FPGA ACCELERATION OF MOLECULAR DYNAMICS SIMULATIONS

YONGFENG GU

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

BOSTON UNIVERSITY

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

FPGA ACCELERATION OF MOLECULAR DYNAMICS SIMULATIONS

by

YONGFENG GU

B.S., Fudan University, 2000
M.S., Fudan University, 2003

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2008

Approved by

First Reader

Martin Herbordt, Ph.D.
Professor of Electrical and Computer Engineering

Second Reader

Roscoe Giles, Ph.D.
Professor of Electrical and Computer Engineering

Third Reader

Wei Qin, Ph.D.
Professor of Electrical and Computer Engineering

Fourth Reader

Sandor Vajda, Ph.D.
Professor of Biomedical Engineering

FPGA ACCELERATION OF MOLECULAR DYNAMICS SIMULATIONS

(Order No.)

YONGFENG GU

Boston University, College of Engineering, 2008

Major Professor: Martin Herbordt, Ph.D.,
Professor of Electrical and Computer Engineering

ABSTRACT

While molecular dynamics simulations (MD) are a fundamental method for gaining the understanding of chemical and biological systems, their computational cost is extremely high: Simulating macromolecules requires thousands of node hours and cell-level systems remain altogether out of reach. We address this issue by using an emerging mode of high performance computing that is based on configurable logic in the form of Field Programmable Gate Arrays (FPGAs). The problem is that, while FPGAs have often delivered 100-fold speed-ups per node over microprocessor-based systems, the applications have generally been limited to those with small regular kernels operating on low-precision integer data types. MD possesses neither. We address this problem by creating an explicitly designed FPGA-coprocessor that can be integrated into generic commercially available systems.

MD is an iterative technique: the forces on each particle are computed, then applied using the equations of motion. We use standard partitioning by computing bonded forces, motion updates, and bookkeeping on the host, while computing the remaining forces (which dominate) on the FPGA accelerator. For the short-range forces we combine the following: cell lists, systematically determined interpolation and precision, handling of exclusion, and

support for models with large numbers of particles. This has required new microarchitectures for the cell list processor and off-chip memory controller; and extensive experimentation to explore the design space to optimize precision, interpolation order, interpolation mode, table sizes, and simulation quality. To perform efficient and accurate numerical computation on FPGA, we created a novel arithmetic mode that is tuned for computing high order polynomial interpolation. For the long-range forces we use the multigrid method: we show that this is an excellent match to FPGAs with the primary operations having a favorable systolic structure and taking advantage of the large number of independently addressable RAMs.

The significance of this work lies at several levels: the 5× to 10× acceleration of MD production code while retaining simulation quality; the turings of algorithms for the FPGA; the system integration; the new arithmetic mode; the numerous novel microarchitectures; and the methods for optimizing MD implementations on FPGAs.

Contents

List of Tables

List of Figures

List of Abbreviations

ASIC	.....	Application Specific Integrated Circuit
b	.....	bit
B	.....	byte
BCB	.....	Bioinformatics and Computational Biology
CPU	.....	Central Processing Unit, the functional core of a PC, one of many in an MPP
DSP	.....	Digital Signal Processor
EDA	.....	Electronic Design Automation
FFT	.....	Fast Fourier Transform
FLOPs	.....	Floating-point operations per second
FPGA	.....	Field Programmable Gate Array
FSB	.....	Front Side Bus
Gb	.....	Gigabits, 2^{30} (10^9) bits
GB	.....	Gigabytes, 2^{30} (10^9) bytes
GFLOPs	.....	Giga FLOPS, 10^9 floating-point operations per second
GPP	.....	General Purpose Processors
GPU	.....	Graphics Processing Unit
GROMACS	.....	Groningen Machine for Chemical Simulations, an open-source molecular modeling program
GUI	.....	Graphical User Interface

HDL	.....	Hardware Design Language, for example VHDL or Verilog
HPC	.....	High Performance Computing
HPRC	.....	High Performance Reconfigurable Computing
HLL	.....	High Level Language, specifically for software design, as opposed to an HDL
IP	.....	Intellectual Property
JTAG	.....	Joint Test Action Group. IEEE standard 1149.1
Kb	.....	Kilobits, 2^{10} (10^3) bits
KB	.....	Kilobytes, 2^{10} (10^3) bytes
LAMP	.....	Logic Architecture Model Parameterization, novel EDA tool set
LSB	.....	Least Significant Bit
LUT	.....	Look Up Table, the hardware element that underlies programmable functions in FPGAs
MAC	.....	Multiplication and Accumulation
Mb	.....	Megabits, 2^{20} (10^6) bits
MB	.....	Megabits, 2^{20} (10^6) bytes
MPP	.....	Massively Parallel Processor
MSB	.....	Most Significant Bit
PAR	.....	Place And Route
PC	.....	Personal Computer
PCI	.....	Peripheral Component Interconnect, a common system bus
PE	.....	Processing Element
PME	.....	Particle Mesh Ewald Sum
RTL	.....	Register Transfer Level
SGI	.....	Silicon Graphics, Inc.

SIMD	.....	Single Instruction, Multiple Data
TB	.....	Terabytes, 2^{40} (10^{12}) bytes
TFLOPs	.....	Tera FLOPS, 10^{12} floating-point operations per second
UI	.....	User Interface
VHDL	.....	VHSIC Hardware Description Language
VLIW	.....	Very Long Instruction Word

Chapter 1

Introduction

1.1 The Problem

In 2005, the President's Information Technology Advisory Committee reported that "together with theory and experimentation, computational science now constitutes the 'third pillar' of scientific inquiry, enabling researchers to build and test models of complex phenomena that cannot be replicated in the laboratory. [Pre05]" And in 2006, a National Science Foundation Blue Ribbon Panel reported that within computational science, "computer simulation has emerged as a powerful tool, one that promises to revolutionize the way engineering and science are conducted. [oSBES06]" The method of Molecular Dynamics simulation (MD) in particular is critical: it lies at the core of computational chemistry and is central to computational biology. Applications of MD range from the practical, e.g., drug design, to basic research in understanding disease processes. An example of the latter can be found in a study by DeMarco and Dagett [DD04] on the PrP protein that lies at the root of amyloid (e.g., "mad cow" and Alzheimer's) diseases. Here, MD simulations indicated that aggregations of misfolded *intermediates* of PrP are the pathogenic species; this crucial finding would have been virtually impossible to discover through any mechanism other than simulation.

MD is compute bound. While studies conducted with a few minutes of PC time can be useful, the reality is that computational demand is virtually insatiable: almost any MD simulation will be improved by simulating longer physical time, a larger physical system, and a more detailed model. Large-scale experiments can be defined as those involving large computing clusters running for months at a time (often called "heroic" computations).

Heroic MD computations have included simulations of viruses (9,000 node-days [FAL+06]), ion channels through cell membranes (50,000 node-days per significant event [KATS06]), and ribosomes (2,000 node-days per 4ns simulated time [Poi06]). Just as with simulations of computer designs–with which computer engineers are most familiar–simplifications improve throughput, but can lead to misleading results. In the virus study, it was found that simulation of the whole virus was required to obtain a stable structure; leaving out the virus core decreased the simulation time, but resulted in instability.

We now summarize the state-of-the-art in the application of MD to biochemical systems: heroic simulations enable basic modeling of macromolecules and certain even-larger structures, with the modeled time scale being inversely related to the length scale. For example, ribosomes can plausibly be simulated for only nanoseconds, but small proteins for milliseconds. Simulations of long physical time-scales, such as the physical seconds required to model protein folding, remain far out of reach, as do even brief simulations of much larger structures, such as the nucleosome. In addition, these large-scale simulations that currently *are* possible, are unavailable to most researchers: e.g., a large fraction of compute time at the National Center for Supercomputing Applications is *already* used for MD. [1]

The problem that this dissertation investigates is how to bring to researchers in computational science substantially more cost-effective MD capability. We address this problem by enabling the use for MD of a new generation of computers based on reconfigurable logic devices: Field Programmable Gate Arrays or FPGAs.

FPGAs are commodity integrated circuits whose (apparent) logic can be determined, or programmed, in the field. This is in contrast to application-specific integrated circuits (ASICs) whose logic is fixed. The tradeoff is that FPGAs are less dense and fast than ASICs. Often, however, their flexibility more than makes up for these drawbacks: FPGAs can be reconfigured in milliseconds to support a new set of parameters, or an entirely

[1] http:// www.ci-partnership.org /Allocations /awards.html

new application. When applied to computational problems, FPGA-based solutions have sometimes provided multiple order-of-magnitude acceleration over microprocessors. And since FPGAs are commodity parts, the system cost remains moderate.

The key to this problem, however, is not the use of the hardware itself, whose technology is mainstream; rather it is to create the particular FPGA configurations to accelerate MD. The challenge here is that the programming model differs drastically from that of standard computers, including parallel ones. As a consequence, efficient implementation on an FPGA-based system–of a complex application such as MD–requires restructuring that application at many levels. Other, simpler, implementation methods, such as direct translation of MD software codes into FPGA logic, result in only a fraction of potential performance being achieved. The approach taken here, rather, *is the FPGA-aware redesign of almost every aspect of MD.* This includes changes in algorithm selection, creation of new algorithms, creation of a new arithmetic mode, experimentally determined precision, integration into production MD systems, optimization of data structures and arithmetic, and creation of microarchitectures to implement numerous functions. These FPGA-aware redesigns comprise the bulk of the contributions of this dissertation.

1.2 Molecular Dynamics Computations

Molecular Dynamics refers to an approach to computer simulations of systems where molecules and atoms interact with one another. There are many variations of MD depending on the detail at which the forces are described, e.g., whether quantum mechanics is modeled, or only Newtonian approximations. Often quantum mechanical MD simulations are used to construct force models that are used by other "higher-level" MD systems wherein forces are approximated with Newtonian mechanics. Another source of variation is how the simulation is advanced. The first MD programs advanced only when events occurred; currently, the dominant method advances simulations by time-step–although discrete event based MD (DMD) remains important, especially in modeling long time scales. In this dissertation we restrict our attention to time-step driven Newtonian MD, which is

now described.

MD is the iterative application of Newtonian mechanics to ensembles of atoms and molecules. MD runs in phases: the forces on each particle are computed, then applied using the equations of motion. In contemporary MD systems, the force computations often involve many terms, including bonded (covalent, hydrogen) and non-bonded (van der Waals, Coulombic). For computational efficiency, the non-bonded forces are often partitioned into short- and long-range components. The complexity of the motion update and the bonded force computations is $O(N)$ in the number of particles, and generally requires only a small fraction of overall compute time. The complexity of the non-bonded force computations is potentially $O(N^2)$ in the direct implementation, and comprises the bulk of the computation. Complexity can be reduced substantially, however. For the short range component, $O(N)$ is obtained by dividing the system into cells and/or maintaining lists of particles within a certain distance of a given particle. For the long-range component, $O(N \log N)$ or better is obtained with transform- or grid-based methods. The choice of method and many other details of the implementation depend on the target computational platform.

1.3 High Performance Computing With Accelerators

The advent of vector supercomputers in the 1960s revolutionized high performance computing. While these architectures remain critical, since the mid-1990s most high-end systems have been based on collections of commodity microprocessors. Since then and until recently, performance of microprocessor-based systems has steadily improved: increasing chip densities have enabled extraction of greater amounts of instruction level parallelism; increases in operating frequency have lead to direct increases in throughput. In the last four years, however, problems with power consumption and heat dissipation have caused operating frequencies to stagnate at under 4GHz. Microprocessor architecture has topped out as well: while advances in process technology continue to provide ever more features per chip, these are no longer primarily used to augment individual CPUs; rather they are

used to *replicate* the processor cores. Research chips with dozens of CPUs have been built, and commercial versions are projected to be available in the next few years.

One of the greatest current challenges in computer engineering is the cost-effective use of these additional cores on-a-chip; and it appears likely that this will continue into the foreseeable future. The problem of developing efficient, portable, applications for parallel processors remains far from solved, even after having been studied intensively for over 40 years. The problems are much worse on a single chip, with the additional constraints of highly restricted memory bandwidth per processor. This realization is central to Intel's vision: for computationally intensive applications, non-microprocessor accelerators are likely to provide the most cost-effective solutions [Bha07]. Therefore, support is being developed for plug-compatible motherboard sockets where, e.g., a Xeon processor chip can be removed and replaced with an accelerator [Int07b, Xtr07b, DRC07]. Support for accelerators is also projected on the processor chip itself [Bha07]. For example, a processor chip may contain some mix of coarse and fine grained CPUs, plus accelerators.

Clearly the computer landscape is shifting dramatically, especially for high-performance computing (HPC). We now briefly review some alternatives, other than FPGAs, that are either currently available, or projected to be available soon.

- **Multicore.** Despite the challenges for their use in general purpose computing, multicore remain the default technology for new HPC platforms. The first generation of multi-core microprocessors, such as those from Intel and AMD, has symmetric mono-cores on one die.

- **Cell Processor.** As the number of cores per chip gets larger, maintaining computational efficiency becomes critical, and alternative designs may be more suitable. Of these asymmetric multi-cores, the best-known is propertyb the IBM Cell processor [Che07], which is also the compute engine in many game consoles. The Cell combines a single general purpose CPU with several vector units.

- **Manycore.** As the number of cores expands into double digits, it is generally referred

to as **manycore.** An 80-core research prototype [Int07a] has been built by Intel. These cores are likely to be heterogeneous.

- **Graphics Processor Units (GPUs).** GPUs are commodity high-end processors designed for graphics computations and found in most PCs. They are especially well-suited for applications requiring floating-point, but that can be cast as simple SIMD operations.

- FPGAs. FPGAs are described in more detail in the next Section; here we make the point that high-end FPGAs suitable for HPC are commodity processors whose primary markets are in router switches and DSP computers.

- **Large-scale SIMD.** Most processors have some SIMD component, such as graphics extensions to the standard instruction set. Large-scale SIMD of the type most famously represented by the Connection Machine [Hil85] remains an alternative with the ClearSpeed processors [Cle05]. SIMD accelerators have some more flexibility than GPUs. They have the disadvantage with respect to multicore, Cell, GPU, and FPGA, of not being the commodity processor for a high-volume application.

Each of these alternatives is currently thriving: hardware design and technology is constantly being updated; much work is being done to advance programming tools; and much research is being conducted to determine the appropriate mappings of applications. Still, these alternatives vary in power, programmability, programmer's model, and basic computational capability. Determining what computations are most cost-effective on which architecture is a critical and heavily studied problem. The consensus appears to be that there is no single correct platform, at least certainly not for all types of HPC computations. In this context, this work can be viewed as an examination of an important part of the HPC problem space with respect to a leading candidate HPC compute engine.

1.4 High End FPGAs as HPC Accelerators

1.4.1 Hardware

Reconfigurable computing is based on technology that conducts computation with reconfigurable circuits. Although this idea can be traced back at least to the 1960's, High Performance Reconfigurable Computing (HPRC) has only become feasible recently, i.e., when FPGAs could deliver significant computation power relative to both ASICs and microprocessors. One factor is that FPGAs are currently driving, rather than lagging, process technology [Tre04]. Another factor is that high-end FPGAs are now in fact hybrid chips with hundreds of ASIC components (especially multipliers and independently accessible memories) in addition to the configurable circuitry. For example, in the last four years, FPGAs have been manufactured with process technologies from 130 nm to 65 nm, taking the same exact of microprocessors. Also, because of their inherently reduced operating frequency, FPGA power consumption is usually less than five watts; in contrast, high performance DSPs and microprocessor consume tens of watts and hundreds of watts, respectively.

High-end FPGAs have the following characteristics.

- Programmable in milliseconds by uploading the desired configuration. This also means that the FPGA can be *reprogrammed* for other applications just as quickly.
- Millions of configurable gate-equivalents.
- Millions of programmable communication paths, both local and global.
- Design modules often available as 'intellectual property' (IP) blocks.
- Gigabit interfaces (Infiniband, Gigabit Ethernet, etc.) to off-chip devices.
- Hardwired on-chip components, particularly hundreds of independently accessible memory modules and ALUs.

- Hundreds of I/O pins that allow for integration into systems with many independently accessible off-chip caches.

And again, FPGA chip development *is driving process technology*, a role held until recently by DRAM chips [But03]. As a result, FPGA-based HPC has the critical attribute for new IT products: the ability to "ride the technology curve" as stated by Moore's Law.[2] As new generations of process technology—and thus FPGAs—emerge, existing *hardware* designs can be ported to them in much the same way that existing software can be loaded onto a new faster computer, thereby obtaining an analogous boost in performance.

1.4.2 Programmability

Among the alternative for HPC platforms, FPGAs are in some ways the most flexible: compared with other processors, which have fixed or tightly constrained data interfaces and predefined micro-functionality, reconfigurable circuits (along with ASICs) can be adopt an optimal data interface and devote most of the chip area (resources or transistors) to a specific application.

On the other hand, migrating HPC applications from microprocessor based systems to multi-core or other thread-based systems is easier for application experts and other programmers without FPGA-specific experience. This is because for thread-based systems, they need only consider how to map their algorithms to new hardware; for FPGA based systems, they also need to worry about the hardware design by themselves. This is either an advantage or disadvantage, depending on the experience of the implementer. With such expertise, one can tell the exact requirement to hardware so that they can choose proper trade-offs or even redesign of algorithm to achieve maximal performance with the same chip area, number of transistors, or power budget. Without such expertise, it is difficult to do this job efficiently, even impossible to achieve speed-up.

An analogous gap also exists when application experts develop applications on parallel supercomputers, and it was not surprising that the best supercomputer programmers are

[2]Transistor density on a chip doubles every 1-2 years

sometimes physicists and chemists. With FPGAs, however, the gap is wider than ever. Application experts are not only are required to learn new languages, instructions, system architectures, and programming models, but also have to face a tremendous design space. One example is that most users do not care much about precision in their applications, as long as the single or double precision floating-point is accurate enough. They do not search the design space in terms of precision, because microprocessors do not provide many options. Dealing with FPGAs, however, they have the freedom to choose any precision, but must first answer the question that what precision is proper. A proper precision here means acceptable accuracy and efficient hardware implementation, sometimes even involving choosing different arithmetic methods. Precision is only one of the new axes added to the design space, but introduces significant complexity. After combining the design space of application algorithms and that of FPGA implementation, the entire space becomes large and finding a optimal solutions can be challenging. For complex applications such as MD, however, exploring the design space carefully is necessary to achieve substantial performance improvement.

Providing universally accessible development methods for FPGA-based accelerators has been challenging. Much work, including this, is done with hardware description languages (HDLs), such as VHDL and Verilog. In order to facilitate users in porting and developing applications on FPGAs, many advanced tools are being introduced. Some of them employ high level languages as a programming interface to FPGA designs; in some, users are still aware of hardware/software partitioning and hardware implementation; others, e.g., Mitrion-C, can directly convert original high level language source code to FPGA designs with only a limited amount of modification.

1.4.3 High Performance FPGA-Based Computing

The computational power of HPRC is derived from two sources: (i) the thousand-fold parallelism possible when an entire chip is configured to perform a particular computation, and (ii) the fact that, unlike thread-based systems, payload is delivered from the pipelines

on every cycle. The promise of HPRC is thus high performance at a lower operating frequency, and thus lower power.

The areas of greatest success for HPRC have been in signal and communication processing. Here, small kernels dominate the computation; these kernels are also highly parallelizable, and can often make do with low precision and/or complexity of arithmetic modes.

Early work in HPRC often reported per-node accelerations in the hundreds, and even thousands. As HPRC has matured, however, a broader range of HPC applications is being addressed and the reported speed-ups have often been far more modest. Some of the well-know difficulties are as follows:

- **Chip area limitations.** While code size is generally not a high-order concern in HPC, in HPRC the size of the code directly affects the chip area required to implement the application. Although the relationship is indirect, the overall implication is that the more complex the application kernel, the more the chip area required to implement it. This results in reduced parallelism and thus performance.

- **Designer limitations.** Complex applications often require substantial expertise and design time to map them efficiently to FPGAs.

- **Amdahl's law limitations.** If the kernel does not dominate sufficiently (i.e., consist of, say, more than 95% of the execution time), then deeper application restructuring maybe necessary.

- **Component limitations.** A key attribute of modern FPGAs is their embedded "hard" components such as multipliers and independently accessible memory blocks (block RAMs). Floating-point support, however, remains modest; this limits substantially the FPGA's potential performance in classic HPC applications (see, e.g., [BHUH06] and references).

The combination of high potential performance coupled with relatively low operating

frequency makes performance of HPRC applications particularly sensitive to the quality of implementation. In an article in IEEE Computer, we enumerate some methods that are likely to be required to obtain high performance of non-trivial HPRC applications [HVG+07]. As with other high-performance platforms, the key is to avoid implementational overhead [Sny86].

Method 1: Use an algorithm optimal for FPGAs. For example, replacing an FFT with a direct correlation.

Method 2: Use a computing mode appropriate for FPGAs. For example, replacing random access with streaming.

Method 3: Use appropriate FPGA structures. For example, using common hardware versions of analogous data structures.

Method 4: Living with Amdahl's Law. For example, sometimes non-kernel code must also be optimized through application redesign.

Method 5: Hide latency of independent functions.

Method 6: Pipeline sequences of functions; use replication and rate-matching to remove bottlenecks.

Method 7: Take advantage of FPGA-specific hardware. For example, in MD, computing the Coulombic force using multigrid requires performing tricubic interpolations at streaming rate [GH07b]. This uses a number of independent block RAMs and a custom memory interleaving. These can be generated automatically [VH06].

Method 8: Use appropriate arithmetic precision.

Method 9: Use appropriate arithmetic mode.

Method 10: Minimize use of high-cost arithmetic operations.

Method 11: Create families of applications, not point solutions. HPRC applications are often complex and highly parameterized: this results in the code having variations not only in data format, but also in algorithm to be applied.

Method 12: Scale application for maximal use of FPGA hardware.

Most of these are used extensively in this work.

1.4.4 High Performance Reconfigurable Computing Platforms

Historically, HPRC has been conducted with coprocessor boards that plug into the PC's PCI bus. Vendors include Annapolis Microsystems, Nallatech, Maxeler, and many others [Ann03, Nal06, Tec07]. High-end HPRC systems with tighter system integration are produced by SRC [SRC05]. In the last few years, SMP vendors integrated FPGA-based nodes into their communication fabrics; examples include Cray and SGI [Cra05, Sil04]. And most recently, vendors such as DRC and XtremeData have created FPGA boards that are plug-compatible with microprocessors themselves, for immediate integration into commodity processor boards [Xtr07b, DRC07].

Systems have also been created that gang together a number of FPGAs. Medium-scale systems are available commercially from SRC, Cray, and SGI. Large-scale systems with hundreds of FPGAs have been built in research environments, particularly in Berkeley and Edinburgh [AAC+05, EPC05].

1.5 HPRC for MD

The research described in this thesis, acceleration of molecular dynamics simulations (MD) with HPRC, is interesting on at least two fronts. First, its acceleration is inherently important: substantial progress has been made in developing efficient and scalable codes (e.g., NAMD [Phi05] and GROMACS [VLH+05]). Second, it appears that, more so than with most floating-point intensive HPC applications, HPRC may offer substantial acceleration. One reason is that the kernels, while non-trivial, may still be "manageable" in the sense that with some optimization they fit on high-end FPGAs. Another is that although high precision is important, there may be room to reduce precision somewhat while still retaining the quality of the MD simulations. This fact has been used, not only by most FPGA implementations of MD, but by ASIC- [AFK+95] and von Neumann-based [VLH+05] versions as well.

The difficulties of HPRC acceleration of MD are of three aspects: accuracy, performance, and model size. We now discuss these in turn, beginning with simulation quality.

Accuracy is always the first criteria of an MD system. This is because, first, the output must be accurate so that users can trust the simulations; and second, even small numerical errors on each particle during each time step may be accumulated to become intolerable, and so prevent the simulation from converging. These factors partly explain why some computational scientists feel safe only with double precision, even after single precision being applied for many years. It is difficult to determine, in general, what precision is sufficient. Some reasons are as follows. Tracking individual particles is impractical, because system will be random only after a few collisions. Tracking invariant physical quantities, such as energy and momentum helps, but is still not sufficient because they are only projections of the simulation model status. Also, even for an ideal simulation system, "the errors introduced by the use of empirical potentials are difficult to quantify [KM02]"; the simulation quality requirement is thus a function of simulation model itself. In fact, as observed by La Penna, et al.: "in our very long simulations we did not see signs of instabilities, nor of any systematic drift" due to using single, rather than double precision floating-point [PLM+97]. Therefore use of reduced precision still enables acceptable simulation quality for many simulations, but requires careful design and proof.

Turning now to performance: computing with fewer bits or simpler arithmetic operations of course improves performance, but the biggest barrier is Amdahl's law. Once the densest computational task (the short range non-bonded forces) has been accelerated, the rest of the tasks become all the more significant and dominate timing. Based on current FPGA technology, it is impractical to map entire MD problem to FPGAs, so some tasks must be left on microprocessor. Therefore, the expected speedup is limited from 10 to 20. Scrupulous design and implementation is required, however, otherwise the potential speedup will be substantially diminished. Moreover, because some optimizations that can be successfully applied in software MD systems are not efficient or feasible on hardware, we need to design FPGA specific algorithms for optimization.

Capacity of the MD system means: number and types of particles, size of simulation box, and the ability to support various simulation options. Simulation duration can also be

regarded as a measure of capacity, but this mostly depends on accuracy and performance. For large and complex simulation models on any contemporary computational platform, data must be stored off-chip, which for HPRC involves careful design of interface, access pattern, coherency, and communication overhead. Supporting more simulation options means implementing more functionality, especially those for rare situations. For software, this may only mean adding instructions in memory; while for current FPGAs, it means more chip area, and consequently lower efficiency and performance.

How these difficulties are handled, together with those generic difficulties universal to HPRC, define the design space for FPGA-based acceleration of MD (FPGA/MD). Specifically, FPGA/MD has been studied by a number of groups [AAS+07, AKE+04, GVH06b, KP06, KUT+97, SP06] with the design space being spanned by several axes:

- **Precision:** Is 53 bits used (double precision), or 24 (single precision), or something else? How is the choice motivated?
- **Arithmetic mode:** Is floating-point used? Block floating-point? Scaled binary? Logarithmic representation? A hybrid representation?
- **MD code:** Is the base system it a standard production system? An experimental system? A reference code?
- **Target hardware:** What model FPGA is used? How is it integrated, on a plug-in board, or in a tightly integrated system?
- **Scope:** MD implementations have a vast number of variations - which are supported? How is the long-range force computation performed? With cut-off or a switch function? Or, is a more accurate, and more computationally complex, method used? Is this done on the FPGA or software?
- **Design flow:** How is FPGA configured? With a standard HDL, or a C-to-gate process, or some combination?

A major goal of this work is to investigate the viability of MD in current generation FPGA technology. While previous studies have made substantial progress, most have made compromises in performance, precision, or model size simulated. A preliminary MD system [AKE+04] was implemented on Transmogrifier 3 (TM3) with four Virtex-E 2000E FPGAs, computing Lennard-Jones force and motion update with fixed-point arithmetic. Other systems [AAS+07, KP06, SP06] reported recently were developed on SRC-6 MAP station, which contains two Xilinx Virtex II FPGAs. They all used single precision floating-point, only computed the short range non-bonded forces (Lennard-Jones and the real part of Smooth Particle Mesh Ewald Sum) on the FPGAs, and used SRC Carte complier to generate FPGA design from high level languages; the difference is the based MD code: [SP06] used their own reference code; [AAS+07] and [KP06] ported NAMD and Amber respectively. Finally, these floating-point based system achieved speedup between 2.7x to 4x over their pure software reference codes. Their speedups are limited by several factors: (i) floating-point arithmetic, even single precision restricts the number of pipelines i.e. parallelism; (ii) only accelerating the short range non-bonded forces computation bonds restricts the overall speedup; and (iii) C-to-gate design flow is not sufficient to explore the entire FPGA design space. We attempt to advance the art with numerous FPGA-centric optimizations, while retaining the MD simulations quality. In particular, our point in the design space is as follows:

- **Precision:** All implementations support variable precision. In addition, experiments measuring energy fluctuation (as described, e.g.: [AFK+95]) were conducted to determine the effects of varying precision on both performance and simulation quality. It was found that 35-bit precision may be optimal for many simulations.

- **Arithmetic mode:** We avoid floating-point, but retain accuracy with a new arithmetic mode that supports only the small number of alignments actually occurring in the computation.

- **Base MD code:** ProtoMol [Mat04], with further performance comparisons with

NAMD [Phi05].

- **Target hardware:** A generic PC and a commercial PCI plug-in board with two Xilinx VP70s [Ann06]. Performance of this configuration with other FPGAs is estimated with area and timing accurate design automation method.
- **Scope:** We describe the short-range force processor implementing cell-list and long-range force coprocessor implementing multigrid.
- **Design flow:** all major components (force pipelines, cell-list framework, off-chip memory controller) were designed from algorithm-level descriptions and implemented using VHDL. Where appropriate, algorithms were restructured to best use FPGA resources.

We find that even using 2004-era FPGA hardware we are able to achieve a 5× to 8× speedup over NAMD with little if any compromise in simulation accuracy.

1.6 Summary of Contributions

1.6.1 Overview

The contributions of this work come into four categories: (i) the acceleration of MD in and of itself, (ii) what this says about the viability of HPRC in general, (iii) the methods developed for MD acceleration, and (iv) the applicability of the methods developed for MD in accelerating other HPC applications. The first of these is immediate: we obtain significant speedup over the original software implementation, as well as over other production codes, while still retaining acceptable simulation accuracy. The second follows from the first: this research demonstrates that FPGAs are not only good for integer applications, but also viable in the floating-point domain, even without dedicated floating-point units. The final two sets of contributions are now described.

1.6.2 Acceleration of MD

We designed two FPGA coprocessors to compute the short range and the long range non-bonded force respectively. They were implemented on an Annapolis Micro-system FPGA development board and integrated into production codes, such as ProtoMol. The new system could perform complete MD simulation (FPGA computing non-bonded forces and host PC doing rest computation) and achieved 5× to 8× wall-clock speedup over NAMD on PC. To fairly compete with production MD codes, we not only implemented basic MD functions, such as the force equations following the textbook, but also implemented advanced algorithms applied by production codes, such as cell-list and exclusion. After accelerating the short range non-bonded force computation, the long range force became significant. In order to gain more speedup, we designed and implemented a long range non-bonded force coprocessor, which is the first published FPGA solution and may be the only one at this time for the problem. We also applied high order polynomial interpolation and analyzed different interpolation methods for better numerical accuracy, and created the semi-floating arithmetic mode that mapped efficiently to FPGAs. Highlights of MD acceleration include:

Short range non-bonded force coprocessor. It consists of a scalable force pipeline array that computes Lennard-Jones force and the short range part of Coulomb force in parallel. The force pipeline does piece-wise high order polynomial interpolation to approximate the force curve; interpolation coefficients are computed with orthogonal polynomial methods, which has the least mean squared approximation error and zero bias.

Implementation of Cell-list method. The force pipeline array is wrapped with a framework that supports the Cell-list method, so that we can reduce the short range non-bonded force evaluation from $O(N^2)$ to $O(N\dot{M})$, by only computing particles close to each other. Cell-list method or similar algorithms are employed by production MD codes for the same purpose. Without such support, we would not be able to improve end-to-end performance to overcome the production codes.

Exclusion of non-bonded forces. Excluding the non-bonded force between bonded

particles is a required by the physical law, but is problematic even for software implementation. It could be either expensive, e.g. excluding bonded pairs during non-bonded force computation, or inaccurate, e.g. computing forces brutally first and subtracting exclusion forces later with underflow. We propose a new method that checks multiple cut-offs based on types for bonded pairs. This method avoids those disadvantages and maps efficiently to FPGA hardware.

Multigrid coprocessor computing the long range non-bonded force. To extend the overall performance, we built the long range non-bonded force coprocessor based on multigrid method, which is the first multigrid FPGA implementation, and is probably so far the only FPGA solution computing the long range non-bonded force. It provides about 3x to 4x speedup over original software version and PME method of NAMD.

Large simulation with off-chip memory. The problem to simulate large models with our coprocessors is that the on-chip SRAMs are not big enough. We therefore design a cache scheme that swaps particle data between on-chip SRAMs and off-chip memory, so that we can simulate large models of up to 256K particles without loosing performance. Using off-chip memory, however, causes another problem that off-chip memory interfaces are not uniform on different hardware platforms and porting our designs to other platforms is difficult. Our solution is to define a general off-chip memory interface of a relaxing bandwidth requirement, only a couple hundred bits per cycle, and our coprocessors are designed based on this general interface rather than any specific hardware.

Semi Floating-point Numbering. Because numerical accuracy is critical to MD simulation, especially to force curve interpolation, performing calculation with pure fixed-point arithmetic is not practical. On the other hand, performing general floating-point arithmetic without dedicated hardware units decreases chip efficiency significantly on current FPGAs. We, therefore, created the Semi Floating-point numbering (Semi-FP) system by compromising floating-point and fixed-point. For most accuracy critical calculations, we used 35-bit Semi-FP, which maps efficiently to FPGAs while preserving computation accuracy.

1.6.3 General Computation Models and Designs

As MD is a special case of general computation problem, the technologies used in our acceleration are applicable to other HPC applications. One example is multigrid method, which is a general method to solve problems in different scales. In the area of numerical computing, it is always used to solve partial differential equations (PDEs), such as our long range non-bonded force coprocessor solving Poisson's equation in real space. Multigrid method involves various operations, e.g. discretization, relaxation, anterpolation/interpolation, local correction and direct solution. It is never trivial to implement and integrate all these operations on FPGA. After we developed the multigrid coprocessor for Coulomb force, we defined a multigrid computation model to adapt other multigrid based applications. In [GH07a], we mapped the linear image diffusion problem to the multigrid prototype and investigated the FPGA solution efficiency by exploring the design space in many axes. Because image processing can be performed with integers of small bit width and does not require discretization, the linear image diffusion processor achieved speedups in middle hundreds in both 2D and 3D cases.

Other general designs include multi-dimensional systolic array convolver for large convolutions, interleaved memory, Semi-FP operators, and so on. They are not just optimized modules or library; they are frameworks implementing certain computation models. Application developers can focus on mapping their problem to these existing frameworks by using their expertise of applications to design application specific operations.

1.7 Organization of the Rest Thesis

The rest of this thesis describes a number of methods to implement different aspects of MD, plus experiments to measure performance, verify the simulations, and determine validity. The methods span multiple "modes": algorithm creation and selection, numerical analysis, FPGA mapping and configuration design, logic design, system programming, and experiment design. These methods also cover multiple aspects of MD systems: short-range force computation, long-range force computation, bookkeeping (cell-lists), and data trans-

fer. Because of the wide range of subtopics and methods, there is perhaps no optimal organization; we have chosen to group topics by method, starting with the theoretical and platform independent, and working our way to FPGA- and finally platform-specific.
Chapter 2 presents MD in greater depth. We begin with the basic concept of MD, advanced algorithms, production codes and special purpose machines.

Chapter 3 presents HPRC in greater depth, including design flow, FPGAs appropriate for HPC, computation model for HPRC, and a survey of current HPRC platforms. We end this chapter with an overview of related work in FPGA/MD.

Chapter 4 and 5 concentrate on algorithm design. There we explain our FPGA algorithms for the two major aspects of MD: numerical computing for the short range force and the multigrid method for the long range force.

Chapter 6 describes aspects of the overall system design. It presents the architecture of the entire system and of the two coprocessors (short- and long-range) in detail, and the off-chip memory interface for supporting large simulation models.

Chapter 7 presents work related to determining validity and performance. We first describe the specific target platform on which we instantiated our designs, and then demonstrate the results to validate the simulation quality and system performance.

Chapter 8 summarizes this thesis, projects possible optimizations to the current system, and describes how to extend it to large-scale parallel systems.

Chapter 2

Molecular Dynamics

2.1 Overview

MD is one case of the N-body problem, i.e., the application of Newton's law so as to integrate particle motion with inter-particle forces as computed with force models. Although the basic theory is straightforward, MD is a computationally intensive problem because simulations are invariably of as many particles as possible. Accelerating MD has been studied for decades in both theoretical and computer engineering areas. In this chapter, we first give a brief introduction of MD, followed by a description of some well-known fast algorithms. Then, we give a survey of production MD software packages and special purpose machines for MD.

2.2 Basic Methods

MD is a set of algorithms for simulating interactions among atoms and molecules with force fields and motion integrators. Improvements in computer technology and in methods of numerical computing have extended MD applications until the can be applied to disparate domains, including theoretical physics, material science, computational chemistry, and computational biology. In biomolecular research, MD is a critical approach in translating structural information into its underlying mechanisms [Ke99]. Biomedical applications of MD [SSB+99] include modeling ligand/receptor complexes, protein folding, protein/DNA interactions, and high-density lipoprotein aggregates.

In this thesis, we limit our scope to classical MD methods, especially those appropriate for biomolecular simulations. Classical MD, unlike the Monte Carlo methods or Discrete

Event Simulation MD (DMD), generally proceeds iteratively, alternating between a force computation phase and a motion integration phase over many time steps.

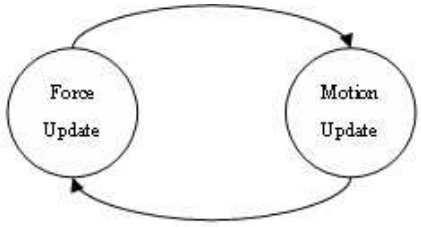


Figure 2·1: Two Phases of MD

In the force computation phase, given the set of particle coordinates, the forces on each particle are evaluated as a function of the interactions with the other particles. Then the overall acceleration is computed with Newton's Second Law. In the motion integration phase, MD updates particles coordinates by integrating the computed acceleration with the particle's current coordinates and velocity. The new coordinates are then the inputs to the force computation phase in the next time step. The following pseudo-code shows this process, in which the leapfrog (Verlet) integrator is applied:

```
For step = 1 to number of steps
Begin
        Update velocity by a half time step
        Update position by a full time step
        Evaluate acceleration on every particle
        Update velocity by a half time step
End
```

The force field components for biochemical simulations commonly belong to one of two categories: bonded and non-bonded forces.

$$\vec{F} = \vec{F}^{bond} + \vec{F}^{angle} + \vec{F}^{diherdral} + \vec{F}^{improper} + \vec{F}^{Lennard-Jones} + \vec{F}^{Coulomb} + \vec{F}^{H-bond} \quad (2.1)$$

The bonded forces are related to covalent bonds and including bond, angle, dihedral,

and improper forces. Because the number of covalent bonds of an N particle simulation model is $O(N)$, the computational complexity of the bonded forces is $O(N)$ for every time step. The non-bonded forces include Lennard-Jones (van der Waals), Coulomb, and hydrogen bond forces. Since the interaction of non-bonded forces is in general between any two atoms, the computational complexity is $O(N^2)$. Even with optimizations that bring the complexity down to $O(N)$ or at least $O(N \log N)$, this is the most computationally intensive part. The Lennard-Jones and the Coulomb forces are computed with the following equations:

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

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

where ϵ_{ab} and σ_{ab} are parameters related to particle types.

By inspecting these equations, it is apparent that the Lennard-Jones force converges very fast in distance. It is safe to switch (approximate) the force to zero for particle pairs distant from each other. The computational complexity consequently reduces to $O(N)$. For the Coulomb force, however, simulations still lose accuracy even when applying a long cutoff. Many algorithms have been invented to accelerate the Coulomb force evaluation and still to maintain accuracy; these include Ewald Sums and its variants, the fast multipole method, and the multigrid method.

For motion update, Leapfrog/Verlet [Ver67, Ske99] algorithms are the most common integrators. All forces are evaluated at the same frequency, as are the particle coordinates and velocities. This type of algorithms is called Single Time-Stepping (STS). For solvated biological molecules, the fastest motion is about 10fs [Mat04], whereas Leapfrog needs at least a time step of 2.5fs to keep system stable. Compared with force evaluation, motion integration itself does not cost much computation. Advanced integration algorithms, however, can improve MD performance by reducing the frequency with which forces must be

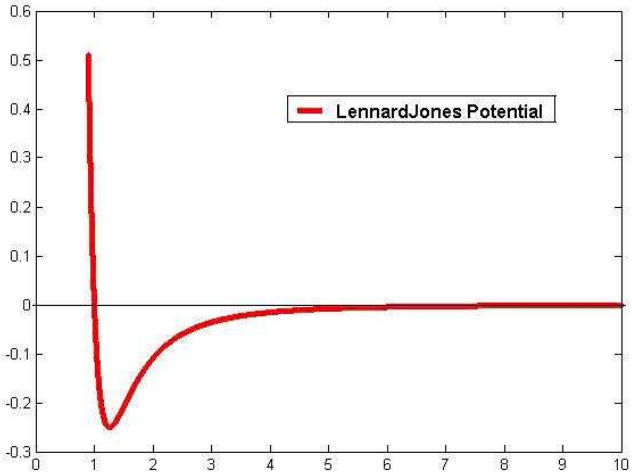


Figure 2·2: Lennard-Jones Potential

evaluated.

Boundary conditions, which define the simulation surroundings, are another important issue. Use of periodic boundary conditions, which were first applied in crystallographic research, is quite common. It constructs an infinite simulation model by duplicating a unit cell in all dimensions. Each particle interacts not only with particles in the unit cell, but also with those in the image cells. Since all image cells are identical, only the particles in the unit cell need to be updated. The method of Ewald Sums works only under periodic boundary condition.

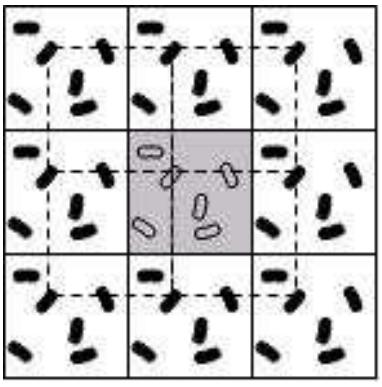


Figure 2·3: Periodic Boundary Condition

Stochastic boundary conditions are an alternative, especially in systems consisting of macromolecules in water. The idea is to apply approximation instead of explicit calculation of the effect from the water molecules far way from the macro molecules, and to randomize unwanted regularities that accumulate because of the "tile" structure artificially imposed on a smooth irregular universe. Simulation models are usually in spherical shape as shown in Figure 2·4.

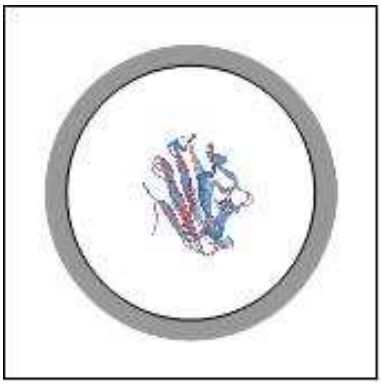


Figure 2·4: Stochastic Boundary Condition

Macromolecules are simulated in the inner spherical space filled with water by using the conventional MD method. In the gray buffer region, the molecular dynamics are computed with the Langevin equations of motion. These two layers are put in a heat bath shown as the rectangle box to keep the system at thermal equilibrium. Water molecules are restricted and evenly distributed in the system by a spherical boundary potential.

One final point: MD has its limitations. For most models, the simulation becomes chaotic almost immediately. Thus MD does not model reality so much as sample it. For application of large molecules, MD errors should be no more than those introduced by approximations in the computation model. Simulations ultimately must be validated with wet-lab experiments

2.3 Fast MD Algorithms

MD simulations usually take considerable amount of time, weeks and months, on supercomputers. Even equipped with advanced parallel computers, large models or long durations still require ever more computing power. Fast MD algorithms have been developed to accelerate simulation while retaining reasonable accuracy. Because of Amdahl's law, most fast MD algorithms focus on two issues: the non-bonded force computation and the motion integrator. In this section, we give a brief review of these algorithms. A major research issue for this work is determining which ones are preferable in HPRC systems.

2.3.1 Non-bonded Force Evaluation

Cell Lists

Of the two non-bonded forces, the van der Waals, especially as modeled by the Lennard-Jones potential, can safely be switched to zero when the distance between two particles is beyond a certain cutoff. Although the Coulomb force in general cannot be switched to zero a simple trick makes this partially possible: the force is simply split into two components, a short-range that converges on the order of the Lennard-Jones, and the remainder, which is necessarily long-range. The long-range component is handled with the long-range methods described in the next subsection; the short range by, for each particle, inspecting particles only in the nearby region. This method of Cell Lists [AT90] is an efficient way to reduce computation.

The cell-list method partitions the simulation space into cubic cells. As shown in Figure 2·5, if the cell size is larger than the short range force's cutoff, r_{cut}, then only forces on particles from same cell or adjacent neighboring cells are necessary to compute. If the cell size is small, we then need to check more neighboring cells to cover the cutoff. This makes the cell-list complex, but better approximates the cutoff sphere and thus saves computation. Cell-lists are constructed after new coordinates are calculated during the motion update phase. For software implementations, particles in each cell are linked in a list and the cell-list is represented as an array of such lists. This data structure is dynamic among time

steps and usually not efficient for hardware implementation. We have developed an FPGA oriented cell-list implementation which we describe in Section 6.2. The cell-list information is also useful for parallelization using spatial decomposition and in mesh- and grid-based algorithms.

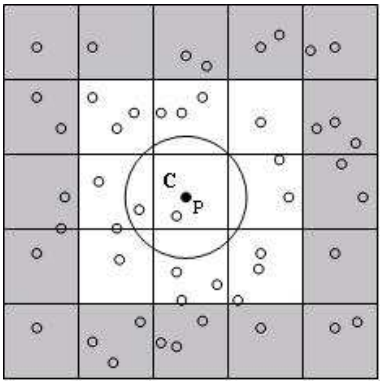


Figure 2·5: Cells in Two Dimensional Space

Multipole Method

Hierarchical tree based algorithms can accelerate the evaluation of long-range non-bonded forces. A similar requirement was invoked by the gravitational N-body problem. The basic idea is to "reduce the cost further by computing the force with errors that are comparable to those introduced by temporal discretization" [STH02], because the potentials of distant particle groups can be computed with a rapidly convergent power series with limited error. When particles are close to each other, forces must be evaluated with original equations. Two similar and related algorithms were proposed in mid-1980's: the Barnes-Hut (BH) and the Fast Multipole Method (FMM). These two algorithms are both based on multipole expansion. For the Coulomb force, the potential at point z caused by charges $\{q_i\}$ at $\{z_i\}$ can be expanded as:

$$\phi(z) \;=\; Q \log z + \sum_{k=1}^{\infty} \frac{a_k}{z^k} \tag{2.4}$$

where $Q = \sum_{i=1}^{N} q_i$ and $a_k = \sum_{i=1}^{N} \frac{-q_i z_i^k}{k}$.

For any $P \geq 1$,

$$\left|\phi(z) - Q\log z - \sum_{k=1}^{p} \frac{a_k}{z^k}\right| \leq \alpha\left|\frac{r}{2}\right|^{p+1} \leq (\frac{A}{c-1})(\frac{1}{c})^p \tag{2.5}$$

where r is the boundary of the charges, $|z| \leq r$ and $c = \left|\frac{z}{r}\right|$, $A = \sum_{i=1}^{N} |q_i|$, and $\alpha = \frac{A}{1-|r/z|}$.

BH constructs an oct-tree structure to describe the charge distribution in the 3D space. Based on this oct-tree, it is easy to distinguish close charges and remote charges. If N particles are of uniform distribution in the simulation box, then–because each leaf node has one particle–the height of this oct-tree is $\log N$, and the complexity of the tree construction is $O(N \log N)$ This is also the complexity of full BH algorithm.

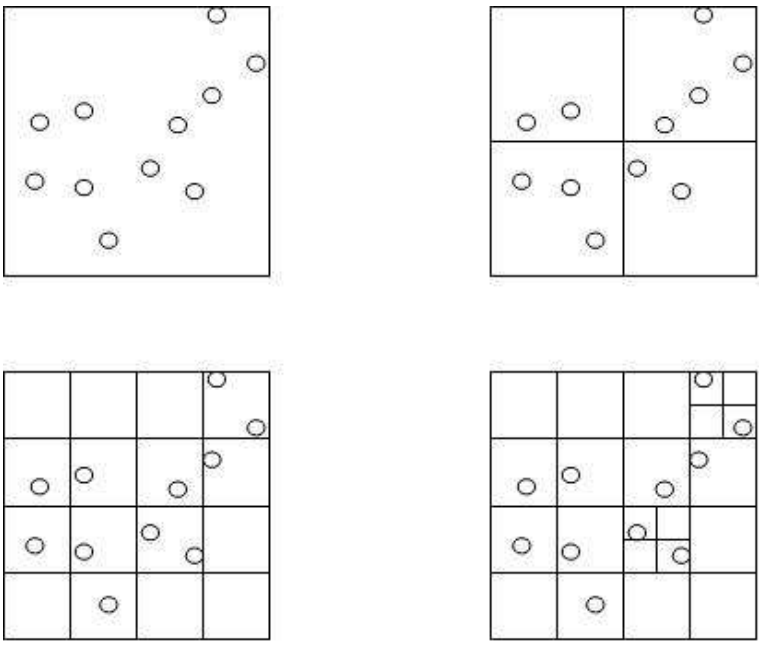


Figure 2·6: Barnes-Hut Space Splitting

FMM computes the multipole expansion for each charge with a hierarchical structure. The simulation box is divided into cubic sub-boxes on each level from the coarsest to the finest. FMM starts from the finest level, where the multipole expansion of potential field due to the local charges is computed for each sub-box. On a coarse level, the same expansions are computed with the data from the next fine level. This process propagates

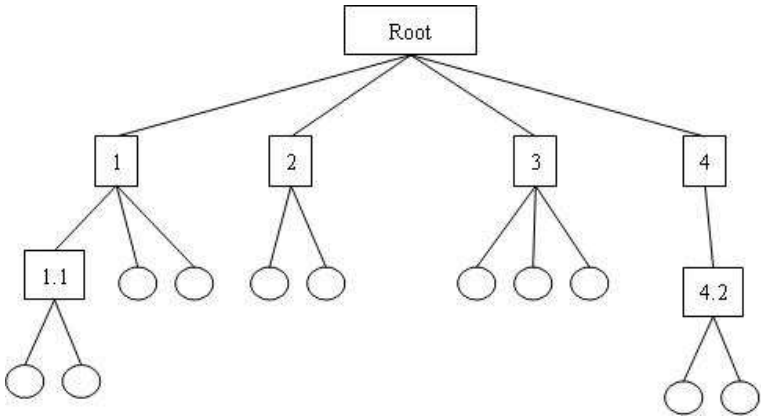


Figure 2·7: Barnes-Hut Tree

the charge distribution information farther and farther till the coarsest level, where all charges are in one sub-box. The second half of the process traverses from the coarsest level to the finest. The multipole expansions caused by distant charges are computed by gathering information from distant sub-boxes. Because of the hierarchy structure, this time only adjacent neighboring sub-boxes need to be counted on each level. The complexity of this process is only $O(N)$.

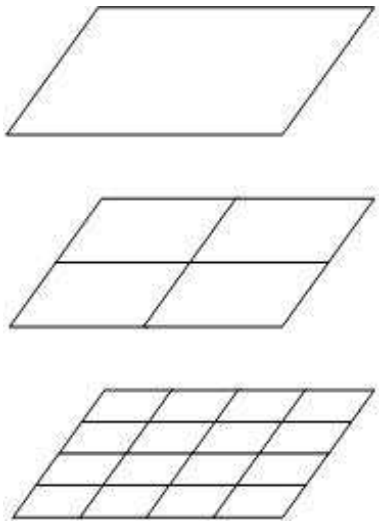


Figure 2·8: Multiscale Calculation for Multipole

Experiments reported in [GS02] compared these methods under both vacuum and periodic boundary conditions. In vacuum system simulations, BH has better performance for

'low-precision' applications, where the force error is 1%. For 'high-precision' applications that are tolerant of 0.1% force error, BH is even slower than naive all-to-all method, while FMM has better performance for simulation models of more than 10^5 particles. In periodic system simulations, BH is still faster than all other methods including Ewald Sum method with 'low-precision,' and FMM is expected to have better performance with 'high-precision' and large simulation models of more than 5×10^4 to 10^5 particles.

Ewald Sums

The method of Ewald Sums is an algorithm to compute the long range Coulomb force in a periodic system. Assuming the unit cell is cubic, the Coulomb potential energy on particle i can be expressed as Equation 2.6 and it is obvious that E_i converges slowly with r_{ij}.

$$E_i = \frac{1}{2}\sum_{j=1}^{N}{}'\sum_{n\in Z^3}\frac{q_i q_j}{|r_{ij}+nL|} \tag{2.6}$$

where $'$ means excluding particle i itself in its original cell; L is the length of the unit cell.

The Ewald Sum method applies a spherical Gaussian positive charge cloud and a negative charge cloud of same distribution to split E_i into two parts: a real part and a reciprocal part. The real part is fast converging in real space and safe to truncate at a cutoff; the reciprocal part is also fast converging, but in reciprocal space. The Coulomb potential energy of this system can be expressed as:

$$E = E^{(r)} + E^{(k)} + E^{(s)} \tag{2.7}$$

$$E^{(r)} = \frac{1}{2}\sum_{i,j=1}^{N}{}'\sum_{n\in Z^3} q_i q_j \frac{erfc(\alpha\,|r_{ij}+nL|)}{|r_{ij}+nL|} \tag{2.8}$$

$$E^{(k)} = \frac{1}{2L^3}\sum_{k\neq(0,0,0)}\frac{4\pi}{k^2}e^{-\frac{k^2}{4\alpha^2}}\left|\rho(\vec{k})\right|^2, \rho(\vec{k}) = \sum_{j=1}^{N} q_j e^{-i\vec{k}\vec{r_j}} \tag{2.9}$$

$$E^{(s)} \;=\; -\frac{\alpha}{\sqrt{\pi}} \sum_i q_i^2 \tag{2.10}$$

where $E^{(r)}$ is the real part, $E^{(k)}$ is the reciprocal part and $E^{(s)}$ is a constant self correction. α is called Ewald parameter.

With optimal α, both real part and reciprocal parts have complexity $O(N^{3/2})$. A bigger α makes $E^{(r)}$ converges faster, but $E^{(k)}$ slower, and *vice versa.* The optimal α is given by the equation:

$$\alpha \;=\; \sqrt{\pi} \left(\frac{T_{real}}{T_{reci}} \cdot \frac{N}{V^2} \right)^{\frac{1}{6}} \tag{2.11}$$

where T_{real} and T_{reci} are the time to compute the real part and reciprocal part respectively. The cutoff of the real part is:

$$r_c \;=\; \frac{\sqrt{\pi}}{\alpha} \tag{2.12}$$

The cutoff of the reciprocal part is:

$$k_c \;=\; 2\alpha\sqrt{P} \tag{2.13}$$

where P is the required precision:

$$P = \ln \epsilon \tag{2.14}$$

where ϵ is the tolerable error.

Particle Mesh Ewald

Particle Mesh Ewald (PME) is an efficient algorithm for evaluating Ewald Sums, and is presented in Darden, et al. [DYP93]. The basic idea of PME is to accelerate Ewald Sums by choosing a large α in Equation 2.8 to reduce the complexity of the real part of the Ewald Sum to $O(N)$. In turn, with the mesh method, it computes the reciprocal part in

complexity of $O(N \log N)$.

There are three basic steps in PME:

1. Assign particles' charge to mesh points
2. Compute energy/force with FFT
3. Interpolate the results back to particles

The complexity of step one and step three are both $O(N)$, step two is $O(N \log N)$ and dominates the overall complexity.

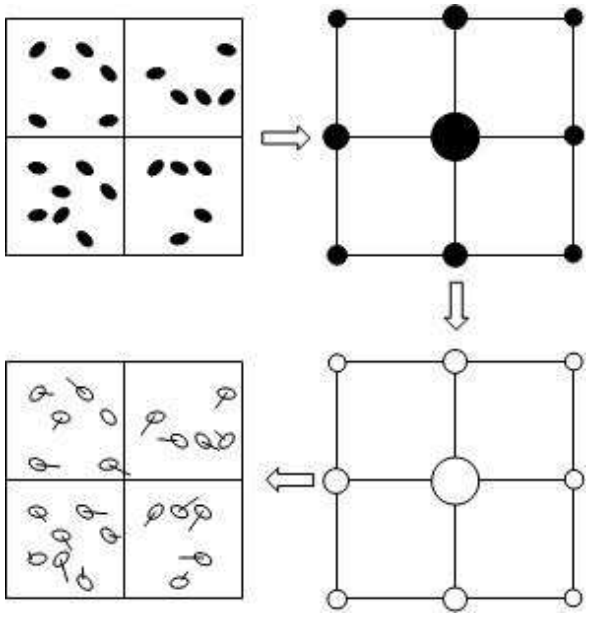


Figure 2·9: PME Steps: the first arrow is charge assignment, the second potential computation, and the third force interpolation)

The original PME uses Lagrange interpolation for charge assignment and force interpolation. Because the Lagrangian weight function is only piece-wise differentiable, energy and force should be computed with the same process but separately, and the energy is not conserved. An improved method, Smooth PME (SPME) is presented by Essmann et al. [EPB+95]. Here, a B-spline interpolation is applied instead of a Lagrange. The most important advantage is that force can be computed by differentiating energy directly, and energy is conserved.

Multigrid Method

Many important computations, from solving systems of equations to solving partial differential equations (PDEs), can be executed by discretizing to a grid and iteratively performing operations in all neighborhoods. Multigrid algorithms improve the convergence rate of basic finite difference methods by using a hierarchy of discretizations, often reducing complexity to $O(N)$ in the number of grid points. Multigrid is of particular interest here because it is also applicable to the long range Coulomb force computation.

There are basically two categories of multigrid methods for Coulomb force. One is presented by Skeel et al. [STH02], and uses a final direct computation. Since the final computation uses the electrostatic equation, which means we have the solution of the PDE from charge distribution to potential distribution, what we save is the computation cost. The limitation is that it only works well for vacuum boundary condition. An extension to the periodic boundary condition was proposed in [IHM05]. The second category is to solve the reciprocal part of Ewald sum numerically with multigrid, such as is presented by Sagui and Darden [SD01]. In this case, the solution of the PDE from charge distribution to potential distribution is unknown, but can be approximated with the multigrid method. They follow the general multigrid method, doing relaxation and correction at every level. Because it is based on Ewald Sums, this method only works for periodic boundary condition.

In this research, we have implemented the first multigrid method. More detail about the original method and our FPGA algorithm is presented in Section 5.2.

2.3.2 Long Time Step Integrator

The longer the time step, the more physical time can be simulated per computation. Although it is hard to prolong the time step for entire system, it is still possible for some forces that are less sensitive to the particles' position than others.

The Multiple Time-stepping (MTS) integrators evaluate different forces with different frequencies. One of the typical MTS integrators is Verlet-I/RESPA/impulse, which splits

the overall force into several components and every component's dynamics corresponds to a different timescale. These components are evaluated at different frequencies and summarized for motion update. Formally, it can be represented with impulse functions as in the following:

$$M\frac{d^2}{dt^2}X \;=\; -\sum_{i}\sum_{n'=-\infty}^{\infty} \delta t\vec{\delta}(t - n' \triangle t) \bigtriangledown U^i(X) \tag{2.15}$$

The potential energy is split into several components, U^i, and is sampled with the Dirac delta function with different time steps. The time steps of slower forces are chosen as multiples of faster forces, so that the algorithm can be implemented as several nested loops. The fast forces are evaluated in inner loops and slow forces are in the outer loops. An example of forces splitting is: the bonded forces are the fast forces; forces in range of a certain cutoff are medium forces; and the remaining are regarded as slow forces. If the faster forces are calculated with time step t, then the medium forces time step is $k_1 t$, and slow force time step is $k_2 t$. The ratio $k_1 k_2$ always determines the overall speedup because most computation is spent on the slow forces [BS98]. Typically, t is 0.5 fs, k_1 is 2 or 4, and k_2 is 8.

Further analysis of the Verlet-I/r-RESPA/impulse algorithm in [GASSS98] reveals that simulations may yields resonance when the outer time step is nearly equal to the period of a fast oscillation. Worse, resonance occurs too when the time step is just less than the half of the shortest period of the fast oscillation. In biomolecular systems, the first resonance effect appears at 5 fs and is called "The Five Femtosecond Time Step Barrier" [SI98].

An enhanced impulse integration algorithm, the mollified impulse method (MOLLY), is proposed in [GASSS98]. MOLLY replaces the slow potential energy and forces as follows:

$$U^{slow}(X) \;\rightarrow\; U^{slow}(A(X)) \tag{2.16}$$

$$F^{slow}(X) \;\rightarrow\; A_x(X)^T U^{slow}(A(X)) \tag{2.17}$$

where $A(X)$ is a time averaging of vibrational motion due to fast forces; $A_x(X)$ is a Jacobian matrix.

$A_x(X)^T$ can be regarded as a filter that eliminates components of the slow force impulse in the direction of the fast forces [IMM+02]. [SI98] reports that MOLLY achieves 5 fs for the outmost level time step with 3 time averaging methods. MOLLY has been tested with ProtoMol and integrated into NAMD2.

Constrained Dynamics, such as SHAKE/RATTLE methods [BPGH81], is another type of integrator under research; this constrains the vibration of bonds. The LN approach for Langevin Dynamics extrapolates the slow forces rather than sampling with an impulse, and in some simulations it extends the period of slow force evaluation to be 48 fs or more [BS98].

2.4 MD Software Packages

There are dozens of MD packages currently in use, some of which have been developed and optimized for more than a decade. Since our research only accelerates part of MD, we must necessarily integrate our coprocessors into existing codes to build a complete MD system. The following is a brief survey of four MD packages: Three, NAMD, GROMACS, and AMBER, are by far the most popular systems; ProtoMol is highly significant because it is optimized for extensibility, and so ideal for use in accelerator research.

2.4.1 NAMD2

NAMD2 [Ke99, Phi05] is a scalable MD package running on a wide range of machines from PC to high-end parallel computers. It is object oriented and implemented in Charm++ (not to be confused with CHARMM; see below).

NAMD2 employs both Spatial Decomposition and Force Decomposition to obtain scalability. The Spatial Decomposition is implemented with cell-lists. The size of cells is larger than the cutoff of forces by a distance annotated as margin. When a particle moves from its old cell to a new one, it is not necessary that it be transferred to the new cell immedi-

ately. Thus, it is not necessary to update the cell-lists every time step. Actually, NAMD2 generates new cell-lists only every 4 to 8 time steps. This method substantially reduces communication cost in parallel implementations.

During simulation, the change in particle spatial distribution requires the system to be rebalanced. Force Decomposition allows the non-bonded force computation to migrate from busy processors to idle ones dynamically. Since the bonded forces take small portion of computation, they do not migrate.

There are therefore two types of load balancing are employed in NAMD2: initial and dynamic. Initial load balancing is done during program startup. From then on, NAMD2 performs the dynamic and measurement-based load balancing by migrating non-bonded force computation objects. NAMD2 has achieved significant efficiency, for example, close to 80% on 128 processors.

2.4.2 ProtoMol

ProtoMol [Mat04] is an MD framework developed in C++. It utilizes inheritance and design patterns of object-orientated programming and is designed especially for experimentation with novel MD algorithms.

The parallelization on ProtoMol is relatively primitive. The parallel version of ProtoMol is modified from the serial version with limited changes. It decomposes the parallelism with force group, which assigns computations to different processors by force type. Consequently, all particle data are duplicated on every processing node. This method is called Replicated Data. The communication cost of each processor in a Replicated Data system is proportional to number of particles. This method is reported to work well with up to tens of processors. ProtoMol has similar performance to serial versions of leading MD packages, such as NAMD2.

2.4.3 CHARMM and AMBER

CHARMM [He94, PZK02] has been under development since 1983 and is coded in FORTRAN. CHARMM partitions the computation of the $O(N^2)$ pair-wise non-bonded forces with Force Decomposition, which means that computation is partitioned in units of single force evaluation. Because no spatial information is applied, neighbor lists do not help in this case. Assuming there are P processors, the communication cost of each processor is $O(N/\sqrt{P})$.

CHARMM minimizes the communication and achieves load balancing with irregular partitioning. An example of a partition across 4 processors is shown in Figure 2·10.

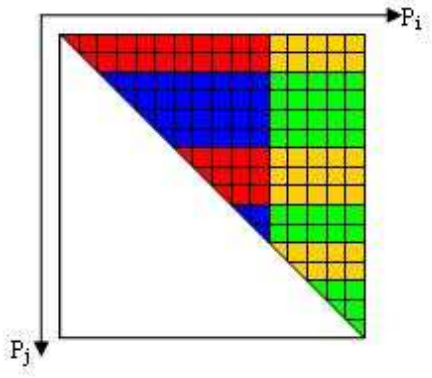


Figure 2·10: Irregular Particle Pairs Partition Applied by CHARMM [He94]

AMBER [CCD+05] shares many characteristics with CHARMM. It was developed in FORTRAN, and is regarded as the "standard" MD code. It improves simulation accuracy by adding new force terms. CHARMM and AMBER have their own force fields. For parallelization, AMBER only applies the Replicated Data method.

2.4.4 GROMACS

GROMACS was developed from GROMOS by rewriting routines from FORTRAN to C. It was originally designed for parallel computer systems. GROMACS has highly optimized codes, such as converting some floating-point operations to integer, supporting SIMD instructions, and assembly level optimization. It is reported to be the fastest MD package

to date; however, it may not be best one in terms of scalability. GROMACS enables parallelism with Message Passing Interface (MPI) standard functions on cluster systems and shared memory super computers. The communication cost is a major barrier preventing GROMACS from scaling to a large number of nodes, although it has significant single node capability.

GROMACS decomposes particles to nodes connected in a ring structure. At every time step, all the coordinates and partial forces are sent around half of the ring, so that for particle pair (i,j), either 'i' is sent to the node of 'j' or 'j' is sent to the node of 'i' to calculate pair-wise forces; then, the results are sent back in the opposite manner for each particle to sum up. As stated in [KSF+07], the all-to-all communication causes the network to lose packets in regualr Ethernet switched clusters, and breaks down the scalability when the number of nodes is greater than two. One solution to this issue is to employ expensive networks with good QoS (high bandwidth and low latency); other solutions include applying link-layer flow control on the cost-efficient Gigabit Ethernet network and explicitly controlling communication pattern. With these methods, the scalability is extended at least to 16- or 32-node systems.

2.5 Special Purpose Machines

The importance of MD and the regularity of its computations have inspired some work in special-purpose architectures. Most of these were originally developed for N-Body computations.

2.5.1 MD-GRAPE

MD-GRAPE is a subset of the GRAPE (GRAvity PipE) family of special-purpose computers. The latter were originally designed for gravitational simulation and were extended to other N-body problems, including MD [KUT+97]. GRAPE chips work as coprocessors connected with a host computer via a bus adaptor.

GRAPE chips are designed to compute the most intensive $O(N^2)$ long range inter-

Figure 2·11: System Level Block Diagram of MD-GRAPE [FTM+96]

particle forces in a general form:

$$\vec{F_i} = \sum_j a_j g(b_j r_s^2) \vec{r}_{ij} \tag{2.18}$$

where r_{ij} is the geometry displacement between particles, r_s is a scale factor, and a_j, g, b_j are parameters for specific forces.

The host computer does the rest of computation. Original GRAPE, however, could only compute forces in form of $1/r^2$, which is not the case of Particle-Particle Particle-Mesh (P^3M). Several modifications were applied on GRAPE, described in [FTM+96, EMF+93, TNO+03, JVS03], to build MD specific GRAPE chip, i.e. MD-GRAPE and MD-GRAPE2. Because they only computed the real part of Ewald Sum, WINE [FMI+93] and WINE2 [NSFE00] were developed for the reciprocal part of Ewald Sum. MD-GRAPE2 and WINE2 were integrated into the Molecular Dynamics Machine (MDM) [NSE+99, NSK+00], where particle data are duplicated on each MD-GRAPE2 board and WINE2 board. MD-GRAPE3 [TNO+03] integrated the real part and reciprocal part computation as well as particle data memory into one chip, and will be applied in the Protein Explorer machine. A block diagram of the MD-GRAPE3 chips (from [TNO+03]) is shown below.

The amount of communication between the host and MD-GRAPE3 is $O(N)$. At the same time, the computation performed on MD-GRAPE3 is proportional to $O(N^2)$. In the Protein Explorer system, the ratio between the communication speed and the calculation speed is 0.25 bytes per one thousand operations, so the communication/computation speed gap is not a big problem.

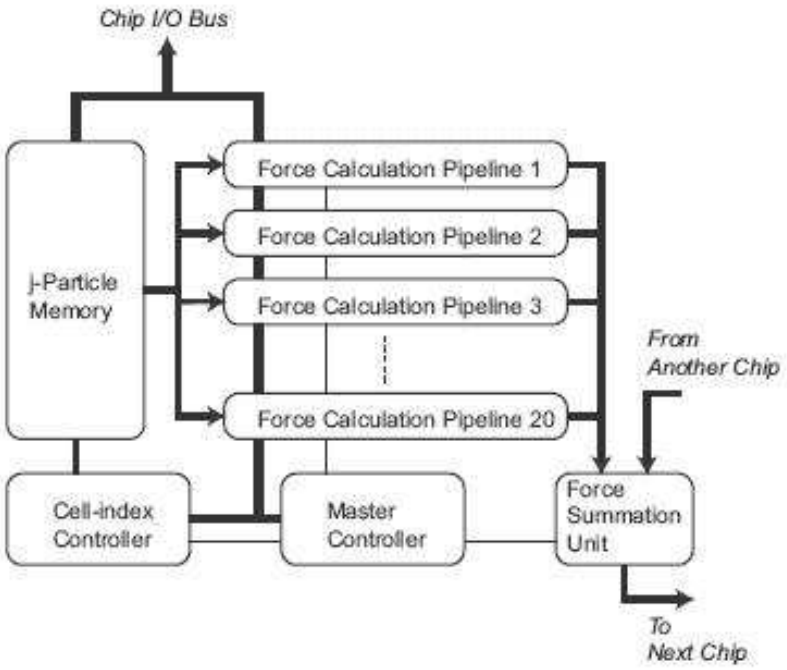


Figure 2·12: MD-GRAPE3 Chip Block Diagram [TNO+03]

2.5.2 MD Engine and MODEL

MD Engine is a hardware accelerator for non-bonded forces computation. The system presented in [TMK+99] contains a Sun Ultra-2 workstation, 76 MD Engine cards, and achieved a 48 times speedup over a single Sun Ultra-2 workstation. The acceleration is performed by the MODEL chips on each MD Engine card, which is able to compute the non-bonded forces as well as the original Ewald Sum method in a general purpose pipeline. The MD Engine cards hook onto a single bus to communicate with the workstation. It is convenient to broadcast particle coordinates and computation parameters to MODEL's local memory via this bus. During the computation, each MODEL chip can run at top speed. However, the communication after force computation in this system is not efficient.

Most of the computation within MODEL pipeline uses a 40-bit floating-point format. MODEL also uses quadratic interpolation to compute nonlinear functions with coefficients from its function memory. To maintain accuracy, different correction methods are applied

to the Lennard-Jones force, the Coulomb force, and the reciprocal part of Ewald Sum individually. This means that although MODEL has a general force pipeline, it is still hard, if not impossible, to adapt new forces.

Chapter 3

FPGA Acceleration of MD

Nowadays, cutting edge FPGAs operate as fast as 500MHz, have more than 200,000 general logic cells, and many dedicated function units such as memories and ALUs. With such large capacity FPGAs, high performance computing (HPC) on FPGAs has been explored in many ways, including floating-point arithmetic, linear algebra, communication, and signal processing. Recently, FPGAs have been employed in supercomputers, such as the Cray XD1, SGI Altix RASC, and SRC MAP station, to work as configurable coprocessors. The latest trend is pin compatible FPGA cards, such as from DRC, XtremeData, and Nallatech, that plug directly into the processor socket on the processor board. This trend of growing support for FPGAs confirms our belief that these devices are promising high-end computation resources, and that research into HPRC is both more feasible and critical than ever.

3.1 FPGA Overview

FPGAs are one type of reconfigurable circuits. Other reconfigurable circuits include Programmable Array Logic (PAL), Generic Array Logic (GAL), and Complex Programmable Logic Devices (CPLD). Among these devices, FPGAs provide the largest number of gates and are the most powerful in terms of ability to implement logic functions.

FPGAs are made of programmable components and programmable connections. The programmable components are able to implement combination logic to translate small scale inputs to outputs with a lookup table (LUT). Storage units, such as flip-flops (FF) are also available in the programmable logic fabric and are necessary to build sequential logic. The programmable connections interconnect the programmable components to combine small

scale functions into large scale functions; the hierarchical structure retains low propagation delay for signals traveling across the chip. As shown in Figure 3·1, the Xilinx Virtex II FPGA chip [Xil03] contains configurable logic blocks (CLBs) as the programmable components. The LUTs in the CLBs are usually implemented with SRAMs, they can therefore be used as normal SRAMs as well. As shown in Figure 3·2, CLBs are connected to the Switch Matrix for global communication; neighboring CLBs are connected with express lines.

Besides these configurable components, modern high-end FPGAs have many dedicated function units and peripherals implemented in custom logic; these include on-chip SRAMs (called Block RAMs or BRAMs), DSP modules (multiply and add units), fast I/Os, and multiple clock control units. Some FPGA families, such as Xilinx Virtex II and Virtex IV, have multiple embedded PowerPC processors, as shown in Figure 3·1. Although the connections among these dedicated function units and CLBs are reconfigurable, these connections are somewhat less flexible; these dedicated components still have much better efficiency than equivalent logic built out of the small-scale configurable components. High-end FPGAs also have hundreds of I/O pins. These are commonly used to integrate FPGAs with off-chip but on-board memory. A typical board-level FPGA system contains several independently addressable SRAM and DRAM banks for use as backing store or user-managed cache.

FPGAs have traditionally been used for glue logic on printed circuit boards (PCBs) to merge signals among chips, or to provide board level registers. One advantage of FPGAs in this scenario is that they can be modified even after the board design has been committed. Another traditional application for FPGAs is for IC prototyping. FPGAs are used to emulate chips?logic before they are fabricated. In a recently published project [LYS07], an Intel Pentium microprocessor was emulated with a Xilinx Virtex IV LX200 FPGA. The advantage over, say, software emulation, is efficiency: FPGA-emulation enables full system simulation including test with run real operating systems, such as Windows XP. Another application is for situations where the expense of developing an ASIC would not

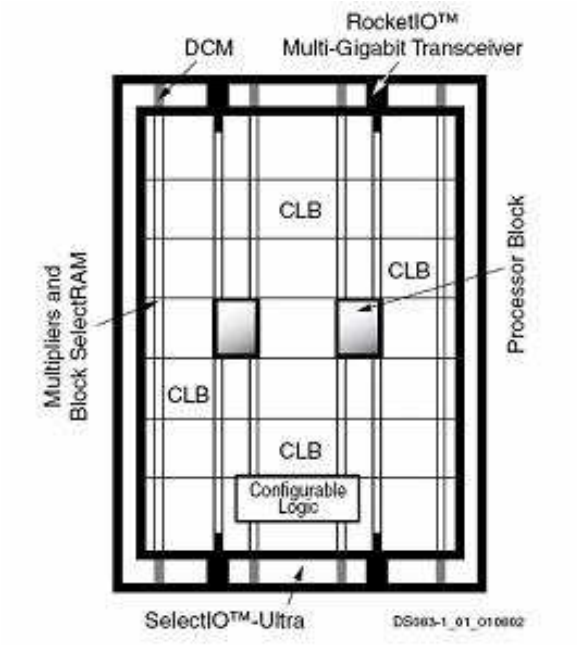


Figure 3·1: Virtex-II Pro Generic Architecture Overview [Xil03]

be justified: here FPGAs provide a direct and cost-effective solution.

3.2 FPGA Design Flow

Generally, FPGA design involves two aspects: function description and function verification. These two parts are shown side-by-side in the design flow as shown in Figure 3·3. Function description specifies the configuration of the chip to implement certain functionality ?this is done in various abstract levels. Function verification checks whether the design matches the specification at each level. Software tools for all of these functions have a long history derived, especially from ASIC design: they are known collectively as Electronic Design Automation (EDA).

A central difference between ASIC and FPGA design for HPRC is the intended application space; this has a direct influence on the tools used to support creation of FPGA configurations. Although there is little difference between what could be implemented on ASIC and FPGA (other than that the ASIC would be more efficient), configurability gives FPGAs viability as a general purpose processor. *General purpose* in this sense does not

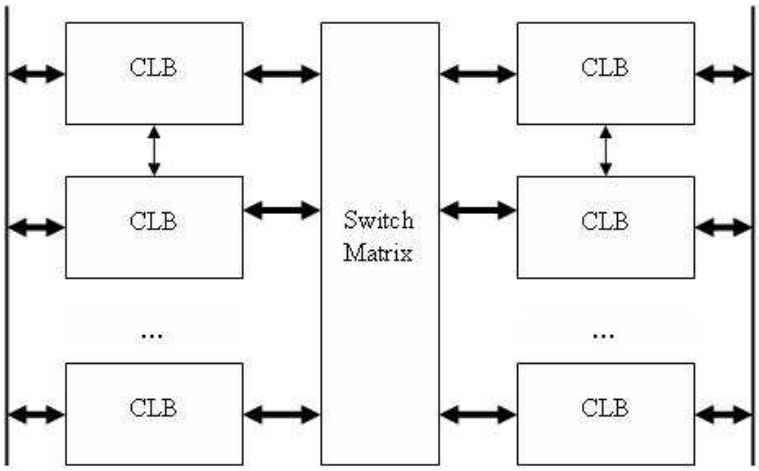


Figure 3·2: FPGA Programmable Connection

imply the flexibility of a microprocessor, rather the fact that they can accelerate many different types of applications. The expectation is that FPGAs will be cost-effective solutions in lower volume applications than ASICs, and that the designer will have less hardware design experience. This last point is enabled by the fact that the FPGA already provides much of the circuitry required for a working solution. To summarize: while much of the EDA infrastructure for HPRC is derived directly from its existing user base, the highest level, closest to the designer is not.

HPRC is still developing rapidly and there is as yet no convergence as to the best methods for describing configurations (the FPGA equivalent of programming). Furthermore, it is likely that several different modes will remain viable, depending on market size and potential mark up.

In theory, it is possible to program each CLB individually. Although this might be possible for small-scale designs for high-volume applications, this is not generally viable for HPRC. The next level is that of the hardware description language (HDL), e.g., VHDL and Verilog. These languages have a C-like syntax and when used by a skillful designer, can be used to produce highly efficient configurations. HDLs of any kind, however, are fundamentally different from high-level languages (HLLs). Whereas HLLs direct a computer to execute a sequence of instructions, HDLs describe the behavior of a circuit at all

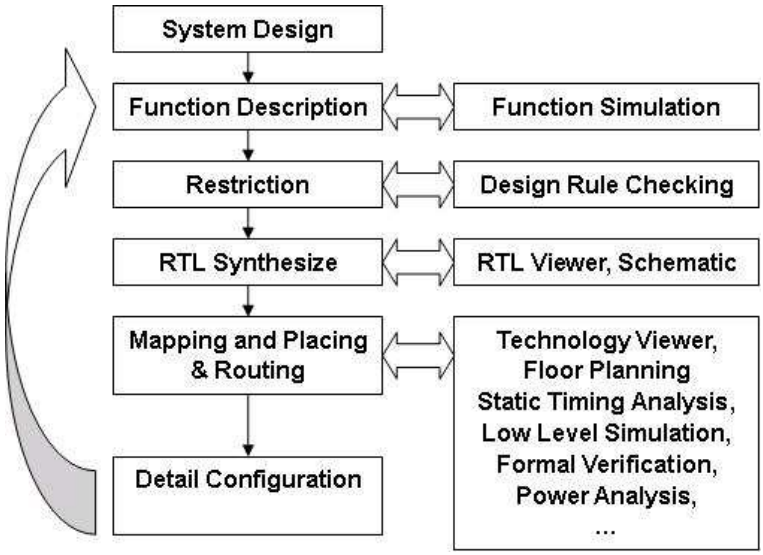


Figure 3·3: FPGA Design Flow

times with respect to stimulus. Of course there is convergence when the HDL is describing a microprocessor!

One of the fundamental problems in computer engineering today is to bridge this semantic gap between HDL and HLL. Dozens of tools have been developed to allow designers to specify functionality in more abstract manner. For example, SystemC [Gro02] uses C++ as programming interface, Matlab using Simulink [Mat06]. Users must follow the restrictions embedded in these tools to specify their requirements, which causes considerable cost of training and legacy codes recoding. Other tools, such as Impulse-C [Imp06] and SRC MAP C/Fortran [SRC06], translate programs from standard high level languages to FPGA configurations. They are, of course, convenient for users to convert legacy codes or develop new applications with little training. However, the performance of the auto-generated FPGA designs is usually not as good as other methods.

In the end, it may turn out that the choice of description method used for a particular HPRC application may matter less than how the algorithm is restructured to map to the FPGA. In this work we design explicitly with VHDL; we believe that the importance and difficulty, particular the need for high precision, warrant this level of design.

3.3 High Performance Reconfigurable Computing

Before addressing the capability of FPGAs for high performance computing (i.e. HPRC), we need to discuss the limitations of other computation resources. The computational power of a general-purpose supercomputer system is enhanced with improvements to individual nodes, the communication network, and parallelism. In recent years, the clock frequency of microprocessors has not significantly increased, because of the power density barrier. Instead, multi-core technology avoids this problem by executing multiple tasks in parallel at lower frequency. For HPC, multi-core may not be that effective, because the overhead of communication and synchronization becomes more significant when the problem is decomposed into very small granularity. On the other side, Application Specific Integrated Circuits (ASICs) of special architecture are desirable to achieve fine grained parallelism. The advantage and disadvantage of ASICs are both obvious. They are more efficient in terms of chip area and power than general purpose microprocessor because of their specialized architecture; however, ASICs are difficult and expensive to develop, and inflexible to adapt new applications.

FPGAs have characteristics of both generalization and specialization: before configuration, they are general for various architectures; once configured, they are application-specific. Although running much slower than high performance microprocessors, FPGAs can achieve better data throughput, because of application specific architecture, such as non-standard data types and customized memory interfaces. The massive connections and programmable components offer FPGAs flexible scalability and enormous parallelism. Compared with ASICs, FPGAs are commodity parts and easy to use: one FPGA acceleration platform can work for many applications.

Applying FPGAs to HPC is promising, but also challenging. Herbordt et al. [HVG+07] summarized the following major reasons. (i) The speedup is limited by Amdahl's law, i.e. the portion of problem to be accelerated with FPGAs. (ii) The operating frequency of FPGAs is slow. (iii) The overhead of parallelism affects FPGA systems as well. (iv) There

are different computation models between microprocessor and FPGAs. (v) Few application experts are good at FPGA design. Therefore, the performance of HPC using FPGAs is very sensitive to the quality of the implementation. Overhead must be scrupulously avoided in implementation, both in tools and in architectures [Sny86].

In general, FPGAs are completely different computation resources from microprocessors for HPC applications. They offer a novel approach to break through the performance barrier and still retain flexibility to address various applications.

3.4 Computation Model

Since the potential speedup of FPGA acceleration is tremendous but difficult to obtain, the development methodology becomes important. As defined for the analogous problem of developing parallel applications [SSOB02, DG04, PV96], a well defined programming model is the common platform for system designers, tool developers, and application developers to work on. By following a unique programming model, application developers can work independently, and also benefit from the optimization done by system designers and tool developers. The situation of HPRC is similar, but more complicated.

As a consequence of the challenges described previously, HPRC applications require careful design; this is problematic because application experts usually lack FPGA expertise. The methodology that merges HPC applications and FPGA development must provide models that experts on both sides can work with. The difficulty for HPRC is that there is no uniform architecture to configure on FPGA. The traditional programming models cannot help application experts use FPGA efficiently. FPGA-specific models therefore are necessary. Since FPGAs in used for HPRC are mostly configured as computation engines to process data, these models shall be computation orientated. The computation models likely to be effective for HPRC are closely related to real hardware to enable real speedup.

Stream processing is a well studied computation model. For problems having a regular computation pattern, massive data processing can be controlled by a few flow controls, and data can be re-used within data path to relieve the memory access bottleneck. Many com-

puter architecture technologies utilize these features, such as SIMD instructions (mostly multimedia instructions), systolic arrays, and GPUs. The stream processing computation model comprises a wide range of support for these kinds of problems, including stream specific languages [TKA02], stream compilers for general programming languages [MPHL03], stream architecture on general purpose processor [RAJ99, PW96], and special stream processors [KDK+01].

FPGAs by nature are appropriate stream processing processors. This is because they have programmable connection to implement various systolic array structures, programmable components to efficiently perform non-standard operations, and a flexible I/O interface to interact with external devices. Many tools based on the stream processing computation model assist application experts in exploiting speedup with FPGAs: two of these are SA-C [BHD+02] and ASC [Men06]. SA-C (Single Assignment C) extends the C language for users to annotate stream tasks. The SA-C compiler takes out these tasks and generates VHDL code. ASC (A stream compiler) extends C++ and also depends on user annotations to recognize stream tasks. ASC uses PamDC, instead of VHDL or Verilog to specify FPGA designs. Both SA-C and ASC define data types for bit operations. Mapping streaming tasks to FPGA, even those as simple as expanding a loop with a deterministic number of iterations and no data dependencies, is not trivial for most software developers. These tools help them improve performance without doing specific FPGA development or optimization, as long as their applications fit in the supported stream processing computation model.

Multigrid is another computation model good for HPRC. It perhaps not as generally applicable as the stream model, but is indicative of the idea of the family of applications as opposed to point solutions. We originally developed a multigrid coprocessor to compute the long-range Coulomb force for Molecular Dynamics simulation [GH07b]. The multigrid method is, in fact, applied in many areas, such as numerical computing and image processing. Therefore, we defined a computation model based on the multigrid coprocessor, and adapted some image processing algorithms to it. For algorithms only doing linear

operations, this multigrid model can directly generate FPGA circuits and performance estimation, and application experts only need to provide algorithm information, such as operation coefficients, data type, and number of iterations. For non-linear multigrid algorithms, because there is no general form to model non-linear operations, our computation model needs to analyze problems case by case, rather than to provide a generic solution. This fact also reveals that creating FPGA computation model is more difficult.

One or a few computation models are definitely not sufficient for all of HPRC: for many families of applications (including MD) we need meticulous implementation to squeeze out cost-effective performance. If the model is too general, it becomes trivial and useless; if it is too specific, numerous potential applications will be left behind. The advanced EDA tools described previously are often helpful, but not universally sufficient. They raise the abstraction level above that of the hardware description, so that application experts can focus on algorithms. FPGA expertise is still required, however, for essential jobs, such as optimizing common architectures and libraries, creating templates, and developing new hardware systems.

3.5 Reconfigurable Computer Systems

Several vendors have released HPRC systems in recent years. Many of them are extended from existed supercomputer systems. Although FPGAs can be configured in different ways, the FPGA chips must be plugged into PCB boards to work with memory, microprocessor, and other devices. There have been various supercomputer system architectures with specific node structure, communication, operating system, and programming model; FPGAs introduce more architecture design choices. For instance, the relationship between microprocessor and FPGA can be either processor/coprocessor, or peer-to-peer in the whole system. In this section, a brief survey introduces some of these HPRC systems.

3.5.1 SGI RASC

SGIs HPRC product is called the Reconfigurable Application Specific Computing (RASC) [Sil04]. The newest version is the RC100, a reconfigurable computer blade. RASC is based on the Altix architecture. The RASC blade can be connected to the Altix 350 system via the NUMAlink low-latency high-bandwidth bus.

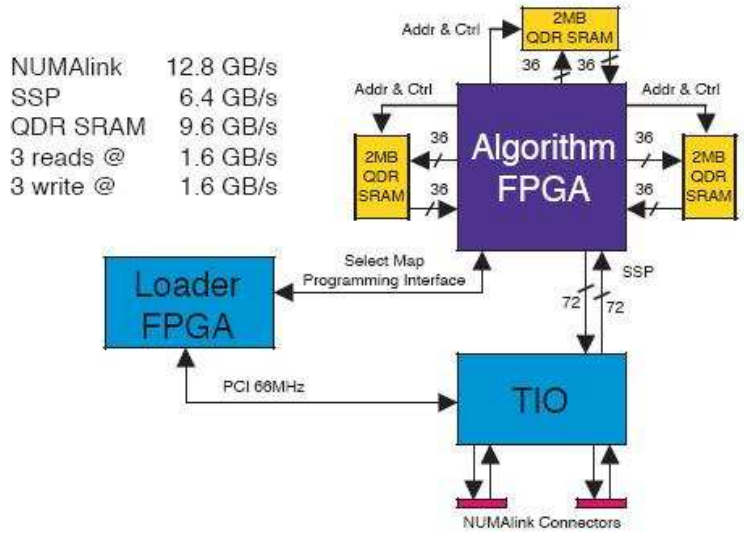


Figure 3·4: SGI's SA Brick [Sil04]

There are two Xilinx Virtex4 LX200 FPGAs on each RASC blade. As shown in Figure 3·4, each FPGA can have up to 5 banks of off-chip SRAM (up to 40MB in total). The interface to the NUMAlink is through the Scalable Systems Port (SSP) and the TIO ASIC chip, which provides a high speed (up to 3.2GB/s) and low latency access to the memory coherency domain. Furthermore, SGI allows users map virtual addresses to the FPGA memory.

As shown in Figure 3·5, SGI has a layered software architecture for HPRC. Users can use the RASC Abstraction Layer API to access FPGAs from the user space; the OS layer handles FPGA device management, such as FPGA configuration bit-file downloading and data transferring; FPGAs are in the hardware layer, which can be developed with various languages and tools, such as VHDL, Verilog, Mitrion-C, and Handel-C. The software

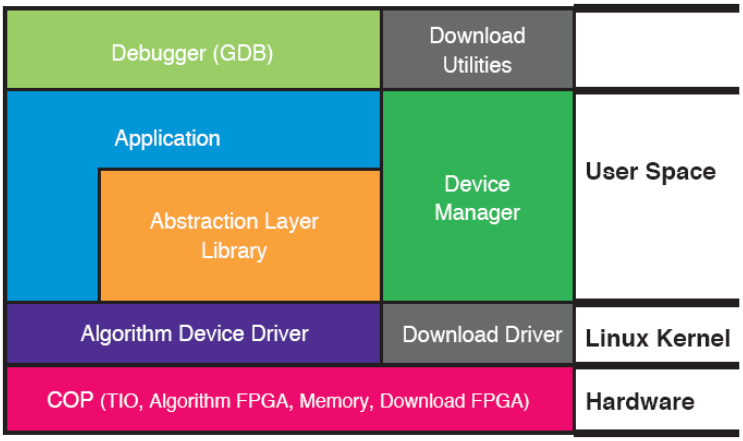


Figure 3·5: SGI Software Architecture for RASC [Sil04]

debugging support is based on GDB with FPGA-aware extension.

3.5.2 Cray XD1 and XT4

Cray integrated FPGAs as coprocessors in their message passing supercomputer. The XD1 [Cra05] was their first HPRC solution. As shown in Figure 3·6, there are up to 6 nodes in one chassis; every node has two AMD Opteron microprocessors and one Xilinx FPGA as the reconfigurable coprocessor. The connection among the FPGA, the microprocessors, and the RapidArray Interconnect System is shown in Figure 3·7. The HyperTransport bus provides two 3.2GBps connections between the FPGA and the Opteron microprocessors, and 4GBps connection from the FPGA to the RapidArray Interconnect System which connects to other nodes. Each FPGA additionally has four 3.2GBps ports to QDR SRAMs and two 2GBps ports to other computation modules. The Cray XT4 is the successor of XT3, which has newer SeaStar interconnection, Opteron Processors, DRAMs, and FPGA coprocessors. Unlike the XD1, where FPGA has dedicated socket, XT4 allows FPGA to be plugged into its processor socket and to access DRAMs directly.

Cray's message passing programming model is extended to execute computation on the FPGA as well: APIs include "fpga_put" and "fpga_get". The FPGA coprocessor itself follows the regular development flow. At the high level, high level languages or tools like

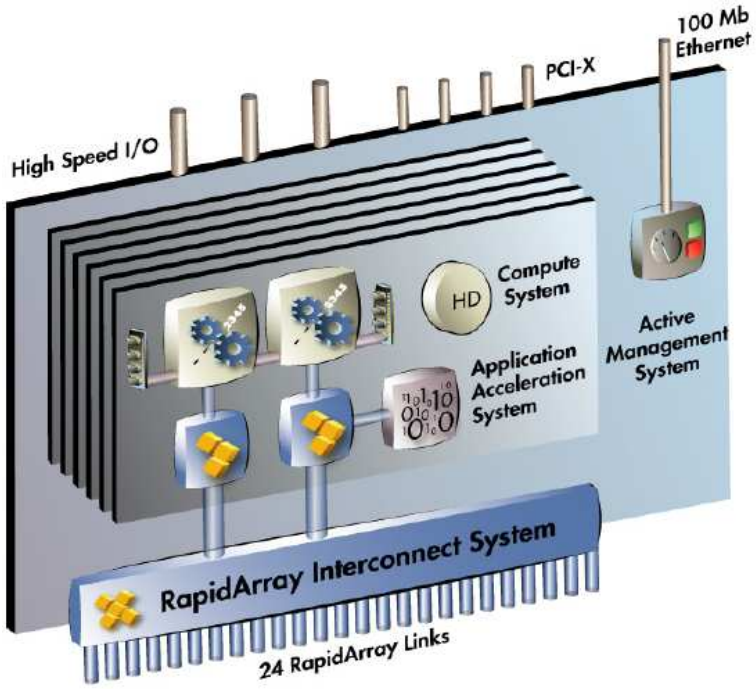


Figure 3·6: Cray XD1 Node [Inc]

MATLAB scripts are compiled to VHDL/Verilog. Then, the VHDL/Verilog codes are synthesized and mapped to FPGA chips.

3.5.3 SRC MAP Station

The HPRC solution from SRC is called MAP. The last versions are the SRC-6, which is based on Xilinx FPGAs, and the SRC-7 which is based on Altera FPGAs. A MAP node is a standalone box plugged into the main system. The MAP systems can be built on different scales: a small system contains only one MAP box plugged into a microprocessor node, which is called MAP station; a cluster based system is constructed with several MAP stations connected with Ethernet; a high-end system has several MAP boxes, microprocessor nodes, and common memories connected with the SRC Hi-Bar Switch bus.

Each MAP box contains two FPGAs as well as on-board SDRAMs and SRAMs. The interface to the microprocessor is up to 14.4GB/s on the SRC-7 and 2.8GB/s on the SRC-6. The general purpose I/O (GPIO) pins allow MAP-to-MAP or peripheral-to-MAP direct

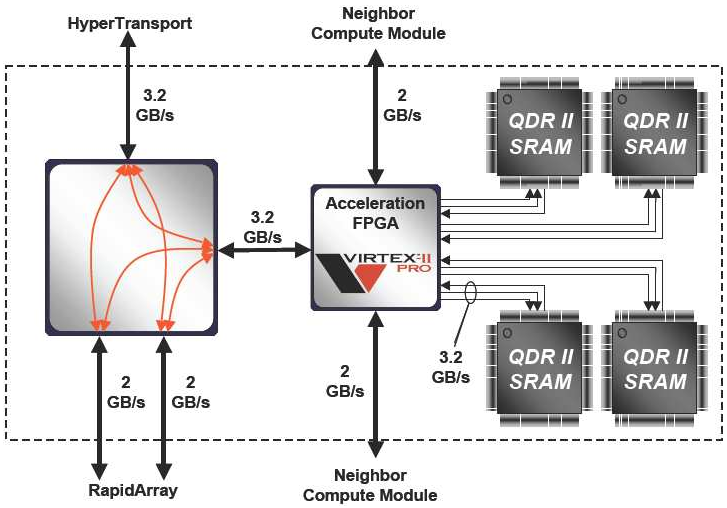


Figure 3·7: XD1 Connection [Inc]

connections with throughput of up to 4.8GBps.

SRC provides a set of MAP development tools from high level language compilers to FPGA debugging tools. The SRC MAP compiler translates C or FORTRAN codes to FPGA logic directly. The main program calls the FPGA logic as a function, which does the following tasks: download the user logic, transfer data via DMA, start and monitor data processing on FPGA, and finally, return control back to the calling program. The SRC MAP compiler generates FPGA configurations from high level languages by using methods such as parallelizing deterministic loops into pipelines and converting non-deterministic loops into state machines; the latter, lack of parallelism, however, sometimes results in little performance. Since the development starts from high level language, software-style debugging and hardware simulation are both supported. SRC defines two programming modes: a debug mode and a simulation mode. In debug mode, the high level language source codes can be quickly compiled and executed on the microprocessor. Also, developers can use the "printf" function to debug codes. In simulation mode, the source codes are compiled to Verilog and then to the Synopsis VCS simulator. Usually, system calls or other functions that require OS intervention are not allowed in MAP codes. For tasks such as on-board memory access, one can use a macro as a library function associated with a piece

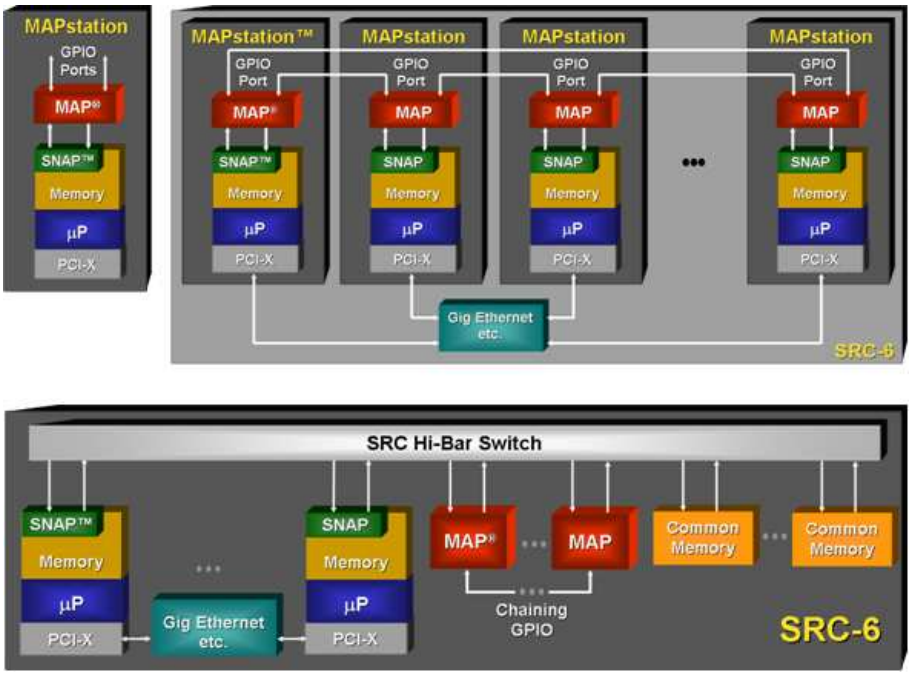


Figure 3·8: SRC MAP Configuration [HP05]

of predefined FPGA logic. Another use of macros is to insert user logic. In this case, one can develop and optimize FPGA logic with a different development flow and integrate it into MAP codes.

3.5.4 XtremeData XD1000

XtremeData released its HPRC solution, the XD1000 coprocessor module, in 2006 [Xtr07b]. It is neither a plug-in FPGA board in a workstation nor a plug-in chassis in a supercomputer. Rather, it is a coprocessor module that plugs directly into an Opteron 940 socket. Within this module, there is a Stratix-II FPGA from Altera, 4MB off-chip SRAM, 32MB FLASH memory, and a $JTAG/I^2C$ port. Through the Opteron 940 socket, it can communicate with the peer Opteron microprocessors via the HyperTransport bus, and can access the memory on the mother board. One system applying this coprocessor module is XtremeData's XD1000 development system. The coprocessor module sits aside an Opteron 2.2GHz microprocessor on a dual-CPU mother board. As shown in Figure 3·10,

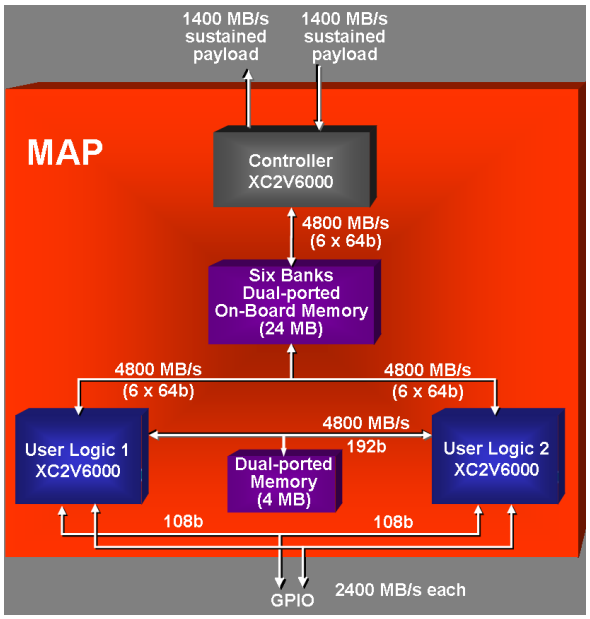


Figure 3·9: SRC MAP Box [HP05]

the XD1000 coprocessor module has its own 2GB DDR memory; the Opteron microprocessor handles peripherals with the south bridge; the bus between two sockets is up to 3.2GB/s. Other possible applications of the XD1000 coprocessor module include standard commercial HPC enterprise, rack, and blade servers. By replacing the Opteron microprocessors with the XD1000 modules, the system can be armed with FPGA computation modules without hardware change on current hardware, and these FPGA modules can take advantage of the high performance interface to microprocessors and memories.

XtremeData provides a platform support package (PSP) with Impulse-C, an extension of ANSI C developed by Impulse Accelerated Technologies. Developers can create accelerator including FPGA logic, memory interface, FPGA/CPU communication routines, and application software with only C-level development. With optimized libraries, XtremeData expects that the speedup from this process could be 50×.

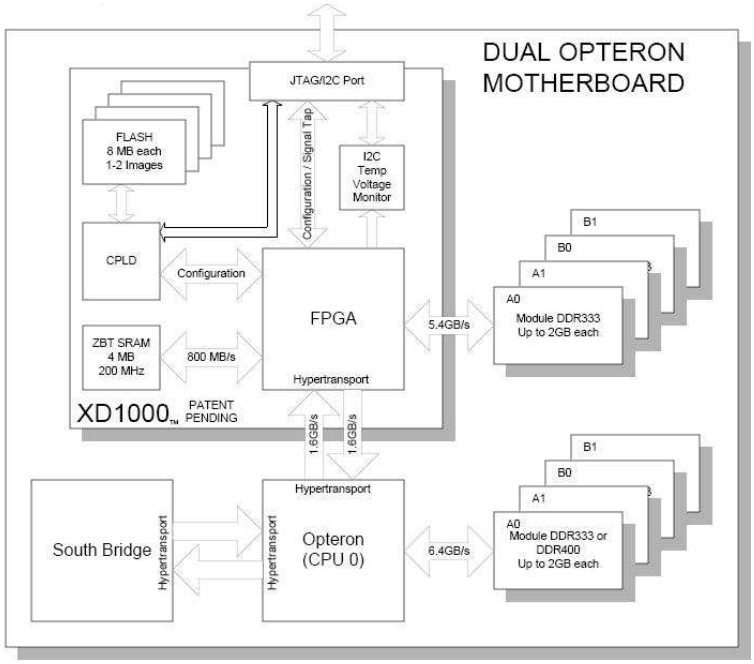


Figure 3·10: XD1000 BLOCK DIAGRAM [Xtr07a]

3.5.5 Annapolis Microsystems Plug-In Boards

Almost every FPGA chip has a plug-in development board that communicates via standard peripheral buses, such as PCI and VME. These boards are essential for creating FPGA applications. As FPGAs have become viable components for use in HPC, these plug-in boards have become suitable for building small-scale reconfigurable computer systems. This is because they contain necessary system-level components (analogous to PC motherboards). Thus (i) not only chips, but also boards are COTS parts; (ii) boards are independent of host, therefore able to work in different systems; (iii) existing PC or workstation based systems, e.g. cluster, can easily incorporate FPGA with these boards.

In these reconfigurable computer systems, the plug-in boards work as loosely coupled coprocessors, receiving data from host and sending results back over the I/O bus. All communication between coprocessor and host in this model is through the standard peripheral bus. This is slower than the tightly coupled solutions describe above; whether or not the communication is adequate depends of course on the applications.

We have found that plug-in boards are adequate for FPGA/MD systems to achieve acceleration of up to 10×. To achieve this for MD (or other non-trivial HPC application) one must explore the problems' data locality and reuse the data downloaded to coprocessors. Because the on-chip SRAMs are too small to hold the working set for most problems, plug-in boards contain substantial off-chip memory to extend the storage space.

Many vendors FPGA plug-in boards, including the Wildstar series from Annapolis Micro System, Inc, and the BenNUEY series from Nallatech, Inc. One example is shown in Figure 3·11, is the WILDSTAR®-II PRO/PCI Board from Annapolis Micro System, Inc., which we use in this work. This board consists of two Xilinx Virtex-II-Pro XC2VP70 -5 FPGAs; each FPGA is surrounded by six off-chip SRAM chips and a DRAM chip of up to 48MB and 128MB, respectively; FPGAs are interconnected through both differential pins and high speedup rocket I/O; both FPGAs communicate with host through the PCI Bus and have a socket to connect to other boards or external signal sources. Since this is a development board, creating FPGA applications is done with a standard design flow: the user develops logic with an HDL (perhaps through a higher level abstraction) and incorporates pre-defined modules from vendor; host software accesses boards with APIs containing the device driver. More detail about the particular version of this system that we used will be presented in Chapter 7.

3.5.6 Summary of Reconfigurable System Products

Current reconfigurable computer systems fall roughly into two categories: super-computer based systems (e.g. SGI RASC, Cray XD1, and SRC MAP Station) and extension board based systems (e.g. XtremeData XD1000 and Annapolis Wildstar plug-in boards). The former are extended from existing super-computer architectures. As a parts of HPC solutions, these systems also have tools to develop and integrated FPGAs into existing programming models. FPGAs can work as coprocessors sitting beside microprocessors in one node, or work as independent processors. In both models, FPGAs are able to communicate with other processors or access remote memory via high-performance buses. Since many HPC

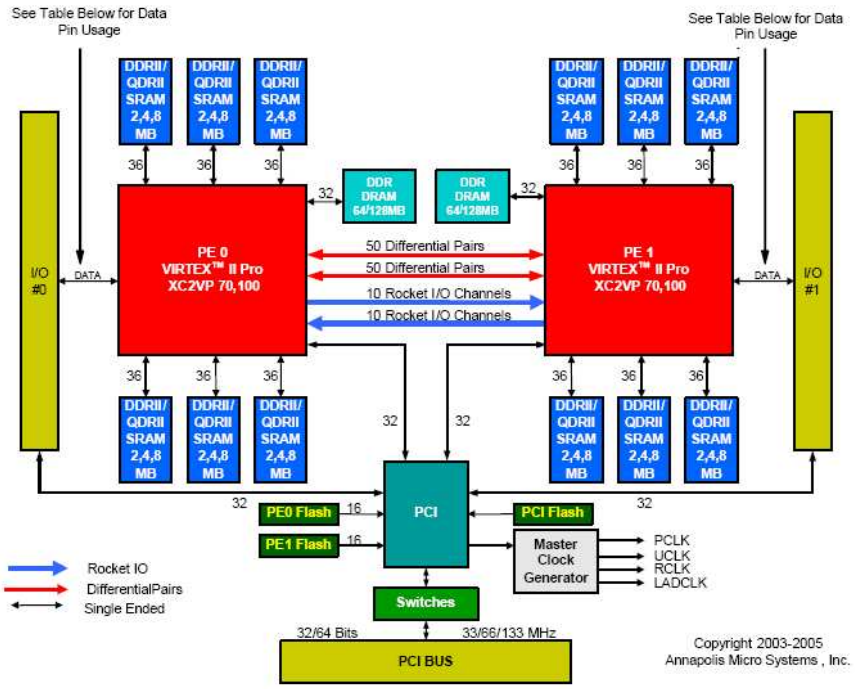


Figure 3·11: Diagram of WildstarII-Pro PCI board [Ann03]

applications have already been developed and tuned on the original super-computers, it is convenient to accelerate them with FPGAs on the extension systems. The disadvantage is that these systems are specific, so they are expensive and difficult to port FPGA designs across different systems. Developers also are required to have intensive knowledge of the systems to develop accelerated designs.

The systems of the second category are based on industry standard interfaces. Therefore, they have more flexibility and are more affordable. FPGA designs are also easier to port across different systems. One concern of this type of system is that the bandwidth of some industry standard interfaces becomes a bottleneck. Although careful FPGA algorithm design may relieve the pressure on communication for some applications, plugging FPGAs into the high-speed front bus (FSB) instead of the peripheral bus (e.g. the XtremeData XD1000 solution) seems promising. This solution can also be applied in super-computer based systems (e.g., the Cray XT4).

3.6 MD Related Work

Research on FPGA acceleration of MD is being carried out in recent years. Some designs and results have been published since 2003.

One of these studies is published by N. Azizi, et al. [AKE+04] in 2004. A preliminary MD system was implemented on a Transmogrifier 3 (TM3) system, which had 4 interconnected Virtex-E 2000E FPGAs and external SRAMs. This system evaluated the all-to-all Lennard-Jones force and the Verlet integration on FPGAs. The numerical computation was performed with fixed-point numbers of different scales. For Lennard-Jones, it applied a method, which is believed equivalent to the second order interpolation. The interpolation was performed on the acceleration curve directly. This system was capable of 8192 particles, but no capacity of supporting multiple particle types was reported. It was also hard to evaluate the simulation accuracy provided by this system with the information in this paper. The speedup of this acceleration is 0.29, slower than original software implementation on a PC of a 2.4GHz CPU. The authors cited the bottlenecks as memory bandwidth, clock speed, and lack of parallelism.

Another study was presented by R. Scrofano, et al. [SP04], in 2004. They applied simplified double precision FP arithmetic to compute the Lennard-Jones potential and force with original equations. Two 119 stage pipelines were placed and routed on a virtual Virtex-II Pro XC2VP125-7 chip, but no real performance was measured. This design was not integrated into any fully functional MD system, so there was no quality measurement either. A new version was developed on an SRC-6 MAP station [SGTP06, SP06]. Both the Lennard-Jones force and the real part of Smooth Particle Mesh Ewald Sum are calculated on FPGA coprocessors with floating-point arithmetic. In particular, $erfc(x)$ and e^{-x^2} in the real part of SPME are computed with lookup table and interpolation. The force computation including cell-list support is implemented by translating software into hardware. Because of the floating-point arithmetic complexity on FPGAs and the off-chip memory access latency, the force computation pipeline is split into two parts connected

with a FIFO. The FPGA coprocessor has to stall when it fetches force data from memory to accumulate and when the FIFO is full. The rest of the computation includes computing the bonded forces and the reciprocal part of the SPME, constructing cell-list, and integrating motions are computed on the general purpose microprocessors. Two simulations of 52K particles and 33K particles are reported. The speedup is about $2.7 - 2.9\times$. Single precision is used. No simulation quality is reported.

Kindratenko, et al., [KP06] reported their work of porting NAMD to the SRC-6 MAP station. Only the short range force computation was implemented on the FPGA chips. A stack of functions was rewritten to be packed in a single function and compiled with the MAP-C compiler [SRC05] to generate FPGA logic. The computation uses single precision floating-point arithmetic and four pipelines are fit in two Virtex-II Pro100 FPGA chips. The overall 3x speedup was achieved for a simulation of 92K particles.

Alam, et al., [AAS+07] also used the SRC-6 MAP Station. They ported Amber to the SRC-6 MAP station by mapping part of the real part of the Particle Mesh Ewald Sum (PME) to two Virtex-II Pro FPGA chips. The computation is carried out with single precision floating-point arithmetic. This system simulated two models of 23K particles and 61K particles respectively, and achieved about 4x speedup in both simulations.

As a part of the GRAPE project, PROGRAPE (PROgrammable GRAPE), using FPGA to replace GRAPE chips, has been developed for 4 versions since 1999 [HFKM00, HN05]. There are some attemps that maps MD to PROGRAPE [Sch07], but no much detail has been found.

Chapter 4

Algorithm Design Part 1: Short Range Forces

We now begin in earnest the discussion of our work in accelerating MD with FPGAs. As described in the introduction, we begin with aspects of the work that are more general and move to the FPGA-specific and finally to experiments with a particular system. In this chapter and the next (after this prologue), we describe overall design considerations related to first the short-range and then the long-range force computations. We begin with a brief overview of some of the issue related to mapping complex applications to FPGAs at the algorithmic level.

Mapping algorithms originally created for general purpose processors (GPPs) to FPGAs is in some ways more challenging than porting algorithms from single processor systems to parallel computers. In addition to general issues universal to parallel application design, including decomposition, communication, and synchronization, the design space for the FPGA coprocessor itself is also very large. The computational capabilities of GPPs and of FPGAs are quite different. GPPs have high operating frequency and hardware units for common data types, e.g. byte/word integer and standard floating-point; but the throughput is usually limited by the number of function units and the memory and I/O bandwidth. FPGAs, on the other side, have high parallelism/bandwidth; are able to efficiently operate with non-standard data types; and are inherent good at data processing. But because the FPGA control logic is usually implemented in state machines and are hardwired together with the data path, FPGAs are much less flexible than GPPs. Therefore, mapping algorithms from GPPs to FPGAs is not just a matter of swapping functions from software to hardware. Algorithms usually require redesign to match FPGAs' strengths, such as maximizing parallelism and throughput, while avoiding weaknesses such as high-precision

floating-point operations. In this research, we investigated and redesigned MD algorithms in several aspects, including the efficient numerical computation of complex expressions, the use of the multigrid method for the long range Coulomb force, and the use of alternative arithmetic mode: semi floating-point arithmetic mode.

The topics of the rest of this chapter concentrate on issues related to creating dedicated processors for computing complex expressions. Key is the fact that there is a large space of possible designs: these must be enumerated and evaluated. In the next section we describe issues related to fast interpolation. Issues are the format of the look-up, the interpolation mode, trade-offs between interpolation order and table size, arithmetic format, numerical precision, and evaluation of the alternatives.

4.1 Numerical Computation of Complex Expressions

4.1.1 General Considerations

Computing the short-range force involves computing complex expressions. There are two issues: the number of operations and their complexity. With respect to number of operations, most high-performance implementations of MD avoid direct computation of the r^{-x} terms, opting instead for polynomial approximation. This still leaves a large number of alternatives and, as we shall see, the choices depend very much on the hardware platform. The complexity of the operations is derived primarily from the assumption that double precision floating-point (DP) will be used (although some systems, GROMACS in particular, relax this assumption). If the precision or other aspects of the arithmetic complexity of DP can be reduced on the FPGA implementation, there will be a proportional increase in performance: chip area saved can be applied directly to increasing parallelism.

Double precision floating-point is considered canonical in applications like MD in traditional computer systems. However, we have applied fixed-point number and its variations in most computations conducted on FPGA. There are two main reasons: (i) floating-point arithmetic costs too much chip area, and (ii) it is more than necessary in many cases. In traditional computer systems, floating-point operations are performed by dedicated hardware

units designed for general applications. In this case, the cost of floating-point operations is acceptable, but it is also usually the only choice. In contrast, since our coprocessors are designed for special purposes, we are free to choose the arithmetic modes. Developing efficient numerical methods for MD-specific computation is both practical and essential to achieve speedup with FPGAs.

We must use proper arithmetic operations to match the FPGAs' capabilities. For instance, high-end FPGAs nowadays have embedded hardware multipliers and adders, such as the DSP modules in Xilinx Virtex IV and Virtex V families. These each contain a dedicated 18-bit $\times$ 18-bit 2's complement signed multiplier, adder logic, and a 48-bit accumulator. On these FPGAs, MAC (multiply-accumulate) is an efficient operation; division and square-root, however, must still be avoided even with fixed-point numbers.

Implementation of lookup tables is also efficient on FPGAs: the hundreds of integrated on-chip SRAMs can be configured to different widths and accessed directly accessed the configurable logic fabric. With these memories, the coprocessor can simultaneously work on a large number of lookup tables without a memory access bottleneck.

Because polynomial interpolation only involves addition, multiplication, and coefficient fetching, it is extremely suitable for FPGAs. We have applied high order polynomial interpolations with a novel variation of fixed-point system-semi floating-point (Semi-FP) for the Lennard-Jones force and the short range part of the Coulomb force computation in our coprocessors. In the following subsections, we describe first our study of the interpolation methods for FPGAs, followed by an introduction to Semi-FP arithmetic.

4.1.2 Interpolation of r^{-x}

Recall that the Lennard-Jones force for particle i can be expressed as:

$$\mathbf{F}_i^{LJ} = \sum_{j \neq i} \frac{\epsilon_{ab}}{\sigma_{ab}^2} \left\{ 12 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{14} - 6 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{8} \right\} \mathbf{r}_{ji} \tag{4.1}$$

where the ϵ_{ab} and σ_{ab} are parameters related to the types of particles, i.e. particle i is type a and particle j is type b. Also that the Coulombic force can be expressed as:

$$\mathbf{F}_i^C = q_i \sum_{j \neq i} \left(\frac{q_j}{|\mathbf{r}_{ji}|^3} \right) \mathbf{r}_{ji} \tag{4.2}$$

We assume that the Coulomb force is split into two components: the slowly converging part that is computed using one of the long-range methods, and the rapidly converging part that together with the Lennard-Jones force comprises the short-range force to be computed.

Most of the complexity in MD is in the short-range, non-bonded, force computation. As describe previously, this has two components, the Lennard-Jones force and the rapidly converging component of the Coulomb force. The Lennard-Jones force is often computed with the so-called 6-12 approximation given in Equation4.1 and shown in Figure 2·2. Since this calculation is the inner loop, considerable care is taken in its implementation: even in serial implementations, the equation is not evaluated directly, but rather through a table look-up plus interpolation.

Previous implementations of FPGA/MD have used table look-up for the entire Lennard-Jones force as a function of particle separation [AKE+04, GVH06a]. The index used is $|r_{ji}|^2$ rather than $|r_{ji}|$ so as to avoid the costly square-root operation.

$$\frac{\vec{F}_{ji}^{LJ}(|r_{ji}|^2, (a,b))}{\vec{r}_{ji}} = \frac{\epsilon_{ab}}{\sigma_{ab}^2}\{12(\frac{\sigma_{ab}}{|r_{ji}|})^{14} - 6(\frac{\sigma_{ab}}{|r_{ji}|})^8\} \tag{4.3}$$

where $\vec{r}_{ji}$ is the displacement between atom i and atom j; a and b are the types of these two atoms.

Because it is possible in principle for any two atoms to interact this way, computing this force has been problematic. Here we describe a new method of short-range force computation that changes what gets looked up, the shape of the table; in later sections, we will discuss the interpolation mechanism, and the method of computing interpolation coefficients.

Previous implementations of FPGA/MD have used table lookup for the entire force as a function of particle separation [AKE+04, GVH06a]. This method is efficient for uni-

form gases where only a single table is required [AKE+04]: because it is not necessary to distinguish atom types, the lookup table is a function only of displacement. As the Lennard-Jones force, however, depend in general atom types, simulations of T different atom types require $T^2/2$ tables. Since table lookup is in the critical path, the tables must be in on-chip SRAMs for FPGA/MD to be viable. In previous work we described a latency hiding technique whereby tables are swapped as needed [GVH06a].

Here we propose a different method, used also in some GPP MD codes. Instead of implementing the force pipeline with a single table lookup for the entire force, we use two tables for the Lennard-Jones force, one each for r^{-14} and r^{-8}. Equation 4.3 now becomes Equation 4.4. Because $|r_{ji}|^2$ is much easier to compute than $|r_{ji}|$, the lookup table is indexed with $|r_{ji}|^2$.

$$\frac{\vec{F}_{ji}^{LJ}(|r_{ji}|^2, (a,b))}{\vec{r}_{ji}} = A_{ab} \times |r_{ji}|^{-14} + B_{ab} \times |r_{ji}|^{-8} = A_{ab} \times R_{14}(|r_{ji}|^2) + B_{ab} \times R_8(|r_{ji}|^2) \quad (4.4)$$

where

$$A_{ab} = 12\epsilon_{ab}\sigma_{ab}^{12} \quad (4.5)$$

and

$$B_{ab} = -6\epsilon_{ab}\sigma_{ab}^{6} \quad (4.6)$$

and where $R_{14}(x) = x^{-7}$ and $R_8(x) = x^{-4}$ are two lookup tables both indexed with $|r_{ji}|^2$.

Thus, the force is computed with two sets of lookup tables. A_{ab} and B_{ab} are coefficient lookup tables indexed with atom types, but they independent of distance; R_{14} and R_8 are curve lookup tables indexed with distance, but independent of atom types. The two-dimensional lookup table Equation 4.3 becomes a joint of two sets of one-dimensional lookup tables. Assuming we interpolate the Lennard-Jones force in cutoff with K intervals, the direct interpolation method requires $K \cdot T^2/2$ coefficients, while $2K + T^2$ by our method.

The advantage is that we can easily support up to 32 atom types without swapping tables; the disadvantage is that a few more operations must be performed in the interpolation pipeline.

Returning now to the Coulomb force computation: because applying a cut-off to

$$\frac{\mathbf{F}_{ji}^{CL}(|r_{ji}|^2(a,b))}{\mathbf{r_{ji}}} = Q_a Q_b |r_{ji}|^{-3} = QQ_{ab}R_3(|r_{ji}|^{-3}) \tag{4.7}$$

often causes unacceptable error, and also because the all-to-all direct computation is too expensive for large simulations, various numerical methods are applied to solve the Poisson equation that translates charge distribution to potential distribution. To improve approximation quality and efficiency, these methods split the original Coulomb force curve in two parts with a smoothing function $g_a(r)$: a fast declining short range part and a flat long range part. For example:

$$\frac{1}{r} = (\frac{1}{r} - g_a(r)) + g_a(r) \tag{4.8}$$

where the two components are shown in Figure 5·4. The short range component can be computed together with Lennard-Jones force using a third look-up table. The entire short range force becomes:

$$\frac{\mathbf{F}_{ji}^{short}}{\mathbf{r_{ji}}} = A_{ab}r_{ji}^{-14} + B_{ab}r_{ji}^{-8} + QQ(r_{ji}^{-3} + \frac{g_a'(r)}{r}) \tag{4.9}$$

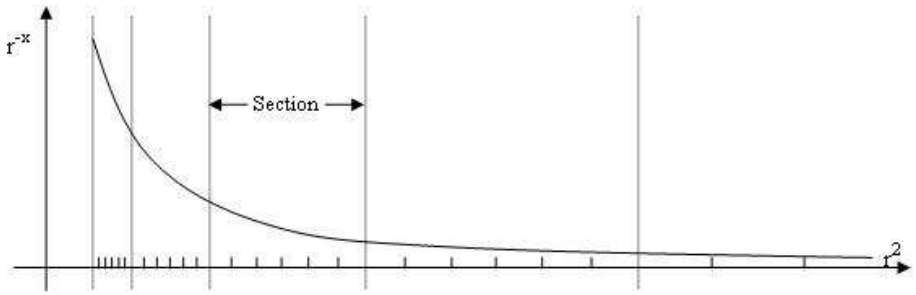


Figure 4·1: Logarithmic Intervals for r^{-x} Interpolation

We next describe an optimization to the lookup tables themselves. The r^{-x} expressions display extreme changes in behavior over the range of possible interaction radii, as shown

in Figure 4·1. It would be an exorbitant and needless cost in table size to use the same number of coefficients and the same step sizes in the well-behaved regions of the curve. Rather, as in [AKE+04], each curve is divided into several sections along the X-axis. Here, the length of each section is twice that of the previous; however, each section is cut into the same number interpolation intervals N.

To improve the accuracy of the force computation, we interpolate using higher order terms. Here we assume a Taylor expansion; below we describe a more accurate alternative. The C_i are the coefficients, x is the offset within the interval, and a is the left end of the interval.

$$f(x) \;=\; C_0 + C_1(x-a) + C_2(x-a)^2 + C_3(x-a)^3 + \;\ldots\; + C_n(x-a)^n + o(x^n) \quad (4.10)$$

When the interpolation is of order M, each interval needs $M+1$ coefficients, and each section needs $N \times (M+1)$ coefficients. Since the section length increases exponentially, extending the curve (in r) only increases the size of coefficient memory very slowly.

Table 4.1: Trade-off between Interval Size N (N is the Number of Intervals per Section) and Interpolation Order M for r^{-7}.

N	M	Average Relative Error	Maximum Relative Error
32	4	2.55e-007	3.67e-006
64	4	7.35e-009	1.08e-007
64	3	3.74e-007	4.19e-006
128	3	2.26e-008	2.55e-007
128	2	2.17e-006	1.73e-005
512	2	3.32e-008	2.66e-007
512	1	1.17e-005	6.04e-005
2048	1	7.31e-007	3.76e-006

M and N are two major design coefficients for the short range force coprocessor. Increasing M or N each improves simulation accuracy, but reduces the chance to have more force pipelines. Interestingly, on the FPGA these two numbers have a resource cost in

different hardwired components: the main cost for finer intervals is in the on-chip SRAMs, while the main cost for higher order interpolation is in hardware multipliers and registers. 4.1 gives a sample of the tradeoff effects on $R_{14} = x^{-7}$. In our first version, where the all particles were stored on chip, $N = 128$ and $M = 3$ appears to be optimal. In the second version, because particles are stored in the off-chip memories and every time only a small portion is swapped in FPGA, less block RAMs are needed for particle data, and the optimal configuration became $N = 512$ and $M = 2$. A consequence of the change in parameters is that, because of the reduction in compute resources required for $M = 2$ versus $M = 3$, the number of pipelines can often be doubled. This doubles performance without affecting simulation quality.

4.1.3 Computing the Coefficients

We now show how to develop polynomial approximations to arbitrary functions so that the approximation is relatively easy to compute with economical use of FPGA resources. The FPGA will implement the approximation F of function f as a sum of terms:

$$F(x) \;=\; C_0 + C_1 x + C_2 x^2 + C_3 x^3 \tag{4.11}$$

where the coefficients C_i are stored separately for a set of intervals that partition the x range of interest. This form is desirable for FPGA implementation because the low powers of x require relatively few hardware multipliers. In order to maintain accuracy in F, it is defined as a piecewise function on a set of intervals that partition the range of interest, with coefficients C_i for any interval chosen to provide the best approximation of f on that interval. The piecewise nature of F is taken for granted in the remainder of this discussion. One obvious way to create such an approximation on an interval near a point is by truncating a Taylor series expansion. When carried to an infinite number of terms, this represents F exactly. Truncating the series causes problems in accurate approximation, however. In order to see how accuracy problems arise, consider the curves in Figure 4·2 illustrating the first few monomials x_i.

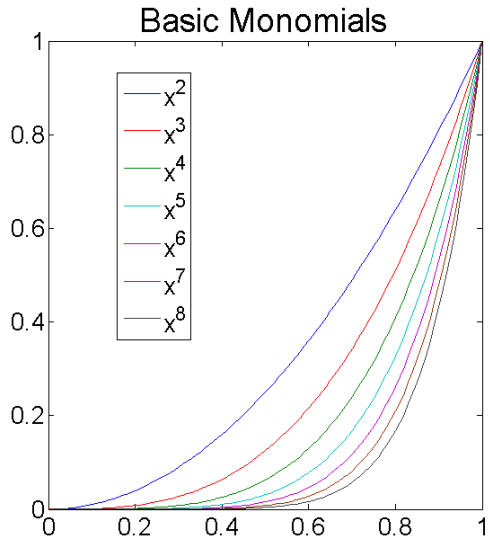


Figure 4·2: Basic Monomials

All of the curves have the same general half-U shape, and become increasingly similar as the degree increases. In other words, the x^2 or x^3 term of a low-order approximation is important in approximations of all higher-order Taylor terms. Said differently, large coefficients of high-order Taylor terms contribute heavily to the coefficients in the low-order approximation. The $i!$ denominator in the Taylor series dominates for asymptotically large i. Still, for the functions and ranges of current interest (r^{-4} and r^{-7}, $0.1 < r < 100$), many Taylor series terms $i > 3$ have appreciably large numerators due to large binomial coefficients in $(x-p)^i$ and large derivative values $d^i f/dp^i$. As a result, the truncated Taylor series omits many large high-order terms that are important to approximation using low-order polynomials. This is especially problematic for the larger intervals where behavior may worse even than that of linear interpolation.

The orthogonal polynomial interpolation expression is also a polynomial of order p and is arranged as a function of $(x - a)$:

$$f(x) \;=\; C_0 + C_1(x-a) + C_2(x-a)^2 + C_3(x-a)^3 + \;\ldots\; + C_p(x-a)^p + o(x^p) \qquad (4.12)$$

where a is the starting point of an interval. The algorithm to compute the coefficients C_i will be addressed later. With the orthogonal polynomial basis $\{Q_{iab}|i = 0, 1, 2, , p-1\}$ on range $[a, b)$ derived from the Legendre Polynomials shown in Figure 4·3, we can construct any polynomial of pth order on $[a, b)$, including the Taylor expansion. An approximation of $F(x)$ in terms of polynomial space is a projection of $F(x)$ from some high-order space down into the low-order space spanned by a set of basis functions. Monomials, $\{1; x; x^2; x^3, \dots\}$, are not the most convenient set of basis functions to construct approximation polynomials. When we expand the approximation polynomial by involving a high order monomial, the weights of the low order terms are to be adjusted, because monomials are not orthogonal to each other. This is another explanation why truncating Taylor expansion doesn't generate the best approximation - the low order terms are not adjusted after high order term involves. However, if the $Q_{iab}(x)$ are orthogonal, we can project $F(x)$ to the axis of $Q_{iab}(x)$ and avoid affecting projections to other axis in the space. Furthermore, if we choose a proper orthogonal basis, such as the Legendre Polynomials, in which the order of basis functions is continuous by increments of '1', we can span the polynomial space by increasing the order by '1' as well. Our FPGA implementation uses polynomials of the form in Equation 4.11. We can gradually improve approximation quality by constructing polynomials with more basis functions (i.e. increasing polynomial order) based on the computation budget. At the same time, the set of basis functions guarantees two nice properties:

- Least squared error, $\|F(x) - P_{ab}(x)\|_2$, on $[a, b)$.
- Approximation bias is zero, i.e. $\int_a^b (F(x) - P_{ab}(x))dx = 0$

The first property guarantees $P_{ab}(x)$ is the best local approximation in the polynomial space of a given order; the second implies that if the data distribution is uniform, the overall approximation bias will cancel out.

Although the orthogonal polynomial interpolation is the best local approximation in each interval, the interpolation polynomials are continuous only within interval. When transformed into Fourier space, as is done in some long range force calculations, spurious

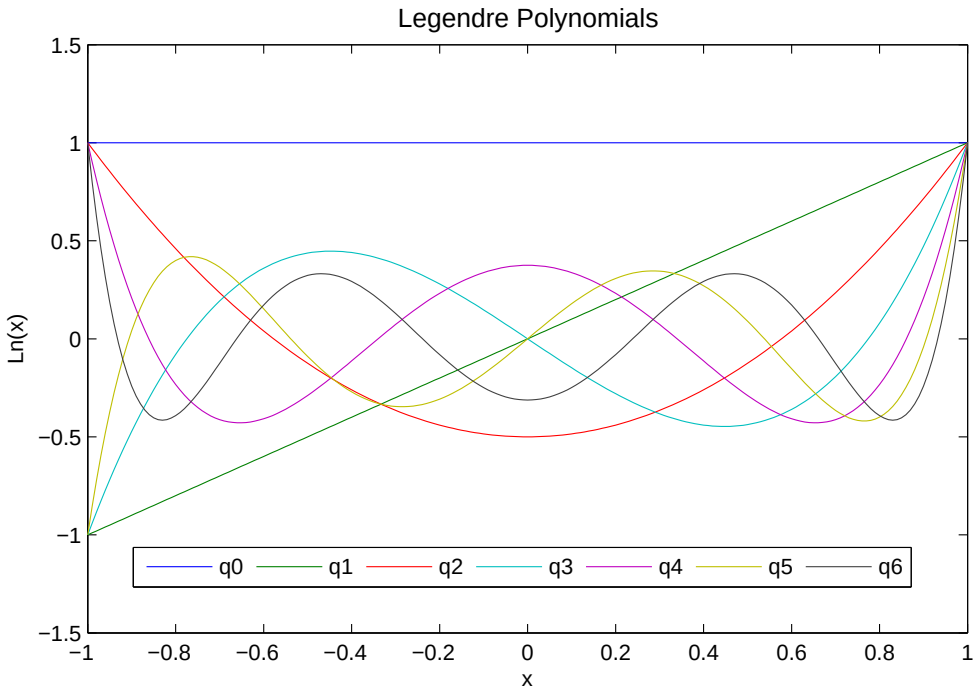


Figure 4·3: Legendre Polynomials

terms may result.

Piecewise Hermite interpolation is an improved version of the original piecewise linear method. Given two points on the target curve and their derivatives $\{(x_0; f(x_0); f'(x_0)); (x_1; f(x_1); f'(x_1))\}$; a polynomial is required to go across these two points with the same derivatives. Since there are four constraints, this polynomial is third order. The four coefficients can be computed by solving the following equations:

$$\begin{aligned} f(x_0) &= dx_0^3 + cx_0^2 + b_0 x + a \\ f'(x_0) &= dx_0^2 + cx_0 + b \\ f(x_1) &= dx_1^3 + cx_1^2 + b_1 x + a \\ f'(x_1) &= dx_1^2 + cx_1 + b \end{aligned} \tag{4.13}$$

4.1.4 Comparing the Interpolation Methods

We formalized the problem here: given computation budget, i.e. interpolation order and the number of pieces (bins or intervals), find optimal interpolation polynomials to minimize the simulation error. This problem then is transformed to find optimal polynomials to minimize interpolation error for target functions.

We compare the three higher order interpolation methods - Taylor, Orthogonal, and Hermite - by plotting their relative RMS error. In the left graph of Figure 4·4 and Figure 4·5, the number of intervals per section varied; in the right graph, the order is varied. We observe that the method of orthogonal polynomials is superior to the others. This is not surprising because all the other interpolation polynomials are subsets in the orthogonal polynomial spanning space. Problems with the Taylor expansion have already been described; for higher order Hermite interpolation, the RMS error is larger than Taylor and Orthogonal, which can be regarded as the cost to maintain the continuities among intervals. In general, increasing interpolation order by 1 achieves a similar effect as multiplying the number of intervals number by 4 or 8.

One concern about piece-wise interpolation is continuity. Hermite guarantees first and second order continuity with a third order polynomial, but the average RMS error is worse than Orthogonal by three digits or more. Moreover, although no effort is made to "line-up" the end-points of each interval with Orthogonal, they do so anyway to the resolution of the arithmetic.

Minimax is another criteria people always apply to interpolation quality. Although Orthogonal by default does not yield minimax error, it guarantees that the average interpolation error is minimized and the approximation bias is zero, if the input is of uniform distribution. Since in MD the short range force curve is approximated in small pieces, it is acceptable to assume that the input is of uniform distribution within each interval. Because of the average error is optimal, the maximal error in a small interval is close to the optimal one.

We return to the trade-off between number of intervals per section and order of the

polynomial used for interpolation, with respect to the method of orthogonal polynomials (see Figigure 4·6). We observe that several combinations of order M versus intervals N have similar error, allowing for hardware-based design choice describe early.

Table 4.2: Relative Root Mean Square Error of r^{-7} with Orthogonal Polynomial Interpolation.

	N=32	N=64	N=128	N=256	N=512
M=1	1.37E-02	4.82E-03	1.70E-03	6.02E-04	2.13E-04
M=2	2.43E-04	4.29E-05	7.59E-06	1.34E-06	2.37E-07
M=3	3.73E-06	3.29E-07	2.90E-08	2.57E-09	2.27E-10
M=4	5.17E-08	2.28E-09	1.01E-10	4.45E-12	1.13E-13

Table 4.3: Relative Root Mean Square Error of r^{-7} with Taylor Polynomial Interpolation.

	N=32	N=64	N=128	N=256	N=512
M=1	9.05E-02	3.04E-02	1.05E-02	3.66E-03	1.28E-03
M=2	5.54E-03	9.17E-04	1.57E-04	2.73E-05	4.78E-06
M=3	3.05E-04	2.49E-05	2.11E-06	1.83E-07	1.60E-08
M=4	1.56E-05	6.30E-07	2.66E-08	1.15E-09	5.02E-11

Table 4.4: Relative Root Mean Square Error of r^{-7} with Hermite Polynomial Interpolation.

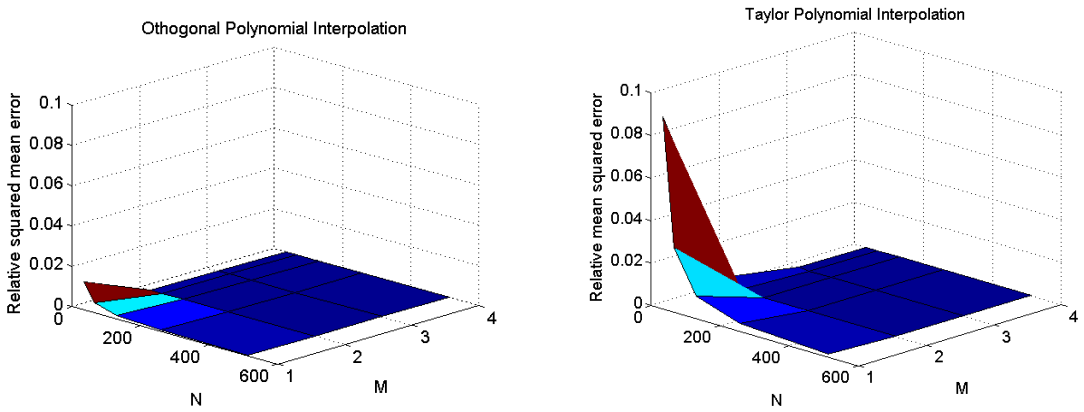
	N=32	N=64	N=128	N=256	N=512
M=3	6.10E-03	1.05E-03	1.82E-04	3.20E-05	5.63E-06

4.1.5 Algorithm to Compute Interpolation Coefficients with Orthogonal Polynomials

We now address the algorithm to compute the coefficients Ci in Equation 4.12. This problem can be defined as following: given any continuous function $F(x)$ and an interval

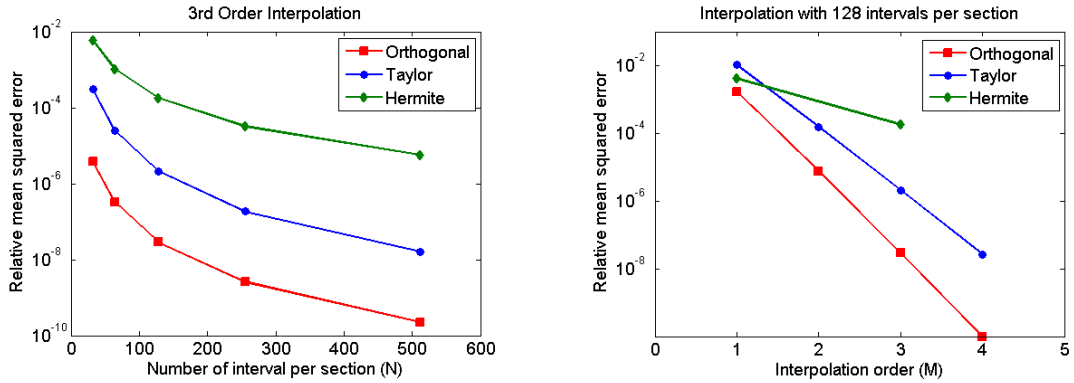
Table 4.5: Relative Root Mean Square Error of r^{-7} with Linear Polynomial Interpolation.

	N=32	N=64	N=128	N=256	N=512
M=1	3.34E-02	1.18E-02	4.17E-03	1.47E-03	5.21E-04



(a) (b)

Figure 4·4: Relative Root Mean Square Error of r^{-7} with Orthogonal and Taylor Polynomial Interpolation. The input range is from 2^{-4} to 2^{7} of logarithmic intervals.



(a) (b)

Figure 4·5: Projections of Data in the Previous Figure to 3rd Order Interpolation (a) and 128 Intervals per Section (b)

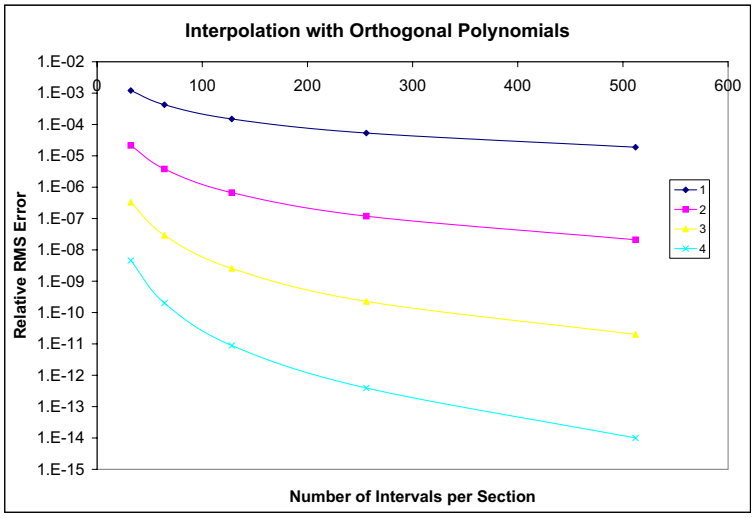


Figure 4·6: Orthogonal Comparisons

set, I, on range $[A, B)$, find a set of best approximation polynomials on every interval, such as $P_{ab}(x)$ on interval $[a, b)$, where $A \leq a < b \leq B$.

Algorithm:

1. Compute orthogonal polynomial basis on range $[a, b)$.

   Given orthogonal polynomial basis on range $[-1, +1]$, Legendre polynomials, which are computed as following:

   $q_0(z) = 1$

   $q_1(z) = z$

   $(n+1)q_{n+1}(z) = (2n+1) \cdot z \cdot q_n(z) - nq_{n-1}(z)$

   The orthogonal polynomial basis on range $[a, b) : Q_{0ab}\ Q_{1ab} \ldots$, are acquired by substitution $z = \dfrac{x - (a+b)}{b-a}$.

2. Compute the coefficients k_i, so that $F(x) \approx \sum_{i \geq 0} k_i Q_{iab}(x) = P_{ab}(x)$ on interval $[a, b)$.

$$k_i = \frac{\int_a^b F(x) Q_{iab}(x)}{\int_a^b Q_{iab}(x) Q_{iab}(x)} \tag{4.14}$$

3. Compute the coefficients C_i, so that $F(x) \approx \sum_{i\geq 0} C_i(x-a)^i = P_{ab}(x)$.
   Substituting x with $x+a$ in $P_{ab}(x) = \sum_{i\geq 0} k_i Q_{iab}(x)$ and expanding it yield C_i.
4. Repeat step 1 to 3, for every interval in I.

Now, we show some properties of these polynomials.

Lemma 1: $\{Q_{iab}|i = 0, 1, 2, \dots\}$ are orthogonal.

Proof:

If $i \neq j$,

$$\langle Q_{iab}(x)Q_{jab}(x)\rangle \;=\; \int_a^b Q_{iab}(x)Q_{jab}(x)dx \;=\; \int_{-1}^{+1} q_i(z)q_j(z)dz \;=\; \langle q_i(z)q_j(z)\rangle \;=\; 0$$

Property 1: Least squared error, $\|F(x) - P_{ab}(x)\|_2$, on $[a, b)$.

Proof:

If $i \neq j$, F(x) can be represented precisely by a polynomial of infinite order, $F(x) = \sum_{i\geq 0} F_i x^i$. With the recursive construction equations, x^j can be derived with linear combination of $\{Q_{iab}|i = 0, 1, 2, \dots j\}$. In another word, the orthogonal basis can be constructed from monomials through the Gram-Schmidt process too. Therefore, the polynomial space of order p is equivalent to the space constructed with the first p basis of $\{Q_{iab}|i = 0, 1, 2, \dots\}$.

The 2-norm between $F(x) = \sum_{i\geq 0} F_i x^i$ and an arbitrary pth order polynomial $L(x) = \sum_{0\leq i<p} L_i x^i$ is:

$$\|\sum_{i\geq 0} F_i x^i - \sum_{0\leq i<p} L_i x^i\|_2$$

They both can be constructed with $\{Q_{iab}|i = 0, 1, 2, \dots\}$ with coefficients $\{k_i\}$ and $\{l_i\}$ computed with Equation 4.11. In particular,

$$\sum_{0\le i<p} k_i Q_{iab}(x) = P_{ab}(x)$$

Because the $\{Q_{iab}|i = 0, 1, 2, \dots\}$ are orthogonal,

$$\begin{aligned} \|\sum_{i\ge 0} F_i x^i - \sum_{0\le i<p} L_i x^i\|_2 &= \|\sum_{i\ge 0} k_i Q_{iab}(x) - \sum_{0\le i<p} l_i Q_{iab}(x)\| \\ &= \|\sum_{0\le i<p} (k_i - l_i) Q_{iab}(x) + \sum_{i\ge p} k_i Q_{iab}(x)\|_2 \\ &\ge \|\sum_{i\ge p} k_i Q_{iab}\|_2 \end{aligned}$$

The least squared is acquired only when $\{l_i\}$ equals the corresponding $\{k_i\}$, i.e., the polynomial coefficients generated by our algorithm.

Lemma 2: $q_n(1) = 1, n \in N$

Lemma 3: $q_n(-z) = (-1)^n q_n(z), n \in N$

Lemma 4: $(2n+1)q_n(z) = \dfrac{d}{dz}(q_{n+1}(z) - q_{n-1}(z)), n \in N$

Lemma 5: $\int_a^b Q_{iab}(x)dx = 0, i \in Z$

Proof:

For $n \ge 1$, because of **Lemma 2** and **Lemma 3**, when n is even,

$$q_{n+1}(1) = q_{n-1}(1) = 1 \text{ and } q_{n+1}(-1) = q_{n-1}(-1) = -1$$

when n is odd,

$$q_{n+1}(1) = q_{n-1}(1) = q_{n+1}(-1) = q_{n-1}(-1) = 1$$

Because of **Lemma 4**,

$$\begin{aligned}
\int_{-1}^{+1} q_n(z)dz &= \frac{1}{(2n+1)}\int_{-1}^{+1}(\frac{d}{dz}(q_{n+1}(z)-q_{n-1}(z)))dz \\
&= \frac{1}{(2n+1)}(q_{n+1}(z)-q_{n-1}(z))|_{-1}^{+1} \\
&= \frac{1}{(2n+1)}(q_{n+1}(1)-q_{n+1}(-1)-q_{n-1}(1)+q_{n-1}(-1)) \\
&= 0
\end{aligned}$$

With substitution $z = \dfrac{x-(a+b)}{(b-a)}$,

$$\int_a^b Q_{iab}(x)dx = \int_{-1}^{+1} q_n(z)dz = 0$$

Property2: Approximation bias is zero, i.e. $\int_a^b (F(x)-P_{ab}(x))dx = 0$.

Proof:

For any $p \geq 0$

$$\begin{aligned}
\int_a^b (F(x)-P_{ab}(x))\,dx &= \int_a^b \left(F(x)-\sum_{0\leq i<p} k_i Q_{iab}(x)\right)dx \\
&= \int_a^b \sum_{i\geq p}(k_i Q_{iab}(x))\,dx \\
&= \sum_{i\geq p}\left(k_i \int_a^b Q_{iab}(x)dx\right) \\
&= 0
\end{aligned}$$

4.2 Semi Floating Point Numbering

The analysis of previous section is based on general mathematic methods. This section shows how we map polynomial interpolation to FPGAs. Floating-point is applied

in most MD packages for two reasons: (i) MD requires accurate numercial computation; (ii) floating-point function units are standard in most microprocessors. However, we do not have standard floating-point function units on current FPGAs; and more importantly, we have alternative choices to do accurate computation. In the section, we first investigate the floating-point number system and logarithmic number system. Next, we introduce our semi floating-point (Sem-FP) number systems and compare it with others.

Floating Point Numbering System

Floating-point number systems, such as the single and the double precision numbers specified in IEEE standard 754, basically split the bit string in three fixed length pieces: sign bit, exponent, and mantissa. With the symbols shown in Figure 4·7, the normalized value can be computed as $F = (-1)^S \cdot M \cdot 2^E$.

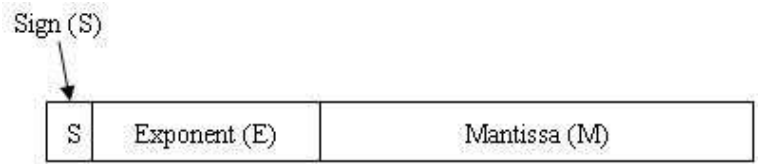


Figure 4·7: Floating Point Number

Floating-point numbering systems can vary in the sizes of E, M, and in support of denormal values. The IEEE Standard 854 Radix-Independent Floating-Point Arithmetic, allows floating-point formats with base 2 or 10, and also provides guidance for other bases and word lengths. We will only use this general model, however, for later analysis.

The dynamic range of a floating-point number system is determined by the length of the exponent part and the finest resolution; the precision is maintained with those bits of the mantissa. For example, the IEEE Standard 754 single precision number has 8 bits for exponent and 23 bits for mantissa; just considering the normalized value, the dynamic range is about $\pm 10^{38}$, while the finest resolution is about 1.18×10^{-38}.

Floating-point addition is more complex than multiplication, because operands must be aligned before adding mantissas, which is not necessary for multiplication. Aligning

mantissas, even with barrel shifters, is still very expensive in terms of logic and latency, because mantissas can be shifted by any distance within its length and in either direction. That is why floating-point adders are usually larger than multipliers, although they both also require that the result be (potentially) normalized by shifting.

Logarithmic Numbering System

Logarithmic number systems (LNS) [SA75, HaAWH+05] can be regarded as a special case of floating-point systems, which only has the sign bit and exponent, but the mantissa is omitted, because it is always equal to '1'. The normalized value can be computed with $F = (-1)^S \cdot 2^E$. Clearly, '0' can not be represented with this formula. In practice, extra tag bits are used to indicate '0' and other non-standard values. Figure 4·8 shows an LNS format.

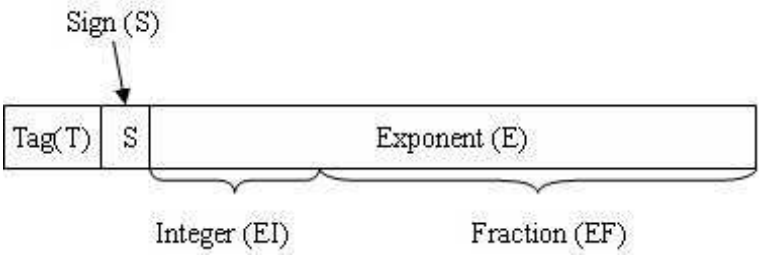


Figure 4·8: Logarithmic Number

The exponent is a fixed-point number and can be partitioned into two parts: Integer (EI) and Fraction (EF). When the length of EI and EF are equivalent to the length of the exponent and mantissa of floating-point, the LNS can cover almost the same dynamic range as the floating-point number system.

As to operators, LNS simplifies multiplication, division, powers, and roots by converting them to addition, subtraction, multiplication, and division, respectively. This is the most significant advantage of using LNS. Meanwhile, the cost is that the addition/subtraction operator is more complicated. For example, to do LNS addition $C = A + B$, where A and B have the same sign bit, the exponent of C must be computed with Equation 4.15.

$$E_c = log_2|A \pm B| = log_2\left|A(1 \pm \frac{B}{A})\right| = log_2|A| + log_2\left|1 \pm \frac{B}{A}\right| = E_A + f(E_B - E_A) \quad (4.15)$$

where

$$f(x) = f(E_B - E_A) = log_2\left|1 \pm \frac{B}{A}\right| = log_2\left|1 \pm 2^{(E_B - E_A)}\right| \quad (4.16)$$

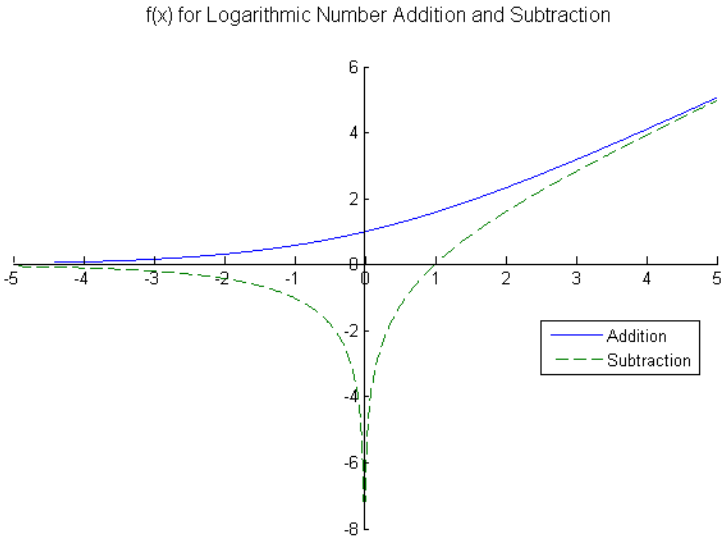


Figure 4·9: f(x) for Logarithmic Addition and Subtraction

$f(x)$ is plotted in Figure 4·9. Describing this curve introduces the complexity of the addition and subtraction operators. Many piece-wise polynomial interpolation schemes have been studied for f(x) [Lew94]. A hybrid approach does multiplication, division, powers, and roots with LNS, but leaves addition and subtraction with floating-point arithmetic. To transform numbers between two formats is evidently necessary; [FSL91] presented an interpolator for transferring numbers between these systems in linear time.

[HaAWH+05] compared the resource usage of different operators of floating-point and LNS on FPGAs. Table 4.6 lists the usage of slices on Xilinx Virtex II Family FPGAs. LNS addition/subtraction also requires extra on-chip SRAMs and hardware multipliers for

interpolation. Neither of these two systems has obvious absolute advantages over the other one. In the rest of this section, we will mostly use the traditional floating-point numbering system as a baseline to compare the fixed-point number systems and our Semi-FP number system.

Table 4.6: Resource Usage of Floating Point and LNS

	Single Precision FP	Single Precision LNS	Double Precision FP	Double Precision LNS
Addition	424	750	836	1944
Subtraction	424	838	836	2172
Multiplication	297	20	820	36
Division	910	20	3376	36

Semi Floating Point Numbering System

Both floating-point and LNS are expensive for FPGAs and become especially so when the computation is large-scale parallel high-order polynomial interpolation. The obvious alternative, fixed-point numbering, has the obvious problem that the dynamic range of coefficients is too big. For instance, we perform 3rd order polynomial interpolation to approximate $f(x) = x^{-7}$ for the Lennard-Jones force, in which x ranges from 2^{-4} to 2^7 for molecular systems. It is sufficient to use Taylor expansion to simplify our analysis, because the coefficients computed with orthogonal polynomials are almost of the same magnitudes. Considering the third order term,

$$C_3 = \frac{f^{(3)}(x)}{3!} = \frac{84}{x^{10}} \tag{4.17}$$

it varies from -9.2×10^{13} to -7.1×10^{-20} according to the curve's high and low ends respectively. For fixed-point, the distance between the most significant bits of these two numbers is more than 100 bits. Putting this another way, we need more than 100 bits to retain the magnitude of -9.2×10^{13} and to still be able to distinguish -7.1×10^{-20} from 0. Arithmetic operations on 100-bit fixed-point numbers require as many resources

and latency as floating-point operations. On the other hand, the *useful* precision of the coefficients is much less than 100 bits: most bits are zero (or sign bit extension), which do not improve accuracy.

The essential differences between fixed-point and floating-point are the capability for big dynamic range and the complexity of the hardware implementation. Fixed-point is preferred in cases where dynamic range is limited. Our major concern is interpolation, or more precisely, piecewise polynomial interpolation. The polynomial interpolation coefficients for r^{-x} do not appear to fall into the category of limited dynamic range. After inspecting the coefficient expressions, however, we observed that their dynamic ranges are still bounded in some sense.

The dynamic range of the coefficients, i.e., in Equation 4.17, is huge because the input range varies to a great extent. As the logarithmic interval scheme shows in Figure 4·1, however, the MSB of the interpolation input varies no more than one bit in each section. Consequently, the coefficient dynamic ranges are bounded, for example to 10 bits for the C_3 in Equation 4.17; this is definitely acceptable. More importantly, it is now possible to employ cheap fixed-point arithmetic, rather than floating-point, to do polynomial interpolation. With respect to the huge dynamic range, we allow same coefficients, e.g., C_3, from different sections, or different coefficients, e.g. C_3 and C_2, using different scale factors, but coefficients in a single section must use the same scale factor.

The operators for interpolation, such as adders can therefore be simplified. The reasoning is as follows. First, for each interval, the scale factors are known. Second, for each interval, differences in scale factors for the addends are known. Third, there are only a small number of differences between pairs of scale factors, and the adders only need to support these pairs. Fourth, the possible pre-computed shifts, and only those shifts, need to be hardwired (as shown in Figure 4·10). Finally, the pre-computed shift is selected at run time based on the formats stored along with the coefficients. Considering the actual force model, the Lennard-Jones force computation uses 11 formats; the Coulomb force, when treated as a short-range force uses 14.

In the implementation, the same coefficient in different sections has different scale factors, but the same precision, i.e., higher coefficient values in the ill-behaved sections (with numbers of form xxx.x) and lower values where the curve is flatter (x.xxx). Although among sections coefficients and other internal data may have different scale factors, the arithmetic operations are exactly the same, and the operands are of same width as well. Hence, the entire interpolation pipeline, including fixed-point multipliers, adders, and pipeline registers, are applicable to all sections. Furthermore, because the variation of the same coefficient across different sections is limited to a predefined set, the cost of operators to shift operands and results is much less than that for floating-point. We called this numbering system semi floating-point (Semi-FP), because the binary point is able to shift, but only to a few predefined places.

The Semi-FP adder and multiplier are shown in Figures 4·10 and 4·11. The Semi-FP adder aligns operands before performing fixed-point addition, whereas the Semi-FP multiplier does not. Both adder and multiplier need shifted results to the pre-defined formats after the corresponding operations. The format signal instructs the switches to align the data.

Although alignments are still necessary, Semi-FP, compared with floating-point, simplifies operators in four aspects. First, as explained early, the switch (shift) logic is much simpler. Second, the scale factor is only determined at the beginning of the interpolation pipeline; operators do not need to extract the scale factor by themselves. Third, the scale factor is not embedded in the data bit-string, so Semi-FP has more bits available for precision. Forth, if two Semi-FP operators are successive, the output switch of the first operator can be combined with the input switch of the second; this is because it is not necessary to maintain the format of the intermediate result.

Table 4.7 shows the resource usages of operators for floating-point and Semi-FP. LogicCore stands for LogiCore Floating-point Operator IP library from Xilinx, and the data in the table are derived from Version 2.0 [Xil06]. We observe that, Semi-FP, especially the adder, takes less resources than floating-point.

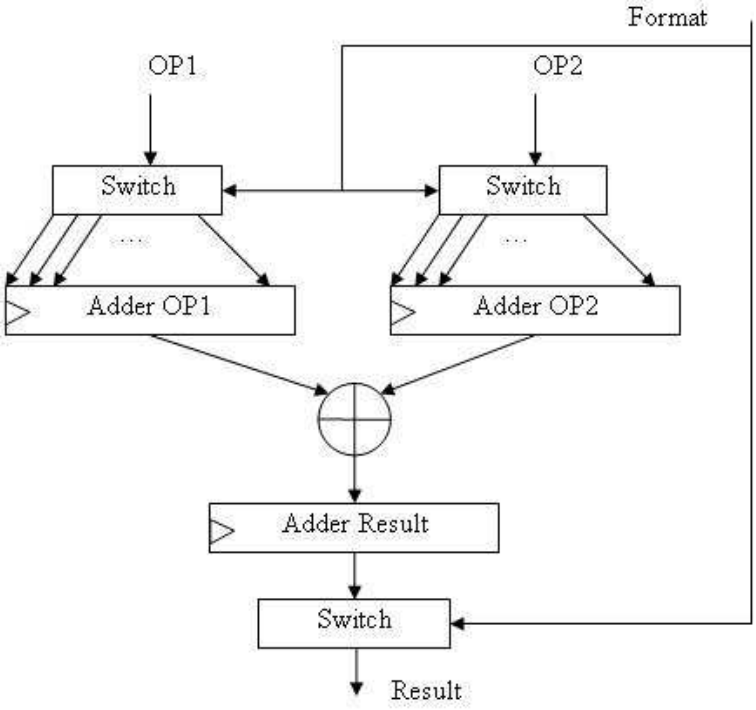


Figure 4·10: Semi-FP Adder

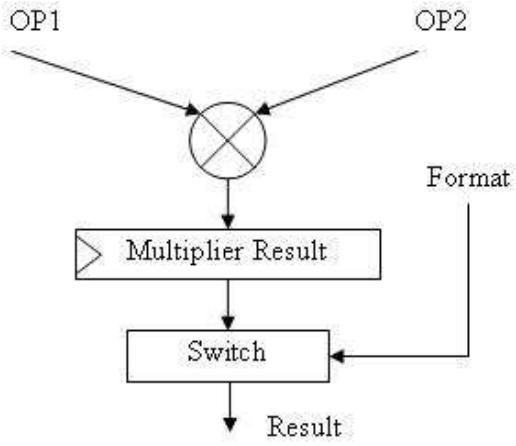


Figure 4·11: Semi-FP Multiplier

Latency is another critical issue about resource consumption. For example, to compute $(A + B) \times A$, the second A must be buffered till $(A + B)$ finishes. In a pipeline, a FIFO is normally used to buffer A, so that every cycle, a new set of (A, B) can be processed. Longer latency of addition here means deeper FIFO requiring more resources. As shown in Table 4.8, Semi-FP has significantly smaller latency than floating-point for adders.

Table 4.7: Resource Usage (in Slices for the Xilinx V2 Family) of Different Components.

Format	Adder	Multiplier	Complete Force Pipeline
LogicCore double precision FP	692	540	19566
LogicCore single precision FP	329	139	6998
Semi-FP – 35 bits	70	316	-
Integer – 35 bits	18	307	-
Combined Semi-FP, integer	-	-	4622

Table 4.8: Latency of Various Components.

Format	Adder	Multiplier
LogicCore double precision FP	12	8
LogicCore single precision FP	11	6
Semi-FP – 35 bits	3	7
Integer – 35 bits	1	6

To implement the pipeline for the complete pair-wise Lennard-Jones force computation, our version uses some integer units as well as 35-bit Semi-FP, but only where no precision can be lost. The overall slice usage is listed in Table 4.7. Semi-FP is more compact than both single precision and double precision versions. A comparison with respect to register usage yields similar result.

4.3 Simulation Quality - Precision vs. Accuracy

In this section we describe the procedure used to determine precision. It is well known that for particular applications, FPGA implementations can achieve speed-ups of 1000×

or more. These applications are characterized by high parallelism, which can be translated into high circuit utilization. They are usually also characterized by low-precision data where the FPGA implementation can trade off data path width for an increased number of function units. Probably for this reason, many researchers are still trying to avoid applications that are "canonically" double precision floating-point, including MD.

We believe that a central area of research in FPGA-based acceleration is analyzing MD to see whether double precision floating-point is actually needed, or whether it is simply used because it has little marginal performance cost on contemporary microprocessors [VHB03, VHB04]. A well-known study by Amisaki, et al. [AFK$^+$95] investigated precision required for MD; they showed that certain important measures relevant to MD simulation quality do not suffer when precisions of various intermediate data are reduced from 53 bits to 25, 29, and 48 bits, respectively. A more extreme observation was made by La Penna, et al. who write that "in our very long simulations we did not see signs of instabilities nor of any systematic drift" due to using single, rather than double precision floating-point [PLM$^+$97]. Clearly, though, this last reference is not the consensus. Now looking at this from the point of view the user, Rapaport states that "obtaining a high degree of accuracy in the trajectories is neither a realistic nor a practical goal. [Rap04]"

The issue of exactly how much precision is required for which particular MD simulations is far from well-studied. This is precisely because MD implementations are nowadays almost universally run on machines where there is little incentive to not using double precision.

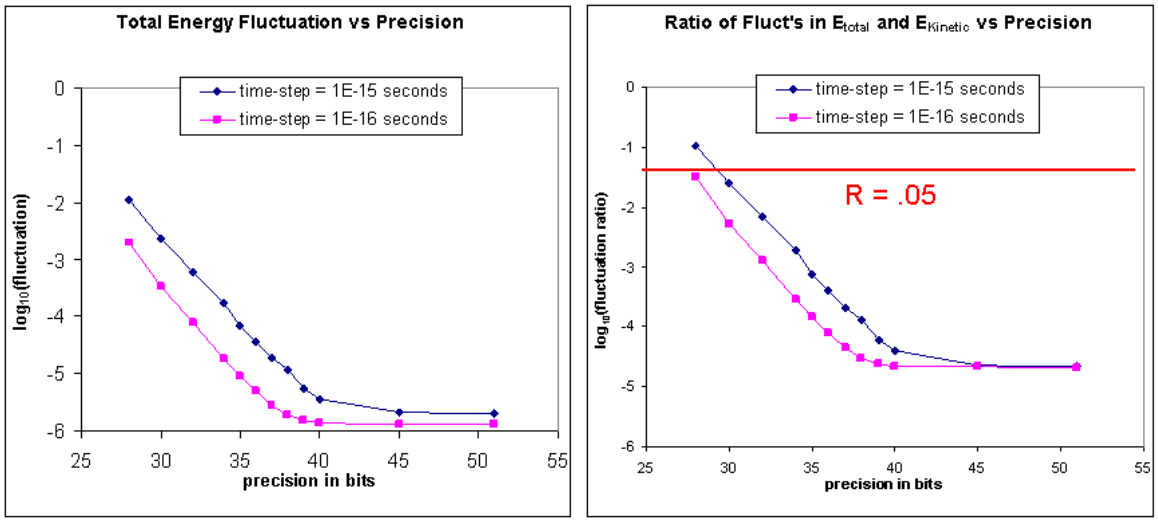
For implementations on configurable circuits, however, the situation is quite different. If it is possible to reduce the precision without appreciably changing the quality of the simulation, then it is possible to increase the computational resources that can be applied. This in turn should result in substantially better overall performance.

Because MD simulations are chaotic, small changes in arithmetic (precision or mode) result in substantial alterations of particle trajectories after only a few collisions. For production users of MD codes, validation is in-the-end accomplished by comparing simulations with experiments. Much more common, however, is to check for simulation quality

by making sure that physical quantities that should be invariant remain so. The relative RMS fluctuation in total energy is defined as:

$$\frac{\sqrt{|\langle E^2\rangle - \langle E\rangle^2|}}{|\langle E\rangle|} \tag{4.18}$$

We ran a set of experiments based on two versions of serial reference code, reproducing as closely as possible the experiments done by Amisaki et al. [AFK+95]. The first used double precision floating-point, the second tracked the hardware implementation, e.g. in varying precision. When the precision of the fixed-point code was set at 50 bits, the results precisely matched that of the floating-point code.



(a) (b)

Figure 4·12: Shown is the effect of precision on two metrics for simulation accuracy: (a) Fluctuation of total energy and (b) the ratio of the fluctuations in total and kinetic energies. Simulations were carried out with two different time-steps.

We also ran a set of experiments to find the relationship between energy fluctuation and precision. In agreement with [AFK+95], we found that the various function units can be tuned independently to derive the optimal FPGA circuits that retain minimal energy fluctuation. For simplicity, however, we present results where the precision of the entire data path is varied in unison. We use two different simulation time scales: time steps were

set to 10^{-15} seconds and 10^{-16} seconds, respectively. A graph showing the results from this set of experiments is shown in the left part of Figure 4·12. One observation is that, in this experiment, a 40-bit datapath results in a similarly low energy fluctuation as a full 53-bit datapath.

Fluctuation of total energy, however, is not the only check that a system is "well-behaved." Another is the ratio of the fluctuations between total energy and kinetic energy $R = \delta E_{total}/\delta E_{kinetic}$. R should be less than .05 [Van04]. We plot R in the right half of Figure 4·12. Note that by this measure, 31 bits are sufficient for time-steps of 10^{-15} seconds and 30 bits are sufficient for time-steps of 10^{-16} seconds. Although greater precision results in "better" behavior, that better behavior may not be needed.

Accounting for the hardware multipliers available on the our FPGAs, we round up to the next time-area discontinuity and obtain a 35-bit data path. This design has a factor of 10x to 50x more energy fluctuation than the best case, but between 100× and 500× lower R than what has been regarded as minimal to indicate "good behavior."

Chapter 5

Algorithm Design Part 2: Long Range Forces

As stated in previous chapters, numerous methods have been developed to compute the long-range part of the slowly converging Coulomb force. Most of them are derived from the famous method of Ewald Sums; usually the critical operation is the 3D FFT. Implementation of 3D FFTs with high precision is still too expensive for current FPGAs to attain speedup, even with carefully optimized IPs, such as those from Xilinx CoreGen [Xil07] and Altera MegaCore [Alt07]. Instead, we choose the multigrid method to compute the long range part of the Coulomb force. The multigrid method has been proved fast and accurate, and matches the functionality of FPGA well.

In this Chapter we demonstrate our long-range force solution using the multigrid method. The presentation has three parts: a general introduction to the multigrid method, the multigrid algorithm for the Coulomb force, and our mapping the algorithm to FPGAs.

5.1 Multigrid Method

The multigrid method is also known as the Multiscale methodology. One way to describe it is as a method of obtaining a global solution by executing local processes at multiple scales. Multigrid is useful in the common case where it is difficult to solve a problem globally because there is too much information to process directly, and the size of solution space is super-linear with respect to the amount of data. In contradistinction, local processes usually generate local solutions, such as local maxima or best approximations, with much less effort. Also important is that the local process compresses the local information, so that the global solution can be developed with the compressed information in a smaller space. So long as the ratio between information and solution space is super-linear, using

local processes can always save the overall computation effort.

Another important part of the multigrid method is the interaction between local and global processes. The local process can be partitioned to finer processes; the global process can be integrated into a coarser problem. Applying this idea recursively, a global process can be decomposed into a hierarchy of local processes. First, the local process compresses local information on a fine level. Next, the compressed information is transferred to a coarser level, so that the local information can be merged there and be "more global." Finally, the global solution is generated with the information merged from multiple levels of local processes. If the processes have the same function and interface, this hierarchical structure can be very regular and preferable for both software and hardware implementations.

A toy example of the multigrid method is a merge sort through a binary tree structure. Assume there is an integer array X[N] of unsorted values. Then N^2 comparisons are needed by naive sorting methods. If the finest local process sorts only two integers, and every coarser process merges two sorted arrays, only $N \log N$ comparisons are needed. During convergence, every local process compresses the information by sorting a partially sorted array, which is constructed by connecting two sorted sub-array. In another words, entropy reduces when an array becomes better ordered. The efficiency comes from two sources: the speed of every local process, and the structure of the Multiscale tree. The former depends on the particular applications; the latter, however, is general. For example, let there be an algorithm that could merge four sorted arrays at once. This would mean that the local process would compress more information than the previous merge algorithm and the rate of convergence would be correspondingly faster.

Complex multigrid applications not only summarize local information to give one global result, but also require results at fine resolution; this depends on global information. Considering the Coulomb force computation, the potential at any give point in the space (result at fine resolution) is influenced by all other charges in the space (global information). For this type of problems, the local-to-global (fine-to-coarse) procedure to merge information

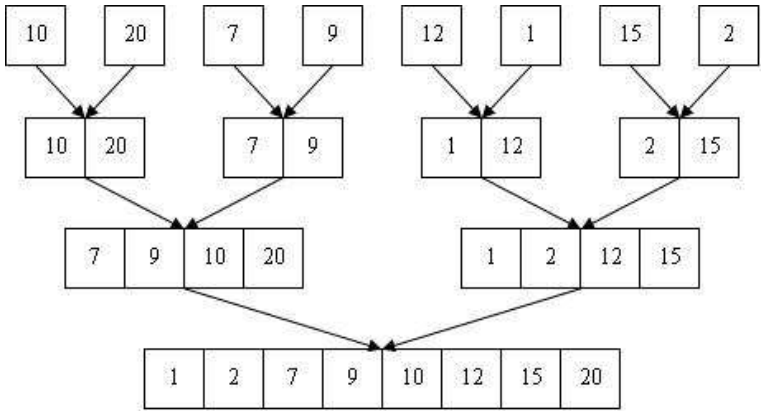


Figure 5·1: Merge Sort through a Binary Tree

is insufficient. Another procedure is required to apply the global solution to local areas (coarse-to-fine). The local process must do additional work: during the local-to-global phase, besides compressing local information for the coarse level process, it must also compute part of the final result. Once the global information is summarized, the local process can extract and apply its influence on the final result (e.g. the potential caused by distant charges). The local processes in the tree structure are thus traversed twice: during the local-to-global and global-to-local procedures. The down and up traversal of the hierarchy is called a V-cycle in the multigrid method.

We now sketch the general multigrid algorithm with PDE notation following the presentation found by Yavneh [Yav06]. Algorithm 1 is a series of recursive V-cycles. The fine-to-coarse operation is referred to as restriction, coarse-to-fine as prolongation.

We begin with q^n, the known parameters on the current grid (e.g. the charge distribution), and finish with the solution un (e.g. the potential distribution) of the PDE $u^n = L \cdot q^n$. The general idea of solving PDEs with the multigrid method is to approximate different components of the solution at multiple scales (levels): the high frequency components are approximated at fine levels, the low frequent components at coarse levels. Because the high frequency components are more local than the low frequent components, they can be computed only with local data, but then with high resolution. Meanwhile, the low frequency components need global information, but tolerate low resolution. This is

Algorithm 1 V-cycle of the general Multigrid

Function u^l =V-Cycle$(u^l, q^z l, l)$

Begin

1. If this is coarsest grid, solve $L^l \times u^l = q^l$ and return u^l.
2. $u^h = Relax0(u^l, q^l, l)$
3. $r^l = q^l - L^l \times u^l$
4. $q^{l+1} = A_l^{l+1} * r^l$
5. $u^{l+1} = 0$
6. u^{l+1} =V-Cycle$(u^{l+1}, q^{l+1}, l+1)$
7. $u^l = u^l + I_{l+1}^l \times u^{l+1}$
8. $u^l = Relax1(u^l, q^l, l)$
9. Return u^l

End

why the local process is allowed to compress information, and the global process can still reach accurate results.

We now return to Algorithm 1. Step 1 solves the PDE directly, if the grid is small enough. Step 2 does the first relaxation to give a guess of the solution. Step 3 computes the residual (error) of the guess. Step 4 restricts the known parameters to a coarser grid (having fewer grid points and unknown variables) with a basis function A. Step 5 specifies the boundary condition, for example setting $u^l + 1$ to zero for a vacuum. Step 6 calls the V-cycle recursively to solve the residual. Step 7 uses function I to prolongate the solution of the residual as a correction from the coarse grid back to the current grid, and integrates the correction with the guess. Step 8 does another round relaxation. Step 9 returns the solution. Figure 5·2 shows a V-Cycle; with a constant number of iterations per level and a geometric reduction in grid points per level, the resulting complexity is $O(N)$.

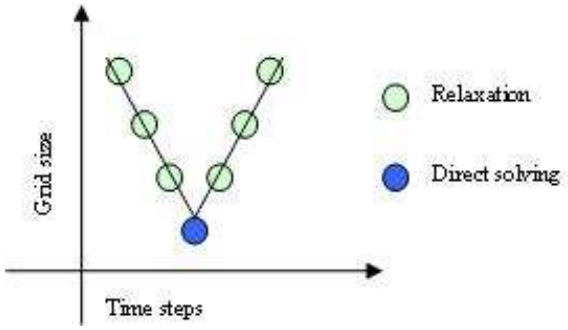


Figure 5·2: V-Cycle

5.2 Multigrid Method for the Coulomb Force Computation

Recall the difficulties with computing the Coulombic force: it converges too slowly to use cell-lists directly, but using a cut-off approximation (shown at right) is not highly accurate. The solution is to split the force into two components (shown below): a fast converging part that can be solved locally, e.g., with cell lists, and the remainder, which is sometimes called the softened part.

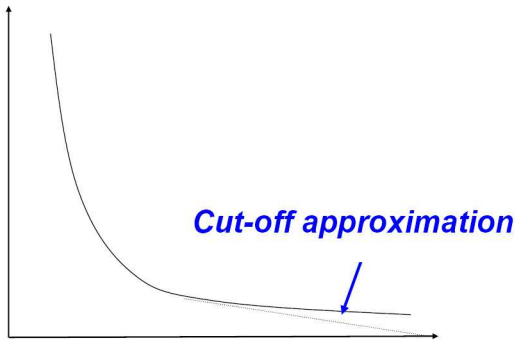


Figure 5·3: Cutoff Approximation of Coulomb Force

This appears to create an even more difficult problem: the softened function converges even slowly than the original. The key idea is to pass the softened function on to the next coarser level, where it is again split. This continues until the coarsest level is reached. There, the problem should be small enough for the direct solution to be efficient.

More formally, the problem of Coulomb force computation is to compute the potential

distribution by solving the Green's function for the given charge distribution. The electrostatic potential is expressed as:

$$V_i^{CL} = \sum_{j \neq i} \frac{q_j}{|r_{ji}|} \tag{5.1}$$

For computational accuracy, 1/r is split into two parts with a smoothing function $g_a(r)$,

$$\frac{1}{r} = (\frac{1}{r} - g_a(r)) + g_a(r) \tag{5.2}$$

so that

$$\frac{1}{r} - g_a(r) \tag{5.3}$$

declines fast enough to be cut-off beyond distance a, while $g_a(r)$ varies slowly with distance. Via the smoothing function, the high frequency parts of $1/r$ become a short range term that can be computed in the same way as the Lennard-Jones force (in O(N) steps). The choice of the smoothing function is beyond the scope of this discussion; it can be a Gaussian distribution, as is used with Ewald Sums, or the one applied in this thesis. The smoothing function, i.e., the PDE to be solved, can be evaluated precisely on a grid when the wavelength of the highest frequency remaining term is equivalent to the grid cell size.

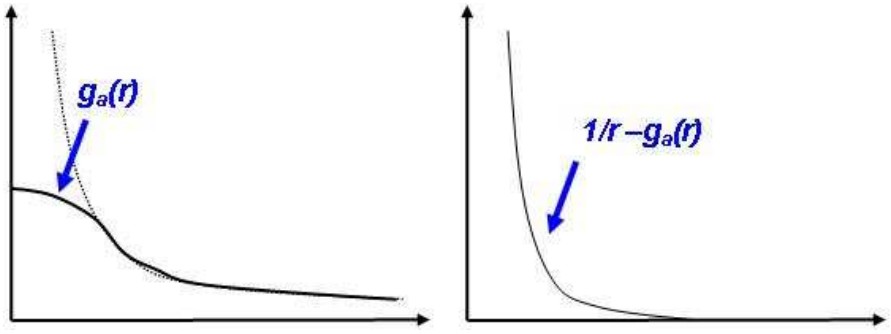


Figure 5·4: Smoothing Function. (a) left is the orignial $1/r$ and the smoothing function $g_a(r)$. (b) right is $1/r - g_a(r)$

Grid-based algorithms map the smoothing function defined in the continuous coordinate space to the one defined in the discrete grid coordinate space. This can be described by the following energy equation:

$$\sum_{i=1}^{N}\sum_{j=1}^{N} q_i q_j g_a(|\vec{r_j} - \vec{r_i}|) \;=\; \sum_{k}\sum_{m} q_{h,m} q_{h,n} g_a(|\vec{r}_{h,k} - \vec{r}_{h,m}|) \tag{5.4}$$

where q_i and q_j are particle charges, and $q_{h,m}$ and $q_{h,n}$ are charges at points m and n on a grid with spacing h. The transform is done by a basis function $\phi(w)$ and will be discussed later. There are many approaches to solve the smoothing functions $g_a(r)$ on a grid, including the Particle Mesh Ewald algorithm, the multigrid based algorithms used here, and even (for small scale problems) direct computation.

We map the smoothing function problem to the multigrid algorithm with the following V-cycle. Given $g_a(r)$ on a grid with spacing h, we define a coarser grid with spacing $2h$ where $g_a(r)$ is smoothed by another smoothing function, $g_{2a}(r)$, as shown in Equation 5.5.

$$g_a(r) \;=\; (g_a(r) - g_{2a}(r)) + g_{2a}(r) \tag{5.5}$$

The local correction $g_a(r)$-$g_{2a}(r)$ takes out the high frequency component of $g_a(r)$ and becomes short range too. The even slower varying $g_{2a}(r)$ can now be approximated more efficiently on the even coarser grid and so on until the coarsest grid. This specifies the first half of V-cycle. On the coarsest grid, the smoothed charge distribution is described with only a small amount of data, so that it can be transformed to the potential distribution by computing Equation 5.1 directly.

After the low frequency part of potential has been computed on the coarsest grid, we still need to incorporate the high frequency component and then to apply it to each particle. The potential on the coarse grid is interpolated back to the fine grid and combined with a local correction on each level. This comprises the second half of the V-cycle. Finally, forces are generated by differentiating the potential on the finest grid in three dimensions, and interpolating the result to each particle's position.

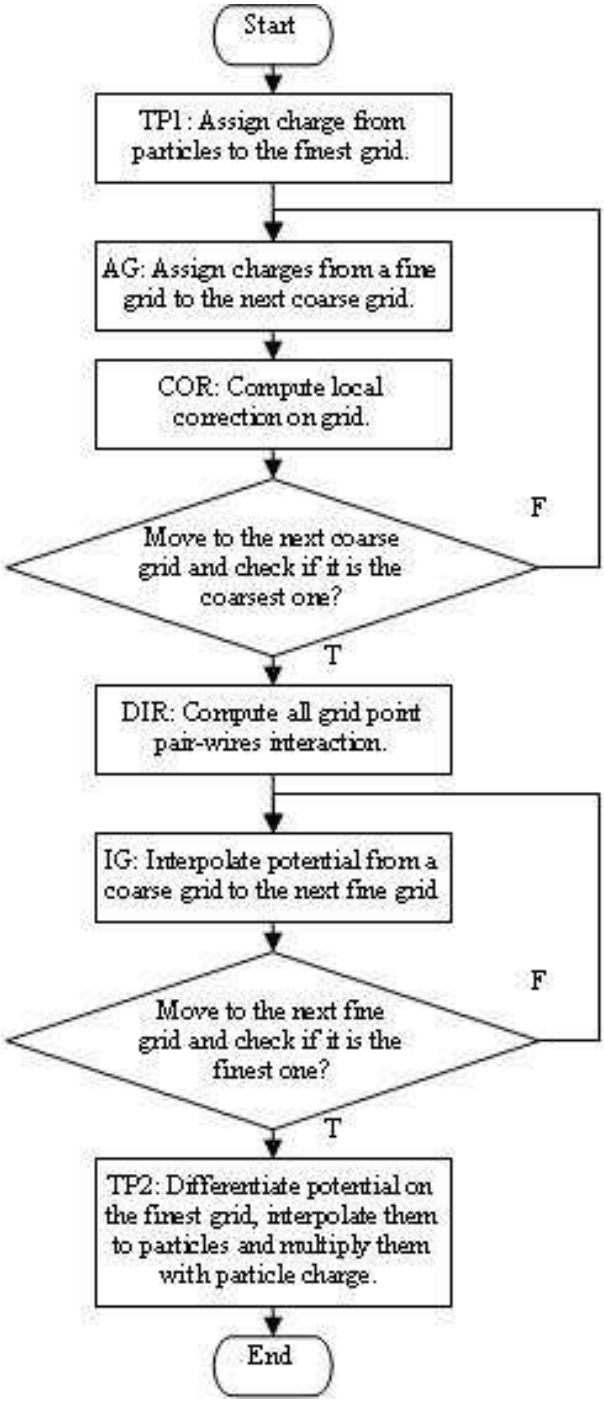


Figure 5·5: Flow Chart of Multigrid Method for the Coulomb Force

The flow chart of this algorithm is shown in Figure 5·5. The operations of this algorithm can be classified into two categories: particle-grid conversion and grid-grid steps. The particle to grid charge assignment (TP1) and the grid to particle potential interpolation (TP2) both apply a basis function $\phi(w)$ and its gradient. A 3rd order and a 5th order basis function are proposed in [STH02]. The assignment is computed as:

$$Q^1(x,y,z) \;=\; \sum_m Q_m \times \phi(|x_m - x|) \times \phi(|y_m - y|) \times \phi(|z_m - z|) \tag{5.6}$$

where $Q^1(x,y,z)$ is the charge on finest grid points (x,y,z); Q_m is the charge of particle m; (x_m, y_m, z_m) are the particle coordinates. The interpolation is computed with:

$$F_{m,x} \;=\; Q_m \times V(x,y,z) \times d\phi(|x_m - x|) \times \phi(|y_m - y|) \times \phi(|z_m - z|) \tag{5.7}$$

where $F_{m,x}$ is the force in the x direction, which is interpolated from the grid point (x,y,z). The basis function $\phi(w)$ must be C^1 continuous, so that the force computed with Equation 5.7 is also continuous.

The other operations (AG, COR, DIR, IG) are of the grid-grid type. They can be represented with 3D matrix convolutions. According to Equation 5.2 and 5.5, the smoothing function matrix on grid Ω_l can be expressed as:

$$G^l \;\approx\; \hat{G}^l + I_{l+1}^l G^{l+1} A_l^{l+1} \tag{5.8}$$

where A_l^{l+1} is the assignment matrix that anterpolates charge from the fine grid Ω^l to the coarse grid Ω^{l+1}; I_{l+1}^l is the interpolation matrix that interpolates potential from coarse grid to fine grid; and $\hat{G}^l$ is the local correction matrix. A_l^{l+1} and I_{l+1}^l apply the same basis function $\phi(w)$. Given a charge matrix Q^l, the potential matrix V^l is computed as:

$$V^l \;\approx\; \hat{G}^l \cdot Q^l + I_{l+1}^l G^{l+1}(A_l^{l+1} \cdot Q^l), l = 1, 2, ..., L-1 \tag{5.9}$$

On the coarsest grid Ω^L, the potential is computed with:

$$V^l \;=\; G^l \cdot Q^L \tag{5.10}$$

where G^L denotes the direct computation between all grid point pairs.

Equation 5.9 reveals that, except on the coarsest grid, three convolutions are performed on every grid: (i) assigning charge distribution to the next coarser grid, (ii) computing the local correction, and (iii) interpolating the potential from the next coarser grid back to current grid. The coarsest grid only has one convolution with G^L. In addition, because G^L, $\hat{G}^L$, A_l^{l+1} and I_{l+1}^l are independent of l and $I_{l+1}^l = (A_l^{l+1})^T$, only three convolution cores need to be precomputed.

5.3 Mapping Multigrid to FPGAs

Following the two types of operations outlined previously, our multigrid coprocessor requires two kinds of computation modules: a particle-grid converter and a grid-grid convolver. Because these operations execute sequentially, the low level computation modules, especially multipliers, can be shared. On-chip memories are also important to the multigrid coprocessor, because they not only store intermediate results as a buffer between successive operations, they also combine computation models of different data-access patterns in terms of architecture. In the rest of this section, we first demonstrate the overall operation sequence, and then present three major components: the particle-grid converter, the grid-grid convolver, and the interleaving memory structure.

5.3.1 Overview

The overall algorithm is shown schematically in Figure 5·6. Starting at the upper left, the perparticle potentials are partitioned into short and long range components. The short range (van der Waals and short range part of the Coulomb force) is computed directly, e.g. with cell lists, while the long range is applied to the finest grid. Here the force is split again, with high-frequency component solved directly and the low-frequency passed on to the next coarser grid. This continues until the coarsest level where the problem is solved

directly. This direct solution is then successively combined with the previously computed finer solutions (corrections) until the finest grid is reached. Here the forces are applied directly to the particles.

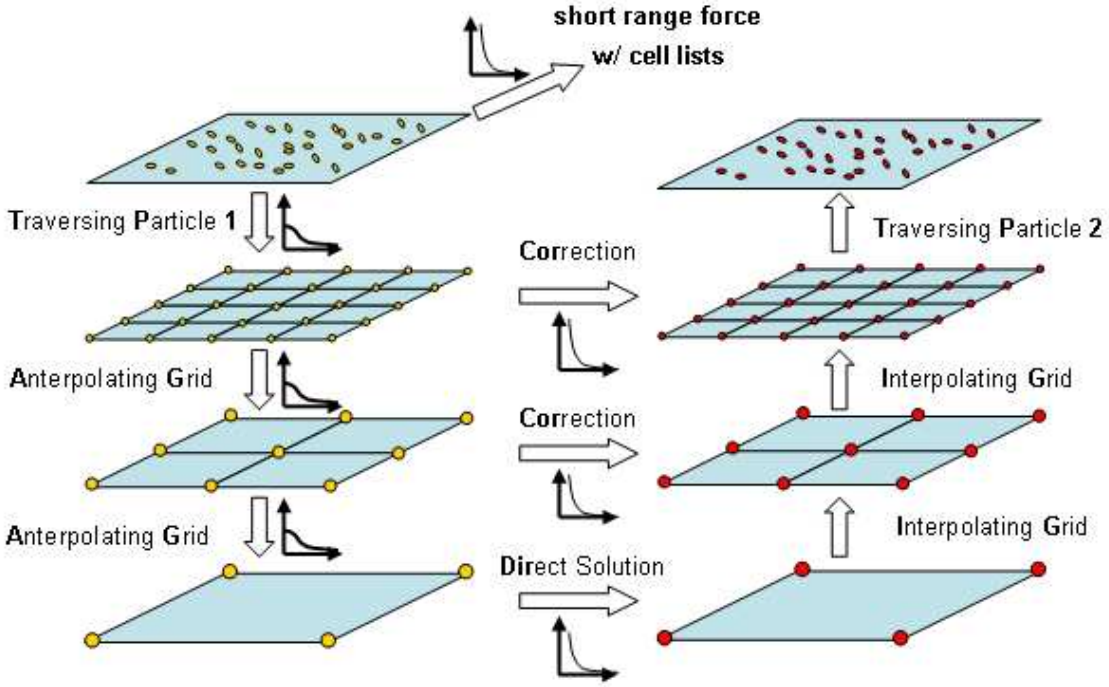


Figure 5·6: Multigrid Method for the Coulomb Force

5.3.2 Particle-Grid Converter

We scale our coordinates to match the finest grid. In one dimension (see Figure 5·7), we can partition the particle position into two components $gi|oi$ where gi is the index of the previous point and oi is the distance from the grid point. It then suffices to use oi alone to compute the contributions of q to any surrounding neighborhood of gi's.

Basis functions $\phi(w)$ (or $d\phi(w)$) are used. This takes 3 steps: (i) scaling the particle coordinates to grid coordinates to extract the grid index gi (x_m, y_m, z_m) and offset oi $(|x_m - x|, |y_m - y|, |z_m - z|)$; (ii) computing $\phi(w)$ (or $d\phi(w)$), the assignment (or the interpolation) weights; and (iii) multiplying the weights by the charge (or the potential) on the grid point. We normalize grid cell sizes to be powers of 2 so that scaling particle

coordinates to grid coordinates only requires zero-cost shifting. The bits to the left of the binary point are then gi, those to the right oi.

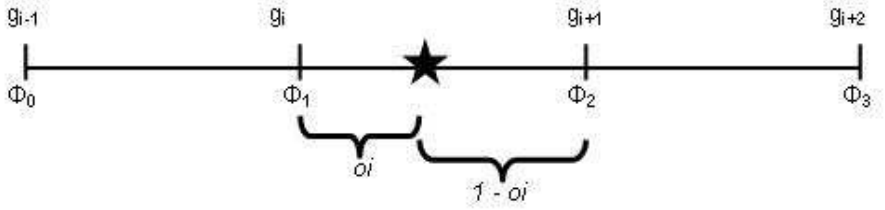


Figure 5·7: Extracting gi and oi from Partilce Coodinates

For $\phi(w)$ and $d\phi(w)$ (we use those derived by [STH02]), instead of computing $\phi(w)$ directly, we modify the basis function to be a set of polynomials of oi for the particle's neighboring grid points. For example, Equation 5.11 is the 3rd order basis function applied by our coprocessor, and w is the distance between particle and grid points:

$$\phi(w) = \begin{cases} (1-|w|)(1+|w|-\frac{3}{2}w^2) & , \quad |w| \leq 1 \\ -\frac{1}{2}(|w|-1)(2-|w|^2) & , \quad 1 \leq |w| \leq 2 \\ 0 & , \quad |w| \geq 2 \end{cases} \tag{5.11}$$

By substituting w with $oi+1$, oi, $1-oi$, and $2-oi$, we have four polynomials in Equation 5.12 corresponding to the four neighboring grid points:

$$\begin{cases} \phi_0(oi) = -\frac{1}{2}oi^3 + oi^2 - \frac{1}{2}oi \\ \phi_1(oi) = \frac{3}{2}oi^3 - \frac{5}{2}oi^2 + 1 \\ \phi_2(oi) = -\frac{3}{2}oi^3 + 2oi^2 + \frac{1}{2}oi \\ \phi_3(oi) = \frac{1}{2}oi^3 - \frac{1}{2}oi^2 \end{cases} \tag{5.12}$$

$$\begin{cases} d\phi_0(oi) = -\frac{3}{2}oi^2 + 2oi - \frac{1}{2} \\ d\phi_1(oi) = \frac{9}{2}oi^2 - 5oi^2 \\ d\phi_2(oi) = -\frac{9}{2}oi^2 + 4oi + \frac{1}{2} \\ d\phi_3(oi) = \frac{3}{2}oi^2 - oi \end{cases} \tag{5.13}$$

Evaluating the basis function polynomials in parallel requires considerable FPGA re-

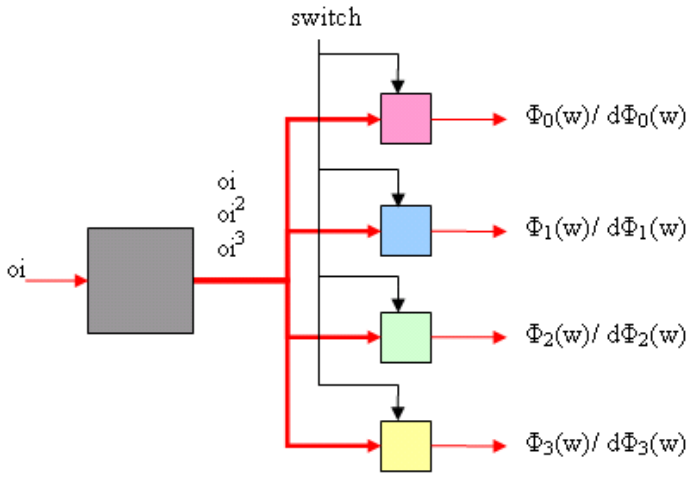


Figure 5·8: Basis function pipeline for $\phi(w)$ and $d\phi(w)$. 'switch' selects output between $\phi(w)$ and $d\phi(w)$

sources, optimization is therefore critical. The basis function pipeline shown in Figure 3 computes all eight polynomials in Equation 5.12 and Equation 5.12 with a common input oi. First of all, because these polynomials are all functions of oi, oi^2, and oi^3, we implement the gray box to generate oi^2 and oi^3 for all expressions. The other four boxes (with different colors) compute $\phi(oi)$ and $d\phi(oi)$ for the four neighboring grid points in one dimension, respectively, as shown in Figure 5·7. Because $d\phi(oi)$ is only needed for the force in Equation 5.7 and $\phi(oi)$ is NOT needed at the same time, $\phi(oi)$ and $d\phi(oi)$ are computed in the same box and selected by the 'switch' signal. After analyzing the structure of the polynomials, further optimization is possible, such as sharing computation between $\phi(oi)$ and $d\phi(oi)$. As to the polynomials of Equation 5.12 and Equation 5.12, even no explicit multiplication is needed by using shifting.

Because evaluating these polynomials—as well as the four for $d\phi(w)$ in Equation 5.13, —in parallel consumes many resources, optimization is critical but possible for the circuits that evaluate these polynomials. The basis function pipeline shown in Figure 5·8 computes all eight polynomials with a common input oi. First of all, because these polynomials are all functions of oi, oi^2, and oi^3, they can share some computations: the gray box generates oi^2 and oi^3 for all other four boxes that compute the eight polynomials. Because $d\phi(w)$ is

only needed for force in Equation 5.7, and $\phi(w)$ is not needed at the same time, $\phi(w)$ and $d\phi(w)$ are combined in the same box, and selected by the *switch* signal.

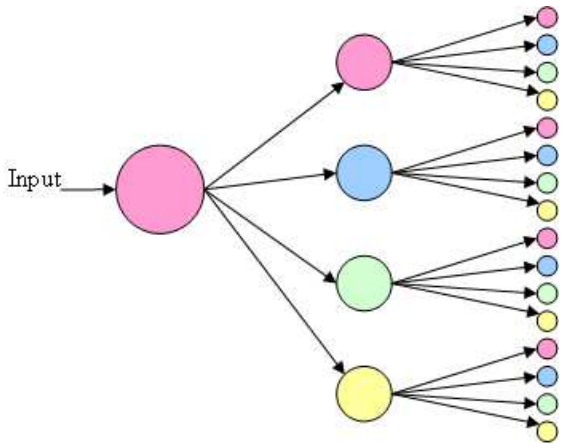


Figure 5·9: One Quarter of a 1:64 Particle-Grid Converter

With a Pth order basis function, one particle is associated with P^3 grid points. Performing the assignment (or interpolation) in parallel both speeds up the computation and reduces the number of basis function pipelines. Figure 5·9 shows one quarter of the tree structure of a $1:4^3$ particle-grid converter. Each color circle multiplies its input by $\phi(w)$ or $d\phi(w)$ from the box of the matching color in the basis function pipeline. Only three basis function pipelines are needed to work with this three-level tree, each pipeline corresponding to the weights along one of three dimensions. The circles of matching color in the last column share the same outputs from a single basis function pipeline (e.g. z direction), as do those in the second column (not shown here) (e.g., the y direction). No such sharing is needed in the first column (e.g., the x direction). This structure has 4^3-way parallelism and 84 multipliers overall (color circles).

During charge assignment, the input is the particle charge; after it is multiplied by the weights in three directions, the outputs from the last column are the charges assigned to the neighboring grid points. For interpolation of the potential, the input is the particle charge as well; the outputs are the weights to be multiplied by the potential from the neighboring grid points. Moreover, because what we really care about is the force on each particle,

the potential is differentiated in three dimensions to yield force fields during interpolation, as in Equation 5.7. The same data (oi and the potentials of the neighboring grid points) must pass through the converter three times, and each time one of the three basis function pipelines has its 'switch' set. Finally, an adder tree does the weighted summation and produces the force. Figure 5·10 exhibits these two processes.

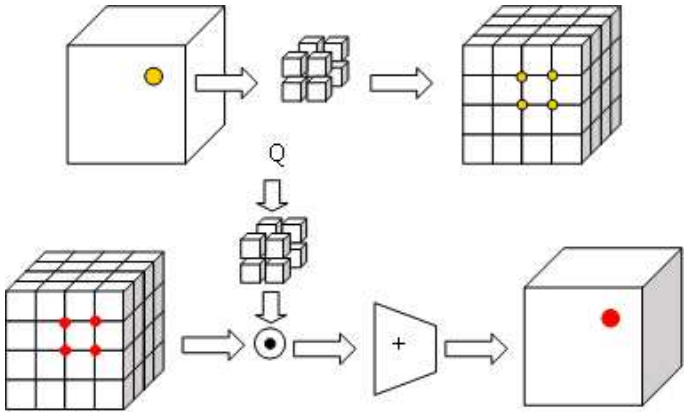


Figure 5·10: Charge assignment (above) and potential interpolation (below)

5.3.3 Grid-Grid Convolver

The grid-grid convolver is extended from our previous work [VH06]. The original design of the systolic array structure is based on the McWhirther-McCanny structure[Swa87]. Without loss generality, let us start from one dimensional convolution $C = A * B$, where $A[0..L-1], B[0..M-1], C[0..(L+M-1)]$.

$$C[k] \ = \ \sum_{i} A[i] * B[k-i] \tag{5.14}$$

The basic operation is multiplication and accumulation (MAC), and is done by one processing element (PE) in the circuit in Figure 5·11. Before convolution, the elements of A are propagated through and stored in the A registers (marked with A[ ]). The Init_A signal is connected to the write-enable of the A registers to enable the propagation. During convolution, the elements of B are broadcast to all PEs, one data per clock. In every cycle,

each PE multiplies the element in the A register by the element of B and accumulates the product with the partial result from the previous PE; the updated partial result is stored in the pipeline register; and the output of the last PE is a single element of C. After all elements of B have been streamed through, another $L-1$ zeros must be broadcasted to flush out the remaining elements of C still in the pipeline registers. At the same time, the A registers are ready to be initialized again.

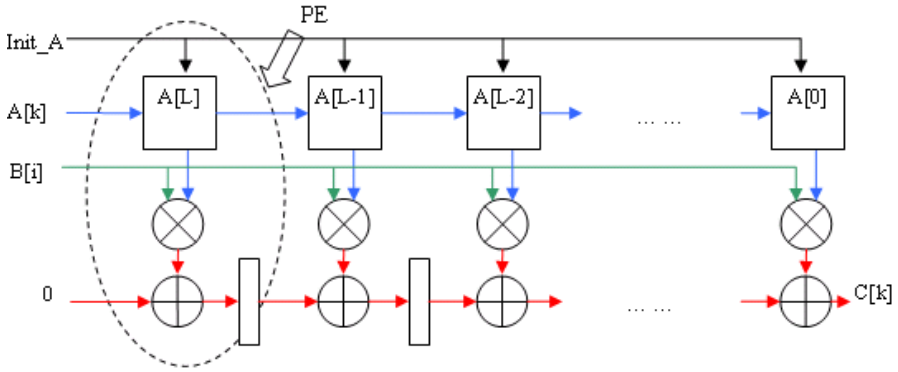


Figure 5·11: 1D Convolver Constructed with Processing Elements

A 2D convolver is constructed with 1D convolvers as shown in Figure 5·12 (a). Because every row of the result matrix C has $L+M-1$ elements, we need a FIFO to hold the extra $M-1$ data before sending them to the next 1D convolver. Since the FIFO is attached after a 1D convolver, it is called a row FIFO. A 3D convolver is constructed with similar structure. In this case, 2D convolvers are connected with planar FIFOs, as shown in Figure 5·12 (b).

The computational capacity of these convolvers is limited by two facts: the number of PEs and the size of FIFOs. The former limits the size of A, while the latter limits the size of B. Because the row and planar FIFOs are usually implemented with on-chip SRAMs while PEs must be implemented with slices, B can be much larger than A; a 1000:1 ratio was implemented in a previous study [VGH04]. Also, we can simply set the FIFO depth in the control logic without modifying connections, but it is extremely difficult to change the length of the PE chain without re-synthesis. These convolvers are difficult to adapt to

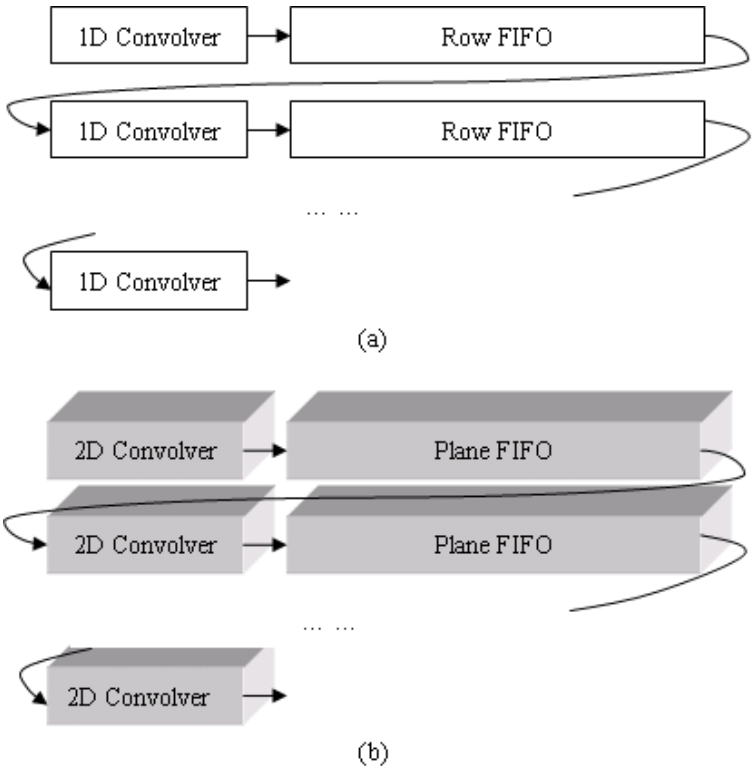


Figure 5·12: 2D/3D Convolver. (a) Top is a 2D convolver constructed with 1D convolvers and row FIFOs. (b) Bottom is a 3D convolver constructed with 2D convolvers and plane FIFOs.

two large matrices of variable sizes because only one of them can be treated as B.

The extension of the convolver for the multigrid coprocessor is that both input matrices are allowed to be larger than the number of PEs in the systolic array, instead of supporting only one large matrix. The motivation is that each multiplication operator requires many hardware multipliers, which limits the number of PEs in this convolver. Therefore, the convolution operations, such as COR and DIR (from Figure 5·6), must be computed in blocks. Also, because of the nature of multigrid, the sizes of the input matrices varies among operations and iterations. On one side, the size of grids varies among different levels; on the other side, the sizes of convolution kernels are not necessary to be same. For instance, the size of COR is determined by the cutoff distance of the smoothing function; the size of DIR is determined by the size of the coarsest grid; and the size of AG and IG

is determined by the supporting range of the basis function. A flexible convolver is thus essential to maintain efficiency.

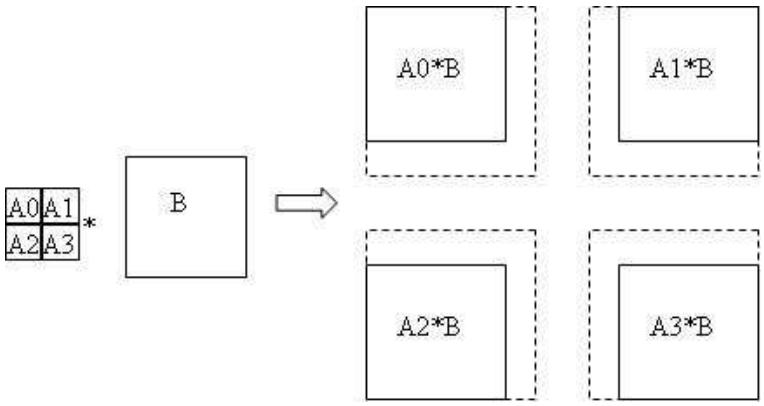


Figure 5·13: Splitting a Convolution

Besides configuring the lengths of the FIFOs to adapt one large input matrix of variable size, we can adapt the other matrix by splitting a convolution into several small pieces and routing results to their proper destinations. As is shown in the example in Figure 5·13, a 2D matrix A is too large to convolve with matrix B directly through the systolic array. It is split into 4 small pieces, A0 to A3, each of which are convolved with B to produce A0*B through A3*B. These are partial results of A*B and spread from the four corners. Summing them up based on their locations yields A*B. Our grid-grid convolver is therefore built with a systolic convolution pipeline wrapped by a complex controller that manages the process. The parallelism is scalable according to the number of PEs that fit on chip, i.e., the scale of the convolution pipeline.

5.3.4 Interleaved Memory

One issue with the particle-grid converter is that a large number of grid points must be accessed on every cycle, such as 64 points with a basis function that supports [-2,+2]; this requires both high bandwidth and highly parallel addressing logic. Fortunately, modern FPGAs, with their hundreds of independent on-chi SRAMs, have just such capability. The interleaved memory design described in [VH06] is one such example.

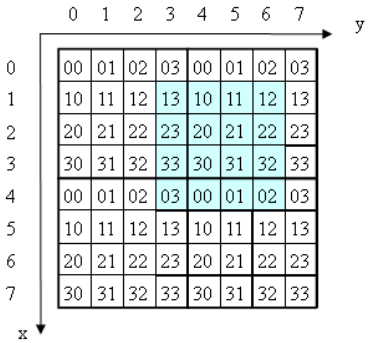


Figure 5·14: A 2D 4^2-way Interleaved Memory

We begin by illustrating the 2D 4×4-way interleaving memory used to record the original 2D grid. Given an address reference (x,y), the grid points within a 4×4 window, i.e. $(x, y), (x, y+1), ..., (x+3, y+3)$, must be accessed. Obviously, 16 independent memory accesses are required for each interleaved memory access. As shown in Figure 5·14, when grid points are stored in 16 separate banks marked from 00 to 33, any 4×4 access window contains grid points from all banks and only one of each. The index of each bank is either the same as that of the bank of the reference address, or the one greater than that in the X and/or Y dimension, respectively. The outputs from the memory banks, i.e., the 00 is always at the top-left corner, not necessary in the expected order, such as the blue window. Here, the data must be shifted (with rotation) in both X and Y directions based on the reference address.

The following example shows the logic how interleaved memory works. Assume that the grids data are stored in 4×4 banks with a zigzag scanning order, i.e., that the grid points from the top-left block are at address 0 in the memory banks, followed by those from the top-right block, then bottom-left and bottom-right. As in Figure 5·14, to access window with the reference address $(1, 3)$ 16 data shown in the blue window are fetched simultaneously. The address of $(1, 3)$ in its memory bank is 0, which is given by expression:

$$bank\_address \; = \; X \times 2 + Y \tag{5.15}$$

where $X = (x \ mod \ 4)$ and $Y = (y \ mod \ 4)$. For the rest grid points, some of them have the same bank addresses as $(1, 3)$, but some do not. For example, the grid point $(1, 4)$ is fetched from memory bank 10, while its bank address is 1, rather than 0. We can tell this adjustment from the reference address as well. For esmaple, the grid right to (x, y) is $(x, y + 1)$; if $y + 1$ produces an overflow beyond 4, the Y in Equation 5.15 must be replace with $Y + 1$. Next, the memory bank outputs are rotationally shifted to the left by $(3 \ mod \ 1)$, and to the top by $(1 \ mod \ 4)$. The inputs to the interleaving memory are shifted in the opposite way before they reach the memory banks. The 3D interleaving memory is analogous.

Chapter 6

System Design

The goal of this research is to build an accelerated MD system on commercial off-the-shelf (COST) platforms. Besides creating FPGA algorithms, system design and implementation is also critical to achieve speedup. Many system design issues needed to be considered and solved in order to create a prototyped system on a real hardware platform. These included problem partitioning, software selection and integration, FPGA design, and communication. Among these, FPGA coprocessor design is perhaps the most critical, because the algorithms discussed in the previous chapter are very complex for current FPGAs, and as we argued in previously, performance is unusually sensitive to the quality of the implementation.

Although our research targets FPGA technology in general, there are still many dependencies with respect to specific hardware. Clearly, the performance depends directly on the achievable clock frequency, the amount of logic, the mix of hardwired components, the off-chip/on-board memory, the host/processor interface, and many other factors. Trivially, these affect the numbers of processing pipelines and their throughput. Not so obvious is their effect on design decisions themselves, such the methods used to manage the memory hierarchy or how bookkeeping overhead is distributed.

In this chapter, Section 6.1 first addresses the overall system architecture and the development environment; Section 6.2 and 6.3 present architectures of the short range force coprocessor and the multigrid coprocessor, respectively; and Section 6.4 explains a cache scheme used to extend system capacity by swapping data between on-chip and off-chip memories.

6.1 System Level Design and Operation

6.1.1 Basic Issues

Because most computation of MD is spent on the non-bonded forces, first priority must be given to their acceleration. The partition of our MD system is thus straightforward: the non-bonded force computation is performed on the FPGA and the remaining computations, such as for the bonded forces and motion update, are computed by the host.

A system-level view of a typical hardware platform is shown in Figure 6·1. It consists of a PC (or other threaded processor) working as the host and two FPGAs being configured as coprocessors. This combination can be used as a complete small-scale MD system or as a computation node in a large system. The interface between the host and the coprocessors, e.g. a bus, is a potential bottleneck in this type of system. We apply the concept of replicated data: each coprocessor has a copy of the data (coordinate, type, and acceleration) of all particles. Coprocessors are therefore self-contained and can operate independently except for data transfer to coordinate updates in the system state.

In the system in Figure 6·1, three computation engines work in parallel in most of time. Once N particles' data are downloaded to the FPGA coprocessors, an $O(N^2)$ (e.g. all-to-all) or an $O(N \times M)$ (e.g. cell-list) force computation can be executed on the coprocessors. Since data is only transferred at the end of a phase (force computation and motion update), and because the amount of data is bounded to at most a few MB, the ratio of computation to communication is very high. This relieves the bottleneck of the FPGA/coprocessor interface. Note that other partitions of the problem, such as would be required to achieve very high speed-ups, would reduce this ratio and motivate a more tightly coupled system. These issues are currently being explored in conjunction with the Desmond system []. For the low-cost solution described here, however, the simple bus-based interface is sufficient.

6.1.2 Basic Operation

We now describe high-level operation of a typical implementation of our system. At every time step (see Figure 6·1), particle coordinates are downloaded from the host to the copro-

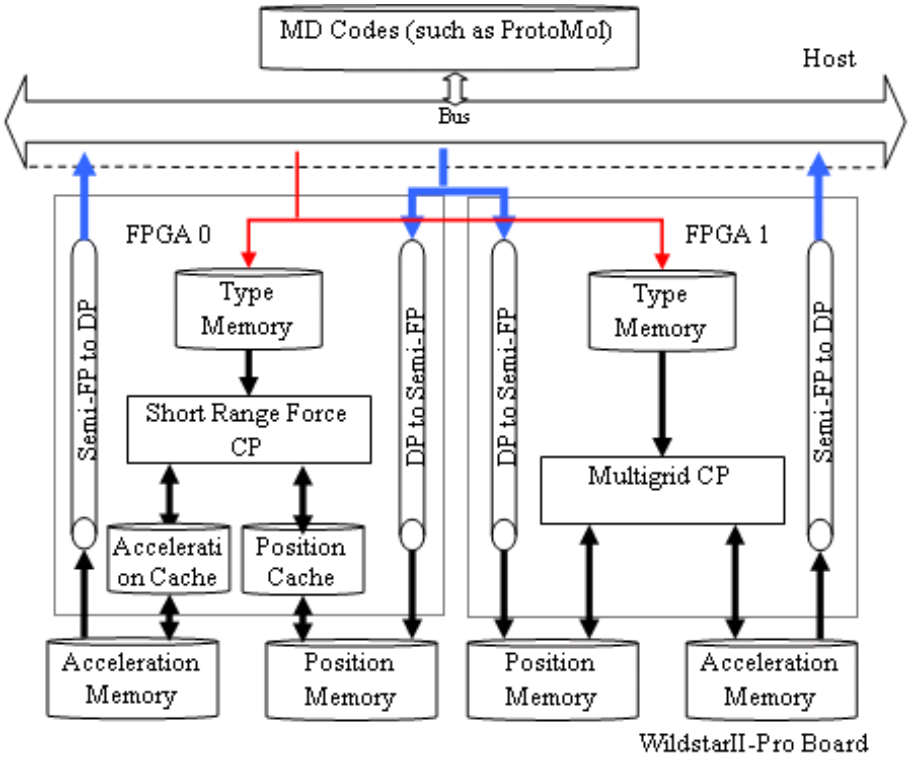


Figure 6·1: System Architecture

cessors through the communication bus. Because the on-chip memory size is limited, most particle data must be stored in the off-chip memory. As a consequence, a caching scheme is implemented that maintains a small fraction of particle state in the on-chip SRAMs and loads data as needed while completely hiding the off-chip memory access latency.

If the data formats are different between the host and the coprocessors, then they are converted on-the-fly during data transfer via the converters on the FPGA. This is necessary here because the host uses double precision floating point, while the coprocessor uses Semi FP. The particle type information is also be downloaded, but no format conversion required. Since the size of type information is much smaller than that of the coordinates and accelerations, they can be stored either on-chip or off-chip depending on the number of particles.

The force computation starts as soon as the transfer of particle coordinates and types finishes. Details about the force coprocessors are given in Section 6.2 and 6.3. Once the

force computation completes, the results, i.e., the accelerations, are uploaded back through the converters and the bus, and on to the host. The final step is cleaning the acceleration memory for the next iteration.

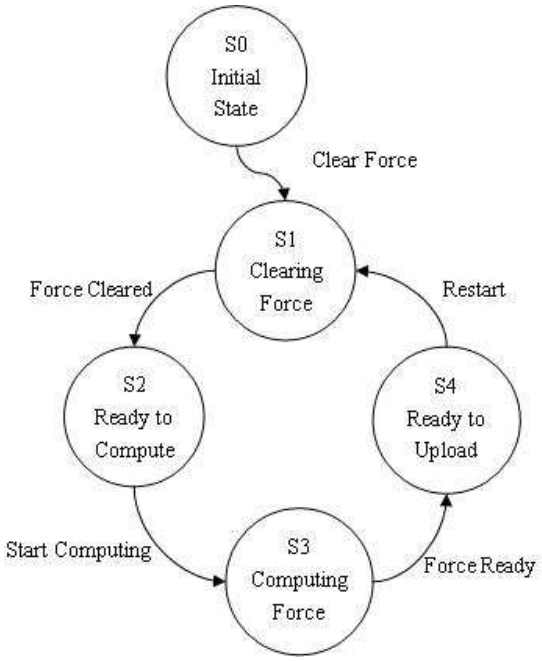


Figure 6·2: State machine that controls data transfer and coprocessor execution

Figure 6·2 shows a simplified view of the coprocessor's top-level state machine, which orchestrates data transfer and coprocessor execution. Before the first force computation, the host explicitly instructs the coprocessor to clear the force memory, and the state machine moves to S2 until the clear operation finishes. Then, the host can start to download particle data. After data download finishes, the start-computing signal is sent to the force computation module. When computation completes, the force computation module issues a force-ready signal, so that the top level state machine can trigger a notification (usually an interrupt) to the host (S4). Then, the host starts the upload. Once the force upload finishes, the restart signal is generated to prepare the coprocessor for the next time step, such as by clearing the acceleration memory. The complete state machine contains more states to control the access timing for the bus interface and the off-chip memory.

6.1.3 Use of Reconfiguration

Reconfiguration is an important capability of this system architecture. The time to load a new configuration into a high end FPGA is small with respect to the time required for either the short- or the long-range force computations for a single iteration. Thus, if there is only one FPGA chip in the system, it can be reconfigured for the long range force and the short range force sequentially. This allows the entire chip capacity to be applied to each computation with little loss in performance.

Another reason to use reconfiguration is to support multiple time-scale force integration. The long-range force may not necessarily be evaluated in every time step (actually every 2nd to 10th time steps is common), because it varies much slower than the short-range and the bonded forces. With reconfiguration, we can keep all FPGA chips computing the short range forces during every time- step, while the long-range force coprocessor is configured once every several time steps. In this case, the reconfiguration overhead is even less significant as it gets amortized over several time steps.

6.1.4 Integration Into Production MD Systems

A central attribute of our system is that it can be integrated into production MD systems. For this study, we adapt the ProtoMol system for acceleration with our FPGA coprocessors. The general idea is to swap functions from software to hardware. There are several issues. (i) the partition between components must be completely clean, i.e., there can be no interleaving of work done by the coprocessor and host other than the global partition. This is trivial in some modular codes, tortuous in others. We have chosen the ProtoMol system because it was designed especially for such module swapping. (ii) Data formats may need to translated between host and coprocessor and back. As we have already described, in our design this is handled entirely by the coprocessor. (iii) Data structures may need to be modified. In this system, we use the cell-list computations performed on the host, but require that data be ordered in a particular way to optimize coprocessor performance. (iv) Some computations still performed by the host after partitioning may need to be

optimized. This situation arises as follows. A module that takes a trivial part of the overall computation time in the unaccelerated code may now be significant when most of the rest of the code now runs at a $10 - 20\times$ speed-up.

Initialization is the first necessary software modification. It includes three steps: (i) FPGA board setup, including clearing interrupts and allocating transfer memories; (ii) FPGA chip configuration, including resetting chips and configuring FPGAs; and (iii) coprocessor initialization, including downloading force parameters and clearing the acceleration memory in preparation for the first time step. The overhead of initialization is trivial in the overall simulation time. What cannot be neglected, however, is the overhead of particle data transfer between host and the coprocessors that occurs during every time step. In addition to communication, pre- and post-processing of are required on the host. This is because the data structure optimized for the software implementation is usually not suitable for hardware. Restructuring introduces extra overhead in both timing and memory. One example is the cell-list implementation for the short range forces, which will be addressed in Section 6.2. Compared with the original software implementation, this overhead may be insignificant, since it is bonded with $O(N)$. However, because it is on the critical path, after accelerating the $O(N^2)$ or $O(N \times M)$ computation with the coprocessors, this overhead becomes more significant and must be optimized.

6.2 Short Range Non-bonded Force Coprocessor

The short range non-bonded forces include the Lennard-Jones force and the short range part of the Coulomb force. Both can be switched to zero if two particles are far away from each other. Because of this property, the cell-list method saves computation effort: only the region near the reference particle need be inspected, rather than the entire simulation space (see Chapter 2).

The forces between particle pairs are computed with the force pipeline array. Parallelism is explored in two dimensions: the replication of the force pipelines, and the concurrency of the pipeline stages. Within the force pipeline, polynomial interpolation is

computed with Semi-FP to approximate the force curve. In this section, the short range non-bonded force coprocessor will be introduced in three parts: the coprocessor architecture with cell-list support, the force pipeline itself, and the polynomial interpolation with Semi- FP operations.

6.2.1 Short Range Non-bonded Force Coprocessor Architecture

Cell-list Representation

In fully software implementations, cell lists are usually represented by an array of linked-lists in which every linked-list is a series of indices pointing to particles of one cell. When particles move among cells, the corresponding indices move among the linked-lists as well, while the layout of particle data (e.g. coordinate, type, and acceleration) remains fixed in memory. The advantage of this method is that less data movement required, because the indices are much compact than the particle data; the disadvantage is that we need a series of random accesses to fetch particles of one cell.

In our FPGA implementation, our short range force coprocessor requires particles of one cell to be fetched in parallel so that they can be sent to force pipeline array with the highest bandwidth possible. Obviously, the link-list method is problematic for our FPGA coprocessor's superscalar architecture. There is certainly enough data bandwidth offered by the on-chip SRAMs; there is not, however, enough addressing logic. This is because, with the fixed particle data layout method, we can not avoid most of the particles from one cell ending up in a single SRAM bank. In this worst (but not infrequent) case, we would have to access them serially. We therefore designed an alternative data structure to implement cell-lists.

Instead of linking indices in a list (as shown in Figure 6·4) and keeping particle layout static, we dynamically group particles by cell in the particle memory. As shown in Figure 6·3, particles from the same cell have the same color and are stored in a single segment. The order of cells in the particle memory is predefined. Thus, we can use an array, the cell-list memory, to indicate the particle-per-cell counts. In practice, every word in the particle

memory usually contains more than one particle (such as two in Figure 6·3) to match the number of pipelines in the force pipeline array. Accordingly, the counts in the cell-list memory are in units of words as well. With this two-level indexing logic (one to locate the cells and the other to locate particles), multiple particles from the same cell can be fetched simultaneously. When the number of particles of one cell is not a multiple of the number of pipelines, some dummy particles must be padded at the end of that cell in the particle memory, so that the first particle of the next cell starts from a word boundary. Because the size of the cell-list memory is very small, we build the cell-list-auxiliary memory in the same on-chip SRAM of the cell-list memory, which indicating the starting addresses of cells in the particle memory.

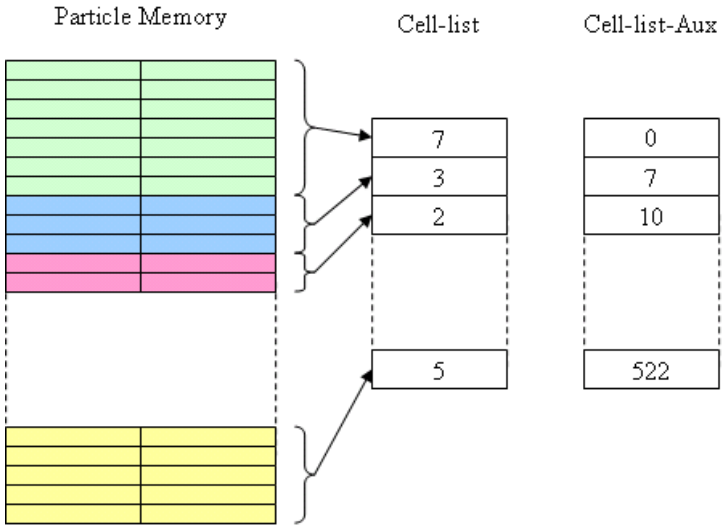


Figure 6·3: Cell-list Representation in FPGA Coprocessor

Data Path and Control Logic

Figure 6·6 shows the architecture of the short range non-bonded force coprocessor with cell-list support. There are three major parts in this architecture. The force pipelines are the computation modules that evaluate the short range forces for given particle pairs. We fit as many pipelines as possible on the FPGA to maximize the parallelism. The off-chip memories store all the particle data, while the caches implemented with the on-chip SRAMs

only contain the working set. The pair-controller generates control signals and addresses to swap particle data between memories and caches and route them through the force pipelines. In this subsection, we explain the logic of the pair-controller and the high-level data path shown in Figure 6·6. The details about the force pipelines and the caches is in a later section.

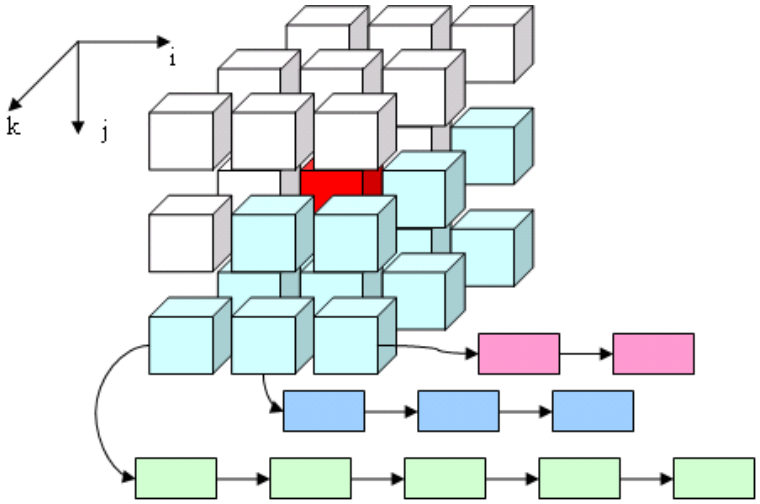


Figure 6·4: Half Cubic Neighbor Pattern

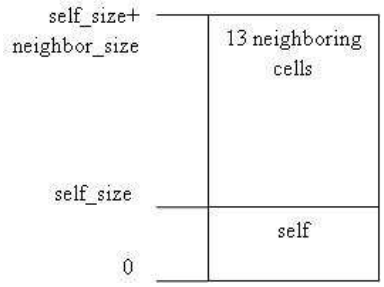


Figure 6·5: Particle Cache Layout

The high-level logic of the pair-controller is described with the pseudo code in Algorithm 2. The outer three loops of *Address_generator*() traverse every cell in the simulation box. The current cell, referred to as $self$, is loaded from the off-chip memory into the cache first. The inner-most loop loads the cells neighboring $self$ into the cache. When the cell size is equal or larger than the short range force cutoff, only 26 adjacent neighboring cells need

to be inspected. In fact, because of the Newton's third law, only half of them are loaded. The blue boxes in Figure 6·4 are one combination of a half of the adjacent neighboring cells. Let the coordinates of the red cell ($self$) be (i, j, k). Then the coordinates of the blue cells are: $(i+?, j+1, k+?)$, $(i+1, j, k+?)$, and $(i, j, k+1)$, where '?' means any of -1, 0, $+1$. *Locate_neighbor*() computes the starting addresses of neighboring cells following this pattern. Finally, $Traverse\_cell()$ fetches particle pairs from the cached cells for the actual force computation.

Reducing the cell size increases the number of neighboring cells within the cutoff. Consequently, we can approximate a more accurate cutoff sphere and remove more particles remote to 'self'. This is the method usually applied in software implementations. For FPGAs, the trade-off is the complexity of the pair-controller versus the number of particles removed. Since even the perfect cutoff sphere will only save less than half of force computation, this optimization is of lower priority.

There are four major signals generated by $Traverse\_cell()$ to control the data path: Pi_addr, Pj_addr, Pi_sel, and Pj_mask. Pi_addr and Pj_addr are the addresses to access particles in cache. Pi refers to the particles in $self$, Pj to the particles the other cells in the neighborhood, including $self$. Because every cache word has N particles (where N is the number force pipelines), Pi_sel is used to select one of them to compute with N Pj particles. In the case that a force pipeline is assigned to compute the force between Pi and itself, the result is excluded by the Pj_mask signal by selecting zero as the force pipeline output. $Traverse\_cell()$ has only two steps: $Traverse\_self()$ and $Traverse\_pairs()$. The former fetches pairs of particles both from $self$; the latter fetches one from $self$ and one from the neighboring cells. Figure 6·5 shows the particle layout in the cache.

$Traverse\_pairs()$ treats all neighboring cells as a big cell and takes further four steps:

1. One cache word of N particles from $self$ is loaded into *array* Pi. Then, one of them is selected with Pi_sel and is copied to the Pi *register*. The Pi *acceleration array* is cleared for later accumulation.

Algorithm 2 Pair-controller Logic

```
Address_generator( )
{
    for(i=0; i<cell_list_size; i++)
    //cell_list_size is the number of cells along one axis.
    {
        for(j=0; j<cell_list_size; j++)
        {
            for(k=0; k<cell_list_size; k++)
            {
                self=i*cell_list_size^2+j*cell_list_size+k;
                Load_cache( self );
                self_size=cell_list_memory[self];
                for(l=0, neighbor_size=0; l<13; l++)
                //only half of 26 neighbors should be traversed
                {
                    neighbor=Locate_neighbor( i, j, k, l);
                    Load_cache( neighbor );
                    neighbor_size+=cell_list_memory[neighbor];
                }
                (Pi_addr,Pj_addr,Pi_sel,Pj_mask)
                        =Traverse_Cell( self_size, neighbor_size );
            }
        }
    }
}
```

2. In the every following cycle, one cache word from the neighboring cells is loaded into the Pj *registers*. The particles in the Pi *register* and Pj *registers* now comprise N particle pairs for N force pipelines. At the end of the force pipelines, the accelerations are accumulated with those of the particles in the Pj *registers*; these accelerations are also summed up through an adder tree before being accumulated and stored in the Pi *accumulation array*.

3. After all the particles in the neighboring cells are computed with the particle in Pi *register*, the next particle in the Pi *array* is loaded into the Pi *register*, and step 2 is repeated.

4. After all particles in the Pi *array* are processed, the results in the Pi *acceleration array* are flushed back to acceleration memory. At the same time, the next N particles in $self$ are ready to load. The process goes back to step 1.

$Traverse\_self()$ has similar steps, except that, the Pj are also from $self$, the Pj_mask does exclusion, but the Pi acceleration array doesn't flush. The pseudo code in Algorithm 3 illustrate the logic of these signals.

6.2.2 Non-bonded Force Exclusion

The problem of non-bonded force exclusion arises as follows. The non-bonded forces (Lennard-Jones force and Coulomb force) exist only between particle pairs without covalent bonds. In another words, the non-bonded forces between bonded particles must be excluded from the non-bonded force calculation.

One naive method is to check whether two particles are bonded before evaluating the non-bonded forces. This method is expensive, however, because it requires on-the-fly bond checking. Another method employed by some MD packages is to combine the bond checking with pair-list construction. In this case, a lists of particle pairs are constructed during the motion update phase, which only contains particle pairs within the short range force cut-off and not excluded. Consequently, only these pairs are processed during the force

Algorithm 3 Traverse_cell, Traverse_self, and Traverse_pairs

```
Travese_cell( self_size , neighbor_size )
{
    (Pi_addr,Pj_addr,Pi_sel,Pj_mask)=Traverse_self( self_size );
    (Pi_addr,Pj_addr,Pi_sel,Pj_mask)
                    =Traverse_pairs( self_size , neighbor_size );
}


Traverse_self( self_size )
{
    for(int i=0; i<self_size; i++)
    {
        Pi_addr=i;
        for(int s=0; s<N; s++)
        {
            Pi_sel=s;
            for(int j=0; j<=i; j++)
            //only half of the pairs shall be computed
            {
                Pj_addr=j;
                if(Pi_addr==Pj_addr)
                {
                    Pj_mask=0x-1<<Pi_sel;
                }
                else
                {
                    Pj_mask=0;
                }
            }
        }
    }
}

Traverse_pairs( self_size , neighbor_size )
{
    Pj_mask=0;
    for(int i=0; i<self_size; i++)
    {
        Pi_addr=i;
        for(int s=0; s<N; s++)
        {
            Pi_sel=s;
            for(int j=0; j<neighbor_size; j++)
            {
                Pj_addr=self_size+j;
            }
        }
    }
}
```

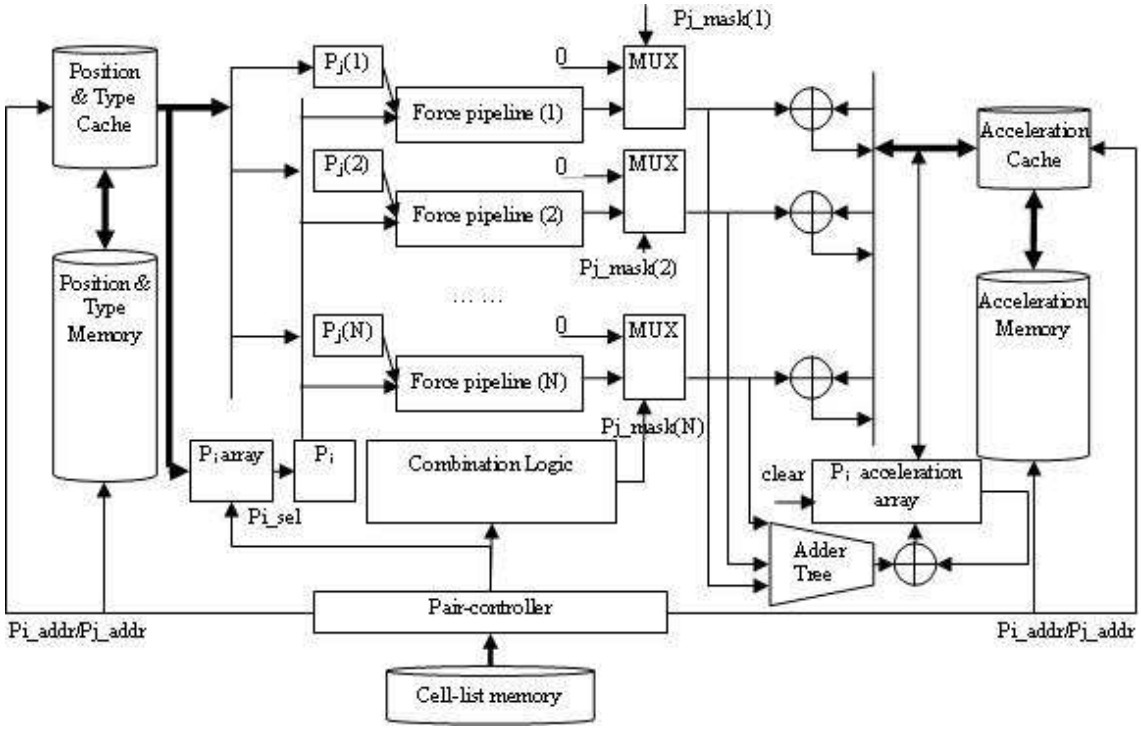


Figure 6·6: Short Range Non-bonded Force Coprocessor

computation phase. This method is more efficient than the naive one, but still requires checking bonds between two particles. This is problematic for our hardware implementation. In order to access particles with pair-lists, particles must have a fixed layout that our cell-list framework expressly avoids. The pair- list is required by every force pipeline, but its size is $O(N)$ and so too big to fit on-chip.

The third method is to compute the non-bonded force between any particle pairs within cut-off and subtract those between the excluded pairs later. The complementary non-bonded forces between excluded pairs are similar to those bonded forces and easy to compute. This method does not need on-the-fly bond checking and is proper for hardware implementation. There is still a problem of this method, however. The complementary force (mostly the r^{14} term of Lennard-Jones force) can be very large because bonded particles can be much closer than non-bonded pairs. Adding and subtracting such large-scale values in floating-point overwhelms the small but real force values. In contrast, adding and

subtracting large complementary forces in fixed-point numbers will not result in loss of precision. The reason is that fixed-point numbers retains bits from the left while floating-point retains those from the right. When a large value is added, a floating-point number will shift its mantissa to the right to fit the large sum, but lose low order bits. Fixed-point number, in this case, will overflow but keep lower bits. Since such large value will be subtracted anyway, the overflow of fixed-point number will be recovered and the precision retained. Meanwhile, floating-point numbers will shift the mantissa to the left, but will not be able to recover the lower bits. In our system, the non-bonded forces are computed on coprocessors with Semi-FP and the complementary force is computed on host along with the other bonded forces. Because mimicking the Semi-FP calculation on host is more expensive than doing floating-point directly, we need another solution to this issue.

Our solution is to apply a short cut-off to the non-bonded force computation, because two non-bonded particles cannot be too close to each other. From the point of view of the force calculation, the unreasonably large non-bonded force values are caused only by particle pairs with narrow separation. Therefore, particle pairs within a certain cut-off must be bonded, and after excluding them the rest of the non-bonded force values have a reasonable dynamic range. It is easy to determine the short cut-off by solving the inequality $\vec{F}^{short}(r^2) < range$, where $range$ is the dynamic range with reasonable force values. Because the dominating term on the left side of the inequality is $\left(\frac{\sigma}{r}\right)^{14}$, and the dynamic range of σ is usually about a factor of 10, we need multiple short cut-offs depending on σ, i.e., the particle type. In addition, since we are dealing with large force values, the short cut-off checking on the short range force coprocessor must exactly match that of the complementary force computation on the host. Hence, we adjust (enlarge) the cut-off slightly so that r^2 can be represented precisely in Semi-FP format. Finally, because the short cut-off checking is performed with Semi-FP and floating-point, there are some extreme cases that one particle pair is excluded by the coprocessor but included in the complementary force again by the host. We solve this problem by counting the excluded pairs on both sides, and if a mismatch happens, the short range forces are totally

recomputed on the host. Since we use either truncation or rounding to zero for Semi-FP operations, r^2 in Semi-FP can only be smaller than in floating-point, which means that the short range force coprocessor will exclude no less pairs than the host, and whenever there is a mismatch, we shall recompute the short range forces for that time step. Fortunately, such mismatches happens less than once every 1000 time steps in experiments, so it does not have obvious effect on performance.

6.2.3 Short Range Non-bonded Force Pipeline

The force pipeline is the compute engine of the short range force coprocessor. The overall system has a modular design so that the pipeline can be swapped according to the force model.

We begin the description an overall description, beginning with its interface. This can be split into two parts: scalable initialization ports and fixed data path ports. The former are used to initialize the force parameter memory. These force parameters include ϵ and σ for the Lennard-Jones force, and particle charges for the Coulomb force. The data path ports are used to transfer particle data, such as coordinates, types, and forces, during the force computation. The force parameter memories are used by every force pipeline. We have built them using the on-chip SRAMs so that they can scale with the number of force pipelines. This number, in turn, is determined by the chip capacity. In addition, multiple force pipelines can share one instance of the force parameter memory via the multi-port interface provided by the on-chip SRAMs. The force pipeline is fine grained, with 61 stages; its implementation is exhibited below. In the rest of this section, we describe in more detail a force pipeline that computes the combination of the Lennard-Jones force and the short range part of the Coulomb force.

As shown in Figure 6·7, the pipeline stages can be grouped into 8 major steps, where r is the distance between particle i and particle j:

1. Compute the displacement in each dimension.

2. Perform periodic boundary refolding if necessary.

3. Compute the square of the distance between particle pairs, r^2.

4. Check the distance. If the r^2 is out of range, a flag is set to force the output to zero at the end of pipeline.

5. Interpolate r^{-x} with r^2.

6. Look up the pre-computed force parameters based on the particle types, e.g., the Lennard-Jones force parameters ϵ and σ.

7. Apply force parameters with r^{-x} to get the pseudo-force.

8. Multiply the pseudo-force with the displacement vector from step 1 to get the force.

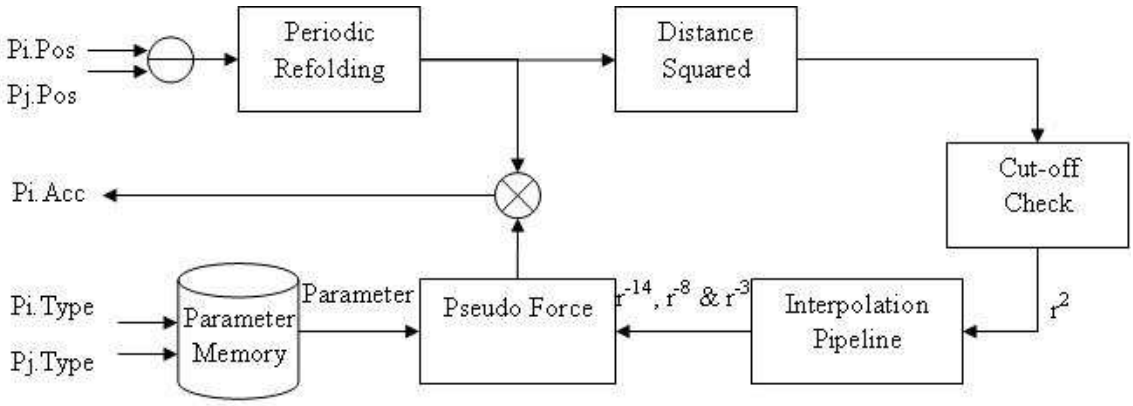


Figure 6·7: Short Range Non-bonded Force Pipeline

Steps 1, 2, 3, 6, and 8 are straightforward: they require only fixed-point arithmetic operations or memory accesses. Step 5 performs the polynomial interpolation with Semi-FP and will be explained in detail in Section 6.2.4. In the rest of this section we give further explanations of steps 4 and 7.

Step 4 checks cutoffs for both short and long ranges. The short range cutoff is checked to exclude the non-bonded force computation among particles with covalent bonds as explained in the previous section; the long range cutoff is checked to switch off the short range forces. In software implementations, since cell-lists have already excluded particles

far away from one other, it is usually not necessary to check this cutoff again. In our coprocessor, however, our cell-list is coarse grained. And because the force is computed in a pipeline, excluding particle pairs from processing does not save any execution time. A more important consideration for this implementation has to do with the Semi-FP interpolation. The r^2 in the force pipeline can only represent values slightly large than the long range cutoff, because we need to save precision of r^2 to maintain interpolation accuracy. If the cell size is same as the switching cutoff, the largest value for r^2 can be 12 times the square of the cutoff, requiring another 4 bits. Therefore, we need to make sure that the two particles are within the distance, which can be presented by r^2 in Semi-FP; otherwise the result is meaningless. Hence, to optimize this computation, we compute r^2 with a lower precision– but a larger range of values–to check the long range cutoff. Meanwhile, the fine-grained format r^2 has been computed and is available for interpolation.

When only computing the Lennard-Jones force, step 7 follows Equation 4.4. This applies the Lennard-Jones force parameters with r^{-14} and r^{-8}. If the short range part of the Coulomb force is also computed here, the pseudo force becomes:

$$\frac{\vec{F}_{ij}^{short}}{\vec{r}_{ij}} = (A \times r^{-14} + B \times r^{-8}) + QQ \times (r^{-3} + \frac{g'_a(r)}{r}) \tag{6.1}$$

A and B are two pre-computed Lennard-Jones force parameters; QQ is the pre-computed product of charges of these two particles; $g_a(r)$ is the smoothing function for the short range part of Coulomb force computation; and a is the cutoff distance of $g_a(r)$. The first term is for the Lennard-Jones force, the second term is for the short range part of the Coulomb force. In particular, if the C^2 continuous smoothing function Equation [STH02] Equation 6.2 is used,

$$g_a(r) = \frac{1}{a}\left(\frac{15}{8} - \frac{5}{4}(\frac{r}{a})^2 + \frac{3}{8}(\frac{r}{a})^4\right) \tag{6.2}$$

the pseudo force becomes:

$$\frac{\vec{F}_{ij}^{short}}{\vec{r}_{ij}} = (A \times r^{-14} + B \times r^{-8}) + QQ \times (r^{-3} + \frac{5}{2a^3} - \frac{3}{2a^5} \cdot r^2) \quad (6.3)$$

For a C^3 continuous smoothing function [STH02],

$$g_a(r) = \frac{1}{a}\left(\frac{35}{16} - \frac{35}{16}(\frac{r}{a})^2 + \frac{21}{16}(\frac{r}{a})^4 - \frac{5}{16}(\frac{r}{a})^6\right) \quad (6.4)$$

the pseudo force becomes:

$$\frac{\vec{F}_{ij}^{short}}{\vec{r}_{ij}} = (A \times r^{-14} + B \times r^{-8}) + QQ \times (r^{-3} + \frac{35}{8a^3} - \frac{21}{4a^5} \cdot r^2 + -\frac{15}{8a^7} \cdot r^4) \quad (6.5)$$

Once r^{-3}, r^{-8}, and r^{-14} are computed by the interpolation pipeline, the pre-computed coefficients and parameters are applied through a simple pipeline to yield the pseudo force, as shown in Figure 6·8 . Because r^{-3}, r^{-8}, and r^{-14} are not required until the middle of the pipeline, some stages that compute the smoothing function can be parallelized with the interpolation pipeline.

6.2.4 Polynomial Interpolation Pipeline with Semi-FP

The polynomial interpolation pipeline is the core of the short range non-bonded force pipeline. It takes r^2 as input and does piece-wise interpolation to approximate r^{-x} with Semi-FP computations and the logarithmic interval scheme in the table look-up. The mathematical background of our interpolation method and details about Semi-FP operations have been addressed in a previous chapter. Now, we show how to integrate these ideas into the polynomial interpolation pipeline.

The basic function of this interpolation pipeline is to compute the polynomial

$$f(x) = C_0 + C_1 \times (x-a) + C_2 \times (x-a)^2 + C_3 \times (x-a)^3 + \ ... \ + C_p \times (x-a)^p. \quad (6.6)$$

Figure 6·10 shows the data path for 3rd order interpolation. The first block interprets

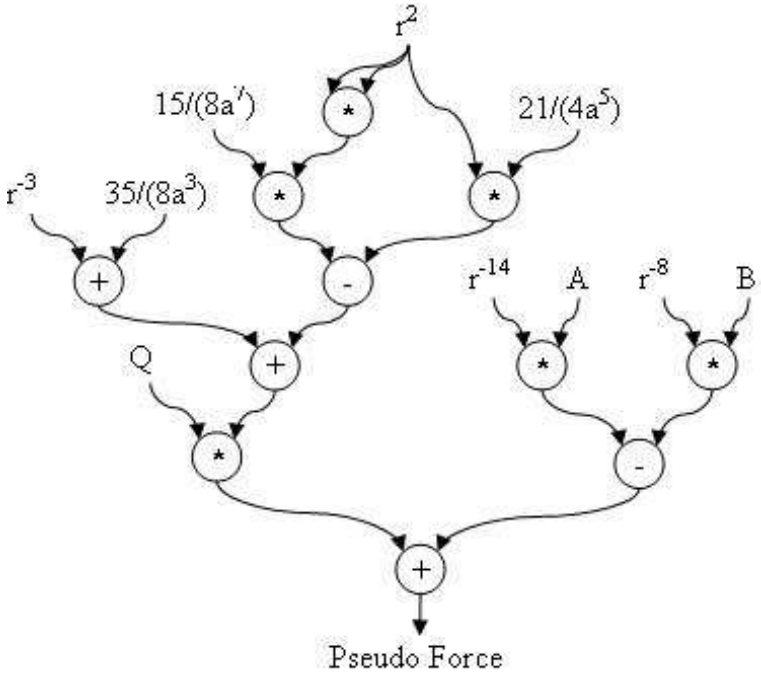


Figure 6·8: Short range non- bonded force pipeline data flow for the C^3 smoothing function

r^2; the coefficient memories store the interpolation coefficients (C_0 to C_3 for a third order interpolation); the remaining blocks compute the interpolation polynomial with Semi-FP adders and multipliers as shown in Figures 4·10 and 4·11. r^2 is interpreted to extract several inputs into the interpolation pipeline.

A simplified example is shown in Figure 6·9. Point a is the left end of an interval. The interval index and the offset into the interval can be extracted from the input, (x), without any additional computation on FPGA, if the section edge and the number of intervals per section are both power of 2. The section index (i.e. $Format$) is determined by the position of the leading 1 in x. For example, let us assume an interpolation input ranging from 2^{-4} to 2^2 and that each section has 16 intervals. In this case, we define the scale factor as -14, i.e., that the most significant bit is 2 ($2^{16-14-1}$), the 2nd bit 1, and so on. As shown in the bit string x in Figure 6·9, because the 6th bit from left is the leading '1', this number can range from 2^{-4} to 2^{-3}, i.e., the first section from left, where the size of each interval is 2^{-4-4}. The next four bits, 0010, indicate that x locates in the 3rd interval from 2^{-4}, i.e.,

the interval from 0.078125 to 0.08203125. The remaining (low-order bits), 110000, are the offset from 0.078125, which is 0.0029296875.

Now, let us inspect the real implementation for interpolating $R_{14} = t^{-7} = (r^2)^{-7} = r^{-14}$. The interpolation input $t = r^2$ has fixed-point format. Because t is defined within the interval 2^{-4} to 2^7 for the Lennard-Jones and the short range part of the Coulomb force, there are 11 sections in our logarithmic interval scheme (see Section 4·10). Thus, the scale factor of t is 7 to fit 2^7, and 11 formats are required to support the 11 sections. In addition, each section is split into 2^7 intervals. The four steps to compute an interpolation polynomial are as follows:

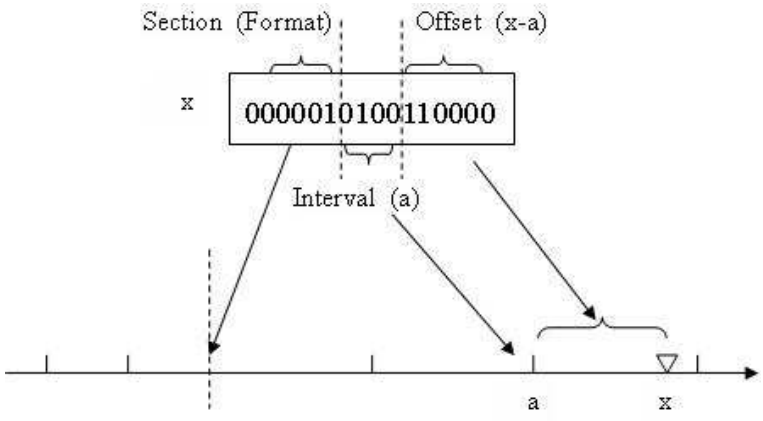


Figure 6·9: Extracting section (i.e. format), interval, and offset for Interpolation

1. Extract $format$, a, and $(t-a)$ from t. The method is shown in Figure 6·9. Locating the leading 1 in t is the most complex part in this step and is done by a module called MSBCheckTree shown in Figure 6·11. For section $(2^{-4}, 2^{-3})$, $(t-a)$ has a scale factor of $-4-7=-11$; this continues, e.g. with section $(2^{-3}, 2^{-2})$ having a scale factor of $-3-7=-10$. The factor is -7, because each section is cut into 2^7 intervals. From now on, the intermediate results have different formats if they are corresponding to different interpolation sections. The information extracted from t is applicable to all interpolation polynomials for the various r^{-x} interpolation pipelines.

2. $C_3 \times (t - a)$. C_3 is fetched from the coefficient memory with the address composed with $format$ and a: $format$ indicates the section and a indicates the offset within a section. The scale factor can be determined with Equation 4.17, e.g., $47 = log(\max_{2^{-4} \le t < 2^{-3}} \{|C_3|\})$ for section $(2^{-4}, 2^{-3})$. The multiplication is performed by a Semi-FP multiplier.

3. $C_3 \times (t - a) + C_2$. C_2 is fetched in the same way as C_3. Its scale factor in $(2^{-4}, 2^{-3})$ is $41 = log(\max_{2^{-4} \le t < 2^{-3}} \{|C_2|\})$. The addition is performed by a Semi-FP adder.

4. Repeat steps 2 and 3 until $t^{-7} = ((C_3 \times (t - a) + C_2) \times (t - a) + C_1) \times (t - a) + C_0$ is computed. In the current implementation, the outputs are in Semi-FP format, i.e., t^{-7} has different formats for different sections. For example, for section $(2^{-4}, 2^{-3})$, t^{-7} has a scale factor of $-4 \times (-7) = 28$, while the scale factor is $-3 \times (-7) = 21$ for section $(2^{-3}, 2^{-2})$.

Because the interpolation pipeline outputs (r^{-x}) are in Semi-FP format, the Semi-FP multipliers must be used to combine the r^{-x} with force parameters to compute the pseudo-forces. This is an issue because the force parameters are in fixed-point format. The outputs of these multipliers have same formats regardless of interpolation section. Hence, the Semi-FP operations end here, and 'format' is no long needed.

Figure 6·11 shows the block diagram of MSBCheckTree. This function locates the position of the leading 1 of $t = r^2$ through a hierarchical structure. The MSBChecker module searches for the leading 1 in its input bit string and generates two outputs: the position of the leading 1 and a flag bit indicating whether the string is all 0s. Consider, for example, a two-level MSBCheckTree. On the top level, the input bit string is split into multiple substrings, so that they are searched locally and in parallel by the MSBChecker modules. On the bottom level, the MSBChecker module checks the bit string that is composed with the flag bits from its upstream MSBCheckers. The leading 1 in the input bit string indicates which substring (called the winning substring) contains the leading 1 of the entire string. Combining the position result of the bottom level of MSBChecker with

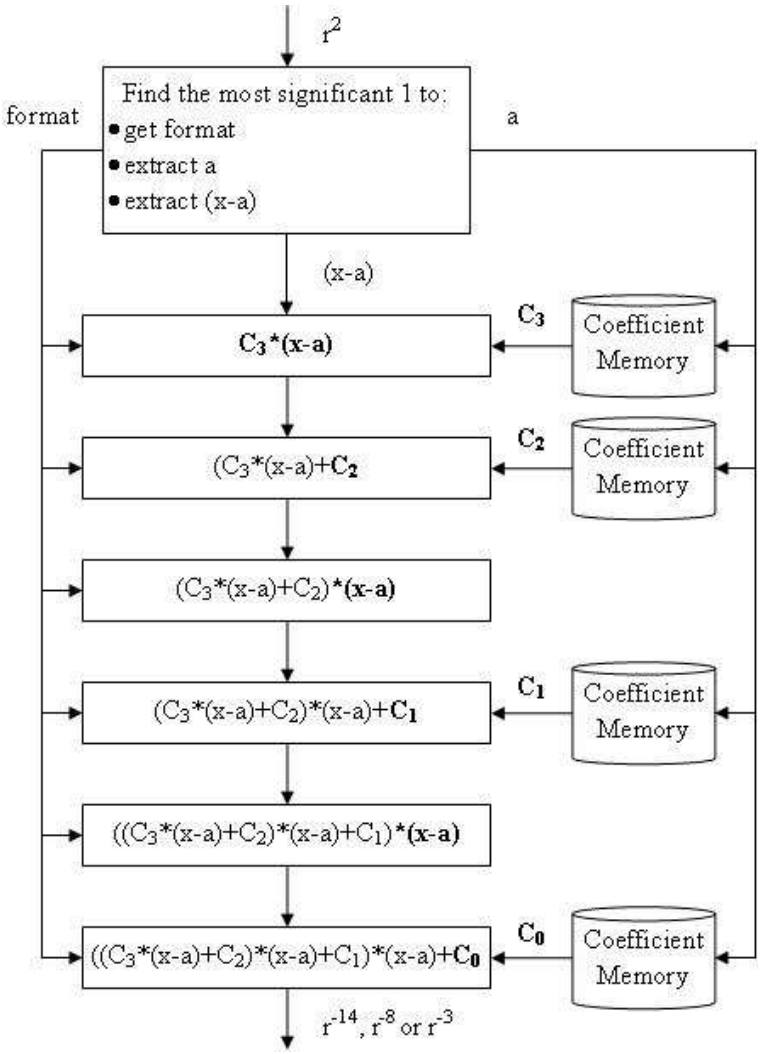


Figure 6·10: Interpolation Pipeline for r^{-x}

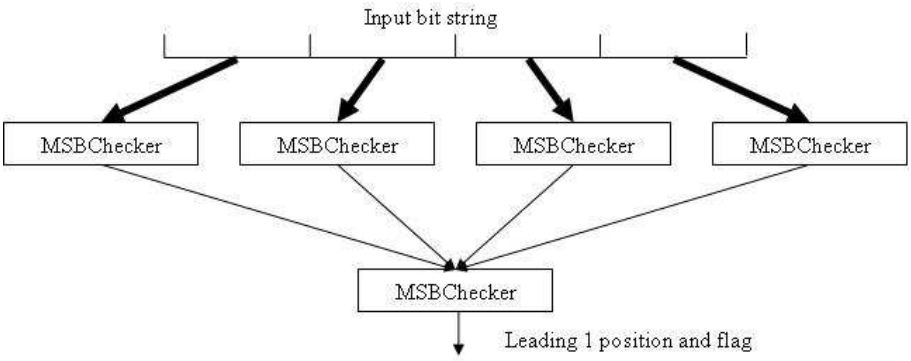


Figure 6·11: MSBCheckTree is constructed with a hierarchy of MSBCheckers.

that of the winning substring derives the global leading 1 position. A large MSBCheckTree works as a recursive to the two-level example: the local search results converge through the tree, and eventually the root MSBChecker yields the global leading 1 position and the flag bit.

6.3 Multigrid Coprocessor for Coulomb Force

The multigrid coprocessor computes the long range part of the Coulomb force. The algorithm and the FPGA design of critical modules have been discussed in Chapter 5. In this section, we first present details of the overall design of the multigrid coprocessor, especially the control logic and interfaces among modules in the data path. In the second part of this section, we discuss implementation considerations with respect to possible design trade-offs.

6.3.1 Multigrid Coprocessor Architecture

Figure 6·12 shows the major components and the data path of the multigrid coprocessor. It consists of a particle-grid converter, a grid-grid convolver, two instances of the interleaved memory, some dual-port memories, control logic, and some miscellaneous components.

Recall the multigrid method for the Coulomb force, five operations (TP1, TP2, AG, IG, COR, and DIR) are executed sequentially in a V-cycle (see Section 5.2); between pairs

of operations data are stored in the memories.

To explore the data path in Figure 6·12, let us start with the two major computation modules, the particle-grid converter and the grid-grid convolver, beginning with the former. The inputs of the particle-grid converter are the particle charge and oi, the least significant bits of its coordinates; the outputs are the weighted charges assigned to the neighboring grid points. Particles are processed sequentially. Before a particle reaches the particle-grid converter, its coordinates and type must be fetched from the position and type memories (shown in Figure 6·1). The address is generated by the control logic. Next, the type-parameter memory converts type into charge. To store the assigned charges into the interleaved Q memory during TP1, or to fetch potentials from the interleaved V memory during TP2, we require that the memory address gi, which is dispatched from the particle coordinates, indicate the neighborhood of the particle being processed. In TP2, the assigned charges are also multiplied with the potentials on the finest grid via a vector multiplier. Finally, the outputs from the vector multiplier are summed via an adder tree to compute the force on that particle.

The other major component of the data path is the grid-grid convolver, which is used in AG, COR, DIR, and IG phases. Its input is either charge or potential on grids and the predefined convolution kernels according the computation phases; its output is either charge or potential on grids. Because the input and output of this convolver during computation are only one datum at each end, regular dual-port memories are good enough for Q and V memories. During convolution, the control logic generates the address of the source memory, selects the correct data with a MUX before the grid-grid convolver, and generates the address and write-enable signal to store the convolution result back into the destination memory. When doing convolution on the finest grid, the interleaved memories must be accessed like a regular dual-port memory. In this case, their address is not gi but rather one generated by the control logic. The MUX above the particle-grid converter selects the proper address according the phase.

One of the components not shown in Figure 6·12, but worth noting, is the memory

used to stor the convolution cores for G^L, $\hat{G}^l$, A_l^{l+1} and I_{l+1}^l (see Section 5.2). The control logic generates the addresses to fetch both the convolution cores and the control signals that initialize the convolver (see Section 5.3.3). The particle-grid converter, the grid-grid convolver, and the vector multiplier all require a large number of multipliers. Fortunately, because the convolver does not at the same time as the other two components, they can share some multipliers.

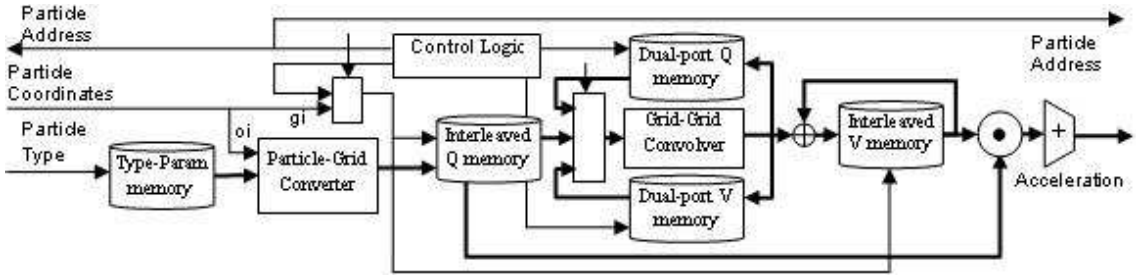


Figure 6·12: Multigrid Coprocessor

The control logic routes data through the data path following the sequence of the V-cycle specified in the flow chart in Figure 5·5. The following six steps elaborate data movement, critical control signals, and addresses from the point of view of timing.

1. The particle coordinates and charge are fetched from the off-chip particle position memory and on-chip Type-Param memory. The coordinates in every direction are split into two parts *oi* and *gi* as described in Section 5.3. The particle-grid converter computes the charges assigned to the neighboring grid points with *oi*. The assigned charges are stored into the interleaved Q memory with address *gi* simultaneously.

2. After charges are assigned, the charge grid convolves with the pre- computed matrix $\hat{G}^l$ to compute the correction; the result is stored in the dual-port V memory.

3. The charge grid also convolves with the pre-computed matrix A_l^{l+1} to anterpolate charges to the next coarser grid; the result is stored in the dual- port Q memory.

4. At the coarsest grid, the potential is computed directly. Instead of matrix $\hat{G}^l$, the charge grid convolves with the pre-computed matrix G^L. The potential grid is stored

in the dual-port V memory.

5. The potential grid convolves with the pre-computed matrix I_{l+1}^{l} to interpolate the potential to the next finer grid. The result is accumulated with the correction computed in step 3.

6. At the finest level, particle coordinates and charge are fetched again to compute the weights for potential interpolation. Potentials from neighboring grid points are fetched simultaneously from the Interleaved V memory, dot multiplied with the weights, and summed with the adder tree. To compute the force in all dimensions, the potential interpolation is performed independently with partial differentiation in three dimensions.

6.3.2 Implementation Consideration

Because the multigrid coprocessor is more complex than the short range non- bonded force coprocessor, scaling and fitting designs to a specific FPGA chip becomes more challenging. For the short range force pipeline, we can easily replicate the force pipeline to maximize the chip usage. The multigrid coprocessor, however, contains many different components that scale non- linearly: some expand cubically, such as the grid-grid convolver; others expand exponentially, such as the particle-grid converter. Furthermore, some components have conflicts for critical resources, especially on the on-chip SRAMs and the hardware multipliers. Careful tuning is thus necessary to balance the system capability and performance. The following setting is based on our implementation on a Xilinx Virtex II pro 70 (VP70) FPGA.

In this implementation, the finest grid can be up to $32 \times 32 \times 32$ with the limited being the number of on-chip SRAMs. We apply the 3rd order basis functions (Equation 5.11). Adopting the 5th order basis functions slightly exceeds the number of the hardware multipliers, but would not be a problem for VP100. The grid-grid convolver has a $4 \times 4 \times 4$ computation kernel; this is also limited by the number of hardware multipliers. The particle-grid converter contains a $1 : 4^2$ two-level tree structure, rather than the preferred

three-level one. Consequently, the memory interleaving is 4^2-way. This configuration is a compromise resulting from resource balancing. The hardware multipliers needed by a $1:4^3$ particle-grid converter and the vector multiplier that follows exceed the capacity of the VP70 chip.

Our experiments also show that a 4^3-way interleaved memory could barely fit on a VP70, but then almost no resources remain for other functions. Another possible design is a $1:2^3$ particle-grid converter. Of course, the 2^3-way interleaved memory is sufficiently small enough to easily fit on chip. The performance however would decay with the parallelism reducing to 8. In contrast, the $1:4^2$ configuration provides reasonable parallelism, while consuming 90% of the hardware multipliers. In fact, the particle-grid converter and the vector multiplier require 36 35-bit multipliers, sharing them with the grid-grid convolver, which needs 64 35-bit multipliers. One consequence of this modification is that the particle-grid converter can only assign one charge to 16 grid points at a time, and so requires four cycles to assign to all 64 grid points.

6.4 Supporting Large Simulations with Explicitly Managed Cache

The limited size of the on-chip SRAMs is the major barrier to simulating large models with our coprocessors. They are used to store three types of data: grid data (in the Multigrid coprocessor), computation parameters (e.g., the Lennard- Jones force coefficients, charges of each atom type, interpolation coefficients, cell-list information, and other static data), and particle data (e.g., coordinate, acceleration, and atom type). As previously discussed, the coprocessors employ the on-chip SRAMs to obtain large bandwidth and complex access patterns and so to achieve high throughput.

The original version of our short range non-bonded force coprocessor could hold up to 10K particles in the on-chip SRAMs. In order to enlarge the system capacity, these data must be stored off-chip; the danger is potential deterioration in bandwidth and flexibility. There are some additional constraints when accessing the off-chip memories. For example, the WildstarII- Pro PCI board from Annapolis Micro Systems, on which we prototyped

our coprocessors, has a 9 clock cycle latency to access the off-chip SRAMs; the SRAM chips can't be read and written concurrently; and a NOP cycle must be inserted between read and write operations. After studying these data, we decided to move the particle data to the off-chip memory. We did this because (i) the particle data has the lowest bandwidth and latency requirement and the simplest access pattern among the three types of data, and (ii) by decoupling the particle memory the coprocessor chip, the simulation size becomes independent of the on-chip SRAM size. The rest of this section addresses the modifications of the coprocessor design to support the caching scheme.

6.4.1 Off-chip Memory Interface and Constrains

We have designed our coprocessor to be maximally hardware independent, in particular with respect to the platform (e.g., the FPGA). If we want to utilize the off-chip resources, however, some generality is necessarily lost. Here, we define an off-chip memory interface and specification, so that we can port, with limited need for modification, our coprocessors to platforms with different off- chip memories. The interface between the off-chip memory and the coprocessors are assumed to be dual-ported (read and write ports) SRAM interface. The off- chip memory is assumed to be able to work at the same frequency as the coprocessor with constant access latency, or have equivalent timing. The bandwidth of the SRAMs is assumed to be no less than the data required for one particle per cycle, where that data is a three dimensional fixed-point coordinate or force vector. In our current system configuration, this is about 100 bits per cycle.

Recall that the particle coordinates and forces are stored on the host in double precision floating-point format. They need to be converted into Semi-FP on the FPGA before they are stored in the off-chip memory. Because the FPGA is physically in between the host and the off-chip memory, the particle data must pass through the FPGA (as shown in Figure 6·13). No additional interface between the bus and the off-chip memory is needed.

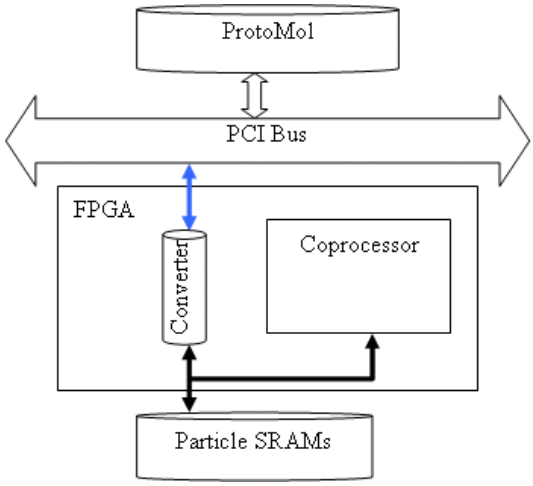


Figure 6·13: Interface among Host, FPGA, and Off-chip SRAM

6.4.2 Coprocessor and Cache Interface

Because the multigrid coprocessor reads and writes one particle per cycle during charge anterpolation and potential interpolation, latency is the only effect of employing the off-chip memory, and only some minor changes are necessary. For the short range force coprocessor, multiple particle data are accessed in every clock cycle by the force pipelines. The fixed connection between the off-chip memory and the coprocessor may not have sufficient bandwidth, and in any case would not be able to scale with the number of force pipelines. To solve this problem, we extended our cell-list method to a cache scheme by utilizing the particles' spatial and temporal locality.

With the cell-list method, particles are grouped based on their locations. At any moment, only particles in a neighborhood (in the same cell or in the neighbor cells) are accessed and processed. Moreover, within the switching cutoff of the short range forces, the number of particles is limited because of the density of the simulation model. Hence, with this spatial locality, it is possible to maintain the throughput of the force pipelines with only a small cache. In addition, increasing the size of the simulation model does not increase the cache capacity requirement. Once a cell and half of its neighbors are loaded into the cache, which contains $O(N)$ particles (where N is the number of particles in a

cell), they are processed by the force pipelines for $O(N^2)$ cycles. The timing ratio between cache swapping (loading/flushing) and computation is thus $O(N)$. This fact implies that if we construct two caches (analogous to blocks in microprocessor caches), we can keep one working with the pipelines at high bandwidth due to the on-chip connection, and have the other one swapping data via a slow interface to the off-chip memory. Since N is not a trivial number in reasonable simulation models, this temporal locality is valid regardless of the size of the simulation. Finally, the cache replacement scheme is straightforward, because it just needs to follow the predefined order to traverse all cells; there are therefore no cache misses.

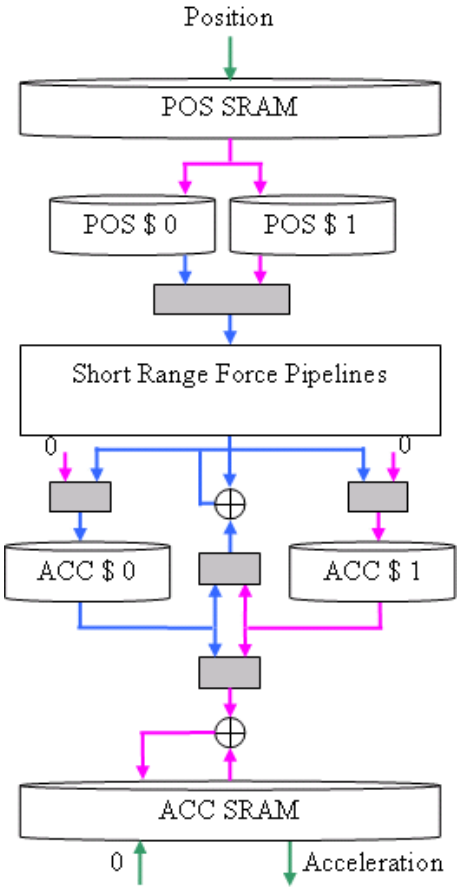


Figure 6·14: Data Path between Cache and Force Pipeline Array. In this example, cache 0 is working with the force pipelines and cache 1 is swapping data with the SRAMs

Figure 6·14 shows the connections among the off-chip memory, caches, and force pipelines. There are two lines of caches numbered 0 and 1. They work either with the force pipelines or swap data with the off-chip memories. Each line has one position cache and one force cache. The position caches are read-only, whereas the force caches are read/write. The gray boxes are multiplexers that control the cache operation. When the force pipelines are computing with one cache line (see, e.g., cache 0 and the blue arrows in Figure 6·14), the other cache line is swapping particles (see, e.g., cache 1 and pink arrows). In particular, the position cache loads the particle coordinates for the next cell; the force cache flushes the result of the previous cell and then initializes itself with 0. The flushed results are added with those read from the off-chip memory (with the adder near the bottom). The adder next to the short range force pipeline accumulates the forces from the pipeline with the partial results in the force cache. The green arrows are the interface between the off-chip memories and the host.

We have implemented the cache scheme on a WildstarII-Pro PCI plug-in board from Annapolis Micro Systems. This board has two FPGAs; there are 6 SRAM chips (Samsung 18Mb (512K×36-bit) DDRII CIO b2 (K7I323682B) around each FPGA, and the total bandwidth per FPGA is 432 bits per cycle. We instantiate two cache lines on the FPGA chip, and each line can store 2048 particles. By using the off-chip SRAMs, our system can simulate up to 256K particles without sacrificing clock frequency. In fact, since fewer on-chip SRAMs are used to store the working set of the particles, more SRAMs can be devoted to the other types of data. The result is actually an improvement in performance.

Chapter 7

Validation and Performance

Simulation quality is the essential criteria for MD systems: any acceleration technology must generate accurate results. Determining simulation error, however, is complex. Generally, it can be divided in two categories: algorithm approximation error and numerical computation error. The former is related to the force model, motion integration, and boundary conditions, while the latter is associated with the numbering system, arithmetic mode, numerical approximation, and error accumulation. We have new approaches in both categories. Investigating the impact on the simulation quality is therefore necessary and critical to validate our research. Performance, however, is our major contribution. Although we have built our system on 4-year old FPGA chips, we have still achieved promising speedups.

In this chapter, we first describe two platforms on which we conducted the experiments. Next, we present the methods and experiments that validate our acceleration. In the third part of this chapter, we present the performance data of our system and compare them with other systems. In this part, we also estimate the performance that would be achieved on similar platforms, but with newer and/or larger FPGA chips. In the last part of this chapter, we present detailed experiments and analysis about the multigrid coprocessor, we respect to both accuracy and performance.

7.1 Experiment Platforms

We have built two types of platforms for validation and performance measurement. Both platforms use ProtoMol 2.03 as the base code and replace the non-bonded force computation with our coprocessor. The first type is a pure software simulator that simulates

our FPGA coprocessors; the other is the FPGA coprocessors themselves. In the software platform, coprocessors are implemented with the C++ Semi-FP library that emulates the Semi-FP operations with bit-level accuracy. With this library, we can easily prototype coprocessors by only constructing major datapath components such as the computation pipelines, but avoid most timing issues. This helps us to explore the design space and to make design decisions about the data width and scale factor of every register. Because it tracks the Semi-FP operation bit-by-bit, this software version system later becomes a test-bench to debug and verify the FPGA version of the system.

We have implemented a complete working system, the platform of which we now describe. The primary components are a PC with a 2.8 GHz Xeon CPU and a WildstarII-Pro PCI board from Annapolis Micro Systems [Ann06]. The board has two Xilinx Virtex-II-Pro XC2VP70 -5 FPGAs. ProtoMol 2.03 is also used (downloaded from the ProtoMol website). The operating system is Windows-XP; all codes are compiled using Microsoft Visual C++ .NET with performance optimization set to maximum. FPGA configurations are coded in VHDL and synthesized with Synplicity integrated into the Xilinx tool flow. Data transfer between host and coprocessors is done with the software support library from Annapolis Microsystems. These transfer routines are efficient with nearly the full PCI bandwidth being used and little system overhead. Both FPGA coprocessors run at a minimum of 75MHz. We use the six off-chip SRAM chips (Samsung 18Mb (512K×36-bit) DDRII CIO b2 (K7I323682B)) to instantiate the cache scheme described in previous chapter.

This working system supports up to 256K particles stored in the off-chip SRAMs (three for coordinates and three for acceleration) and up to 32 particle types. The particle type memory is implemented on the FPGAs. Four cache lines are implemented on-chip and each cache line is capable of storing 2K particles. Other system specifications are: 3rd order polynomial interpolation is applied to compute short range forces; the 3rd order basis function (5th order exceeds the multiplier count on VP70) and the 5th order smoothing function are applied in the multigrid coprocessor; and 35-bit Semi-FP or integer are used for all numerical computation. Please note that the use of 35-bit precision is motivated

as described previously; there is no constraint inherent in any aspect of this work that constrains precision other than user specification.

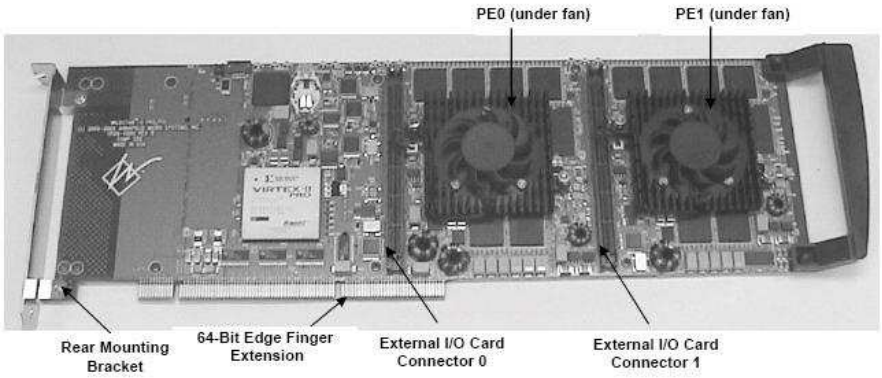


Figure 7·1: WildstarII-Pro PCI board from Annapolis Micro Systems [Ann03]

7.2 Simulation Quality Experiments

Clearly, accuracy of the calculation is a concern when converting a floating- point application to fixed-point arithmetic. Because of the inherently chaotic nature of the calculation, "Obtaining a high degree of accuracy in the trajectories is neither a realistic nor a practical goal. [Rap04]" Rather, quality assurance in MD is determined by observing fidelity of emergent physical properties. Perhaps the most common of these is energy fluctuation (see, e.g., [BS98, SZB96]). Our goal is therefore to achieve energy fluctuation similar to the original calculations.

To measure energy fluctuation, one physical model simulated was bovine pancreatic trypsin inhibitor in water, a model of approximately 1100 particles and 26 atom types, run for 0.01 ns (i.e. 10,000 time steps with 1 fs per step). We say approximately, because the number of waters was varied within 10% between runs to obtain an ensemble. The bonded force in the force model includes angle, bond, diherdral, and improper diherdral. Lennard-Jones and Coulomb forces are computed with 3rd order polynomial interpolation and semi floating-point numbers. Periodic boundary conditions were used and switched cut-off at that size. The original ProtoMol was compared with the ProtoMol modified as described in

the previous section. After 10 runs we have found comparable energy fluctuation between the two systems, with both being close to 0.014.

The second physical model simulated was also of bovine pancreatic trypsin inhibitor in water. This model has more than 14,000 particles and 26 atom types. The bonded forces were same as the first model; the non-bonded forces were evaluated with different methods. The Lennard-Jones force and the short range part of the Coulomb force were computed with 3rd order polynomial interpolation and semi floating-point numbers. In addition, cell-lists were applied with cell size of 10Å, which was also the short range cut-off. The long range part of the Coulomb force was computed with the multigrid method with two levels of grids, finest grid spacing of 4Å, 5th order smoothing functions, and 3rd order basis functions. The boundary condition was vacuum. After 0.01 ns (i.e. 10,000 time steps with 1 fs per step) simulation running on both the original ProtoMol and our accelerated version, we measured the total energy fluctuation. They were very close to each other: 3.50×10^{-4} of the orignial ProtoMol versus 3.55×10^{-4} of the accelerated version. The ratio between the total energy fluctuation and the kinetic energy fluctuation as mentioned in Section 4.3 were both 0.018, below the criteria of .05 [Van04].

The third physical model simulated was of bovine pancreatic trypsin inhibitor in water as well, more than 11,000 particles and 26 atom types. The Lennard-Jones force and the bonded forces were computed as previous models; the Coulomb force was computed with PME method implemented in ProtoMol on host. In particular the PME method applied a real part cutoff of 10Å, 3rd order BSpline interpolation, and the α factor (see Chapter 2.3.1) of 0.312. We compared the simulation quality with different semi-floating point implementations for 0.2 ns (1 fs per step) duration. The energy fluctuation for 35-bit semi-floating point without rounding, 35-bit semi-floating point with rounding to zero, 45-bit semi-floating point with rounding to zero, and double precision floating-point were all about 4.0×10^{-4}. A 0.5 ns simulation applying 35-bit semi-floating point without rounding yielded an energy fluctuation of 3.0×10^{-4}.

7.3 Performance Experiments

For performance comparisons, we simulated the Protein Data Bank *Molecule of the Month* for January, 2007, Importin Beta bound to the IBB domain of Importin Alpha.[1][2] This complex has roughly 77K particles. The simulation box is 93Å×93Å×93Å. We ran for 1000 time-steps. Table 7.2 profiles the relative contributions of various components of both the baseline and the accelerated versions of ProtoMol. Two points are noteworthy for the accelerated version: (i) that the short range force dominates, concurring, e.g., with [SP06], and (ii) that the overhead is a small fraction (roughly 6%) of the execution time. The total speed-up is 9.8×. For further reference, we also downloaded and ran a NAMD binary (v2.6 b1) [3]; these results are also shown in Table 7.2. NAMD is somewhat faster than ProtoMol; the resulting speed-up is 8.8×. These numbers are clearly preliminary as there is substantial room for performance improvement in both baseline and FPGA-accelerated configurations; this is now described.

	Short Range Force	Long Range Force	Bonded Forces	Motion Integration	Comm. & overhead	init. & misc.	TOTAL
FPGA Accelerated Protomol	348.3	61.0	21.5	20.8	25.6	9.2	425
PC-only ProtoMol	3867.8	234.1	21.6	21.5	—	12.9	4157
PC-only NAMD		177.3					3726

Table 7.1: Profile of the 77K particle model simulation

Baseline Code Optimization. ProtoMol has been optimized for experimentation of the kind described here. In contrast, others codes (such as NAMD and GROMACS) have been heavily optimized for performance. Moreover, the data structures in ProtoMol are designed for serial execution but not compatible with our coprocessor interfaces. The overhead to convert data structure in current system can be optimized by redesigning these data structures with consideration of FPGA coprocessors.

Long-range computation. The serial multigrid long range force computation shown

[1] http://www.rcsb.org/pdb/explore/explore.do?structureId=1QGK
[2] http://www.rcsb.org/pdb/static.do?p=education_discussion/molecule_of_the_month/pdb85_1.html
[3] http://www.ks.uiuc.edu/Development/Download/download.cgi?PackageName=NAMD

in 7.1 seems slow (see, e.g. [IHM05]). This is currently being investigated, but in any case, the NAMD SPME code is a bit faster. Commonly, the long-range force is only computed periodically.

Periodic force integration. Commonly, the long-range force is only computed periodically. In the NAMD benchmarks it is computed every four time- steps.

FPGA Accelerated Code Dynamic Reconfiguration. While the performance of the FPGA-based multigrid computation appears to be good, the use of computational resources is disproportionate to its execution time (again, as observed previously by [SP06]). Fortunately, the wall-clock time of each time-step is long in comparison to the time it takes to reconfigure the FPGA. Especially when used with periodic force integration, dynamic reconfiguration is an attractive alternative. For our current hardware, this allows both VP70s to be used primarily for the short-range computation, nearly doubling performance.

Larger chip, higher speed-grade. A larger chip of the same family (the Xilinx V2 VP100) allows the short- range force unit to be implemented with four pipelines rather than just the two that fit in the VP70. This (similarly) results in a near doubling of performance. A higher speed-grade results in a roughly 15

Newer chip. Using the Xilinx Virtex-5 improves operating frequency by another factor of 1.7.

Reduced precision. If reduced precision is acceptable, this also allows the operating frequency to be increased (see Table 7.3), and for the Virtex-5, the number of pipelines to be doubled.

Optimizations. Virtually no optimization has been done on the FPGA configurations; professional design using FPGA-specific tools (such as guided placement) could result in substantial performance improvement. For example, while our arithmetic units are area efficient, they are far slower than the corresponding elements in Xilinx library. Another obvious optimization that has not yet been undertaken is tuning the MD cell size.

Table 7.2 gives the wall clock execution time (per timestep) for an assortment of configurations. The serial codes (except for the bottom line, which was obtained from the

System Tested	Performance
Serial configuration as described	
ProtoMol w/ multigrid every cycle	4.2
NAMD 2.6 w/ PME every cycle	3.7
NAMD 2.6 w/ PME every 4th cycle	3.2
Accelerated configuration as described (2 VP70s)	
ProtoMol w/ multigrid every cycle	.43
ProtoMol w/ multigrid every 4th cycle (dynamic reconfiguration)	.29
ProtoMol w/ multigrid every 4th cycle (dynamic reconfiguration), reduced precision	.21
Accelerated configuration, simulation only	
ProtoMol w/ single V2 VP100 w/ multigrid every 4th cycle (dynamic reconfiguration)	.36
ProtoMol w/ single V2 VP100 w/ multigrid every 4th cycle (dynamic reconfiguration), reduced precision	.33
ProtoMol w/ single V5 LX330T w/ multigrid every 4th cycle (dynamic reconfiguration)	.32
ProtoMol w/ single V5 LX330T w/ multigrid every 4th cycle (dynamic reconfiguration), reduced precision	.19
From NAMD website	
NAMD w/ PME every 4th cycle 90K particle Model	2

Table 7.2: Performance of various configurations given per time-step in seconds of wall-clock time for the 77K particle simulation

NAMD website) were all run on the same PC that serves as host to our FPGA coprocessor. The next set of configurations uses the same PC and the FPGA coprocessor as described previously. The third set is simulation only. These configurations assume the same board but with the two VP70s replaced with a single VP100 of the same speed grade. Timing and area estimates are obtained using the same tool flow through post-place-and-route. The area estimate from such measurements is usually exact and the timing within 10%. The PC-only ProtoMol runs used a cell size of 5Åand a Lennard-Jones cut-off of 10Å. The NAMD runs used a pair-list distance of 13.5Åand a Lennard-Jones cut-off of 10Å. The accelerated ProtoMol runs used a cell size of 10Åand a Lennard-Jones cut-off of 10Å. Finally, we note that NAMD performance of 2 second per time-step per node has been reported for slightly larger simulation models (obtained from the NAMD web site).

Configuration		*FPGA*			
Pipelines	Precision	Virtex-II VP70	Virtex-II VP100	Virtex-4 SX55	Virtex-5 LX330T
4	35 bit	16.6	11.2	12.6	9.5
4	24 bit	10.0	10.0	9.2	7.5
8	24 bit	NA	NA	NA	9.0

Table 7.3: Shown is the clock period in nanoseconds for various accelerator configurations for various Xilinx FPGAs. VP70 numbers are validated in hardware, the others are post place-and-route.

Comparing MD performance of FPGA-based systems will be frought with difficulty until double precision floating-point is fully supported. Table 7.3 shows the highest performing configurations in terms of operating frequency and number of pipelines across recent families of Xilinx FPGAs. We have generated a number of new data points which we now interpret. We have shown experimentally the following speed-ups; the first two comparisons show little if any loss in simulation quality (numbers from Table 7.2):

- 11.0× when comparing NAMD run in our lab versus ProtoMol accelerated with two VP70s (3.2 versus .29); this reduces to around 6.5× when comparing with external NAMD reports (less-than-2 versus .29).
- When using a single VP100 rather two VP70s the speed-ups are 8.9× and 5.3× (3.2 versus .36 and less-than-2 versus .36).
- When the precision requirement is relaxed, the speed-up for a single VP100 increases to 9.7× versus NAMD run in our lab (3.2 versus .33), and 5.8× versus external NAMD reports (less-than-2 versus .33).
- For the new Virtex-5 LX330T with reduced precision, the speed-ups are 16.8× and 10× (3.2 versus .19 and less-than-2 versus .19). Especially in this last case, the various overhead components are significant.

Intriguing is what this says about the future potential of HPRC for heavily floating-point applications. From the technology point of view, adding hard floating-point units to future generation FPGAs, to go with the hard block RAMS and multipliers, would make a tremendous difference. Also making a big difference would be increasing the numbers of

those other hard components in proportion to the process density.

If FPGA component architecture does not change, HPRC for MD may still be promising. We have shown that a factor of 5× to 9× speed-up is achievable using a VP100 accelerator versus a highly tuned code. The Virtex-5 holds promise of significantly improving that performance. Since the FPGA configurations were done entirely with a modest amount of student labor, there is potential for substantially increasing that speed-up. For such an important application as MD, this effort is likely to be reasonable.

7.4 Detailed Analysis of Multigrid Coprocessor

We did further analysis on the multigrid coprocessor with respect to both simulation accuracy and performance. Beside the energy fluctuation of the entire simulation model, the force error on particles is another measurement to evaluate the simulation quality. This measurement uses forces computed with the most accurate method as a reference to compare the forces computed by alternative methods. We follow the methods used by [STH02]; four types of error are measured to compare the quality of various smoothing and basis functions.

$$F_{abs} = \max_i \|\vec{\tilde{F}}_i - \vec{F}\| \tag{7.1}$$

$$F_{max} = \frac{max_i({m_i}^{1/2}\|\vec{\tilde{F}}_i - \vec{F}\|)}{N^{-1}\sum_i({m_i}^{1/2}\|\vec{F}_i\|)} \tag{7.2}$$

$$F_{avg} = \frac{N^{-1}\sum_i({m_i}^{1/2}\|\vec{\tilde{F}}_i - \vec{F}\|)}{N^{-1}\sum_i({m_i}^{1/2}\|\vec{F}_i\|)} \tag{7.3}$$

$$U_{pot} = \left|\frac{\tilde{U} - U}{U}\right| \tag{7.4}$$

F_{abs} is the maximum absolute error of the force on one particle; Fmax is the normalized error of the force on one particle; F_{avg} is the normalized average error of the force on every

particle; U_{pot} is the potential energy error.

$\vec{F_i}$ and U are the reference force and energy evaluated with software. This method performs direct all-to-all computation and with double precision floating-point. $\tilde{\vec{F}}_i$ and $\tilde{U}$ are the force and energy evaluated with the multigrid method with either double precision floating-point or fixed-point numbers.

The experiments are performed with the 14K-particle BPTI molecules. The fixed multigrid settings of these tests are as follows: 2 levels of grid, 3rd order basis function, the smoothing function cutoff of $10\mathring{A}$, and the ratio between grid levels of 2. We varied the order of the smoothing function (the "order" in the following tables and figures) between 3 and 5, and the size of the finest grid (H). In most cases, the finest grid size is no smaller than $3 \times 3 \times 3$. However, because our coprocessor currently only handles finest grid sizes of powers of 2, we did tests with $H = 2$ and $H = 4$. The simulations ran for 10 time steps of 1fs. The numbers in the following tables are the average errors.

From the force errors we can infer that the fixed-point version has quality similar to the double precision version under same settings. If simulations strictly require that the finest grid size be 3, our fixed-point version can achieve better accuracy with $H = 2$, but will take longer to compute. The potential energy from the fixed-point version is computed with the double precision floating-point numbers rather than the fixed-point numbers, because energy is usually evaluated far less frequently than force in the real simulations; and it can be evaluated with more accurate but slower methods. The potential energy error shows that the potential energy is less sensitive to arithmetic mode than the force.

F_{abs}					
Double Precision		H=2	H=3	H=4	H=5
	Order=3		6.977E-01	1.133E+00	1.877E+00
	Order=5		2.173E-01	5.141E-01	1.248E+00
Fixed Point (35 bits)					
	Order=3	3.846E-01		1.134E+00	
	Order=5	1.125E-01		5.150E-01	

Table 7.4: Absolute Force Error

Table 7.8 and 7.9 shows the performance profiles of the multigrid computation for

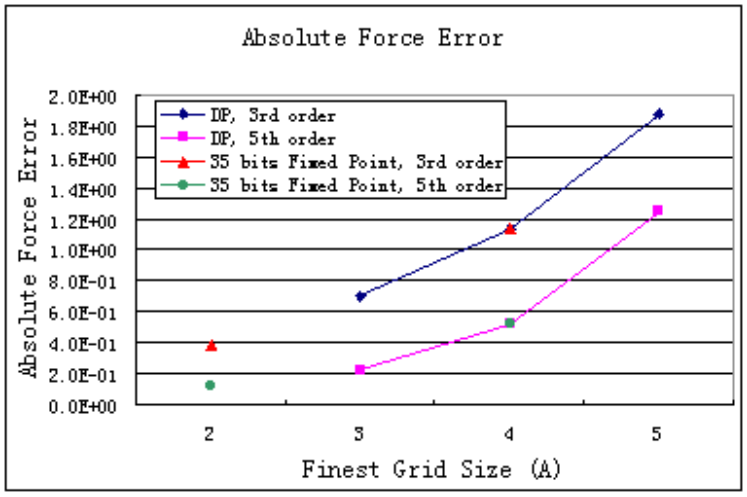


Figure 7·2: Absolute Force Error

F_{avg}					
Double Precision		H=2	H=3	H=4	H=5
	Order=3		4.900E-03	7.489E-03	1.173E-02
	Order=5		1.897E-03	3.792E-03	7.507E-03
Fixed Point (35 bits)					
	Order=3	3.877E-03		7.490E-03	
	Order=5	1.053E-03		3.792E-03	

Table 7.5: Average Force Error

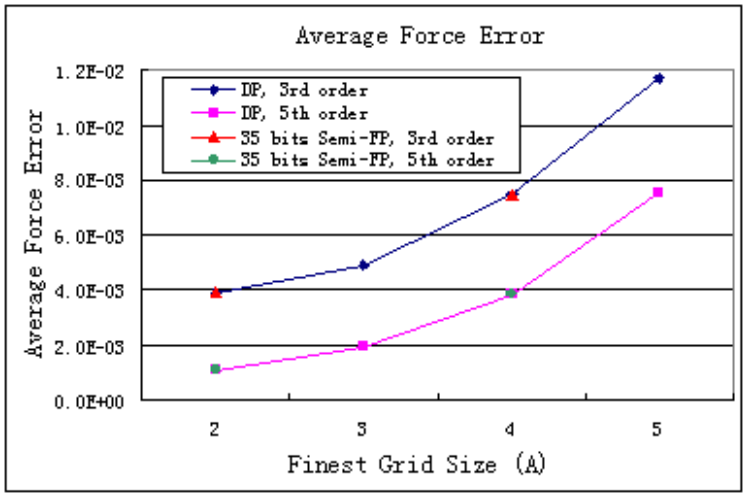


Figure 7·3: Average Force Error

F_{max}					
Double Precision		H=2	H=3	H=4	H=5
	Order=3		2.216E-02	3.802E-02	7.138E-02
	Order=5		8.324E-03	1.850E-02	4.423E-02
Fixed Point (35 bits)					
	Order=3	1.372E-02		3.803E-02	
	Order=5	3.910E-03		1.852E-02	

Table 7.6: Maximum Force Error

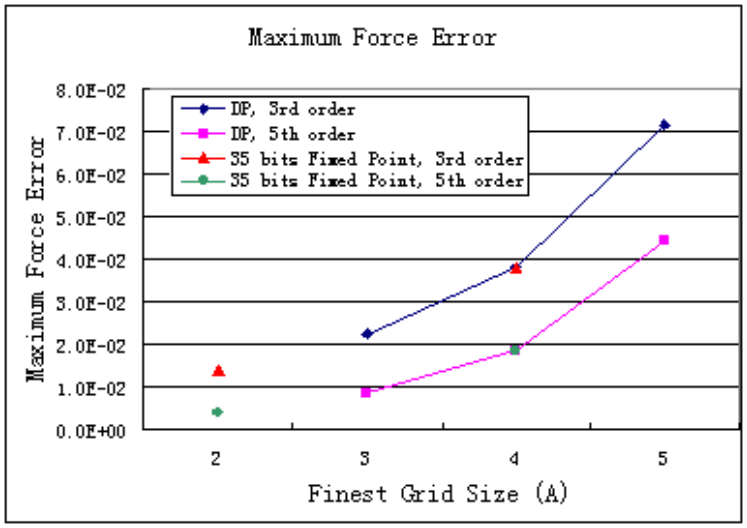


Figure 7·4: Maximum Force Error

U_{pot}					
Double Precision		H=2	H=3	H=4	H=5
	Order=3		1.988E-04	4.457E-04	1.139E-03
	Order=5		5.939E-05	1.661E-04	6.615E-04
Fixed Point (35 bits)					
	Order=3	1.874E-05		4.457E-04	
	Order=5	5.539E-06		1.661E-04	

Table 7.7: Potential Energy Error

Version	Particle-Grid		Convolutions				Overhead	Total
	TP1	TP2	AG	IG	COR	DIR		
PC Only	13.49%	19.59%	2.94%	3.20%	26.72%	34.06%	0%	100%
FPGA Accelerated	6.72%	20.17%	0.85%	1.39%	17.55%	21.81%	31.49%	100%

Table 7.8: Profile of the multigrid Coulomb force coprocessor running the 77K particle model

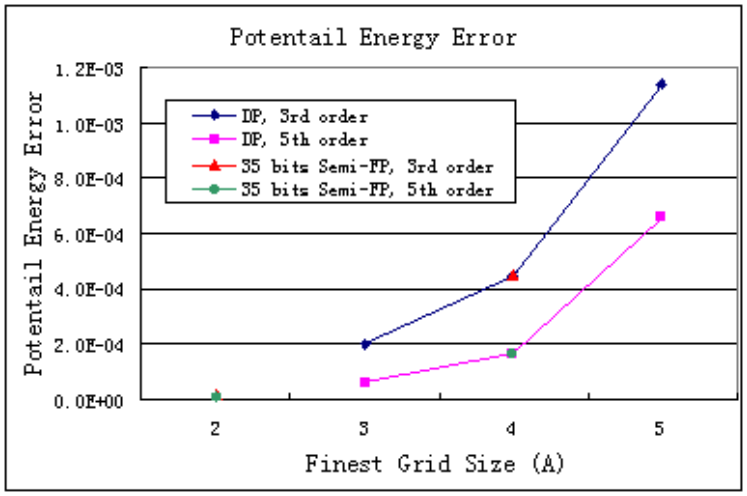


Figure 7·5: Potential Energy Error

	TP1	TP2	AG Anterpolate	IG Interpolate	COR Correction	DIR Direct Sol.	Overhead	**Total**
Convolution characteristic			$14^3 \otimes 4^3$	$17^3 \otimes 4^3$	$28^3 \otimes 10^3$	$17^3 \otimes 17^3$		
ProtoMol Multigrid on **PC**	31.6s	45.9s	6.9s	7.5s	62.6s	79.8s	0s	**234s**
FPGA Multigrid on **VP70**	4.1s	12.3s	.52s	.85s	10.7s	13.3s	19.2s	**61s**
FPGA Fraction of peak	---	---	56%	61%	43%	38%	---	**31%**
Speed-up	7.7x	3.7x	13.1x	8.7x	5.9x	6.0x	---	**3.8x**

Table 7.9: Detail characteristic of multigrid computation. Time in seconds per 1000 timesteps

the 77K particle model on the FPGA-accelerated and the PC-only versions. Most of the execution time is spent on convolutions. Therefore, the performance of the convolver has significant impact on the FPGA acceleration. As described earlier, we spent most of our chip resource on the systolic array convolver to build 64 PEs and each PE performs one pair of multiplication and addition (MAC) per cycle (i.e. 64-MACs per cycle in total). As the coprocessor runs as 75MHz, the peak performance of this convolver is about 9.6GFLOPS. The real performance, however, is less than the peak; the efficiency is reduced for many reasons.

First, the convolver pipeline must be initialized and flushed before and after every convolution, and the PEs are idle during these operations. For example, to anterpolate charges from the finest grid (28^3) to the coarsest grid (17^3), we need 21952 64-MACs to apply a third order basis function; indeed, we spent 39304 cycles to perform this operation with the 64-PE convolver; the efficiency is about 56%. Second, when the convolution kernel is larger than the convolver, the convolution must be partitioned, and pipeline initialization and flushing cause more efficiency loss. Third, because the 64 (i.e. 4^3) PEs are fixed connected in the convolver, the convolution kernel must be expanded if its edge length is not a multiple of 4. These padded zeros reduce efficiency as well, e.g., in COR and DIR. Fourth, if we handle the direct force evaluation on the coarsest grid with all-to-all computation, we only need to compute half of the grid point pairs because of the third Newton's law ($F_{ij} = -F_{ji}$). When cast as a convolution, however, it is hard to apply F_{ij} to grid point i and j at the same time in the convolver, causing a $2\times$ loss. Fifth, there is further coprocessor overhead (e.g. communication), that comprises almost 1/3 of the execution time of the FPGA accelerated version.

Overall, the efficiency of the multigrid coprocessor is 31%, or about 3GFLOPS at 75MHz. Because the I/O requirement is low and independent of further parallelism achieved in the layout, the system is scalable to larger FPGA chips without modification.

This profile indicates that to improve the performance, we should improve the performance of the convolver first. Having more PEs in the convolver definitely increases

parallelism and provides higher peak GFLOPS. A larger convolver is also able to handle larger convolution kernels and reduces the overhead of splitting convolutions. At least two approaches can achieve these purposes: implementing the coprocessor on a larger and faster FPGA chip, and applying reduced precision, if acceptable. To improve coprocessor efficiency, it is always important to match the problem to the hardware, in this case the size of the convolver. Moreover, because the overhead inherent in the FPGA coprocessor becomes significant especially after acceleration, Amdahl's law again bounds the speedup. Having a high bandwidth and low latency communication interface between host and FPGA can optimize the overall performance as well.

Chapter 8

Summary and Future Directions

In this final chapter, we conclude this thesis with a summary of the major work performed in this research, and discuss two major directions in which we plan to extend our MD/HPRC solution.

8.1 Summary

In this research we developed a set of technology to accelerate MD with HPRC. Our research includes FPGA algorithm design, numerical analysis, coprocessor micro-architecture design, FPGA implementation, system integration, and system verification. We instantiated a small scale system that was able to perform complete MD simulation and achieve substantial speedup over production software. We also demonstrated that HPRC is a viable technology for accelerating applications in the floating-point domain, even without direct hardware floating-point support. We conclude with some lessons:

Explicit FPGA design is required high quality FPGA design. Compared with the other MD/HPRC attempts surveyed in Section 3.6, we mapped the problem to hardware by reconstructing algorithms, and configured the FPGA with explicit HDL design. In this way, we could apply sophisticated modules such as the interleaved memory and the cell-list framework into the FPGA coprocessors. This design flow requires more development effort than direct translation of software from a high level language; it also provides more design flexibility and thus the potential for much more efficient FPGA design. We believe the importance of MD has made the extra effort well worthwhile.

Algorithm reconstruction is essential for high chip efficiency. Our performance basically comes from two sources: massive parallelism/bandwidth embedded in the FPGA

architecture and the application specific designs that enabled much of this potential to be achieved. The former is the basic feature of FPGA that shows potential speedup over microprocessor even when running at a low frequency; the latter is the essential approach to achieve potential speedup. For HPRC, 'application specific' means both application specific architecture and application specific algorithm, e.g., arithmetic mode. We therefore explored the design space of both algorithm and architecture to find sweet spots that yield optimal performance.

Tuning arithmetic mode and precision require careful verification. One of design choices is using Semi-FP instead of floating-point. Since simulation quality is the fundamental criteria of MD, we were very cautious about numerical analysis. We compared various numerical computation methods and conducted experiments to prove that our alternative arithmetic method and algorithms retained numerical accuracy (with little detriment to physical significance). After all, there is as yet no consensus on the best method for evaluating MD simulation quality. Any statistic error measurement, e.g. force error or energy fluctuation, is a macro characteristic of the simulation model, while users care about micro-structure from MD simulations. Because of this gap, statistic error measurement is not sufficient to tell if a simulation is accurate enough. Real experiments are, of course, the final method to ensure simulation results but are beyond the scope of this research.

Maintainability of FPGA designs is more important than absolute performance. We instantiated our designs on a real 2004-era platform and integrated our coprocessors into ProtoMol for performance analysis. Without low level FPGA optimization, we had about 10x speedup over NAMD. Based on simulation results, the speedup can reach 16x without major modification if latest FPGA are used. To gain more performance from large and complex designs such as our coprocessors, we can either do low level optimization or port them to a newer (better) chip. Based on current FPGA manufacturing technology, we have found that FPGAs still have plenty of room to increase operating frequency and chip capacity, and that various HPRC systems emerge. Considering the difficulty of FPGA

development, maintainability preferable than absolute performance.

8.2 Future Directions

To improve performance further, there are at least two directions in which we shall extend: node level optimization and system level parallelization.

8.2.1 Node Level Optimization

At this time, we can perform complete MD simulations by accelerating the dense computation kernel (short range and long range non-bonded forces) with FPGA coprocessors. Some modules in these coprocessors, however, still have room for further optimization. For example, we used a general convolver in the multigrid coprocessor for the grid-grid computation. In fact, the convolution kernels are all symmetric, which means a big number of PEs in the systolic array are doing same dot-products during every cycle. By reusing the results from PEs, we can theoretically save 7/8 or more PEs, i.e. another 8x speedup in the convolver.

Porting our coprocessor to new FPGA chips is the second node level optimization. The current system is implemented on 2004-era FPGAs, which were manufactured with 130nm technology and can operate as fast as 200MHz (with 100MHz being likely), while the newest FPGAs are manufactured with 65nm and can operate at more than 500MHz (with 200MHz being likely). Porting our coprocessors to the new FPGAs can increase both parallelism and operating frequency. However, because some of our current designs are optimized for old FPGA architecture, e.g. the 18Kb on-chip SRAM and the 18-bit hardware multiplier, some modules must be adjusted to adapt new FPGA architecture, e.g., the 25-bit multiplier and 36Kb on-chip SRAM in the Xilinx Virtex 5 series.

In order gain more performance, we need more configurable components. Mapping designs across multiple FPGA chips can serve this purpose. Reconfiguration, in some cases, can provide more configurability as well. Other low-level optimization such as floorplanning could improve performance as well.

8.2.2 System Level Parallelization

The next extension of this research is to use the FPGA coprocessors in parallel systems. Since parallel MD packages have been developed for many years, integrating our FPGA coprocessors to them is a reasonable possibility. As described earlier, MD packages apply various parallelization methods, such as force and spatial decomposition; these require different software/hardware decompositions and thus accelerator implementations. Our current designs must be modified accordingly.

In the case of spatial decomposition, the short range non-bonded force coprocessor can be easily integrated into each node, as it is self-contained with a block of space in the simulation box. For the long range non-bonded force, communication becomes critical. In fact, because of the acceleration, communication overhead is relatively more costly than in an unaccelerated parallel system.

Based on performance profiling, some functions may be added to or removed from the coprocessor. One possibility is to allocate one dedicated node to compute the long range force at lower frequency than other forces: particle-grid conversion can be done on the particles' host nodes before being transferred to the dedicated node. Since the data transfer between the dedicated node and the host nodes can be slow; within the dedicated node, our multigrid coprocessor can solve Poisson's equation with some modifications. For example, to compute large models (the finest grid is larger than 64^3), the multigrid coprocessor must be modified to swap grid data off-chip as well as to support more levels of grids. Moreover, because of the Amdahl's law, the speedup of current system will finally be limited by the part we do not accelerate, i.e. the bond force computation, motion update, and overhead, acceleration about these parts will become more and more important.

On the hardware platform side, large scale parallel reconfigurable computers have various system architectures that keep changing. It is impossible to provide a general solution for all systems, and thus design methodology plays an important role. Separating modules to be platform dependent or independent, as in the case of our off-chip memory interface, is one feasible way. The platform-dependent modules must be redesigned for specific plat-

forms; the platform independent modules are better to be scalable, so that they can adapt hardware of same architecture. In the long term, EDA tools can be used to specify these generic parameters and resynthesize optimal FPGA designs based on system specification.

References

[AAC+05] Arvind, Krste Asanovic, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, David Patterson, Jan Rabaey, and John Wawrzynek. Ramp: Research accelerator for multiple processors - a community vision for a shared experimental parallel hw/sw platform. Technical Report UCB/CSD-05-1412, EECS Department, University of California, Berkeley, Sep 2005.

[AAS+07] S.R. Alam, P.K. Agarwal, M.C. Smith, J.S. Vetter, and D. Caliga. Using FPGA devices to accelerate biomolecular simulations. *Computer*, 40(3):66–73, 2007.

[AFK+95] T. Amisaki, T. Fujiwara, A. Kusumi, H. Miyagawa, and K. Kitamura. 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, 1995.

[AKE+04] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow. Reconfigurable molecular dynamics simulator. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 197–206, 2004.

[Alt07] Altera, Corp. *FFT MegaCore Function User Guide*, 2007.

[Ann03] Annapolis Micro Systems, Inc., Annapolis, MD. *WILDSTAR-II Hardware Reference Manual*, 2003.

[Ann06] Annapolis Micro Systems, Inc., Annapolis, MD. *WILDSTAR II PRO for PCI*, 2006.

[AT90] M.P. Allen and D.J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, 1990.

[Bha07] A. V. Bhatt. Keynote talk 1. Presentation, the International Conference on Field Programmable Logica and Applications, 2007.

[BHD+02] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. *Journal of Supercomputer*, 21(2):117–130, 2002.

[BHUH06] M.J. Beauchamp, S. Hauck, K.D. Underwood, and K.S. Hemmert. Architectural modifications to improve floating-point unit efficiency in FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 515–520, 2006.

[BPGH81] H. J. C. Berendsen, J. P. M. Postma, W. F. Van Gunsteren, and J. Hermans. Interaction models for water in relation to protein hydration. In B. Pullman, editor, *Intermolecular Forces*. Reidel Publishing Company, Reidel, Dordrecht, The Netherlands, 1981.

[BS98] E. Barth and T. Schlick. Overcoming stability limitations in biomolecular dynamics. I. combining force splitting via extrapolation with Langevin dynamics in LN. *Journal of Chemical Physics*, 109(5):1617–1632, 1998.

[But03] M. Butts. Molecular electronics: All chips will be reconfigurable. Tutorial, the 13th International Conference on Field Programmable Logic and Applications, September 2003.

[CCD+05] D.A. Case, T.E. Cheatham III, T. Darden, H. Gohlke, R. Luo, K.M. Merz, Jr., A. Onufriev, C. Simmerling, B. Wang, and R.J. Woods. The Amber biomolecular simulation programs. *Journal of Computational Chemistry*, 26:1668–1688, 2005.

[Che07] Chen, T., et al. QCDOC: Cell broadband engine architecture and its first implementationa performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[Cle05] ClearSpeed Technology, www.clearspeed.com. *CSX600: Advanced Product Data*, 2005.

[Cra05] Cray, Inc., www.cray.com/products/xd1. *Cray XD1 Supercomputer*, 2005.

[DD04] M.L. DeMarco and V. Daggett. 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, 2004.

[DG04] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[DRC07] DRC Computer Corporation, www.drccomputer.com. *DRC Reconfigurable Processor Unit*, 2007.

[DYP93] T. Darden, D. York, and L. Pedersen. Particle Mesh Ewald: an $N \log(N)$ method for Ewald sums in large systems. *Journal of Chemical Physics*, 98:10089–10092, 1993.

[EMF+93] T. Ebisuzaki, J. Makino, T. Fukushige, M. Taiji, D. Sugimoto, T. Ito, and S. K. Okumura. Grape project: An overview. *Publications of the Astronomical Society of Japan*, 45:269–278, 1993.

[EPB+95] U. Essmann, L. Perera, M.L. Berkowitz, T. Darden, H. Lee, and L.G. Pedersen. A smooth particle mesh Ewald method. *Journal of Chemical Physics*, 103:8577–8593, 1995.

[EPC05] EPCC. FPGAs: Self-wiring supercomputer is cool and compact. *Edinburgh Parallel Computing Centre (EPCC) news*, July 2005.

[FAL+06] P.L. Freddolino, A.S. Arkhipov, S.B. Larson, A. McPherson, and K. Schulten. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure*, 14:437–449, 2006.

[FMI+93] T. Fukushige, J. Makino, T. Ito, S. Okumura, T. Ebisuzaki, and D. Sugimoto. WINE-1: Special-purpose computer for n-body simulations with periodic boundary conditions. *Publ. Astronomical Society of Japan*, 44:361–375, 1993.

[FSL91] C. F. E. Wu F. S. Lai. A hybrid number system processor with geometric and complexarithmetic capabilities. *IEEE Transactions on Computers*, 40(8):952–962, 1991.

[FTM+96] T. Fukushige, M. Taiji, J. Makino, T. Ebisuzaki, and D. Sugimoto. A highly parallelized special purpose computer for many-body simulations with and arbitrary central force: MD-GRAPE. *The Astrophysical Journal*, 468:51–61, 1996.

[GASSS98] B. Garcia-Archilla, J.M. Sanz-Serna, and R.D. Skeel. Long-time-step methods for oscillatory differential equations. *SIAM Journal on Scientific Computing*, 20(3):930–963, 1998.

[GH07a] Y. Gu and M. C. Herbordt. Amenability of multigrid computations to fpga-based acceleration. In *High Performance Embedded Computing Workshop*, 2007.

[GH07b] Y. Gu and M. C. Herbordt. FPGA-based multigrid computations for molecular dynamics simulations. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.

[Gro02] T. Grotker. *System design with SystemC*. Kluwer Academic Publishers, Boston MA, 2002.

[GS02] P. Gibbon and G. Sutmann. Long-range interactions in many-particle simulation. In J. Grotenhorst, D. Marx, and A. Murmatsu, editors, *Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms*. John von Neumann Institue for Computing, NIC Series, Vol. 10, 2002.

[GVH06a] Y. Gu, T. VanCourt, and M. C. Herbordt. Accelerating molecular dynamics simulations with configurable circuits. *IEE Proceedings on Computers and Digital Technology*, 153(3):189–195, 2006.

[GVH06b] Y. Gu, T. VanCourt, and M. C. Herbordt. Improved interpolation and system integration for FPGA-based molecular dynamics simulations. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 21–28, 2006.

[HaAWH+05] M. Haselman, M. Beauchamp and. A Wood, S. Hauck, K. Underwood, and K.S. Hemmert. A comparison of floating point and logarithmic number systems for fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[He94] Y. Hwang and et al. Parallelizing molecular dynamics programs for distributed memory machines: An application of the chaos runtime support library. Technical Report CS-TR-3374 and UMIACS-TR-94-125, Department of Computer Science and UMIACS, Maryland, 1994.

[HFKM00] T. Hamada, T. Fukushige, A. Kawai, and J. Makino. PROGRAPE-1: A programmable, multi-purpose computer for many-body simulations. *Publications of Astronomical Society of Japan*, 52:943–954, 2000.

[Hil85] W.D. Hillis. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985.

[HN05] T. Hamada and N. Nakasato. Massively parallel processors generator for reconfigurable system. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[HP05] J. Hammes and D. Poznanovic. Application development on the src computers, inc. systems. In *Proceedings. 19th IEEE International Conference on Parallel and Distributed Processing Symposium*, pages 78a–78a, 2005.

[HVG+07] M.C. Herbordt, T. VanCourt, Y. Gu, B. Shkhwani, A. Conti, J. Model, and D. DiSabello. Achieving high performance with FPGA-based computing. *IEEE Computer*, 40(3):42–49, 2007.

[IHM05] J.A. Izaguirre, S.S. Hampton, and T. Matthey. Parallel multigrid summation for the n-body problem. *Journal of Parallel and Distributed Computing*, 65:949–962, 2005.

[IMM+02] J. A. Izaguirre, Q. Ma, T. Matthey, J. Willcock, B. Moore T. Slabach, and G. Via Montes. Overcoming instabilities in verlet-i/r-respa with the mollified impulse method. In T. Schlick and H. H. Gan, editors, *Proceedings of 3rd International Workshop on Methods for Macromolecular Modeling, volume 24 of Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Berlin, New York, 2002.

[Imp06] Impulse Accelerated Technologies, Inc. Web page. http://www.impulsec.com/, 2006.

[Inc] Cray Inc. Cray arsc presentation xd1 fpga.

[Int07a] Intel Corporation. From a few cores to many: A tera-scale computing research overview. In *White Paper*, 2007.

[Int07b] Intel Corporation, www.intel.com. *Intel QuickAssist Technology Accelerator Abstraction Layer (AAL)*, 2007.

[JVS03] Hong Jiang J. V. Sumanth, D. R. Swanson. Performance and cost effectiveness of a cluster of workstations and md-grape 2 for md simulation. In *the Second International Symposium on Parallel and Distributed Computing*, pages 244–249, 2003.

[KATS06] F. Khalili-Araghi, E. Tajkhorshid, and K. Schulten. Dynamics of K^+ ion conduction through Kv1.2. *Biophysical Journal: Biophysical Letters*, 91:L72–L74, 2006.

[KDK+01] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, , and S. Rixner. Imagine: media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[Ke99] L. Kale and et al. NAMD2: greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[KM02] M. Karplus and J.A. McCammon. Molecular dynamics simulations of biomolecules. *Nature Structural Biology*, 9(9):646–652, 2002.

[KP06] V. Kindratenko and D. Pointer. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[KSF+07] C. Kutzner, D. Van Der Spoel, M. Fechner, E. Lindahl, U. Schmitt, B. L. De Groot, and H. Grubmuller. Software news and update speeding up parallel gromacs on high-latency networks. *Journal of Computational Chemistry*, 28(12):2075–2084, 2007.

[KUT+97] Y. Komeiji, M. Uebayasi, R. Takata, A. Shimizu, K. Itsukashi, and M. Taiji. Fast and accurate molecular dynamics simulation of a protein using a special-purpose computer. *Journal of Computational Chemistry*, 18(12):1546–1563, 1997.

[Lew94] D. M. Lewis. Interleaved memory function interpolators with application to anaccurate lns arithmetic unit. *IEEE Transactions on Computers*, 43(8):974–982, 1994.

[LYS07] S. L. Lu, P. Yiannacouras, and T. Suh. An fpgabased pentium in a complete desktop system. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 53–59, 2007.

[Mat04] T. Matthey. ProtoMol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Transactions on Mathematical Software*, 30(3):237–265, 2004.

[Mat06] Mathworks. Simulink hdl coder 1. In *Datasheet*, 2006.

[Men06] O. Mencer. ASC: a stream compiler for computing with FPGAs. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 15(9):1603–1617, 2006.

[MPHL03] O. Mencer, D.J. Pearce, L.W. Howes, and W. Luk. Design space exploration with a stream compiler. In *Proceedings of the Conference on Field-Programmable Technology (FPT)*, 2003.

[Nal06] Nallatech Ltd., www.nallatech.com. *Product Line Card*, 2006.

[NSE+99] T. Narumi, R. Susukita, T. Ebisuzaki, G. McNiven, and B. Elmegreen. Molecular dynamics machine: Special-purpose computer for molecular dynamics simulations. *Molecular Simulation*, 21(5/6):401–415, 1999.

[NSFE00] T. Narumi, R. Susukita, H. Furusawa, and T. Ebisuzaki. 46 tflops special-purpose computer for molecular dynamicssimulations: Wine-2. In *Proceedings of the 5th International Conference on Signal Processing (WCCC-ICSP 2000)*, pages 575–582, 2000.

[NSK+00] T. Narumi, R. Susukita, T. Koishi, K. Yasuoka, H. Furusawa, A. Kawai, and T. Ebisuzaki. 1.34 tflops molecular dynamics simulation for nacl with a special-purpose computer: (mdm). In *Proceedings of the ACM/IEEE International Conference on Supercomputing*, page 54, 2000.

[oSBES06] NSF Blue Ribbon Panel on Simulation-Based Engineering Science. *Simulation-BAsed Engineering Science*. National Science Foundation, 2006.

[Phi05] Phillips, J.C., et al. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.

[PLM+97] G. La Penna, S. Letardi, V. Minicozzi, S. Morante, G.C. Rossi, and G. Salina. Parallel computing and molecular dynamics of biological membranes. *ArXiv Physics eprints*, Physics/99709024, 1997.

[Poi06] D. Pointer. Personal communication, 2006.

[Pre05] President's Information Technology Advisory Committee. *Computational Science: Ensuring America's Competitiveness*. National Coordination Office for Information Technology Research and Development, http://www.nitrd.gov, 2005.

[PV96] D. Pasetto and M. Vanneschi. Machine independent analytical models for cost evaulation of template-based programs. Technical Report TR-96-08, Dipartimento di Informatica, Universita di Pisa, Corso Italia 40, 56125 Pisa, Italy, 1996.

[PW96] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996.

[PZK02] J. Phillips, G. Zheng, and L. Kale. NAMD: biomolecular simulation on thousands of processors. In *Proceedings of the ACM/IEEE International Conference on Supercomputing*, pages 1–18, 2002.

[RAJ99] P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media isa extensions. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, 1999.

[Rap04] D.C. Rapaport. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, 2004.

[SA75] E. E. Swartzlander and A. G. Alexopoulos. The sign/logarithm number system. *IEEE Transactions on Computers*, 24:1238–1242, 1975.

[Sch07] A. Schiller. *Einsatz von FPGAs fur molekulardynamische Rechnungen, Diplomarbeit*. Fachhochschule Aachen, 2007.

[SD01] C. Sagui and T. Darden. Multigrid methods for classical molecular dynamics simulations of biomolecules. *Journal of Chemical Physics*, 114:6578–6591, 2001.

[SGTP06] R. Scrofano, M. Gokhale, F. Trouw, and V. Prasanna. A hardware/software approach to molecular dynamics on reconfigurable computers. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[SI98] R. D. Skeel and J. A. Izaguirre. The five femtosecond time step barrier. In P. Deuflhard, J. Hermans, B. Leimkuhler, A. Mark, S. Reich, and R. D. Skeel, editors, *Computationa Molecular Dynamics: Challenges, Methods, Ideas, volume 4 of Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Berlin Heidelberg New York, 1998.

[Sil04] Silicon Graphics, Inc., www.sgi.com/pdfs/3721.pdf. *Extraordinary Accleration of Workflows with Reconfigurable Application-Specific Computing from SGI*, 2004.

[Ske99] R.D. Skeel. Integration schemes for molecular dynamics and related applications. In M. Ainsworth, J. Levesley, and M. Marletta, editors, *The Graduate Student's Guide to Numerical Analysis*. Springer-Verlag, Berlin, 1999.

[Sny86] L. Snyder. Type architectures, shared memory, and the corollary of modest potential. *Annual Review of Computer Science*, 1:289–317, 1986.

[SP04] R. Scrofano and V. Prasanna. Computing lennard-jones potentials and forces with reconfigurable hardware. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2004.

[SP06] R. Scrofano and V. Prasanna. Preliminary investigation of advanced electrostatics in molecular dynamics on reconfigurable computers. In *Proceedings of the ACM/IEEE International Conference on Supercomputing*, 2006.

[SRC05] SRC Computer, Inc., www.srccomp.com. *SRC C Programming Environment v2.0 Guide*, 2005.

[SRC06] SRC Computer, Inc. Web page. www.srccomp.com, 2006.

[SSB+99] T. Schlick, R.D. Skeel, A.T. Brunger, L.V. Kale, J.A. Board, J. Hermans, and K. Schulten. Algorithmic challenges in computational molecular biophysics. *Journal of Computational Physics*, 151:9–48, 1999.

[SSOB02] H. Shan, J. P. Singh, L. Oliker, and R. Biswas. A comparison of three programming models for adaptive applications on the origin2000. *Journal of Parallel and Distributed Computing*, 62(2):241–266, 2002.

[STH02] R.D. Skeel, I. Tezcan, and D.J. Hardy. Multiple grid methods for classical molecular dynamics. *Journal of Computational Chemistry*, 23:673–684, 2002.

[Swa87] E.E. Swartzlander. *Systolic Signal Processing Systems*. Marcel Drekker, Inc., 1987.

[SZB96] S.j. Stuart, R. Zhou, and B.J. Berne. Molecular dynamics with multiple time scales: The selection of efficient reference system propagators. *Journal of Chemical Physics*, 105(4):1426–1436, 1996.

[Tec07] Maxeler Technologies. http://www.maxeler.com/, 2007.

[TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, 2002.

[TMK+99] S. Toyoda, H. Mihagawa, K. Kitamura, T. Amisake, E. Hashimoto, H. Ikeda, A. Kusumi, and N. Miyakawa. Development of MD engine: High-speed accelerator with parallel processor design for molecular dynamics simulations. *Journal of Computational Chemistry*, 20(2):185–199, 1999.

[TNO+03] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya. Protein Explorer: A petaflops special-purpose computer system for molecular dynamics simulations. In *Supercomputing*, 2003.

[Tre04] N. Tredennick. Reconfigurable systems emerge. Keynote Talk, the International Conference on Field Programmable Logic and Applications, August 2004.

[Van04] D. Van der Spoel. Gromacs exercises. CSC Course, Espo, Finland, February 2004.

[Ver67] L. Verlet. Computer experiments on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical Review*, 159:98–103, 1967.

[VGH04] T. VanCourt, Y. Gu, and M.C. Herbordt. FPGA acceleration of rigid molecule interactions. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2004.

[VH06] T. VanCourt and M.C. Herbordt. Application-dependent memory interleaving enables high performance in FPGA-based grid computations. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 395–401, 2006.

[VHB03] T. VanCourt, M.C. Herbordt, and R. Barton. Case study of a functional genomics application for an FPGA-based coprocessor. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 365–374, 2003.

[VHB04] T. VanCourt, M.C. Herbordt, and R. Barton. Microarray data analysis using an FPGA-based coprocessor. *Microprocessors and Microsystems*, 28(4):213–222, 2004.

[VLH+05] D. Van der Spoel, E. Lindahl, B. Hess, G. Groenhof, A.E. Mark, and H.J.C. Berendsen. GROMACS: fast, flexible, and free. *Journal of Computational Chemistry*, 26:1701–1718, 2005.

[Xil03] Xilinx, Inc. *Virtex-II Pro. Platform FPGAs: Functional Description*, 2003.

[Xil06] Xilinx, Inc. *Product Specification — Xilinx LogiCore Floating Point Operator v2.0*, 2006.

[Xil07] Xilinx, Inc. *Fast Fourier Transform v4.1 Product Specification*, 2007.

[Xtr07a] XtremeData, Inc., www.xtremedata.com. *XD1000 BLOCK DIAGRAM*, 2007.

[Xtr07b] XtremeData, Inc., www.xtremedata.com. *XD1000 Development System*, 2007.

[Yav06] I. Yavneh. Why multigrid methods are so efficient. *Computing in Science and Engineering*, 8:12–22, 2006.