Computing Models for FPGA-Based Accelerators

Field-programmable gate arrays are widely considered as accelerators for compute-intensive applications. A critical phase of FPGA application development is finding and mapping to the appropriate computing model. FPGA computing enables models with highly flexible fine-grained parallelism and associative operations such as broadcast and collective response. Several case studies demonstrate the effectiveness of using these computing models in developing FPGA applications for molecular modeling.

For many years, computational scientists could depend on continual access to ever faster computers. In the past 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're commonly used to replicate the CPUs. Production chips with hundreds of CPU cores are projected to be delivered in the next several years. Replicating cores, however, is only one of several viable strategies for developing next-generation high-performance computing (HPC) architectures.

Some promising alternatives use field-programmable gate arrays.[1] FPGAs are commodity integrated circuits whose logic can be determined, or programmed, in the field. This is in contrast to other classes of ICs (such as application-specific ICs, or ASICs), whose logic is fixed at fabrication time. FPGAs are less dense and slower than ASICs, but their flexibility often more than makes up for these drawbacks. Applications accelerated with FPGAs often deliver 100-fold speedups per node over microprocessor-based systems. This, combined with the current ferment in computer architecture activity, has resulted in such systems moving toward the mainstream, with the largest vendors providing integration support.

Even so, few developers of HPC applications have thus far test-driven FPGA-based systems. Developers commonly view FPGAs as hardware devices requiring the use of alien development tools. New users might also disregard the hardware altogether by translating serial codes directly into FPGA configurations (using one of many available tools). Although this results in rapid development, it can also result in unacceptable performance loss.

Successful development of FPGA-based HPC applications (that is, high-performance reconfigurable computing, or HPRC) requires a middle path. Developers must avoid getting caught up in logic details while keeping in mind an appropriate FPGA-oriented computing model. Several such models for HPRC exist, but 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

1521-9615/08/$25.00 © 2008 IEEE
Copublished by the IEEE CS and the AIP

Martin C. Herbordt, Yongfeng Gu, Tom VanCourt, Josh Model, Bharat Sukhwani, and Matt Chiu
Boston University

can exploit more degrees of freedom than are 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. Andre DeHon and his colleagues discuss these issues from a design pattern viewpoint.[2] To make their presentation concrete, we describe several case studies from our work in molecular modeling.

FPGA 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 lets the developer separate an application's design, including the algorithms, from its coding and compilation. In other words, 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 concentrate on what we don't yet know how to do. In this sense, computing models are sometimes similar to programming models, which can mean "the conceptualization of the machine that the programmer uses."[3]

With complex applications, there's often a trade-off between programmer effort, program portability and reusability, 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 let application developers achieve all three simultaneously.

A common computing model for single-threaded computers is the RAM.[4] 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). Although the RAM model has often been criticized as being unnecessarily restrictive (see, for example, John Backus's famous paper advocating functional programming[5]), it's also how many programmers often conceptualize single-threaded programs. Using this model simply means assuming that the program performs computing tasks in sequence and that all data references have 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, programmers might need to consider more machine details, especially in the memory hierarchy.

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? What application and hardware restrictions must we work under? The issue is utility: does the computing model enable good application design? Does the best algorithm emerge? Several classes of parallel machines exist—shared memory, networks of PCs, networks of shared-memory processors, and multicore—and the preferred mapping of a complex application might vary significantly among the classes.

Three computing models (and their combinations) span much of the multithreaded architecture space. According to David Culler and his colleagues,[3] these models, each based on the threaded model, are

- *shared address*, in which multiple threads communicate by accessing shared locations;
- *message passing*, in which multiple threads communicate by explicitly sending and receiving messages; and
- *data parallel*, which retains the single thread but lets operations manipulate larger structures in possibly complex ways.

The programmer's choice of computing model depends on the application and 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.

Low-Level FPGA Models

Historically, the computing model for FPGAs was a "bag of gates" that designers could configure into logic designs. In the past few years, embedded components such as multipliers, independently addressable memories (block RAMs, or BRAMs), and high-speed I/O links have begun to dominate high-end FPGAs. 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 multigigabit I/O ports; and
- libraries of existing designs analogous to the various system and application libraries commonly used by programmers.

As with microprocessors, making FPGAs appropriate for HPC requires added support. This too is part of the low-level model. A sample system is Annapolis Microsystems' Wildstar board. Although now dated, this design is particularly well balanced. The design's seven independently addressable memory banks per FPGA (SRAMs and SDRAM) are critical (see Figure 1a). Because HPRC applications manage memory explicitly, they offer no hardware caching support. Communication with the host takes place over an I/O bus (PCI).

In the past few years, HPRC systems have tended toward tighter integration of the FPGA board into the host system—for example, by making FPGA boards plug-compatible with Intel front-side bus slots (see Figure 1b). The effect is to give FPGAs access to main memory (and other system components) equal to that of the microprocessors.

Why FPGAs for HPC?

A first step in defining higher-level FPGA-based 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, because a several orders-of-magnitude gap exists between microprocessor performance and the computational potential of the underlying substrate.[6] Whereas fabrication costs limit ASICs mostly to high-volume applications, FPGAs offer a compromise. They can often achieve much of an ASIC's performance but are available off the shelf.

Practically, the enormous potential performance derivable with FPGAs comes from two sources:

- *Parallelism*. A factor of 10,000× parallelism is possible for low-precision computations.
- *Payload per computation*. Because most control is configured into the logic itself, designers don't need to emulate overhead instructions (such as array indexing and loop computations).

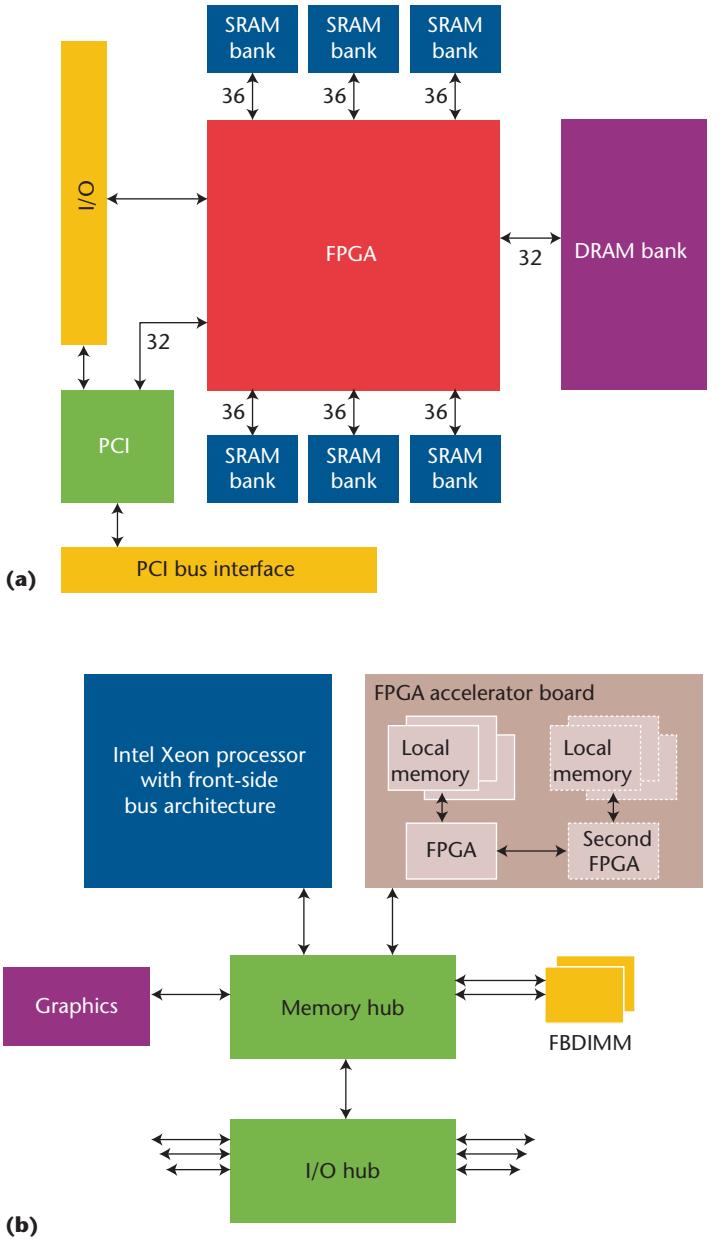

Figure 1. Field-programmable gate arrays in high-performance computing. (a) In this coprocessor board, the seven independently addressable memory banks per FPGA are critical. (b) The diagram shows an Intel view of accelerator integration into a multiprocessor system.

On the other hand, significant inherent challenges exist. One is the low operating frequency, usually less than 1/10th that of a high-end microprocessor. Another is Amdahl's law: to achieve the speedup factors required for user acceptance of a new technology (preferably 50×),[7] almost 99 percent of the target application must lend itself

to substantial acceleration.[8] As a result, the performance of HPC applications accelerated with FPGA coprocessors is unusually sensitive to the implementation's quality.

FPGA Computation Basics

The next step in defining higher-level FPGA-based computing models is to examine FPGA attributes for how they translate into the capability just described. If we view FPGAs as a configurable bag of computer parts, we must lay these parts out in two dimensions and in finite space. This puts a premium on connecting computational blocks with short paths, exploiting long paths with high fan out (namely, broadcast), and low-precision computation. As with microprocessors, HPRC systems must support various working set sizes and the bandwidth available to swap those working sets. The HPRC memory hierarchy typically has several distinct levels. Most have analogs in conventional PCs, but with somewhat different properties, especially with regard to supporting fine-grained parallelism:

- *On-chip registers and lookup tables.* The FPGA substrate consists of registers and LUTs through which logic is generated. These components can be configured into computational logic or storage, with most designs having a mix. Although 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 26 Kbytes of registers and 427 Kbytes of LUT RAM; the aggregate potential bandwidth at 200 MHz is 12 terabits per second (Tbps).
- *On-chip BRAMs.* High-end FPGAs have several hundred independently addressable multiported BRAMs. For example, the Xilinx Virtex-5 LX330T has 324 BRAMs with 1.5 Mbytes total storage and each accessible with a word size of up to 72 bits; the aggregate potential bandwidth at 200 MHz is 1.2 Tbps.
- *Onboard 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; recent boards, such as the SGI RASC, have almost 100 Mbytes. 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 49 gigabits per second, but between 1.6 Gbps and 5 Gbps is more common.
- *Onboard DRAM.* Many boards either have both SRAM and DRAM or replace SRAM completely with DRAM. Recent boards support multiple Gbytes of DRAM. The bandwidth is similar to that with SRAM but has higher access latency.
- *Host memory.* Several recent boards support high-speed access to host memory through, for example, SGI's NumaLink, Intel's Front Side Bus, and Hypertransport, used by AMD systems. Bandwidth of these links ranges from 5 to 20 Gbps or more.
- *High-speed I/O links.* FPGA applications often involve high-speed communication. High-end Xilinx FPGAs have up to 24 3-Gbps ports.

The more degrees of freedom in the target architecture, the less likely that a single computing model will let application developers achieve all three simultaneously.

The actual performance naturally depends on the existence of configurations that can use this bandwidth. In our 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're on target with our mapping.

Putting these ideas together, we can say that a good FPGA computing model 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 are often connected as either a few long pipelines (sometimes with 50 stages or more) or broadside with up to a few hundred short pipelines.

Another critical factor of a good FPGA model is that code size translates into FPGA area. We achieve the best performance, of course, if we use the entire FPGA, usually through fine-grained parallelism. Conversely, if a single pipeline doesn't fit on the chip, performance might be poor. Poor performance can also occur with applications that have many conditional computations. For example, consider a molecular simulation in which the main computation is determining the potential between pairs of particles. 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 because each function takes up part of the chip, whether it's being used or not. In the worst case, only a fraction of the FPGA is ever in use. All might not be lost, however: designers might still be able to maintain high utilization by scheduling tasks among the functions and reconfiguring the FPGA as needed.

FPGA Computing Models

Concepts such as "high utilization" and "deep pipelines" are certainly critical, but are still far removed from the application conceptualization with which most programmers begin the design process. We found several computing models to be useful during this initial stage. That is, we're on our way to a plausible design if we can map our application into one of these models. Please note that the models overlap and are far from exhaustive.[2]

Streaming. The streaming model is well-known in computer science and engineering. It's 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 the serial computer model in the number and complexity of streams supported and the seamless concatenation of computation with the I/O ports. Streaming is basic to the most popular HPRC domains: signal, image, and communication processing. Many FPGA languages, such as Streams C,[9] ASC,[10] and Score[11]; IP libraries; and higher-level tools such as Xilinx's Sysgen for digital signal processing explicitly support streaming.

The use of streams is obvious in the 1D case—for example, when a signal passes through a series of filters and transforms. But with FPGAs, streaming geometrically—that is, considering the substrate's dimensionality—can also be effective. For example, we can make a 1D stream long by snaking computing elements through the chip. Other ways involve changing the aspect ratio (for example, with broadside sourcing/sinking through the hundreds of BRAMs) or using stream replication, which is analogous to mapping to parallel vector units. Less obvious, but still well-known, is the 2D streaming array used for matrix multiplication. In our work, we use 2D streams for performing ungapped sequence alignment. We use the first dimension to perform initial scoring at streaming rate and the second dimension to reduce each alignment to a single maximal local score.

Associative computing. Associative (or content-addressable) computing is characterized by its basic operations:[12]

- broadcast,
- parallel tag checking,
- tag-dependent conditional computing,
- collective response, and
- reduction of responses.

This model is basic to computing with massively parallel SIMD arrays and with artificial neural networks.

CPU internals, such as reorder buffers and translation look-aside buffers, also use this model. Although analogous software operations are ubiquitous, they don't approach the inherent performance offered by an FPGA's support of hardware broadcast and reduction. Instead of accessing data structures through $O(\log N)$ operations or complex hashing functions, FPGAs can often process associative data structures in a single cycle.

Highly parallel, possibly complex, memory access. We already mentioned that using the full bandwidth at any level of the memory hierarchy will likely make the application highly efficient. In addition, on an FPGA, you can configure complex parallel memory-access patterns. Much study in the early days of array processors focused on this problem.[13] The objective was to enable parallel conflict-free access to slices of data, such as array rows or columns, and then align that data with the correct processing elements. With the FPGA, the programmable connections let designers tailor this capability to application-specific reference patterns.[14]

Standard hardware structures. In a way, this model is trivial—it uses preexisting components. The value added here is with their use. Standard data structures such as FIFOs, stacks, and priority queues are common in software but often have much higher relative efficiencies in hardware. The model's power is twofold:

- to use such structures when called for, and
- to steer the mapping toward the structures with the highest relative efficiency.

One such hardware structure—the systolic array used for convolutions[15] and correlations—is perhaps the most commonly used in all of HPRC.

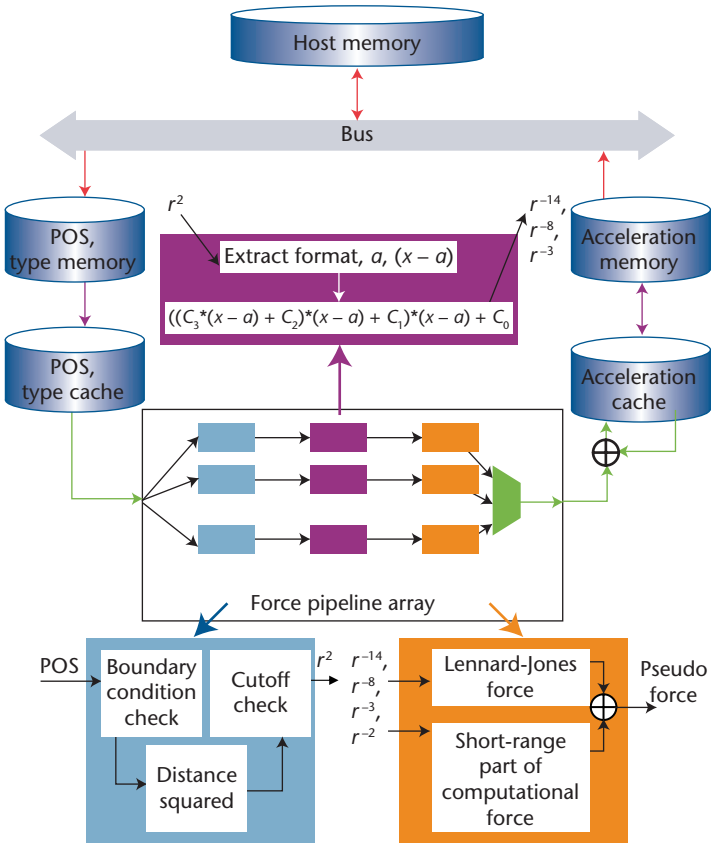

Figure 2. Pipeline for short-range force computation. The three-part force computation includes components for checking validity, adjusting for boundary conditions, and computing r^2 (blue); computing the exponentials in r (purple); and combining these terms with the particle type coefficients to generate the force (orange).

Functional parallelism. Although having function units lying idle is the bane of HPRC, functional parallelism can also be one of 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 best. For example, a simulator might 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 the latency completely hidden.

Case Studies in Molecular Modeling

Methods for simulating molecules lie at the core of computational chemistry and are central to computational biology. Applications of molecular modeling range from the practical (for example, drug design) to basic research in understanding disease processes. Molecular modeling is also compute bound. Whereas studies conducted in a few minutes on a small desktop system are often useful, the reality is that the computing demand is virtually insatiable. Simulating a larger physical system for a longer physical time with a more detailed model will improve almost any molecular simulation. 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 10 orders of magnitude, making their acceleration all the more critical. We describe several case studies that demonstrate the effectiveness of FPGA-based accelerators in molecular modeling.

Short-Range Force Computation

Molecular dynamics is an iterative application of Newtonian mechanics to ensembles of atoms and molecules. Time steps alternate between force computation and motion integration. The nonbonded force computation's short- and long-range components dominate execution. Because these components have different characters, especially when mapped to FPGAs, we consider them separately. The short-range force part, especially, has been well-studied for FPGA-based systems.[16–20]

Molecular dynamics forces might 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:

$$\mathbf{F}^{\text{total}} = F^{\text{bond}} + F^{\text{angle}} + F^{\text{torsion}} + F^{\text{H Bond}} + F^{\text{nonbonded}}. \quad (1)$$

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.

We express the LJ force for particle i as

$$\mathbf{F}_i^{LJ} = \sum_{j \neq 1} \frac{\in_{ab}}{\sigma_{ab}^2} \left\{ 12 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{14} - 6 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{8} \right\} r_{ji}, \quad (2)$$

where the $\in ab$ and σ_{ab} are parameters related to the particle types—that is, particle i is type a and particle j is type b.

We express the Coulombic force as

$$F_i^C = q_i \sum_{j \neq 1} \left(\frac{q_j}{|r_{ji}|^3} \right) r_{ji}. \quad (3)$$

In general, we must compute the forces between all particle pairs, leading to an undesirable $O(N^2)$ complexity. The common solution is to split the nonbonded forces into two parts:

- a fast-converging short-range part consisting of the LJ force and the nearby Coulombic component, and
- the remaining long-range Coulombic part (which we describe later).

This solution reduces the short-range force computation's complexity to $O(N)$ by only processing forces among nearby particles.

Figure 2 shows the short-range computation kernel, using the streaming computational model.[21] Particle positions and types are the input, and accelerations are 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.

We also implement the wrapper around this kernel in the FPGA. The wrapper ensures that particles in neighborhoods are available together in the BRAMs. The wrapper logic swaps these neighborhoods in the background as the computation progresses. The force computation has three parts:

- Part 1 (shaded blue in Figure 2) checks for validity, adjusts for boundary conditions, and computes r^2.
- Part 2 (purple) computes the exponentials in r. As is typical even in serial molecular dynamics codes, we don't compute these terms directly, but rather with table lookup followed by interpolation. Figure 2 shows third-order interpolation.
- Part 3 (orange) combines the r^{-n} terms with the particle type coefficients to generate the force.

Most current high-end FPGAs are 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, we can restore it by adjusting the interpolation. This allows for a trade-off of BRAM (table size) and computational fabric (interpolation order).

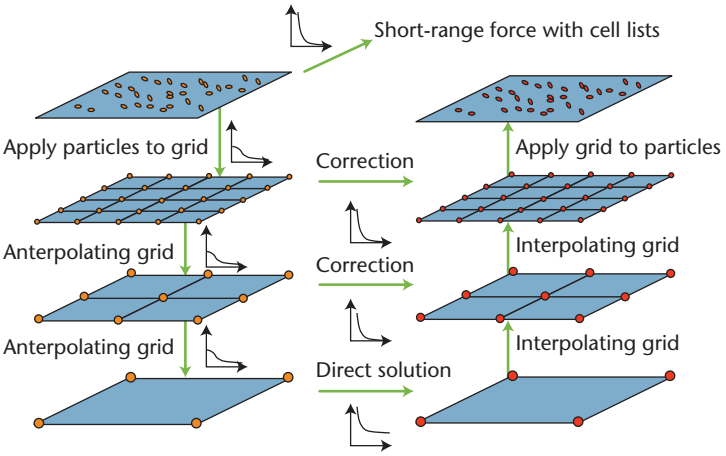

Figure 3. Schematic of the multigrid method for the Coulomb force. The left side shows the successive splitting, the lowest level the direct solution, and the right side the successive mergers with the previously computed corrections.

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 using the fast Fourier transform (FFT). Because these have so far proven difficult to map efficiently to FPGAs, however, the multigrid method might be preferable[22] (a description of its application to electrostatics is available elsewhere[23]).

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 accuracy, 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 splitting process, each time passing the remainder to the next coarser level, where it's split again. This continues until a level is reached where the problem size (N) is small enough for the direct all-to-all solution to be efficient.

Figure 3 shows the schematic of the overall multigrid algorithm. Starting at the upper left, the algorithm partitions the per-particle potentials into short- and long-range components. It computes the short-range components directly, as we described earlier, and applies the long-range component to the finest grid. Here, it splits the force again, with the high-frequency component solved directly and the low-frequency passed on to the next coarser grid. This continues until it reaches the coarsest

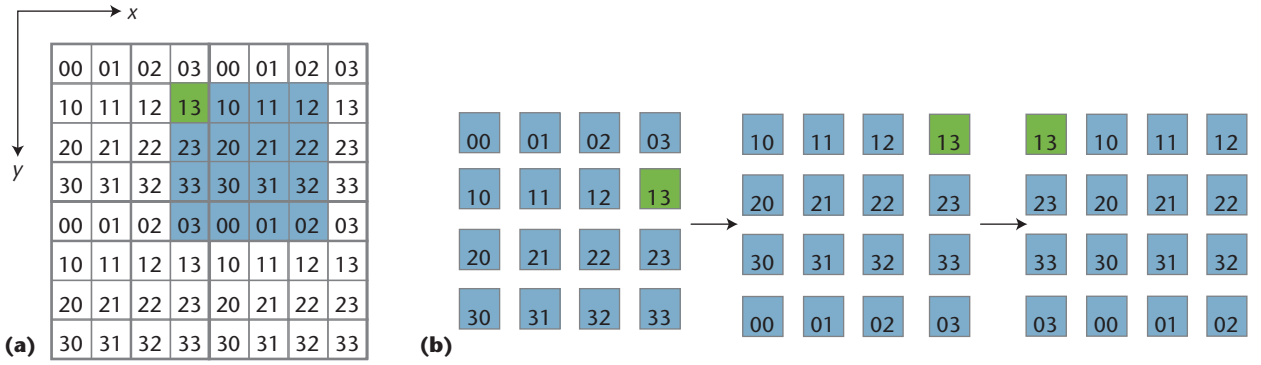

Figure 4. An example of a 2D interleaved memory reference. The diagrams show (a) the grid points (shaded) to be recovered, and (b) the two rotations needed to get the shaded points into correct position.

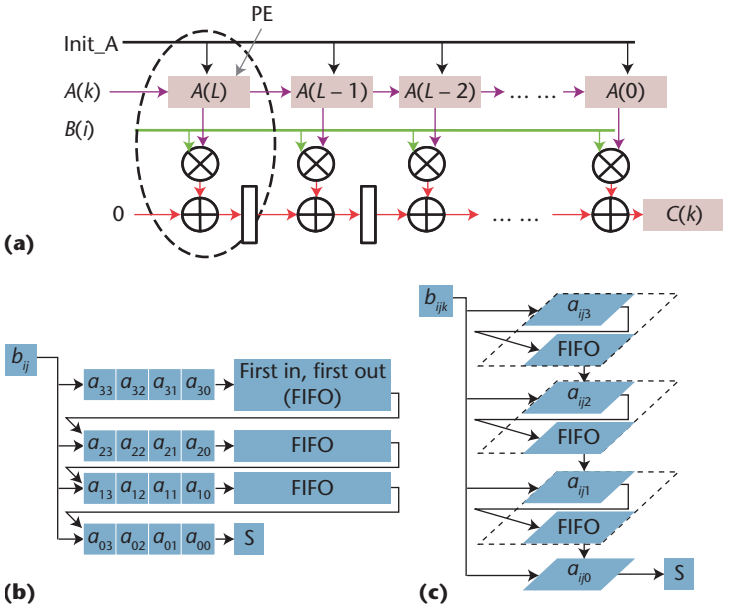

Figure 5. Iterative application of the systolic array. We apply (a) a 1D systolic convolver array and its extension to (b) 2D and (c) 3D.

level, where it solves the problem directly. We then successively combine this direct solution with the previously computed finer solutions (corrections) until we reach the finest grid. Here, we apply the forces directly to the particles.

When mapping to an FPGA, we partition the computation into three functions:

- applying the charges to a 3D grid,
- performing multigrid to convert the 3D charge density grid to a 3D potential energy grid, and
- 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, memory access. We begin with judicious selection of coordinates. We can then almost immediately convert the real-space position 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 tricubic interpolation can be computed in just a few cycles: data are fetched in parallel and reduced to a single value.

In practice, getting the fetched grid points to their correct processing elements requires additional routing, as Figure 4 shows in 2D. In Figure 4a, an index indicates 16 memory banks, each with four elements. Any 4 × 4 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 Figure 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.[15] Figure 5 shows its iterative application to build up 2D and 3D convolvers.

Discrete Event-Based Molecular Dynamics

Increasingly popular is molecular dynamics with simplified models, such as the approximation of

forces with step-wise potentials.[24] This approximation results in simulations that advance by discrete event rather than time step.

Discrete event-based molecular dynamics (DMD) is an intuitive, hypothesis-driven modeling method based on tailoring simplified models to the physical systems of interest.[25] Using intuitive models, simulation length and time scales can exceed those of time-step-driven molecular dynamics by eight or more orders of magnitude.[26] Even so, not only is DMD still compute bound, causality concerns make it difficult to scale to a significant number of processors.

Figure 6a gives an overview of discrete event simulation. The primary DES components are the event queue, event processor, event predictor (which can also cancel previously predicted events), and system state. DES parallelization generally follows one of two approaches:

- *conservative*, which guarantees causal order, or
- *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 DMD has no such window. 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 associative computing model.[27] We process the entire simulation as a single long pipeline (see Figure 6b). Although dozens of events are processed simultaneously, at most one event is committed per cycle. To achieve maximum throughput, we must accomplish several tasks within a single cycle:

- update the system state,
- process all causal event cancellations,
- process new event insertions, and
- advance the event-priority queue.

This process, in turn, uses the associative primitives of broadcast, tag check, and conditional execution. When the event-processing pipeline commits an event, it broadcasts the relevant particles' IDs to the events in the priority queue. If an ID match exists, the predicted event is cancelled. Similarly, when the pipeline predicts events, it broadcasts their time stamps throughout the priority queue. Existing events compare their time stamps to that of the new event, and the event-processing pipeline inserts it accordingly.

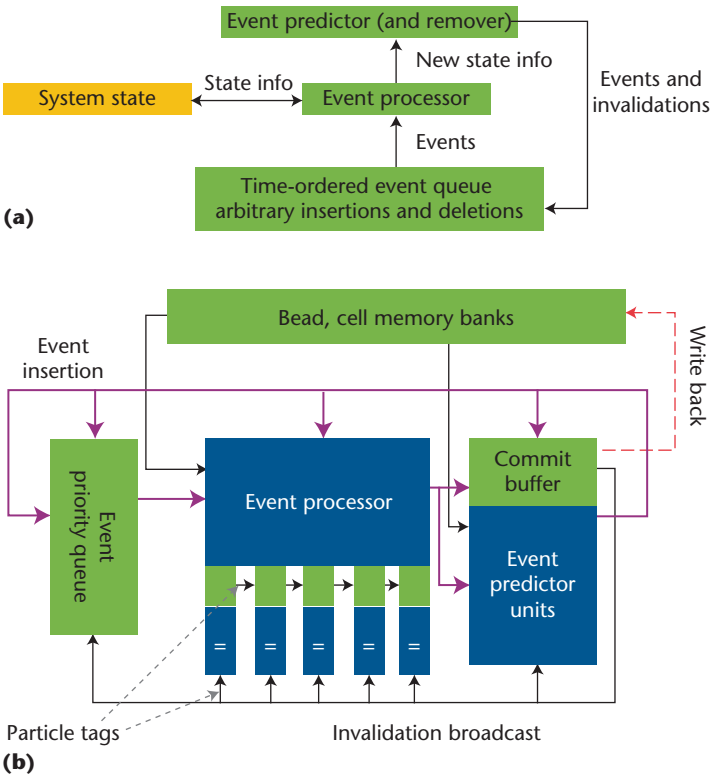

Figure 6. Event-based molecular dynamics. The block diagrams show (a) a generic discrete event simulation and (b) an FPGA mapping of discrete molecular dynamics.

Docking Rigid Molecules

Another case study involved applications of docking,[28] in which computations approximate molecules as rigid structures mapped to grids. Docking applications 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.

Noncovalent bonding between molecules, or *docking*, is basic to the processes of life and the effectiveness of pharmaceuticals. Although researchers sometimes use detailed chemical models, such techniques are computationally exorbitant and infeasible for answering the first question: at what approximate offsets and orientations could the molecules possibly interact at all? Many docking applications use less costly techniques to initially estimate the docked pose and the relative offset and rotation that give the strongest interaction. They might assume rigid structure as a simplifying approximation. Then 3D voxel grids repre-

sent the interacting molecules and 3D correlation helps determine the best fit.[29]

We base our approach on a combination of standard hardware structures (in particular, the systolic convolution array) and latency hiding with functional parallelism. This gives us a three-stage algorithm.[30]

(Virtual) molecule rotation. We test the molecules against one another in rotated orientations. FFT versions rotate molecules explicitly, but direct correlation lets us implement the rotations by accessing elements of one of the molecules through a rotated indexing sequence. Because explicitly storing these indices would require exorbitant memory, we generate them on the fly. The index-generation logic (an 18-parameter function) supplies the indices just in time, hiding the rotation's latency entirely. This is also a good example of how we can easily implement function-level parallelism on an FPGA.

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

Data reduction filter. The correlation can generate millions of scores but only a few will 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.

An open 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 (VHSIC Hardware Description Language [VHDL]) together with our own LAMP tool suite,[31] which supports reusability across variations in application and target hardware. The latter, unfortunately, isn't yet publicly available. Otherwise, we believe that important characteristics include

- support for streams, which many HPRC languages have;
- support for embedding IP, again, supported by most HPRC languages;
- support for object-level parameterization, which is rarely fully supported; and
- access to essential FPGA components as virtual objects, which also is rarely fully supported.

Although you can use a computational model's characteristics only if you can access them, you can still get good results with higher-level tools. Paradoxically, the more general the development tools, the more care might 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 therefore not so much in saving effort, but rather in increasing design quality. In this respect, the benefit is similar to that with using appropriate parallel computing models. It might not take any longer to get a working system using an inappropriate model, but achieving good performance might prove impossible. CISE

Acknowledgments

We thank the anonymous referees for their many helpful comments and suggestions. This work was supported in part by the US National Institutes of Health through awards R01 RR023168-01 and R21 RR020209-1, and facilitated by donations from Xilinx Corporation, SGI, and XTremeData.

References

1. M. Gokhale et al., "Promises and Pitfalls of Reconfigurable Supercomputing," *Proc. 2006 Conf. Eng. Reconfigurable Systems and Algorithms*, CSREA Press, 2006, pp. 11–20.
2. A. DeHon et al., "Design Patterns for Reconfigurable Computing," *Proc. IEEE Symp. Field Programmable Custom Computing Machines*, IEEE CS Press, 2004, pp. 13–23.
3. D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaugmann, 1999.
4. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
5. J. Backus, "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM*, vol. 21, no. 8, 1978, pp. 613–641.
6. E. Roza, "Systems-on-Chip: What Are the Limits?" *Electronics and Comm. Eng. J.*, vol. 12, no. 2, 2001, pp. 249–255.
7. D. Buell, "Reconfigurable Systems," keynote talk, *Reconfigurable Systems Summer Institute*, July 2006; http://gladiator.ncsa.uiuc.edu/PDFs/rssi06/presentations/00_Duncan_Buell.pdf.
8. G. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," *Proc. Am. Federation of Information Processing Societies (AFIPS) Conf.*, AFIPS Press, 1967, pp. 483–485.
9. J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspec-

tive," *Proc. Field Programmable Gate Arrays*, IEEE CS Press, 2001, pp. 134–140.

10. O. Mencer, "ASC: A Stream Compiler for Computing with FPGAs," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, 2006, pp. 1603–1617.
11. A. DeHon et al., "Stream Computations Organized for Reconfigurable Execution," *Microprocessors and Microsystems*, vol. 30, no. 6, 2006, pp. 334–354.
12. A. Krikelis and C. Weems, "Associative Processing and Processors," *Computer*, vol. 27, no. 11, 1994, pp. 12–17.
13. D.H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Computers*, vol. 24, no. 12, 1975, pp. 1145–1155.
14. T. VanCourt and M. Herbordt, "Application-Dependent Memory Interleaving Enables High Performance in FPGA-Based Grid Computations," *Proc. IEEE Conf. Field Programmable Logic and Applications*, IEEE CS Press, 2006, pp. 395–401.
15. E. Swartzlander, *Systolic Signal Processing Systems*, Marcel Drekker, 1987.
16. S. Alam et al., "Using FPGA Devices to Accelerate Biomolecular Simulations," *Computer*, vol. 40, no. 3, 2007, pp. 66–73.
17. N. Azizi et al., "Reconfigurable Molecular Dynamics Simulator," *Proc. IEEE Symp. Field Programmable Custom Computing Machines*, IEEE CS Press, 2004, pp. 197–206.
18. V. Kindratenko and D. Pointer, "A Case Study in Porting a Production Scientific Supercomputing Application to a Reconfigurable Computer," *Proc. IEEE Symp. Field Programmable Custom Computing Machines*, IEEE CS Press, 2006, pp. 13–22.
19. R. Scrofano et al., "A Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers," *Proc. IEEE Symp. Field Programmable Custom Computing Machines*, IEEE CS Press, 2006, pp. 23–32.
20. J. Villareal and W. Najjar, "Compiled Hardware Acceleration of Molecular Dynamics Code," *Proc. IEEE Conf. Field Programmable Logic and Applications*, IEEE CS Press, 2008, pp. 667–670.
21. Y. Gu, T. VanCourt, and M. Herbordt, "Explicit Design of FPGA-Based Coprocessors for Short Range Force Computation in Molecular Dynamics Simulations," *Parallel Computing*, vol. 34, no. 4–5, 2008, pp. 261–271.
22. Y. Gu and M. Herbordt, "FPGA-Based Multigrid Computations for Molecular Dynamics Simulations," *Proc. IEEE Symp. Field Programmable Custom Computing Machines*, IEEE CS Press, 2007, pp. 117–126.
23. R. Skeel, I Tezcan, and D.J. Hardy "Multiple Grid Methods for Classical Molecular Dynamics," *J. Computational Chemistry*, vol. 23, no. 6, 2002, pp. 673–684.
24. D. Rapaport, *The Art of Molecular Dynamics Simulation*, Cambridge Univ. Press, 2004.
25. F. Ding and N. Dokholyan, "Simple but Predictive Protein Models," *Trends in Biotechnology*, vol. 3, no. 9, 2005, pp. 450–455.
26. N. Dokholyan, "Studies of Folding and Misfolding Using Simplified Models," *Current Opinion in Structural Biology*, vol. 16, no. 1, 2006, pp. 79–85.
27. J. Model and M. Herbordt, "Discrete Event Simulation of Molecular Dynamics with Configurable Logic," *Proc. IEEE Conf. Field Programmable Logic and Applications*, IEEE CS Press, 2007, pp. 151–158.
28. D. Kitchen, "Docking and Scoring in Virtual Screening for Drug Discovery: Methods and Applications," *Nature Reviews—Drug Discovery*, vol. 3, Nov. 2004, pp. 935–949.
29. E. Katchalski-Katzir et al., "Molecular Surface Recognition: Determination of Geometric Fit between Proteins and Their Ligands by Correlation Techniques," *Proc. Nat'l Academy of Science* (PNAS), vol. 89, Mar. 1992, pp. 2195–2199.
30. T. VanCourt and M. Herbordt, "Rigid Molecule Docking: FPGA Reconfiguration for Alternative Force Laws," *J. Applied Signal Processing*, vol. 2006, no. 1, 2006, pp. 1–10.
31. T. VanCourt, *LAMP: Tools for Creating Application-Specific FPGA Coprocessors*, PhD thesis, Dept. of Electrical and Computer Eng., Boston Univ., 2006.

Martin C. Herbordt *is an associate professor in the Department of Electrical and Computer Engineering at Boston University, where he directs the Computer Architecture and Automated Design Lab. His research interests include computer architecture, applying configurable logic to high-performance computing, and design automation. Herbordt has a PhD in computer science from the University of Massachusetts. He is a member of the IEEE, the ACM, and the IEEE Computer Society. Contact him at herbordt@bu.edu.*

Yongfeng Gu *is a member of the technical staff at the MathWorks. His research interests include reconfigurable computing, computer architecture, and hardware/software codesign. Gu has a PhD in computer and systems engineering from Boston University. Contact him at maplegu@gmail.com.*

Tom VanCourt *is a senior member of the technical staff, software engineering, at Altera Corp. His research interests include applications and tools for reconfigurable computing. VanCourt has a PhD in computer systems engineering from Boston University. He is a member of the IEEE, the ACM, and the IEEE Computer Society. Contact him at tvancour@altera.com.*

Josh Model *is an associate technical staff member at MIT's Lincoln Laboratory. His research interests include the use of FPGAs in scientific computing and hyperspectral image processing. Model has an MS in electrical engineering from Boston University. Contact him at jtmodel@bu.edu.*

Bharat Sukhwani *is a PhD candidate in the Department of Electrical and Computer Engineering at Boston University. His research interests include FPGA acceleration of scientific applications, high-level design environments for FPGA-based systems, and VLSI CAD tools for nanotechnology devices. Sukhwani has an MS in electrical and computer engineering from the University of Arizona. He is a student member of the IEEE. Contact him at bharats@bu.edu.*

Matt Chiu *is a PhD candidate in the Department of Electrical and Computer Engineering at Boston University. Chiu has an MS in electrical engineering from the University of Southern California. Contact him at mattchiu@bu.edu.*