·2: Mapping particle data from cell-list data structure back to patch data structure using two IDs

directly from patch data using these two IDs. The computation results from the FPGA are also copied from DMA buffer to patch data structure using these IDs. These IDs allow a mapping between cell-list data structure and patch data structure in $O(1)$ time.

While copying data to the DMA buffer, coordinate and charge data are converted from double precision to single precision floating point. In addition to coordinate and charge data, cell-list data structure (number of particles in each cell) and particle type data are also sent to the FPGA using DMA. In total, there are three DMA transfers before invoking the FPGA kernel and one DMA transfer after the kernel finishes. The last DMA is for results, which are received from the FPGA as single-precision

floating point, and then converted to double-precision floating point while they are copied to patch data structure.

5.2.2 Software Pipelining

As shown in Figure 5·1, ideal intra-node scaling would be achieved if there is no additional overhead in transferring data to multiple FPGAs, as opposed to sending to only one FPGA; and the computation on the kernel itself achieves a perfect scaling. It follows that the time for DMA transfers to second and successive FPGAs needs to be be hidden behind the computation time of the FPGA kernel or other tasks on the host. In fact, ideally, all communication should be overlapped with computation. Here, we describe the software pipelining that we use in this work to improve intra-node scaling.

Figure 5·3 shows the sequence in which the on-board FPGAs are invoked by the host in every timestep. As shown there, cell-list data structure needs to be computed first. For the first FPGA, right after setting necessary parameters, we kick off DMA1, where cell-list data is transferred to Bank C of the FPGA. While this DMA is running, we prepare particle data for this FPGA. This includes converting double-precision floating point to single-precision floating point and copying data to DMA buffer. Once DMA1 finishes, we send coordinate and charge data as DMA2 and type data as DMA3. We need to wait for the completion of DMA1, because both cell-list (DMA1) and type (DMA3) are saved in Bank C.

For the successive FPGAs, we first wait for DMA2 and DMA3 of previous FPGA to finish. Now, this was necessary because of an implementation issue that restricted the design to use the same DMA channel for a certain Bank for all on-board FPGAs. So, DMA to Bank C of all FPGAs must finish before we can start DMA1 of any FPGA. Once the completion of DMA2 and DMA3 is confirmed, “start” signal is sent to that FPGA to begin computation. While computation goes on there, the current

```
Create Cell-list;
For (index = 0; index < FPGA_Count; index++){
        Set Parameters for FPGA[index];
        If (index > 0){
           Wait for DMA1 and DMA2 of FPGA[index-1];
           Send "start" Signal to FPGA[index-1];
        }
        Send Cell-list Data to FPGA[index];
        //Start DMA1 of FPGA[index]
                Copy Particle Data to DMA Buffer for FPGA[index]
        Wait for DMA1 of FPGA[index]
        Send Coordinate and Charge Data;
        //Start DMA2 of FPGA[index]
        Send Type Data;
        //Start DMA3 of FPGA[index] )

        If (index == FPGA_Count -1){
           Wait for DMA1 and DMA2 of FPGA[index];
           Send "start" Signal to FPGA[index];
        }
}
```

Figure 5·3: Software pipelining to overlap communication and computation during the kick-off of multiple FPGA kernels

FPGA begins its own DMA1 and follows the loop. For the last FPGA on the board, after kicking off its DMA2 and DMA3, software immediately waits for completion of the data transfer and then sends a "start" signal to it.

We now describe how software pipelining is used while reading the results back from FPGA to the host. We keep polling for the "done" signal from the first FPGA. As soon as we receive that, DMA is kicked-off to read the result. Once the DMA is complete, we begin copying the data (including conversion from single-precision floating point to double-precision floating point) to the patch data structure. While doing so, we periodically check for "done" signal from the next FPGA. If we receive

that, we kick-off the DMA to read results from that FPGA and then continue copying result for the current FPGA. Once copying is done, we wait for the completion of any DMA that may have started already. Otherwise we just keep polling for the "done" signal of the next FPGA. This loop continues until results from all of the FPGAs are read.

5.3 Intra-node Partitioning

As described in Section 4.3, our FPGA kernel uses half-moon mapping scheme. It assumes that each cell in the simulation space will become a reference cell at some point during the computation, when it will check 18 of its 27 neighboring cells for possible interactions, according to half-moon scheme. For any particle A in the reference cell, inter-particle range-limited non-bonded forces for all particles that are to the right hand side of A and inside cut-off distance, will be computed right away. But for the particles to the left of A, force will be computed when the cells containing those particles become reference cells. Thus, the kernel goes through all cells in the cell-list and treat them as reference cells. After all cells have become reference cell once, computation for that timestep is done.

5.3.1 Method 1

To use multiple FPGAs without modifying the kernel, we need to make sure that force computation is not duplicated in the result in any way. This can be done by assigning computations of certain cells (target cells) to each FPGAs. Since neighboring cells are also required for a complete evaluation of the range-limited non-bonded forces, we will also have to send the data for particles in the neighboring cells. Once all cells (target cells and neighboring cells) are treated as reference cells, the forces of the particles in the target cells will be evaluated completely and can be read back and used in the software. The results for the neighboring cells will simply be discarded.

The advantage of this method are listed below.

- No change required in the kernel.
- Results from one FPGA can be used in software without waiting for results from other FPGAs.
- The amount of data communication between the host and the FPGAs can be reduced.

The disadvantages are as follows.

- FPGAs will go through more reference cells than required in a single FPGA implementation.
- The need for data of the neighboring cells will limit the reduction of data communication.
- Additional time is required on the host to convert cell IDs accordingly (see the discussion below)

One problem arises in this method is the traversal logic in the kernel. The kernel simply starts at cell 0 and increments cell ID by 1 to find the next reference cell. This continues until all cells are treated as reference cell once. Cell ID is a simple function of the coordinates of a cell and and is shown in Equation 5.1.

$$Cell\_ID = ((zIndex \times yDim) + yIndex) \times xDim + xIndex \tag{5.1}$$

Here, $xDim, yDim, zDim$ are number of cells in x, y, z directions and $xIndex, yIndex, zIndex$ are the indexes of the cell in x, y, z dimensions ($zDim$ not used in the equation).

This means, original number of cells and indexes cannot be used in the partitioned version. Number of cells must be updated and indexes must be shifted or rotated

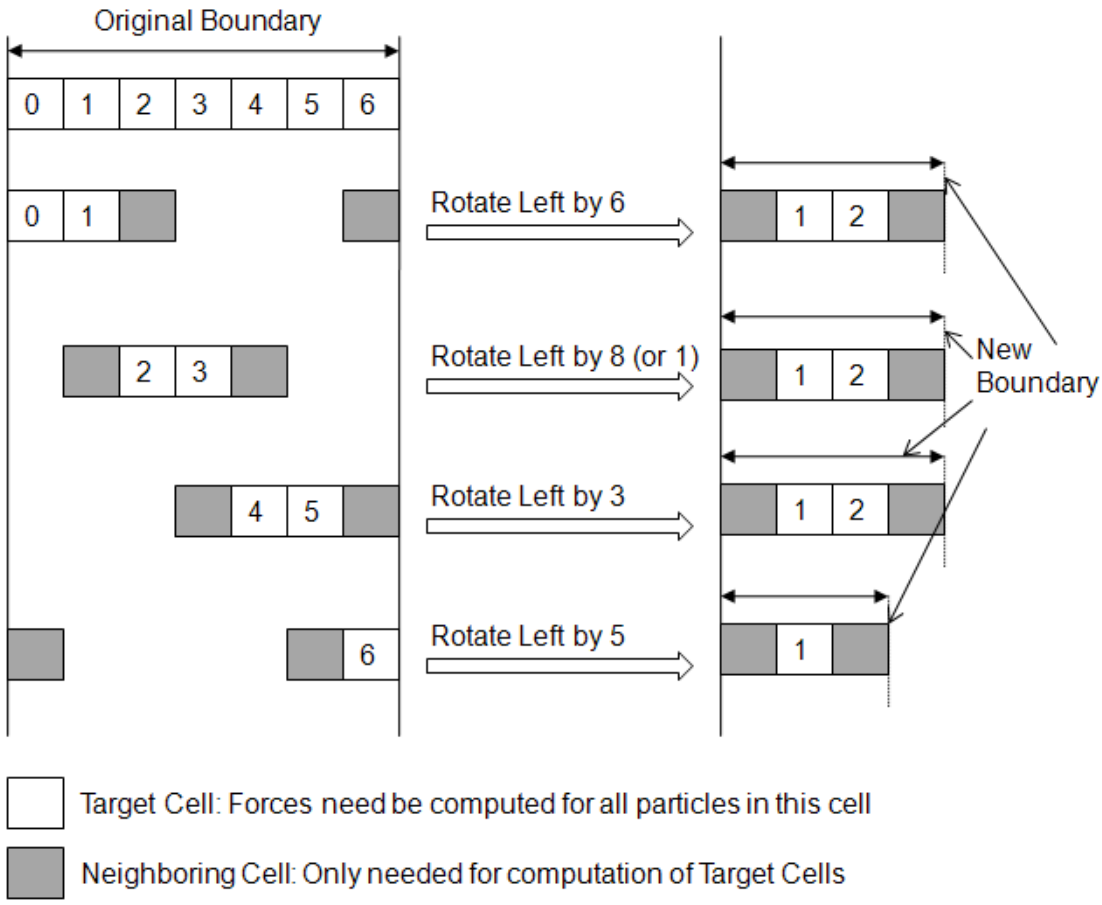


Figure 5·4: Example of partitioning of cells using Method 1 in one dimension

such that cell ID starts from 0 and increases by 1 until all required cells (target cells and neighboring cells) are accounted for. To accomplish this we implement a simple rotation in each direction. As shown in Figure 5·4, we rotate left by a certain amount such that the required cells are grouped together and index begins with 0. The amount of rotation in each dimension is determined by finding the longest range of cells in that dimension (periodicity needs to be considered too) which will not be needed for computation. The amount of rotation is simply the index of the first such cell plus the number of such cells. Figure 5·4 shows this operation for one dimension only. Once we repeat this in all three dimensions, we get the compact indexes of the

cells that we need. New number of cells are simply the number of required cells in each dimension. It should be noted that the neighboring cells may end up having unwanted interactions with other neighboring cells due to periodic boundary condition. But this is not a problem since these results will be discarded anyway.

In Figure 5·4 seven cells in one dimension are assigned to four FPGAs. For the first partition, 3 cells starting from cell 3 are not needed. Therefore, we rotate to left by 6. Cells 0, 1 now become cells 1, 2, and neighboring cells become cells 0 and 3. Similarly for the second partition, cells 5,6,0 are not needed. Therefore, we rotate to left by 8 or 1. For the next partitioning, cells 0,1,2 are not needed. So we rotate to left by 3. For the last partition, cells 1 to 4 are not needed. So we rotate to left by 5.

5.3.2 Method 2

Although Method 1 does not require modification of the FPGA kernel and has potential of reducing the amount of data communication between host and FPGA, in practice the reduction in data was not significant for the benchmark we used. This is because, to accommodate neighboring cells in all three dimensions, we end up requiring many more cells than the target cells. For example, a partitioning of 2x2x2 target cells require 3x3x3 total cells. In addition to that, for the bordering regions of the partitioning, same computations are duplicated in multiple FPGAs. These issues significantly affect performance.

To avoid these issues, we modified the FPGA kernel to allow setting two cell IDs to indicate starting and ending cell IDs for cell traversal. We now send the same data, data of particles in all cells, to all FPGAs, but each FPGA only treats a designated portion of the cells as reference cells. For example, if we have 16 cells and 4 FPGAs, FPGA[0] will compute cells[0-3], FPGA[1] will compute cells[4-7] and so on. Results from all FPGAs now will be combined at the software level.

This method allows nearly perfect scaling for the computation on the kernel, as

we can simply divide the cells evenly among the FPGAs. It only required changes in parameter setting and cell traversal logic portion of the FPGA design. The only disadvantages are as follows.

- Now we have to send all data to each of the FPGAs.
- Results from all FPGAs now need to be combined on the host.

5.4 Results

We implement 4 force computation pipelines in each FPGA and run the design at 125 MHz on the Gidel board, described in Section 4.2. The host is a Dell Precision T5400 workstation with Intel Xeon CPU E5405 (Harpertown) @2GHz. It has 4 processing cores built with 45 nm process, each having a 32 KB L1 I-Cache and a 32 KB L1 D-Cache. It has two 6MB L2 caches each shared by two cores. The workstation has a main memory of 2GB. The operating system is Ubuntu 8.04 (Linux kernel 2.6.24). We use Gidel ProcWizard (version 8.9) to generate the interface with the board and Altera Quartus II (version 9.1) for compilation and bit-stream generation.

As shown in Figure 5·1, using software pipelining and Method 2 for partitioning, we were able to hide all overhead due to send DMAs (Data trasfter from host to FPGA), except the overhead of the very first one. A bottleneck in scaling was the initial creation of cell-list, which had to be done separately in serial. Although we implemented software pipelining for reading data back from the FPGAs, this did not seem to have noticeable effect, most likely because we distributed load evenly between the FPGAs. This means, an FPGA that starts processing 4 ms later will also finishes 4 ms later. Since reading back results took only 3 ms, there was nothing to overlap in the mean time. One way to balance this would be to assign smaller workloads to the FPGAs that start later. This was not implemented, because the impact on the overall end-to-end speed-up would not be significant (see Table 5.1).

Table 5.1: Speed-up using FPGAs over a single CPU core

	Kernel-only Runtime	Kernel-only Speed-up	End-to-end Runtime	End-to-end Speed-up
1 CPU Core	∼1437 ms	1x	1957 ms	1x
1 CPU Core + 1 FPGA	129 ms	11.14x	648 ms	3.02x
1 CPU Core + 4 FPGAs	53.25 ms	26.99x	580 ms	3.37x
Theoretical limit	0 ms	∞	520 ms	3.76x

Table 5.1 summarizes the results of intra-node parallelization. The kernel-only (range-limited non-bonded force computation portion, including data communication) speed-up improved from 11.14x to 26.99x using all 4 FPGAs in the Gidel board. However, since we are using 1 CPU core only, the rest of the computation (PME, motion integration etc.) still takes significant amount of time. The end-to-end speed-up was about 3x with a single FPGA, and 3.37x with all on-board FPGAs. The improvement in kernel-only speed-up is likely to be more significant once the rest of the computation on the CPU is parallelized.

5.5 Chapter Summary

In this chapter we described our work on parallelizing the kernel to use all on-board FPGAs. Using suitable partitioning and software pipelining techniques, we were able to achieve a 3.37x end-to-end speed-up over a single CPU core. This is reasonably close to the 3.76x speed-up that is theoretically possible, given the serialization constraints that arise from the kernel design. We particularly discussed partitioning, data conversion, and communication issues in this regard.

Chapter 6

Full-parallel FPGA-accelerated MD

In this chapter we describe the integration framework we created for FPGA-accelerated MD. We begin with a description of our target software, including the reasons for choosing it. Then we present the challenges that need to be addressed in order to integrate an FPGA kernel into such a full-parallel production-level MD package. We then provide a detailed description of how the actual integration was done and what features the framework provides. Finally we evaluate the results and provide suggestions for future designs.

6.1 Description of the Target Software (NAMD)

Many of the widely used MD packages were introduced in Section 2.2.2. Among them, we chose NAMD (version 2.8) as the target software of our integration framework. NAMD is developed and maintained by the Theoretical and Computational Biophysics Group of UIUC [83, 121, 130]. It is designed for high-performance simulation of large biomolecular systems and is especially known for scaling well on various platforms [94, 112]. It runs on top of Charm++, a machine independent parallel programming system, which provides higher level of abstraction for inter-processor communication, synchronization, load-balancing etc [82, 157]. The source code of NAMD is high-level object-oriented and mostly written in C++, as opposed to low-level codes like assembly language, such as those found in GROMACS [79]. The software is widely used and the source code is

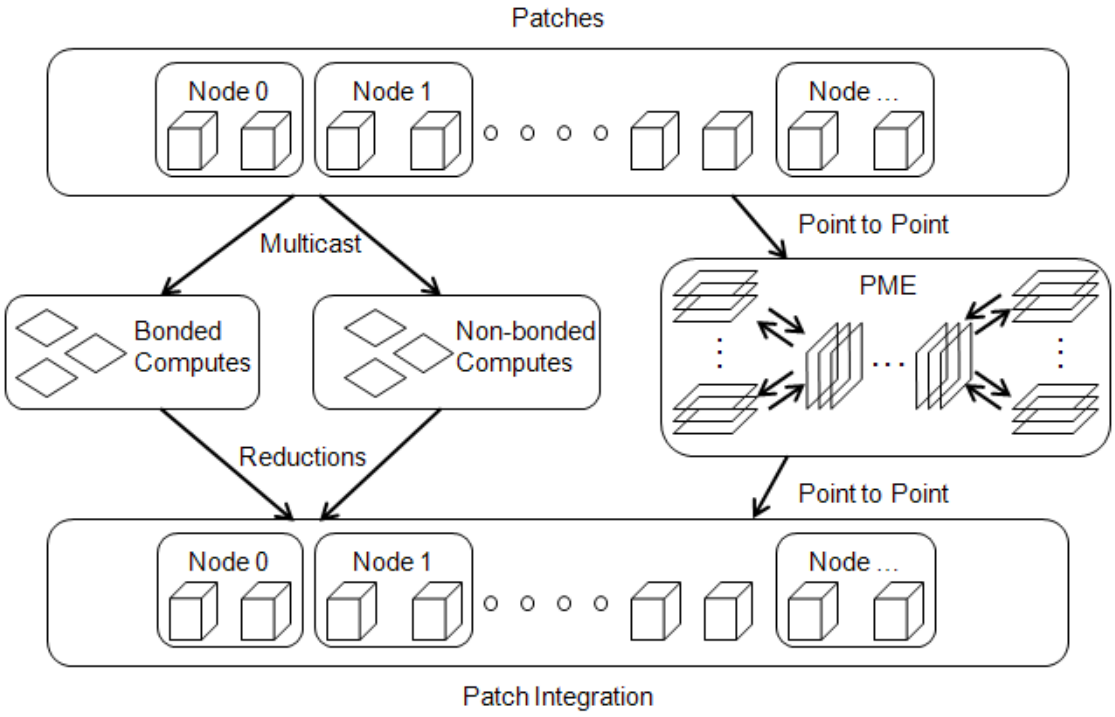


Figure 6·1: Partitioning in NAMD [84]

publicly available for free. In addition, it has a GPU-accelerated version that achieves good speed-up and reasonable scaling for complex systems.

NAMD parallelizes the simulation in two levels, as shown in Figure 6·1. First the simulation space is divided into patches, where each patch is large enough in size such that only the 26 nearest-neighboring patches are involved in bonded and range-limited non-bonded interactions. The patches fill the simulation space in a regular grid and ensures that atoms in any pair of non-neighboring patches are separated by at least the cutoff distance at all times during the simulation. Atoms are reassigned to patches as required, at regular intervals. The number of patches is determined by the size of the simulation independently of the number of processors. Additional parallelism can be generated through options that double the number of patches, by reducing size, in one or more dimensions. Patches are distributed among processors

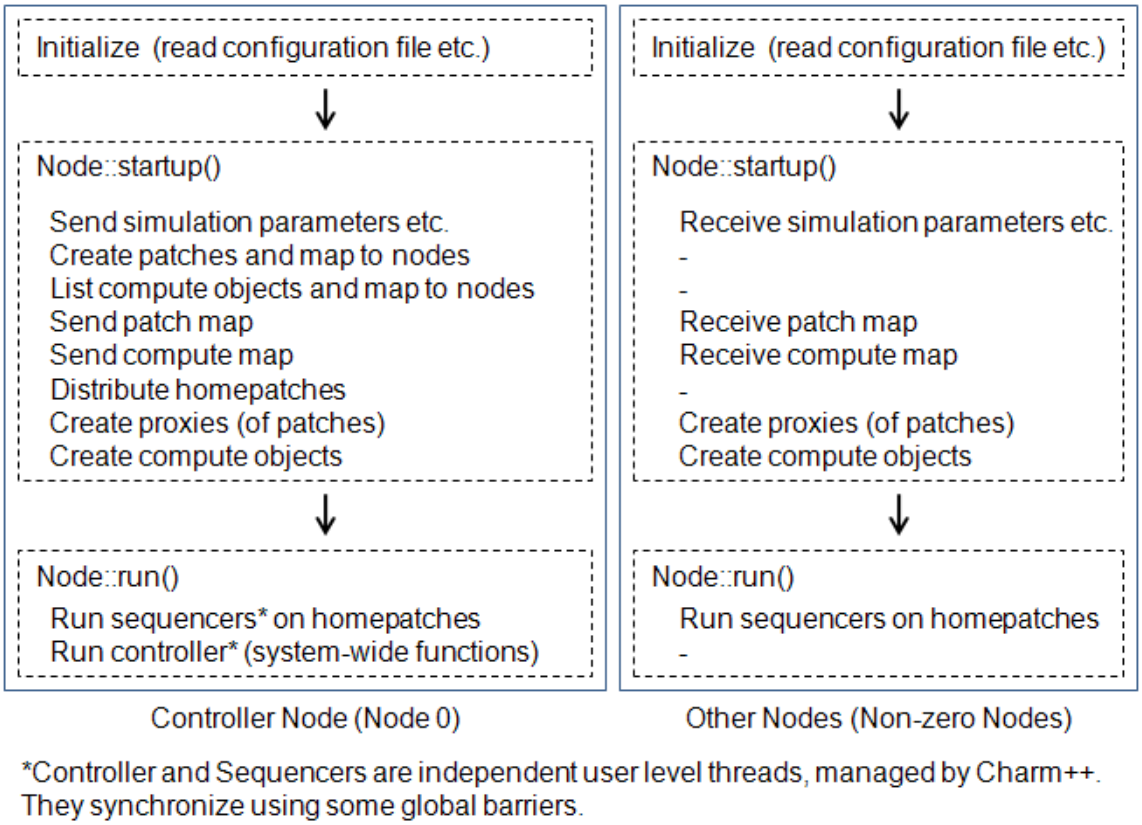


Figure 6·2: Startup sequence of NAMD

to achieve the first level of parallelism, the spatial decomposition.

The next level of parallelism is achieved by creating compute objects. Each compute object evaluates the forces between particle-pairs of the same patch or two neighboring patches. In the simplest case, one patch will have 27 such compute objects. In practice, by using Newton's 3rd law, each patch has 14 such compute objects. These compute objects are then further distributed among the processor nodes. Thus, massive parallelism is achieved.

When NAMD is run, patches are distributed as evenly as possible, keeping nearby patches on the same processor when there are more patches than processors, or spreading them across the machine when there are not. A patch is called a **homepatch** of the processor that it resides in. During the simulation, each

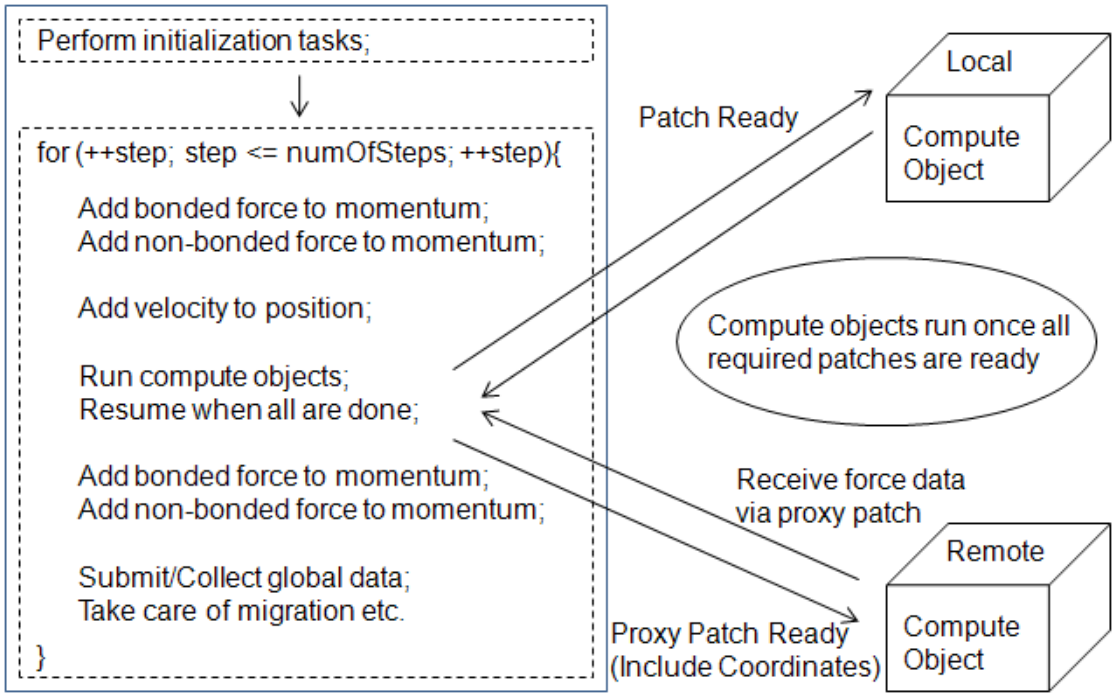


Figure 6·3: Sequencer algorithm on a homepatch

homepatch is responsible for integrating the equations of motion for the atoms of the patches it contains. After the patches, the compute objects (roughly 14 times the number of patches) are distributed across the processors, minimizing communication by grouping compute objects responsible for the same patch together on the same processors. At the beginning of the simulation, the actual processor time consumed by each compute object is measured, and this data is used later to redistribute compute objects to balance the workload among processors.

Figure 6·2 shows a simplified startup sequence of NAMD on each processor. Processor 0 or Node 0 creates a list of patches and compute objects for the target system and sends this information (patch map and compute map) to all other processors. Each processor or node then actually creates the compute objects that will run on that processor. If any compute object requires a patch that is not assigned to the same processor (called a **remote patch** for this processor), a **proxy**

patch is created for it on that processor. All communication from this processor to the actual patch, which resides in a different processor (called a **local patch** for that processor), is done via this proxy patch. This allows higher level of abstraction, as well as the least amount of communication. For example, multiple compute objects on a processor can share a single proxy patch.

Each processor also receives the data for its homepatches and then runs a **sequencer** for each homepatch. In addition, Node 0 also runs a **controller** for the global maintenance of the simulation. Controller and sequencers are independent user level threads, managed by Charm++, and they synchronize through some global barriers. Figure 6·3 shows the skeleton of a sequencer algorithm that runs on each homepatch. For every timestep of Δt, a sequencer performs force integration and motion update for all particles in the corresponding patch in a two-step leapfrog style method, as described in Section 2.1.5. In the first step, velocity is updated for $\Delta t/2$ (using the force values from the previous timestep) and position is updated for Δt. All compute objects that need data from this patch are notified at this point. The sequencer resumes after all these compute objects finish processing. Velocity is updated for the remaining of the timestep, $\Delta t/2$ (using the newly computed force values from the current timestep), and all global values are evaluated. Particle migration from one patch to another is also performed at this point. This loop continues until the end of the simulation.

In this study, we use ApoA1 benchmark of NAMD for verification and performance evaluation. The benchmark is described in Section 4.2. Here, we note that NAMD uses neighbor-list method for range-limited force computation and that we compute the long-range portion of PME every timestep, although it is possible to compute it only every few timesteps (e.g. every four timesteps).

6.2 Challenges in Integrating FPGA Kernel into NAMD

6.2.1 Scale and Complexity of the Software

The scale and complexity of NAMD probably poses the toughest challenge in integrating an FPGA kernel into it. The binary of NAMD is compiled from thousands of lines of code from hundreds of files in various different languages, e.g. C/C++, Tcl, Charm++ etc. Table 6.1 shows the scale and complexity of the package in terms of file count and line count. It considers the relevant files in the "src" folder of NAMD2.8 only. As we can see, there are around 400 source files with more than 100,000 lines of code (excluding blank/comment lines).

Understanding this gigantic software alone is an extremely challenging task. Therefore, it should not come as a surprise that integrating an FPGA kernel efficiently into NAMD has never been attempted before. The only such work that used NAMD as the baseline code used a simplified version of it [91]. The simplified code ran approximately 4x slower than the production version. Yet, the speed-up (over the simplified code) achieved by the FPGA-accelerated version was limited to only 1.3x to 3x.

In fact, the scale and complexity issue is not particularly unique to NAMD. All production-level MD packages are highly optimized for performance, which make it very difficult to extract an interface for efficient integration of FPGA kernel. The only attempt, that we are aware of, to integrate an FPGA-accelerator into a

Table 6.1: Scale and complexity of NAMD in terms of file count and line count of the source code ("src" folder of NAMD2.8 only)

File Type	Total Files	Total Lines	Blank	Comment	Code
C/C++ (.c/.C)	175	108,749	14,788	14,610	79,351
C/C++ header (.h)	201	31,898	5,542	5,590	20,766
Charm++ interface (.ci)	23	1,160			

```
#include "ComputeNonbondedInl.h"

#define NBTYPE  NBPAIR
#include "ComputeNonbondedBase.h"
#define CALCENERGY
#include "ComputeNonbondedBase.h"
#undef CALCENERGY
#define FULLELECT
#include "ComputeNonbondedBase.h"
#define CALCENERGY
#include "ComputeNonbondedBase.h"
#undef CALCENERGY
#define MERGEELECT
#include "ComputeNonbondedBase.h"
#define CALCENERGY
#include "ComputeNonbondedBase.h"
#undef CALCENERGY
#undef MERGEELECT
#define SLOWONLY
#include "ComputeNonbondedBase.h"
#define CALCENERGY
#include "ComputeNonbondedBase.h"
#undef CALCENERGY
#undef SLOWONLY
#undef FULLELECT
#undef  NBTYPE
```

```
#define NBTYPE  NBSELF
#include "ComputeNonbondedBase.h"
#define CALCENERGY
#include "ComputeNonbondedBase.h"
#undef CALCENERGY
#define FULLELECT
#include "ComputeNonbondedBase.h"
#define CALCENERGY
#include "ComputeNonbondedBase.h"
#undef CALCENERGY
#define MERGEELECT
#include "ComputeNonbondedBase.h"
#define CALCENERGY
#include "ComputeNonbondedBase.h"
#undef CALCENERGY
#undef MERGEELECT
#define SLOWONLY
#include "ComputeNonbondedBase.h"
#define CALCENERGY
#include "ComputeNonbondedBase.h"
#undef CALCENERGY
#undef SLOWONLY
#undef FULLELECT
#undef  NBTYPE
```

```
#define INTFLAG
#define CALCENERGY

#define NBTYPE  NBPAIR
#include "ComputeNonbondedBase.h"
#define FULLELECT
#include "ComputeNonbondedBase.h"
#define MERGEELECT
#include "ComputeNonbondedBase.h"
#undef MERGEELECT
#undef FULLELECT
#undef  NBTYPE

#define NBTYPE  NBSELF
#include "ComputeNonbondedBase.h"
#define FULLELECT
#include "ComputeNonbondedBase.h"
#define MERGEELECT
#include "ComputeNonbondedBase.h"
#undef MERGEELECT
#undef FULLELECT
#undef  NBTYPE

#undef CALCENERGY
#undef INTFLAG
```

Figure 6·4: Source code of "ComputeNonbondedStd.C" in NAMD2.8

full-parallel production-level MD package was described in Section 2.2.3, where an FPGA kernel was integrated into LAMMPS. The result was a 2x slowdown in end-to-end performance, although the kernel itself was reported to be 12x faster than the corresponding section of the software.

A compute object in NAMD starts computing forces once the particle data of the patch or patches it works on are updated for the current timestep. Depending on the simulation parameters, there are a vast majority of slightly different tasks that these compute objects need to do. To have one version of software code for each of these tasks will require maintaining around 27 different functions. To avoid this nightmare, NAMD uses C preprocessor to parse a single function definition containing flags to differentiate the different tasks. While this provides a single maintenance point, it makes the initial understanding of the code (from the reader's perspective) extremely challenging. For example, the functions that evaluate the

```
#undef NAME
#undef CLASS
#undef CLASSNAME
#define NAME CLASSNAME(calc)

#undef PAIR
#if NBTYPE == NBPAIR
 #define PAIR(X) X
 #define CLASS ComputeNonbondedPair
 #define CLASSNAME(X) ENERGYNAME( X ## _pair )
#else
 #define PAIR(X)
#endif

#undef SELF
#if NBTYPE == NBSELF
 #define SELF(X) X
 #define CLASS ComputeNonbondedSelf
 #define CLASSNAME(X) ENERGYNAME( X ## _self )
#else
 #define SELF(X)
#endif

#undef ENERGYNAME
#undef ENERGY
#undef NOENERGY
#ifdef CALCENERGY
 #define ENERGY(X) X
 #define NOENERGY(X)
 #define ENERGYNAME(X) SLOWONLYNAME( X ## _energy )
#else
 #define ENERGY(X)
 #define NOENERGY(X) X
 #define ENERGYNAME(X) SLOWONLYNAME( X )
#endif

#undef SLOWONLYNAME
#undef FAST
#ifdef SLOWONLY
 #define FAST(X)
 #define SLOWONLYNAME(X) MERGEELECTNAME ( X ## _slow )
#else
 #define FAST(X) X
 #define SLOWONLYNAME(X) MERGEELECTNAME( X )
#endif

.
. (About 36 more lines of similar code)
.


#undef FEPNAME
#undef FEP
#undef LES
#undef INT
#undef PPROF
#undef LAM
#undef CUDA
#undef ALCH
#undef TI
#define FEPNAME(X) LAST( X )
#define FEP(X)
#define ALCHPAIR(X)
#define NOT_ALCHPAIR(X) X
#define LES(X)
#define INT(X)
#define PPROF(X)
#define LAM(X)
#define CUDA(X)
#define ALCH(X)
#define TI(X)

#ifdef FEPFLAG
 #undef FEPNAME
 #undef FEP
 #undef ALCH
 #define FEPNAME(X) LAST( X ## _fep )
 #define FEP(X) X
 #define ALCH(X) X
#endif

#ifdef TIFLAG
 #undef FEPNAME
 #undef TI
 #undef ALCH
 #define FEPNAME(X) LAST( X ## _ti )
 #define TI(X) X
 #define ALCH(X) X
#endif
#ifdef LESFLAG
 #undef FEPNAME
 #undef LES
 #undef LAM
 #define FEPNAME(X) LAST( X ## _les )
 #define LES(X) X
 #define LAM(X) X
#endif
#ifdef INTFLAG
 #undef FEPNAME
 #undef INT
 #define FEPNAME(X) LAST( X ## _int )
 #define INT(X) X
#endif
#ifdef PPROFFLAG
 #undef FEPNAME
 #undef INT
 #undef PPROF
 #define FEPNAME(X) LAST( X ## _pprof )
 #define INT(X) X
 #define PPROF(X) X
#endif
#ifdef NAMD_CUDA
 #undef CUDA
 #define CUDA(X) X
#endif

#define LAST(X) X
void ComputeNonbondedUtil :: NAME
 ( nonbonded *params )
```

Figure 6·5: A portion of source code of "ComputeNonbondedBase.h" in NAMD2.8

range-limited non-nonded forces are defined in “ComputeNonbondedUtil.h” and implemented in “ComputeNonbondedStd.C”. “ComputeNonbondedStd.C” includes another file, called “ComputeNonbondedBase.h”, multiple times with different preprocessor definitions for each inclusion. That is how various different functions are defined using just a single maintenance point, “ComputeNonbondedBase.h”. Appropriate functions are then chosen to run in “ComputeNonbondedUtil.C”, in accordance with simulation objectives. Figure 6·4 shows the source code of “ComputeNonbondedStd.C” and Figure 6·5 shows a portion of code from “ComputeNonbondedBase.h”, where various preprocessor values are actually used. Note that, even the function name itself is also generated using a defined value (ComputeNonbondedUtil :: NAME in Figure 6·5). This means, one will not be able to search for a function using its name and simply replace it with a corresponding FPGA kernel call. The preprocessor defines are also used extensively inside the body of the function. Therefore, a significant level of understanding of the entire software is required to make any meaningful change.

6.2.2 Gathering Particle Data

In NAMD, data of a particle are managed by the patch it belongs to. Patches are distributed among processors and they send particle data to the corresponding compute objects, which may not necessarily reside in the same processor. Thus, there is a certain amount of data communication already required in the CPU-only version. This amount increases significantly when we use our FPGA kernel. The reason is that, FPGA requires a relatively large chunk of data to produce results efficiently. Although this is generally true for any accelerator, because they need to amortize the additional data communication time between the host and the accelerator, it becomes especially problematic in the current design of the FPGA kernel. Our kernel uses half-moon mapping scheme (as discussed in Section 4.3.4),

which requires 18 neighboring cells of a certain cell to be present in the same FPGA. In the CPU-only version, particles in these 18 neighboring cells could be assigned to different processors, according to the distribution of the patches. But now they all need to be communicated to the processor that invokes the FPGA kernel, in every timestep. This gets further complicated by the fact that the unit of data management in NAMD is patch, not cell; and patch dimensions are much larger than cell dimensions. So, collecting data for a single neighboring cell actually requires collecting data for one or more of the neighboring patches, increasing the amount of data communication further more.

6.2.3 Overlapping Communication and Computation

As mentioned before, NAMD uses the Charm++ parallel programming system and runtime library, where the computation is decomposed into objects that interact by sending messages to other objects on either the same or remote processors. These messages are asynchronous and one sided. This means, a particular method is invoked on an object whenever a message arrives for it rather than having the object waste resources while waiting for incoming data. This message-driven programming style effectively hides communication latency of one compute object behind the computation time of other compute objects. This advantage decreases when we use the current FPGA kernel. The problem is caused by the previously stated fact that the FPGA requires a relatively large chunk of data to get started. Therefore, a large amount of communication needs to take place before the kernel can start computation. And a large amount of data also becomes ready at once, when the kernel finishes. This lack of finer granularity makes overlapping of communication and computation challenging.

6.3 Integration Methods

In this work, we aim at limiting the required amount of edit in NAMD to as low as possible. Only a few files are actually edited and all edits are separated from the original source code using a preprocessor define (ASHFAQ_FPGA), except one line in "WorkDistrib.ci". Care is take to maintain the original structure and organization of the source code. Only 3 files are added; a header file for the driver of the Gidel FPGA board, a header and and a source for the software interface of the FPGA kernel. These additional files are included in the compilation process (in the "Make" environment) and the driver module for the Gidel board is also linked appropriately. The aforementioned preprocessor define can be used to generate binary executable file for either the regular NAMD or the FPGA-accelerated NAMD.

6.3.1 Creating FPGA Compute Object

A new compute object type is defined in "ComputeMap.h" and it is named "computeNonbondedFPGAType", following the naming convention of NAMD. A regular compute object in NAMD contains either one or two patch IDs, depending on whether it computes inter-particle forces within one patch or between two neighboring patches. The FPGA kernel, however, works on more patches than these regular compute objects do. Instead of updating the structure "ComputeData" to allow saving arbitrary number of patch IDs, we only added four additional numbers. The first one is the ID of a patch, and the rest are the number of patches that the FPGA compute object is responsible for, in three dimensions (x, y, and z).

All patches are initially assigned three coordinates or indexes to identify them uniquely, and patch ID is only a simple function of those coordinates as shown in Equation 6.1.

$$Patch\_ID = ((zIndex \times yDim) + yIndex) \times xDim + xIndex \tag{6.1}$$

Here, $xDim, yDim, zDim$ are number of patches in x, y, z directions and $xIndex, yIndex, zIndex$ are the indexes of the patch in x, y, z dimensions ($zDim$ not used in the equation).

Using the four numbers of the FPGA compute object, we can easily express the space (in terms of patches) an FPGA compute object is responsible for. For example, if there are 6x6x4 patches, (1,2,2,1) will mean that the FPGA compute object is responsible for all patches whose indexes (x, y, z) lie within (1-2, 0-1, 0-0), a total of 4 patches (patch IDs: 1, 2, 5, 6). A new function, "storeComputeFPGA", is also defined to store the FPGA compute object type in the map or list of the compute objects (compute map). This function saves all parameters of the FPGA compute object, including the ID of the node or the processor that this compute object will be assigned to and run on. The function is implemented in "ComputeMap.C".

During the runtime, an FPGA compute object is listed in the compute map, along with other regular compute objects of NAMD, and mapped to a node. This is done in "WorkDistrib::mapComputes" function in "WorkDistrib.C", which is called on Node 0 only, as shown in Figure 6·2. A new function, "mapComputeNonbondedFPGA", is defined in "WorkDistrib.h" and implemented in "WorkDistrib.C" for this purpose and it is called from "WorkDistrib::mapComputes". After all processors or nodes receive the compute map, the node that owns the FPGA compute object actually creates and initializes the object. This procedure is the same as with any other compute object in NAMD and the source code is in "ComputeMgr.C". The function "ComputeMgr::createCompute" is modified to accommodate the creation of the FPGA compute object. Finally, the actual FPGA compute object is defined and implemented in a new pair of files, named "ComputeNonbondedFPGA.h" and "ComputeNonbondedFPGA.C" respectively.

The internal functionality of the FPGA object is basically the same as described in Chapter 5. Data are converted from patch data structure to cell-list data structure and then sent to the FPGA by DMA. A "done" signal is received after the kernel finishes, when the results are copied back to the host and converted to patch data structure. Software pipelining is used to minimize serial overhead. A major difference between the version described there and this final version is that, in this version, while the FPGA kernel runs, CPU overlaps other computations using the Charm++ message passing infrastructure. This will be described in Section 6.3.2.

It should be noted that multiple FPGA compute objects can be created in this fashion, depending on the input parameters from the user. The user can provide the total number of partitions (each partition represents one FPGA compute object) or number of partitions in each direction. The files "SimParameters.h" and "SimParameters.C" are modified to accommodate a few new parameters. However, in this work we create one FPGA compute object for every four processors, since our target system consists of a quad-core CPU and a Gidel FPGA board. This means, for actual runs using the FPGA board, only one FPGA object is created.

Since the range-limited non-bonded forces will be computed on FPGA now, we need to modify the original NAMD functions that would otherwise do this job. This is done by updating the corresponding function definition in "ComputeNonbondedBase.h" such that it does not compute forces when FPGA is computing. More discussion on this will follow in Section 6.3.3.

6.3.2 Managing Data Communication

We discussed in Section 6.1 how proxy patches are created when the patches required by one or more compute objects are not on the same processor. This is done in the "createProxies" function in "ProxyMgr.C". We extend the function to accommodate necessary proxy patch creation for the FPGA compute object as well. We make

the “ProxyMgr” a friend class of “ComputeMap” class (in “ComputeMap.h”). This allows us to access the compute map data of the FPGA compute object and create proxy patches for it. The FPGA kernel requires all neighboring patches of the original compute target patches to be on the same processor. If any of them is not on the same processor, a proxy is created for it accordingly. As mentioned before, only one proxy of a patch is created on a processor even if multiple compute objects require it.

In every timestep, “WorkDistrib::messageEnqueueWork” function is called for every compute object once the corresponding patches (or proxy patches) become ready. This in turn calls the “doWork” function of the compute object, as long as there are things to do. This is done by sending a message for the corresponding compute object from “messageEnqueueWork”. A new function, called “enqueueFPGA”, is defined and implemented in “WorkDistrib.h” and “WorkDistrib.C” respectively, following the convention of regular compute objects of NAMD, to do the same (call “doWork”) for the FPGA compute object. This function is also registered in “WorkDistrib.ci” for Charm++ message passing. The net effect is that, like other compute objects, the function “messageEnqueueWork” is also called for the FPGA compute object once the required patches (or proxy patches) are ready. The function “messageEnqueueWork” in turn calls the “doWork” function of the FPGA compute object. The “doWork” function is implemented, along with other necessary code for the object, in “ComputeNonbondedFPGA.C”.

The FPGA compute object uses repeated calls to “messageEnqueueWork” to overlap other computation and communication while it is waiting for the FPGA kernel to finish. This is done by using a flag named “fpga_work_started”. When the compute object runs for the first time during a timestep, computation on the FPGA kernel is kicked off (including the DMA transfers) and this flag is set to 1. Then

repeated calls to "messageEnqueueWork" are made to check if the kernel finished computing (if "done" signal is received). The flag is used to distinguish whether the FPGA compute object should kick off the FPGA kernel or wait for the "done" signal. After receiving the "done" signal, the flag is reset to 0. For the calls to check the completion of the kernel, the priority of the message (the message that is used to enqueue the "doWork" function) is lowered such that other computation can be overlapped while the FPGA kernel is running. This is done by defining a new level of priority, "PROXY_DATA_PRIORITY", in "Priorities.h" (following the convention used in GPU-accelerated NAMD).

6.3.3 Computing Energy and Handling Exclusion

In a typical MD simulation, force is computed every timestep but energy is not. This is because, computation of energy requires additional runtime and the measurement of energy is not needed to maintain the correct execution of the simulation. Rather it is only needed to study the system (e.g. stability of the simulation); and computing it every many timesteps (e.g. once in every 100 or 1,000 timesteps) suffices.

Therefore, in our FPGA kernel, we only compute forces and leave energy computation to the host. In the timestep where energy needs to be computed, the FPGA kernel is not invoked at all. Instead, the original functions in NAMD are used. But for the timesteps where only force is computed, original NAMD functions, defined in "ComputeNonbondedBase.h" and shown the last line in Figure 6·5, are modified such that they do not compute regular range-limited non-bonded forces. These forces are now computed on the FPGA.

As described in Section 4.3.5, excluded particle-pairs are handled in the FPGA by adding a saturation force. This force needs to be subtracted accordingly on the host. We accomplish this by re-using the aforementioned original NAMD functions. We add codes such that, in a timestep where energy is not computed, these

functions subtract the saturation forces for the excluded particles. The regular NAMD functions maintain three neighborlists for each particle. One for excluded particles, one for modified particles and the other for the rest. We use the first and the second to subtract forces (and also re-compute as necessary). The subtraction is done simply by reversing the sign of the forces, using a preprocessor define, following the design style of NAMD. The saturation distance, instead of the actual distance between a particle-pair, is used for subtraction when the actual distance is smaller than the saturation distance. These modifications were done by editing the files "ComputeNonbondedBase.h" and "ComputeNonbondedBase2.h".

6.4 Simulated FPGA Kernel and Other Features

One of the key features of our integrated framework is the ability to simulate the FPGA kernel. We have created a simulated version of our FPGA kernel, which can be invoked from the FPGA compute object, allowing users to study or verify the kernel without requiring an actual FPGA board. It should be noted that this simulator only mimics the functionality of the kernel and cannot simulate system level functions (e.g. DMA transfers) which require the presence of the actual hardware device.

The simulated kernel for the range-limited non-bonded force computation is defined as a separate class inside "ComputeNonbondedFPGA.h/C" and has the same interface as the actual kernel driver. It has local variables to contain various values (e.g. simulation parameters like number of cells, coefficients for table look-up etc.) required for the simulation. DMA buffers are pre-allocated, and no DMA transfer is required because the host (FPGA compute object) is already designed to copy to and from the DMA buffers. Another major difference between the real kernel and the simulated version is that, the simulated version still consumes CPU runtime. With the simulated FPGA kernel, the end-to-end runtime is roughly 4x

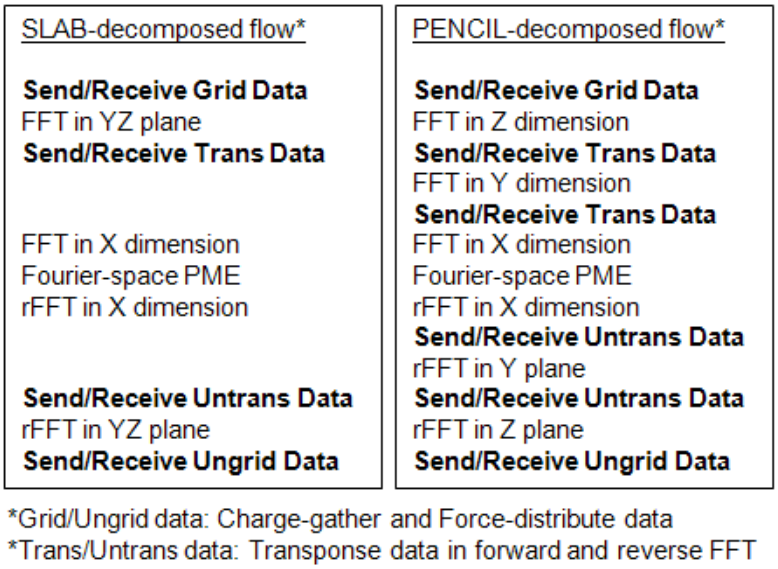


Figure 6·6: Two partitioning schemes for computing long-range portion of electrostatic force using the PME method

slower than the original CPU-only NAMD.

Although our FPGA kernel only computes range-limited non-bonded forces, we also implemented an interface for the FPGA-acceleration of long-range portion of electrostatic forces using PME method. This simply assumes that each processor will make a kernel call to off-load the FFT computations in PME method to the FPGA. Using this, we also provide a sample study to demonstrate what can be done using this integrated infrastructure. As data communication is a limiting factor in parallel 3D FFT, we choose to study how the amount of data communication changes when two different partitioning methods, namely slab-decomposition and pencil-decomposition, are used. We instrumented the files "ComputePme.h" and "ComputePme.C" files for this purpose.

Figure 6·6 shows the flow of computation using the two partitioning methods. In slab-decomposition, the FFT grids are partitioned in one dimension only. This results in fewer, but larger, number of partitions. On the other hand, in

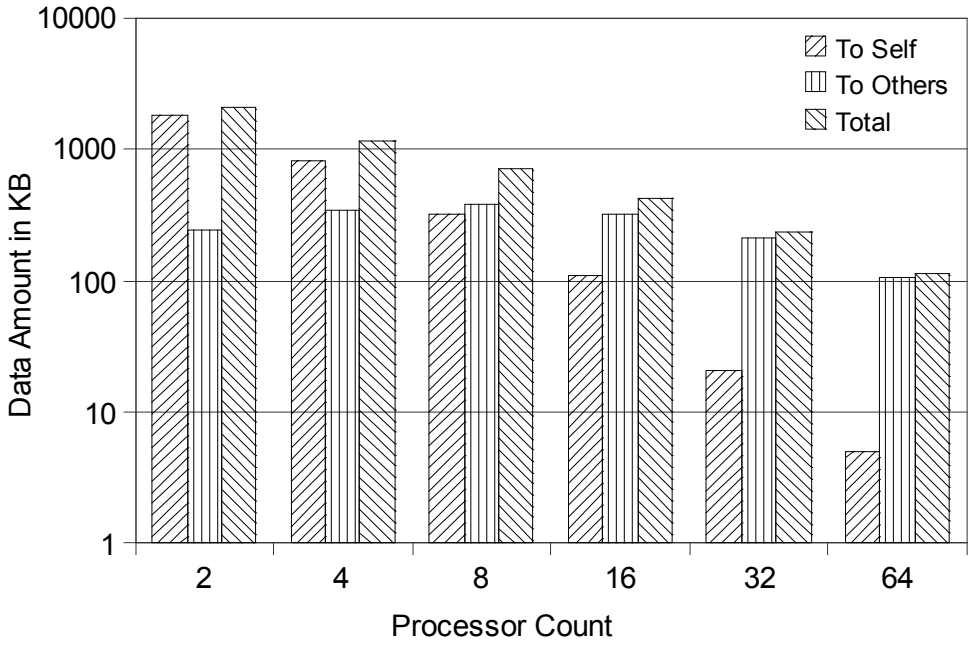


Figure 6·7: Amount of SendGrid data (in ApoA1) per processor per PME-cycle (independent of partitioning scheme)

pencil-decomposition, the FFT grids are partitioned in two dimensions. So, it results in more, but smaller, number of partitions. The first batch of communication is required to send the contribution of particle charge to grid points (Grid data). Then 2D forward FFT is done for slab-decomposition, 1D forward FFT is done for pencil-docomposition. This is followed by the next batch of communication, the communication of transpose data (Trans data). For pencil-decomposition, one more round of 1D forward FFT and Trans data communication is performed. Then a 1D forward FFT, PME computation in Fourier space and a 1D reverse FFT (rFFT) is performed. Then transpose data in reverse direction (Untrans data) is communicated, followed by a 2D reverse FFT and 1D reverse FFT for slab-decomposition and pencil-decomposition respectively. For pencil-decomposition, another round of communication of Untrans data and a 1D

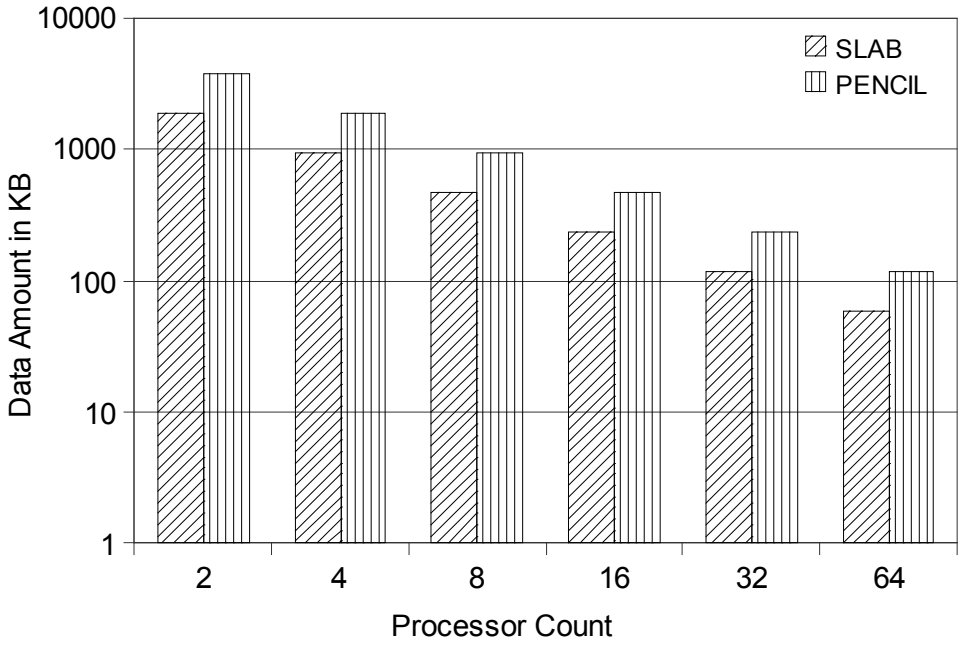


Figure 6·8: Amount of SendTrans data (in ApoA1) per processor per PME-cycle for the two partitioning schemes

reverse FFT is performed. Finally force contribution of grid points are communicated (Ungrid data) and interpolated back to particles.

Figure 6·7 shows the average amount of Grid data that is sent from one processor (SendGrid data) to other processors or to itself every time PME computation is done (PME-cycle). Since the grand total of this data (per processor data $\times$ number of processors) depends only on the patches, which are created independent of partitioning schemes or processor counts, it is constant for all processor counts (except when processor count is 1 and no communication is required). So, the total amount of SendGrid data in the graph scales linearly with the number of processors. However, as the number of processors increases, the proportion of communication to self decreases, and for large processor counts, most of the communication becomes inter-processor. It should be noted that nearly the same amount of Grid data are also received by each processor, and nearly the same

amount of Ungrid data are also sent and received by each processor.

Figure 6·8 compares the average amount of Trans data that is sent by each processor (SendTrans data) to other processors or to itself every PME-cycle. As we can see, pencil-decomposition requires twice the amount of data communication than required in slab-based decomposition. As with Grid data, Trans data also depends on patches, making the grand total of the data constant for all processor counts (except when processor count is 1 and no communication is required).

Thus the integrated framework can be used to study various characteristics of a system, which should be especially helpful in designing new systems.

6.5 Results

6.5.1 Speed-up

Using the integrated framework, the speed-up was measured using a quad-core host machine, the Gidel FPGA board and the ApoA1 benchmark. The host machine details and FPGA configuration are provided in Section 5.4, where as the Gidel FPGA board and the ApoA1 benchmark are described in Section 4.2. Figure 6·9 shows how the runtime changes from 1 CPU core to 4 CPU cores, when only CPU cores are used. As we can see, a perfect speed-up is prevented by the inter-processor data communication, although the impact of the data communication is relatively modest compared to the total runtime.

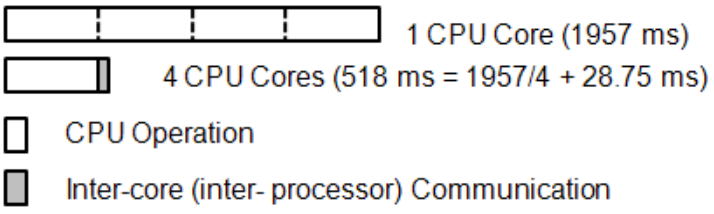


Figure 6·9: Graphical illustration of CPU-only runtime for ApoA1 benchmark in NAMD2.8

Table 6.2: Speed-up using FPGAs over a quad-core CPU

	Kernel Only	Other Computation	Data Communication	End-to-end Runtime	End-to-end Speed-up
4 CPU Cores	~359.25 ms	130 ms	28.75 ms	518 ms	1x
4 CPU Cores + 4 FPGAs	25 ms (rest overlapped)	130 ms	78 ms	233 ms	2.22x
Theoretical limit	0 ms	130 ms	28.75 ms	158.75 ms	3.26x

When the quad-core version is accelerated using the Gidel FPGA board, an end-to-end speed-up of 2.22x is achieved (a total of 1957/233 = 8.39x speed-up over a single CPU core). Table 6.2 summarizes this result and also shows the theoretical limit of achievable speed-up, 3.26x, in the current settings. The "Kernel" in the table refers to the range-limited non-bonded force computation. Next, we analyze the difference between the achieved speed-up and its theoretical limit.

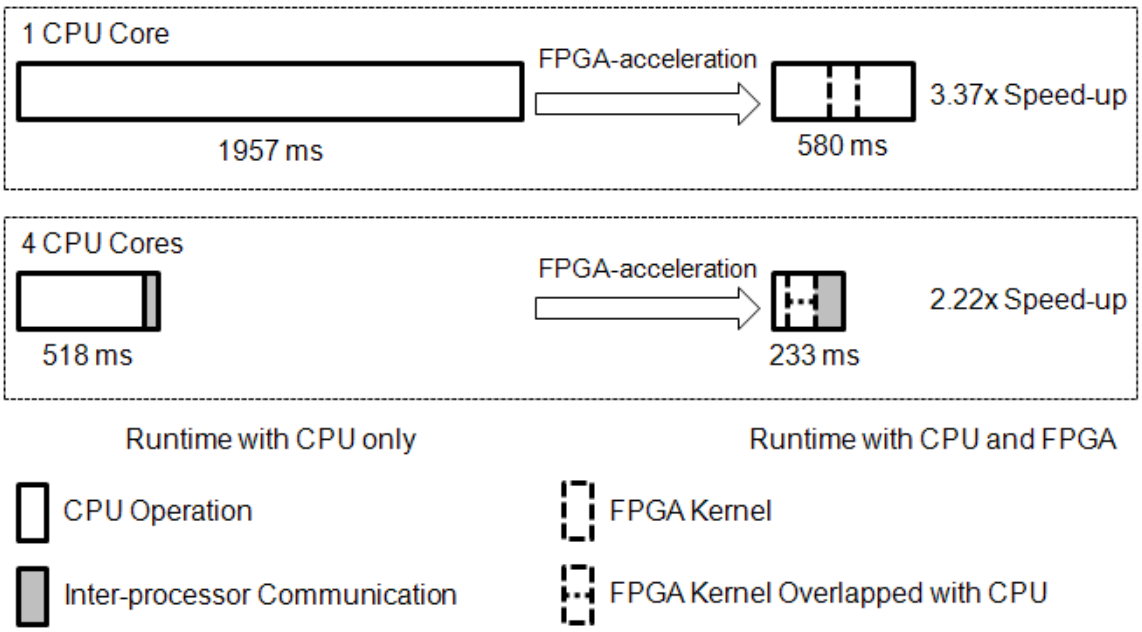


Figure 6·10: Graphical illustration FPGA-accelerated runtime for ApoA1 benchmark in NAMD2.8

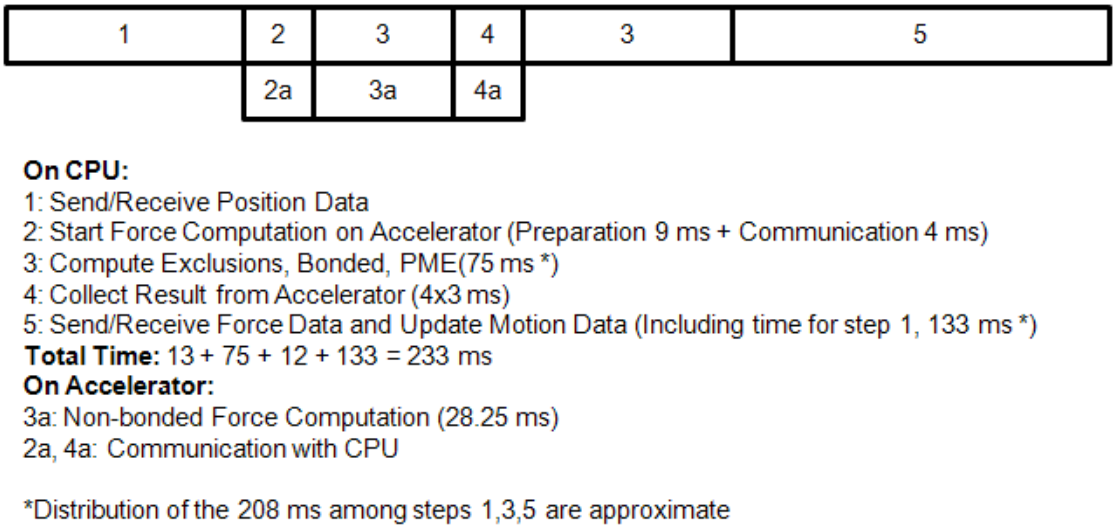


Figure 6·11: Current overlap scenario

As shown in Figure 6·10, the amount of data communication increases in the FPGA-accelerated version. This issue was anticipated and discussed in Section 6.2.2. Amount of data communication increases significantly because the FPGA kernel requires all neighboring patches of a patch to be present on the same processor to compute its force. Since FPGA kernel is launched by only one CPU core in the current benchmark and hardware setup, practically all data from all patches must be communicated to this processor. This means, the amount of patch data communication is now 4x compared to a quad-core CPU-only version. From the previously measured data communication time (from Figure 6·9, around 30 ms for each core in the CPU-only version), we conclude that this increase in data communication time is the biggest bottleneck in achieving further speed-up in the current settings.

Figure 6·11 shows the overlap scenario of the current implementation. As we can see, the computation time of the FPGA kernel is effectively hidden behind the CPU workload. This means, further improvement of the kernel computation time itself

will have modest effect on the overall speed-up. This was confirmed by running the FPGA design at faster clock speed. The end-to-end speed-up remained almost the same.

6.5.2 Re-evaluating Kernel Design

In light of the results from the final integrated version, here we re-evaluate the design of the FPGA kernel that we used. First, the kernel uses cell-list method and traverses one reference cell (called home cell) after another to evaluate the range-limited non-bonded force. While this is efficient for a hardware implementation, it requires extra data conversion time on the software. This ultimately leads to additional runtime, reducing the advantages of the FPGA acceleration.

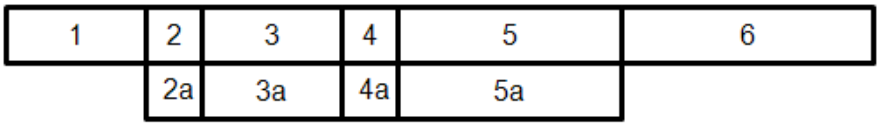
Another feature of the kernel is the half-moon mapping scheme, which was proved to improve the efficiency of the force computation pipelines on the FPGA by balancing workload. However, this requires two-thirds of the neighboring cells (18 cells) of a cell to compute its forces. The FPGA traverses through the cells one by one, each time computing interactions with 18 neighboring cells. This requires additional data communication to bring the neighboring cells to the processor that drives the FPGA.

In addition to these two issues, in the current implementation we invoked the FPGA kernels for all on-board FPGAs from the same processor. Driving each FPGA from an individual processor could save some of the serialization time. However, the above mentioned data communication issue will very likely diminish any savings earned this way. Another implementation level issue was the inability to use a large number of DMA channels for data transfer (although Gidel seems to support a large number), which could have allowed more aggressive software pipelining.

The kernel also does not support multi-step execution of PME, where the long-range portion of electrostatic force is computed once in a few cycles, as opposed to every cycle.

6.5.3 Suggestions for Future Designs

Achieving good speed-up over any highly-optimized parallel MD package requires good understating and careful consideration of the software. First, we need to address the issue that theoretical limit of end-to-end speed-up can be improved significantly. This can be done by using multi-step execution of PME, e.g. computing the long-rage portion of electrostatic force in every 4 cycles. This will increase the proportion of range-limited non-bonded computation time compared to the overall runtime. This proportion is only about 70% now, which will go close to 90% with multi-step execution of PME (note that multi-step execution also reduces data communication). A reasonable practical goal for speed-up by accelerating only range-limited non-bonded force computation will be roughly 6x - 10x. Achieving speed-up over 10x will very likely require accelerating other portions of MD, e.g. long-rage force computation, motion update etc. But, even to realize the 6x - 10x speed-up by accelerating range-limited non-bonded force computation only, several design issues will have to be handled.



On CPU:
1: Send/Receive Position Data
2: Start Force Computation for Remote Patches on Accelerator
3: Compute Exclusions, Bonded, PME
4: Collect Result from Accelerator and
Start Force Computation for Local Patches on Accelerator
5: Send/Receive Force Data
6: Collect Result from Accelerator and Update Motion Data

On Accelerator:
3a: Non-bonded Force Computation for Remote Patches
5a: Non-bonded Force Computation for Local Patches
2a, 4a: Communication with CPU

Figure 6·12: A good overlap scenario

One of the major changes required will be to enable FPGA kernels such that patches can be used directly in the kernel. Sorting the compute objects by patch and computing those objects accordingly on the FPGA is likely to achieve better results than converting them into cells and requiring data of the neighboring cells. The FPGA kernel already has very inexpensive yet highly efficient filters to create neighbor-lists on the fly. These can be used more aggressively to tackle the problem that a large proportion of the particle-pairs will be outside the cut-off distance if we use patches directly. This approach will not only reduce data conversion time significantly, but also eliminate the need for additional inter-processor communication due to the FPGA accelerator.

Computation on CPU and computation on FPGA, along with data communication, will have to be overlapped adequately too. This can be done efficiently by splitting the FPGA kernel call into at least two calls. One for computing forces that involve remote patches and the other for computing forces for local patches, as shown in Figure 6·12. It should be noted that having the fastest kernel does not necessarily guarantee the best speed-up. Efficient overlap is one of the most important keys in achieving good speed-up, especially for parallel implementations.

An open issue with accelerators in general is that, accelerated systems do not scale well with a large number of compute nodes. This is unavoidable with current hardware, since inter-processor communication is very slow compared to the computation on an accelerated system. We will comment on how FPGAs can be used to solve this issue in Chapter 8.

6.6 Chapter Summary

In this chapter we described our work on the integrated framework for FPGA-accelerated MD. We introduced NAMD as the target software and discussed the challenges in integrating an FPGA kernel into such a highly-optimized parallel software. We described how the integration was actually done in this work and how the framework can be used for various architectural studies. Using this framework, we achieved an end-to-end speed-up of 2.22x over a quad-core CPU, making it the first FPGA-accelerated MD ever to achieve a positive end-to-end speed-up. We analyzed our speed-up result and compared it to its theoretical limit. We also discussed the issues with the current kernel design and provided guidelines for future designs.

Chapter 7

Communication Requirements for FPGA-centric MD

FPGA-centric clusters use FPGAs for both computation and communication and thereby address three fundamental problems of future High Performance clusters: efficient use of silicon, power, and removing communication bottlenecks. In this chapter we instrument our framework, described in Chapter 6, to determine the plausibility of using such clusters for MD, in particular by determining the communication requirements for such a cluster. We begin by reviewing MD on a single FPGA-based node and use the estimated performance of an optimized implementation to determine the time budget for the communication. We then quantify the data communication characteristics for a production MD code (NAMD) in two ways: analytically and by instrumenting the code. We apply this information to clusters of various sizes and node complexity. The conclusion is that a cluster with 256 FPGAs distributed in 64 nodes is appropriately provisioned, even for modest simulations, with a bidirectional 3D torus where each link consists of 1-2 of an FPGA's serial ports.

7.1 Justification of FPGA-centric MD

7.1.1 Communication Bottleneck in MD

A critical issue in high performance MD is data movement. By Amdahl's Law, scalability of parallel applications that have significant communication depends on

the number of nodes, the problem size, and the ratio of communication to computation. Accelerators are an additional factor in that they add additional communication overhead between host and accelerator and also reduce the communication-computation ratio. Right now MD scales well to 100s of CPU nodes for well-designed codes and sufficiently large simulations (e.g., [20, 132]), and for much smaller simulations using dedicated hardware [154].

The problems are as follows:

1. **Scaling with accelerators.** Adding accelerators to nodes makes scalability more challenging, as seen, e.g., with respect to three of the most prominent MD codes, NAMD [131, 161], AMBER [151], and GROMACS [62]. The problem is two-fold: (i) there is additional overhead to move data on and off of the accelerator and (ii) the reduction in compute time per time-step makes the communication latency harder to hide.

2. **Small problems for long time-scales.** At some point no matter what the hardware, the number of nodes will become too large or the problem size too small to achieve strong scaling. It appears, however, that significant MD problems in the range of 10s of thousands of particles can currently achieve strong scaling only with specialized hardware [154].

3. **Technology trend.** Nodes will continue to get ever more powerful. Process technology appears able to advance with Moore's law [115] for a few more generations even while operating frequencies remain static. The fraction of silicon available for computation is increasing both through the use of accelerators (on and off chip) and with the increased emphasis on performance in new generation high-end CPUs [173]. All these factors decrease the communication-computation ratio.

The implication is that not only is communication critical now, but it is only going to get more so. In current high-end clusters this communication includes transfers between accelerator and CPU. While projected integration of accelerator and CPU will help, problems 2 and 3 remain. The obvious solution has two parts: provisioning nodes with appropriate bandwidth and reducing latency by integrating communication directly into the computation chip. The first part is straightforward and the second has been well studied. Here are some solutions, past, present, and future:

Past: In the early 1990s several projects looked at integrating communication and run-time systems into the chip fabric (e.g., [43]) and reduced latency of the entire communication stack to a few cycles.

Present: A current solution is the Anton processor [153] which uses dedicated hardware to pipeline communication with computation and effectively reduce amortized communication latency to zero.

Future: The need for direct communication solutions for GPUs has been stated by Patterson as one of his Top Three Next Challenges [127] and proposed by Dally in his plan for GPUs as Exascale accelerators [42].

7.1.2 FPGAs for Data Communication

In this work we examine a solution for the present that is built entirely with commodity hardware. In particular, we examine the possibility of, and requirements for, large-scale clusters scalable for MD based on FPGAs as the central component *for both computation and communication*. This design is motivated as follows.

- It has been demonstrated that FPGAs can be competitive for single node MD acceleration. In particular, the range-limited parts of the 92K ApoA1 benchmark can be computed in less than 20 ms per time-step using high-end

FPGAs [33]. The range-limited force is computed with full electrostatics and is compatible with production MD codes.

- Many prototype FPGA-centric clusters have been built, including products from BEEcube [16] and SciEngines [145]. One example, the Novo-G Reconfigurable Supercomputer at the University of Florida, has a peak performance of 100 GFLOPS while drawing less than 12 kilowatts power and requiring no additional cooling infrastructure [58].

- The primary market for high-end FPGAs is as communication processors, especially in high-end routers [28, 40]. The use of FPGAs for communication offload is also well established. For example BittWare [18] has long had products that use this approach for large-scale DSP. For HPC, computers with FPGAs for communication are currently in use for physics computations [13, 142].

7.2 Target Systems

7.2.1 FPGA-based Systems

We briefly state our assumptions about the target systems with FPGA-based accelerators. They are typical for current products.

- The overall system consists of some number of standard nodes. Typical node configurations have 1-4 accelerator boards plugged into a high-speed connection (e.g., the Front Side Bus or PCI Express). The host node runs the main application program. CPUs communicate with the accelerators through function calls.

- Each accelerator board consists of 1-4 FPGAs, memory, and a bus interface. On-board memory is tightly coupled to each FPGA either through several interfaces

(e.g., 6 x 32-bit) or a wide bus (128-bit). 4GB - 64GB of memory per FPGA is currently standard.

- Besides configurable logic, the FPGA has dedicated components such as independently accessible multiport memories (e.g., 2000 x 1KB) called Block RAMs (or BRAMs) and a similar number of multipliers. FPGAs used in High Performance Reconfigurable Computing typically run at 200 MHz, although with optimization substantially higher operating frequencies can sometimes be achieved.

- FPGAs have substantial I/O and communication capability. High-end FPGAs have on the order of 1000 I/O pins which have latency of 5-6 ns. They also have dozens multi-Gbps interconnects; some members of the Altera Stratix V family have dozens of 14 Gbps interconnects. Latency on the Gbit interconnects for FPGA-FPGA communication can be less than 100 ns. FPGAs on a board typically communicate via I/O with latency of a few cycles while inter-node FPGA-FPGA communication is typically done via the Gbit interconnects.

We are initially targeting the Novo-G, the High Performance Reconfigurable Supercomputer at the University of Florida [58], which has nearly 400 FPGAs and both a commercial off-the-shelf interconnect and direct FPGA-FPGA connections.

7.2.2 MD on FPGA-based Systems

We now give an overview of the assumed accelerator design (details were described in Chapter 4 with suggestions for improvement in Chapter 6) with the goal of justifying the communication budget in the next section.

Our accelerated MD system runs on one to four FPGAs of a Gidel PROCStar III board. The PROCStar III is a PCI-based system with an 8-lane PCI Express (PCIe x8) host interface. Each processing unit contains an Altera Stratix III SE260 FPGA

and three memory banks, each of which has a 128-bit interface. The system has also been tested in simulation on an Altera Stratix-V, the current generation FPGA. The host processor runs the main application program and communicates with the accelerator board through function calls. The program is partitioned as follows. The accelerators process the range-limited forces, while the hosts process the balance of the computation. In each iteration, particle data are downloaded to the accelerator and forces are uploaded to the host.

Main computation pipeline: The main computation pipeline is partitioned into two levels. The first is the filter pipeline; it determines whether the particle pair has a non-zero force. The second level, the force pipeline, accepts the particle pairs that pass the filter and computes their mutual force. Six to eight force pipelines fit on the Stratix-III, each with 8-10 filter pipelines. This number doubles for the Stratix-V.

Host-accelerator data transfers: At the highest level, processing is built around the timestep iteration and its two phases: force computation and motion update. During each iteration, the host transfers position data to, and acceleration data from, the accelerator's on-board memory.

Board-level data transfers: Force calculation is built around the processing of successive home cells. Position and acceleration data of the particles in the cell set are loaded from board memory into on-chip caches. When the processing of a home cell has completed, acceleration data is written back. Focus shifts and a neighboring cell becomes the new home cell.

Force pipelines to accumulation cache: To support an optimization due to Newton's Third Law, two copies are made of each computed force. One is accumulated with the current reference particle. The other is stored by index in one of the large BRAMs on the Stratix.

The performance for the range-limited force computation is as follows. Each of

the 12-16 pipelines on the Stratix-V runs at 200 MHz and completes a payload force calculation every cycle. All data transfer latencies are hidden as described in [33] yielding a compute time of less than 20 ms per time-step, or roughly 10x the speed of an 8-core CPU.

A 3D FFT kernel for long-range portio of PME is not implemented yet, but highly efficient IP is available for 1D FFTs from both Xilinx [177] and Altera [5]. The remaining parts of the computation are mostly the communication described in the next section.

7.3 MD Communication and Support Requirements

In this section we first describe the major types of communication. Then we quantify the communication analytically and experimentally.

7.3.1 MD Communication Description

Several tiers of inter-processor communication take place during a parallel run of MD. The majority of these transfers are due to the non-bonded force computation, while most the rest are required for maintenance of the simulation, e.g., re-assignment of particles to nodes as particles move during each timestep.

Range-limited: Position data of each particle need to be updated every timestep before the range-limited non-bonded forces can be computed for that particle. The owner node of a particle (the node that is responsible for updating the motion data of that particle) sends position data (and also charge data as required) to all nodes that require this data at the beginning of a timestep. After computing the forces, these nodes send the force data back to the owner node. The owner node then combines these results to update the motion of that particle for that timestep.

With cell-based decomposition (or neighbor lists) the amount of communication

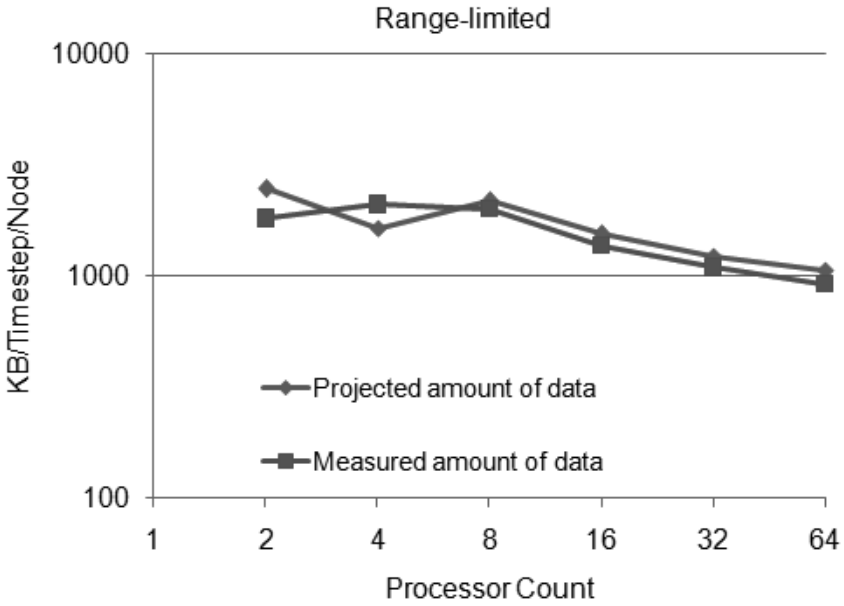


Figure 7·1: Projected and measured data communication for range-limited forces

per particle is a constant with respect to the problem size and the number of processors. Assuming spatial decomposition and assignment of those spatially decomposed cells to nodes we have two scenarios defined by the ratio of problem size to cluster size.

For large problem to cluster size, multiple cells are assigned to each node, potentially cubes of slabs. If the number of cells per node is the same for all nodes, each node can simply compute forces for 13 neighboring cells for each of the cells that is assigned to it. If that is not the case, e.g., if some nodes own two cells, some own 3 cells, then additional decomposition involving the force computation improves load-balance.

For small problem to cluster size, multiple nodes are assigned to each cell. There is further decomposition of neighbor pairs to node. That is, force computation of a cell will be decomposed such that it can be done in parallel by multiple nodes. In the

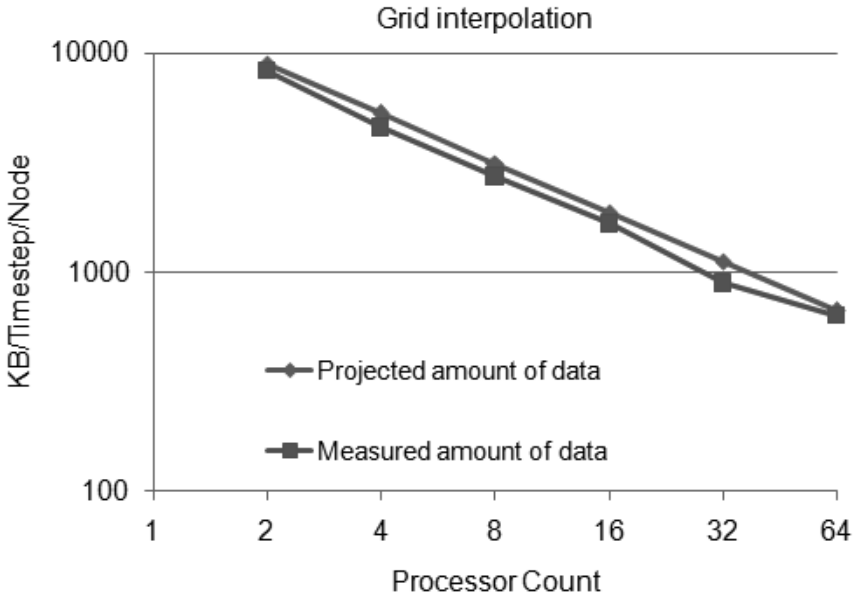


Figure 7·2: Projected and measured data communication for grid interpolation

simplest case, there are 13 cell-pairs for each cell. These 13 cell-pairs are assigned to multiple nodes for computation. The motion integration will still take place in one node.

Data communication for range-limited force computation is typically limited to neighboring nodes, as long as node assignment corresponds to the physical simulation space.

Grid Interpolation: For the long-range part of the electrostatic force computation, first the charge contribution of a particle to nearby grid points need to be interpolated. This contribution is computed by the owner node and is sent to the processor responsible for that particular grid point. These grid data are then used to compute forces in Fourier space, using the 3D FFT, after which the computed result is sent back to the owner node for motion update. While sending the force contribution may be restricted to a subset of neighboring nodes, receiving such data

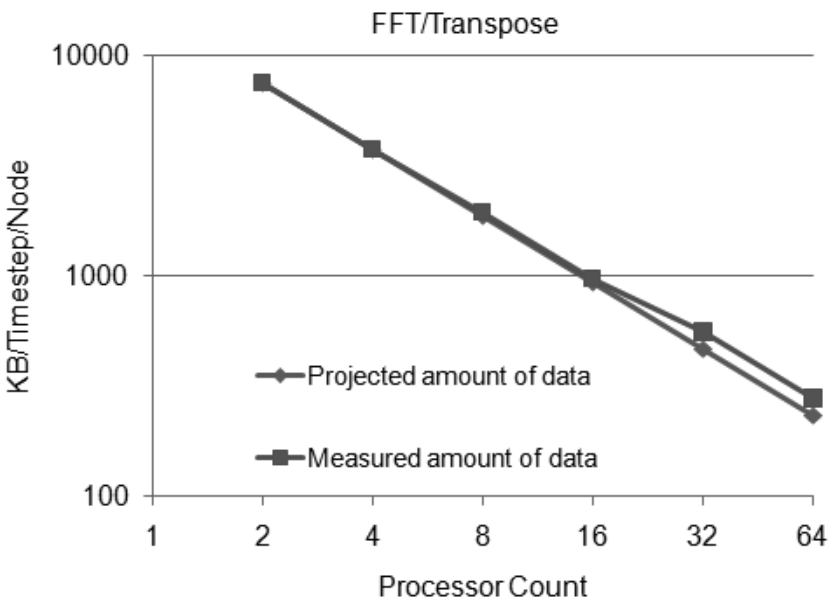


Figure 7·3: Projected and measured data communication for FFT/transpose

involves distant nodes, since unit of grid point assignment can span the entire length of one or two dimensions of the simulation space. The same is true for sending and receiving computation results since this is merely the same communication in the reverse direction.

FFT/Transpose: The 3D FFT-based long-range force computation requires all-to-all communication among nodes. if the 3D FFT uses slab-based decomposition, then this involves sending/receiving transpose data once each for the forward and reverse FFT. For pencil-based decomposition, this needs to happen twice for both the forward and reverse FFT.

Others (Migration etc.): While the above three types of communication comprise the majority of the data communication, there are a few other types of communication required to maintain the simulation. These include migration of particles and other system-wide communication among nodes, e.g. synchronization. Migration becomes

necessary when a particle crosses the boundary of its current cell by a predetermined margin. At that point, it is re-assigned to another cell and this information must be communicated among the involved nodes. Data communication required for migration of particles is only a fraction of other communication and is limited among neighboring nodes.

7.3.2 MD Communication Characterization

Communication in MD is primarily determined by the size of the problem, the number of computing nodes, and the compute capability of the nodes. In this subsection we provide some simple formulas to predict this communication amount and validate them using a production MD code. The data are shown per node, assuming a direct communication between every node-pair. We use NAMD2.8 [130] and ApoA1 benchmark for measuring data communication. The source code of NAMD2.8 was instrumented for this purpose as described in Chapter 6. The ApoA1 benchmark consists of 92,224 particles and uses periodic boundary condition with an original simulation box of $108\mathring{A} \times 108\mathring{A} \times 78\mathring{A}$. It uses a cut-off radius of $12\mathring{A}$ for the range-limited force computation and a switching function is applied to smooth the force when the inter-particle distance is between $10\mathring{A}$ and $12\mathring{A}$. The Coulomb force is evaluated using PME. The cell dimension used for cell-decomposition is approximately $18\mathring{A}$, which results in $6 \times 6 \times 4$ cells. The number of grid points used for PME is $108 \times 108 \times 80$.

Range-limited: For a large number of cells per node, the communication required for the range-limited force can be approximated by assuming the import region to be a spherical volume. If the volume of region owned by a node is $4 \times \pi \times r^3/3$, then the import volume would be $4 \times \pi \times (r+m)^3/3 - 4 \times \pi \times r^3/3$, where r is the radius of the spherical volume owned by the node and m is the inter-particle interaction distance.

The number of particles can be derived by using the density of the system, which is 0.1 for a typical bio-medical system.

For a small number of cells per node, this approximation does not hold due to the cell-based decomposition. However, assuming that neighboring cells are assigned to each node, we can predict the communication using the following equation

$$D = (C + 13) \times P \times d \times 2 \tag{7.1}$$

where D is the amount of data per node, C is the number of cells per node, P is the number of particles per cell and d is the amount of data amount per particle. The 2 at the end of the equation is for a node's contribution to other nodes.

For multiple nodes per cell, most of the computation requires importing data from other nodes. This results in a communication amount of (Number of compute objects per node + 1) $\times$ (Number of Particles per cell) $\times$ (Amount of Data per particle) $\times$ 2.

Figure 7·1 shows the projected and measured data communication for range-limited force computation. For low processor counts (2 and 4), where there are many cells per node, we use import volume; for the rest we use Equation 7.1

Grid Interpolation: The communication for the grid interpolation can be approximated by the following equation

$$D = (G/n) \times d \times 4 \times n^{1/4} \tag{7.2}$$

where D is the amount of data per node, G is the total number of grid points, n is the number of nodes and d is the amount of data per grid point. The 4 in the equation accounts for sending/receiving of grid data before and after the force computation. The $n^{1/4}$ is empirically determined.

Figure 7·2 shows the projected and measured data communication for grid interpolation.

FFT/Transpose: The communication for 3D FFT can be approximated by the following equation (for slab-based decomposition)

$$D = (G/n) \times d \times 4 \tag{7.3}$$

where D is the amount of data per node, G is the total number of grid points, n is the number of nodes and d is the amount of data per grid point. The 4 in the equation accounts for sending/receiving of grid data in the forward and reverse FFTs.

Figure 7·3 shows the projected and measured data communication for FFT/transpose stages.

7.4 FPGA Cluster Communication Requirements

In this section we estimate the communication requirements for MD of an FPGA-centric cluster. Figure 7·4 shows an estimate of required time per timestep for various simulation sizes on a CPU-only system, assuming a perfect linear speed-up. These numbers are based on benchmark results from the NAMD web site [130]. This data, along with the communication characterization from the previous section, can be used to determine the required bandwidth per node. Figure 7·5 shows this data. The values for node counts 128 and 256 are computed using the method from the previous section.

The bandwidth requirement of an accelerator-based system can be roughly determined by comparing to an equivalent number of CPU cores. With high-end FPGAs, it should be possible to achieve order of magnitude speed-up over an 8 core CPU, assuming communication is also improved as required. This is shown in

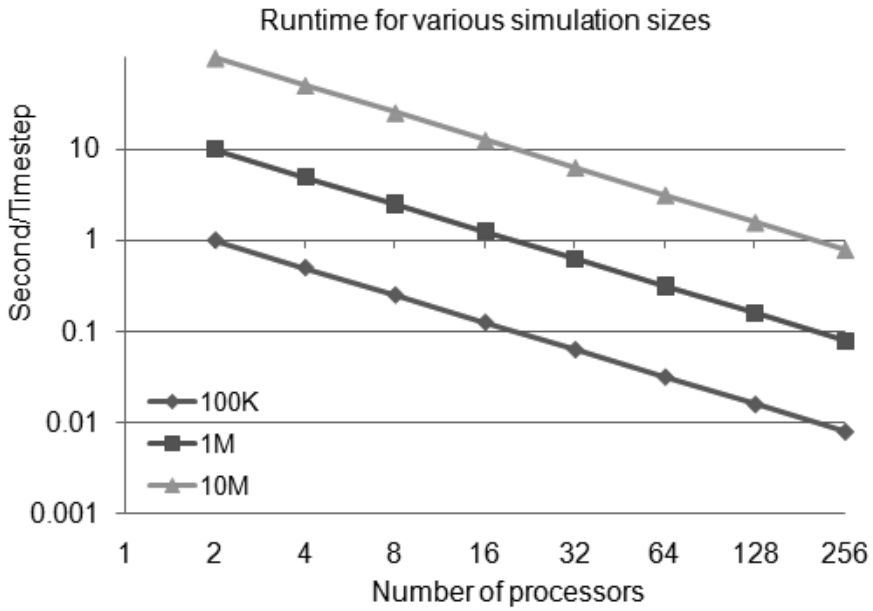


Figure 7·4: Time per timestep for various simulation sizes and core counts assuming perfect scaling. This is computation only and gives the time budget for communication

Figure 7·5 in terms of CPU core-equivalence. For example, an FPGA-based node equivalent to 256 CPU cores will have the bandwidth requirement of the top-most line in this graph.

It should be noted that several adjustments are likely to be necessary as system and implementation are specified further. The bandwidth requirement shown is the absolute amount of data, assuming an all-to-all network. In practice, we must consider the header, the amount of hops, and other implementation issues. Therefore, actual bandwidth requirement is likely to be at least twice that shown in Figure 7·5. Although this data is derived using a benchmark of about 100K particles, it should give a reasonable estimate for larger simulations too, since both runtime/timestep and amount of data communication will increase with the size of the simulation. Another point to note is that, this estimation includes intra-node communication, which is significant at low node counts, but can be ignored for high

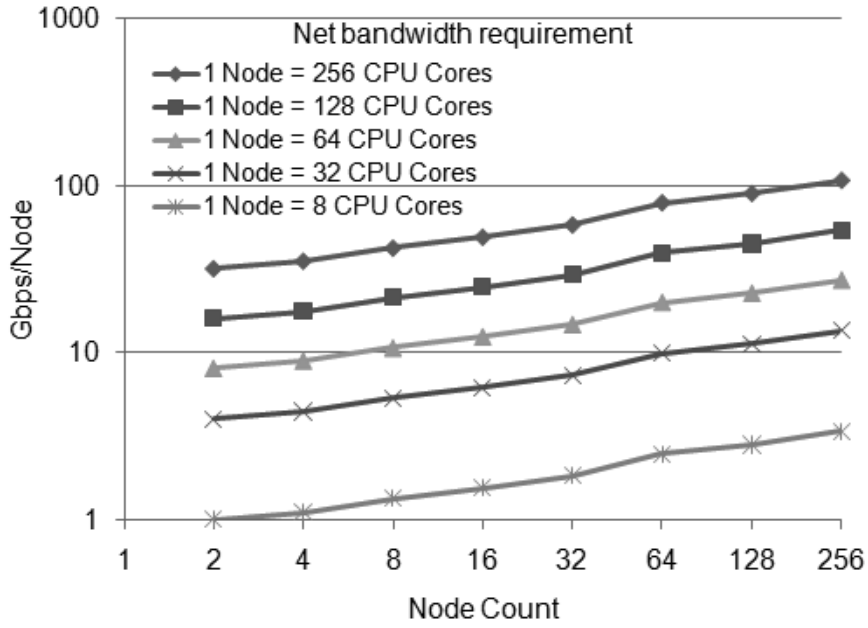


Figure 7·5: Bandwidth requirement for various systems for a 100K particle problem size. Systems are ideal with all-to-all interconnect and no in-channel particle filtering

node counts. That is, at low node counts, the observed inter-node bandwidth requirement will be significantly lower than what we present here.

We now examine the communication at the packet level to determine whether latency (time-of-flight) is likely to have an impact on channel provisioning. That is, we determine whether packet count must be considered (in this preliminary study) or bandwidth alone is adequate. A packet for range-limited computation consists of all position/charge/force data of all particles in a cell. With an 18$\mathring{A}$ cell and a particle density of 0.1 particles per cubic angstrom, the average number of particles in a cell is approximately 600. With double-precision floating point, the position/charge data amount is 32 Bytes per particle, while force data amount is 24 Bytes per particle giving an average packet size of 16 KB. With a 14 Gbps transceiver, this corresponds to 8.72 microseconds per data packet. This time is significantly larger than what we assume for inter-node data latency (a few hundred nanoseconds). An efficient

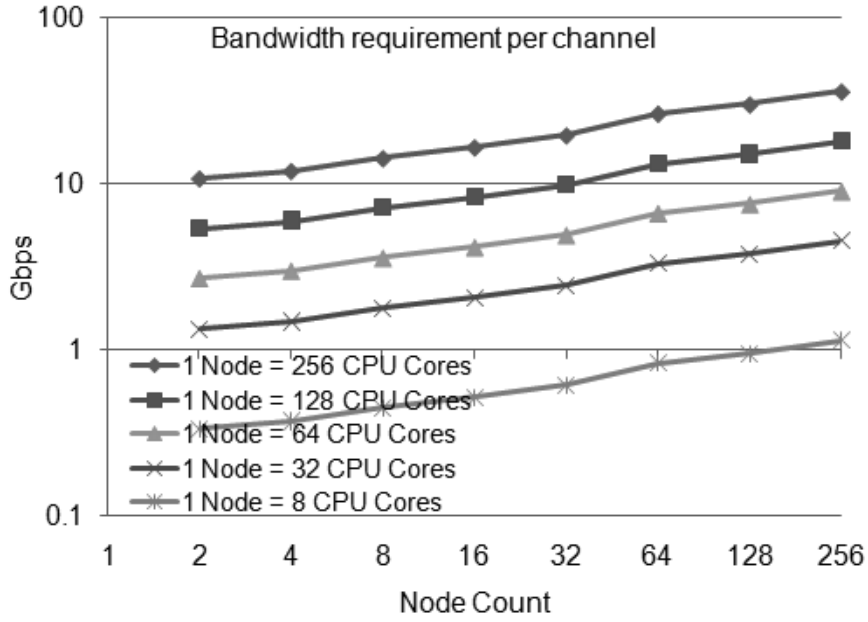


Figure 7·6: Bandwidth per channel requirement for various systems for a 100K problem size. Some likely system information is integrated such as number of hops per packet and in-channel particle filtering

implementation is likely to be able to hide this latency. Similar arguments hold for grid interpolation and FFT/transpose. Therefore, for further discussion, we only consider bandwidth.

Next we determine the actual bandwidth requirement, assuming a 3D bi-directional torus network. Our goal here is to determine bandwidth needed for each of the 12 channels on a node in such a network topology. Data communication for range-limited non-bonded force computation is contained within neighboring nodes at 1-3 hops. This will on average cause about 2 times increase in data communication. At the same time, however, the FPGA easily supports in-channel filtering to remove particles not needed by a particular neighbor. For patch and cut-off sizes described earlier, this results in a reduction of data to be transferred (weighted by number of hops) to 73% of the original. For long-range communication, all-to-all communication is required which roughly doubles the data

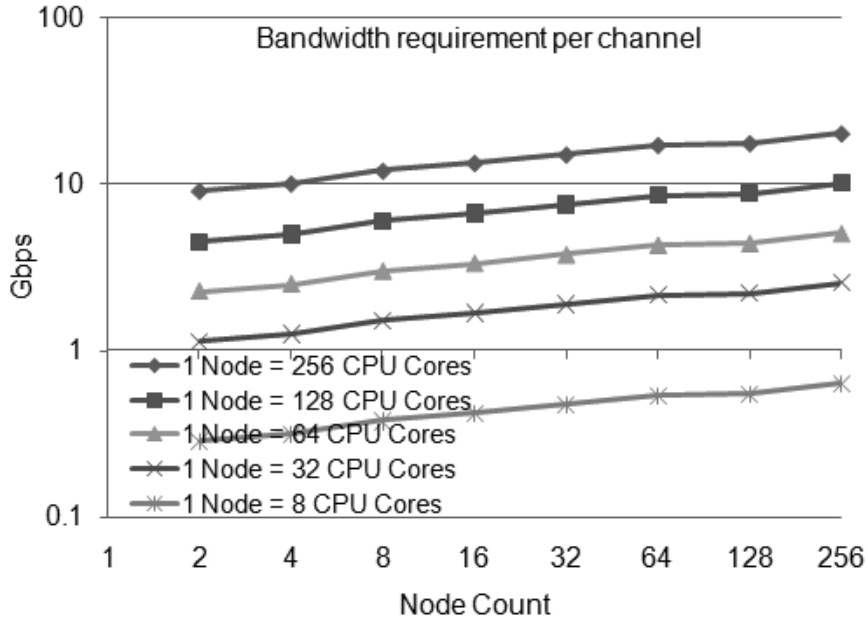


Figure 7·7: Bandwidth per channel requirement for various systems for a problem size of 1 million. Some likely system information is integrated such as number of hops per packet and in-channel particle filtering

amount for a $4 \times 4 \times 4$ node system and further doubles it on an $8 \times 8 \times 8$ node system.

The final bandwidth requirement for a 100K simulation is shown in Figure 7·6. The series `node = 256 cores` represents projected performance of a 4 FPGA node. For a system with 64 such nodes, configured in a 3D bidirectional torus, each channel must support 27 Gbps bandwidth; this is possible with 2 14 Gbps serial links. The aggregate of 24 links is a small fraction of the 176 available among the four FPGAs on such a node, assuming the appropriate Altera Stratix V FPGA.

Figure 7·7 shows the same results for a 1 million particle simulation. As expected, this significantly reduces the communication requirement. Again examining the series `node = 256 cores` we see that the last three points (64, 128, and 256 nodes) require bandwidths of 17.2, 17.7, and 20.3 Gbps, respectively. This approaches what can be achieved with a single serial port, especially in the next generation of FPGAs where

the link capacity is likely to double.

7.5 Chapter Summary

In this chapter, we have performed an initial study on communication network requirements for an FPGA-centric cluster executing molecular dynamics simulations. We find that current production boards (with 4 FPGAs) are appropriate for use as nodes in such a system. Even for relatively small simulations which are hard to scale to large clusters (<100K particles) we find that only a small fraction of the FPGA's communication capability is required. Some of this is due to the fact that the FPGA channel is programmable which can significantly reduce the amount of data that needs to be transferred. We conclude that a cluster with 256 FPGAs distributed in 64 nodes is appropriately provisioned, even for modest simulations, with a bidirectional 3D torus where each link consists of 1-2 of an FPGA's serial ports. This study should serve as a baseline in the creation of a scalable FPGA-accelerated MD solution.

Chapter 8

Conclusions

We conclude this dissertation by summarizing our work on parallelization of DMD and integration of FPGA-accelerated MD into full-parallel production MD package. We also discuss some of the lessons we learned from this study and how this work can be extended in the future.

8.1 Summary

In this research we have worked towards achieving fast yet scalable molecular dynamics simulation using FPGAs and multicore processors. We enhanced the performance of an existing FPGA kernel for range-limited non-bonded force computation, and then incorporated it into a full-parallel production-level MD package. We also parallelized DMD, an emerging alternative to timestep-driven MD, by taking advantage of the shared memory architecture of multicore processors.

In our PDMD work, we first systematically studied the difficulties in parallelizing DMD and defined three possible kinds of hazards. To circumvent these difficulties, we used event-based decomposition, as opposed to the existing approaches that use spatial decomposition. In our method, multiple events in DMD are processed in parallel, but committed in serial. We utilized a recently developed data structure and the shared-memory architecture of multicore processors to minimize the serial commitment and other synchronization time. We presented three possible implementations of our approach and found that, having a dedicated

helper thread and multiple worker threads resulted in the best performance scaling. This implementation achieved more than 5.5x (8.5x) speed-up for 3D systems on an 8 (12) core CPU. We analyzed our results in terms of available concurrency in the application and architectural features of the hardware.

In our work on FPGA-accelerated MD, we first enhanced the performance of an existing FPGA kernel by taking advantage of the BRAM architecture of the FPGAs. We explored the design space of the table interpolation method and found that, we can trade-off the granularity of the table with the order of interpolation without sacrificing simulation quality. Unlike in CPU-based systems, the BRAM architecture in the FPGAs allows us to have finer-grained tables and lower order of interpolation. This saves logic resources on the FPGA, which are then used to implement more force computation pipeline. The net result is a 50% improvement in performance, compared to direct computation.

Next, we parallelized the FPGA kernel to utilize multiple on-board FPGAs. We used partitioning and software pipelining methods to achieve a 3.37x end-to-end speed-up over a single CPU core, using the Gidel FPGA board. We discussed the serialization issues and determined the theoretical upper bound of speed-up to be 3.76x, when only the range-limited non-bonded force computation is accelerated and long-range portion of PME is computed every timestep.

Then we integrated the FPGA kernel into NAMD, a widely used full-parallel production MD package. We discussed how the scale and complexity of a production-level package make efficient integration a very challenging task. We described how we actually integrated the FPGA-kernel, and how the integrated framework can be used in architectural studies. Our integrated version achieved 2.22x end-to-end speed-up over a quad-core CPU, making it the first ever full-parallel production-level FPGA-accelerated MD to achieve a positive end-to-end speed-up.

Finally, we studied the plausibility of using FPGA-centric clusters for MD. We instrumented our framework to quantify the data communication characteristics of MD. We applied this information to clusters of various sizes and node complexity, and found that a cluster with 256 FPGAs distributed in 64 nodes is appropriately provisioned, even for modest simulations, with a bidirectional 3D torus where each link consists of 1-2 of an FPGA's serial ports.

8.2 Observations

Need to know both the application and the platform it runs on: In this work, we found that knowing both the hardware and the software is absolutely essential in achieving high performance. The design space to explore not only consists of choices in the application, but also includes the features of the available hardware platforms. A proper understanding is, therefore, required to determine how an application can be best mapped on a target hardware. We see an example of this in our PDMD work, where we determined that spatial decomposition is likely to have too much data communication, making it a bad choice for parallelization. We rather chose a method that circumvents this issue by taking advantage of the shared memory architecture of our target hardware, multicore processors. Another example is how we took advantage of the BRAM architecture of the FPGAs to implement finer-grained tables, a choice not likely to be favorable on CPUs, for table interpolation method.

Implementation-level details matter: From our experience in the PDMD work, we find that implementation-level details have serious consequence in the final performance. We had three implementations of the same task-decomposed approach, where only the synchronization methods were different. The achieved performance varied significantly, ranging from slow-down to near-linear speed-up.

Data communication is the key in parallel applications: Our integration work on the FPGA-accelerated MD shows that correctly handling data communication is the key in parallel applications. Much of the benefits of acceleration can be diminished if the accelerator incurs additional data communication. Efforts need to be made not only to minimize data communication between the host and the accelerator, but also to minimize data communication among host processors.

Striking the right balance is more important than making individual pieces optimal: By using our integrated framework for FPGA-accelerated MD, we can see that, by overlapping accelerator runtime with CPU computation and communication time, a relatively slower accelerator can achieve the same or even better performance than a faster accelerator. The available features of the accelerator, e.g. the ability to split force computations for remote and local patches, is no less important than the throughput of its force computation pipeline.

Top-down approach is necessary to achieve end-to-end speed-up using FPGAs: While many previous work on FPGA-accelerated MD used simplified software to prove a concept, the methods developed using such simplified software are sometimes in conflict with the structure of the highly optimized production-level software. This ends up requiring significant modifications in the software, diminishing the advantages of acceleration. To achieve meaningful end-to-end speed-ups, it is crucial to take a top-down approach, where the accelerators will incorporate the needs of the software, not the other way around.

8.3 Future Directions

8.3.1 Hardware Implementation of Task-decomposed DMD

A hardware implementation of our task-decomposed PDMD is highly promising as a future work. Considering the non-recurring cost of ASICs, FPGAs seem to be the most viable option. GPUs may also be a good choice, but in that case, functional decomposition (described in Section 3.1.2) may be a better choice, since GPUs are especially good at running many tiny identical threads.

8.3.2 FPGAs for Data Communication of MD

While CPU-only MD remains compute-bound for at least a few hundred compute nodes, that is not the case for accelerated versions. Communication among compute nodes become a bottleneck even for small systems. The need for fast data communication is especially crucial in evaluating the long-range portion of electrostatic force, which is often based on 3D FFT, and requires all-to-all communication during a timestep. Without substantial improvement in such inter-node communication, FPGA-acceleration will be limited to only a few times of speed-up. This presents a highly promising area of research where FPGAs can be used directly for communication between compute nodes. FPGAs are already used in network routers and seem like a natural fit for this purpose [28]. As shown in Chapter 7, our integrated framework for FPGA-accelerated MD can provide useful insights in making crucial design choices.

8.3.3 FPGA-centric MD Engine

As Moore's law [115] continues, FPGAs are equipped with more functionality than ever. It is possible to have embedded processors on FPGAs [5, 177], either soft or hard, which makes it feasible to create an entirely FPGA-centric MD engine. In such

an engine, overall control and simple software tasks will be done on the embedded processors while the heavy work, like the non-bonded force computations, will be implemented on the remaining logic. Data communication can also be performed using the FPGAs, completely eliminating general purpose CPUs from the scene. Such a system is likely to be highly efficient, both in terms of computational performance and energy consumption. We already showed the plausibility of such a system in Chapter 7. This work can be extended to actually create such a system and implement an FPGA-centric MD engine.

8.3.4 Broader Application

Many of the methods developed in this work can be generalized and extended to other scientific problems. In fact, the task-decomposed method used in parallelizing DMD has already been found useful in achieving better energy efficiency and thermal behavior for a parallel workload, called DeDup (Data Deduplication), in the PARSEC benchmark suite [88, 89]. The integrated framework of the FPGA-accelerated MD can be used to study quantitatively various design trade-offs in future designs. For example, we have instrumented the code to extract data communication pattern which can potentially be used to study performance and energy efficiency of a system where certain data communication can be reduced at the cost of additional local filtering computation.

References

[1] Accelera (2012). Accelerating bio-molecular simulations. Accelera website. http://www.acellera.com/acemd/.

[2] Adcock, S. A. and McCammon, J. A. (2006). Molecular dynamics: Survey of methods for simulating the activity of proteins. *Chemical Reviews*, 106(5):1589–1615. http://dx.doi.org/10.1021/cr040426m.

[3] Allen, M. P. (2004). Introduction to molecular dynamics simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins, Lecture Notes, NIC Series*, 23:1–28.

[4] Altera (2006). FPGA architecture. http://www.altera.com/literature/wp/wp-01003.pdf.

[5] Altera (2012). Altera website. http://www.altera.com.

[6] AMBER (2011). NVIDIA GPU acceleration support. http://ambermd.org/gpus/benchmarks.htm.

[7] AMBER (2012). AMBER on Tesla GPUs. http://www.nvidia.com/object/amber_on_tesla.html.

[8] Amisaki, T., Fujiwara, T., Kusumi, A., Miyagawa, H., and Kitamura, K. (1995). Error evaluation in the design of a special-purpose processor that calculates nonbonded forces in molecular dynamics simulations. *Journal of Computational Chemistry*, 16(9):1120–1130. http://dx.doi.org/10.1002/jcc.540160906.

[9] Anderson, J. A., Lorenz, C. D., and Travesset, A. (2008). General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359. http://dx.doi.org/10.1016/j.jcp.2008.01.047.

[10] Annapolis (2003). *WILDSTAR-II Hardware Reference Manual*. Annapolis Micro Systems Inc., USA.

[11] Annapolis (2006). *WILDSTAR II PRO for PCI*. Annapolis Micro Systems Inc., USA.

[12] Azizi, N., Kuon, I., Egier, A., Darabiha, A., and Chow, P. (2004). Reconfigurable molecular dynamics simulator. In *The 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 197–206. http://dx.doi.org/10.1109/FCCM.2004.48.

[13] Baker, Z., Bhattacharya, T., Dunham, M., Graham, P., Gupta, R., Inman, J., Klein, A., Kunde, G., McPherson, A., Stettler, M., and Tripp, J. (2009). The PetaFlops Router: Harnessing FPGAs and accelerators for high performance computing. In *Proceedings of High Performance Embedded Computing (HPEC)*, pages 1–3.

[14] Bargiel, M., Dzwinel, W., Kitowski, J., and Moscinski, J. (1991). C-language molecular dynamics program for the simulation of Lennard-Jones particles. *Computer Physics Communications*, 64(1):193–205. http://dx.doi.org/10.1016/0010-4655(91)90061-O.

[15] Baxter, R., Booth, S., Bull, M., Cawood, G., Perry, J., Parsons, M., Simpson, A., Trew, A., McCormick, A., Smart, G., Smart, R., Cantle, A., Chamberlain, R., and Genest, G. (2007). Maxwell - a 64 FPGA supercomputer. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 287–294. http://dx.doi.org/10.1109/AHS.2007.71.

[16] BEEcube (2012). BEEcube website. http://www.beecube.com.

[17] Berendsen, H., van der Spoel, D., and van Drunen, R. (1995). GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1-3):43–56. http://dx.doi.org/10.1016/0010-4655(95)00042-E.

[18] BittWare (2012). BittWare website. http://www.bittware.com.

[19] Board, J. A. J., Humphres, C. W., Lambert, C. G., Rankin, W. T., and Toukmaji, A. Y. (1997). Ewald and multipole methods for periodic N-body problems. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (PPSC)*, pages 1–8.

[20] Bowers, K. J., Chow, E., Xu, H., Dror, R. O., Eastwood, M. P., Gregersen, B. A., Klepeis, J. L., Kolossvary, I., Moraes, M. A., Sacerdoti, F. D., Salmon, J. K., Shan, Y., and Shaw, D. E. (2006). Scalable algorithms for molecular dynamics simulations on commodity clusters. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, pages 84:1–84:13. http://dx.doi.org/10.1145/1188455.1188544.

[21] Bowers, K. J., Dror, R. O., and Shaw, D. E. (2007). Zonal methods for the parallel execution of range-limited N-body simulations. *Journal of*

Computational Physics, 221(1):303–329. http://dx.doi.org/10.1016/j.jcp.2006.06.014.

[22] Bowers, K. J., Lippert, R. A., Dror, R. O., and Shaw, D. E. (2010). Improved twiddle access for Fast Fourier Transforms. *Transaction on Signal Processing*, 58(3):1122–1130. http://dx.doi.org/10.1109/TSP.2009.2035984.

[23] Brandt, A. (1977). Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390. http://dx.doi.org/10.1090/S0025-5718-1977-0431719-X.

[24] Brooks, B. R., Brooks, III, C. L., Mackerell, Jr., A. D., Nilsson, L., Petrella, R. J., Roux, B., Won, Y., Archontis, G., Bartels, C., Boresch, S., Caflisch, A., Caves, L., Cui, Q., Dinner, A. R., Feig, M., Fischer, S., Gao, J., Hodoscek, M., Im, W., Kuczera, K., Lazaridis, T., Ma, J., Ovchinnikov, V., Paci, E., Pastor, R. W., Post, C. B., Pu, J. Z., Schaefer, M., Tidor, B., Venable, R. M., Woodcock, H. L., Wu, X., Yang, W., York, D. M., and Karplus, M. (2009). CHARMM: The biomolecular simulation program. *Journal of Computation Chemistry*, 30(10):1545–1614. http://dx.doi.org/10.1002/jcc.21287.

[25] Brooks, C. and Case, D. A. (1993). Simulations of peptide conformational dynamics and thermodynamics. *Chemical Reviews*, 93(7):2487–2502. http://dx.doi.org/10.1021/cr00023a008.

[26] Brown, W. M. (2011). GPU acceleration in LAMMPS. http://lammps.sandia.gov/workshops/Aug11/Brown/brown11.pdf.

[27] Buldyrev, S. V. (2008). Application of discrete molecular dynamics to protein folding and aggregation. In *Aspects of Physical Biology*, volume 752 of *Lecture Notes in Physics*, pages 97–131. Springer-Verlag. http://dx.doi.org/10.1007/978-3-540-78765-5_5.

[28] Byrne, J., Bolaria, J., and Halfhill, T. R. (2011). A guide to FPGAs. Technical report, The Linley Group.

[29] Case, D., Darden, T., Cheatham, T., Simmerling, C., Wang, J., Duke, R., Luo, R., Walker, R., Zhang, W., Merz, K., Roberts, B., Wang, B., Hayik, S., Roitberg, A., Seabra, G., Kolossvry, I., Wong, K., Paesani, F., Vanicek, J., Liu, J., Wu, X., Brozell, S., Steinbrecher, T., Gohlke, H., Cai, Q., Ye, X., Wang, J., Hsieh, M., Cui, G., Roe, D., Mathews, D., Seetin, M., Sagui, C., Babin, V., Luchko, T., Gusarov, S., Kovalenko, A., and Kollman, P. (2010). AMBER 11. University of California, San Francisco.

[30] Case, D. A., Cheatham, T. E., Darden, T., Gohlke, H., Luo, R., Jr., K. M. M., Onufriev, A., Simmerling, C., Wang, B., and Woods, R. J. (2005). The

Amber biomolecular simulation programs. *Journal of Computational Chemistry*, 26(16):1668–1688. http://dx.doi.org/10.1002/jcc.20290.

[31] Chellappa, S., Franchetti, F., and Püschel, M. (2008). How to write fast numerical code: A small introduction. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 196–259. Springer-Verlag. http://dx.doi.org/10.1007/978-3-540-88643-3_5.

[32] Chiu, M. and Herbordt, M. C. (2009). Efficient particle-pair filtering for acceleration of molecular dynamics simulation. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 345–352. http://dx.doi.org/10.1109/FPL.2009.5272272.

[33] Chiu, M. and Herbordt, M. C. (2010a). Molecular dynamics simulations on high-performance reconfigurable computing systems. *ACM Transaction on Reconfigurable Technology and Systems (TRETS)*, 3(4):23:1–23:37. http://dx.doi.org/10.1145/1862648.1862653.

[34] Chiu, M. and Herbordt, M. C. (2010b). Towards production FPGA-accelerated molecular dynamics: Progress and challenges. In *Fourth International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)*, pages 1–8. http://dx.doi.org/10.1109/HPRCTA.2010.5670800.

[35] Chiu, M., Herbordt, M. C., and Langhammer, M. (2008). Performance potential of molecular dynamics simulations on high performance reconfigurable computing systems. In *Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)*, pages 1–10. http://dx.doi.org/10.1109/HPRCTA.2008.4745685.

[36] Chiu, M., Khan, M. A., and Herbordt, M. C. (2011). Efficient calculation of pairwise nonbonded forces. In *The 19th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 73–76. http://dx.doi.org/10.1109/FCCM.2011.34.

[37] Chiu, S. (2011). Accelerating molecular dynamics simulations with high-performance reconfigurable systems. PhD dissertation, Boston University, USA.

[38] Cho, E., Bourgeois, A., and Fernndez-Zepeda, J. (2008). Examining the feasibility of reconfigurable models for molecular dynamics simulation. In *Algorithms and Architectures for Parallel Processing*, volume 5022 of *Lecture*

Notes in Computer Science, pages 109–120. Springer-Verlag. http://dx.doi.org/10.1007/978-3-540-69501-1_13.

[39] Cho, E., Bourgeois, A., and Tan, F. (2007). An FPGA design to achieve fast and accurate results for molecular dynamics simulations. In *Parallel and Distributed Processing and Applications*, volume 4742 of *Lecture Notes in Computer Science*, pages 256–267. Springer-Verlag. http://dx.doi.org/10.1007/978-3-540-74742-0_25.

[40] Cisco (2012). Cisco website. http://www.cisco.com.

[41] Culler, D., Singh, J. P., and Gupta, A. (1998). *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition.

[42] Dally, W. J. (2010). Throughput computing. Keynote Talk, ACM/IEEE Conference on Supercomputing (SC).

[43] Dally, W. J., Fiske, J. S., Keen, J. S., Lethin, R. A., Noakes, M. D., Nuth, P. R., Davison, R. E., and Fyler, G. A. (1992). The Message-Driven Processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39. http://dx.doi.org/10.1109/40.127581.

[44] Darden, T., York, D., and Pedersen, L. (1993). Particle Mesh Ewald: An N.log (N) method for Ewald sums in large systems. *Journal of Chemical Physics*, 98(12):10089–10092. http://dx.doi.org/10.1063/1.464397.

[45] DeMarco, M. L. and Daggett, V. (2004). From conversion to aggregation: Protofibril formation of the prion protein. *Proceedings of the National Academy of Sciences of the United States of America*, 101(8):2293–2298. http://dx.doi.org/10.1073/pnas.0307178101.

[46] Dokholyan, N. V. (2006). Studies of folding and misfolding using simplified models. *Current Opinion in Structural Biology*, 16(1):79–85. http://dx.doi.org/10.1016/j.sbi.2006.01.001.

[47] Dror, R., Grossman, J., Mackenzie, K., Towles, B., Chow, E., Salmon, J., Young, C., Bank, J., Batson, B., Deneroff, M., Kuskin, J., Larson, R., Moraes, M., and Shaw, D. (2011). Overcoming communication latency barriers in massively parallel scientific computation. *IEEE Micro*, 31(3):8–19. http://dx.doi.org/10.1109/MM.2011.38.

[48] Dror, R. O., Grossman, J. P., Mackenzie, K. M., Towles, B., Chow, E., Salmon, J. K., Young, C., Bank, J. A., Batson, B., Deneroff, M. M., Kuskin, J. S., Larson, R. H., Moraes, M. A., and Shaw, D. E. (2010). Exploiting 162-nanosecond end-to-end communication latency on Anton. In *Proceedings*

of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pages 1–12. http://dx.doi.org/10.1109/SC.2010.23.

[49] Eaton, W. A., Muoz, V., Thompson, P. A., Chan, C.-K., and Hofrichter, J. (1997). Submillisecond kinetics of protein folding. *Current Opinion in Structural Biology*, 7(1):10–14. http://dx.doi.org/10.1016/S0959-440X(97)80003-6.

[50] Ebisuzaki, T., Makino, J., Fukushige, T., Taiji, M., Sugimoto, D., Ito, T., and Okumura, S. K. (1993). GRAPE project: an overview. *Publications of the Astronomical Society of Japan*, 45:269–278.

[51] Engle, R. D., Skeel, R. D., and Drees, M. (2005). Monitoring energy drift with shadow Hamiltonians. *Journal of Computational Physics*, 206(2):432–452. http://dx.doi.org/10.1016/j.jcp.2004.12.009.

[52] Essmann, U., Perera, L., Berkowitz, M. L., Darden, T., Lee, H., and Pedersen, L. (1995). A smooth particle mesh Ewald method. *Journal of Chemical Physics*, 103(19):8577–8593. http://dx.doi.org/10.1063/1.470117.

[53] Ewald, P. P. (1921). Die berechnung optischer und elektrostatischer gitterpotentiale. *Annalen der Physik*, 369(3):253–287. http://dx.doi.org/10.1002/andp.19213690304.

[54] Flower, D. R., Phadwal, K., Macdonald, I. K., Coveney, P. V., Davies, M. N., and Wan, S. (2010). T-cell epitope prediction and immune complex simulation using molecular dynamics: state of the art and persisting challenges. *Immunome Research*, 6(Suppl 2):S4. http://dx.doi.org/10.1186/1745-7580-6-S2-S4.

[55] Freddolino, P. L., Arkhipov, A. S., Larson, S. B., McPherson, A., and Schulten, K. (2006). Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure*, 14(3):437–449. http://dx.doi.org/10.1016/j.str.2005.11.014.

[56] Fujimoto, R. M. (1990). Parallel discrete event simulation. *Communications of the ACM - Special issue on simulation*, 33(10):30–53. http://dx.doi.org/10.1145/84537.84545.

[57] Fukushige, T., Taiji, M., Makino, J., Ebisuzaki, T., and Sugimoto, D. (1996). A highly parallelized special-purpose computer for many-body simulations with an arbitrary central force: MD-GRAPE. *Astrophysical Journal*, 468:51–61. http://dx.doi.org/10.1086/177668.

[58] George, A., Lam, H., and Stitt, G. (2011). Novo-G: At the forefront of scalable reconfigurable supercomputing. *Computing in Science and Engineering*, 13(1):82–86. http://dx.doi.org/10.1109/MCSE.2011.11.

[59] Gidel (2009a). Gidel PROCStarIII data book. Gidel Ltd.

[60] Gidel (2009b). Gidel website. http://www.gidel.com.

[61] Gidel (2009c). PROCWizard user's manual. Gidel Ltd.

[62] GROMACS (2012). GROMACS installation instructions for GPUs. http://www.gromacs.org/Downloads/Installation_Instructions/GPUs.

[63] Gu, Y. (2008). FPGA acceleration of molecular dynamics simulations. PhD dissertation, Boston University, USA.

[64] Gu, Y. and Herbordt, M. C. (2007a). Amenability of multigrid computations to FPGA-based acceleration. In *Proceedings of High Performance Embedded Computing (HPEC)*, pages 1–2.

[65] Gu, Y. and Herbordt, M. C. (2007b). FPGA-based multigrid computation for molecular dynamics simulations. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 117–126. http://dx.doi.org/10.1109/FCCM.2007.42.

[66] Gu, Y., Vancourt, T., and Herbordt, M. C. (2005). Accelerating molecular dynamics simulations with configurable circuits. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 475–480. http://dx.doi.org/10.1109/FPL.2005.1515767.

[67] Gu, Y., Vancourt, T., and Herbordt, M. C. (2006a). Accelerating molecular dynamics simulations with configurable circuits. *IEE Proceedings - Computers and Digital Techniques*, 153(3):189–195. http://dx.doi.org/10.1049/ip-cdt:20050182.

[68] Gu, Y., Vancourt, T., and Herbordt, M. C. (2006b). Improved interpolation and system integration for FPGA-based molecular dynamics simulations. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. http://dx.doi.org/10.1109/FPL.2006.311190.

[69] Gu, Y., Vancourt, T., and Herbordt, M. C. (2006c). Integrating FPGA acceleration into the Protomol molecular dynamics code: Preliminary report. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 315–316. http://dx.doi.org/10.1109/FCCM.2006.52.

[70] Gu, Y., Vancourt, T., and Herbordt, M. C. (2008). Explicit design of FPGA-based coprocessors for short-range force computations in molecular dynamics simulations. *Parallel Computing*, 34(4-5):261–277. http://dx.doi.org/10.1016/j.parco.2008.01.007.

[71] Guo, H., Su, L., Wang, Y., and Long, Z. (2009). FPGA-accelerated molecular dynamics simulations system. In *International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing*, pages 360–365. http://dx.doi.org/10.1109/EmbeddedCom-ScalCom.2009.71.

[72] Hagen, S. J., Hofrichter, J., Szabo, A., and Eaton, W. A. (1996). Diffusion-limited contact formation in unfolded cytochrome c: estimating the maximum rate of protein folding. *Proceedings of the National Academy of Science of the United States of America*, 93(21):11615–11617. http://dx.doi.org/10.1073/pnas.93.21.11615.

[73] Hardy, D. J. (2007). NAMD-Lite. http://www.ks.uiuc.edu/Development/MDTools/namdlite/. University of Illinois at Urbana-Champaign.

[74] Harvey, M. J. and De Fabritiis, G. (2009). An implementation of the Smooth Particle Mesh Ewald method on GPU hardware. *Journal of Chemical Theory and Computation*, 5(9):2371–2377. http://dx.doi.org/10.1021/ct900275y.

[75] Harvey, M. J., Giupponi, G., and Fabritiis, G. D. (2009). ACEMD: Accelerating biomolecular dynamics in the microsecond time scale. *Journal of Chemical Theory and Computation*, 5(6):1632–1639. http://dx.doi.org/10.1021/ct9000685.

[76] Hennessy, J. L. and Patterson, D. A. (2006). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition.

[77] Herbordt, M., Kosie, F., and Model, J. (2008). An efficient O(1) priority queue for large FPGA-based discrete event simulations of molecular dynamics. In *The 16th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 248–257. http://dx.doi.org/10.1109/FCCM.2008.49.

[78] Herbordt, M. C., Khan, M. A., and Dean, T. (2009). Parallel discrete event simulation of molecular dynamics through event-based decomposition. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 129–136. http://dx.doi.org/10.1109/ASAP.2009.39.

[79] Hess, B., Kutzner, C., van der Spoel, D., and Lindahl, E. (2008). GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation*, 4(3):435–447. http://dx.doi.org/10.1021/ct700301q.

[80] Hockney, R., Goel, S., and Eastwood, J. (1974). Quiet high-resolution computer models of a plasma. *Journal of Computational Physics*, 14(2):148–158. http://dx.doi.org/10.1016/0021-9991(74)90010-2.

[81] Ito, T., Makino, J., Fukushige, T., Ebisuzaki, T., Okumura, S. K., and Sugimoto, D. (1993). A special-purpose computer for N-body simulations: GRAPE-2A. *Publications of the Astronomical Society of Japan*, 45:339–347.

[82] Kalé, L. and Krishnan, S. (1993). CHARM++: A portable concurrent object oriented system based on C++. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 91–108. http://dx.doi.org/10.1145/167962.165874.

[83] Kalé, L., Skeel, R., Bhandarkar, M., Brunner, R., Gursoy, A., Krawetz, N., Phillips, J., Shinozaki, A., Varadarajan, K., and Schulten, K. (1999). NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312. http://dx.doi.org/10.1006/jcph.1999.6201.

[84] Kalé, L. V., Zheng, G., Lee, C. W., and Kumar, S. (2006). Scaling applications to massively parallel machines using projections performance analysis tool. *Future Generation Computer Systems*, 22(3):347–358. http://dx.doi.org/10.1016/j.future.2004.11.020.

[85] Karplus, M. and McCammon, J. A. (2002). Molecular dynamics simulations of biomolecules. *Nature Structural Biology*, 9(9):646–652. http://dx.doi.org/10.1038/nsb0902-646.

[86] Kasap, S. and Benkrid, K. (2011). A high performance implementation for molecular dynamics simulations on a FPGA supercomputer. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 375–382. http://dx.doi.org/10.1109/AHS.2011.5963962.

[87] Khalili-Araghi, F., Tajkhorshid, E., and Schulten, K. (2006). Dynamics of K+ ion conduction through Kv1.2. *Biophysical Journal*, 91(6):72–76. http://dx.doi.org/10.1529/biophysj.106.091926.

[88] Khan, M. A., Hankendi, C., Coskun, A. K., and Herbordt, M. C. (2011a). Application level optimizations for energy efficiency and thermal stability. In *Proceedings of High Performance Embedded Computing (HPEC)*, pages 1–2.

[89] Khan, M. A., Hankendi, C., Coskun, A. K., and Herbordt, M. C. (2011b). Software optimization for performance, energy, and thermal distribution: Initial case studies. In *International Green Computing Conference and Workshops (IGCC)*, pages 1–6. http://dx.doi.org/10.1109/IGCC.2011.6008575.

[90] Khan, M. A. and Herbordt, M. C. (2011). Parallel discrete molecular dynamics simulation with speculation and in-order commitment. *Journal of Computational Physics*, 230(17):6563–6582. http://dx.doi.org/10.1016/j.jcp.2011.05.001.

[91] Kindratenko, V. and Pointer, D. (2006). A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 13–22. http://dx.doi.org/10.1109/FCCM.2006.5.

[92] Komeiji, Y., Uebayasi, M., Takata, R., Shimizu, A., Itsukashi, K., and Taiji, M. (1997). Fast and accurate molecular dynamics simulation of a protein using a special-purpose computer. *Journal of Computational Chemistry*, 18(12):1546–1563. http://dx.doi.org/10.1002/(SICI)1096-987X(199709)18:12%3C1546::AID-JCC11%3E3.0.CO;2-I.

[93] Krantz, A. T. (1996). Analysis of an efficient algorithm for the hard-sphere problem. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 6(3):185–209. http://dx.doi.org/10.1145/235025.235030.

[94] Kumar, S., Huang, C., Zheng, G., Bohm, E., Bhatele, A., Phillips, J. C., Yu, H., and Kalé, L. V. (2008). Scalable molecular dynamics with NAMD on the IBM Blue Gene/L system. *IBM Journal of Research and Development*, 52(1-2):177–188. http://dx.doi.org/10.1147/rd.521.0177.

[95] Kuon, I., Azizi, N., Darabiha, A., Egier, A., and Chow, P. (2004). FPGA-based supercomputing: an implementation for molecular dynamics. In *ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 253–253. http://dx.doi.org/10.1145/968280.968340.

[96] LAMMPS (2012a). LAMMPS GPU benchmarks. http://users.nccs.gov/~wb8/gpu/bench.htm.

[97] LAMMPS (2012b). LAMMPS molecular dynamics simulator. http://lammps.sandia.gov.

[98] Langhammer, M. (2008). Floating point datapath synthesis for FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 355–360. http://dx.doi.org/10.1109/FPL.2008.4629963.

[99] Larson, R., Salmon, J., Dror, R., Deneroff, M., Young, C., Grossman, J., Shan, Y., Klepeis, J., and Shaw, D. (2008). High-throughput pairwise point interactions in Anton, a specialized machine for molecular dynamics simulation. In *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342. http://dx.doi.org/10.1109/HPCA.2008.4658650.

[100] Lee, S. (2005). An FPGA implementation of the Smooth Particle Mesh Ewald reciprocal sum compute engine. Master's thesis, The University of Toronto, Canada.

[101] Lindahl, E., Hess, B., and Van Der Spoel, D. (2001). GROMACS 3.0: a package for molecular simulation and trajectory analysis. *Journal of Molecular Modeling*, 7(8):306–317. http://dx.doi.org/10.1007/s008940100045.

[102] Lubachevsky, B. D. (1991). How to simulate billiards and similar systems. *Journal of Computational Physics*, 94(2):255–283. http://dx.doi.org/10.1016/0021-9991(91)90222-7.

[103] Lubachevsky, B. D. (1992). Simulating billiards: Serially and in parallel. *International Journal in Computer Simulation (IJCS)*, 2:373–411.

[104] MacKerell, A. D., Banavali, N., and Foloppe, N. (2000). Development and current status of the CHARMM force field for nucleic acids. *Biopolymers*, 56(4):257–265. http://dx.doi.org/10.1002/1097-0282(2000)56:4%3C257::AID-BIP10029%3E3.0.CO;2-W.

[105] Makov, G. and Payne, M. C. (1995). Periodic boundary conditions in *ab initio* calculations. *Physical Review B*, 51(7):4014–4022. http://dx.doi.org/10.1103/PhysRevB.51.4014.

[106] Malladi, R. K. (2009). Using Intel VTune performance analyzer events / ratios & optimizing applications. http://software.intel.com/en-us/articles/.

[107] Marin, M. (1997). Billiards and related systems on the bulk-synchronous parallel model. In *Proceedings of the Eleventh Workshop on Parallel and Distributed Simulation (PADS)*, pages 164–171. http://dx.doi.org/10.1145/268823.268916.

[108] Marin, M. and Cordero, P. (1995). An empirical assessment of priority queues in event-driven molecular dynamics simulation. *Computer Physics Communications*, 92(23):214–224. http://dx.doi.org/10.1016/0010-4655(95)00120-2.

[109] Marin, M., Risso, D., and Cordero, P. (1993). Efficient algorithms for many-body hard particle molecular dynamics. *Journal of Computational Physics*, 109(2):306–317. http://dx.doi.org/10.1006/jcph.1993.1219.

[110] Matthey, T., Cickovski, T., Hampton, S., Ko, A., Ma, Q., Nyerges, M., Raeder, T., Slabach, T., and Izaguirre, J. A. (2004). Protomol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Transactions on Mathematical Software*, 30(3):237–265. http://dx.doi.org/10.1145/1024074.1024075.

[111] MD3DLJ (1989). A pc/workstation c - language program for L-J molecular dynamics. ftp://ftp.dl.ac.uk/ccp5/MD3DLJ.C.

[112] Mei, C., Sun, Y., Zheng, G., Bohm, E. J., Kalé, L. V., Phillips, J. C., and Harrison, C. (2011). Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 61:1–61:11. http://dx.doi.org/10.1145/2063384.2063466.

[113] Miller, S. and Luding, S. (2004). Event-driven molecular dynamics in parallel. *Journal of Computational Physics*, 193(1):306–316. http://dx.doi.org/10.1016/j.jcp.2003.08.009.

[114] Model, J. and Herbordt, M. (2007). Discrete event simulation of molecular dynamics with configurable logic. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 15–158. http://dx.doi.org/10.1109/FPL.2007.4380640.

[115] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117. http://dx.doi.org/10.1109/JPROC.1998.658762.

[116] Moraitakis, G., Purkiss, A. G., and Goodfellow, J. M. (2003). Simulated dynamics and biological macromolecules. *Reports on Progress in Physics*, 66(3):383. http://dx.doi.org/10.1088/0034-4885/66/3/203.

[117] Narumi, T., Ohno, Y., Futatsugi, N., Okimoto, N., Suenaga, A., Yanai, R., and Taiji, M. (2006). A high-speed special-purpose computer for molecular dynamics simulations: MDGRAPE-3. *NIC Workshop, From Computational Biophysics to Systems Biology, NIC Series*, 34:29–36.

[118] Narumi, T., Susukita, R., Ebisuzaki, T., Mcniven, G., and Elmegreen, B. (1999). Molecular Dynamics Machine: Special-purpose computer for molecular dynamics simulations. *Molecular Simulation*, 21:401–415. http://dx.doi.org/10.1080/08927029908022078.

[119] Narumi, T., Susukita, R., Furusawa, H., and Ebisuzaki, T. (2000a). 46 TFLOPS special-purpose computer for molecular dynamics simulations: WINE-2. In *5th International Conference on Signal Processing Proceedings (ICSP)*, volume 1, pages 575–582. http://dx.doi.org/10.1109/ICOSP.2000.894557.

[120] Narumi, T., Susukita, R., Koishi, T., Yasuoka, K., Furusawa, H., Kawai, A., and Ebisuzaki, T. (2000b). 1.34 TFLOPS molecular dynamics simulation for NaCl with a special-purpose computer: MDM. In *ACM/IEEE Conference on Supercomputing (SC)*, pages 54:1–54:20. http://dx.doi.org/10.1109/SC.2000.10016.

[121] Nelson, M. T., Humphrey, W., Gursoy, A., Dalke, A., Kalé, L. V., Skeel, R. D., and Schulten, K. (1996). NAMD: a parallel, object-oriented molecular dynamics program. *International Journal of High Performance Computing Applications*, 10(4):251–268. http://dx.doi.org/10.1177/109434209601000401.

[122] Nilsson, L. (2009). Efficient table lookup without inverse square roots for calculation of pair wise atomic interactions in classical simulations. *Journal of Computational Chemistry*, 30(9):1490–1498. http://dx.doi.org/10.1002/jcc.21169.

[123] NVIDIA (2012a). Fermi - the next generation CUDA architecture. http://www.nvidia.com/object/fermi_architecture.html.

[124] NVIDIA (2012b). High performance computing. http://www.nvidia.com/object/tesla_computing_solutions.html.

[125] NVIDIA (2012c). Tesla bio workbench - enabling new science. http://www.nvidia.com/object/tesla_bio_workbench.html.

[126] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., and Phillips, J. (2008). GPU computing. *Proceedings of the IEEE*, 96(5):879–899. http://dx.doi.org/10.1109/JPROC.2008.917757.

[127] Patterson, D. (2009). The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges. Expert Commentary. http://www.nvidia.com/object/fermi-architecture.html.

[128] Paul, G. (2007). A complexity O(1) priority queue for event driven molecular dynamics simulations. *Journal of Computational Physics*, 221(2):615–625. http://dx.doi.org/10.1016/j.jcp.2006.06.042.

[129] Petersen, H. G. (1995). Accuracy and efficiency of the Particle Mesh Ewald method. *Journal of Chemical Physics*, 103(9):3668–3679. http://dx.doi.org/10.1063/1.470043.

[130] Phillips, J. C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R. D., Kalé, L., and Schulten, K. (2005). Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802. http://dx.doi.org/10.1002/jcc.20289.

[131] Phillips, J. C., Stone, J. E., and Schulten, K. (2008). Adapting a message-driven parallel application to GPU-accelerated clusters. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 8:1–8:9. http://dx.doi.org/10.1109/SC.2008.5214716.

[132] Phillips, J. C., Zheng, G., Kumar, S., and Kalé, L. V. (2002). NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 36:1–36:18. http://dx.doi.org/10.1109/SC.2002.10019.

[133] Plimpton, S. (1995). Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19. http://dx.doi.org/10.1006/jcph.1995.1039.

[134] Ponder, J. W. and Case, D. A. (2003). Force fields for protein simulations. *Advances in Protein Chemistry*, 66:27–85. http://dx.doi.org/10.1016/S0065-3233(03)66002-X.

[135] President's Information Technology Advisory Committee (2005). Computational science: Ensuring america's competitiveness. National Coordination Office for Information Technology Research and Development. http://www.nitrd.gov.

[136] Proctor, E. A., Ding, F., and Dokholyan, N. V. (2011). Discrete molecular dynamics. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(1):80–92. http://dx.doi.org/10.1002/wcms.4.

[137] PubMed (2012). US National Library of Medicine, National Institutes of Health. PubMed website. http://www.ncbi.nlm.nih.gov/pubmed/.

[138] Rapaport, D. (1980). The event scheduling problem in molecular dynamic simulation. *Journal of Computational Physics*, 34(2):184–201. http://dx.doi.org/10.1016/0021-9991(80)90104-7.

[139] Rapaport, D. C. (2004). *The art of molecular dynamics simulation*. Cambridge University Press, 2nd edition. http://dx.doi.org/10.1017/CBO9780511816581.

[140] Reinders, J. (2005). *VTune Performance Analyzer Essentials*. Intel Press.

[141] Riken (2006). Completion of a one-petaflops computer system for simulation of molecular dynamics. http://www.riken.jp/engn/r-world/info/release/press/2006/060619/index.html. Press Release.

[142] Rinke, S. and Homberg, W. (2010). QPACE: Energy-efficient high performance computing. In *PRACE Workshop: New Languages and Future Prototypes*. http://www.prace-ri.eu/PRACE-Workshop-New-Languages.

[143] Sagui, C. and Darden, T. A. (1999). Molecular dynamics simulations of biomolecules: long-range electrostatic effects. *Annual Review of Biophysics and Biomolecular Structure*, 28(1):155–179. http://dx.doi.org/10.1146/annurev.biophys.28.1.155.

[144] Schofield, P. (1973). Computer simulation studies of the liquid state. *Computer Physics Communications*, 5(1):17–23. http://dx.doi.org/10.1016/0010-4655(73)90004-0.

[145] SciEngines (2012). SciEngines website. http://www.sciengines.com.

[146] Scrofano, R., Gokhale, M., Trouw, F., and Prasanna, V. K. (2006). A hardware/software approach to molecular dynamics on reconfigurable computers. In *The 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 23–34. http://dx.doi.org/10.1109/FCCM.2006.46.

[147] Scrofano, R., Gokhale, M. B., Trouw, F., and Prasanna, V. K. (2008). Accelerating molecular dynamics simulations with reconfigurable computers. *IEEE Transactions on Parallel and Distributed Systems*, 19(6):764–778. http://dx.doi.org/10.1109/TPDS.2007.70777.

[148] Scrofano, R. and Prasanna, V. K. (2004). Computing Lennard-Jones potentials and forces with reconfigurable hardware. In *International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA)*, pages 284–290.

[149] Scrofano, R. and Prasanna, V. K. (2006). Preliminary investigation of advanced electrostatics in molecular dynamics on reconfigurable computers. In *ACM/IEEE Conference on Supercomputing (SC)*, pages 90:1–90:12. http://dx.doi.org/10.1145/1188455.1188550.

[150] Severin, P. M. D., Zou, X., Gaub, H. E., and Schulten, K. (2011). Cytosine methylation alters DNA mechanical properties. *Nucleic Acids Research*, 39(20):8740–8751. http://dx.doi.org/10.1093/nar/gkr578.

[151] Shainer, G., Ayoub, A., Lui, P., and Liu, T. (2011). Raising the speed limit: New GPU-to-GPU communications model increases cluster efficiency. *Scientific Computing: Information Technology for Science*.

[152] Shan, Y., Klepeis, J., Eastwood, M., Dror, R., and Shaw, D. (2005). Gaussian split Ewald: A fast Ewald mesh method for molecular simulation. *Journal of Chemical Physics*, 122(5):54101:1–54101:13. http://dx.doi.org/10.1063/1.1839571.

[153] Shaw, D. E., Deneroff, M. M., Dror, R. O., Kuskin, J. S., Larson, R. H., Salmon, J. K., Young, C., Batson, B., Bowers, K. J., Chao, J. C., Eastwood, M. P., Gagliardo, J., Grossman, J. P., Ho, C. R., Ierardi, D. J., Kolossváry, I., Klepeis, J. L., Layman, T., McLeavey, C., Moraes, M. A., Mueller, R., Priest, E. C., Shan, Y., Spengler, J., Theobald, M., Towles, B., and Wang, S. C. (2007). Anton, a special-purpose machine for molecular dynamics simulation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. SIGARCH Computer Architecture News, 35(2):1-12. http://dx.doi.org/10.1145/1250662.1250664.

[154] Shaw, D. E., Dror, R. O., Salmon, J. K., Grossman, J. P., Mackenzie, K. M., Bank, J. A., Young, C., Deneroff, M. M., Batson, B., Bowers, K. J., Chow, E., Eastwood, M. P., Ierardi, D. J., Klepeis, J. L., Kuskin, J. S., Larson, R. H., Lindorff-Larsen, K., Maragakis, P., Moraes, M. A., Piana, S., Shan, Y., and Towles, B. (2009). Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 39:1–39:11. http://dx.doi.org/10.1145/1654059.1654099.

[155] Shi, G. and Kindratenko, V. (2008). Implementation of NAMD molecular dynamics non-bonded force-field on the cell broadband engine processor. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8. http://dx.doi.org/10.1109/IPDPS.2008.4536470.

[156] Shirvanyants, D., Ding, F., Tsao, D., Ramanchandran, S., and Dokholyan, N. V. (2012). DMD: An efficient and versatile simulation method for fine protein characterization. *The Journal of Physical Chemistry B*. In press. http://dx.doi.org/10.1021/jp2114576.

[157] Shu, W. and Kalé, L. (1991). Chare Kernel - a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11(3):198–211. http://dx.doi.org/10.1016/0743-7315(91)90044-A.

[158] Sigurgeirsson, H., Stuart, A., and Wan, W. L. (2001). Algorithms for particle-field simulations with collisions. *Journal of Computational Physics*, 172(2):766–807. http://dx.doi.org/10.1006/jcph.2001.6858.

[159] Skeel, R. D., Tezcan, I., and Hardy, D. J. (2002). Multiple grid methods for classical molecular dynamics. *Journal of Computational Chemistry*, 23(6):673–684. http://dx.doi.org/10.1002/jcc.10072.

[160] Smith, S. W., Hall, C. K., and Freeman, B. D. (1997). Molecular dynamics for polymeric fluids using discontinuous potentials. *Journal of Computational Physics*, 134(1):16–30. http://dx.doi.org/10.1006/jcph.1996.5510.

[161] Stone, J. E., Hardy, D. J., Ufimtsev, I. S., and Schulten, K. (2010). GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling*, 29(2):116–125. http://dx.doi.org/10.1016/j.jmgm.2010.06.010.

[162] Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., and Schulten, K. (2007). Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640. http://dx.doi.org/10.1002/jcc.20829.

[163] Sumanth, J., Swanson, D., and Jiang, H. (2003). Performance and cost effectiveness of a cluster of workstations and MD-GRAPE 2 for MD simulations. In *Second International Symposium on Parallel and Distributed Computing*, pages 244–249. http://dx.doi.org/10.1109/ISPDC.2003.1267670.

[164] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210.

[165] Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *Queue*, 3(7):54–62. http://dx.doi.org/10.1145/1095408.1095421.

[166] Taiji, M. (2004). MDGRAPE-3 chip: A 165-Gflops application-specific LSI for molecular dynamics simulations. In *IEEE Hot Chips Symposium*, pages 1–12.

[167] Taiji, M., Narumi, T., Ohno, Y., Futatsugi, N., Suenaga, A., Takada, N., and Konagaya, A. (2003). Protein Explorer: A petaflops special-purpose computer system for molecular dynamics simulations. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 15:1–15:10. http://dx.doi.org/10.1145/1048935.1050166.

[168] Toyoda, S., Miyagawa, H., Kitamura, K., Amisaki, T., Hashimoto, E., Ikeda, H., Kusumi, A., and Miyakawa, N. (1999). Development of MD Engine: High-speed accelerator with parallel processor design for molecular dynamics simulations. *Journal of Computational Chemistry*, 20(2):185–199. http://dx.doi.org/10.1002/(SICI)1096-987X(19990130)20:2%3C185::AID-JCC1%3E3.0.CO;2-L.

[169] Urbanc, B., Borreguero, J. M., Cruz, L., and Stanley, H. E. (2006a). Ab initio discrete molecular dynamics approach to protein folding and aggregation. *Methods in enzymology*, 412:314–338. http://dx.doi.org/10.1016/S0076-6879(06)12019-4.

[170] Urbanc, B., Cruz, L., Teplow, D. B., and Stanley, H. E. (2006b). Computer simulations of Alzheimer's Amyloid β-Protein folding and assembly. *Current Alzheimer Research*, 3(5):493–504. http://dx.doi.org/10.2174/156720506779025170.

[171] Van Der Spoel, D., Lindahl, E., Hess, B., Groenhof, G., Mark, A. E., and Berendsen, H. J. C. (2005). GROMACS: fast, flexible, and free. *Journal of Computational Chemistry*, 26(16):1701–1718. http://dx.doi.org/10.1002/jcc.20291.

[172] Verlet, L. (1967). Computer "Experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review*, 159(1):98–103. http://dx.doi.org/10.1103/PhysRev.159.98.

[173] Vladimirov, A. (2012). Arithmetics on Intel's Sandy Bridge and Westmere CPUs: not all FLOPs are created equal. Technical report, Colfax Research. http://research.colfaxinternational.com/.

[174] Wikipedia (2012). Protein folding - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Protein_folding.

[175] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76. http://dx.doi.org/10.1145/1498765.1498785.

[176] Wolinski, C., Trouw, F., and Gokhale, M. (2003). A preliminary study of molecular dynamics on reconfigurable computers. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 304–307.

[177] Xilinx (2012). Xilinx website. http://www.xilinx.com.

[178] Young, C., Bank, J. A., Dror, R. O., Grossman, J. P., Salmon, J. K., and Shaw, D. E. (2009). A 32x32x32, spatially distributed 3D FFT in four microseconds on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 23:1–23:11. http://dx.doi.org/10.1145/1654059.1654083.

[179] Yun, S. and Guy, H. R. (2011). Stability tests on known and misfolded structures with discrete and all atom molecular dynamics simulations.

Journal of Molecular Graphics and Modelling, 29(5):663–675. http://dx.doi.org/10.1016/j.jmgm.2010.12.002.

[180] Zhou, Y. and Karplus, M. (1997). Folding of a model three-helix bundle protein: a thermodynamic and kinetic analysis. *Proceedings of the National Academy of Sciences of the United States of America*, 94(26):14429–14432.

Curriculum Vitae

Md. Ashfaquzzaman Khan

CONTACT

Dept. of Electrical and Computer Engineering, Boston University
8 Saint Mary's Street, Boston, MA 02215, USA

Tel: 917-496-2699, E-mail: azkhan@bu.edu
Homepage: `http://people.bu.edu/azkhan/`

RESEARCH INTERESTS

- Computer Architecture and Multicore/Reconfigurable Systems
- High-performance Computing and Acceleration of Bio-medical Applications
- CAD/EDA Tools and Simulation Technologies

EDUCATION

PhD, Computer Engineering, Sept 2008 - Jan 2013 (Defense: June 2012)

- Dept. of Electrical and Computer Engineering, Boston University, Boston, MA
- Advisor: Prof. Martin C. Herbordt
- Dissertation Title: "Scalable Molecular Dynamics Simulation using FPGAs and Multicore Processors"
- GPA: 3.88/4.0 **(Ranked 1st in the ECE PhD Qualifying Exam 2009)**

Master of Eng., Electronic Engineering, April 2004 - March 2006

- Graduate School of Engineering, Tohoku University, Sendai, Japan
- Thesis Title: "A Study on Time-multiplexing of Reconfigurable LSI"
- GPA: 3.93/4.0

Bachelor of Eng., Electronic Engineering, April 2000 - March 2004

- Faculty of Engineering, Tohoku University, Sendai, Japan
- GPA: 3.88/4.0, Major GPA: 3.92/4.0 **(Ranked 1st in the class)**

PROFESSIONAL EXPERIENCE

Research Assistant, Boston University, MA (Sept 2008 - June 2012)

- Worked towards PhD under the supervision of Prof. Martin C. Herbordt. Focused on acceleration of bio-medical applications, particularly Molecular Dynamics Simulation and its discrete version, using advanced systems like FPGAs, GPUs and Multicore processors. Also studied energy/temperature aspects of parallel applications in general. Actively took part in graduate level course preparation/instruction and preparation of research proposals.

Intern, Intel Corporation, Hudson, MA (May 2011 - October 2011)

- Enabled UPF-based (Unified Power Format, an industry standard) power aware front end design flow, especially for next generation CPUs. Specific works included the following items. 1. Proposing a UPF-based design methodology to work seamlessly with the existing design flow and to ensure smooth transition. 2. Converting power specification from XML-based Intel-internal format to UPF-based description. 3. Developing a library of Tcl procedures to reduce the verbosity of native UPF without sacrificing readability. 4. Establishing a backward compatible conversion path from UPF to XML, to support other Intel-internal tools. 5. Validating the new methods and the converted design. 6. Developing documentation and training materials.

Intern, Microsoft Research, Redmond, WA (May 2010 - July 2010)

- Developed gNOSIS, a board-level debugging and verification tool for FPGA designs. gNOSIS uses the Capture/Readback features of the FPGA to checkpoint the entire state of the circuit with little or no modification to the DUT. It then correlates the design registers provided in the netlist with their state in the FPGA configuration memory, and with the expected state. If the states match, execution proceeds by restoring the state of the FPGA and continuing execution for a set number of cycles. When an error is encountered, the time and location of the error is reported and the last good checkpoint is used for further debugging. gNOSIS eliminates the manual labor and long wait times required by currently available tools (e.g. Chipscope), and provides greater visibility at a lower cost.

Engineer, Sony Corporation, Japan (April 2006 - August 2008)

- Developed a system level simulator for Cell Broadband Engine Architecture (CBEA). Written in SystemC, this simulator can provide cycle information for a given benchmark circuit, when it will run on CBEA, with 95% accuracy. The simulator runs a few hundred times faster than Cell SDK and thus provides a quick way to evaluate different architectural parameters. I also made an extended version of the simulator, which can handle up to 128 SPEs (Synergistic Processing Elements).
- Developed Noise Reduction Application for Sony digital camera, Cybershot series (Market arrival time: 2009).

Intern, Panasonic Corporation, Japan (August 2003 - September 2003)

- Verified scripts that were written to accelerate system-level design of digital circuits.

Part-time Programmer, DataFair Ltd., Japan (Feb 2002 - Mar 2006)

- Developed business software for wholesale product suppliers using Visual Basic 6, MS-SQL and MS-ACCESS.

COMPUTER SKILLS

Platforms: Linux/Unix, Windows
Software Development Environments: Eclipse, MS Visual Studio, Emacs
Software Programming Languages:

- C, C++, Assembly
- Perl, Tcl
- HTML, JavaScript, PHP, MySQL, Visual Basic, MS-SQL
- Others: Ruby, Java, MATLAB

Hardware Description/Programming Languages: Verilog, VHDL, SystemC
EDA Tools: Cadence Virtuoso Layout Editor, Dracula DRC; Synopsys Design Compiler, NanoSim, CosmosScope, VCS NLP; Mentor Graphics Modelsim; Springsoft Verdi
FPGA IDEs: Xilinx ISE, Altera Quartus
Others: Pthread, Openmp, VTune, Perf, Pfmon, Cuda

PATENT AND PUBLICATIONS (Selected)

PATENT

- Md. Ashfaquzzaman Khan, Yasushi Fukuda, **"Data Processing Apparatus, Method Therefor, And Computer Program"**, Pending Application No. 12/487799, Filed 2009, USA (Original: Pending Application No. P2008-161516, Filed 2008, Japan)

PUBLICATIONS

- Md. Ashfaquzzaman Khan, Martin C. Herbordt (2011), **"Parallel Discrete Molecular Dynamics Simulation with Speculation and In-Order Commitment"**, Journal of Computational Physics, 230 (17): 6563-6582
- Md. Ashfaquzzaman Khan, Can Hankendi, Ayse K. Coskun, Martin C. Herbordt (2011), **"Application Level Optimizations for Energy Efficiency and Thermal Stability"**, Annual Workshop on High Performance Embedded Computing (HPEC)
- Md. Ashfaquzzaman Khan, Can Hankendi, Ayse K. Coskun, Martin C. Herbordt (2011), **"Software Optimization for Performance, Energy, and Thermal Distribution: Initial Case Studies"**, International Green Computing Conference and Workshops (IGCC), pp 1-6
- Matt Chiu, Md. Ashfaquzzaman Khan, Martin C. Herbordt (2011), **"Efficient Calculation of Pairwise Nonbonded Forces"**, Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp 73-76
- Md. Ashfaquzzaman Khan, Richard Neil Pittman, Alessandro Forin (2010), **"gNOSIS: A Board-level Debugging and Verification Tool"**, International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp 43-48

- Martin C. Herbordt, Md. Ashfaquzzaman Khan, Tony Dean (2009), **"Parallel Discrete Event Simulation of Molecular Dynamics Through Event-Based Decomposition"**, International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp 129-136
- Bharat Sukhwani, Matt Chiu, Md. Ashfaquzzaman Khan, Martin C. Herbordt (2009), **"Effective Floating Point Applications on FPGAs: Examples from Molecular Modeling"**, High Performance Embedded Computing (HPEC)
- Martin C. Herbordt, Bharat Sukhwani, Matt Chiu, Md. Ashfaquzzaman Khan (2009), **"Production Floating Point Applications on FPGAs"**, Symposium on Application Accelerators in High Performance Computing (SAAHPC)
- Roel Pantonial, Md. Ashfaquzzaman Khan, Naoto Miyamoto, Koji Kotani, Shigetoshi Sugawa, Tadahiro Ohmi (2007), **"Improving Execution Speed of FPGA using Dynamically Reconfigurable Technique"**, Asia and South Pacific Design Automation Conference (ASP-DAC), pp 108-109
- Md. Ashfaquzzaman Khan, Naoto Miyamoto, Roel Pantonial, Koji Kotani, Shigetoshi Sugawa, Tadahiro Ohmi (2006), **"Improving Multi-Context Execution Speed on DRFPGAs"**, Asian Solid-State Circuits Conference (A-SSCC), pp 275-278
- Md. Ashfaquzzaman Khan, Naoto Miyamoto, Takeshi Ohkawa, Amir Jamak, Soichiro Kita, Koji Kotani, Tadahiro Ohmi (2004), **"An Aproach to Realize Time-sharing of Flip-Flops in Time-multiplexed FPGAs"**, International Conference on Field Programmable Technology (ICFPT), pp 351-354

AWARDS AND HONORS (Selected)

- **Ranked 1st in the ECE PhD Qualifying Exam:** April 2009, Department of Electrical & Computer Engineering, Boston University, Boston, MA.
- **Dean's Fellowship Award 2008:** Awarded by the Dean of the College of Engineering, Boston University. The award included tuition for PhD course and monthly stipend.
- **ASP-DAC 2007 Special Feature Award:** Awarded at the University LSI Design Contest of the ASP-DAC 2007, in recognition of the design and implementation of FP3, a dynamically reconfigurable LSI. I developed this LSI as a part of my master's research work.
- **Outstanding Research Award 2006:** Awarded by the Electrical & Information Eng. Managing Board, Graduate School of Engineering, Tohoku University, in recognition of excellent research for master's thesis.
- **Tohoku University President's Award 2004:** Awarded by the President of Tohoku University, in recognition of excellent undergraduate academic performance. I topped the merit list of the Department of Electronic Engineering.

- **Monbukagakusho Scholarship 1999-2006:** Awarded by the Ministry of Education, Culture, Sports, Science and Technology, Japan. I received it for two terms, 1999-2004 and 2004-2006, for a total of seven years. The scholarship provided full tuition fee and a monthly stipend for undergraduate and master's studies in Japan.
- **Certificate of Excellence 1997:** Awarded by the Adjutant General of Army, Bangladesh, in recognition of excellent performance in the countrywide Higher Secondary Certificate (H.S.C.) examination. I topped the merit list in my region. I also became the 2nd top ranked student in the entire country.
- **Bangladesh Prime Minister's Award 1995:** Awarded by the Prime Minister of the People's Republic of Bangladesh for ranking in the merit list (top 20) in the Secondary School Certificate (S.S.C.) examination.
- **Others:** Dean's Award 2004, IEICE Encouragement Prize 2004

PROFESSIONAL ACTIVITIES (REVIEWS)

- HEART (Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies): 2012
- DAC (Design Automation Conference): 2012
- SASP (Symposium on Application Specific Processors): 2011
- JEST (Journal of Electronic Science and Technology): 2011
- FCCM (Symposium on Field-Programmable Custom Computing Machines): 2011, 2010
- PARCO (Journal of Parallel Computing): 2010
- LSPP (Large-Scale Parallel Processing Workshop): 2010
- PDS (Transactions on Parallel and Distributed Systems): 2009

Last updated: June 14, 2012