Computing Models for FPGA-Based Accelerators with Case Studies in Molecular Modeling *†

Martin C. Herbordt Yongfeng Gu‡ Tom VanCourt§

Josh Model¶ Bharat Sukhwani Matt Chiu

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

Abstract: Field Programmable Gate Arrays (FPGAs) are widely being considered as accelerators for compute intensive applications – and not just as add-on components as been possible for many years, but also tightly integrated with the main processor. This interest has been motivated, in part, by reported speed-ups of $100\times$. This performance, however, have sometimes failed to translate to production applications. Moreover, the unique issues in programming FPGAs has left some application writers waiting for better tools before giving them serious consideration. In our work we have found that a critical phase of FPGA application development is finding, and mapping to, the appropriate computing model. This leads to the central point of this article: that there are several such models, and that they differ significantly from models generally used in programming. For example, whereas parallel computing models are often based on thread execution and interaction, FPGA computing can take advantage of additional degrees of freedom than available in software. This enables models based on the fundamental characteristics from which FPGAs get their capability, including highly flexible fine-grained parallelism and associative operations such as broadcast and collective response. After a review of FPGA architecture, systems, and constructs, we present five such models. The final part of the paper illustrates their use in applications involving molecular modeling.

* A version of this work has been accepted for publication in *Computing in Science and Engineering*.

† This work was supported in part by the NIH through awards #R21-RR020209-01 and and #R01-RR023168-01A1, and facilitated by donations from XtremeData, Inc., SGI, and Xilinx Corporation. Web: http://www.bu.edu/caadlab. EMail: herbordt@bu.edu.

‡ Currently with The MathWorks, Inc.

§ Currently with Altera, Inc.

¶ Currently with MIT Lincoln Laboratory

1 Introduction

For many years computational scientists could depend on continual access to ever faster computers. In the last few years, however, power concerns have caused microprocessor operating frequencies to stagnate. Moreover, while advances in process technology continue to provide ever more features per chip, these are no longer used primarily to augment individual microprocessors; rather they are commonly used to replicate the CPUs. Production chips with hundreds of CPU cores are projected to be delivered in the next several years. At the same time, however, it has become clear that replicating cores is only one of several viable strategies for developing next generation high-performance computing architectures.

Some promising alternatives are based on Field Programmable Gate Arrays (FPGAs) [16]. FPGAs are commodity integrated circuits (ICs) whose logic can be determined, or *programmed*, in the field. This is in contrast to other classes of ICs (e.g., Application Specific Integrated Circuits or ASICs) whose logic is fixed at fabrication time. The tradeoff is that FPGAs are less dense and fast than ASICs; often, however, the flexibility more than makes up for these drawbacks. Applications accelerated with FPGAs have often delivered 100-fold speed-ups per node over microprocessor-based systems. This, combined with the current ferment in computer architecture activity, has resulted in such systems moving towards the mainstream.

These architectural developments have been accompanied by a similar level of activity in hardware and software integration. On the hardware side, e.g., FPGA cards from XtremeData [36] and DRC [10, 29] plug directly into the processor sockets on the PC (or server) mother board making them co-equals with the processor chips in accessing main memory, I/O, and

the other processors. On the software side, e.g., Intel has developed QuickAssist to be "a common software framework that exposes a unified accelerator interface on top of FPGA accelerator modules.[20]" This integration support is perhaps the crucial factor in differentiating this from previous generations of accelerators [7].

Even so, few developers of high performance computing (HPC) applications have thus far "test-driven" FPGA-based systems. One reason, besides the newness of their viability, is that FPGAs are commonly viewed as hardware devices and thus require use of alien development tools. Another is that new users may disregard the hardware altogether by translating serial codes directly into FPGA configurations (using one of many available tools; see, e.g., [19] for a survey). While this results in rapid development, it may also result in unacceptable loss of performance when key features are not used to their capability.

We have found that successful development of FPGA-based HPC applications (i.e., High Performance Reconfigurable Computing or HPRC) requires a middle path: that the developer must avoid getting caught up in logic details, but at the same time should keep in mind an appropriate FPGA-oriented computing model. This leads to the central point of this paper: that there are several such models for HPRC, and that they differ significantly from models generally used in HPC programming. For example, whereas parallel computing models are often based on thread execution and interaction, FPGA computing can take advantage of additional degrees of freedom than available in software. This enables models based on the fundamental characteristics from which FPGAs get their capability, including highly flexible fine-grained parallelism and associative operations such as broadcast and collective response (see DeHon, et al., for a perspective of these issues from the point of view of design patterns [11]).

To make their presentation concrete, we give examples from our own work in molecular modeling. Methods for simulating molecules are critical: they lie at the core of computational chemistry and are central to computational biology. Applications of molecular modeling range from the practical, *e.g.,* drug design, to basic research in understanding disease processes.

Molecular modeling is compute bound. While studies conducted with a few minutes of PC time are often useful, the reality is that the computational demand is virtually insatiable: almost any molecular simulation will be improved by simulating a larger physical system for longer physical time with a more detailed model. Large-scale computational experiments run for months at a time. Even so, the gap between the largest published simulations and cell-level processes is at least ten orders of magnitude, making their acceleration all the more critical. Our case studies are as follows:

Molecular Dynamics (MD)
MD is an iterative application of Newtonian mechanics to ensembles of atoms and molecules. Time-steps alternate between force computation and motion integration. The short-range and long-range components of the non-bonded force computation dominate execution. As they have very different character, especially when mapped to FPGAs, we consider them separately. The short-range force part, especially, has been well-studied for FPGA-based systems (see, e.g., [2, 5, 18, 22, 32, 40]).

Discrete Molecular Dynamics (DMD)
Increasingly popular is MD with simplified models, such as the approximation of forces with step-wise potentials (see, e.g., [30]). This approximation results in simulations that advance by discrete event rather than time-step.

Modeling Molecular Interactions (Docking)
Finally, we examine part of the vast field of docking, viz., computations that approximate molecules as rigid structures mapped to grids (see, e.g., [23] for a docking survey).

These applications differ from one another in performance benefit, with acceleration of production codes over a single processor ranging from up to $10\times$ for MD [18], to $25\times$ for Rigid Molecule Docking [34], to over $100\times$ for DMD [28]. The applications also differ in central data type, data structure, and algorithm, and so provide a good view of the richness of the space of effective FPGA computational models.

The rest of this paper is organized as follows. After a brief introduction to computing models and FPGAs, we present five effective FPGA computing models. In the succeeding section, we illustrate their use in the applications just described. Our goal is to show applications where HPRC is likely to work, and how to recognize this. Finally, we hope that these illustrations will help new HPRC developers get started in thinking about how to map their applications to FPGAs.

2 FPGA Computation Models

2.1 Computing Models

Models are vital to many areas of computer science and engineering and range from formal models used in complexity theory and simulation to intuitive models sometimes used in computer architecture and software engineering. Here we consider the latter: by computing model we mean *an abstraction of a target machine used to facilitate application development.* This abstraction allows the developer to separate the design of an application, including the algorithms, from its coding and compilation. Put another way, a computing model lets us put into a black box the hardware capabilities and software support common to the class of target machines, and thus to concentrate on what we do not yet know how to do. Computing models in this sense are sometimes similar to *programming models*, which can mean "the conceptualization of the machine that the programmer uses [9]."

With complex applications there is often a tradeoff between programmer effort, program portability/reuseability, and program performance. The more degrees of freedom in the target architecture, the more variable the algorithm selection, and the less likely that a single computing model will allow all three to be achieved simultaneously.

A common computing model for single threaded computers is the RAM [1]. There the target machine is abstracted into a few components: input and output streams (I/O), sequential program execution, and a uniform random access memory (RAM). While the RAM model has often been criticized as being unnecessarily restrictive (see, e.g., Backus's famous paper advocating functional programming [6]), it is also the way many programmers often conceptualize single-threaded programs. Using this model simply means assuming that computing tasks are carried out in sequence, and that data are all referenced with equal cost. Programs so designed, when combined with software libraries, compilers, and good programming skills, often run efficiently and portably on most machines in this class. For high performance, more machine details, especially in the memory hierarchy, may need to be considered.

For multithreaded machines, with their additional degrees of freedom, selecting a computing model is more complex. What features can we abstract and still achieve performance and portability goals? Is a single model feasible? Under what application and hardware restrictions? The issue is utility: does the computing model enable good application design? Does the best algorithm emerge? The problem is as follows. There are a number of classes of parallel machines—shared memory, networks of PCs, networks of shared memory processors, multicore—and *the preferred mapping of a complex application may vary significantly among the classes.*

While an active topic of research, three computing models (and their combinations) span much of the space of multithreaded architecture. Following Culler, et al. [9], these are (i) shared address, (ii) message passing, and (iii) data parallel. Each is based on the threaded model. In shared address, multiple threads communicate by accessing shared locations. In message passing, multiple threads communicate by explicitly sending and receiving messages. Data parallel retains the single thread, but now operations can manipulate larger structures in possibly complex ways. The programmer's choice of computing model depends on the application and the target hardware. For example, the appropriate model for a large computer system comprised of a network of shared memory processors might be message passing among multiple shared address spaces.

2.2 Low Level FPGA Models

Historically, the computing model for FPGAs was a "bag of gates" that could be configured into logic designs. In the last few years, high-end FPGAs have come to be dominated by embedded components such as multipliers, independently addressable memories (Block RAMs or BRAMs), and high-speed I/O links. Aligned with these changes, a new low level computing model has emerged: FPGAs as a "bag of computer parts." A designer using this model would likely consider the following FPGA features when designing an application:

- Reconfigurable in milliseconds;
- Hundreds of hardwired memories and arithmetic units.
- Millions of gate-equivalents;
- Millions of communication paths, both local and global;
- Hundreds of gigabit I/O ports and tens of multi-gigabit I/O ports;
- Libraries of existing designs analogous to the various system and application libraries commonly used by programmers.

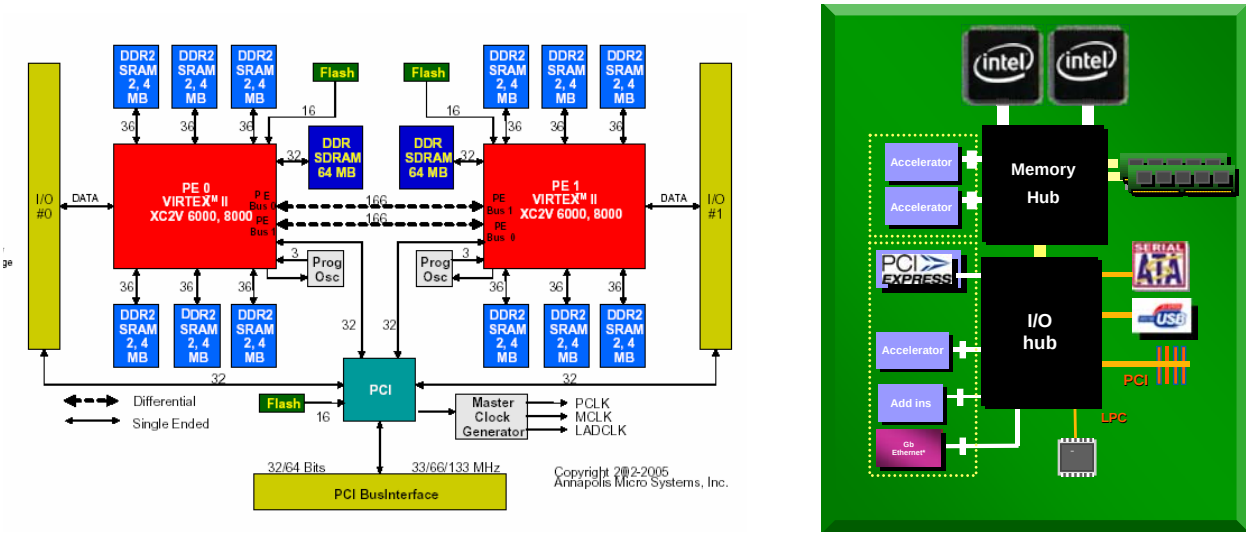

Figure 1: Left panel shows the the Annapolis Microsystems Wildstar-II coprocessor board [4]. Right panel shows an Intel view of accelerator integration into a multi-processor system [7].

As with microprocessors, making FPGAs appropriate for HPC requires added hardware support and this too is part of the low level model. A sample system is the Wildstar board from Annapolis Microsystems shown in the left panel of Figure 1. Although now dated, we found this design to be particularly well balanced. Critical are the seven independently addressable memory banks per FPGA (SRAMs and SDRAM). Since memory is managed explicitly in HPRC applications, there is no hardware caching support. Communication with the host is over an I/O bus (PCI). In the last few years, the trend with HPRC systems is towards tighter integration of the FPGA board into the host system, e.g., by making FPGA boards plug-compatible with Intel Front Side Bus slots (see right panel of Figure 1). The effect is to give FPGAs access capability to main memory (and other system components) equal to that of the microprocessors.

2.3 Why FPGAs for HPC?

A first step in defining higher level FPGA-base computing models is to consider how FPGAs get their performance for HPC. Microprocessors owe much of their tremendous success to their flexibility. This generality has a cost, however, as there is a several order-of-magnitude gap between microprocessor performance and the computational potential of the underlying substrate [31]. While fabrication costs are limiting ASICs mostly to high volume applications, FPGAs offer a compromise: they are often able to achieve much of the performance of an ASIC but are available “off the shelf.”

Practically, the enormous potential performance derivable with FPGAs comes from two sources: *parallelism* – $10,000\times$ is possible for low precision computations, and *payload per computation* – since most control is configured into the logic itself, overhead instructions (such as array indexing and loop computations) do not need to be emulated. On the other hand, there are significant, inherent, challenges. One is the low operating frequency, usually less than 1/10th that of a high-end microprocessor. Another is Amdahl’s law [3]: to achieve the speed-up factors required for user acceptance of a new technology (preferably $50\times$ [8]), close to 99% of the target application must lend itself to substantial acceleration. As a result, performance of HPC applications accelerated with FPGA coprocessors is unusually sensitive to the quality of the implementation.

2.4 FPGA Computation Basics

IThe next step in defining higher level FPGA-based computing models is to examine FPGA attributes in more detail and see how these translate into the capability just described. If FPGAs can be viewed in the first order as a configurable bag of computer parts, these parts must still be laid out in two dimensions and in finite space. This puts a premium on (i) connecting

computational blocks with short paths, (ii) taking advantage of long paths with high fan out, *viz.*, broadcast, and (iii) low precision computation.

Another issue, as with microprocessors, is support for various working set sizes and the bandwidth available to swap those working sets. There are typically several distinct levels in the HPRC memory hierarchy. Most have analogs in a conventional PC, but with somewhat different properties, especially to support fine-grained parallelism.

1. On-chip registers and look-up tables (LUTs). The FPGA substrate consists of registers and look-up tables through which logic is generated. These components can be configured into either computational logic or storage, with most designs having some mix. While all register contents can potentially be accessed every cycle, LUTs can only be accessed one or two bits at a time. For example, the Xilinx Virtex-5 LX330T has 26KB of registers and 427KB of LUT RAM; the aggregate potential bandwidth at 200MHz is 12TB/s.

2. On-chip BRAMs. High-end FPGAs have several hundred independently addressable multi-ported BRAMs. For example, the Xilinx Virtex-5 LX330T has 324 BRAMs with 1.5MB total storage and each accessible with a word size of up to 72 bits; the aggregate potential bandwidth at 200MHz is 1.2TB/s

3. On-board SRAM. High-end FPGAs have hundreds of signal pins that can be used for off-chip memory. Typical boards, however, have between two and six 32-bit independent SRAM banks, with recent boards, such as the SGI RASC having close to 100MB. As with the on-chip BRAMs, off-chip access is completely random and per cycle. The maximum possible such bandwidth for the Xilinx Virtex-5 LX330T is 49GB/s, but a figure between 1.6GB/s and 5GB/s is more common.

4. On-board DRAM. Many boards either also have DRAM, or replace SRAM completely with DRAM. Recent boards support multiple GB of DRAM. The bandwidth numbers are similar to those with SRAM, but with higher access latency.

5. Host memory. Several recent boards support high-speed access to host memory through, *e.g.,* SGI's NumaLink, Intel's Front Side Bus, and Hypertransport used by AMD systems. Bandwidth of these links ranges from 5GB/s to 20GB/s or more.

6. High-speed I/O links. FPGA applications commonly involve high-speed communication. High-end Xilinx FPGAs have up to 24 3GB/s ports.

The actual performance naturally depends on the existence of configurations that can use this bandwidth. In our own work, we frequently use the entire available BRAM bandwidth, and almost as often use most of the available off-chip bandwidth as well. In fact, we interpret this achievement for any particular application as an indication that we are on target with our mapping.

Putting these ideas together, a good FPGA computing model is one that lets us create mappings that make maximal use of one or more levels of the FPGA memory hierarchy. These mappings commonly contain large amounts of fine-grained parallelism. The processing elements (PEs) are often connected as either a few long pipelines (sometimes with 50 stages or more), or broadside with up to a few hundred very short pipelines.

Another critical factor in finding a good FPGA model is that code size translates into FPGA area. The best performance is, of course, achieved if the entire FPGA is used continuously, usually through fine-grained parallelism as just described. Conversely, if a single pipeline does not fit on the chip, performance may be poor. Poor performance can also occur with applications that have many conditional computations. For example, consider a molecular simulation where determining the potential between pairs of particles is the main computation. Moreover, let the choice of function to compute the potential depend on the particles' separation. For a microprocessor, invoking each different function probably involves little overhead. For an FPGA, however, this can be problematic: each function takes up part of the chip, *whether it is being used or not*. In the worst case, only a fraction of the FPGA is ever in use. Note that all may not be lost: it may still be possible to maintain high utilization by scheduling tasks among the functions and reconfiguring the FPGA as needed.

2.5 FPGA computational models

Concepts such as "high utilization" and "deep pipelines" are certainly valid, but not as useful as higher level models. We continue our discussion of *knowing when you've got a good mapping*, but this time from a top-down perspective. In particular, we have found that we've got a good mapping if we can fit our application into one of the following computational models. Please note that they overlap and are far from

exhaustive (see, *e.g.,* work by DeHon *et al.* [11]).

Model 1. Streaming
The streaming model is well-known across computer science and engineering and is characterized, as its name suggests, by streams of data passing through arithmetic units. Streams can source/sink at any level of the memory hierarchy. The FPGA streaming model differs from that in serial computers in the number and complexity of streams supported, and also in the seamless concatenation of computation with the I/O ports. Streaming is basic to the most popular HPRC domains: signal, image, and communication processing. It is supported explicitly by numerous FPGA languages such as Streams C [15], ASC [27], SCORE [12], and many others; by IP libraries; as well by higher level tools such as Sysgen for DSP from Xilinx.

The use of streams is obvious in the one dimensional case, for example with a signal passing through a series of filters and transforms. But with FPGAs it can also be effective to consider streaming geometrically, *i.e.,*, by considering the dimensionality of the substrate. For example, a one dimensional stream can be made long by snaking computational elements boustrophedonically through the chip. Other ways involve changing the aspect ratio, *e.g.,* with broadside sourcing/sinking through the hundreds of BRAMs; or through stream replication, which is analogous to mapping to parallel vector units. Less obvious, but still well-known, is the two dimensional streaming array used for matrix multiplication. In our own work, we use two dimensional streams for performing ungapped sequence alignment: the first dimension is used to perform initial scoring at streaming rate, while the second dimension reduces each alignment to a single maximal local score.

Model 2. Associative computing
Associative (or content addressable) computing is characterized by its basic operations: broadcast, parallel tag checking, tag-dependent conditional computing, collective response, and reduction of responses [25]. This model is basic to computing with massively parallel SIMD arrays and with neural networks. It is also used in CPU internals such as reorder buffers and translation look-aside buffers, and in router switches. While analogous software operations are ubiquitous, they do not approach the inherent performance offered by an FPGA's support of hardware broadcast. Rather than accessing data structures through $O(\log N)$ operations or complex hashing functions, in FPGAs they can often be processed in a single cycle.

Model 3. Highly parallel, possibly complex, memory access
Already mentioned is that if you can use the full bandwidth at any level of the memory hierarchy, the application is likely to be highly efficient. Added here is that on an FPGA, complex parallel memory access patterns can be configured. This problem was the object of much study in the early days of array processors (see, e.g., [26]): the objective was to enable parallel conflict-free access to slices of data, such as array rows or columns, followed by alignment of that data with the correct processing elements. With the FPGA, the programmable connections allow this capability to be tailored to the application-specific reference patterns. In [38], e.g., we route combinations of vectors in a Steiner System.

Model 4. Standard hardware structures
In a way this method is trivial – use components (sometimes as IP blocks) that have already been created. The value added here is not with there existence, but with their use: Standard data structures such as FIFOs, stacks, and priority queues are standard in software, but may have much higher relative efficiencies in hardware. The power of the model is two-fold: to use such structures when called for, and to steer the mapping towards those structures with the highest relative efficiency. One such hardware structure is perhaps the most commonly used in all of HPRC: the systolic array used for convolutions and correlations [35].

Model 5. Functional parallelism
While having function units lying idle is the bane of HPRC, functional parallelism can also be one its strengths. Again, the opportunity has to do with FPGA chip area versus compute time: functions that take a long time in software, but relatively little space in hardware are the best. For example, a simulator may require frequent generation of high-quality random numbers. Such a function takes relatively little space on an FPGA, can be fully pipelined, and can thus provide random numbers with latency completely hidden.

3 Case Studies in Molecular Modeling

3.1 Molecular Dynamics

MD forces 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:

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

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:

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

In general, the forces between all particle pairs must be computed leading to an undesirable $O(N^2)$ complexity. The common solution is to split the non-bonded forces into two parts: a fast converging short-range part, which consists of the LJ force and the nearby component of the Coulombic, and the remaining long-range part of the Coulombic (which is described in the next subsection). The complexity of the short-range force computation is then reduced to $O(N)$ by subdividing the simulated space into cells, and for each particle, processing only forces between it and other particles in its neighborhood.

The short-range compute kernel is illustrated in Figure 2. The streaming computing model is used [18]. Particle positions and types are the input, the accelerations the output. Streams source and sink in the BRAMs. The number of streams is a function of FPGA hardware resources and the computation parameters, with the usual range being from two to eight.

The wrapper around this kernel is also implemented in the FPGA: it ensures that particles in neighborhoods are available together in the BRAMs; these are swapped in the background as the computation progresses. The force computation has three parts, as shown in blue, purple, and orange, respectively. The first part checks for validity, adjusts for boundary conditions, and computes r^2. The second part computes

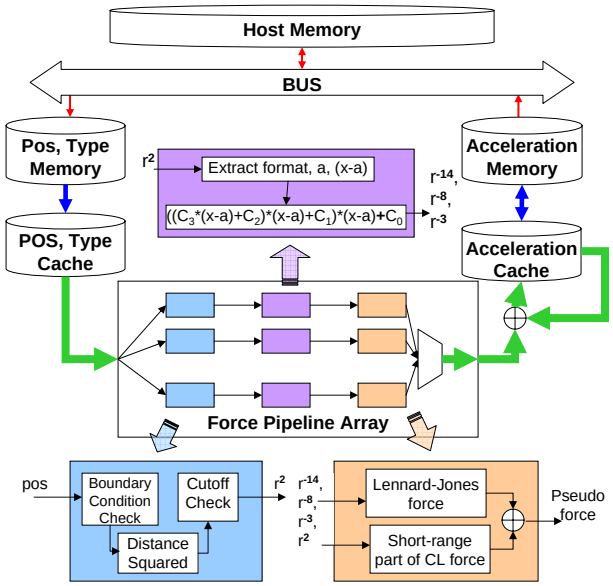

Figure 2: Pipeline for short-range force computation.

the exponentials in r. As is often done even in serial MD codes, these terms are not computed directly, but rather with table look-up followed by interpolation. Third order is shown in Figure 2. The final part combines the r^{-n} terms with the particle type coefficients to generate the force.

We find most current high-end FPGAs to be well-balanced with respect to this computation: designs simultaneously use the entire BRAM bandwidth and most of the computation fabric. If the balance is disturbed it is possible to restore it by adjusting the interpolation: this allows for a trade-off of BRAM (table size) versus computational fabric (interpolation order).

3.2 Using Multigrid for Long-Range Force Computation

Numerous methods reduce the complexity of the long-range force computation from $O(N^2)$ to $O(N \log N)$, often by using the Fast Fourier Transform (FFT). As these have so far proven difficult to map efficiently to FPGAs, however, the multigrid method may be preferable [17] (see, e.g., [33] for its application to electrostatics).

The difficulty with the Coulombic force is that it converges too slowly to restrict computation solely to proximate particle pairs. The solution begins by splitting the force into two components, a fast converging part that can be solved locally without loss of accu-

racy, and the remainder. This splitting appears to create an even more difficult problem: the remainder converges more slowly than the original. The key idea is to continue this process of splitting, each time passing the remainder on to the next coarser level, where it is again split. This continues until a level is reached where the problem size (*i.e.,* N) is small enough for the direct all-to-all solution to be efficient.

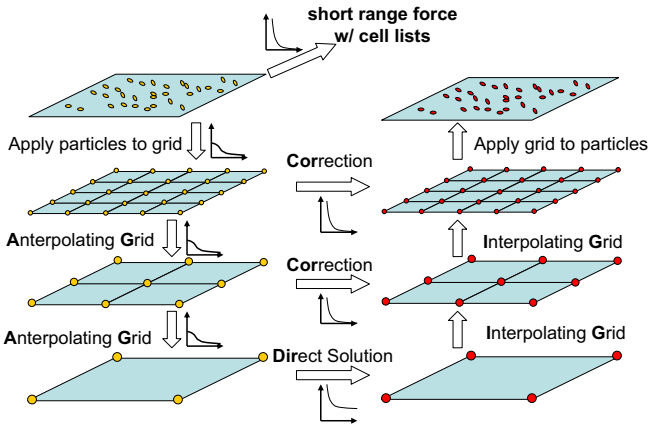

Figure 3: Schematic of the multigrid method for the Coulomb force.

The overall multigrid 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 is computed directly as shown in the previous subsection, while the long range component is applied to the finest grid. Here the force is split again, with the 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.

When mapping to an FPGA, we partition the computation into three functions: (i) applying the charges to a 3D grid, (ii) performing multigrid to convert the 3D charge density grid to a 3D potential energy grid, and (iii) applying the 3D potential to the particles to compute the forces. The two particle-grid functions are similar enough to be considered together, as are the various phases of the grid-grid computations.

The particle-grid computations in our implementation involve one real-space point and its 64 grid neighbors. For the HPRC mapping we use the third computing model: highly parallel, possibly complex,

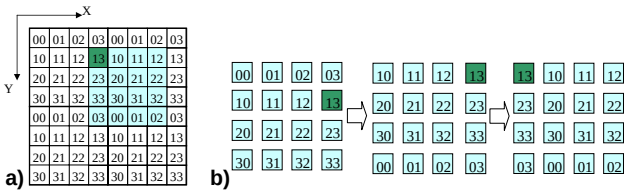

Figure 4: Shown is an example of a 2D interleaved memory reference. a) Shows the grid points (shaded) to be recovered. b) Shows the two rotations needed to get the shaded points into correct position.

memory access. We begin with judicious selection of coordinates: the real-space position can then almost immediately be converted into the BRAM indices and addresses of each of the 64 grid points. A standard initial distribution of grid points guarantees that the BRAMs will be disjoint for every position in real space. There follows the remarkable result that an entire tri-cubic interpolation can be computed in just a few cycles: data are fetched in parallel and then reduced to a single value.

In practice, getting the fetched grid points to their correct PEs requires additional routing as shown in 2D in Figure 4. There, 16 memory banks are indicated by index, each with four elements. Any 4x4 square overlaying the grid will map to independent memory banks, allowing fully parallel access, but is likely to be misaligned. For example, the green overlay would be fetched in the position shown at the beginning of 4b), and then require two rotations to get into correct alignment. The 3D routing is analogous.

For the 3D grid-grid convolutions we use the fourth computational model: use of a standard hardware structure. Here the structure is the well-known systolic array [35]. Its iterative application to build up two and three dimensional convolvers is shown in Figure 5.

For MD performance (with short-range and long-range force computations), measurements were made for a medium sized simulation (77K particles, 93 $Å^3$ box). The long-range force was computed every four cycles. Technology for both processor and accelerator were current in 8/2007. The reference code was NAMD with NAMD performance being obtained from the group web site. Depending on configuration, performance increase ranged from $5\times$ to $10\times$.

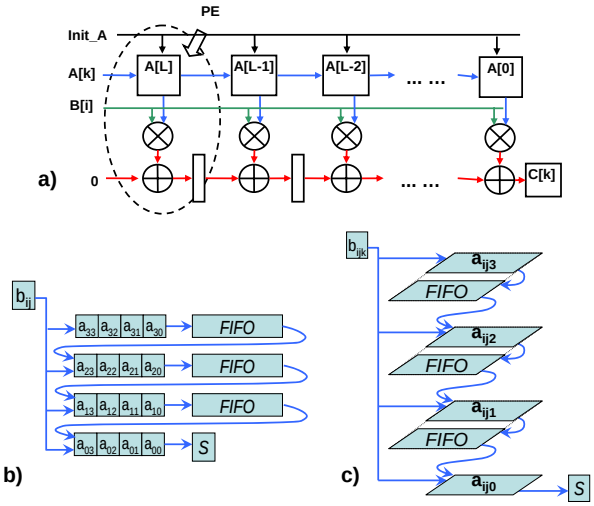

Figure 5: Shown are a) a one dimensional systolic covolver array, and its extension to b) two, and to c) three dimensions.

3.3 Discrete Event Based Molecular Dynamics (DMD)

The foundation of DMD is intuitive, hypothesis-driven, modeling based on tailoring simplified models to the physical systems of interest [13]. Using intuitive models, simulation length and time scales can exceed those of time-step driven MD by eight or more orders of magnitude [14]. Even so, not only is DMD still compute bound, causality concerns make it difficult to scale to a significant number of processors.

Discrete event simulation (DES) is sketched in the left panel of Figure 6: the primary components are the event queue, event processor, event predictor (which can also cancel previously predicted events), and system state. Parallelization of DES has generally taken one of two approaches: (i) conservative, which guarantees causal order, or (ii) optimistic, which allows some speculative violation of causality and corrects violations with rollback. Neither approach has worked well for DMD. The conservative approach, which relies on there being a "safe" window, falters because in DMD there is none. Processed events invalidate predicted events anywhere in the event queue with equal probability, and potentially anywhere in the simulated space. For similar reasons, the optimistic approach has frequent rollbacks, resulting in poor scaling.

We take a different approach based primarily on the second computational model: associative computing [28]. We process the entire simulation a single long pipeline (see right panel of Figure 6). While dozens of events are processed simultaneously, at most one event is committed per cycle. To achieve maximum throughput, the following must be done within a single cycle: (i) update the system state, (ii) process all causal event cancellations and (iii) new event insertions, and (iv) advance the event priority queue. This, in turn, uses the associative primitives of broadcast, tag check, and conditional execution. When an event is committed, the IDs of the particles it involves are broadcast to the events in the priority queue. If there is an ID match, the predicted event is cancelled. Similarly, when events are predicted, their time-stamp is broadcast throughout the priority queue. Existing events compare their time-stamps to that of the new event and it is inserted accordingly.

For simple force models and simulations of 10-20,000 particles in a 32^3 box, we have achieved throughputs of 50M events per second, while independent reference codes run at up to 200K events per second [28].

3.4 Docking Rigid Molecules

Non-covalent bonding between molecules is basic to the processes of life and to the effectiveness of pharmaceuticals. While detailed chemical models are sometimes used, such techniques are computationally exorbitant and infeasible for answering the fundamental question: at what approximate offsets and orientations could the molecules possibly interact at all? Less costly techniques are used for initial estimates of the docked pose, the relative offset and rotation that give the strongest interaction. Many applications assume rigid structure as a simplifying approximation: 3D voxel grids represent the interacting molecules and 3D correlation is used for determining the best fit [21]. This is the specific application we address here.

Our approach is based on a combination of methods four and five: using standard hardware structures, in particular the systolic convolution array, and latency hiding with functional parallelism. This results in the following three stage algorithm [39].

1. (Virtual) molecule rotation. The molecules must be tested against one another in rotated orientations. FFT versions rotate molecules explicitly. Direct correlation, however, allows us to implement the rotations by accessing elements of one of the molecules through a "rotated" indexing sequence. If these indices were stored explicitly, however, they would require exorbitant memory – we therefore generate them on-the-

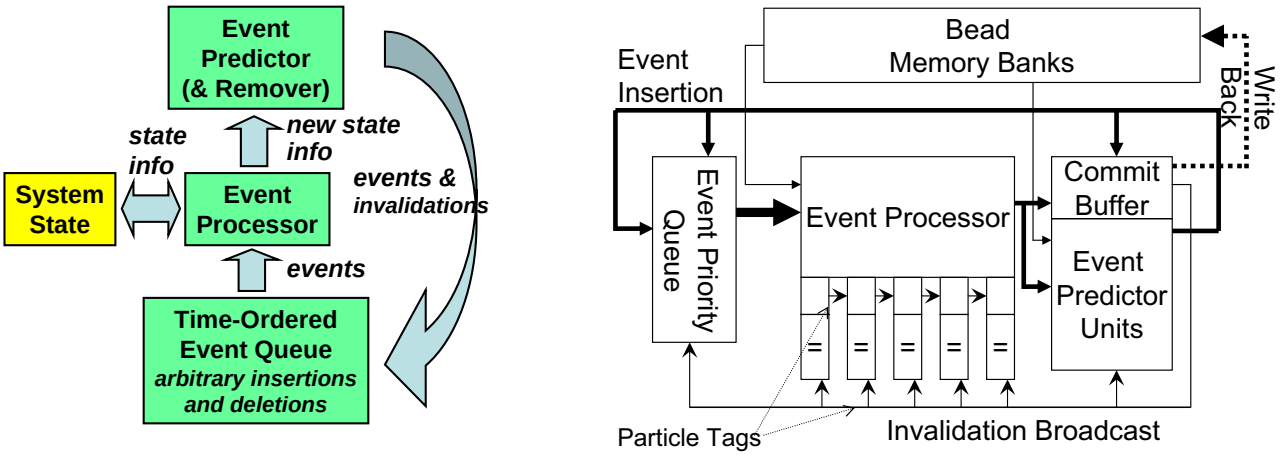

Figure 6: Shown are block diagrams of a) a generic Discrete Event Simulation and b) an FPGA mapping of Discrete Molecular Dynamics.

fly. The index generation logic (an 18 parameter function) supplies the indices just-in-time; the latency of the rotation is therefore entirely hidden. This is also a good example of how function-level parallelism can be easily implemented on an FPGA.

2. Generalized correlation. The correlation array is based on the structure used in the multigrid example (see Figure 5) generalized with respect to arbitrary scoring functions.

3. Data reduction filter. The correlation can generate millions of scores, but only a few are likely to be interesting. The challenge is to return at least a few scores from every significant local maximum (potential binding), rather than just the n highest scores. We address multiple maxima by partitioning the result grid into subblocks and collecting the highest scores reported in each.

For Docking, we obtained performance with respect to the PIPER docking code [24, 34]. Here, for typical small molecule docking (128^3 receptor, 10^3 ligand), the correlation, rotation, and filtering steps take 97% of the execution time while the initial charge assignment to the grids, which has not yet been accelerated, takes 3%. The total speed-up over a single-threaded version of the code is currently $25\times$. This could increase substantially when the charge assignment is mapped to the FPGA, using, e.g., a method similar to that used for multigrid.

4 Discussion

The goal of FPGA-based computing is to achieve substantial per-processor speed-ups. This is (necessarily) accomplished by using the inherent capability of the underlying silicon, tempered by overhead required to enable configurability. In this paper we have described methods that we believe to be generally useful in this endeavor, together with examples of how we have applied them to problems in modeling molecules.

One question is how computing models relate to programmer effort. A more basic question is which tools support which models. In our lab we use a hardware description language (VHDL) together with our own LAMP tool suite [37] which supports reusability across variations in application and target hardware. The latter unfortunately is not yet publically available. Otherwise, we believe that important characteristics are as follows: support for streams, which many HPRC languages have; support for embedding IP, again, supported by most HPRC languages; support for object-level parameterization, which often is not supported fully; and access to essential FPGA components as virtual objects, which is also not usually fully supported. Although it is trivially true that characteristics of a computational model can only be used if they can be accessed, this does not mean that good results cannot be obtained with higher level tools. Paradoxically, the more general the development tools, the more care may be needed because their effects with respect to the underlying substrate are harder to predict.

Returning to programmer effort, in our own experience, we rarely spend more than a few months before getting working systems, although more time is usually needed for test, validation, and system integration. The advantage of having a good computing model is not therefore not so much in saving effort, but rather in increasing the quality of the design. In this respect we believe the benefit is similar to that with using appropriate parallel computing models. It may not take any longer to get a working system using an inappropriate model, but achieving good performance may prove impossible.

Acknowledgments. We thank the anonymous referees for their many helpful comments and suggestions, particularly that the computing model discussion be extended and be based on available degrees of freedom.

References

[1] Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

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

[3] Amdahl, G. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings* (1967), pp. 483–485.

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

[5] Azizi, N., Kuon, I., Egier, A., Darabiha, A., and Chow, P. Reconfigurable molecular dynamics simulator. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines* (2004), pp. 197–206.

[6] Backus, J. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM 21*, 8 (1978), 613–641.

[7] Bhatt, A. Accelerating with many-cores and special purpose hardware. Keynote Talk, Field Programmable Logic and Applications, 2007.

[8] Buell, D. Reconfigurable systems. Keynote Talk, Reconfigurable Systems Summer Institute, July 2006.

[9] Culler, D., Singh, J., and Gupta, A. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan-Kaufmann, San Francisco, CA, 1999.

[10] D'Amour, M. Reconfigurable computing for acceleration in hpc. *FPGA and Structured ASIC Journal* (2008).

[11] DeHon, A., Adams, J., DeLorimier, M., Kapre, N., Matsuda, Y., Naeimi, H., Vanier, M., and Wrighton, M. Design patterns for reconfigurable computing. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines* (2004).

[12] DeHon, A., Markovsky, Y., Caspi, E., Chu, M., Huang, R., Perissakis, S., Pozzi, L., Yeh, J., and Wawrzynek, J. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems 30* (2006), 334–354.

[13] Ding, F., and Dokholyan, N. Simple but predictive protein models. *Trends in Biotechnology 3*, 9 (2005), 450–455.

[14] Dokholyan, N. Studies of folding and misfolding using simplified models. *Current Opinion in Structural Biology 16* (2006), 79–85.

[15] Frigo, J., Gokhale, M., and Lavenier, D. Evaluation of the Streams-C C-to-FPGA compiler: An applications perspective. In *Proceedings of the ACM Symposium on Field Programmable Gate Arrays* (2001).

[16] Gokhale, M., Rickett, C., Tripp, J., Hsu, C., and Scrofano, R. Promises and pitfalls of reconfigurable supercomputing. In *Proceedings of the 2006 Conference on the Engineering of Reconfigurable Systems and Algorithms* (2006), pp. 11–20.

[17] Gu, Y., and Herbordt, M. FPGA-based multigrid computations for molecular dynamics simulations. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines* (2007), pp. 117–126.

[18] Gu, Y., VanCourt, T., and Herbordt, M. Explicit design of FPGA-based coprocessors for short-range force computation in molecular dynamics simulations. *Parallel Computing 34*, 4-5 (2008), 261–271.

[19] Holland, B., Vacas, M., Aggarwal, V., DeVille, R., Troxel, I., and George, A. Survey of C-based application mapping tools for reconfigurable computing. In *Proceedings of the 8th International Conference on Military and Aerospace Programmable Logic Devices* (2005).

[20] Intel Corporation. *Platform-Level Services for Accelerators: Intel QuickAssist Technology Accelerator Abstraction Layer (AAL)*, 2007.

[21] Katchalski-Katzir, E., Shariv, I., Eisenstein, M., Friesem, A., Aflalo, C., and Vakser, I. Molecular surface recognition: Determination of geometric fit between proteins and their ligands by correlation techniques. *Proc. Nat. Acad. Sci. 89* (1992), 2195–2199.

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

[23] Kitchen, D., Decornez, H., Furr, J., and Bajorath, J. Docking and scoring in virtual screening for drug discovery: Methods and applications. *Nature Reviews - Drug Discovery 3* (2004), 935–949.

[24] Kozakov, D., Brenke, R., Comeau, S., and Vajda, S. PIPER: an FFT-based protein docking program with pairwise potentials. *Proteins: Structure, Function, and Genetics 65* (2006), 392–406.

[25] Krikelis, A., and Weems, C., Eds. *Associative Processing and Processors*. IEEE Computer Society Press, 1997.

[26] Lawrie, D. H. Access and alignment of data in an array processor. *IEEE Transactions on Computers C-24*, 12 (1975), 1145–1155.

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

[28] Model, J., and Herbordt, M. Discrete event simulation of molecular dynamics with configurable logic. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications* (2007), pp. 151–158.

[29] Morris, K. COTS Supercomputing. *FPGA and Structured ASIC Journal* (2007).

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

[31] Roza, E. Systems-on-chip: What are the limits? *Electronics and Communication Engineering Journal 12*, 2 (2001), 249–255.

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

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

[34] Sukhwani, B., and Herbordt, M. Acceleration of a production rigid molecule docking code. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications* (2008), p. TBD.

[35] Swartzlander, E. *Systolic Signal Processing Systems*. Marcel Dekker, Inc., 1987.

[36] Urban, K. In-socket accelerators: When to use them. *HPC Wire June 5, 2008* (2008), http://www.hpcwire.com.

[37] VanCourt, T. *LAMP: Tools for Creating Application-Specific FPGA Coprocessors*. PhD thesis, Department of Electrical and Computer Engineering, Boston University, 2006.

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

[39] VanCourt, T., and Herbordt, M. Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing v2006* (2006), 1–10.

[40] Villareal, J., Cortes, J., and Najjar, W. Compiled code acceleration of NAMD on FPGAs. In *Proceedings of the Reconfigurable Systems Summer Institute* (2007).