Promises and Pitfalls of Reconfigurable Supercomputing

Maya B. Gokhale, Christopher D. Rickett, Justin L. Tripp, Chung Hsing Hsu
Los Alamos National Laboratory

Ronald Scrofano
University of Southern California

Abstract

Reconfigurable supercomputing (RSC) combines programmable logic chips with high performance microprocessors, all communicating over a high bandwidth, low latency interconnection network. Reconfigurable hardware has demonstrated an order of magnitude speedup on compute-intensive kernels in science and engineering. However, translating high level algorithms to programmable hardware is a formidable barrier to the use of these resources by scientific programmers . A library-based approach has been suggested, so that the software application can call standard library functions that have been optimized for hardware. The potential benefits of this approach are evaluated on several large scientific supercomputing applications. It is found that hardware linear algebra libraries would be of little benefit to the applications analyzed. To maximize performance of supercomputing applications on RSC, it is necessary to identify kernels of high computational density that can be mapped to hardware, carefully partition software and hardware to reduce communications overhead, and optimize memory bandwidth on the FPGAs. Two case studies that follow this approach are summarized, and, based on experience with these applications, directions for future reconfigurable supercomputing architectures are outlined.

1. Introduction

Reconfigurable supercomputing (RSC) puts the extreme performance potential of programmable hardware to work on computationally intensive algorithms in science and engineering. Combining high performance microprocessors, large Field Programmable Gate Arrays (FPGAs), and low latency, high bandwidth interconnect, reconfigurable supercomputers have demonstrated 10–100$\times$ acceleration on compute-intensive scientific kernels, e.g. [1, 2]. Leading supercomputer vendors[3, 4, 5] offer machines that include programmable logic, and new software tools are appearing [6, 7] that compile high level languages to hardware.

While reconfigurable supercomputers offer the potential to improve computational performance by orders of magnitude, the programmable hardware approach to high performance still presents significant obstacles to overall acceleration of scientific and engineering applications. FPGAs operate at a clock speed one order of magnitude slower than microprocessors. To overcome this gap, hardware designs must take advantage of as much parallelism as possible. However, these multiple hundred thousand line programs are dominated by 64-bit floating point computation, which consumes too much area and memory bandwidth on present-day FPGAs to be competitive with dedicated 64-bit floating point units on microprocessors. Computation is applied to gigabyte memory arrays that can't fit on current FPGA boards, requiring the communication of large blocks of data between software and hardware. The computational "hot spots" must be located, and the code partitioned between software and hardware, with the kernel being re-written either in a Hardware Description Language (HDL) or a C dialect that can be compiled to hardware.

In this paper, we review the state of the art in reconfigurable computing systems and tools. We then analyze a set of "grand challenge" scientific codes to assess the prospects for their acceleration on RSC architectures. We show two case studies of scientific reconfigurable supercomputing applications and define architectural enhancements that are needed in future reconfigurable supercomputers.

2. Reconfigurable Supercomputing Architectures

The Single Program Multiple Data (SPMD) model of parallel processing is widely used in scientific applications. In this model, an application is partitioned among a collection of identical node programs that communicate via message passing. This is a macroscopic form of parallelism at a coarse granularity. At the other end of the spectrum, fine grained instruction level parallelism is also well understood. This is the domain of vectorizing compilers, VLIW compilers, and hand-microcoded library functions, in which instructions to co-processor function units are intermixed with the sequential microprocessor instruction stream. Coprocessors share system clock and register files with the microprocessor.

Reconfigurable supercomputing architectures represent an intermediate level of parallelism between task and instruction level. These FPGA-based acceleration engines operate asynchronously from their microprocessor hosts. They are typically accessible from the microprocessor via an I/O bus or interconnection network. The FPGA modules usually consume significantly lower power than a microprocessor and perform best in data streaming applications such as multimedia or network packet processing. The FPGA modules typically have a non-cached memory sub-system, and concurrent access to multiple memory banks is required to achieve high performance at a low clock rate. Reconfigurable supercomputers thus combine large capacity programmable logic chips, high performance microprocessors, and low latency, high bandwidth interconnection networks into computing clusters, as shown in Figure 1.

Reconfigurable supercomputers can execute applications that contain I/O and control-intensive portions, which execute on the general purpose processors, and compute/data-intensive portions, which execute on the FPGAs. Figure 1 shows the most generic form of reconfigurable supercomputer in which the CPU nodes and FPGA nodes are peers on the interconnection network. Each sort of node includes local memory, interface to other devices, and interface to the interconnection network. The SGI RASC architecture conforms most closely to this configuration [4]. On the SGI architecture, the programmable logic resource includes FPGA, on-board memory, and a specialized ASIC, the TIO chip, that communicates with SGI's NUMAlink interconnection network. FPGAs can communicate with each other directly over NUMAlink. Communication with CPUs is also over the same NUMAlink. In the SGI architecture, the general purpose processor is an Itanium 2, with its own NUMAlink interface chip, the SHUB. SGI's implementation is unique among reconfigurable supercomputers in that it supports a coherent shared address space across the entire NUMAlink network.

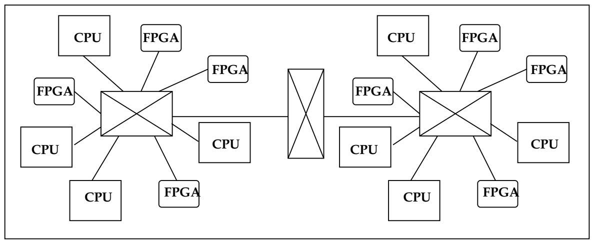

Figure 1. Generic Reconfigurable Supercomputer Architecture

The Cray XD1 architecture differs from the generic reconfigurable supercomputer in that the FPGA module communicates directly with an interconnect module associated with a CPU. Thus, each FPGA module is linked to a specific CPU, in contrast to the SGI, in which FPGA resources are not associated with any particular CPU. The Cray uses Opteron processors and has a proprietary interconnection network based on hypertransport (HT). FPGAs can be aggregated either directly using the on-chip Rapid I/O interconnect[1] or through the "RapidArray" hypertransport interconnect. The latter requires that the CPUs associated with FPGAs manage the communication, adding latency to the hypertransport interconnect path. As shown in Figure 2, the FPGA module on the XD1 board connects directly to the RapidArray link of one of the dual socket Opterons.

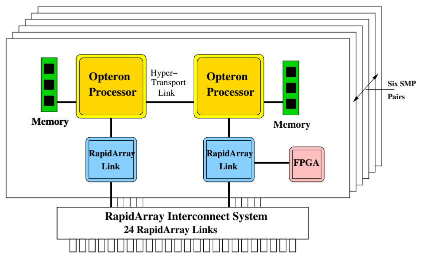

Figure 2. XD1 Processor Module

A recently announced architecture by AMD and Xilinx reduces latency between CPU and FPGA by connecting an FPGA directly to the hypertransport bus of an Opteron processor. This interconnect mechanism should greatly reduce latency between CPU and FPGA and could potentially increase realizable bandwidth.

[1]Although to date, firmware and software to support this feature are not available.

Table 1. Reconfigurable Supercomputer Comparison

Product	Proc	Comm BW	FPGA	Memory	Inter-FPGA
SRC 6	Xeon	2.8GB/s	2 V2-6K	24MB	Chaining 4.8 GB/s
Cray XD1	Opteron	3.2GB/s	V2Pro50	16MB	RapidIO 3.125 GB/s
SGI RASC	Itanium 2	6.4GB/s	V2-6K	16MB	NUMAlink (shared)

Another architectural alternative places the FPGA in the memory subsystem of the general purpose computer. This interconnect mechanism, used by SRC Computer, attaches the FPGA module into the DIMM slot of the processor (see Figure 3). The SRC MAPStation is a 2.8 GHz dual Intel Xeon microprocessor with an attached FPGA board (the MAP processor). The FPGA board connects to the Xeon via a “SNAP” Interface, which plugs directly into the DIMM slots on the motherboard, giving an interconnect bandwidth of 1400MB/s each for input and output. SRC provides a chaining option to allow multiple FPGAs to be aggregated on a single task.

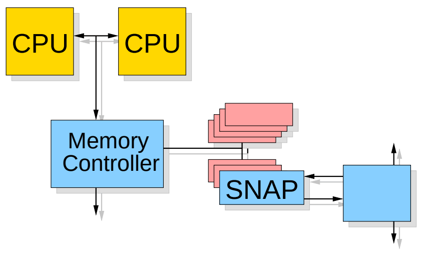

Figure 3. SRC Overview diagram

The most extreme form of reconfigurable supercomputing architecture eliminates the high performance microprocessor, creating large clusters of FPGAs that include programmable logic and embedded microprocessor on a single chip. This architecture was first developed by [8] for logic emulation. Starbridge Systems [9] offers up to 11 Virtex-II FPGAs and 36GB of DRAM per board, and a research system, the Berkeley Emulation Engine [10, 11] has been built for wireless network component emulation. While these “sea of gates” architectures are useful for emulation, they lack the high speed serial computational resources required for scientific supercomputing.

Table 1 summarizes the three commercially available (as of mid-2006) reconfigurable supercomputers. The next generation architecture from SGI is the RASC blade with 80MB SRAM or 20GB DRAM. The SRC 7 offers similar improvements in memory, I/O and clock rate over SRC 6.

3. Tools

There is a great need for software development environments that encompass the FPGA acceleration resources. Desired tools include profilers, performance estimation tools, compilers, and debuggers that operate seamlessly across software and hardware.

Unfortunately, the current state of the art is very far removed from this vision. On the Cray XD1, most system development is performed by first manually partitioning the software and hardware for the reconfigurable system. The FPGA design is written in an HDL like VHDL or Verilog. This design is combined with behavioral system cores that help simulate the entire system operation. Once the system has been verified in simulation, the design is then synthesized into a form that can be used to program the FPGAs. With the FPGA programmed, the software can then be integrated with the FPGA hardware. Recently, compilers are emerging that can compile both software and hardware for the XD1 [6].

The SRC development environment is more advanced than other reconfigurable supercomputers. Carte, SRC’s Programming Environment, allows the programmer to express FPGA designs in C or FORTRAN. While the programmer is still responsible for manually partitioning the design, the high-level language algorithm descriptions can be automatically synthesized into configuration bit streams for the FPGAs, which somewhat reduces the design effort. Further, the programmer can also write C or FORTRAN code to transfer data between the MAP and the microprocessor. Other compilers from C-like languages to FPGAs include Celoxica [7] and Mitrion-C (a dataflow language in C syntax) [12]. Most of the compilers include high level simulation, and the SGI software suite includes an “FPGA-aware” debugger.

4. Exploiting Reconfigurable Supercomputers

The programmable logic, on-chip memory, and dedicated arithmetic blocks on modern FPGAs offer opportunities for increasing both coarse-grained (application-level) and fine-grained (instruction-level) parallelism. The spatial parallelism available on FPGAs makes it possible to create arithmetic units and datapaths customized to individual statement sequences and loops. The customized datapaths and pipelines can also be replicated up to the capacity of the target device, offering additional parallelism.

However, effectively exploiting the potential of programmable logic for scientific applications can be extremely challenging. Many of these codes contain hundreds of thousands of lines of FORTRAN and have specific requirements as to system software, support libraries, communications libraries, and data file formats. It may be difficult to configure a reconfigurable supercomputer to even compile existing codes. Once compiled and running in the RSC software environment, it can be difficult to find the right parts of the code to translate to hardware. The capacity of FPGAs, though increasing rapidly, is still relatively small, especially for floating point computations, so the acceleration parts of the code must be carefully chosen. Although improving rapidly, the immaturity of high level tools makes hardware implementation a time-consuming process. Early adopters find that system integration issues consume more time than hardware development, making it difficult to assess what benefit the hardware resources actually afford.

4.1. FPGA Math Libraries

In view of these challenges, several research groups have suggested a library approach. Optimized hardware implementations of common library functions are created. Then, rather than modify the application itself, a new library wrapper layer is added to an existing library. When the application calls a library routine, the hardware version of the routine, if available, is executed. Thus the use of reconfigurable hardware is transparent to the application developer. Hardware implementations of functions in linear algebra libraries have been recently been built (e.g., [13, 14, 15]) and their performance analyzed. Unfortunately, performance results show that with today's FPGA architectures, single precision floating point implementations of many linear algebra routines are not appreciably faster than high performance microprocessors, and further, when the cost of communication is factored in, FPGA implementations are actually slower. Double precision floating point is significantly slower than on a microprocessor. In a study of floating point FFT implementations on FPGAs, it is postulated that future FPGA architectures will be better suited to floating point computation [16]. However, this prediction depends on the doubling of memory bandwidth with every new generation, which is unrealistic to expect due to the high memory bandwidth required by scientific application. In addition, FFTs with 4096 or more inputs must be performed in order to see performance benefit.

Let us assume for the moment that the combination of improved FPGA architectures (perhaps with dedicated floating point IP), optimized hardware libraries, and lower latency, higher bandwidth communications can be achieved, leading to a factor of 5 performance advantage for the FPGA library routine. The 5$\times$ figure is based on speculation that FPGAs in the future will outperform microprocessors as predicted by [16]. What does that factor of 5 mean in terms of overall speedup of the scientific application?

Supercomputing benchmarks such as LINPACK [17] heavily use linear algebra library routines. LINPACK is a collection of subroutines coded in FORTRAN that solve linear algebra and linear least squares problems on various sorts of matrices. Our profiling of LINPACK on a 2.2GHz Xeon (2GB memory, Intel ifort 8.0 -O2) shows that nearly three quarters (74%) of the execution time is spent in DGEMM, a double precision dense matrix multiply. Discounting communication costs, speeding up DGEMM by 5$\times$ would speed up LINPACK by 2.45$\times$. A 10$\times$ DGEMM speedup would yield overall 2.99$\times$. And this very respectable improvement would be totally transparent to the application developer since it is encapsulated in the library routine.

LINPACK is of course a benchmark, and thus not indicative of most scientific applications. To begin to quantify the benefits of the library approach, we studied the performance profiles of four large, floating point intensive, scientific supercomputing applications. The Parallel Ocean Program (POP)[18] is an ocean circulation model developed at Los Alamos and used in global climate prediction. Two applications were obtained from the NSF HPC Acquisition Benchmark Set [19], part of the NSF petascale computing environments initiative [20]. The NSF benchmarks are used to characterize system performance and include both system architecture and application benchmarks. The GAMESS and MILC benchmarks from the NSF suite were used in this study. MILC is a set of codes for doing SU(3) lattice gauge theory that runs on many MIMD parallel machines currently available, and on various workstations [21]. GAMESS is a quantum chemistry code [22]. The samples we looked at specifically with GAMESS were water-16.fmo2-{cc,dft,mp2,rhf} [23]. The final code, GROMACS, is a well known open source molec-

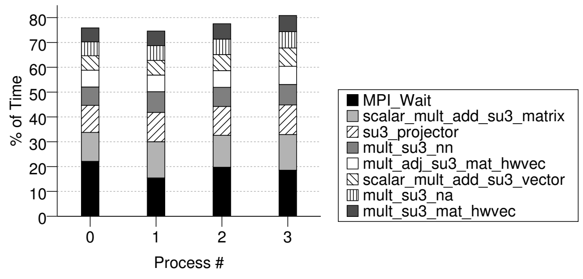

Figure 4. MILC Profile

ular dynamics package [24].

MILC, POP and GAMESS were profiled by using TAU [25] to automatically instrument the routines. TAU provides automatic instrumentation by inserting timer calls at the start and end of a function; timers are started at the beginning of a routine and stopped at any point of return for the routine. The instrumented source then gets compiled into the resulting object. The TAU configuration used for these applications would give both a total time spent in each routine, and call-path information to show what execution path(s) caused what amount of run time. The codes were run on a dual socket 2.4 GHz Xeon (2GB memory, icc/icpc 8.0, ifort 9).

The MILC profile, using four processes, is shown in Figure 4. The profile shows routines that use greater than 4% of the run time. The average percent of time spent in these routines (excluding time in MPI communications) is 55%. We assume that the software must wait for the hardware to complete, but for the moment omit communications costs for data arrays. We assume all seven routines are mapped to hardware. These routines are BLAS level 2 and 3 (vector-matrix and matrix-matrix operation). Using Amdahl's law with a 5× estimated speedup, the maximum speedup for MILC is 1.83×. If a speedup of 8× were possible in the linear algebra routines, the overall application speedup would be at most 2×.

The POP profile is shown in Figure 5. POP shows a much more challenging profile than MILC. The top routine is a conjugate gradient solver (PCG) that uses an average of 10.7% of the run time. The best that this code can achieve, assuming zero time in the PCG is a 12% speedup. Using a hardware library to do PCG would have little impact on POP run time.

The GAMESS code was profiled in sequential mode on four different input data files. Library routines dxpy and dgmm (wrappers to DAXPY and DGEMM) account for 10% of the run time or less, again pointing to disappointing results for a library-based acceleration approach.

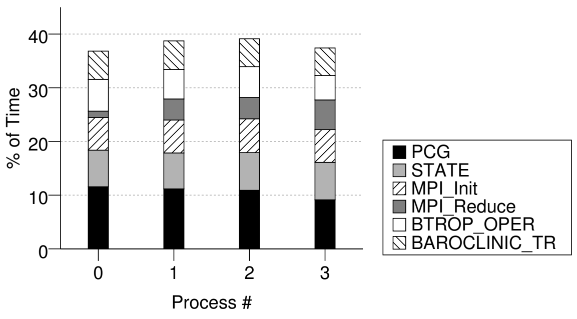

Figure 5. POP Profile

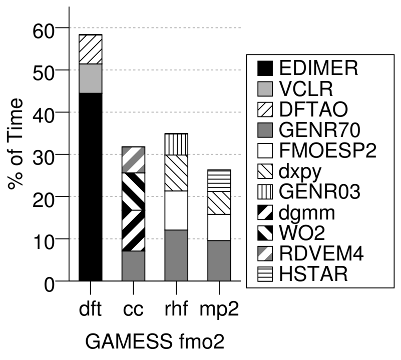

Figure 6. GAMESS Profile

GROMACS profiling is summarized in Figure 7, which shows gprof results on a Xeon from running GROMACS in a palmitic acid simulation. This application does not use linear algebra libraries at all. Instead, the nonbonded force calculations are coded in SSE assembly language (see inl3300_sse and inl3330_sse).

The above analysis did not consider communication costs. The FPGA modules of RSC systems function as attached processors rather than tightly coupled co-processors. Therefore, data must be transferred back and forth over the interconnection network between microprocessor and FPGAs. To better understand the amount of data that must be transferred, calls

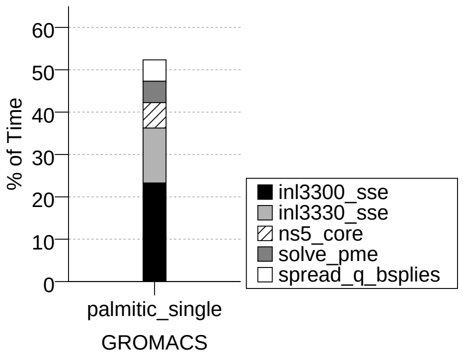

Figure 7. GROMACS Profile

to DGEMM in the GAMESS application were instrumented to keep track of matrix sizes being multiplied. Figure 8 shows the results for the water-16.fmo2-cc input data: the array sizes are clustered in the .5MB to 1.25MB range and there are many invocation ($>$ 40,000 – 60,000). This suggests that using the FPGA will require many small data transfers, negating any acceleration by the FPGA due to the required deep (high latency) pipelines.

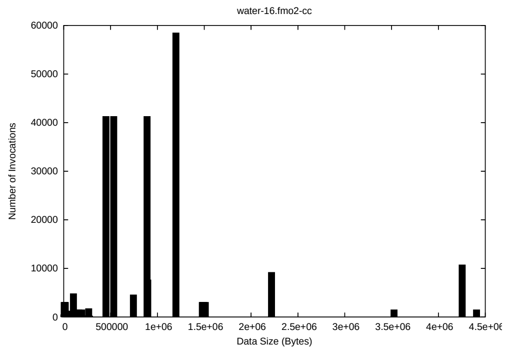

Figure 8. DGEMM Calls using water-16.fmo2-cc input

4.2. Re-thinking Applications for RSC

The profile study indicates that many scientific applications cannot depend on standard libraries, in particular the basic linear algebra library, to gain benefit from reconfigurable hardware, as a relatively small amount of time is spent in the library functions. Only one application (MILC) showed appreciable speedup. Further, most of the applications profiled did not show one, or even a couple of, small, computationally dense kernels whose hand optimization into hardware would yield significant speedup. Most of the routines with greater than 20% time are large subroutines that distribute their time calling lots of other functions. This is particularly the case in POP, where each routine takes less than 10% of the overall run time.

In view of these results, it is likely that simple approaches with minimal impact on existing scientific code will not benefit from using reconfigurable supercomputers. It is not sufficient either to call hardware versions of standard library function or to translate computational kernels as written in existing scientific applications into hardware.

Successful reconfigurable supercomputing applications must be developed from "first principles." This methodology requires a re-thinking of the scientific problem to be solved, careful partitioning between hardware and software to maximize hardware task granularity and communication requirements. It may be necessary to re-think numeric representations and data structures. The successful application capitalizes on the strengths of each resource – massive spatial parallelism for the reconfigurable hardware, and fast execution of control-flow-dominated code on the high performance microprocessor.

5. Applications Examples

Our experience shows that performance benefit can be gained with a combination of software and hardware for some scientific supercomputing problems. In this section we present two case studies of work in progress. In each case, it has been necessary to profile and analyze an existing application, devise a partitioning strategy that uses each resource to best advantage, create a new hardware/software version of the application, and then iterate with performance analysis and migration of functionality to hardware.

5.1. Metropolitan road traffic simulation

Los Alamos National Laboratory, in conjunction with Sandia National Laboratory, hosts the National Infrastructure Simulation and Analysis Center (NISAC), which can analyze critical, inter-dependent infrastructure elements and their potential vulnerabilities. For example, NISAC applications can simulate large metropolitan road networks, power grids, or even epidemics. In this case study, a road network simulator, part of the TRANSIMS [26] suite of tools, was mapped to the Cray XD1 reconfigurable supercomputer. This application has two characteristics that make it a good candidate for RSC. First, the simulation is based on a simple cellular automaton (CA) whose implementation can exploit the spatial parallelism available on FPGAs, Second, unlike most scientific simulation applications, CA processing requires simple logic on small integer data types.

TRANSIMS models the road network as a cellular automaton in which each cell simulates a 7.5 meter section of road. At each simulation step, each cell has a simple algorithm to determine whether it holds a car, whether it should forward a car to another road cell or should receive a car from another road cell. While the decision procedure is quite simple for a single lane road, it can become more complicated when there are multiple lanes, acceleration lanes, and intersections. Profiling (see Figure 9) showed a potential for up to 5$\times$ speedup if the lane change, velocity, and position updates could be put in hardware. Since the intersection logic is complicated, that module would best be done in

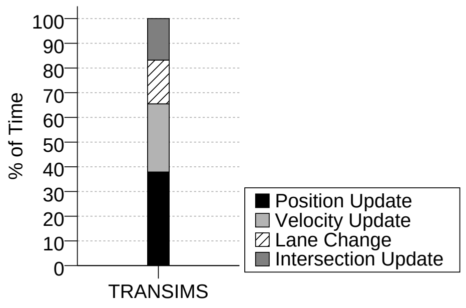

Figure 9. TRANSIM Profile

software. However, due to memory limitations on the current generation XD1 FPGA board, only the position and velocity update of single straight lanes was put into hardware, all other aspects of the simulation being performed in software.

The hardware algorithm uses a *streaming* approach, whereby a road cell processing engine (PE) reads cell state from a memory, updates the state and writes it back to memory. The PE can be internally pipelined, and multiple PEs can be instantiated on the FPGA. The XD1 memory subsystem consists of 4 quad data rate 64-bit memories of 4MB each (.5MW). The state of a single straight lane can be represented in 32 bits, allowing 4M road cells to be stored. Portland has 6.6M road cells, so at most 61% of the work can be done in hardware. Multiple lanes, intersections, and merging nodes would require at least 64 bits of state, so that only 2M road cells could be processing. We chose not to stream the road cells from the Opteron as the communication costs would dominate the time—a maximum of 1.6MB/s can be sent from the Opteron. In contrast, we can read at 3.2GB/s from each memory—12.8GB/s, almost an order of magnitude more bandwidth than the interconnection network port. The design has 8 processing engines, with each group of 2 sharing a memory.

At each simulation step, the hardware sends cars into road cells belonging to software and vice versa. Therefore, at the end of a simulation step, there is a brief communication between software and hardware to update state. While the naive approach would have the Opteron read data from the FPGA, this was found to be prohibitively expensive on the Cray. The write is a factor of 200 faster than the read. Therefore, the hardware writes cell state updates into Opteron memory and vice versa.

Table 2 compares the processing rates for (i) the raw FPGA without considering communication, (ii) streaming algorithm with host reading back updated state from FPGA memory, (iii) streaming algorithm with FPGA writing back updated state to host memory, and (iv) software only. This compares relative times to process the single straight lanes. If results calculated by the FPGA are read directly by the host, the speedup is 4.5×. Pushing the results up to the host results in a speedup of 34.4×.

Table 2. Comparison of Streaming including Communication Costs

	No comm. V2p50	w/o Push V2p50	Push V2p50	2.2GHz Opteron
Cells/sec	7.2×10^8	2.56×10^7	1.96×10^8	5.7×10^6
Speedup	126.3	4.5	34.4	1.0

Since hardware acceleration is limited to about 60% of the road cells due to memory limitations on the current generation Cray XD1, the sequential part of the execution dominates the overall run time. Measured speedup on the Cray using this design is 22%. If all single straight lanes of this data set could be processed in hardware, the maximum speedup would be about 30%, as shown in Figure 10. Our hardware design is capable of processing multiple lanes, and the goal with this application is to migrate position and velocity update of all road segments onto hardware, which will be possible in the next generation of RSC.

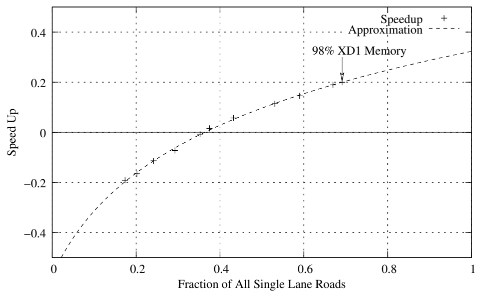

Figure 10. TRANSIM XD1 profile

This case study shows that future reconfigurable supercomputers must maximize memory bandwidth to the FPGAs in the form of multiple parallel, deep memory banks. In this application, chaining of FPGAs would also help improve speedup as additional FPGAs could be applied to the problem without having to go through the microprocessor.

5.2. Molecular dynamics simulation

Molecular dynamics (MD) simulations are used in many different fields, from materials science to nanoscience. MD simulates the motion of atoms in a system over time. Given an initial position $\vec{r}_i(0)$ and an initial velocity $\vec{v}_i(0)$ for each atom i, the simulation computes the new position and velocity of each atom in small (femtosecond) time increments to track the trajectory of the system from time 0 to time t_f, the final simulation time.

In this work (see [27]), the velocity Verlet update algorithm [28] is used:

1. calculate $v_i\left(t+\frac{\Delta t}{2}\right)$ based on $v_i(t)$ and acceleration $a_i(t)$;
2. calculate $r_i(t+\Delta t)$ based on $v_i\left(t+\frac{\Delta t}{2}\right)$;
3. calculate $a_i(t+\Delta t)$ based on $r_i(t+\Delta t)$ and $r_j(t+\Delta t)$ for all atoms $j \neq i$;
4. calculate $v_i(t+\Delta t)$ based on $a_i(t+\Delta t)$.

The most compute-intensive step of the Verlet algorithm is to calculate accelerations (step 3). To get an atom's acceleration, it is necessary to compute the forces acting on the atom. The forces that act upon the atoms in the simulation can be broken into two types: bonded and nonbonded. Bonded forces—namely bond stretch, angle bend, and dihedral torsion—only act between atoms that share bonds and thus are not usually a bottleneck. Nonbonded forces, which include short-range Lennard-Jones forces and long-range electrostatic (Coulomb) forces, can act between any atoms in the system and require more computational time. Consequently, the calculation of nonbonded forces appears promising for hardware acceleration. However, before developing a hardware implementation for nonbonded force calculation, we must determine how much benefit doing so will provide for the complete application.

Our first step was to develop our own basic software MD simulation program. This was required due to the high degree of optimization for parallel processing or cluster systems found is most existing MD simulation packages (e.g. GROMACS). This also allows us to focus on a basic MD simulation that could provide meaningful scientific results.

In our simulation, all interactions are simulated as atom-atom interactions, regardless of the type of molecule of which the atoms are a part. Also, only the shifted-force technique for long-range electrostatic interactions is implemented rather than the more accurate—but more complex—Ewald summation or particle mesh Ewald techniques [29]. It is important to note that we can include the more advanced techniques in our software and incrementally port those techniques to hardware.

Once written, we profiled the performance of our simulation program on two real-word simulations. Table 3 shows the result of this profiling. The palmitic acid simulation simulates an acid that forms a one molecule thick monolayer on water. This simulation has 52,558 atoms and a time step of 2 fs was used. The CheY Protein simulation simulates this protein in water. This simulation has 32,932 atoms and a time step of 2 fs.

Table 3. Profile of our software MD implementation for two simulations

Task	% of Computation Time	
	Palmitic Acid	CheY Protein
Nonbonded Forces	75.15	74.09
Building Neighbor List	20.75	22.7
Dihedral Torsion Forces	2.14	1.50
Other	1.92	1.09

From the profile, it is clear that nonbonded force calculation makes up the bulk of the simulation time. Using Amdahl's law, we see that accelerating nonbonded force calculation can provide a maximum speedup of 4× for the overall palmitic acid simulation and 3.9× for the overall CheY protein simulation. This task is where we focus our acceleration effort.

The algorithm for nonbonded force calculation is given in Figure 11. In the figure, `position` and `force` represent arrays in memory that hold the position of and force acting upon each atom i. The list of neighbors j of atom i is precomputed using the Verlet neighbor list technique [28]. CALC_NBF represents evaluating the equations for shifted force Lennard-Jones and Coulomb interactions. $\vec{r}_{ij}$ is the distance vector between atoms i and j. r_C is the cutoff distance: if two atoms are farther apart than r_C, it is assumed that they do not interact. Though not shown in the figure, potential energy is also calculated at this time. The shifted force equations used to calculated Lennard-Jones force and potential energy ($\vec{f}_{ij}^{LJ}$ and U_{ij}^{LJ}, respectively) and electrostatic force and potential energy ($\vec{f}_{ij}^{E}$ and U_{ij}^{E}, respectively) between two atoms are given in Equations 1 through 4, where q_i represents the charge on atom i and A_{ij}, B_{ij}, C_{ij}, and D_{ij} are constants whose values depend upon the types of atoms i and j.

$$\vec{f}_{ij}^{LJ} = \left(\frac{6}{r_{ij}^2}\left(\frac{2A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}}\right) + \frac{D_{ij}}{r_{ij}}\right)\vec{r}_{ij} \qquad (1)$$

$$U_{ij}^{LJ} = \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} + C_{ij} - D_{ij}r_{ij} \qquad (2)$$

$$\vec{f}_{ij}^{E} = \frac{q_i q_j}{r_{ij}}\left(\frac{1}{r_{ij}^2} - \frac{1}{r_C^2}\right)\vec{r}_{ij} \qquad (3)$$

$$U_{ij}^{E} = \frac{q_i q_j}{r_{ij} r_C}\left(r_C - r_{ij}\right)^2 \qquad (4)$$

The basic idea for the hardware implementation of the nonbonded force calculation is to, at each time step, transfer the current atomic positions and the neighbor list to the FPGA's external memory. A pipeline that takes advantage of instruction-level parallelism is implemented with pipelined single precision floating point cores to execute the algorithm in Figure 11. This pipeline is replicated to achieve parallelism. Once all of the forces have been calculated, they are transferred back to the microprocessor and the rest of the tasks are executed in software. In practice, several factors limit the performance of the hardware implementation.

```
1  foreach atom $i$ do
2      $\vec{r}_i \leftarrow$ position[$i$]
3      $\vec{f}_i \leftarrow$ force[$i$]
4      foreach neighbor $j$ of $i$ do
5          $\vec{r}_j \leftarrow$ position[$j$]
6          if $|\vec{r}_i - \vec{r}_j| < r_c$ then
7              $\vec{f}_{ij} \leftarrow$ CALC_NBF($\vec{r}_i, \vec{r}_j$)
8              $\vec{f}_i \leftarrow \vec{f}_i + \vec{f}_{ij}$
9              $\vec{f}_j \leftarrow$ force[$j$]
10             force[$j$] $\leftarrow \vec{f}_j - \vec{f}_{ij}$
11         end
12     end
13     force[$i$] $\leftarrow \vec{f}_i$
14 end
```

Figure 11. Algorithm for nonbonded force calculation

With current FPGA technology, floating point cores—even single-precision ones—require a large amount of area. As can be seen from Equations 1 through 4, this implementation will require many floating point operations in its pipeline. As a result, there is not enough area to replicate the pipeline and achieve parallelism that way; only one pipeline can be instantiated. Even if there were enough area on the FPGA to replicate multiple pipelines in parallel, transferring data into the FPGA would then become a bottleneck because, with currently available reconfigurable supercomputers, there is not enough bandwidth between the FPGA and its external memory to provide data to several pipelines in parallel.

Another difficulty arises from data dependencies within the algorithm itself. For example, line 9 of the algorithm requires a read from the force memory and line 10 of the algorithm requires a write to the force memory. On some reconfigurable supercomputers, the memories are single-ported. This read/write conflict will cause the pipeline to stall. A further penalty may be incurred if time is required to switch the external memory between read mode and write mode.

There is an additional data hazard due to the pipelining of the floating point cores. For nonbonded force calculation, the pipelining leads to a data hazard. Specifically, the subtraction and write back to memory in line 10 may not have finished by the time the read in line 9 tries to read that data. There is no way to forward data within the floating point cores, so the only solution is to stall the pipeline, which diminishes performance.

To combat these two data dependencies, we developed a *write-back* design. In this design, only the inner loop of the algorithm is pipelined. In line 10, rather than writing to external memory, the updated force is written to temporary internal storage. This eliminates the need to read from and write to a single-ported memory in the same cycle. Once the inner loop has completed, all of the forces stored internally are written back to external memory. Because a neighbor j can only appear in atom i's neighbor list once, this removes the hazard due to pipelining. Once the forces have been written to external memory, the next iteration of the outer loop can begin. The drawback of this design is that the inner loop pipeline must be drained for every iteration of the outer loop. This limits the effectiveness of the pipelining.

Table 4. MAPstation performance results

Simulation	Latency (s/step)		Speed-up
	SW only	SW + HW	
Palmitic Acid	1.28	0.66	2.0×
CheY Protein	0.74	0.39	1.9×

The write-back design was implemented on the SRC MAPstation. Despite the factors limiting performance, we were able to achieve an overall speedup of about 2× for both the palmitic acid and the CheY protein simulations, as shown in Table 4. So, while our MD program is less sophisticated than large simulation packages such as GROMACS, we were able to target our acceleration efforts and achieve a 2× speedup. The code, as is, can be used to simulate large numbers of atoms in a simplified simulation. Additional functionality can easily be added to the simulation program in software and then, if necessary, accelerated with hardware.

This application, like the traffic simulation, uses all available memory bandwidth. Performance is limited by the cost of the floating point operators, the requirement of 100MHz clock frequency, and the speed of the communications. Improvement of these factors in future reconfigurable supercomputers will translate into better performance for the MD application.

6. Conclusions

Reconfigurable supercomputing offers the potential for acceleration well beyond Moore's Law improvements in microprocessors. However, the performance benefit may not be apparent in scientific applications as currently structured. The acceleration comes at the cost of re-writing the application to devise appropriate partitioning schemes between software and hardware and synthesizing circuits for the hardware portion of the partitioned application. As the case studies show, with a complete re-factoring of the application to take advantage of the reconfigurable supercomputing architectures, compute kernels show speedup of more than order of magnitude. However, overall application speedup depends on minimizing the amount of sequential code. Improvements in reconfigurable supercomputing architectures must occur before application speedup over 2$\times$ can be realized.

References

[1] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, P. rsen F. Curt, and D. W. Prather, "FPGA-based acceleration of 3D finite-difference time-domain method," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), 2004.

[2] M. Gokhale, J. Frigo, C. Ahrens, J. L. Tripp, and R. Minnich, "Monte carlo radiative heat transfer simulation on a reconfigurable computer," in *International Conference on Field Programmable Logic and Applications*, August 2004.

[3] Cray, Inc., Seattle, WA, *Cray XD1 Datasheet*, September 2004.

[4] www.sgi.com/products/rasc, "Sgi products: Rasc technology," 2006.

[5] www.srccomp.com, "SRC Computers," 2006.

[6] www.impulseC.com, "ImpulseC Accelerated Technologies," 2005.

[7] Celoxica, Ltd, "HANDEL-C language overview," tech. rep., Celoxica, Ltd, Aug 2002.

[8] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal, "Logic emulation with virtual wires," *IEEE Transactions on Computer Aided Design*, vol. 16, pp. 609–626, June 1997.

[9] www.starbridgesystems.com, "Starbridge systems, inc.," 2006.

[10] C. Chang, K. Kuusilinna, B. Richards, and R. Brodersen, "Implementation of bee: a real-time large-scale hardware emulation engine," *FPGA 2003*, pp. 91–99, Feb. 2003.

[11] C. Chang, J. Wawrzynek, and R. W. Brodersen., "Bee2: A high-end reconfigurable computing system," *Design and Test of Computers*, pp. 114–125, Mar/Apr 2005.

[12] mitrion.com, "Mitrion website," 2006.

[13] L. Zhuo and V. K. Prasanna, "Design tradeoffs for blas operations on reconfigurable hardware," in *International Conference on Parallel Processing*, 2005.

[14] L. Zhuo and V. K. Prasanna, "High performance linear algebra operations on reconfigurable systems," in *SC|05*, 2005.

[15] M. C. Smith, J. S. Vetter, and S. R. Alam, "Scientific computing beyond cpus: Fpga implementation of common scientific kernels," in *MAPLD*, 2005.

[16] K. S. Hemmert and K. D. Underwood, "An analysis of the double-precision floating-point fft on fpgas," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), IEEE Press, April 2005.

[17] www.netlib.org/linpack, "Linpack," 2006.

[18] climate.lanl.gov/Models/POP/index.htm, "Parallel ocean program (POP)," 2006.

[19] www.nsf.gov/publications/, "Hpc acquisition benchmarks," 2006.

[20] www.nsf.gov/publications/, "High performance computing system acquisition: Towards a petascale computing environment for science and engineering NSF05625," 2006.

[21] www.physics.utah.edu/ detar/milc/milcv6.html, "The milc code (version: 6.20sep02)," 2006.

[22] M.W.Schmidt, K.K.Baldridge, J.A.Boatz, S.T.Elbert, M.S.Gordon, *et al.*, "General atomic and molecular electronic structure system," pp. 1347–1363, 1993.

[23] P. Piecuch, S. Kucharski, K. Kowalski, and M. Musial, "Gamess," *Comput.Phys. Commun.*, pp. 71–96, 2002.

[24] www.gromacs.org/, "Gromacs: The world's fastest molecular dynamics – and it's gpl," 2006.

[25] www.cs.oregon.edu/research/tau/home.php, "Tau - tuning and analysis utilities," 2006.

[26] transims.tsasa.lanl.gov, "TRANSIMS Web Page," 2006.

[27] R. Scrofano, M. Gokhale, F. Trouw, and V. K. Prasanna, "A hardware/software approach to molecular dynamics on reconfigurable computers," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), IEEE CS, IEEE Press, April 2006.

[28] M. Allen and D. J. Tildeseley, *Computer Simulation of Liquids*. New York: Oxford University Press, 1987.

[29] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen, "A smooth particle mesh Ewald method," *Journal of Chemical Physics*, vol. 103, pp. 8577–8593, November 1995.