Published in final edited form as:

Parallel Comput. 2008 May ; 34(4-5): 261–277.

Explicit Design of FPGA-Based Coprocessors for Short-Range Force Computations in Molecular Dynamics Simulations ^{*,†}

Yongfeng Gu[‡], Tom VanCourt[§], and Martin C. Herbordt

Computer Architecture and Automated Design Lab, Department of Electrical and Computer Engineering, Boston University; Boston, MA 02215

Abstract

FPGA-based acceleration of molecular dynamics simulations (MD) has been the subject of several recent studies. The short-range force computation, which dominates the execution time, is the primary focus. Here we combine: a high level of FPGA-specific design including cell lists, systematically determined interpolation and precision, handling of exclusion, and support for MD simulations of up to 256K particles. The target system consists of a standard PC with a 2004-era COTS FPGA board. There are several innovations: new microarchitectures for several major components, including the cell list processor and the off-chip memory controller; and a novel arithmetic mode. Extensive experimentation was required to optimize precision, interpolation order, interpolation mode, table sizes, and simulation quality. We obtain a substantial speed-up over a highly tuned production MD code.

1 Introduction

With microprocessors hitting the power wall, alternative architectures for high performance computing (HPC) are receiving substantial attention. Of these, HPC using reconfigurable computing (HPRC) is receiving its share: The promise of HPRC is high performance at lower operating frequency, and thus lower power. The areas of greatest success have been in signal and communication processing where small kernels dominate the computation; these kernels are also highly parallelizable, and can make do with comparatively low-precision and/or low-complexity arithmetic.

Work in HPRC must often address the following well-known difficulties:

- **Chip area limitations.** While in HPC code size is generally not a high-order concern, 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

^{*}This work was supported in part by the NIH through awards #R01 RR023168-01 and #R21 RR020209-01, and facilitated by donations from Xilinx Corporation, SGI, and XTremeData, Inc.

[†]Preliminary versions of parts of this work were presented at the 2006 International Conference on Field Programmable Logic and Applications and the 2007 Reconfigurable Systems Summer Institute

[‡]Currently with The MathWorks, Inc.

[§]Currently with Altera, Inc.

Web: <http://www.bu.edu/caadlab>.

Publisher's Disclaimer: This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final citable form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

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 may be 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., [6] and references).

The case study described here, acceleration of molecular dynamics simulations (MD) with HPRC, is interesting on at least two fronts. First, its acceleration is inherently important: although substantial progress has been made in developing codes that are efficient and scalable to hundreds of nodes, MD is still compute bound (see, e.g., [9]). 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- [3] and von Neumann-based [24] versions as well.

HPRC acceleration of MD has been studied by a number of groups [1,5,13,16,14,22] with the design space being spanned by several axes:

- Precision: Is double precision used, or 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?
- Base MD code: Is 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 switching function? Or, is a more accurate, and more computationally complex, method used? Is this done on the FPGA or in software?
- HPRC design flow: How is the FPGA configured? With a standard HDL, or a C-to-gates process, or some combination?

The goal of the work described here is to investigate the viability of MD in current generation FPGA technology. While previous studies have made substantial progress, most have made compromises in either performance, precision, or model size simulated. We attempt to advance the art with numerous FPGA-centric optimizations, while retaining the MD simulation quality. In particular, our point in the design space is as follows:

- Precision: 35-bit, as derived from experiments measuring energy fluctuation (as described, e.g., in [3]).

- 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 [18], with further performance comparisons with NAMD [19].
- Target hardware: a generic PC and a commercial PCI plug-in board with two Xilinx VP70s [4]. Performance for this configuration with other FPGAs is estimated with area- and timing-accurate design automation methods.
- Scope: we describe the short-range force processor. The design is compatible with various implementations of the long-range component of the electrostatic force (e.g., [11,17,22]).
- Design Flow: All major components (force pipelines, cell-list processor, 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 $5\times$ to $10\times$ speed-up over the NAMD MD code [19] and slightly more over the ProtoMol MD code [18] with little if any compromise in simulation accuracy. If single precision is acceptable, then using a new Virtex-5 promises significantly greater speed-ups. Particular contributions include: a systematically designed force pipeline including analysis of interpolation methods, arithmetic and precision; handling of exclusion; and FPGA-specific designs for cell-lists and handling of large models.

The rest of this work is organized as follows. In the next section we give an overview of MD computation and the algorithms that we use to implement it. There follows a description of the design and implementation of the major components. After that we describe our validation and performance experiments, and conclude with a discussion of potential future implications.

2 Methods

2.1 MD Review

MD is an iterative application of Newtonian mechanics to ensembles of atoms and molecules (see, e.g., Rapaport [20] for details). MD simulations generally proceed in phases, alternating between force computation and motion integration. For motion integration we use the Verlet method. In general, the forces depend on the physical system being simulated and may include van der Waals attraction and Pauli repulsion (approximated together as the Lennard-Jones, or LJ, force), Coulomb, hydrogen bond, and various covalent bond terms:

$$\mathbf{F}^{total} = \mathbf{F}^{bond} + \mathbf{F}^{angle} + \mathbf{F}^{torsion} + \mathbf{F}^{H\ Bond} + \mathbf{F}^{non-bonded} \quad (1)$$

Because the hydrogen bond and covalent terms (bond, angle, and torsion) affect only neighboring atoms, computing their effect is $O(N)$ in the number of particles N being simulated. The motion integration computation is also $O(N)$. Although some of these $O(N)$ terms are easily computed on an FPGA, their low complexity makes them likely candidates for host processing, which is what we assume here. The LJ force for particle i can be expressed as:

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

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 . The Coulombic force can be expressed as:

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

A common way of computing the non-bonded forces is by applying a cut-off. Then the force on each particle is the result of only particles within the cut-off radius. Since this radius is typically less than a tenth of the size per dimension of the system under study, the savings are tremendous, even given the more complex bookkeeping required to keep track of cell- or neighbor-lists.

The problem with cut-off is that, while it may be sufficiently accurate for the rapidly decreasing LJ force, the error introduced in the slowly declining Coulombic force may be unacceptable. A number of methods have been developed to address this issue with some of the most popular being based on the Ewald method (see, e.g., [8]). The disadvantage for HPRC is that these methods involve a three dimensional FFT, which though viable [17], is difficult to implement efficiently on an FPGA. An alternative method uses multigrid: this has sufficient accuracy [15] and, as we have shown elsewhere [11], maps well to the target hardware. The key point for this work is that the Coulomb force is generally split into two components: a slowly converging part that is computed using one of the long-range methods, and a rapidly converging part that, together with the LJ force, comprises the short-range force to be computed.

2.2 Short-Range Force Computation Basics

Most of the complexity in MD is in the short-range, non-bonded, force computation. As just described, this has two parts, the LJ force and the rapidly converging component of the Coulomb force. The LJ force is often computed with the so-called 6-12 approximation given in Equation 2 and shown in Figure 1. Since these calculations are in the inner loop, considerable care is taken in their implementation: even in serial codes, the LJ equation is not evaluated directly, but rather through a table look-up with interpolation.

Previous implementations of FPGA/MD have used table look-up for the entire LJ force as a function of particle separation [5,12]. The index used is $|r_{ji}|^2$ rather than $|r_{ji}|$ so as to avoid the costly square-root operation. This method is efficient for uniform gases where only a single table is required [5], because it is not necessary to distinguish atom types, and the lookup table is a function only of displacement. As the Lennard-Jones force depends on atom types, however, 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 RAMs for FPGA/MD to be viable. In previous work we described a latency hiding technique whereby tables are swapped as needed [12].

Here we propose a different method for LJ force computation for FPGA/MD, which is used also in some MD codes. Instead of implementing the force pipeline with a single table lookup, we use two, one each for r^{-14} and r^{-8} . Equation 2 can be rewritten as:

$$\frac{\mathbf{F}_{ji}^{LJ}(|r_{ji}|^2(a,b))}{r_{ji}} = A_{ab}|r_{ji}|^{-14} + B_{ab}|r_{ji}|^{-8} = A_{ab}R_{14}(|r_{ji}|^2) + B_{ab}R_8(|r_{ji}|^2) \quad (4)$$

where $A_{ab} = 12\epsilon_{ab}\sigma_{ab}^{12}$ and $B_{ab} = 6\epsilon_{ab}\sigma_{ab}^6$, and where $R_{14}(x) = x^{-7}$ and $R_8(x) = x^{-4}$ are two lookup tables both indexed with $|r_{ji}|^2$.

The force is computed with two sets of lookup tables. A_{ab} and B_{ab} are coefficient lookup tables indexed with atom types, but are 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 required for Equation 2 becomes two sets of one-dimensional lookup tables. Assuming we interpolate the Lennard-Jones force with K intervals, the direct interpolation method requires $KT^2/2$ coefficients, while only $2K + T^2$ are required by the latter method. As a result, 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, we begin by rewriting the equation as

$$\frac{\mathbf{F}_{ji}^{CL}(|r_{ji}|^2(a,b))}{r_{ji}} = Q_a Q_b |r_{ji}|^{-3} = Q Q_{ab} R_3(|r_{ji}|^{-2}), \quad (5)$$

where Q_{ab} is a precomputed parameter (analogous to A_{ab} and B_{ab}) again indexed with ab , and R_3 is a lookup table (analogous to R_{14} and R_8) again indexed with $|r_{ji}|^{-2}$. Because applying a cut-off here 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:

$$\frac{1}{r} = \left(\frac{1}{r} - g_a(r)\right) + g_a(r) \quad (6)$$

where the two components are shown in Figure 2. The short range component can be computed together with Lennard-Jones force using a third look-up table. The entire short range force becomes:

$$\frac{\mathbf{F}_{ji}^{short}}{r_{ji}} = A_{ab} r_{ji}^{-14} + B_{ab} r_{ji}^{-8} + Q Q_{ab} (r_{ji}^{-3} + \frac{g'_a(r)}{r}) \quad (7)$$

where the first term is the Pauli repulsion, the second is the van der Waals attraction, and the third is the short-range component of the Coulomb force.

2.3 Interpolation

2.3.1 Interpolation Method—The common method of computing the short-range forces is with table look-up and interpolation. As can be seen in Figure 3, each curve is divided into several sections along the X-axis such that the length of each section is twice that of the previous. Each section, however, is cut into the same number of 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. When the interpolation is order M , each interval needs $M + 1$ coefficients, and each section needs $N * (M + 1)$ coefficients. Since the section length increases exponentially, extending the curve (in r) only slowly increases the size of coefficient memory.

M and N are two major design coefficients for the short range force coprocessor. Increasing M or N each improves simulation accuracy. Interestingly, on the FPGA these two numbers have a resource cost in different components: the main cost for finer intervals is in the independently addressable on-chip block RAMs (BRAMs), while the main cost for higher order interpolation is in hardware multipliers and registers. Table 1 gives a sample of the tradeoff effects for $R_{14}(x) = x^{-7}$. For small simulations, where the entire model fits on-chip,

$N = 128$ and $M = 3$ appears to be optimal. For large simulations, where most particles are stored off-chip, fewer BRAMs are needed for particle data, and the optimal configuration becomes $N = 512$ 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.

2.3.2 Computing the Coefficients—We now show how to develop polynomial approximations to arbitrary functions so that the approximation can be computed with most economical use of FPGA resources (for background see, e.g., Golub and Ortega [10]). The FPGA implements the approximation F of function f as a sum of terms:

$$F(x) = a_0 + a_1x + a_2x^2 + a_3x^3 \quad (8)$$

where the coefficients a_i are stored separately for a set of intervals that partition the range of interest x . 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 a_i for any interval chosen to provide the best approximation of f on that interval. The piecewise nature of F is assumed here, but discussed further in the next subsection. One obvious way to create such an approximation on an interval near point p is by truncating a Taylor series expansion (as was done above). 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 illustrating the first few monomials x^i .

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 be worse even than that of linear interpolation.

Instead, we propose to approximate f using a set of orthonormal functions on each interval. Following the conventions of linear algebra, orthogonal functions $p(x)$ and $q(x)$ have the property that $p(x) \cdot q(x) = 0$, for a suitable definition of dot product ' $\cdot$ '. Likewise, normal functions $p(x)$ have the property that $p(x) \cdot p(x) = 1$. A set of orthonormal functions is a set in which all members are normal and are orthogonal to all other members, and forms a basis set that spans some space of functions. The conventional dot product of two functions on some interval $[A, B]$ is defined as $\int_A^B p(x)q(x)dx$.

An approximation of f in terms of basis set b_i is a projection of f from some high-order function space down into the low-order space spanned by that set of basis functions. Because all terms in f are involved in the projection into that basis, this technique avoids the problem of skipping high-order terms in f that contribute significantly to low-order terms in the approximation. Because the intended FPGA implementation uses polynomials of the form in Equation 8, the set of basis functions is chosen to span the function space $\{1, x, x^2, x^3\}$. Instead, a set of *orthogonal polynomials* offers many advantages (see, e.g., [10]. Gram-Schmidt orthogonalization is used to generate such a set of orthonormal basis functions.

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

$$f(x_0) = dx_0^3 + cx_0^2 + bx_0 + a$$

$$f'(x_0) = 3dx_0^2 + 2cx_0 + b$$

$$f(x_1) = dx_1^3 + cx_1^2 + bx_1 + a$$

$$f'(x_1) = 3dx_1^2 + 2cx_1 + b$$

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

We compare three higher order interpolation methods—Taylor, Orthogonal Polynomials, and Hermite—by plotting their relative RMS error. In the left graph of Figure 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 often applied 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, i.e., $\int_a^b (F(x) - P_n(x))dx = 0$ if x has a uniform distribution. Since in MD the short range force curve is approximated in small pieces, it is acceptable to assume that x has 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, now with respect to the method of orthogonal polynomials (see Figure 6). We observe that several combinations of order M versus intervals N have similar error, allowing for the hardware-based design choice described in Section 2.3.

2.3.4 Interpolation Pipeline Overview—We now describe the interpolation pipeline (see Figure 7). Given that the interpolation function is third order, it necessarily has the format

$$F(x) = ((C_3(x - a) + C_2)(x - a) + C_1)(x - a) + C_0,$$

where $x \equiv r^2 = \text{input}$, $a = \text{the index of the interval from the beginning of the section (see Figure 3)}$, and $x - a = \text{the offset into the interval}$. The coefficients $C_0, \dots, C_3$ are unique to each interval, and are retrieved by determining the section and interval.

Proper encoding makes trivial the extraction of the section, interval, and offset. For example, let the number of bits of x be 14, and the (necessarily fixed) number of intervals per section be 8. Then for $x = 00001111001100$: 00001 determines the section (position of the leading 1); 111 determines the interval (3 bits for 8 sections); and the remaining bits 001100 are $x - a$, the offset into the interval. As can be seen in Figure 7, there are four table look-ups from coefficient memory, (one for each coefficient), three multiplies, and three adds.

2.4 Semi Floating Point Arithmetic

As previously discussed, floating point computations are very expensive on FPGAs; here we describe an alternative, *semi floating point*, that takes advantage of the characteristics of the MD computation just described. We begin by noting that the FPGA's floating point difficulties are primarily with addition; this is now addressed.

The critical observations concern the computation shown in Figure 7. First, for each interval, the scale factors (exponents) are known. Second, for each interval, *differences* in scale factors (exponents) for the addends are known. Third, *there are only a small number of differences between pairs of scale factors*, and only these need to be supported by the adder.

We use these operations to construct an efficient “semi floating point” adder as follows: First, the possible pre-computed shifts (differences), and only those shifts, are hardwired and stored in the look-up table (the “format” shown in Figure 7). Second, the precomputed shift is selected at run time based on the formats stored along with the coefficients, and extracted along with the coefficients (see Figure 7). Finally, the format is routed to the semi floating point adder where it is used by the shift selectors, as shown in Figure 8. The LJ pipeline uses 11 formats.

We now compare the semi floating point to full floating point implementations. The semi floating point pipeline has 35-bit precision for reasons discussed previously [12]; results for other path widths are analogous. For the floating point pipelines we use LogiCore Floating-point Operator v2.0 from Xilinx [26]. Table 2 gives FPGA resources required to implement various components (in terms of slices, the unit of organization of modern FPGAs). The final column gives the number of slices for the three versions for one entire non-bonded, short-range, force pipeline. Our version uses some integer units as well as the semi floating point, but only at points in the computation where no precision can be lost. The SP and DP pipelines could perhaps also be optimized this way, but the complexity of the conversions makes this less advantageous there than it is for the semi FP pipeline. A comparison with respect to register use yields similar results. This experience with floating point units is similar to that of two other studies [16,21].

2.5 Non-Bonded Force Exclusion

In MD simulations, non-bonded forces are only computed between particle pairs without covalent bonds. Therefore, particle pairs that *are* covalently bonded must be excluded from the non-bonded force calculation. There are several possible methods. One is to check whether two particles are bonded before evaluating the non-bonded forces. This is expensive, however, because it requires on-the-fly bond checking. Another method, employed by some MD packages, combines bond checking with pair-list construction, the latter being an alternative to cell-lists (see Section 3.2). Lists of particle pairs are constructed during motion update which only contain particle pairs within the short range force cut-off and not excluded. This method

is more efficient than the first, but still requires checking for bonds between particle pairs. Moreover, it is problematic for our hardware implementation. A third method is to compute the non-bonded force between all particle pairs within the cut-off distance but to 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 allows for efficient hardware implementation. There is a problem here too, however. The complementary force consists primarily of the r^{-14} term of the Lennard-Jones force and can be very large: bonded particles can be much closer than non-bonded pairs. Adding and subtracting such large scale values can overwhelm real but small force values.

Our solution is based on certain physical observations. If two particles are so close that the r^{-14} term is very large, they must be covalently bonded. The opposite, however, is not necessarily true: two particles outside that range may still be covalently bonded. This calls for a two-part solution. To account for the region where particles are necessarily covalently bonded, we apply a *short* cut-off—particle pairs within the short cut-off are excluded from the non-bonded computation. Particle pairs outside the short cut-off, but still covalently bonded, have the non-bonded force computed, but then reversed on the host during the covalent force computation.

We determine the short cut-offs by solving the inequality $F^{short}(r^2) < range$, where range is the dynamic range corresponding to reasonable force values. Because the dominant term of 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 the various σ , i.e., particle types.

3 Pipeline Design

3.1 System and Pipeline Overview

The generic system design is shown in Figure 9. The host is assumed to be a standard PC and the coprocessor a PCI plug-in board. More tightly coupled systems (e.g. as from XTremedata, or DRC [7,25]) would simplify host-coprocessor communication, but not change the overall design. The base software system is ProtoMol, a high-performance MD framework in C++ from Notre Dame University, which has been design especially for ease of experimentation. Hardware/Software partitioning was therefore a matter of swapping out the appropriate routines and replacing them with our own. Particle position data are down- and up-loaded in DP; fixed point conversion on both ends is done in the FPGA (by the converters shown in Figure 9). All motion integration, bonded force computations, and energy computations are performed with the original ProtoMol code on the host.

The overall short-range force pipeline is shown in Figure 10; this is based on the all-to-all non-bonded force coprocessor presented previously [12]. We now describe its modification to support cell lists and large models.

3.2 Modifying the Force Pipeline to Support Cell Lists

Short range non-bonded forces can be switched to zero when the distance between two particles is beyond a cutoff. This cut-off is legitimized in several ways: rapid convergence, as in the case of the LJ force; partitioning of a force into components, as is done for the Coulombic force when the long-range component is handled independently; or when the resulting error is either tolerable or can be corrected elsewhere. By inspecting particles only in the nearby regions, performance is substantially improved. The method of cell-lists [2] is an efficient way to reduce computation with this idea.

The cell-list method partitions the simulation space into cubic cells. Sometimes, the length of the cell edge is made no smaller than the cutoff radius, r_{cut} . Then, given a particle P and its cell C , all particles within r_{cut} of P must be in cells adjacent to C . An optimization is to decrease the cell size. This leads to a better approximation of the surrounding sphere and so decreases the number of extraneous particles examined, but also requires more complex bookkeeping.

Particles are classified into cells every iteration after their new positions are determined (after motion integration). Each cell has a list structure containing indices of its particles. Although the particular particles in each cell-list vary from iteration to iteration, the cell structure itself is fixed. Thus the traversal order of the cells can be predefined.

In the current design, the cell lists are constructed on the host by the original MD code, although the data structures are revised before download to the coprocessor. In particular, instead of lists of indices, particle coordinates and types are grouped by cell. This information is downloaded to the coprocessor every time-step. One coprocessor-centric optimization is to transfer particle type with its position instead of retaining it statically in type memory. Another is that the particle-per-cell counts are downloaded to distinguish cells. Based on this information, particle data are addressed with two-level indexing logic: the first level locates cells while the second locates particles within the cells. To support multiple parallel force pipelines, some dummy particles may be padded at the end of a cell.

Details of the force pipeline array are shown in Figure 11. Given N force pipelines, it works in following the steps:

1. N particles from one cell A are loaded into the P_i array, and one of them is selected to be stored in the P_i register. Its current acceleration is fetched from acceleration memory for later accumulation.
2. In every following cycle, N particles from a neighboring cell B are loaded into the P_j registers. As results emerge from the ends of the force pipelines, they are accumulated with both the particles from cell B and the particle in the P_i register (through the adder tree). P_i 's running accumulation is buffered in the P_i acceleration array.
3. After all particles in cell B are computed with the particle in the P_i register, the next particle in P_i array is loaded into the P_i register. Step 2 is then repeated.
4. After all particles in the P_i array have been computed, the results in the P_i acceleration array are stored back into the acceleration array. At the same time, the next N particles in cell A are ready to be loaded and the process goes back to step 1.

All combinations of neighboring cell pairs are traversed in a nested loop that wraps these four steps.

The modules implemented in VHDL include: the two-level indexing logic; new position, type, and acceleration memory; pair-controller; and host/coprocessor interface. The coprocessor has been integrated into ProtoMol with software functions to arrange particle data and cell-list information for downloading and to post-process results.

3.3 Microsequencer Detail

A microsequencer can be used to generate the particle addresses as described above. Address generation consists of nested loops for the following index operations:

- Traversal of all i, j, k indices representing 3D positions of reference cells,
- Traversal of all possible pairings of particles within the reference cell,

- Traversal of a chosen set of cells adjacent to the reference cell, and
- Traversal of all possible pairings of a particle in the reference cell and a particle in the neighbor cell.

Rather than implement control in a state machine built from random logic, maintainability and flexibility argued in favor of a more regular control structure. A custom, microcoded sequencer was chosen. Its major structure is outlined in Figure 12. All arrows in that diagram represent multi-bit datapaths of various widths.

The sequencer (everything except the *Index Registers*) executes microcode held in the *Program Store*. Every microword in the *Program Store* includes one of three branch control *Opcode* values, which choose the source of the next *Program Counter* (PC) value:

- **Branch.** Conditional logic is implemented by steering chosen *Index Test* bits (e.g. zero tests for loop termination) into the *LSBs* of the next address. A *Test Select* field of the microword decides which bits are to be used, or provides a constant value for unconditional branching. *MSBs* of the next address are taken verbatim from a field of the microword.
- **Conditional call to subroutine.** A field of the microword chooses *Index Test* bits to form a boolean value. If the boolean is true, two things happen. First, the current PC value is incremented and saved in the *PC stack*. Second, the PC is loaded from the *MSB* field of the microword, padded with zero bits in the *LSBs*. If false, the *PC stack* is unchanged and the PC is incremented. The *PC stack* is a single register, so only one level of subroutine call depth is supported.
- **Return.** The *PC Stack* value is unconditionally loaded into the *Program counter*.

On reset, control transfers unconditionally to a PC value of zero. An input to this address generator enables or disables stepping to the next index value, and controls whether the *Index Registers* are updated. This instruction set is tailored to the specific needs of the current application. In particular, the conditional subroutine call addresses the application's boundary-checking logic in a way that limits the number of idle control cycles. Conditional branch instructions provide all the loop control features needed for this application; a "halt" instruction is implemented as an unconditional branch to the current address with other control fields set to hold the system idle. Other instructions, such as conditional return, unconditional branch or subroutine call, interrupt-like response to outside signals, or selectable reset addresses could also have been used. A deeper stack, or different return address convention (e.g. *LSB* toggled) could also be used in other implementations. PC values have no significance: system behavior depends only on the content of the microword and on *Index Register* values, not on any of the PC bits.

In addition to branch control, other fields of the microword control six *Index Registers*, to reset, decrement, or hold their values. Since one of the registers can be reset from multiple data sources, its control field is wider than that of any other index register. Other fields, not shown, handle selection of neighbor cells and provide control values to the component's outputs.

3.4 Supporting Large Models with Explicitly Managed Cache

We have so far assumed that the simulation model fits entirely on the FPGA. With current generation FPGAs, this works well up to about 10K particles; larger models require using off-chip memory. The disadvantage of off-chip data access is not latency as much as bandwidth: while FPGAs clock at only 100-200MHz, the on-chip BRAMs have collectively on the order of 20Tb/S memory bandwidth. This bandwidth is used in its entirety during pipeline operation

and is a major enabling factor in achieving high performance. In this subsection we describe a method of supporting large models with little or no loss of performance.

For the short-range processor two types of data are used: (i) computation parameters, such as the charge of each atom type, interpolation coefficients, cell-lists, and other static data; and (ii) particle data, including position, acceleration, and atom type. Accesses to the latter are far more amenable to latency hiding and so only they are moved off chip.

Off-chip memory interface—We assume an integrated system with several dual ported SRAM banks. This is commonly available (see Section 4), and serves as an explicitly managed cache. The SRAMs work at the same frequency as the FPGA and have constant access latency. The bandwidth of the SRAMs is thus greater than one particle data record per cycle, which consists of the particle information plus either a three dimensional position *or* acceleration. In our current system configuration, this is about 100 bits.

The particle data are stored in double precision in the host memory and are converted into fixed point number format on the FPGA before being stored in the off-chip memory. This is shown in Figure 13a; there is no need for any additional interface between the host and the SRAMs.

Coprocessor modification—The order-of-magnitude smaller bandwidth of off-chip memory is not sufficient to feed the force pipelines directly. There is, however, substantial temporal and spatial locality in data access due to the organization of particles into cells. This forms the basis of our data management.

With cell-lists, particles are grouped by location. And because each particle interacts only with particles in the same and neighboring cells, at any moment just a small fraction of particles will be accessed. Additionally, within the cut-off of the short range forces, the number of particles is limited because of the density of the simulation model. Hence, the spatial locality makes it possible for just a small cache to be sufficient to maintain the force pipeline throughput. Also, since cell size is generally independent of model size, increasing the size of the simulation model does not require an increase in cache size.

Once a cell and half of its neighbors are loaded in the cache with, say, N particles, they will be processed by the force pipelines for $O(N^2)$ cycles. The ratio between the time for cache swapping (loading/flushing) and computation is thus $O(N)$. Double buffering allows us to simultaneously maintain full pipeline throughput while swapping data. Controlling the cache replacement scheme is straightforward: cells are traversed in a predefined order.

Figure 13b shows the connections between SRAMs, caches, and force pipelines. There are two sets of caches numbered with 0 and 1 working with the force pipelines or swapping data with the off-chip SRAMs, respectively. Each set has one position cache and one acceleration cache. The position caches are read-only and the acceleration caches are both read and written by the force pipelines. The gray boxes are multiplexers that control the operation of the caches. When the force pipelines are computing with one set of caches for a cell (say, 0), the other set of caches (say, 1) are swapping. The position cache is loading the position data for the next cell, and the acceleration cache is sequentially flushing its results and then initializing to 0. The flushed results are accumulated with those already in the acceleration SRAM.

4 Implementation, Validation, and Performance

4.1 Implementation

Accelerator implementation is highly hardware dependent. Table 3 shows the highest performing configurations in terms of operating frequency and number of pipelines across

recent families of Xilinx FPGAs. The timing for the Virtex-II VP70 is wall clock time while the others are post place-and-route. The operating frequency obtained for each chip is related to how difficult the configuration is to route. The performance of the short-range computation (excluding overhead discussed below) is nearly linear in both the clock and the number of pipelines.

We have implemented the cache scheme presented in Section 3.4 on a WildstarII-Pro PCI plug-in board from Annapolis Micro Systems, which contains two Xilinx Virtex-II-Pro XC2VP70-5 FPGAs [4]. Each FPGA has six SRAM banks for a total capacity of 12M bits and bandwidth of 432b/cycle. We instantiate two sets of caches on each VP70, and each set can store 2048 particles. By using the off-chip SRAMs and the caches, our system can simulate models of up to 256K particles per FPGA without sacrificing clock frequency.

The primary target system consists of a workstation class PC with a 2.8 GHz Xeon CPU and the board just described. ProtoMol 2.03 was used (downloaded from the ProtoMol website). The operating system was Windows-XP; all codes were compiled using Microsoft Visual C++ .NET with performance optimization set to maximum. FPGA configurations were coded in VHDL and synthesized with Synplicity integrated into the Xilinx tool flow. Data transfer between host and coprocessor was effected 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. To enable end-to-end comparisons of entire MD codes, we use an FPGA-based long-range force computation (see [11] for details). FPGAs run at 60MHz in both short- and long-range computation modes.

In our experiments, besides the precision, we also vary how often the long-range force is computed, and whether the FPGA(s) is (are) dynamically reconfigured. In the baseline configuration, both FPGAs on the board are permanently configured to process short-, and long-range forces, respectively. In further experiments, especially where the long-range force is only integrated every fourth cycle, one of the FPGAs is dynamically reconfigured between short- and long-range configurations. For our validation experiments, we compute energy every cycle; for performance experiments, we compute energy every thousand cycles for both accelerated and reference codes.

4.2 Experiments – Validation

The design was validated against three serial reference codes: our base code ProtoMol [18], which uses double precision floating point; and two versions our own code (fixed and float) that tracks the hardware implementation. Validation has several parts. First, the hardware tracker matches ProtoMol exactly when the hardware tracker has the same floating point datapath. Second, the fixed-point hardware tracker exactly matches the FPGA implementations. The missing link is the relationship between the fixed-point and floating point versions of the hardware tracker. These can only be compared indirectly, *e.g.*, by measuring the fluctuation in a physical invariant [3,12]. We simulated a model of more than 14,000 particles and 26 atom types (bovine pancreatic trypsin inhibitor). After 10,000 time steps running on both the original ProtoMol and our accelerated version, we measured the total energy fluctuation. They were both roughly $5 * 10^{-4}$ with that of the FPGA version being slightly lower.

4.3 Experiments – Performance

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 This complex has roughly 77K particles. The simulation box is 93Å×93Å×93Å. We ran for 1000 time-steps.

Table 4 profiles the relative contributions of various components of both the unaccelerated ProtoMol and the baseline accelerated version describe in Section 4.1. Precision of the accelerated version is 35 bits; there are four force pipelines. Two points are noteworthy for the accelerated version. The first is that the short range force dominates, concurring, e.g., with [22], so that the FPGA configured to compute the long-range force is mostly idle. The second is 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)². NAMD is somewhat faster than ProtoMol; the resulting time was 3726 for speed-up is 8.8×. These numbers are clearly preliminary as there is substantial room for performance improvement in both serial and FPGA-accelerated configurations; this is now described.

Serial 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.

Long-range computation: The serial multigrid long range force computation shown in Table 4 seems slow (see, e.g. [15]). This is currently being investigated, but in any case, the NAMD SPME code is a bit faster.

Periodic force integration: Commonly, the long-range force is only computed periodically. In the NAMD benchmarks it is computed every four time-steps.³

FPGA Accelerated Code

Dynamic Reconfiguration: While the performance of the FPGA-based long-range (multigrid) computation appears to be good, the use of computational resources is disproportionate to its execution time (again, as observed previously by [22]). 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 a 50% higher operating frequency. A higher speed-grade results in a roughly 15% increase in operating frequency and thus performance.

Newer chip: Using the Xilinx Virtex-5 improves operating frequency by another 18%, and with reduced precision, allows the number of pipelines to be doubled.

Reduced precision: If precision reduced to 24 bits is acceptable, this also allows the operating frequency to be increased (see Table 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

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

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 5 gives the wall clock execution time (per time-step) for an assortment of configurations. The serial codes—except for NAMD result, which was obtained from the NAMD website—were all run on the same PC that serves as host to our FPGA coprocessor.⁴ The next set of configurations uses the same PC, but this time accelerated with the FPGA coprocessor as described. The final set are simulation only. These assume the same board but with the two VP70s replaced, first with a single VP100 of the same family, and then with a Virtex-5 LX330T. 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 5Å and a Lennard-Jones cut-off of 10Å. The NAMD runs used a pair-list distance of 13.5Å and a Lennard-Jones cut-off of 10Å. The accelerated ProtoMol runs used a cell size of 10Å and a Lennard-Jones cut-off of 10Å. The cell sizes differ because the relative cost of bookkeeping overhead differs between serial and accelerated versions. 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 in fall 2007, at the same time as our final HPRC results were generated).

5 Discussion

We have described a study of FPGA acceleration of the short-range force computation in molecular dynamics simulations. We differentiate our work in that it combines the following: a high level of FPGA-specific design including cell lists, systematically determined interpolation and precision, handling of exclusion, and support for MD simulations of up to 256K particles.

Comparing MD performance of FPGA-based systems will be fraught with difficulty until double precision floating point is fully supported. Even so, 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 5):

- 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 to 24 bits, 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 single 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

⁴The NAMD website shows performance for a number of systems: we selected the one with the highest performance per node, for a single node configuration; this performance will certainly have improved by publication time.

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 $5\times$ to $9\times$ 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.

One trade-off of developing an application for a non-standard architecture, especially one that has not yet converged to a single programmer's model, is portability. Unlike, say, an HPC standard such as FORTRAN plus OpenMP plus MPI, some reworking of the HPRC code is likely to be necessary before it can be used with high efficiency on a different HPRC platform. In our experience, porting from the Virtex-II VP70 to the VP100 of the same family was immediate, although a day's worth of tuning improved performance substantially. Porting to the Virtex-5 family took slightly longer. We have not yet attempted to port the code to Altera FPGAs, but estimate that this will take on the order of a few weeks. Naturally, we have the advantage of having deep knowledge of our code; it could take substantially longer depending on experience with FPGAs and with MD.

On the other hand, we have done no optimization at the level of LUTs or slices. Besides the time and expertise that this requires, it also makes retaining efficiency while porting more challenging. Right now, porting is mostly a question of proper use of the target memory hierarchy, plus some basic path checking. We expect that the next generation HPRC tools will provide support to simplify this process.

A further question is usability: can this HPRC/MD application only run with ProtoMol? Or can it also be used with other MD codes? This issue is currently receiving some attention [23]. The basic idea is that HPRC/MD consists mostly of low-level kernel routines whose interfaces could be standardized (in a way similar to other common HPC functions). The goal is to make integration of HPRC/MD into an existing MD code simply a matter of changing target libraries.

Another trade-off is development effort. This project was primarily the work of a single graduate student for one year. Of that time, perhaps one third was spent on code development, debugging and test, with most of the rest spent on optimization experiments, writing verification code, validation, and documentation. While the developer had some VHDL experience, he was not a professional designer.

With respect to extensions to larger, multi-FPGA systems: the FPGA implementation described here is for a single node. In particular, because of the way that the MD workload is partitioned between the host and the FPGA, the same methods that are used to distribute work in an MPP are fully applicable here.

We end with a comment about prospects of this work with respect to presumed advances in process technology and computer architecture, especially to many-cores. FPGAs use the same high-end process technology as microprocessors, and so in the grossest sense have the same upgrade path. More specifically, FPGAs increase resources in each generation just as microprocessors add resources, although it is in terms of memory, LUTs, and multipliers, rather than cores and cache. One advantage that FPGAs have is that their operating frequencies are still increasing, and so they have two axes on which performance can increase rather than one. Another potential advantage is that using these resources on an FPGA is immediate: we simply

add pipelines to the chip, multiplying performance with minimal overhead. This is certainly no harder than mapping to additional cores.

Acknowledgements

We thank the ProtoMol group, first for providing the code under open source license (<http://protomol.sourceforge.net>), and also for answering many questions that arose during this work. We also thank the anonymous reviewers for their dozens of helpful comments and suggestions through which this article has been much improved.

References

1. Alam S, Agarwal P, Smith M, Vetter J, Caliga D. Using FPGA devices to accelerate biomolecular simulations. *Computer* 2007;40(3):66–73.
2. Allen, M.; Tildesley, D. *Computer Simulation of Liquids*. Oxford University Press; 1990.
3. Amisaki T, Fujiwara T, Kusumi A, Miyagawa H, Kitamura K. Error evaluation in the design of a special-purpose processor that calculates nonbonded forces in molecular dynamics simulations. *Journal of Computational Chemistry* 1995;16(9):1120–1130.
4. Annapolis Micro Systems, Inc. WILDSTAR II PRO for PCI. Annapolis, MD: 2006.
5. Azizi N, Kuon I, Egier A, Darabiha A, Chow P. Reconfigurable molecular dynamics simulator. *Proc Field Prog Custom Computing Machines* 2004:197–206.
6. Beauchamp M, Hauck S, Underwood K, Hemmert K. Architectural modifications to improve floating-point unit efficiency in FPGAs. *Proc Field Prog Logic and Applications* 2006:515–520.
7. Bursky D. Programmable logic: Opteron gets an assist. *EE Times*: July 12, 2006. 2006
8. Darden T, York D, Pedersen L. Particle Mesh Ewald: an $N \log(N)$ method for Ewald sums in large systems. *J Chemical Physics* 1993;98:10089–10092.
9. Freddolino P, Arkhipov A, Larson S, McPherson A, Schulten K. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure* 2006;14:437–449. [PubMed: 16531228]
10. Golub, G.; Ortega, J. *Scientific Computing: An Introduction with Parallel Computing*. Academic Press; Boston, MA: 1993.
11. Gu Y, Herbordt MC. FPGA-based multigrid computations for molecular dynamics simulations. *Proc Field Prog Custom Computing Machines* 2007:117–126.
12. Gu Y, VanCourt T, Herbordt MC. Accelerating molecular dynamics simulations with configurable circuits. *IEE Proceedings on Computers and Digital Technology* 2006;153(3):189–195.
13. Gu Y, VanCourt T, Herbordt MC. Improved interpolation and system integration for FPGA-based molecular dynamics simulations. *Proc Field Prog Logic and Applications* 2006:21–28.
14. Hamada T, Nakasato N. Massively parallel processors generator for reconfigurable system. *Proc Field Prog Custom Computing Machines*. 2005
15. Izaguirre J, Hampton S, Matthey T. Parallel multigrid summation for the n-body problem. *Journal of Parallel and Distributed Computing* 2005;65:949–962.
16. Kindratenko V, Pointer D. A case study in porting a production scientific supercomputing application to a reconfigurable computer. *Proc Field Prog Custom Computing Machines*. 2006
17. Lee, S. Master's thesis. University of Toronto; 2005. An FPGA implementation of the Smooth Particle Mesh Ewald Reciprocal Sum Compute Engine (RSCE).
18. Matthey T. ProtoMol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Trans Mathematical Software* 2004;30(3):237–265.
19. Phillips JC, et al. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry* 2005;26:1781–1802. [PubMed: 16222654]
20. Rapaport, D. *The Art of Molecular Dynamics Simulation*. Cambridge University Press; 2004.
21. Scrofano R, Gokhale M, Trouw F, Prasanna V. A hardware/software approach to molecular dynamics on reconfigurable computers. *Proc Field Prog Custom Computing Machines*. 2006
22. Scrofano R, Prasanna V. Preliminary investigation of advanced electrostatics in molecular dynamics on reconfigurable computers. *Supercomputing*. 2006

23. Stahlberg E, Popig D, Ryle D, Steinke T, Babst M, Taher M, Anderson K. Molecular simulations with hardware accelerators: A portable interface definition for FPGA supported acceleration. Proceedings of the Reconfigurable Systems Summer Institute. 2007
24. van der Spoel D, Lindahl E, Hess B, Groenhof G, Mark A, Berendsen H. GROMACS: fast, flexible, and free. Journal of Computational Chemistry 2005;26:1701–1718. [PubMed: 16211538]
25. Woods N. Integrating FPGAs in high-performance computing: A system, architecture, and implementation perspective. Proc Field Prog Gate Arrays. 2007
26. Xilinx, Inc. Product Specification — Xilinx LogiCore Floating Point Operator v2.0. 2006

Figure 1.
Shown is the Lennard-Jones potential.

Figure 2.
Shown is the Coulombic force and its partition into fast and slowly converging components.

Figure 3. Table look-up varies in precision across r^{-k} . Each section has a fixed number of *intervals*.

Figure 4.
Basic monomials.

Figure 5.
Comparisons of various interpolation modes, orders, and intervals.

Figure 6.
Comparisons of various orthogonal interpolations. Varied are the order and the number of intervals per section.

Figure 7.
Position of the leading 1 determines the operand format in the interpolation pipeline.

Figure 8.
Semi FP adder with explicit alignment.

Figure 9.
System block diagram.

Figure 10.
Force pipeline.

Figure 11.
Force pipeline described in [12] extended to support cell lists.

Figure 12.
Sequencer for generating list addresses.

Figure 13.
Support for large models.

Table 1

Shown is the trade-off between interval size (N is the number of intervals per section) and interpolation order M for r^{-14} .

N	M	Average Error	Maximum Error
32	4	2.55E-7	3.67E-6
64	4	7.35E-9	1.08E-7
64	3	3.74E-7	4.19E-6
128	3	2.56E-8	2.55E-7
128	2	2.27E-6	1.73E-5
512	2	3.32E-8	2.66E-7
512	1	1.17E-5	6.04E-5
2048	1	7.31E-7	3.76E-6

Table 2

Shown is the resource usage (in slices for the Xilinx Virtex-II family) of various components.

<i>Format</i>	<i>Adder</i>	<i>Multiplier</i>	<i>Complete Force Pipeline</i>
LogiCore DP FP	692	540	19566
LogiCore SP FP	329	139	6998
Semi FP: 35-bit	70	316	—
Integer: 35-bit	18	307	—
Combined semi FP, integer	—	—	4622

Table 3
Shown are the clock frequencies for various accelerator configurations for various Xilinx FPGAs. The VP70 numbers have been validated in hardware, the others are post place-and-route.

Configuration		FPGA			
Pipelines	Precision	Virtex-II VP70	Virtex-II VP100	Virtex-4 SX55	Virtex-5 LX330T
4	35 bit	60 MHz	89 MHz	79 MHz	105 MHz
4	24 bit	100 MHz	100 MHz	109 MHz	133 MHz
8	24 bit	NA	NA	NA	111 MHz

Table 4

Profile of a 77K particle simulation run for 1000 time-steps (units in seconds). Long range forces are computed every iteration. For the FPGA accelerated version, 35-bit precision was used and the long-range and short-range forces overlap.

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

Table 5

Performance of various configurations given per time-step in seconds of wall-clock time for a 77K particle simulation.

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
cycle 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