FPGA-BASED MULTIGRID COMPUTATION FOR MOLECULAR DYNAMICS SIMULATIONS

Yongfeng Gu **Martin C. Herbordt**

Computer Architecture and Automated Design Laboratory
Department of Electrical and Computer Engineering
Boston University, Boston, MA 02215
{maplegu|herbordt}@bu.edu

Abstract: FPGA-based acceleration of molecular dynamics (MD) has been the subject of several recent studies. Implementing long-range forces, however, has only recently been addressed. Here we describe a solution based on the multigrid method. We show that multigrid is, in general, an excellent match to FPGAs: the primary operations take advantage of the large number of independently addressable RAMs and the efficiency with which complex systolic structures can be implemented. The multigrid accelerator has been integrated into our existing MD system, and an overall performance gain of *5x* to *7x* has been obtained, depending on hardware configuration and reference code. The simulation accuracy is comparable to the original double precision serial code.

1. INTRODUCTION

Molecular Dynamics simulations (MD) are a fundamental tool for gaining the understanding of chemical and biological systems. In one of its most impressive applications, the complete satellite tobacco mosaic virus was simulated [8]. The computational cost, however, was over a month of compute time on a 256-node cluster. As this and many other studies indicate, accelerating molecular dynamics is one of the most important problems in high-performance computing.

MD is an iterative technique that runs in phases: the forces on each particle are computed, then applied using equations of motion. In modern MD systems, the force computations often involve many terms, including bonded (covalent, hydrogen) and non-bonded (van der Waals, Coulombic). The non-bonded force is often partitioned into short- and long-range components. The complexity of the motion update and the short-range force computations is *O(N)* in the number of particles, and generally requires only a small fraction of overall compute time. The complexity of the non-bonded force computations is $O(N^2)$ in the direct implementation, and comprises the bulk of the computation. Complexity can be reduced substantially, however. For the short range component, *O(N)* is obtained by dividing the system into cells and/or maintaining lists of particles within a certain distance of a given particle. For the long-range component, *O(NlogN)* or better is obtained with transform- or grid-based methods.

Several recent efforts have demonstrated the viability of FPGA-based acceleration of MD (FPGA/MD) [1][2][9][10][12][13][18][19]. The FPGA/MD design space defined by these studies is spanned by several axes:

- Precision: Is 53 bits used (double precision), or 24 (single precision), or something else? How is the choice motivated?
- Arithmetic mode: Is floating point used? Block floating point? Scaled binary? Logarithmic representation? A hybrid representation?
- 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?
- Design flow: How is the FPGA configured? With a standard HDL, or a C-to-gates process, or some combination?
- Scope: MD implementations have a vast number of variations – which are supported? In particular, how is the long-range force computation performed? With cutoff or a switching function? Or, is a more accurate, and more computationally complex, method used? Is this done on the FPGA or in software?

This study concentrates on the last of these issues: We use the multigrid method to implement the long-range force computation on the FPGA.

Multigrid for FPGA/MD has two major advantages. First, it is a fast and accurate method for solving boundary value problems such as the electrostatic computation that arises in MD [3][5][7][11][16][17][20]. Second, it maps extremely well to FPGAs. The primary operations are: applying charges onto a 3D grid, performing convolutions on 3D grids, and applying a 3D grid back onto the particles. The first and third of these can be implemented to take advantage of the FPGA's high-performance support of complex memory access (as demonstrated, e.g., in [22]); the second has often been shown to yield high efficiency. A further advantage of using multigrid is that implementing the 3D FFT on the FPGA is avoided, although this is approach has also been shown to be viable [14].

We integrate our long-range force computation into our existing ProtoMol-based FPGA/MD system, and show factors of *5x* to *7x* speed-ups (on 2004 era hardware) over PC-only execution while retaining accuracy.

The significance of this work is as follows: we demonstrate an FPGA algorithm for the multigrid grid method; we show that multigrid can be used to provide substantial speed-up for the MD long-range force computation; and that, therefore, substantial acceleration can be obtained for complete FPGA/MD simulations.

2. COULOMB FORCE COMPUTATION AND MULTIGRID METHOD

2.1. MD Force Computation

In general, the forces being simulated depend on the physical system under study and can include van der Waals (Lennard-Jones or LJ), electrostatic (Coulomb), hydrogen bond, and various covalent bond terms:

$$\vec{F} = \vec{F}^{bond} + \vec{F}^{angle} + \vec{F}^{torsionl} + \vec{F}^{H-bond} + \vec{F}^{non-bond} \quad \textbf{Eq 1}$$

Because the hydrogen bond and covalent terms (bond, angle, torsion) affect only neighboring atoms, computing their effect is *O(N)* in the number of particles *N* being simulated. These, as well as the motion update (also *O(N))*, are therefore generally computed on the host. The LJ force for a particle i can be expressed as:

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

where ε_{ab} and δ_{ab} are parameters related to the types of particles. The Coulombic force can be expressed as:

$$\vec{F}_i^{CL} = q_i \sum_{j \neq i} \left(\frac{q_j}{|r_{ji}|^3} \right) \vec{r}_{ji} \quad \textbf{Eq 3}$$

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 is often 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.

The Ewald method computes the long range Coulomb force with periodic boundary conditions, i.e., where the system is replicated infinitely in all directions. As the Coulomb force on a particle involves interactions with all other particles, the result is an infinite summation. The Ewald method solves this by splitting the summation into two parts, real and reciprocal. The real part is fast converging and can be computed accurately with a cut-off in *O(N)*; the reciprocal part is also fast converging, but in reciprocal space. The overall complexity, including FFT, is $O(N^{3/2})$. Various improvements reduce the complexity to *O(NlogN)* [6][7][25].

An alternative approach is the multigrid method, which can operate directly on the Coulomb force rather than the reciprocal part of the Ewald sum. It therefore does not need an FFT. Also, it does not have to be applied with the periodic boundary condition; and finally, it can be applied to systems with non-uniform distributions.

2.2. The Multigrid Method

Many important computations, from solving systems of equations to solving partial differential equations (PDEs), can be executed by discretizing to a grid and iteratively performing operations in all neighborhoods. Multigrid algorithms improve the convergence rate of basic finite difference methods by using a hierarchy of discretizations, often reducing complexity to *O(N)* in the number of grid points. The up and down traversal of the grid hierarchy is called a V-cycle.

```
Function u^l = V-Cycle(u^l, q^l, l)
Begin
    1. If this is coarsest grid, solve L^l * u^l = q^l
       and return u^l.
    2. u^h = Relax0( u^l, q^l, l )
    3. r^l = q^l - L^l * u^l
    4. q^(l+1) = A^(l+1)_l * r^l
    5. u^(l+1) = 0
    6. u^(l+1) = V-Cycle( u^(l+1), q^(l+1), l+1 )
    7. u^l = u^l + I^l_(l+1) * u^(l+1)
    8. u^l = Relax1( u^l, q^l, l )
    9. Return u^l
End
```

Algorithm 1. The general multigrid V-cycle

We now sketch the general multigrid algorithm following the presentation by Yavneh [24]. Algorithm 1 is a series of recursive V-cycles. The fine-to-coarse operation is referred to as *restriction*, coarse-to-fine as *prolongation*.

We begin with q^n, the known parameters on the current grid (e.g., the charge distribution), and finish with the solution u^n (e.g., the potential distribution) of the PDE $u^n = L^* q^n$. Step 1 solves the PDE directly, if the grid is small enough. Step 2 does the first relaxation to give a guess of the solution. Step 3 computes the residual (error) of the guess. Step 4 restricts the known parameters to a coarser grid (having fewer grid points and unknown variables) with a basis function A. Step 5 specifies the boundary condition, for example, by setting u^{l+1} to zero for a vacuum. Step 6 calls the V-cycle recursively to solve the residual. Step 7 uses function I to prolongate the solution of the residual as a correction from the coarse grid back to the current grid, and integrates

the correction with the guess. Step 8 does another round of relaxation. Step 9 returns the solution.

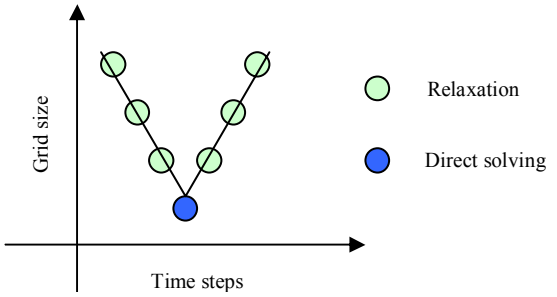
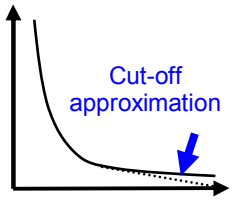


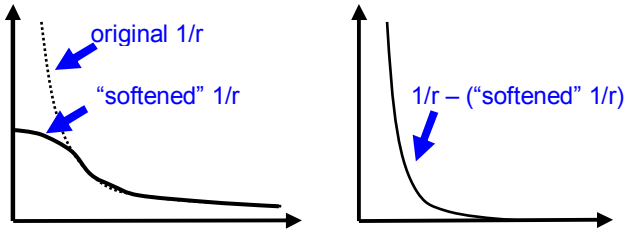
Figure 1. V-cycle schematic

Figure 1 shows a V-Cycle. With a constant number of iterations per level and a geometric reduction in grid points per level, the resulting complexity is O(N).

2.3. Multigrid for Coulomb Force Computation

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





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

More formally (and following the presentation by Skeel, et al. [20]), the problem of Coulomb force computation is to compute the potential distribution by solving the Green's function for the given charge distribution. The electrostatic potential is expressed as:

$$V_i^{CL} = \sum_{j \neq i} \frac{q_j}{|r_{ji}|} \qquad \textbf{Eq 4}$$

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

$$\frac{1}{r} = \left(\frac{1}{r} - g_a(r)\right) + g_a(r) \qquad \textbf{Eq 5}$$

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

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

$$\sum_{i=1}^{N}\sum_{j=1}^{N} q_i q_j g_a(|\vec{r}_j - \vec{r}_i|) = \sum_{k}\sum_{m} q_{h,m} q_{h,n} g_a(|\vec{r}_{h,k} - \vec{r}_{h,m}|) \qquad \textbf{Eq 6}$$

where q_i and q_j are particle charges, and $q_{h,m}$ and $q_{h,n}$ are charges at points m and n on a grid with spacing h. There are many approaches to solving the smoothing functions on a grid, including the Particle Mesh Ewald algorithm, the multigrid based algorithms used here, and even (for small scale problems) direct computation.

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

$$g_a(r) = (g_a(r) - g_{2a}(r)) + g_{2a}(r) \qquad \textbf{Eq 7}$$

The local correction $g_a(r)$-$g_{2a}(r)$ takes out the high frequency component of $g_a(r)$ and becomes short range too. The even slower varying $g_{2a}(r)$ can now be approximated more efficiently on the even coarser grid and so on until the coarsest grid. This specifies the first half of V-cycle.

After the potential has been computed on the coarsest grid, we still need to compute the slowly varying component and then to apply it to each particle. First, the potential on the coarsest grid is interpolated back to the finer grids and combined with local corrections on each level. This comprises the second half of the V-cycle. Finally, forces are generated by differentiating the potential on the finest grid in three dimensions, and interpolating the result to each particle's position.

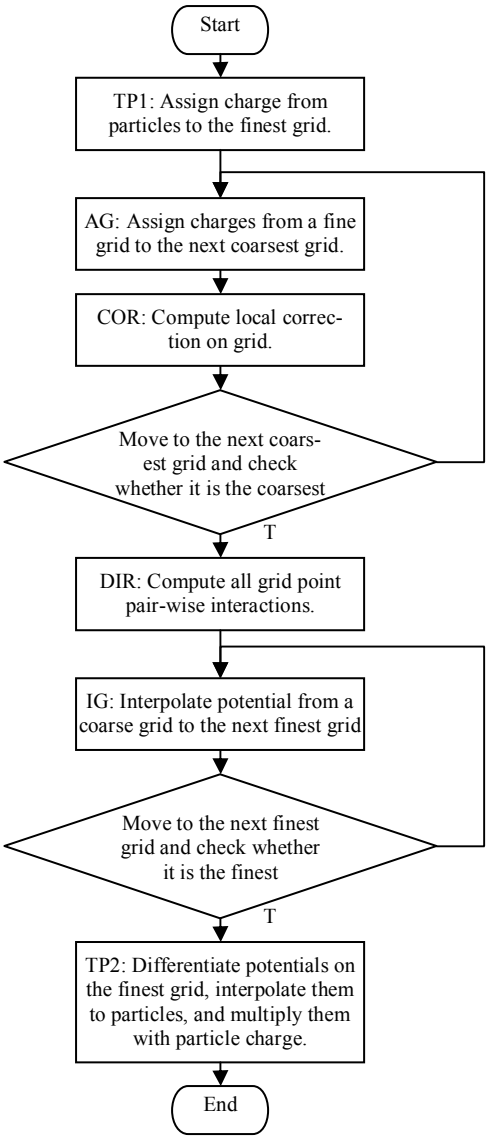


Figure 2. Flowchart of multigrid method for the Coulomb force

The flow chart of this algorithm is shown in Figure 2. The operations can be classified into two types: particle-grid conversion and grid-grid steps. The particle to grid charge assignment (TP1) and the grid to particle potential interpolation (TP2) both apply a basis function $\Phi(w)$ and its gradient. A 3rd order and a 5th order basis function are proposed in [20]. The assignment is computed as:

$$Q^1(x,y,z) = \sum_m Q_m * \phi(|x_m - x|) * \phi(|y_m - y|) * \phi(|z_m - z|) \quad \textbf{Eq 8}$$

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

$$F_{m,x} = Q_m * V(x,y,z) * d\phi(|x_m - x|) * \phi(|y_m - y|) * \phi(|z_m - z|) \quad \textbf{Eq 9}$$

where $F_{m,x}$ is the force in x direction, which is interpolated from the grid point (x,y,z). The other operations (AG, COR, DIR, IG) can be represented as 3D matrix convolutions. According to Eq 5, the smoothing function matrix on grid Ω^l can be expressed as:

$$G^l \approx \hat{G}^l + I^l_{l+1} G^{l+1} A^{l+1}_l \quad \textbf{Eq 10}$$

where A^{l+1}_l is the assignment matrix that anterpolates charge from the fine grid Ω^l to the coarse grid Ω^{l+1}; I^l_{l+1} is the interpolation matrix that interpolates potential from coarse grid to fine grid; and $\hat{G}^l$ is the local correction matrix. Given a charge matrix Q^l, the potential matrix V^l is computed as:

$$V^l \approx \hat{G}^l \cdot Q^l + I^l_{l+1} G^{l+1} (A^{l+1}_l \cdot Q^l), \text{ l=1,2,.., L-1} \quad \textbf{Eq 11}$$

On the coarsest grid Ω^L, G^L denotes the direct computation between all grid point pairs.

Eq 11 reveals that, except on the coarsest grid, three convolutions are performed on every grid: (i) to assign charge distribution to the next coarser grid, (ii) to compute the local correction, and (iii) to interpolate the potential from the next coarser grid back to current grid. The coarsest grid only has one convolution with G^L. In addition, because G^L, $\hat{G}^l$, A^{l+1}_l and I^l_{l+1} are independent of l and $I^l_{l+1} = (A^{l+1}_l)^T$, only three convolution cores need to be precomputed.

3. DESIGN FOR FPGA ACCELERATION

3.1. Overview

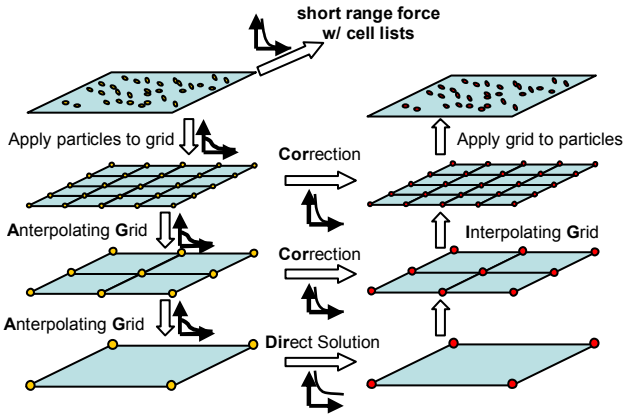


Figure 3. Multigrid method for the Coulomb force

The overall algorithm is shown schematically in Figure 3. Starting at the upper left, the per-particle potentials are partitioned into short and long range components. The short range (van der Waals and short range component of the Coulombic) is computed directly, e.g. with cell lists, while the long range is applied to the finest grid. Here the force is split again, with high-frequency component solved directly and the low-frequency passed on to the next coarser grid. This continues until the coarsest level where the problem is solved directly. This direct solution is then successively combined with the previously computed finer solutions

(corrections) until the finest grid is reached. Here the forces are applied directly to the particles.

Following the two types of operations just described, our multigrid coprocessor requires two kinds of computation modules: a particle-grid converter and a grid-grid convolver. Because these operations are executed sequentially, computation modules can be shared. On-chip memories are a critical part of the multigrid coprocessor: not only do they provide scratch space between operations, they also merge computation models with different data access interfaces. In the rest of this section, we describe: the FPGA-specific multigrid method, our computation modules, and the interleaving memory structure.

3.2. Particle-Grid Converter

The particle-grid converter applies Eq 8 (or Eq 9) to perform assignment (or interpolation) between particles and their the neighboring grid points.

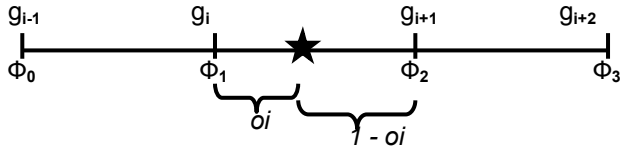


Figure 4. Extracting grid index and offset

We scale our coordinates to match the finest grid. In one dimension (see Figure 4), we can partition the particle position into two components *gi*|*oi* where *gi* is the index of the previous point and *oi* is the distance from the grid point. It then suffices to use *oi* alone to compute the contributions of *q* to any surrounding neighborhood of *gi*'s.

The basis function *Φ(w)* (or *dΦ(w)*) is used. This takes 3 steps: (i) scaling the particle coordinates to grid coordinates to extract the grid index *gi* (x_m,y_m,z_m) and offset *oi* (|x_m-x|,|y_m-y|,|z_m-z|); (ii) computing the assignment (or interpolation) weights *Φ(w)* (or *dΦ(w)*); and (iii) multiplying the weights by the charge (or potential) on the grid point. We normalize grid cell sizes to be powers of 2 so that scaling particle coordinates to grid coordinates only requires zero-cost shifting. The bits to the left of the binary point are then *gi,* those to the right *oi*.

For *Φ(w)* (or *dΦ(w)*) (we use those derived by Skeel, et al [20]), instead of computing *Φ(w)* directly, we modify the basis function to be a set of polynomials of *oi* for the particle's neighboring supporting grid points. For example, Eq 12 is the 3rd order basis function applied by our coprocessor, and *w* is the distance between particle and grid points:

$$\phi(w) = \begin{cases} (1-|w|)(1+|w|-\frac{3}{2}w^2), & |w| \le 1 \\ -\frac{1}{2}(|w|-1)(2-|w|^2), & 1 \le |w| \le 2 \\ 0, & |w| \ge 2 \end{cases} \qquad \textbf{Eq 12}$$

By substituting *w* with *oi+1*, *oi*, *1-oi*, and *2-oi*, we have four polynomials corresponding to the four neighboring grid points (as shown in Figure 4):

$$\begin{cases} \phi_0(oi) &= -\frac{1}{2}oi^3 + oi^2 - \frac{1}{2}oi \\ \phi_1(oi) &= \frac{3}{2}oi^3 - \frac{5}{2}oi^2 + 1 \\ \phi_2(oi) &= -\frac{3}{2}oi^3 + 2oi^2 + \frac{1}{2}oi \\ \phi_3(oi) &= \frac{1}{2}oi^3 - \frac{1}{2}oi^2 \end{cases} \qquad \textbf{Eq 13}$$

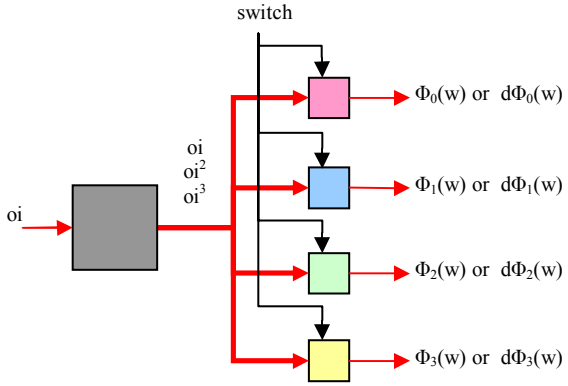


Figure 5. Basis function pipeline for Φ(w) and dΦ(w). 'switch' selects the output from Φ(w) and dΦ(w).

These polynomials, as well as the four for *dΦ(w)*, are functions of *oi*, oi^2, and oi^3. The basis function pipeline shown in Figure 5 computes all eight polynomials from a common input.

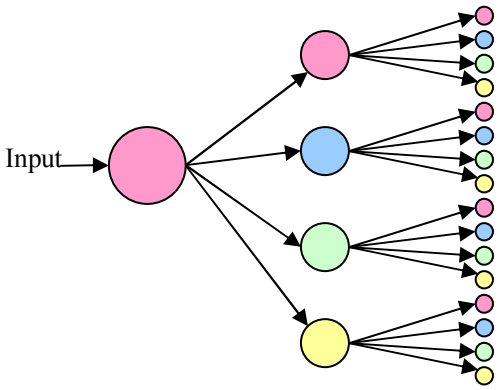


Figure 6. One quarter of a 1:64 particle-grid converter tree structure.

With a P^{th} 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 6 shows one quarter of the tree structure of a $1:4^3$ particle-grid converter. Each color circle multiplies its input by *Φ(w)* (or *dΦ(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.

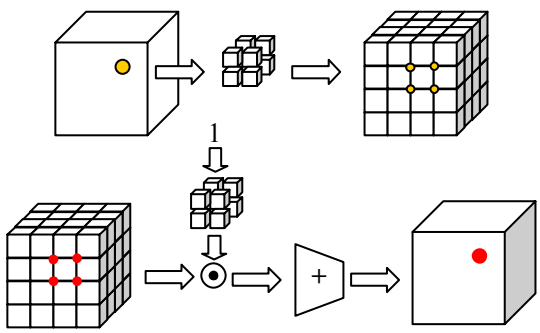


Figure 7. Charge assignment (above) and potential interpolation (below)

During charge assignment, the input is the particle charge, and the outputs are the charges assigned to neighboring grid points. For interpolation of the potential, the input is the constant 1 (see Figure 7), and the 'switch' of each basis function pipeline (see Figure 5) is set to select differentiation in three dimensions. The outputs are the weights to be multiplied with the potential values from the neighboring grid points. The weighted sum is the potential on a particle. Figure 7 shows these two processes.

3.3. Interleaved Memory

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. The interleaved memory design described in [22] is one such example.

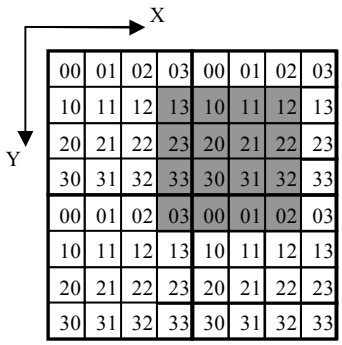


Figure 8. A 2D 4^2-way interleaved memory

We begin by illustrating the 2D 4x4-way interleaving memory used to store the original 2D grid. Given an address reference (x,y), the grid points within a 4x4 window, i.e. (x,y), (x,y+1), …, (x+3,y+3), must be accessed. Obviously, 16 independent memory accesses are required for each interleaved memory access. As shown in Figure 8, when grid points are stored in 16 separate banks marked from 00 to 33, any 4x4 access window contains exactly one grid points from each bank. The bank's index is either the same as that of the bank of the reference address, or the one greater than that in the X and/or Y dimension, respectively. The outputs from memory banks are shifted (with rotation) in both X and Y based on the reference address. In Figure 8, processing the reference address (1,3) outputs a rotational shift left by 3 mod 4, and up by 1 mod 4. Inputs are shifted the opposite way. 3D interleaving memory is analogous.

3.4. Grid-Grid Convolver

The grid-grid convolver is extended from our previous work [23]. The original design of the systolic array structure is described by McWhirther and McCanny [21]. The extension for the multigrid coprocessor is that both input matrices are allowed to be larger than the number of PEs in the systolic array, while our previous convolver only allowed one large matrix. The motivation is that the multiplication operator in this convolution requires several HW multipliers, which limits the number of PEs in the systolic array. The convolution operations, such as COR and DIR (from Figure 2), must therefore be computed in blocks. Also, because of the nature of multigrid, the size of input matrices varies among operations and iterations. A flexible convolver is thus essential to maintain efficiency.

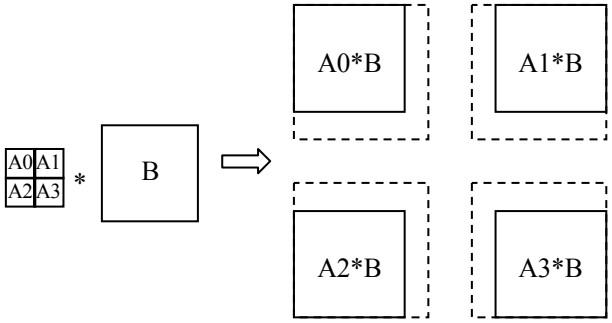


Figure 9. Splitting a convolution

The 3D-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. 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 9, 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.

4. IMPLEMENTATION AND RESULTS

4.1. System Level Design and Interface

The overall system is that used in previous work [9]: The host PC has a 2.8 GHz Xeon CPU and runs Windows-XP. This PC is also used to run the serial reference codes. The FPGA coprocessor is implemented on a WildstarII-Pro PCI plug-in board from Annapolis Micro Systems, which has two Xilinx Virtex-II-Pro XC2VP70 -5 FPGAs. Both are used and clocked at 75MHz.

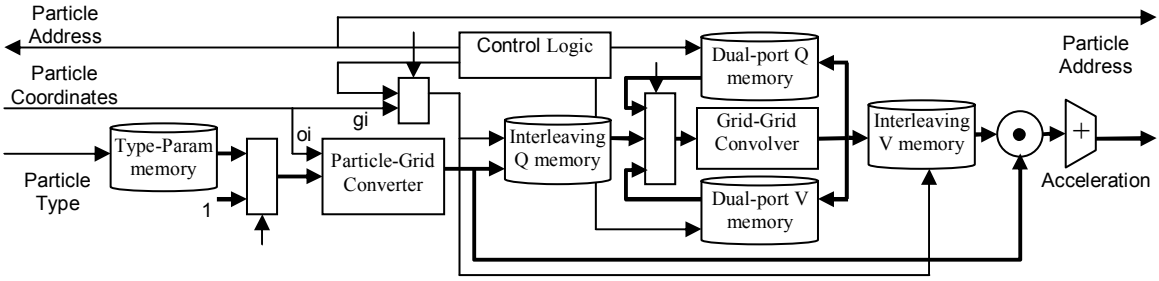


Figure 10. Multigrid schematic

All codes are compiled using Microsoft Visual C++ .NET with performance optimization set to maximum. Communication between PC and FPGA board is via APIs from Annapolis Micro Systems. At every time-step, particle data are DMA'ed back and forth over the PCI bus. FPGA configurations were created in VHDL; the development flow uses Xilinx tools and the Synplify Pro synthesizer.

The base MD system is ProtoMol 2.03 [15], a high-performance MD framework designed especially for ease of experimentation. The FPGA accelerator is integrated into ProtoMol by swapping the corresponding force calculation modules and performing the necessary translations as described in [10]. The bonded forces (Angle, Bond, Dihedral, and Improper) and motion integration are computed on the host PC. The arithmetic precision is 35 bits, as motivated in [10]. The arithmetic mode is semi-floating point, as described in [9].

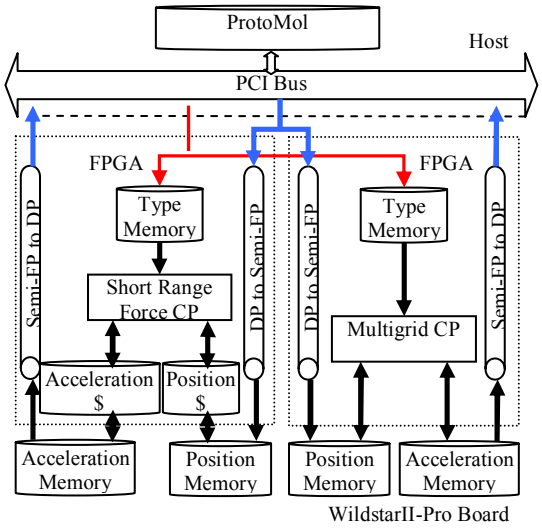


Figure 11. System block diagram

4.2. Multigrid Implementation

The overall system design is shown in Figure 11. 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 can execute in parallel; a system with a single FPGA, however, can operate efficiently by reconfiguring between computations. The position memory and the type memory are duplicated. The converters translate double precision floating point numbers into 35-bit semi-floating point format on-the-fly.

For models larger than about 10K particles, swapping data off-chip is required. The grid-grid convolutions do not need much bandwidth, so swapping there does not degrade performance. On the other hand, the particle-grid conversions rely heavily on the on-chip memories. Our solution is to process particles in groups based on their location. At any moment, only particles from one neighborhood are processed, which implies that only a small block of the finest grid is being accessed. Therefore, grid points are swapped in only when they are needed. The cell-lists constructed for the short range forces are applied for this purpose. Given the modest clock rate and the small fraction of time spent on these operations, it is not surprising that little relative slowdown results.

The multigrid implementation is shown in Figure 10. 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 algorithm in Figure 2, 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. Computations in involving the finest grid, however, need 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.

The 3rd order basis functions (Eq 12) are used. Adopting the 5th order basis functions slightly exceeds the VP70's HW multiplier count, but would not be a problem for the VP100. The finest grid size can be up to 32^3, and the grid cell size can be 1, 2, 4 or 8. The convolver contains 64 PEs, for a sustained throughput of 4.8 GFlops (35 bit precision). The system is capable of simulating 256K particles and 32 particle types.

Following the ProtoMol reference code, the multigrid coprocessor currently computes the long range term with two grid levels. The finest grid size is 28^3 with a convolution kernel of 13^3; COR is therefore run for only a single iteration. This is because the next level is used for computing the direct solution (DIR) on a 17^3 grid. Only two levels are needed because of the relative sizes of grid and kernel:

Table 1. Profile of the multigrid Coulomb force coprocessor. Fraction of total time spent in each phase in Figure 2.

Version	Particle-Grid		Convolutions				TOTAL
	TP1	TP2	AG	IG	COR	DIR	
PC Only	7.5%	12.3%	5.0%	3.6%	43.9%	27.7%	100%
FPGA Accelerated	1.1%	3.1%	1.3%	2.0%	60.7%	31.8%	100%

Table 2. Performance: Time required to run for 1000 time-steps (units in seconds)

	Short Range Forces	Long Range Forces	Bonded Forces	Motion Integration	Comm. & overhead	Init & misc.	TOTAL
FPGA ProtoMol Multigrid every cycle	533.3	75.3 (Multigrid)	21.5	20.8	25.6	9.2	589
PC only ProtoMol Multigrid every cycle	3867.8	234.1 (Multigrid)	21.6	21.5	0	12.9	4157
PC only NAMD SPME every cycle		177.3 (SPME)					3726

another iteration of COR (with a 17^3 grid and 13^3 kernel) would be similar work to DIR, and leave DIR still to be run.

It turned out that the particle-grid converter needed to have a $1{:}4^2$ two-level tree structure rather than the preferred three-level one. Consequently, the memory interleaving is 4^2-way. This configuration is a compromise resulting from resource balancing on our chip: the HW multipliers needed by a 1: 4^3 particle-grid converter--plus the attached vector multiplier--would have exceeded VP70 chip capacity. Our experiments also showed that a 4^3-way interleaved memory could just fit on a VP70, with the result that few resources would have remained for other functions. Another possibility would have been a 1: 2^3 particle-grid converter. While the 2^3-way interleaved memory is sufficiently small, this choice would have reduced performance. In contrast, the 4^2 configuration provides reasonable parallelism, while consuming 90% of the HW multipliers.

4.3. Validation

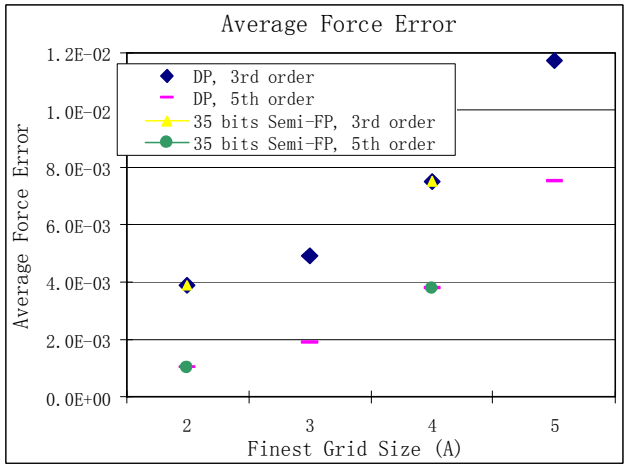
For validation, we simulated a model of more than 14,000 particles and 26 atom types (bovine pancreatic trypsin inhibitor or BPTI). After 10,000 time steps running on both the original ProtoMol and our accelerated version, we measured the total energy fluctuation. Both versions showed roughly $5*10^{-4}$ with the FPGA version showing slightly lower fluctuation.

We are also able to validate the quality of the multigrid computation directly. Since the force computation is well-defined, we are able to evaluate the output with respect to an alternative of arbitrarily high accuracy. Again following the methods used by Skeel et al. [20], we evaluate multigrid with respect to F_{avg}, the normalized average error of the force on every particle:

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

where $\vec{F}_i$ is the force computed using the reference method and $\tilde{\vec{F}}_i$ the force computed by the FPGA. Several other measures are suggested, but as our experiments show similar results for all, we present only those for F_{avg}.

For the model, we again use 14K particles including water and BPTI. The FPGA multigrid is as described: 2 levels of grid, 3rd and 5th order bases functions, smoothing cut-off of 10Å, and per dimension ratio between grid levels of 2. We used two grid spacings: 2Å and 4Å. For the serial version of ProtoMol we use the same parameters, but add two grid sizes, 3Å and 5Å, and use double precision floating point. Results are shown below.



The most important result is that the difference between the serial and accelerated versions of multigrid is negligible, with the variation being a few parts in 10,000. Also interesting is the tradeoff between function order and grid size: using 5th order gives a similar result to the third order, but with 1/8th the computation.

4.4. Performance

Table 1 shows execution time profiles of the accelerated and unaccelerated versions of the multigrid processor. For both, most of the time is spent on the convolutions. Preliminary analysis shows that this fraction can be reduced, without penalty in accuracy, by reducing the convolution core to a size appropriate for the basis functions. Still, this profile indicates that to improve the performance, we should improve the performance of the convolver first. More PEs is the obvious answer and would require more HW multipliers. The major difference between the PC and FPGA profiles is the fraction of time spent performing the particle-grid computations; clearly the interleaved memory scheme yields excellent performance.

Performance comparisons of various configurations are shown in Table 2. The model used was Importin Beta bound to the IBB domain of Importin Alpha and has 77,000 particles in a 93Å^3 box. The ProtoMol configurations are as just described; NAMD was downloaded from the main web site [http://www.ks.uiuc.edu/Research/namd/] and compiled and run on the same PC. For multigrid, the FPGA version shows a speed-up of 3x over the software version. For reference, the NAMD computation of the long-range forces using SPME was slightly faster than the ProtoMol computation using multigrid. Izaguirre, et al. [11] have performed a more in-depth comparison and found multigrid to be faster, especially for parallel systems. The overall performance gain is a factor of *7.3x* speed-up over unaccelerated ProtoMol. When compared with NAMD run on the same PC, the speed-up is *6.3x*.

4.5. MD Performance Discussion

Benchmarking MD simulations is complex because of the number of parameters, configurations, and platforms that can be varied. In this subsection we present some results that, although based on post-place-and-route area and timing estimates, may give a more accurate indication of the state-of-the-art of FPGA/MD than those shown in Table 2.

Most MD simulations only perform the long-range force computation on a fraction of time-steps, with every 4th being typical. The performance of NAMD run this way improves from 3.7 to 3.2 seconds per time-step. In the two-FPGA configuration which we have described here, there is no advantage to this because the computations are already overlapped. If, however, both FPGAs are configured to compute the short-range force, and then reconfigured to compute the long-range force every fourth time-step, then the performance per time-step improves from .59 to .39 seconds. For comparison, the NAMD web site reports single PE times of 2 seconds per time-step for a similar sized model.

Another variable is the FPGA model. Moving from a VP70 to the larger VP100 (same V2 generation) allows us to double the number of pipelines per chip. Replacing the dual VP70s with a single VP100, and leaving the rest of the configuration fixed, results in slightly better performance – the computational capability is the same, but communication overhead between the FPGAs is removed. Our best estimate of performance difference between NAMD and our system running ProtoMol accelerated by a VP100 is therefore about *5x* (2 versus .38 seconds).

5. SUMMARY AND FUTURE WORK

This work extends our previous research on FPGA/MD by adding support for long-range force computation using the multigrid method. We find that the primary FPGA computational methods applicable to multigrid are an interleaved memory structure and a systolic array convolver. The speed-up obtained, and the opportunity for obvious optimizations, shows this approach to be promising.

Broader significance follows that of the multigrid method itself. Because of its efficiency, multigrid-based applications have been developed in various areas. Compared with FFT algorithms, the multigrid algorithms are more flexible with respect to boundary condition, data types, and operators. The other side of the coin is that multigrid algorithms are closely correlated with the problem being solved, such as the structure of the PDE, and so are harder to generalize.

Two features make multigrid an appropriate application for FPGAs:

- **Memory bandwidth and access flexibility**. With the on-chip block RAMs and configurable connections, FPGAs can provide significant bandwidth plus addressing logic.
- **Operations.** For the general multigrid algorithm, the operators--between continuous space and discrete grid space and between grids of different resolution--can be more complex than real number multiplication. It is easy to adapt the FPGA to new operators while still maintaining (relative) performance.

The multigrid coprocessor presented here is a reasonable prototype for multigrid algorithms: the control logic executes the V-cycle; the particle-grid converter and grid-grid convolver are well-isolated modules, ready to be adapted to new operators; and the interleaving memory can support a variety of access patterns. We are looking into applying this prototype elsewhere.

As to the entire system, using two FPGAs in such a well-partitioned application has its obvious advantages. For customers having only one FPGA chip, reconfiguration is a feasible solution. Since the particle coordinates and types are identical for both coprocessors, it will be advantageous if the chip could be partially reconfigured.

Acknowledgments. This work is supported in part by the NIH through award #RR020209-01. It was also facilitated by donations from Xilinx Corporation and from XtremeData, Inc. We would like to 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. Finally, we thank the anonymous reviewers for their helpful suggestions.

6. REFERENCES

[1] S.R. Alam, P. Agarwal, M.C Smith, J.S Vetter and D. Caliga, "Using FPGA Devices to Accelerate Biomolecular Simulations," *IEEE Computer*, vol 40, pp.66-73, 2007.

[2] N. Azizi, I. Kuon, A Egier, et al, "Reconfigurable Molecular Dynamics Simulator," *Proc. FCCM*, 2004.

[3] S. Banerjee and J.A. Board, "Efficient Charge Assignment and Back Interpolation in Multigrid Methods for Molecular Dynamics," *J. Comp. Chem.* vol. 26, 2005.

[4] A. Brandt and C.H. Venner, "Multilevel Evaluation of Integral Transforms with Asymptotically Smooth Kernels," *SIAM J Scientific Computing,* vol. 19, pp. 468-492, 1998.

[5] E.L. Briggs, D.J. Sullivan, and J. Bernholc, "Real-space multigrid-based approach to large-scale electronic structure calculations," *Phys. Rev. B*, vol. 54, pp. 14362-14375, 1996.

[6] T. Darden, D. York, and L. Pedersen, "Particle mesh Ewald: an O(N log N) method for Ewald sums in large systems," *J. Chem. Phys*, vol. 98, pp. 10089-10092, 1993.

[7] U. Essmann, L. Perera, and M.L. Berkowitz, "A smooth particle mesh Ewald method," *J. Chem. Phys*, vol. 103, pp. 8577-8593, 1995.

[8] P.L. Freddolino, A.S. Arkhipov, S.B. Larson, et al, "Molecular Dynamics Simulations of the Complete Satellite Tobacco Mosaic Virus," *Structure*, vol. 14, pp. 437-49, 2006.

[9] Y. Gu, T. VanCourt, and M.C. Herbordt, "Improved Interpolation and System Integration for FPGA-Based Molecular Dynamics Simulations," *Proc. FPL*, 2006.

[10] Y. Gu, T. VanCourt, and M. C. Herbordt, "Accelerating Molecular Dynamics Simulations with Configurable Circuits," *IEE Proc. CDT*, vol. 153 (3), 2006.

[11] J.A. Izaguirre, S.S. Hampton, and T. Matthey, "Parallel Multigrid Summation for the N-Body Problem," *J. Parallel Distrib. Comput*, vol. 65, pp. 949-962, 2005.

[12] V. Kindratenko and D. Pointer, "A case study in porting a production scientific supercomputing application to a reconfigurable computer," *Proc. FCCM*, 2006.

[13] Y. Komeiji, M. Uebayasi, R. Takata, et al, "Fast and Accurate Molecular Dynamics Simulation of a Protein Using a Special-Purpose Computer," *J. Comp. Chem*, vol. 18, pp. 1546-1563, 1997.

[14] S. Lee, *An FPGA Implementation of the Smooth Particle Mesh Ewald Reciprocal Sum Compute Engine (RSCE)*, Master's Thesis, University of Toronto, 2005.

[15] T. Matthey, "ProtoMol, An Object-Oriented Framework for Prototyping Novel Algorithms for Molecular Dynamics," *ACM Trans. Mathematical Software*, vol. 30, pp. 237-265, 2004.

[16] C. Sagui amd T. Darden, "Multigrid methods for classical molecular dynamics simulations of biomolecules," *J. Chem. Phys*, vol. 114, pp. 6578-6591, 2001.

[17] B. Sandak, "Multiscale Fast Summation of Long-Range Charge and Dipolar Interactions," *J. Comput.Chem*, vol. 22, pp. 717-731, 2001.

[18] R. Scrofano and V. Prasanna, "Preliminary Investigation of Advanced Electrostatics in Molecular Dynamics on Reconfigurable Computers," *Proc. Supercomputing*, 2006.

[19] R. Scrofano and V. Prasanna, "A Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers," *, Proc. FCCM*, 2006.

[20] R.D. Skeel, I. Tezcan, and D.J. Hardy, "Multiple Grid Methods for Classical Molecular Dynamics," *J. Comput. Chem*, vol. 23, pp. 673-684, 2002.

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

[22] T. VanCourt and M.C. Herbordt, "Application-Dependent Memory Interleaving Enables High Performance in FPGA-Based Grid Computations," *Proc. FPL*, 2006.

[23] T. VanCourt, Y. Gu, and M.C. Herbordt, "FPGA Acceleration of Rigid Molecule Interactions," *Proc. FPL*, 2004.

[24] I. Yavneh, "Why Multigrid method are so efficient," *Computing in Science & Engineering*, vol. 8, pp. 12-22, 2006.

[25] D. York and W. Yang, "The fast Fourier Poisson method for calculating Ewald sums," *J. Chem. Phys*, vol. 101, pp. 3298-3300, 1994.