Compiled Code Acceleration of NAMD on FPGAs

Jason Villarreal, John Cortes, and Walid A. Najjar
Department of Computer Science and Engineering
University of California, Riverside

***Abstract*— Spatial computing, as implemented on reconfigurable platforms, such as FPGAs, presents an effective solution to high performance computing applications where computations are applied repeatedly to streams of data. Bridging the gap between program descriptions and hardware system implementations, however, is a challenging problem. ROCCC is a C/C++ to VHDL compiler that focuses on extensive loop, array and procedure code transformations. In this paper we report on using ROCCC to compile NAMD, a modern and popular molecular dynamics program, and executing it on the SGI Altix 4700 with a RASC blade. The critical region of NAMD is a loop nest that computes the non-bonded forces on the atoms in a molecule. An instance of this loop consists of 52 floating point operations performed on several input arrays. Using ROCCC, this loop was translated to VHDL and then mapped onto the Virtex 4 LX200 of the RASC blade. Our results show a speedup exceeding 800x over a 1.6 GHz Itanium processor with a potential sustained rate of 7.7 Gflops on the FPGA for a single precision floating point implementation.**

I. INTRODUCTION

Spatial computing, the implementation of a computation as a circuit rather than as a sequence of instructions, is extremely effective in applications where one or more computations are applied repeatedly to a very large set of data. These include image, video and signal processing, cryptography, dynamic programming, and many high-performance computing applications. One such high performance computing application that fits this profile is NAMD (Nanoscale Molecular Dynamics).

As discussed in [21], FPGAs are now highly capable of performing many floating point operations in parallel, making their use in high performance computing applications even more attractive.

The difficulty in reconfigurable computing is programming the system and translating sequential computing systems into an equivalent and efficient spatial implementation.

The most common approach for creating these systems is to manually partition the original system description into a spatial section that is going to go into hardware and a sequential section that is going to go into software, and then hand-code the hardware section using a hardware description language such as VHDL or Verilog. This hardware portion must then interface with other hardware components on an environment specific basis as well as interface to the software portion of the code using a vendor-supplied library.

The main problem with this approach is that the system designer requires some hardware knowledge. Software designers typically lack the experience to design good hardware. Hardware description languages are not well suited to algorithmic or behavioral descriptions of problems and require low level knowledge that most software designers are unfamiliar with. Additionally, most hardware description languages have several tedious details and idiosyncrasies that could cause problems are often unfamiliar to a software designer. Further, software companies are reluctant to try new languages and will tend to stay with computation models they have used in the past.

A popular alternative is to recode spatial portions of a system in a special high level language such as MapC for the SRC platform [10], ImpulseC [13], or Mitrion-C for the Mitrion Virtual Processor [18], that require explicit parallel information to be added to the code. The parallelism is described either with special constructs or by writing the code in a parallel, single assignment way. This typically involves reworking and reimplementing large portions of the original code base to support these constructs as these constructs differ significantly from the original sequential constructs.

We have developed the ROCCC compiler system as an alternative. ROCCC is an optimizing compiler that translates C code into pipelined VHDL for use on FPGAs. ROCCC is designed to translate computationally intensive kernels written in a subset of C into highly parallel spatial implementations in order to improve performance of the entire system. Sections of the system may still have to be rewritten, but can be coded in a subset of C (without any explicit parallel statements) that should be familiar to most software designers.

We applied our toolset to NAMD, a popular molecular dynamics implementation used for the simulation of large biomolecular systems.

Molecular dynamics simulations are based on the use of a molecular mechanics force field and the availability of three-dimensional structural templates derived using crystallographic data (from X-ray or neutron diffraction) or nuclear magnetic resonance (NMR) data [16]. The force field is based on a potential energy function that accounts for all pairwise interactions of a biomolecular system. The potential energy function is used to simultaneously solve Newton's equations of motion for all atoms of the biomolecular system.

Current processors barely allow for molecular dynamics simulations up to microsecond timescales, using explicit solvation models and atomic resolution. Atomic-level resolution and the presence of explicit solvent molecules are necessary for a realistic physiochemical representation of the biomolecular system. Another limiting factor is the size of the simulated system. Although the first molecular dynamics simulation was performed for a protein of a mere 500 atoms in 1976, the biologically interesting systems of today involve 10^4-10^6

atoms [1]. There is a great need to increase both the size of the simulated systems and the accessible timescales. The size is important to resemble the actual physically functional systems, as biomolecules do not act alone but form large complexes and multicomponent assemblies. The timescale is important to address unsolved fundamental problems, such as protein folding, but also the driving principles of biomolecular interactions.

NAMD is one of the most popular molecular dynamics software packages and has been optimized for use on multiprocessor systems. The most computationally intensive portion of a NAMD execution is the computation of the pairwise interactions between nonbonded atoms.

Using ROCCC, we were able to completely place a single precision implementation of the most frequently executed loop of NAMD in hardware starting from a C description. For the single precision loop we obtained a speedup of 808 times over the same loop iteration running on a 1.6 GHz Itanium processor.

The rest of the paper is organized as follows: Section 2 describes the ROCCC compiler. Section 3 describes the way we implemented NAMD and section 4 describes the final results. Section 5 describes related work in molecular dynamics and high level synthesis from C, and section 6 presents our conclusions and future directions.

II. THE ROCCC SYSTEM

The ROCCC compiler (ROCCC) is a system developed with the intention of converting portions of C code into highly parallel hardware implementations to be placed on reconfigurable computing platforms.

The focus of ROCCC is on generating highly parallel and optimized circuits from portions of C programs rather than statement-by-statement translation of C programs to VHDL. The main distinguishing feature of the ROCCC framework is its emphasis on compile-time loop transformations and optimizations. The objectives of ROCCC optimizations are:

1) Maximize the parallelism in the circuit
2) Maximize the clock rate at which the resultant circuit operates.
3) Minimize the number of off-chip memory accesses
4) Minimize the area of the circuit.

ROCCC is not designed to compile a whole program, instead ROCCC relies on the user to identify the functions that are the most compute-intensive in a given application. These functions typically consist of one or more loop nests. The functions are compiled to hardware and invoked via a specialized API that interfaces between the host processor and the FPGA fabric. The remainder of the code executes in software on the host. The source code that will be translated to hardware is constrained such that the code contains no pointers, no break or continue statements, and all memory addresses must be resolvable at compile-time.

The ROCCC system implements two intermediate representations that are referred to as Hi-CIRRF and Lo-CIRRF (Compiler Intermediate Representation for Reconfigurable Fabrics) [8]. Hi-CIRRF can be represented by C code with special macros that represent specialized hardware blocks. Lo-CIRRF is an intermediate representation much closer to the actual hardware we generate.

The ROCCC system is currently built on the SUIF2 [22] and Machine-SUIF [11] [19] platforms, with the SUIF2 platform being used to process many optimizations and generate Hi-CIRRF while the Machine-SUIF platform is used to specialize hardware further and process Lo-CIRRF into VHDL. High level information such as loop-level bounds and strides and extraction of high-level hardware components such as controllers and address generators are extracted at the Hi-CIRRF level.

We have added new analysis and optimization passes to SUIF2 and Machine-SUIF that target reconfigurable devices. Specifically, taking the IR generated by SUIF's front end as input, our compiler detects and optimizes memory accesses. Our compiler also takes the IR generated by the Machine-SUIF front end as input and generates the data flow information. The array access pattern information, which is obtained through memory reference analysis, combined with the pipeline information, which is created during data flow generation, is fed into the controller generation pass to generate controllers in VHDL. We rely on commercial tools to synthesize the VHDL code generated by our compiler.

A. Available Passes

The ROCCC system's optimizations include loop normalization, invariant code motion, peeling, full and partial unrolling, fusion, tiling (a.k.a. blocking), strip mining, interchanging, skewing, unswitching, forward and induction variable substitution. ROCCC's procedure or global transformations include code hoisting and sinking, constant propagation and folding, elimination of algebraic identities, copy propagation, dead and unreachable code elimination, scalar renaming, reduction parallelization, division and multiplication by constant approximations. Finally, ROCCC's array transformations include array read after write (RAW) and write after write (WAW) elimination, scalar replacement, array renaming, and feedback reference elimination. With the exception of feedback reference elimination the rest of the transformations are also implemented in traditional/parallelizing compilers.

The designer is allowed to pick and choose from among all of the above transformations and specify the ordering and the granularity of each pass.

B. Hardware Generated

The hardware that we generated from the NAMD code consisted of 52 floating point operations. In addition, we had to generate a special floating point accumulator construct built up from one pipelined floating point adder. The entire datapath is pipelined (including the floating point units embedded in the datapath) and data is pushed into the datapath from a smart buffer [7], which handles the fetching of data from memory and manages the reuse of that data, each clock cycle.

The interface between the FPGA and the microprocessors is platform dependent, and the communication cost is independent of the hardware we generate.. We currently generate an interface with two of the SRAMs located on the SGI Reconfigurable Application-Specific Computing (RASC) blade, which have a bus width of 128 bits and a data throughput rate of 3.2 GB/s each. One SRAM is used to store input for use by the FPGA and one is used as a buffer for output that is sent back to the processors on the Altix machine. The actual transfer of memory to and from the host processors and FPGA is handled by the proprietary RASC API library and is only utilized by ROCCC.

C. High Level Code Restrictions

There are a few restrictions that the software designer must follow in order to have a section of code translated into hardware. None of these restrictions require any low level knowledge of the hardware being constructed or the platform being targeted .

The first restriction is that only loop nests are suitable to go through ROCCC and be converted into hardware. This follows our scheme that we are not converting entire programs into hardware, but only critical kernels of computation, which are going to be found in loop nests.

A second restriction we put on the software designer is that we only allow well structured memory accesses through arrays. Scalar variables can be passed in and out of the hardware without any changes to the C code, but accessing random memory through pointers is not allowed and arrays must be accessed in a linear fashion. This is because memory transfers to FPGAs are slow and there is no cache hierarchy to hide this latency. By streaming the data in bursts and storing only what we know will be reused through compile-time analysis we can maximize the amount of computation versus the memory transfer overhead.

Reuse of memory elements are automatically detected, but all accesses inside the loop nest to array elements must be based off of the loop indices. We further restrict the use of loop indices to be only used as loop indices and not modified inside the loop nest.

There are also a few minor restrictions to how the C is written, such as declarations and initializations must be separate in the C code with the exception of array initializations, but these do not fundamentally change the code in any way other than syntactically.

III. EXPERIMENTAL EVALUATION

A. SGI RASC RC100 Blade

The SGI Altix 4700 is a distributed shared memory multiprocessor machine that contains from 16 to 512 Intel Itanium 2 processors as a single system image implementing a cache-coherent system. The single system image is supported by the NUMAlink architecture, in which all processors and memory are tied together with special crossbar switches.

The Altix system supports the SGI Reconfigurable Application-Specific Computing (RASC) blade, which is our target. The RASC blade connects to the processing nodes of the Altix system through the NUMAlink interconnect system. This connection system makes the two Virtex 4 LX200 FPGA chips appear as part of the single system image visible from every processor and allows the FPGAs to have the same bandwidth (6.4 GB/s/FPGA) as the rest of the system. The RASC blade also supports the streaming of the same data to different algorithm blocks on different FPGAs concurrently [15].

The provided RASC API library handles the configuration of DMAs to transfer data between main memory and the RASC blade and handles memory transfers between the processors' main memory and the SRAMs on the FPGA without any additional hardware support necessary.

B. Critical Region

The most frequently executed section of code was previously identified as the code responsible for computing the forces between an atom and a list of paired nonbonded atoms [10]. This list of pairs differed for each atom and for each time frame, but the data was independent of the computations performed on them.

For one time step, each atom must calculate the electric and Van Der Waal forces between itself and every other atom. Depending on the distances between these atoms, NAMD performs different calculations. For example, if two atoms were far away, NAMD would perform a fast version of the Van Der Waal energy calculation that provides less accuracy than the calculations for two atoms that were closer.

There were 60 distinct ranges of distances that NAMD identified and performed different calculations for. All of the atoms in each range was calculated individually and collected into a list that was iterated through in a loop that performed the specific calculations required for that range. In our sample runs, most of these ranges were empty, so we took only the most frequently executed configuration and converted that to hardware.

We also created a hardware component that consisted of the union of all the different configurations and translated that into hardware as well, but the final size of the hardware component of this combined loop was too large to fit on the FPGA we were targeting, so this approach proved infeasible.

As an alternative to the union approach, we observed that we could take the distance calculation that was being performed in software and modify the hardware to include this. Instead of simply taking one loop into hardware, we could reengineer this section of NAMD into something more appropriate for hardware. We separated out each range's loop instance and created a hardware implementation for that one range that included the distance calculation. We would then have 60 different FPGAs, each running one range's set of calculations. We would then stream all atoms to all FPGAs simultaneously and each FPGA would either perform the calculations on that atom or ignore that atom based upon the distance that was calculated. In this paper, we only report on the most frequently executed of these ranges without the distance calculation and

are currently working on an implementation with distance calculation integrated in.

C. Transformations

Once the critical region was identified, we then performed several transformations on that region to change that region into C code that ROCCC could process. One of the restrictions of ROCCC is that memory accesses must follow a linear progression. The original code accessed memory in a non-linear pattern. Our transformations made memory accesses linear across the pairs of atoms and condensed many arrays into one general array. The resulting code is functionally equivalent to the original code and is valid C code that can be compiled by any standard C compiler.

The original code's nonlinear memory accesses were caused by a linear progression through a pairlist of atoms which determined the indices into another array. We transformed this code into three passes. The first pass fetched and stored the non-linear memory accesses into an array in a linear fashion. The second pass performed the computations on this data and stored the results in a separate array in the same linear order. The third pass processed the output array of the second pass and placed the data back into the original data arrays in the correct order as determined by the pairlist.

The first and third pass were partitioned to remain in the software while the second pass was translated into hardware on the FPGA. Further, in order to prevent the overhead we introduced as well as the overhead of activating the hardware, we only conditionally perform this section, having a check in the host code to call the hardware only when the number of iterations of the critical loop is sufficiently large enough to overcome the overhead. ROCCC determines which portion of the C code is going to be translated into hardware by identifying calls to the empty functions begin_hw() and end_hw(), so no additions to the C language were necessary to make these transformations.

In the linearization of the original code, many variables were fetched from non-linear locations in memory and transformed into arrays. The inclusion of all these arrays in parallel used more I/O pins than were available on the chip and resulted in unnecessary overhead. To alleviate this problem, we packed all of the necessary information for the critical loop into one array that was accessed linearly twelve variables at a time, resulting in 48 bytes per cycle for the single precision implementation. This data could be streamed in and managed by the smart buffer using fewer pins and not consuming all of the I/O resources of the LX200 chip.

The only dependencies in the loop were for variables that accumulate over all iterations. These dependencies can be eliminated by storing each iteration's value in an array and summing the array after all iterations are performed. In the hardware we generated, this was reduced to a single accumulator component based upon one floating point adder.

Due to the lack of dependencies, the loop can be further parallelized until either the memory bandwidth is saturated or the FPGA is filled by unrolling the loop. Currently, the memory bandwidth is the limiting factor.

Using ROCCC we generated a VHDL description to be placed on an FPGA and act as a co-processor that was explicitly called from the C code. The C code explicitly called library functions to pass data to and from the FPGA and turn the hardware computation engine on.

The VHDL implementation was targeted to the Xilinx LX200 FPGAs located on an SGI-RASC blade.

IV. Results

Our generated code was synthesized using Xilinx ISE 8.1 and optimized for area targeting the Xilinx LX200 FPGA. To handle the floating point operations we used the floating point cores from Xilinx Core Generator and integrated these into our main datapath pipeline. The cores are fully pipelined and caused a delay of 10 cycles for floating point addition and subtraction and 9 cycles for floating point multiplication. The critical loop contained 52 floating point arithmetic operations, consisting of additions, subtractions, multiplications, and divisions.

The original C code contained six floating point divisions. All six of these divisions were on constant values, and four of these divisions were of powers of two. We were able to optimize away these divisions. We modified the ROCCC system to identify divisions of floating point by constants in the original C code and replace them with more efficient hardware than a floating point divider. The divisions by a power of two were replaced with an integer subtracter that subtracted the exponent portion of the floating point number. The division by a constant that was not a power of two was replaced with a multiplication of the inverse of that number. The inverse was calculated at compile-time and retained the precision of the original constant. These optimizations reduced both the area and latency of the circuit dramatically.

These floating point optimizations were performed by our compiler and reflected only in the hardware we generated using our compiler and was not performed by gcc or icc for the software comparison. Further, opportunities for these optimizations were detected automatically and required no knowledge from the C programmer.

The variables in the original C code that acted as accumulators over all iterations of the loop had no corresponding floating point accumulator component. We created one built around a pipelined floating point adder and incorporated this into our datapath.

A. Single Precision Results

We place the entire critical loop on the chip using single precision floating point operations. This implementation took approximately 44% of the entire FPGA and had a clock speed of 149 MHz.

We compared the throughput of our circuit against software implementations running on an Itanium processor running at 1.6 GHz. We used both gcc 4.1.0 and icc 9.1 to compile our code and compare results.

	X	Y	Z	All Streams
Additions	13	14	14	21
Subtractions	4	4	4	8
Multiplications	10	11	10	17
Divisions	6	6	6	6
Total	33	35	34	52

TABLE I
THE NUMBER OF FLOATING POINT OPERATIONS FOR THE X STREAM, Y STREAM, Z STREAM, AND ALL STREAMS TOGETHER IN THE MOST CRITICAL LOOP

Implementation	Slices	Clock speed
Single Precision	39478 (44%)	149 MHz
Double Precision X	56262 (63%)	168 MHz
Double Precision Y	56354 (63%)	150 MHz
Double Precision Z	56352 (63%)	167 MHz

TABLE II
THE NUMBER OF SLICES AND CLOCK SPEED OF OUR HARDWARE IMPLEMENTATIONS ON THE LX200 FPGA

Clock Speed	Vs. Measured gcc	Vs. Measured icc
Single prec.	5355	808
Double prec. X	2635	145

TABLE III
CIRCUIT SPEEDUP VERSUS SOFTWARE IMPLEMENTATIONS. DOUBLE PRECISION REPRESENTS CLOCK RATE FOR MAXIMUM THROUGHPUT.

Using gcc 4.1.0 to compile the critical loop we observed a run time of 35.92 microseconds per loop iteration (which we measured using the user time field of the "time" command in Unix). Since our implementation is pipelined and generates one value per cycle, our rate is 6.707 nanoseconds per iteration, giving us a 5355x speedup over the observed run time.

Using icc 9.1, the run time in software went down to 5.424 microseconds per iteration, as icc is a heavily optimized compiler for the Itanium processor. Compared with our rate we still have a 808x speedup over the actual run time.

The NUMAlink architecture can provide 6.4 GB/s to an SRAM located on the RASC blade. The single precision implementation requires 5.856 GB/s, so using one SRAM we can fully supply our datapath with enough data to execute each cycle. With 52 floating point operations performed each clock cycle, this gives us performance of 7.7 GFLOPS.

B. Double Precision Results

When we attempted to synthesize the critical loop using double precision floating point operations, we quickly exceeded the capacity of the chip. We noticed, however, that the computation performed inside the critical loop consisted of three independent computational streams, one for the x vector, one for the y vector, and one for the z vector of each atom. Each of these vectors could be separated out from one another and implemented in double precision on a single FPGA. The number of floating point operations for each loop configuration is summarized Table 1.

In order to perform the complete double precision implementation, it is possible to have each vector implemented on a different FPGA executing in parallel. The SGI-RASC board supports sending the same data streams to multiple FPGAs simultaneously, resulting in only one pass through the input data feeding the entire implementation. The speedup would then be comparable to having the entire implementation on one FPGA and reading/writing to memory once, like the single precision implementation does.

Since the SGI-RASC board does not currently have three FPGAs, we separated out the x vector stream, y vector stream, and z vector stream from the most critical loop and used the same procedure as above to generate a double precision hardware implementation. Table 2 summarizes the result of placing these implementations on the LX200 chip. The double precision implementation of each individual stream stream only took up approximately 63% of the FPGA and had a clock speed of 168 MHz. A slice on the Virtex 4 architecture is defined to be two registers and two 4-input/2-output lookup tables.

Since all three vectors are approximately the same, we examined the x vector stream as a representative stream. Using gcc 4.1.0, the observed time per iteration that we saw for the double precision x vector stream was 31.36 microseconds, giving us a 5259x speedup over the actual.

Using icc 9.1, the observed time per iteration for the double precision x vector was 1.72 microseconds, giving our hardware a 289x speedup over the software implementation.

The double precision x vector stream, however, requires 16.099 GB/s (96 bytes per cycle to run constantly). The FPGA on the RASC blade can be configured to fetch a total of 48 bytes from three memories per cycle, so we would have to clock our datapath at half speed to provide enough data every clock cycle. For the double precision x vector stream, this results in 2.767 GFLOPS instead of the ideal 5.534 GFLOPS our hardware could support.

Table 3 summarizes our throughput compared to software implementations. The double precision x stream has results for running at the clock speed required by the memory bandwidth.

V. RELATED WORK

There are two commercial tools that are similar to ROCCC: Mitrion [18] and ImpulseC [13]. The Mitrion's approach is to instantiate a Mitrion Virtual Processor (MVP) on the FPGA, a massively parallel core that is programmed using the Mitrion-C language, a single assignment flavor of C. The Mitrion-

C language has a special loop statement using the key word foreach. In Mitrion-C, memory interfaces have to be defined by the user through particular key words such as memread and memwrite. The keyword wait provides timing information.

In [4] Gu reported an 88 time improvement on molecular dynamics code. The code that they used was not NAMD but rather their own hand coded implementation that used a fixed point implementation rather than a floating point implementation.

In [10], the authors put portions of NAMD on an FPGA. They modified the code to be 32-bit single precision floating point numbers as well as other restrictions. They also ported the work to MAP C, a language specific to one specific architecture whereas we create our datapath from C.

In [17], the authors wrote their own molecular dynamics program to compute nonbonded force interaction. They did not use NAMD and restricted their computations to single precision.

The MDGRAPE-3 project [12] is a specially built supercomputer dedicated to molecular dynamics problems. The MDGRAPE-3 is running a custom molecular dynamics program that has custom hardware support and is inflexible.

Celoxica introduced Handel-C [9] as a solution to the high level synthesis problem. Handel-C is a low level hardware/software construction language with C syntax and supports behavioral descriptions and uses a Communicating Sequential Process (CSP) communication model. The ROCCC system is designed to deal with portions of high level C code and does not require that the C code be written with hardware description hooks.

SystemC [23] is a library of C++ classes that specify hardware constructs and supports a subset that is synthesizable. Writing code in SystemC, however, is like writing code in a hardware description language. SystemC is designed to provide roughly the same expressive functionality of VHDL or Verilog and is suitable for designing software-hardware synchronized systems.

Streams-C [6] relies on the CSP model for communication between processes, both hardware and software. Streams-C can meet relatively high-density control requirements. The compiler generates both the pipelined datapath and the corresponding state machine to sequence the basic and pipeline blocks of the datapath. Streams-C does not handle two-dimensional arrays.

ImpulseC is the commercialization of Streams-C. Streams-C relies on the user explicitly partitioning the code into hardware and software processes and setting up communicating sequential processes based communication channels between them. Streams-C can meet relatively high-density control requirements. The compiler generates both the pipelined datapath and the corresponding state machine to sequence the basic and pipeline blocks of the datapath. ROCCC supports two-dimensional array access and performs input data reuse analysis on array accesses to reduce the memory bandwidth requirement.

SA-C [2] is a single-assignment, high-level, synthesizable language. Because of special constructs specific to SA-C (such as window constructs) and its functional nature, its compiler can easily exploit data reuse for window operations. SA-C does not support while-loops and requires uses to write algorithms in a single-assignment fashion.

The DEFACTO [5] system takes C as input and generates VHDL code. DEFACTO allows arbitrary memory accesses within the datapath. The memory channel architecture has its FIFO queue and a memory-scheduling controller.

GARP's [3] compiler is designed for the GARP reconfigurable architecture. The compiler generates a GARP configuration file instead of standard VHDL. GARP's memory interface consists of three configurable queues. The starting and ending addresses of the queues are configurable. The queues' reading actions can be stalled. GARP does not handle 2d arrays.

SPARK is another C to VHDL compiler [20]. Its optimizations include code motion, variable renaming, and loop unrolling. The transformations implemented in SPARK reduce the number of states in the controller FSM and the cycles on the longest path. SPARK does not perform optimizations on input data reuse.

Compared to previous efforts in translating C to HDLs, ROCCC's distinguishing features are its emphasis on maximizing parallelism via loop transformations, maximizing clock speed via pipelining, and minimizing area and memory accesses, a feature unique to ROCCC. ROCCC handles 2D arrays and can optimize memory accesses for window operations.

VI. CONCLUSIONS AND FUTURE WORK

We successfully compiled the critical region of NAMD from C into an efficient pipelined datapath to fit on the Virtex 4 LX200 FPGA on the SGI RASC blade with trivial modifications to the source code. The single precision implementation gave us a real speedup of over 808 times versus the best software implementation. We also were able to compile individual streams of computation for each of the different vectors in double precision, resulting in an improvement of over 145 times over the software implementation.

As future work we hope to be able to dynamically reconfigure small sections of the chip to accommodate the y and z portions of the computation, by swapping out the x computations and replacing them with the y and z computations. This will allow us to execute all of the computation in hardware and reuse the data stream without sending that stream to multiple FPGAs.

VII. ACKNOWLEDGMENTS

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.

REFERENCES

[1] S. A. Adcock and J.A. McCammon. Molecular Dynamics: Survey of Methods For Simulating The Activity Of Proteins. Chemical Reviews 106, 1589-1615, 2006.

[2] W. Bohm, J. Hammes, B. Draper, M. Cahwathe, C. Ross, R. Rinker, and W. Najjar. Mapping a Single Assignment Programming Language to Reconfigurable Systems. Supercomputing, 21:117-130, 2002.

[3] T. J. Callahan, J. R. Hauser, J. Wawrzynek. The Garp Architecture and C Compiler. IEEE Computer, April 2000.

[4] Y. Gu, T. VanCourt, and M. Herbordt. Accelerating Molecular Dynamics Simulations with Configurable Circuits. Computers and Digital Techniques, Volume 153, Issue 3, May 2006, pages 189-195.

[5] P. Diniz, M. Hall Park, J. Park, B. So and H. Ziegler. Bridging the Gap between Compilation and Synthesis in the DEFACTO System. Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing Synthesis (LCPC'01), Oct. 2001.

[6] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Lalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In IEEE Symp. on FPGAs for Custom Computing Machines (FCCM), 2000.

[7] Z. Guo, A. B. Buyukkurt, and W. Najjar. Input Data Reuse in Compiling Window Operations Onto Reconfigurable Hardware. ACM Symposium on Languages, Compilers, and Tools for Embedded Systems (LCTES), June 2004.

[8] Z. Guo and W. Najjar. A Compiler Intermediate Representation for Reconfigurable Fabrics. International Conference on Field Programmable Logic and Applications, August 2006.

[9] Handel-C Language Overview. Celoxica, Inc. http://www.celoxica.com. 2004.

[10] Volodymyr Kindratenko and David Pointer. A case study in porting a production scientific supercomputing application to a reconfigurable computer. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2006.

[11] Machine-SUIF. http://www.eecs.harvard.edu/hube/research/machsuif.html, 2004

[12] The MDGRAPE-3 computer. http://mdgrape.gsc.riken.jp

[13] David Pellerein and Scott Thibault. Practical FPGA Programming in C. Prentice Hall, 2005.

[14] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kale, and Klaus Schulten. Scalable molecular dynamics with NAMD. Journal of Computational Chemistry, 26:1781-1802, 2005.

[15] Reconfigurable Application-Specific Computing User's Guide. SGI, 2006.

[16] L. Saiz and M. L Klein. Computer Simulation Studies of Model Biological Membranes. Accounts of Chemical Research 35, 422-429, 2002.

[17] R. Scrofano and V. Prasanna. Preliminary Investigation of Advanced Electostatics in Molecular Dynamics on Reconfigurable Computers. International Conference for High Performance Computing, Networking, Storage and Analysis, 2006.

[18] SGI Reconfigurable Application Specific Computing: Accelerating Production Workflows. White paper, SGI.

[19] M. D. Smith and G. Holloway. An introduction to Machine SUIF and its portable libraries for analysis and optimization. Division of Engineering and Applied Sci-ences, Harvard University.

[20] SPARK project. http://mesl.ucsd.edu/spark/, 2005.

[21] Dave Strenski. FPGA Floating Point Performance – a pencil and paper evaluation. HPC Wire, January 12, 2007, http://www.hpcwire.com/hpc/1195762.html

[22] SUIF Compiler System. http://suif.stanford.edu, 2004

[23] SystemC Consortium. http://www.systemc.org, 2005.