# A case study in porting a production scientific supercomputing application to a reconfigurable computer

Volodymyr Kindratenko, David Pointer
*National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign*
*kindr@ncsa.uiuc.edu, pointer@ncsa.uiuc.edu*

## Abstract

*This case study presents the results of porting a production scientific code, called NAMD, to the SRC-6 high-performance reconfigurable computing platform based on Field Programmable Gate Array (FPGA) technology. NAMD is a molecular dynamics code designed to run on large supercomputing systems and used extensively by the computational biophysics community. NAMD's computational kernel is highly optimized to run on conventional von Neumann processors; this presents numerous challenges to its reimplementation on FPGA architecture. This paper presents an overview of the SRC-6 architecture and the NAMD application and then discusses the challenges, solutions, and results of the porting effort. The rationale in choosing the development path taken and the general framework for porting an existing scientific code, such as NAMD, to the SRC-6 platform are presented and discussed in detail. The results and methods presented in this paper are applicable to the large class of problems in scientific computing.*

## 1. Introduction

Computational scientists aggressively seek floating point application performance improvements beyond those implied by Moore's Law. Reconfigurable computing (RC) [1] based on the Field Programmable Gate Array (FPGA) technology is one of the technologies that have the potential to yield performance improvements for many demanding computational tasks. Until recently, however, its potential has been largely locked from the scientific computing community due to limited FPGA resources (e.g., lack of support for floating point arithmetic), general unavailability of the tightly coupled RC/CPU systems, and a lack of high-level programming languages suitable for a rapid FPGA code development. This changed with the recent introduction of several high-performance RC (HPRC) platforms, such as Cray XD1 [2], SGI RASC [3], and, most relevant to this paper, SRC-6 MAP™ [4] and accompanying Carte™ development tools [5]. These synergetic systems provide a tight coupling between the code executed on the host microprocessor and the FPGA and exploit coarse-grain functional parallelism through conventional parallel processing as well as fine-grain parallelism through direct hardware execution on FPGAs.

Even though commercially available high-level languages, such as MAP C [5] and Mitrion-C [6], greatly reduce the complexity of code development efforts for RC platforms, porting an existing scientific code to an RC platform using one of these languages is not as simple as just recompiling the code with a different compiler to run on a different microprocessor system. It requires adaptation of the code to the available FPGA resources – something scientific code application developers are not familiar with. The software developers need to be hardware-savvy in order to produce an efficient code, but even then it is not always possible to gain the maximum performance without writing low-level routines in hardware languages, such as VHDL or Verilog.

The goal of this paper is to demonstrate the process of porting an existing scientific code to a modern RC platform and to show the difficulties that must be surmounted in order to produce a workable solution. More specifically, this paper reports on the efforts to port a production-grade molecular dynamics (MD) code, NAMD [7], to a modern production RC platform, the SRC-6 MAP [4], using a high-level language, MAP C [5]. While a number of papers [8-12] have been published in the past few years about implementing a textbook MD code on an RC platform, to our knowledge this is the first attempt to port an *existing production-grade* MD code that is extensively optimized to run on large parallels systems and is routinely used by the scientific community. To our knowledge, this is also the first attempt to port an MD code to a mainstream *production* HPRC system using a high-level programming language rather than to an experimental RC platform using VHDL or Verilog, as in [10-12], or even a hardware system specifically designed to run MD simulation [8, 9].

NAMD [7] is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. It currently accounts for the largest number of compute cycles on NCSA's production supercomputing systems and, as such, has a continuous demand for performance improvements. Therefore it was natural to select this code to port to an RC platform in this test-case study.

SRC-6 MAP [4] was selected as the target platform because it is one the most readily available production RC systems on the market. The development toolset, called Carte [5], supplied by the vendor was another reason, as it provided a clear path for code partitioning and code porting to the FPGA as well as a convenient debugging and simulation environment.

This paper is intended to serve as a collection of the RC code development recipes that one might refer to when considering porting a scientific code to an RC platform. As we go through the code examples, we demonstrate why even a 3x speedup of the code ported to an RC platform (compared to the code executed on the host CPU platform) requires some significant software development efforts. We also demonstrate how the code performance on an RC platform can be estimated even before it is written. This allows a programmer to evaluate the suitability of the given RC system for the given problem.

## 2. Related work

A number of related efforts to develop an MD code on various hardware platforms have been reported in the literature. They range from developing Application Specific Integrated Circuit (ASIC) chips [8-9] to FPGA-based systems [10-12]. We will review the work by N. Azizi et al. [11] and R. Scrofano and V. Prasanna [12] as it is the most relevant to our effort.

In [11], the authors used the Transmogrifier 3 (TM3) FPGA platform and implemented the code in VHDL. A complete simulation cycle, including particle pair interaction calculations and position updates, is implemented on TM3 platform. Location of each particle in the simulation is stored in the SRAM banks on the TM3. For each time step of the simulation, the pair generator module calculates the distances between pairs of particles, the Lennard-Jones (L-J) force calculator module uses this distance to compute force between them, and the acceleration update module sums up the computed force to obtain the total acceleration for each particle. Finally, the Vertel update module updates the position and velocity of each particle based on the computed acceleration. These steps are repeated until all particle pairs have been examined. The L-J force calculator stores the L-J potential function in a lookup table and uses this table to interpolate a more accurate value. Varying precision, typically between 22 and 76 bits, was used to represent various data fields. The system was able to simulate an 8,192 particle model at a rate of 37 seconds per one time step while running at 26 MHz. The CPU-based benchmark code runs in 10.8 seconds on a 2.4 GHz P4 platform. The authors extrapolate that with better FPGA memory organization and faster FPGAs, a speedup of 40x to 100x over a microprocessor implementation can be achieved.

In [12], a fully pipelined VHDL implementation of L-J potential and force equations on a Xilinx Virtex-II Pro XC2VP125 FPGA is presented. The 119-stage pipeline – consisting of several adders, multipliers, divides, and a square root operation and running at 122 MHz – is set up to accept one squared particle-particle distance as input and computes two 64-bits wide results (potential and force) as output using IEEE 754 double precision floating point arithmetic. The authors compare their design with similar code executed on several microprocessor platforms and find their design to achieve 3.9 GFLOPS throughput compared to 1.5 GFLOPS throughput obtained on an Itanium2 900 MHz system.

Both [11] and [12] are concerned with the implementation of L-J force calculator, although [11] goes a step further by implementing the complete simulation cycle. It is not clear from [12] how the interface between the host system and the design executed on the FPGA is set up and how the developed system can be integrated with a complete MD simulation. None of the related efforts to implement an FPGA-based MD simulation design are concerned with the problem of accelerating existing MD simulation software and none of the FPGA-based MD codes that we have read about have been tested with more than just a few thousand particles.

## 3. SRC-6 and Carte

The SRC-6 MAPstation [4] used in the course of this study consists of a commodity dual CPU Xeon motherboard, a MAP Series C processor, and an 8 GB common memory module, all interconnected with a 1.4 GB/s low latency switch. The SNAP™ Series B interface board is used to connect the CPU board to the Hi-Bar switch. The SNAP plugs directly into a CPU board's DIMM memory slot.

The MAP Series C processor module contains two user FPGAs, one control FPGA, and memory. There are six banks (A-F) of on-board memory (OBM); each bank is 64 bits wide and 4 MB deep for a total of 24 MB. The programmer is responsible for application data transfer to and from these memory banks via the use of SRC programming macros invoked from the FPGA application. There is an additional 4 MB of dual-ported memory dedicated solely to data transfer between the two FPGAs.

The two user FPGAs in the MAP Series C are Xilinx Virtex-II XC2V6000 FPGAs. They each contain 67,584 4-input lookup tables, 67,584 flip flops, 144 dedicated 18x18 integer multipliers, and 324 KB of internal dual-ported block RAM (BRAM). The 144 hardware multipliers are sufficient to have up to 36 single precision floating point multiplication operators in the design. These FPGA elements are not directly visible to the programmer but are interconnected appropriately as determined by the programmer's MAP C algorithm code, the SRC Carte programming environment [5] tools, and the Xilinx FPGA place and route tools. The FPGA clock rate of 100 MHz is set by the SRC programming environment.

The MAP Series E processor module, also used in the course of this study, is identical to the Series C module with the exception of the user FPGAs. The two user FPGAs in the MAP Series E are Xilinx Virtex-II Pro XC2VP100 FPGAs. They each contain 88,192 4-input lookup tables, 88,192 flip flops, 444 dedicated 18x18 integer multipliers, and 999 KB of internal dual-ported block RAM. The 444 hardware multipliers are sufficient to have up to 111 single precision floating point multiplication operators in the design.

The Carte programming environment [5] for the SRC MAPstation is highly integrated, and all compilation targets are generated via a single makefile. The two main targets of the makefile are a debug version of the entire program and the combined microprocessor code and FPGA hardware programming files. The debug version is useful for code testing before the final time-intensive hardware place and route step. The Intel icc compiler is used to generate both the CPU-only debug executable and the CPU-side of the combined CPU/MAP executable. The SRC MAP compiler is invoked by the makefile to produce the hardware description of the FPGA design for final combined CPU/MAP target executable. This intermediate hardware description of the FPGA design is passed to the Xilinx ISE place and route tools, which produce the FPGA bit file. Lastly, the linker is invoked to combine the CPU code and the FPGA hardware bit file(s) into a unified executable.

## 4. NAMD

A detailed description of the underlying physical model used in NAMD can be found in [7]. Here we are mostly concerned with the problem of porting the existing NAMD code written in C++ to FPGA using MAP C language, therefore we provide analysis of the source code instead.

The core of NAMD simulation is a function that computes force exerted on each atom (a simplified version of which is given in Code Listing 1 of the Appendix, *calc_both* subroutinge). This force later is applied to move the atoms according to the Newtonian equation of motion. In the current implementation, we only focus on the van der Waal's forces (approximated by the L-J 6-12 potential) and electrostatic interactions between the nonbonded atom pairs. Several optimization techniques have been employed in NAMD to reduce the time needed to compute the forces. Thus, the entire simulation space is divided into 3D cells, called *patches*, whose size is related to the *cutoff radius* beyond which no interaction calculations are performed. The atoms from each patch are run against themselves and against the atoms from the 13 neighbor patches (Newton's third law of motion is applied here to eliminate redundant calculations) and the corresponding interaction calculations are performed on these patches. Cutoff radius is applied to limit the calculations to only those atoms that are close to each other. The smooth particle-mesh Ewald (SPME) method is used for full electrostatic computations with the direct component of PME sum substituting the Coulomb equation. Interpolation tables are used for both L-J potential and Coulomb: All of the *fast_* terms are Coulomb (or PME direct sum), and all of the *vdw* terms belong to L-J potential. The use of these tables eliminates the need for floating point division and square root operations.

Thus, as one can see, NAMD works very differently from a textbook MD code. Its algorithmic structure and code optimization strategies are well tailored to the microprocessor architecture. The textbook code, on the other hand, typically implements a set of full-blown calculations and has none of the optimizations applied, which makes it easier to speed up the basic force computations on FPGAs.

The code snapshots shown in the Appendix are taken from an earlier version of the benchmark kernel extracted from NAMD version 2.4 with a few additional modifications made per our request: The exclusion or modification of nonbonded interactions between pairs of atoms connected by three or fewer bonds is eliminated, bonded force calculations are omitted, and the code is modified to use a reduced numerical resolution (single precision floating point instead of the double precision as in the production code). These 'minor' code alterations of course make it impossible to perform a direct performance comparison of our code with the production NAMD code, but they are necessary to make the problem of porting the code to the FPGA tractable. Nevertheless, even with these modifications, the code is still valid for properly formulated problems.

The computational complexity of a molecular dynamics code, such as NAMD, is well understood [7]. It is the nonbonded particle-particle interaction that is responsible for the vast majority of the computational time: In a brute force approach, there would be $N^2$ particle-particle interactions. NAMD avoids the $O(N^2)$ computational complexity by applying the numerous optimization techniques mentioned above, yet the nonbonded particle-particle interaction kernel is still the most computationally expensive portion of the code, and it is responsible for over 80% of the overall time spent by the simulation. This is an important observation as it will be instrumental in helping us to decide what portion of the NAMD code should be ported to MAP.

A typical NAMD simulation consists of 100,000 atoms, with 300,000 atoms being occasionally used at the high-end, simulated for the duration of several nanoseconds on a 64+ microprocessor system [7]. As an example, for a representative system of 92,224 atoms simulated for a 1 ns simulation (one million time steps of 1 femtosecond each), a typical simulation time on the NCSA's 128 CPU SGI Itanium platform is 5.1 hours, with the simulation of a single 1 femtosecond step taking about 0.018 seconds. The same system of atoms simulated on a 64 CPU Xeon cluster requires over 16 hours.

## 5. Porting NAMD to SRC-6 MAP

How does one approach the problem of porting an existing scientific code to an HPRC platform? First, the programmer needs to look into what sort of computational operations need to be accelerated. This is typically accomplished with the help of a code profiling tool, such as *gprof*. Second, one needs to identify what portion of the code embedding these operations should be ported. There is an important difference between the code that needs to be accelerated and the code that should be ported to FPGA. For example, looking at Code Listing 1, it is obvious that the inner loop calculations in *calc_both* subroutine are responsible for all the execution time. Yet if we only port the inner loop calculations to FPGA, we risk wasting too much time due to the frequent data transfer between the MAP and the host system and MAP function call overhead associated with invoking the SRC-6 MAP subroutine. Therefore, we should consider porting code that includes the outer loop instead.

But how about just one level above, that is, the *runComputes* subroutine? This would eliminate the need to call the MAP subroutine multiple times, right? It turns out that the amount of data that the *runComputes* subroutine operates on, in general, is unbounded, and there is a risk that for some simulations we may not have enough memory available on MAP. On the other hand, the *calc_both* subroutine operates on small data patches, typically from a few hundred to a thousand data points. Therefore, by porting the *calc_both* subroutine instead of the *runComputes* subroutine, we make a compromise between the number of MAP function calls and the amount of data that needs to be available on MAP at once. By porting the outer loop inclusively rather than porting only the inner loop calculations, we make a compromise between the number of MAP function calls and the complexity of the code to be implemented on MAP.

Now that we know what code segment should be ported to MAP, how can we be sure that there are enough FPGA resources for the code to fit? Resource needs for hardware multipliers and memory can be estimated based on the number of the multiplications and memory allocations in the original code. However, FPGA space needs are much harder to gauge. There are over 30 multiplication operations in the original NAMD code shown in Code Listing 1. Not all of them, however, operate on the floating point numbers, and some of them are used to compute pointer offsets. Therefore, as we port the code, we will likely reduce the number of multiplications. However, even with 30 multiplication operations present, there are just enough hardware multipliers to accommodate all of them while using SRC's multiply macros. We also estimate that the *calc_both* subroutine requires access to less than 100 KB of memory, which is certainly not a problem for XC2V6000 FPGA. However, we will have to postpone finding out about space requirements for the design until we actually map it to the FPGA.

## 5.1. Host CPU code

Now that we decided what code segment to port to MAP, what is the next step? Even though MAP C is very much an ANSI C-like language, a significant effort is required to port NAMD to MAP C. There are two main difficulties: absence of the high-level data structures present in ANSI C and memory usage considerations on the SRC-6 MAP. The first issue can be addressed by converting C/C++ data structures to one-dimensional arrays; the second issue requires a detailed analysis of the execution structure of the code in order to develop a fully pipelined FPGA implementation. There are other considerations that will become obvious as we attempt to port the code.

What data are involved with the nonbonded force calculations? First, atom data stored in the *CompAtom* class: atom displacement, charge, and type. Second, three force values stored in the *Force* class. And finally, two lookup tables accessible through *ljTable* and *table_four* pointers. Note that the lookup tables remain constant between the calls to the *calc_both* subroutine, therefore they can be loaded to the MAP memory only once, whereas atom and force data change between the function calls.

How do we send C++ data structures and classes to the MAP memory? In order to do this, we need to copy the data from these structures to 1D arrays whose characteristics are set according to the memory structure supported by SRC-6 MAP. The basic memory transfer unit on this system is a 64 bits wide word that can be split on the FPGA into 4, 8, 16, or 32 bits wide data types, including a 32 bits wide integer and a 32 bits wide floating point number [5]. Therefore, we have to pack our data into arrays of 64 bits wide words that later can be unpacked on the FPGA into appropriate data types. This is trivial for the lookup tables as they both consist of double precision (64 bits wide) floating point numbers that we further reduce to the single (32 bits wide) precision. Code Listing 2 shows how these lookup tables are copied to the *fpga_table_four* and *fpga_lj_pars* linear arrays.

Data translation for the *CompAtom* and *Force* classes is not as trivial as it is not all of the same type (e.g., displacement and charge are stored as 32 bits wide floating point numbers whereas force values and atom type are stored as integers). First, we declare two data structures, *fpga_I_D* and *fpga_O_D*, that will hold all the required input and output data for each atom (see Code Listing 2). We need to make sure that the size of these data structures is a multiple of 64 bits; thus, in the case of *fpga_O_D* we add an extra 32 bits wide field which will be unused, but needs to be present (this can be optimized later). By using continuous 1D arrays of elements of data structures like these, we ensure that the entire dataset to be sent to MAP will occupy a continuous memory block and thus can be transferred to MAP at once.

Now that we have all the data translation code written as shown in Code Listing 2, we are ready to outsource the actual computations to MAP. Note that we need to copy atom input data and force output data between the MAP memory and the host memory each time we call the MAP function.

Note that now the *calc_both* subroutine is composed only with the data transfer code; the actual calculations are hidden in the *map_compute_func* subroutine, which is to be implemented on MAP. Also note that the lookup table data translation and memory allocation for the atom input and force output data needs to be done only once per the entire simulation, whereas atom input and force output data need to be copied in and out for each data patch, that is, for each call to the *calc_both* subroutine (a static variable *firsttime* is used to flag this).

At this point, we can also consider the issue of performance measurements and obtain some results for the code executed purely on the host system. First, we move the actual calculations code from *calc_both* to the *map_compute_func* subroutine with the appropriate modifications to the data structures used. Second, we instrument the code with timers (*gettimeofday* library function calls) in such a way that we can measure the overall executing time for the *calc_both* subroutine and the time spent by the *map_compute_func* subroutine alone. We accumulate these time measurements over the duration of one simulation step, which includes multiple calls to *calc_both*. We run the code with the dataset containing 92,224 atoms [7] just for one simulation step. This results in 2,016 calls to the *calc_both* subroutine (there are 144 data patches in the dataset and each patch is used 14 times). The original code shown in Code Listing 1 executes in 9.25 seconds (wall clock) on a 2.8 GHz Intel Xeon processor, whereas the code shown in Code Listing 2 executes in 9.96 seconds, where the *map_compute_func* routine is responsible for 9.9 seconds and the remaining 0.06 seconds are due to the data translation overhead. So far our modifications of the code have resulted in the overall slowdown by approximately 0.71 seconds.

We should point out that the overall compute time for this code heavily depends on the simulation parameters set by the user, particularly the cutoff radius value. In the present dataset, a vast majority of the atoms are outside the cutoff radius from each other, therefore the "*r2 > cutoff2*" test eliminates a significant number of unnecessary calculations. Just as a thought experiment, if all the atoms in the neighborhood patches would be within the cutoff radius from each other (or if the cutoff radius is set to be large enough), the overall compute time would be just over 800 seconds instead of 9.25 seconds. We will refer to this observation in the sections that follow.

## 5.2. Single chip MAP Series C design

Now that we have taken care of data translation between the host platform and MAP, all that is left is to translate the actual computational kernel code to FPGA! First, one should consider porting the code *as is* to obtain an executable that produces the correct results. Only then one should consider how to optimize the code to gain the best overall performance. Why? First, even the simplest code translation to MAP C will likely involve dealing with a number of issues, such as data mapping to MAP memory, pipelining the loops, etc. Thus, many issues will be resolved. Second, one will have a 'reference design' for analysis and comparison to future results. Third, one will be able to get a good measure of the FPGA resources utilization to achieve just the basic implementation. This will be handy in deciding if any of the likely optimizations will fit on the chip; thus one could avoid starting with a design that cannot fit on the chip. And finally, at this stage of the development one can estimate the overall code performance even before the code is written and compare it with the performance after the code is running. If the results agree in bulk, one is likely to be on the right track as far as understanding the code and the issues at hand.

Let us first analyze the possible performance of a straight *as is* code port to SRC-6 MAP. From running our 92,224 atoms test case consisting of 144 atom patches, we know that: there are 144*14=2,016 calls to the *calc_both* subroutine; 144 calls involve the same patches and 1,872 calls involve pairs of the neighbor patches; and each call on average involves checking 640 atoms from one patch against 640 atoms from another patch. Thus, there are ~$640^2$*1,872+((640-1)*640/2)*144=796,216,320 interaction calculations to be performed. If we design a fully pipelined implementation of the inner loop of the *calc_both* subroutine that will take one FPGA clock cycle to run just one of these calculations, then it will likely take 796,216,320/1e8≈7.96 seconds to run just the inner loop calculations (MAP FPGAs operate at 1e8 Hz) plus the pipeline depth penalty for each call to the inner loop. It is difficult to know in advance what the pipeline depth might be; for example, if we estimate it to be 150 clock cycles, then the overall pipeline depth penalty can be computed as ~(640*1,872+639*144)*150/1e8≈1.94 seconds. There are also data to be transferred in and out: (640*2*1,872+640*144)*12*4=119,439,360 bytes of data in total, which will take about 119,439,360/16/1e8≈0.08 seconds plus some additional time to arm the DMA engine and distribute the data to MAP memory, as needed. Also, there is a MAP function call overhead on the order of 220 microseconds per call (based on our own measurements), resulting in 220*2,016*1e-6≈0.44 seconds plus about 100 milliseconds to initialize the PFGA when it is called the first time [13] and plus some time to load up the lookup tables. Therefore, based on the estimated number of clock cycles needed to perform the required operations, we would expect our MAP code to run in over 11 seconds with the bulk of time (7.96+1.94=9.9 seconds) spent doing actual calculations and some time spent on data transfer and MAP function call overhead.

Thus, such a straightforward code port will likely result in an overall slowdown (original code runs in just 9.25 seconds on the CPU). As we see from our estimated performance, the vast majority of time running the code on MAP will be spent doing nonbonded atoms interaction calculations; therefore, in order to gain some speedup we should consider ways to optimize this portion of the (yet nonexistent) design. We can get a speedup if we can have two or more atom interactions calculated simultaneously. However, looking at the number of floating point operations in Code Listing 1, it may not be possible to place more than one interaction calculation engine on the FPGA chip due to the lack of hardware multipliers. We will return to this issue later.

Now we provide a step-by-step implementation sequence of the computational core. First, we declare the MAP function as shown in Code Listing 3. Next, looking at the code in Code Listing 1, we need to have simultaneous access to nine values from the *table_four* lookup table and two values from the *ljTable* lookup table. Therefore, we need to distribute the data from these tables in the MAP memory accordingly so that we can access all the required data in a single clock cycle. Since these tables are relatively small, we can allocate them in the FPGA's BRAM memory (see Code Listing 3). The original *table_four* lookup table actually contains 12 values per record, some of which are needed to perform electrostatic energy calculations, which we omitted from our code. We will ignore these unneeded values for now. In order to achieve some efficiency with the lookup table data transfer to BRAM, we stripe *table_four* between six OBM memory banks, thus allowing access to all 12 lookup table values at once by splitting 64 bits wide words in half. *ljTable* can simply be transferred to just one OBM bank and then split and copied to two BRAM arrays (Code Listing 3). Note that the lookup data transfer needs to be done only the first time the code is called as the BRAM content does not change between multiple calls to the MAP subroutine.

Now we are ready to transfer the actual simulation data consisting of eight 32-bits-wide values. We do need to have access to all eight values in a single clock cycle in order to achieve a fully pipelined implementation. Therefore, we distribute the input atom data among four separate 64-bits-wide memory banks such that they can be accessed simultaneously (see Code Listing 3). We could just DMA in the data and then distribute it as needed; instead, we have chosen to stream it in and perform the data distribution simultaneously, thus cutting the number of clock cycles spent on these operations. Also, atom displacement, charge, and type data are read-only, whereas force data needs to be updated as the result of the computations. Therefore, we put the atom force data to BRAM (*cl_bram* and *dl_bram* BRAM arrays) as it is dual ported and can be updated within the same clock cycle. Fortunately, the nature of the calculations is such that we do not have to worry about memory clash, and therefore we can safely instruct the MAP C compiler via *#pragma loop noloop_dep* macro to pipeline the inner loop even tough there are memory dependencies.

Now that all the necessary data have been loaded to the MAP memory, we can port the actual inner loop calculations. What is left at the end is to transfer the results accumulated in *cl_bram* and *dl_bran* to the host process. We implement this via streaming (Code Listing 3).

When we compile the resulting code, MAP C compiler informs us that the inner loop was successfully pipelined with the pipeline depth of 159 clock cycles. The placing and routing report indicates that the design occupies 59% of the hardware multipliers, 61% of BRAM banks, and 87% of SLICEs and the design meets timing requirement of 10 ns. We also instrumented the code to take timing measurements (not shown in the code provided). When we run the resulting code, we find that the overall execution time is 12.05 seconds (compared to 9.25 seconds of the same code running on the CPU): 10.26 seconds are spent doing the actual calculations, which is close to our theoretical estimate of 9.9 seconds, and the remaining 1.79 seconds are spent on the data transfer and MAP function call overhead. Thus, the net result is 1.3x slowdown of the MAP-based NAMD code compared to the original CPU-based implementation. The results produced by our MAP-based code are correct for the most part, with some minor errors due the reduced numerical resolution used in our MAP implementation.

A question arises: What would constitute a fair performance comparison? Should we use the overall execute time that includes associated function call overheads, (that is, 12.05 seconds), or should we use just the time spent on FPGA performing only the relevant calculations (10.26 seconds in our case) as the performance measure? We have chosen to use the overall execution time that includes the compute time and all the related function call overheads.

It is interesting to note that since we have a fully pipelined implementation of the inner loop, we always spend just one clock cycle to either perform the entire set of calculations (if the atoms are within the cutoff radius from each other) or to skip them entirely (if the atoms are too far from each other). This is a very different behavior from what is happening with the microprocessor-based implementation. There, we skip a significant number of calculations if the atoms are too far from each other. The CPU-based code, execution time therefore, will change when we change the cutoff radius whereas the FPGA-based model execution time will remain constant. Thus, if the cutoff radius was set large enough to include all the atoms in the neighbor patches, the overall compute time of the CPU-only design would be just over 800 seconds as shown in Section 5.1, whereas the FPGA-based design would still execute in just 12.05 seconds, which would constitute a 66x speedup.

## 5.3. Dual chip MAP Series C design

Since our initial design already occupies 87% of available SLICEs, we are unable to utilize the data-level parallelism on the same chip; that is, we cannot place two compute engines on the same FPGA. However, we should be able to utilize the secondary FPGA chip available in the SRC-6 MAP. Thus, in our next design, we spread the calculations between two FPGA chips in such a way that the primary chip performs half of the calculations and the secondary chip performs another half of the calculations and the results are merged at the end as they are streamed out. We still keep inner loops on both chips pipelined with the same pipeline depth as in the single chip design. The primary chip is responsible for bringing in the data and distributing it between two chips, merging the results from both chips, and streaming them out.

The place and route report indicates that the same 59% of the hardware multipliers are used on the primary chip; however, only 50% of the BRAM blocks are required since now the data are distributed between two chips. Also, about 92% of SLICEs are now in use on the primary chip as there is more logic required to implement the inter-chip communication. On the secondary chip, we use 61% of the hardware multipliers, 50% of BRAM, and 78% of SLICEs. The overall execution time now is 6.92 seconds--5.13 seconds are due to the calculations (about half of the previous design) and 1.79 seconds are due to other expenses, as before. Comparing 6.92 seconds to the original 9.25 seconds of the same code running on the CPU, we have achieved a 1.3x speedup. (It would be over 100x for a larger cutoff radius, as we discussed in the previous section.)

## 5.4. Dual chip MAP Series E design

The first dual chip design simply takes advantage of the trivial parallelism; that is, we cut the overall compute time in half by dividing the work between two equivalent compute engines. Unfortunately, this all we can do on the SRC-6 MAP Series C that uses XC2V6000 FPGAs; there is simply not enough space on the chip to place any more compute pipelines. SCR-6's XC2VP100-based MAP Series E system offers more space, hardware multipliers, and BRAM memory blocks to place one additional compute engine on each chip. Thus, our next step is to port the code to the MAP Series E (XC2VP100 FPGA)-based system.

It is a simple recompile to get our original code developed for the MAP Series C system to run on the MAP Series E system. However, we want to extend the code to divide the work between four compute engines, two per FPGA. This requires distributing lookup tables and input data between twice as many memory banks so that we can have independent access to all the required data in a single clock cycle. This also requires designing a more elaborate result assembling engine since the forces for each input atom are now partially computed by each of four compute engines. The primary chip is responsible for bringing in the input data and distributing it between the compute engines and also assembling the final results and streaming them out, as before. Perhaps the only other significant difference on this design is that we do not use OBM memory banks anymore to store the input data; instead we use the BRAM memory blocks as there is more BRAM memory available in XC2VP100 FPGAs. OBM memory is still used to store the results obtained on the secondary chip.

When we compile this design, we observe that the pipeline depth for the inner loops of each of the compute engines is now 150 clock cycles due to the elimination of the OBM memory access and other minor optimizations that we have implemented in the process of modifying the FPGA design. The place and route report indicates that the SLICEs utilization on the primary chip is 97%, but we use only 28% of the available hardware multipliers and 40% of BRAM blocks. SLICEs utilization of the secondary chip is 87%, and the hardware multipliers and BRAM memory usage are the same as on the primary chip. The design meets timing specifications and executes in just 3.57 seconds, of which 2.58 seconds are due to the calculations and the remaining 0.99 seconds are due to the usual function call overhead and data transfer overhead. Note that on the XC2VP100-based SRC-6 MAP system this extra overhead is somewhat smaller than on the XC2V6000-based system. Thus, we achieved a 2.5x speedup as compared to the CPU-based code. (Note that it would be over 200x for a larger cutoff radius.)

Perhaps we can still gain some performance if we merge the inner and outer loops in a fully pipelined manner and therefore eliminate the pipeline depth penalty mentioned in Section 5.2. This, of course, requires a significant code revision as we cannot reuse the same memory banks for the inner and outer loop. When we made all the necessary changes, the overall compute time became 3.07 seconds: 2.08 seconds due to the calculations (thus, we saved about 0.5 seconds) and 0.99 seconds due to various overheads This resulted in a 3x overall speedup.

## 5.5. An alternative design

We cannot continue adding more compute engines as we have reached the limits of available space on the MAP Series E FPGAs. However, we may be able to achieve a better performance if we restructure our code to take advantage of the fact that force calculations are not required for the vast majority of atom pairs. We currently check the distance between all atom pairs, and then perform the force calculations on atom pairs that pass the distance check. This wastes a clock cycle for each pair of atoms that fails the distance check. Calculating the distance between a pair of atoms does not require much of the FPGA's resources, therefore it might be possible to implement several distance compute engines in parallel so that we can search through all the input data much faster and call the interaction calculations section of the code only as many times as there are force values to compute. Ideally, we would like to start the interaction calculations as soon as one of the distance compute engines identifies a pair of suitable atoms. Unfortunately, this is difficult to implement with the current MAP C version as all the streaming primitives supplied by SRC can only have a single source and there is no multiple input/single output FIFO primitives available. Perhaps a suitable data buffering mechanism can be implemented in a low-level language, but this is beyond the scope of our present work. Instead, our parallel distance compute engines store found atom pairs in separate arrays that are later merged into a single array, which is then used by the interaction calculations engine – not an optimal solution.

The performance of this approach is difficult to predict as it is wholly data-dependent. If we introduce N simultaneous distance compute engines, then we can sort through the input data in 1/N of the time that would be required using the implementation discussed in Section 5.2. But we cannot know in advance how many atom pairs are within the cutoff radius, and therefore we cannot tell how many clock cycles it will take to run the force calculations. The worst case would be if all the atoms are within the cutoff radius and therefore we add M/N clock cycles in addition to our usual M clock cycles to the calculations where M is the total number of atom pairs. The best case scenario would be when none of the atoms are within the cutoff radius and thus we discover this in just M/N clock cycles instead of our usual M clock cycles. The reality, of course, is somewhere in-between.

We have implemented this approach on the XC2VP100-based SRC-6 MAP Series E system with six distance compute engines and one force compute engine per FPGA. The design complexity, as compared to all our previous implementations, has increased significantly, yet the code executes in just 3.02 seconds as compared to 3.08 seconds for our previous design.

## 6. Discussion and conclusions

In the previous section, we outlined a path that one may find useful in order to port a scientific code, such as NAMD, to SRC-6 and other RC platforms. To summarize, we began with the run-time analysis of NAMD computational core and identified what part of the code should be ported in order to accelerate the overall code performance and what compromises are to be made if we decide to port one or another section of the code. We then isolated the code to be ported to FPGA into a separate standalone function and took care of data translation between the C++ classes and 1D arrays suitable for FPGA. With this code we conducted performance measurements, estimated FPGA resource requirements, and predicted the execution time of the MAP-based code to gain an idea of what to expect when we first port the code to MAM. We then proceeded to port the code inside the outsourced function with the goal of obtaining a MAP implementation that worked and produced the correct numerical results. We outfitted this design with time measurement macros to verify that our theoretical performance estimates were inline with the actual code. Our initial estimation of the compute time on the FPGA was mostly correct; however, our estimation of the performance penalty due to the MAP function call overhead was not as precise. We then proceeded to perform several design iterations of the code based on our observation of the FPGA resource utilization and design performance.

Our experience with this and other codes shows that a sensible approach is to run over several iterations, starting with the simplest, most straightforward implementation and gradually adding to it until we either find the best solution or run out of FPGA resources. We started with just over 100 lines of code that runs on a CPU in just 9.25 seconds for one simulation step and ended with over 1,000 lines of code that requires both the CPU and MAP involvement, although the actual calculations are done on MAP, and executes in just 3.07 seconds. Our MAP-based implementation is three times faster then the original code. These results, of course, are data-dependent, as we have seen that for a larger cutoff radius the original CPU code executes in over 800 seconds, which would constitute a 260x speedup. Our 3x speedup perhaps is not worth the effort; however, the 260x speedup would well be worth the development effort undertaken.

As one can see, a noticeable effort in our work was due to the need to translate data between the C++ data storage mechanisms and the system defined MAP/FPGA data storage architecture. When starting to develop a code from scratch to run on an FPGA architecture, one would implement from the beginning the data storage mechanisms compatible between the CPU and FPGA. This, however, is rarely the case for an existing code, and it adds to the amount of work to be done in porting the code.

Our final design described in Section 5.4 occupies almost all available SLICEs, yet utilization of memory banks and hardware multipliers is low. Thus, we have reached the SLICEs limit before any other resource limits. This perhaps indicates that we should consider ways to restructure the code to make greater use of other available resources. One way may be to overlap calculations with data transfer for the next dataset and thus use more of the available on-chip memory.

One might ask why have we achieved such a modest speedup while numerous other papers and reports claim 10x-100x speedup for many codes.

Consider the fact that until now most FPGA code development has begun with writing code that implements a textbook algorithm, with no or little optimization. When such an unoptimized code is ported to an RC platform and care is taken to optimize the FPGA design, a 10x-100x speedup is easily achievable. In contrast, we began with code that has been in existence for the past decade and that has been thoughtfully optimized to run on the CPU-based platform. Such code successfully competes with its FPGA-ported counterpart.

The difference between the production-grade, highly optimized molecular dynamics code and a textbook MD example perhaps can be best demonstrated as follows. Both [10] and [11] use the same textbook MD code and report comparable results for running the reference code with 8,192 particles. Thus, in [10] a single simulation step runs in 9.5 seconds on a 2.4 GHz Xeon platform. For comparison [7], NAMD runs one simulation step involving 92,224 atoms in just over 2 seconds on a 1.6 GHz Itanium 2 platform!

In light of this, a 3x performance increase of this heavily optimized code is significant in that it illustrates the potential of reconfigurable system technology. Remember that it is a 100 MHz FPGA achieving a 3x application performance improvement over a 2.8 GHz CPU, and FPGAs are on a faster technology growth curve than CPUs [14].

## 8. Acknowledgements

This work was funded by the National Science Foundation grant SCI 05-25308. NAMD was developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign. We are especially grateful to Dr. James Phillips from the Theoretical and Computational Biophysics Group for his help with NAMD. Access to SRC-6 MAP Series C and E MAPstations was provided by SRC Computers Inc. We also would like to thank David Caliga, Dan Poznanovic, and Jeff Hammes, all from SRC Computers Inc., for their help and support with the SRC-6 system. Our special thanks are for Dr. Craig Steffen and David Raila from the NCSA's Innovative Systems Laboratory for their contribution to the work and Trish Barker form NCSA's Office of Public Affairs for the help in preparing this publication.

## 9. References

[1] M.B. Gokhale, and P.S. Graham, *Reconfigurable Computing : Accelerating Computation with Field-Programmable Gate Arrays*, Springer, Dordrecht, 2005

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

[3] Silicon Graphics Inc., Mountain View, CA, *SGI RASC Technology Datasheet*, 2005.

[4] SRC Computers Inc., Colorado Springs, CO, *SRC Systems and Servers Datasheet*, 2005.

[5] SRC Computers Inc., Colorado Springs, CO, *SRC C Programming Environment v 1.9 Guide*, 2005.

[6] Mitrionics Inc., Lund, Sweden, *The Mitrion-C Programming Language*, 2005.

[7] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD", *Journal of Computational Chemistry*, 26:1781-1802, 2005.

[8] S. Toyoda, H. Miyagawa, K. Kitamura, T. Amisaki, E. Hasimoto, H. Ikeda, A. Kusumi, and N. Miyakawa, "Development of MD engine: High-speed acceleration with parallel processor design for molecular dynamics simulations", *Journal of Computational Chemistry*, 20(2):185–199, 1999.

[9] T. Fukushige, M. Taiji, J. Makino, T. Ebisuzaki, and D. Sugimoto, :A highly parallelized special-purpose computer for many-body simulations with an arbitrary central force: Md-grape", *The Astrophysical Journal*, 468:51–61, 1996.

[10] Y. Gu, T. Van Court, D. DiSabello, M.C. Herbordt, "Preliminarly report: FPGA acceleration of molecular dynamics computations", in Proc. *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM'05), pp. 269-270, Nappa, CA, 2005.

[11] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow, "Reconfigurable Molecular Dynamics Simulator", in Proc. *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM'04), pp. 197-206, Nappa, CA, 2004.

[12] R. Scrofano, and V.K. Prasanna, "Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware", In Proc. *International Conference on Engineering of Reconfigurable Systems and Algorithms* (ERSA'04), pp. 284-292, Las Vegas, NV, 2004.

[13] O. Fidanci1, D. Poznanovic, K. Gaj, T. El-Ghazawi, N. Alexandridis, "Performance and Overhead in a Hybrid Reconfigurable Computer", In Proc. *10th Reconfigurable Architectures Workshop* (RAW'2003), p. 176b, Nice, France, 2003.

[14] K. Underwood, "FPGAs vs. CPUs: Trends in Peak Floating Point Performance," In Proc. *12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, 2004, pp 171-180.

## 10. Appendix

http://www.ncsa.uiuc.edu/~kindr/papers/FCCM06-appendix.pdf