High Performance Molecular Dynamics Simulations with FPGA Coprocessors [*†]

Yongfeng Gu **Martin C. Herbordt**

Department of Electrical and Computer Engineering
Boston University; Boston, MA 02215
EMail: {maplegu|herbordt}@bu.edu

Abstract: FPGA-based acceleration of molecular dynamics simulations (MD) has been the subject of several recent studies. Here we report on an implementation that we believe to be the first to combine a high-level of FPGA-specific design, systematically determined precision, hardware support for complex force models, and support for simulations of over 250K 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; a novel arithmetic mode; and restructurings of algorithms, e.g., multigrid, to map efficiently to FPGA resources. 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, with, e.g., a new national center and multiple special issues of prominent publications focusing on this technology. 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. Here, small kernels dominate the computation; these kernels are also highly parallelizable, and can make do with comparatively low-precision and/or low-complexity arithmetic.

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-known 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 subtantial 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., [5] 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 efficient and scalable codes (e.g., NAMD [17] and GROMACS [24]), MD is still compute bound (see, e.g., [8]). 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 has been used, not only by most FPGA implementations of MD, but by ASIC- [2] and von Neumann-based [24] versions as well.

HPRC acceleration of MD has been studied by a number of groups [1, 4, 11, 13, 14, 20] 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

[*]This work was supported in part by the NIH through award #RR020209-01 and facilitated by donations from Xilinx Corporation. Web: http://www.bu.edu/caadlab.

[†]One of the goals of this work is to describe the complete implementation of a complex application. This work has been carried out over more than two years and some of the methods have been previously described in detail, e.g. at FPL [10, 11] and FCCM [9].

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?
- 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, model size simulated, or force model complexity. 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 [2]).
- 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 [16], with further performance comparisons with NAMD [17].
- Target hardware: a generic PC and a commercial PCI plug-in board with two Xilinx VP70s [3]. Performance for this configuration with other FPGAs is estimated with area and timing accurate design automation methods.
- Scope: the long-range component of the electrostatic force is computated using multigrid [6, 21] and is implemented entirely on the FPGA.
- 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× to 8× speed-up over NAMD with little if any compromise in simulation accuracy. 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., [18] 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 LJ, Coulomb, hydrogen bond, and various covalent bond terms:

$$\mathbf{F}^{total} = F^{bond} + F^{angle} + F^{torsion} + F^{HBond} + F^{non-bonded}$$

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 do. The LJ force for particle i can be expressed as:

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

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:

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

A standard way of computing the long-range 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., [7]). The disadvantage for HPRC is that these methods involve a three dimensional FFT, which though viable [15], is difficult to implement efficiently on a FPGA. An alternative method uses multigrid: this has sufficient accuracy [12] and, as we will show, maps well to the target hardware.

2.2 Short-Range Force Computation

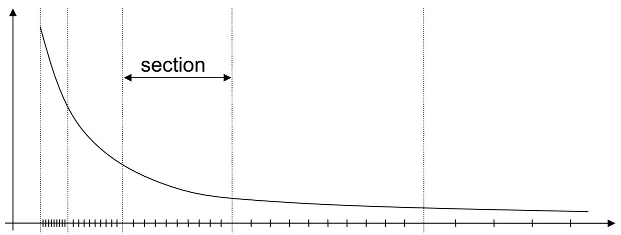

Figure 1: Table look-up varies in precision across r^{-k}. Each section has a fixed number of *intervals*.

A standard way of computing the short-range force is with table look-up with interpolation. As can be seen in Figure 1, 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 increases the size of coefficient memory very slowly.

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

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 block RAMs, while the main cost for higher order interpolation is in hardware multipliers and registers. Table 1 gives a sample of the tradeoff effects. For our system configuration (described below), $N = 128$ and $M = 3$ appears to be optimal.

Next, we compare three higher order interpolation methods—Taylor, Orthogonal Polynomials, and Hermite—by plotting their relative RMS error. In the left graph of Figure 2, the number of intervals per section varied; in the right graph, the order is varied. We observe that the method of orthogonal polynomials is far superior to the others and so is the one used.

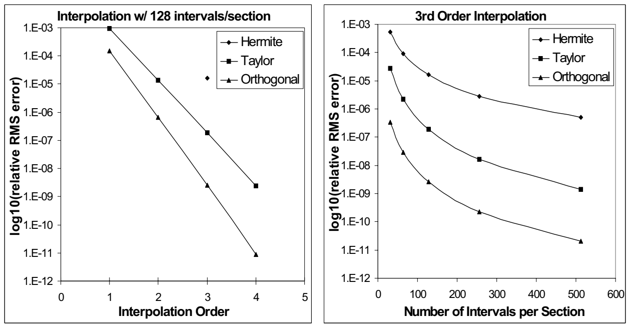

Figure 2: Interpolation comparisons.

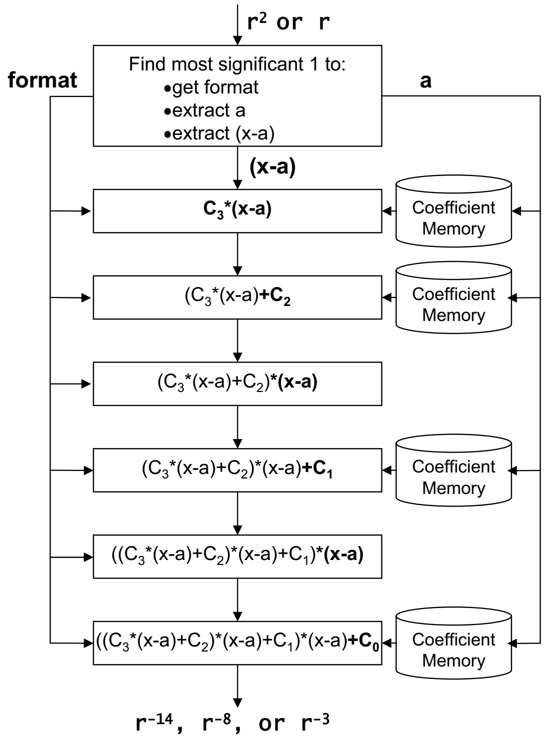

Figure 3: Position of the leading 1 determines the operand format in the interpolation pipeline.

We now describe the interpolation pipeline (see Figure 3. 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 $x \equiv r^2$ = input, a = the index of the interval from the beginning of the section (see Figure 1), and $x - a$ = the offset into the interval. The coefficients $C_0, \ldots, C_3$ are unique to each interval, and are retrieved by determining the section and interval.

Proper encoding makes extraction of the section, interval, and offset trivial. 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 3, there are four table look-ups from coefficient memory, (one for each coefficient), three multiplies, and three adds.

2.3 Semi Floating Point

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.

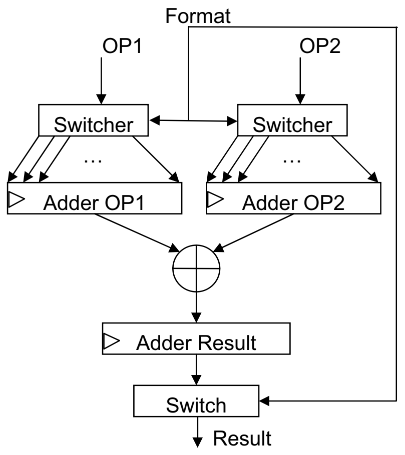

Figure 4: Semi FP adder with explicit alignment.

The critical observations concern the computation shown in Figure 3. 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. Fourth, the possible pre-computed shifts (and only those shifts) are hardwired as shown in Figure 4. Finally, 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 3). The LJ pipeline uses 11 formats, the Coulomb 14; as there is no overlap in formats, the combined LJ/Coulomb pipeline uses 25 formats.

We now compare the semi floating point to full floating point implementations. For the latter we use LogiCore Floating-point Operator v2.0 from Xilinx [25]. Table 2 gives the number of slices (Xilinx V2 family) required to implement various components. The final column gives the number of slices for the three versions for one entire non-bonded 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.

Table 2: Shown is the resource usage (in slices for the Xilinx V2 family) of various components.

Format	Adder	Mult.	Complete Force Pipeline
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

The practical result is that, for the Xilinx Virtex2-Pro VP70, two force pipelines can be implemented using either single precision floating point or semi floating point, the former resulting in a loss of precision from 35 to 24 bits. As shown below and in [2], this difference could be critical to simulation accuracy. On the other hand, semi floating point precision could scale at least up to 40 bits (without serious optimization), with a slight reduction in operating frequency. With respect to double precision floating point: fitting even one pipeline on the VP70 is impossible with our current pipeline design. This experience with floating point units is similar to that of two other studies [13, 19].

2.4 Multigrid overview

We now sketch the long-range force implementation using multigrid, for details please see [9]. The basic problem of Coulomb force computation is to compute the potential distribution by solving the Green's function for the given charge distribution. Grid-based algorithms map a smoothing function defined in the continuous coordinate space to one defined in the discrete grid coordinate space. The operations can be classified into two types: particle-grid (and grid-particle) and grid-grid. It follow that the multigrid processor requires two kinds of computation modules: a particle-grid (and grid-particle) converter and a grid-grid convolver.

The particle-grid converter performs assignment (or interpolation) between particles and their the neighboring grid points using appropriately chosen basis functions. This takes three steps: (i) scaling particle coordinates to grid coordinates, (ii) computing the the assignment and/or the interpolation weights, and (iii) multiplying the weights by the charge (or the potential on the grid point).

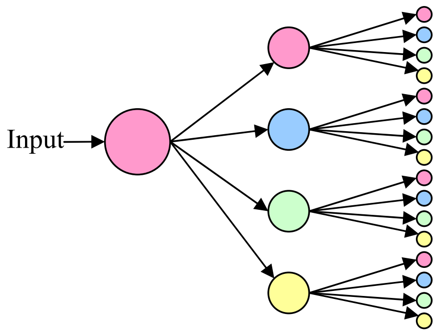

Figure 5: One quarter of a 1:64 particle-grid converter tree structure.

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 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 cube of the matching color in the basis function pipeline. The circles of matching color in the last column share the same outputs from a single basis function pipeline, as do those in the second column (not shown here). This structure has 4^3-way parallelism with only three basis function pipelines for three dimensions. One issue with the particle-grid converter is that a large number of grid points must be accessed on every cycle; this requires both high bandwidth and highly parallel addressing logic. Fortunately, modern FPGAs, with their hundreds of independent Block RAMs, have just such capability.

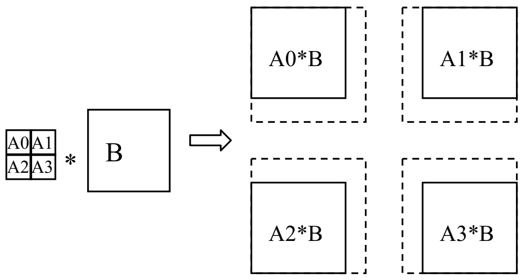

Figure 6: Splitting a convolution.

The 3D grid-grid convolver is constructed from 2D-convolvers in series with 2D FIFOs. The 2D-convolvers, in turn, are constructed with 1D-convolvers in series with 1D FIFOs (see, e.g., [22]). Configuring the lengths of these FIFOs allows us to adapt the logic to handle large input matrices of various sizes. That is, by splitting a big convolution into several small ones and routing results to the proper destinations, we can handle various large matrices. As is shown in the example in Figure 6, the 2D matrix A is too big to convolve with matrix B directly. Therefore, it is split into 4 small pieces, A0 to A3, each of which is 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 location yields A*B.

2.5 Precision versus Quality

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:

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

We ran a set of experiments based on two versions of our serial reference code, reproducing as closely as possible the experiments done by Amisaki, *et al.* [2]. The first used double precision floating point, the second tracked the hardware implementation using 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.

We also ran a set of experiments to find the relationship between energy fluctuation and precision. In agreement with [2], 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 datapath 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 7. 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 [23]. We plot R in the right half of Figure 7. 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 hard multipliers available on the current Xilinx FPGAs, we round up to the next time-area discontinuity and obtain a 35-bit datapath. This design has from a factor of 10× to 50× more energy fluctuation

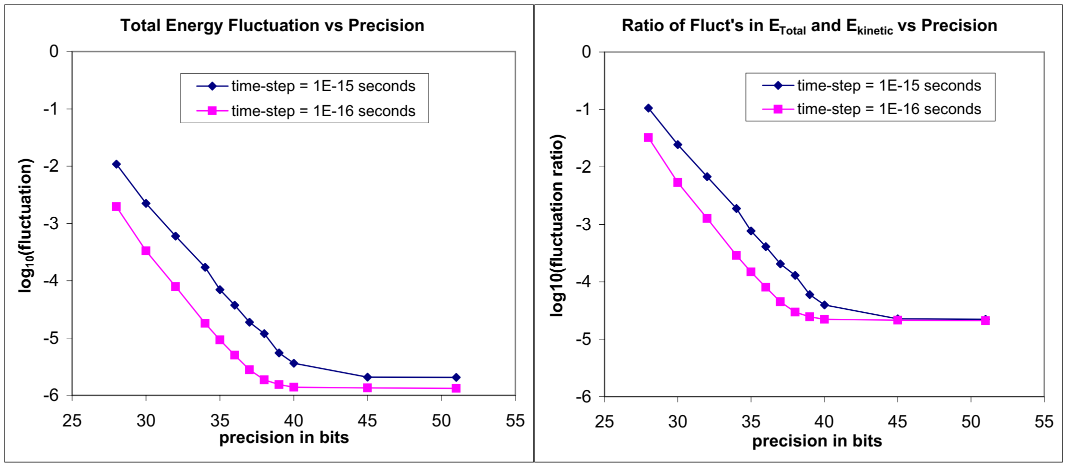

Figure 7: Shown is the effect of precision on two metrics for simulation accuracy: (a) Fluctuation of total energy and (b) the ratio of the fluctuations in total and kinetic energies.

than the best case, but between 100× and 500× lower R than what has been regarded as minimal to indicate "good behavior."

3 Design and Implementation

3.1 System Level Design

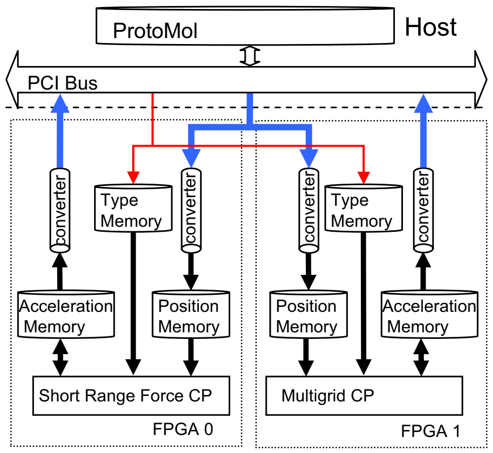

Figure 8: System block diagram.

One version of the overall system design is shown in Figure 8. We assume a two FPGA coprocessor such as the Annapolis Microsystems WidstarII-Pro. The short-range forces, with cell list support, are computed on one FPGA, while the long-range forces (using multigrid) are computed on the other. These FPGAs can execute concurrently. A potential optimization, especially when the long-range force is not computed every time-step (every 2nd to 10th cycle is common), is to use both FPGAs for the short-range force and only configure one of the FPGAs for the long-range force when needed. Position memory and type memory are duplicated. The converters translate double precision floating point numbers used by the host into 35-bit semi-floating format on-the-fly.

3.2 Multigrid Implementation

The multigrid implementation is shown in Figure 9. It consists of the particle-grid converter, the grid-grid convolver, the interleaved memory interface, control logic, and various miscellaneous components. The control logic routes data, according to the multigrid algorithm, by providing appropriate MUX settings and memory addresses; the compute modules can thus be re-used in multiple operations. Because the grid-grid convolver only inputs and outputs one datum per clock cycle, it uses the block RAMs directly. The finest grid, however, needs the complex memory interleaving described above. The Q-store and V-store hold the charges and potentials, respectively. The Type-Param memory is used to translate the particle-type indices into charge. A low-level optimization is that the final vector-product multiplier shares HW multipliers with the convolver.

3.3 Short-Range Force Implementation

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.

The cell lists are constructed on the host by the original MD code, although the data ordering is modified 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

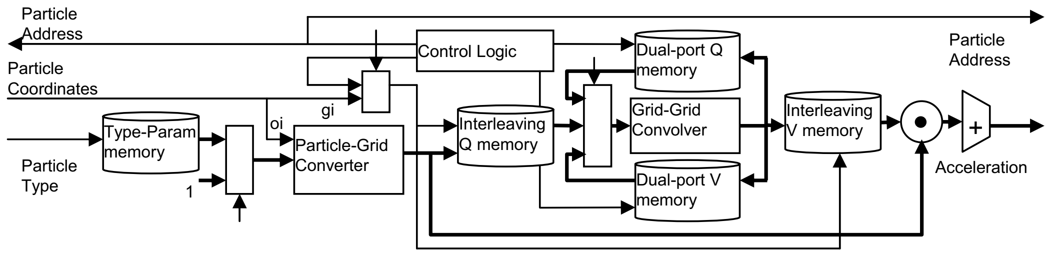

Figure 9: Multigrid schematic.

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.

The force pipeline array is shown in Figure 10. 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 acceleration is fetched from acceleration memory for later accumulation.
2. In every following cycle, M particles from a neighboring cell B are loaded into the P_j registers. At the end of each force pipeline, the results are not only accumulated with accumulations of particles from cell B, but also with that of the particle in the P_i register through an adder tree. P_i's temporary 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.

To deal with large models we have implemented a programmer-controlled caching scheme. For this we use the six SRAMs around each FPGA on our Annapolis Microsystems board. These have a total capacity of 12Mb and a bandwidth of 432 bits per cycle. We instantiate two sets of caches on each FPGA chip, each of which can store 2048 particles. Total capacity is 256K particles. Performance is independent of model size up to the capacity of the system.

4 Validation and Results

The primary target system consists of a PC with a 2.8 GHz Xeon CPU and a WildstarII-Pro PCI board from Annapolis Micro Systems [3]. The board has two Xilinx Virtex-II-Pro XC2VP70 -5 FPGAs. 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. FPGAs run at 75MHz in both short- and long-range computation modes. Model sizes of up to 256K particles are supported.

The design was validated against three serial reference codes: our base code ProtoMol [16], 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, however, as is done using the method described in the previous section. 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.

For performance comparisons, we simulated the Protein Data Bank *Molecule of the Month* for January, 2007,

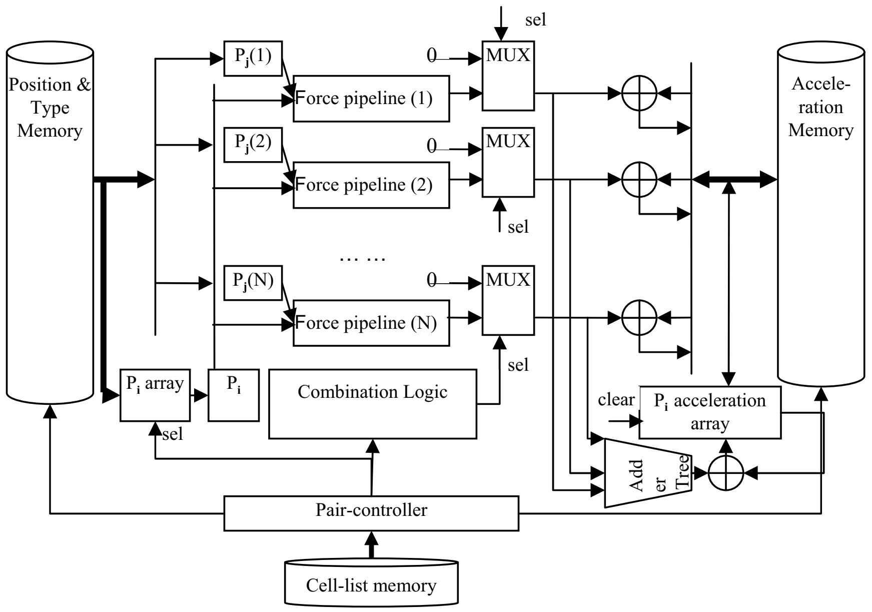

Figure 10: Force pipeline array.

Table 3: 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, the long-range and short-range forces overlap.

	Short Range Force	Long Range Force	Bonded Forces	Motion Integration	Comm. & overhead	init. & misc.	TOTAL
FPGA Accelerated Protomol	533.3	75.3	21.5	20.8	25.6	9.2	590
PC-only ProtoMol	3867.8	234.1	21.6	21.5	—	12.9	4157
PC-only NAMD		177.3					3726

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 3 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 [20], and (ii) that the overhead is a small fraction (roughly 6%) of the execution time. The total speed-up is 7×. For further reference, we also downloaded and ran a NAMD binary (v2.6 b1) [2]; these results are also shown in Table 3. NAMD is somewhat faster than ProtoMol; the resulting speed-up is 6.3×. These numbers are clearly preliminary as there is substantial room for performance improvement in both baseline and FPGA-accelerated configurations; this is now described.

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.

Long-range computation. The serial multigrid long range force computation shown in Table 3 seems slow (see, e.g. [12]). 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 multigrid computation appears to be good, the use of computational resources is disproportionate to its execution time (again, as observed previously by [20]). 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-

[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

Table 4: 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	.59
ProtoMol w/ multigrid every 4th cycle (dynamic reconfiguration)	.37
ProtoMol w/ reduced precision and multigrid every 4th cycle (dynamic reconfiguration)	.22
Accelerated configuration, simulation only	
ProtoMol w/ single VP100 w/ multigrid every 4th cycle (dynamic reconfiguration)	.36
ProtoMol w/ single VP100 w/ multigrid every 4th cycle (dynamic reconfiguration), reduced precision	.31
From NAMD website	
NAMD w/ PME every 4th cycle 90K particle Model	2

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% increase in operating frequency and thus performance.

Newer chip. Our experiments with the Xilinx V4 and V5, and with the Altera Stratix-II are still in progress. Although we do not anticipate a substantial increase in logic per chip that could result in increased parallelism, an increase in operating frequency is likely.

Reduced precision. If reduced precision is acceptable, this also allows more pipelines to be implemented per FPGA, again, with a proportional increase in performance. Four pipelines are then possible using the VP70, increased from the two with 35-bit precision.

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

5 Discussion

We have described a study of FPGA acceleration of molecular dynamics simulations. We differentiate our work in that it combines the following: a high level of FPGA-specific design, systematically determined precision, support for complex force models, and support for MD simulations of up to 256K particles.

Comparing MD performance of FPGA-based systems will be frought 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 4):

- 8.6× when comparing NAMD run in our lab versus ProtoMol accelerated with two VP70s (3.2 versus .37); this reduces to around 5× when comparing with external NAMD reports (less-than-2 versus .37).

- Similar results are obtained by using a single VP100 than two VP70s (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 10× versus NAMD run in our lab (3.2 versus .31), and 6.5× versus external NAMD reports (less-than-2 versus .31).

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 technology does not change, HPRC for MD may still be promising. We have shown that a factor of 5× to 10× speed-up is achievable using a VP100 accelerator versus a highly tuned code. 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.

Acknowledgments. 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 helpful suggestions.

References

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

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

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

[4] Azizi, N., Kuon, I., Egier, A., Darabiha, A., and Chow, P. Reconfigurable molecular dynamics simulator. In *Proc. Field Prog. Custom Computing Machines* (2004), pp. 197–206.

[5] Beauchamp, M., Hauck, S., Underwood, K., and Hemmert, K. Architectural modifications to improve floating-point unit efficiency in FPGAs. In *Proc. Field Prog. Logic and Applications* (2006), pp. 515–520.

[6] Briggs, E., Sullivan, D., and Bernholc, J. Real-space multigrid-based approach to large-scale electronic structure calculations. *Phys. Rev. B 54* (1996), 14362–14375.

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

[8] Freddolino, P., Arkhipov, A., Larson, S., McPherson, A., and Schulten, K. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure 14* (2006), 437–449.

[9] Gu, Y., and Herbordt, M. C. FPGA-based multigrid computations for molecular dynamics simulations. In *Proc. Field Prog. Custom Computing Machines* (2007).

[10] Gu, Y., VanCourt, T., and Herbordt, M. C. Accelerating molecular dynamics simulations with configurable circuits. In *Proc. Field Prog. Logic and Applications* (2005).

[11] Gu, Y., VanCourt, T., and Herbordt, M. C. Improved interpolation and system integration for FPGA-based molecular dynamics simulations. In *Proc. Field Prog. Logic and Applications* (2006), pp. 21–28.

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

[13] Kindratenko, V., and Pointer, D. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *Proc. Field Prog. Custom Computing Machines* (2006).

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

[15] Lee, S. An FPGA implementation of the smooth particle mesh ewald reciprocal sum compute engine (RSCE). Master's thesis, University of Toronto, 2005.

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

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

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

[19] Scrofano, R., Gokhale, M., Trouw, F., and Prasanna, V. A hardware/software approach to molecular dynamics on reconfigurable computers. In *Proc. Field Prog. Custom Computing Machines* (2006).

[20] Scrofano, R., and Prasanna, V. Preliminary investigation of advanced electrostatics in molecular dynamics on reconfigurable computers. In *Supercomputing* (2006).

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

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

[23] van der Spoel, D. Gromacs exercises. CSC Course, Espo, Finland, February 2004.

[24] van der Spoel, D., Lindahl, E., Hess, B., Groenhof, G., Mark, A., and Berendsen, H. GROMACS: fast, flexible, and free. *Journal of Computational Chemistry 26* (2005), 1701–1718.

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