International Conference on Field Programmable Logic and Applications, Heidelberg, Germany, September 2008 (FPL 2008)

COMPILED HARDWARE ACCELERATION OF MOLECULAR DYNAMICS CODE

Jason Villarreal and Walid A. Najjar

Department of Computer Science and Engineering
University of California, Riverside
{*villarre, najjar*}@*cs.ucr.edu*

ABSTRACT

The objective of Molecular Dynamics (MD) simulations is to determine the shape of a molecule in a given biomolecular environment. These simulations are very demanding computationally, where simulations of a few milliseconds can take days or months depending on the number of atoms involved. Therefore, MD simulations are a prime candidate for FPGA-based code acceleration. We have investigated the possible acceleration of the commonly used MD program NAMD. This code is highly optimized for software based execution and does not benefit from an FPGA-based acceleration as written. We have therefore developed a modified version, based on the calculations NAMD performs, that streams a set of data through a highly pipelined circuit on the FPGA. We have used the ROCCC compiler toolset to generate the circuit and implemented it on the SGI Altix 4700 fitted with a RASC RC100 blade.

1. INTRODUCTION

The objective of Molecular Dynamics (MD) simulations is to determine the shape of a molecule in a given biomolecular environment. MD simulations are based upon 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. The force field is based on a potential energy function that accounts for all pairwise interactions of a biomolecular system. This 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. There is a great need to increase both the size of the simulated systems and the accessible timescales. The size should 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.

Dedicated hardware solutions are being developed [7], but such solutions are extremely expensive and impractical for many researchers. Companies and researchers would like to have the flexibility of a software solution that allows researchers the ability to tune the accuraccy versus time and perform different approximations.

Field programmable gate arrays (FPGAs) provide a middle ground between flexible software and dedicated hardware. Specific parallel portions of reliable molecular dynamics software packages could be implemented on FPGAs providing the speedup of hardware without sacrificing the flexibility in the software package.

FPGAs are efficient when massive fine-grained parallelism can be exploited and large data driven pipelines can be created. Implementations on FPGAs have all the advantages of custom hardware, such as creating a datapath with a nonstandard bitwidth without any overhead, but run at a much slower clock rate than a standard processor.

Recently, Strenski [12] has shown that FPGAs are now capable of outperforming a dual core 2.5 GHz Opteron on double precision floating point operations. FPGAs performance, however, is limited by the memory bandwidth and requires both streams of data for input and output in order to be effective.

The difficulty in using FPGAs is in programming the system and translating sequential computing systems into an equivalent and efficient spatial implementation. This difficulty comes from the very different programming and optimization strategies for a sequential processor. Most high performance systems, such as the molecular dynamics simulations we examined, are heavily optimized for a particular microprocessor architecture, such as supercomputer systems, and not suited for a completely different paradigm. The memory model in these software solutions assumes communication through a shared memory and has heavy use of random accesses to memory which are hidden by the cache architecture. FPGAs have no caching architecture and accesses to memory are extremely expensive, making the optimized software not amenable to a hardware implementation as written. We overcome the difficulty of programming FPGAs by automatically compiling hardware from sections of code in the original systems using the ROCCC compiler [1]. The ROCCC compiler allows us to change a small portion of the original implementation in C and generate a hardware co-processor, with the rest of the software remaining unchanged.

NAMD [8] is a popular MD software package. We have studied this package extensively with the objective of accelerating their most frequently executed code segment on FPGAs. In this paper, we report that NAMD is so finely optimized for a sequential software execution that no speedup can be achieved from an FPGA-based acceleration of the most frequently executed loop. We offer an extensive analysis of the code as well as a quantitative evaluation of the obstacles to acceleration. We have developed our own implementation, based on the NAMD approach, that we have compiled with ROCCC and implemented on a Xilinx Virtex 4 LX200 on the SGI Altix RASC RC100 blade. Our results show that the main bottleneck is the data bandwidth in and out of the FPGA. This bottleneck is shown to reduce the throughput by a factor of three.

Acceleration of MD algorithms has been looked at previously in [9], [11], [10], and others. This papers differs from the previous works in several aspects. We report on an extensive analysis of why the acceleration would not succeed on standard software codes. Based on this analysis we have developed a code structure that is more suitable for FPGA-based code acceleration. We use ROCCC, a C to VHDL compiling tool, to generate the hardware mapped to the FPGAs. We analyze the I/O bottleneck on the FPGA in the SGI RASC RC 100 blade.

2. MOLECULAR DYNAMICS

In this section, we describe how the basis of molecular dymamics is reflected in NAMD and how the most frequently executed region is structured. We also explore the specific implementation details chosen in NAMD.

2.1. Basis of Molecular Dynamics

The main algorithm in molecular dynamics simulations is the calculation and summation of all forces between all atoms at each timestep. A typical timestep is one femtosecond (10^{-15} second). The calculations performed in molecular dynamics break down into two main categories, *bonded* forces and *nonbonded* forces. Bonded forces refer to the forces between all the atoms in the same molecule while nonbonded forces, which are much more extensive, refer to the set of forces between atoms in different molecules or different sections of large molecules.

The nonbonded forces are the most computationally intensive section of molecular dynamics simulations [9] and are based upon several physical formulations. These formulations include the Lennard-Jones Potential, which calculates both the attractive Van Der Waal force at long distances and the repulsive force at close distances, and the Coulombic forces, which calculate the electrostatic attraction and repulsion between two atoms.

The Lennard-Jones potential is shown in equation 1. σ and ϵ are both constants based upon the types of the two atoms being compared while r represents the distance between them.

$$u(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{1}$$

Equation 2 shows the equation necessary to compute the Coulombic forces. In this equation, ϵ is again a constant based upon the types of atoms (different from the ϵ in Equation 1), r is the distance between the atoms, and q_1 and q_2 are the charges of the respective atoms.

$$F = \frac{(q_1 * q_2)}{4\epsilon\pi * r^2} \tag{2}$$

The values of ϵ and σ are typically calculated before compilation and stored in a table which is only indexed during the execution of the program. The value r must be computed for every pair of atoms at every timestep based upon the x, y, and z coordinate of the atoms.

All of these values are real numbers and are typically represented in software using single precision or double precision floating point numbers. The calculation of these forces, then, requires many floating point computations which must be scheduled in limited hardware on a sequential processor.

In software, in order to conserve time, many computations are not performed. The long-range forces are typically approximated using methods such as Ewald summation or the smooth particle mesh Ewald summation, both of which aggregate collections of atoms over long distances into one entity and assume a regular structure. These methods trade off accuracy for speed, but do so on the long range interactions, which contribute very little to the total forces. In the C code, these formulations translate into different floating point calculations that replace the basic Coulomb equations.

Another time saving optimization is the use of distance based cutoffs in software. Atoms beyond a user defined cutoff are approximated using one of the methods described above, whereas atoms that are closer will have the full computation performed.

2.2. NAMD

NAMD (NAnoscale Molecular Dynamics) [8] is a popular molecular dynamics simulation implementation. The structure of NAMD is highly optimized for execution on supercomputers and multiprocessor systems.

The general execution flow of NAMD is shown in Figure 1. The main iteration per timestep involves first picking an atom and then arranging all other atoms into lists dependant on the distance. There are 60 distinct ranges that an atom can fall into, and different calculations are performed based upon the range. Each list is determined before the loop body that calculates the forces and used as input to that loop body. Most of the lists are not filled and one loop instance accounts for approximately 80% of the nonbonded force calculation time.

While the number of floating point operations in each loop

```
for all timesteps
{   for A in all atoms
  {
    for C in all cutoff values
    {
      Create list L of all atoms within C ;
      for all B in L
        Compute forces between A and B ;
    }
  }
}
```

Fig. 1. NAMD Nonbonded Computation Execution. Cutoffs are user defined or precalculated.

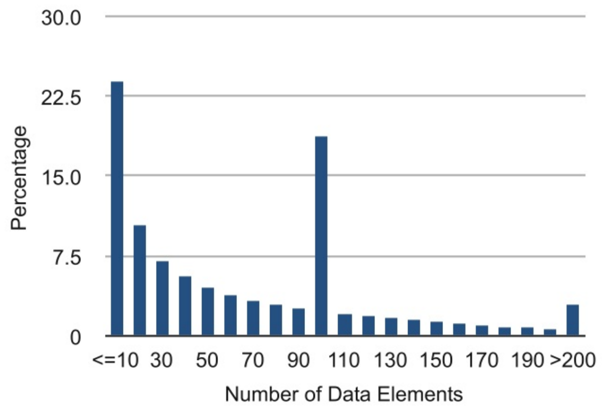

Fig. 2. Distribution of data set size per invocation of our hardware accelerator. Data sets of size zero are excluded. Over 83% of all calls are with less than 100 data elements.

body varied from 39 to over 135, the one loop instance that accounted for the most computation time contained 52 floating point operations and calculated the Lennard-Jones Potential and Coulombic forces as described in the previous section. We previously converted this loop instance into hardware by using ROCCC in [14].

3. FPGA ACCELERATION OF MD

3.1. Experience with NAMD

We replaced the original frequently executed loop of NAMD in the software with a call to our hardware. We ran this combination on an SGI Altix 4700 with RASC blade, but the results were not as expected. The software with a hardware co-processor did not accelerate the program, but actually ran 16,200X slower than the original software implementation on a single CPU.

While exploring the cause of this slowdown, we noted that each hardware activation takes approximately 2 milliseconds of setup and memory transfer. The circuit itself was running at 100 MHz, but required 15 floating point values per loop iteration. The available bandwidth into the FPGA was 128 bits per cycle, so a total of 4 cycles was required before the start of the pipeline could be filled. With this information, we determined that we would need 50,000 data elements before the 2 millisecond overhead accounted for half of the time accessing the hardware. The RASC blade is capable of running multiple clocks, however, and should be able to retrieve data with a clock rate of 200 MHz, making certain the co-processor is fed twice as fast. With a dual clock scheme, we would need to transfer 25,000 data elements before the overhead accounted for half of the time accessing the hardware.

After analysing the data transfer pattern, we determined that even though this section of code was the most frequently executed, there were a very low number of iterations per execution. As shown in Figure 2, 24% of all calls to the hardware sent less than 10 data elements and over 83% sent less than 100 elements. These numbers were observed when running the apoa dataset, a representative molecular dynamics dataset, and correlate directly with the amount of data sent to the hardware. Note that this does not include the amount of times the hardware was called with zero data elements, in those cases we did not transfer control to the hardware. The total number of invocations with any data elements is only approximately 29%.

3.2. A Streamed Data Implementation

A kernel that performs the same calculations as the popular software described above, but benefits from hardware acceleration would be ideal. In order for acceleration to occur, the data must be streamed into the FPGA. In order to stream the data, we bring together the distance calculations as well as the LJ potential and Coulomb forces so we don't have to perform any cutoffs calculations.

In hardware, the most efficient way of performing the force calculations and summations is not clear. The massive amount of parallelism provided by hardware may make it more benefical to always calculate the full forces every timestep than to use an approximation method consisting of more hardware components. Our hardware implementation is based upon the base Coulomb forces and LJ potential, which are the most frequently executed instructions in NAMD.

We found that as written, current popular implementations of molecular dynamics code can not be accelerated through the use of a hardware co-processor that simply replaces the most frequently executed region in software. Based off of this knowledge, we implemented a version of the common calculations that was amenable to hardware acceleration. The flow of our implementation is shown in Figure 3.

The code we wrote stores the X, Y, Z coordinates as well as the current charge of an atom in a structure and all atom instances are stored in an array of these structures. The calculations performed in the hardware operate on pairs of atoms

```
for each timestep
{
  Stream every pair of atoms to the FPGA ;
  // In Hardware
  Compute LJ potential and Coulombic forces on pairs;
  // In Software
  Accumulate all forces over original atom list ;
}
```

Fig. 3. Execution flow of streamlined nonbonded computation

and the pipeline requires two atoms per clock cycle. In addition to the data stored in each atom, the constants ϵ and σ need to be determined. Currently, we look these up in a table in software and pass them into the hardware along with the atom data, but this could also be performed in hardware.

In the first step, our code creates a stream of atom pairs by pairing and packing all atoms together. In contrast to software, we do not pass pointers around, but instead create a reorganized array that acts as a stream of atom pairs that we send to the hardware.

Because we are not working on the original atom data in place, but rather many copies of the original atoms, this data needs to be accumulated, which is handled by our third pass. Traversing the output stream in the same order the original stream was assembled, we update the original atom list in place. Once the update is done, we can move on to the next timestep and repeat this process.

3.3. Hardware Implementation

The software implementation we created was passed through ROCCC and generated hardware. We only took the second section of our code, the floating point computations, into hardware. The first and third sections of our code were identical in the software and hardware-accelerated versions. We used single precision floating point numbers in both software and hardware.

Table 1 summarizes the structure of the hardware accelerator that was created. The accelerator fits in 31,292 slices, which corresponds to 35% of the LX200 FPGA located on the RASC blade. The hardware consists of 27 floating point operations in a 125 stage pipeline, with the pipeline expecting 12 floating point numbers each iteration. The clock speed we achieved was 100 MHz.

We are able to transfer large amounts of memory back and forth from the hardware because we make no distinction between different cutoffs and create a very large stream of data. For instance, if we have 1024 distinct atoms, we send 48 MB worth of data to the hardware each timestep.

3.4. Results

Table 2 shows the results of running the hardware implementation on the SGI Altix 4700 with RASC blade versus running the software implementation. The RASC blade has two Virtex LX200 FPGAs with memory interfaces that have bandwidth of 128 bits per cycle. Our hardware required 12 floating point values per pipeline stage, so the data reads must be split up into three read cycles. The first row in Table 2 shows the amount of time it took to process various amounts of data using this implementation in seconds.

Using a dual clock scheme on the RASC blade, where one clock reads data from memory at 200 MHz and one clock controls our co-processor at 100 MHZ, data is provided every 2 out of 3 cycles. The second row of Table 2 represents the time for such a scheme to process the same amount of data.

The memory interface is out of our control, however, and the pipeline we create can handle a full 12 floating point numbers per cycle if they are provided. The third row of Table 2 shows the amount of time necessary to process the same amount of data if 12 floating point numbers were provided every clock cycle.

We compared this time with two different Xeon processors, one running at 3.0 GHz with a 4MB Cache and one running at 2.8 GHz Xeon with 512KB of Cache. When we have full bandwidth, our hardware is able to outperform software execution runs on both a modest and higher end processor, and our current hardware implementation on the RASC blade outperforms the software execution on a modest Xeon processor. Due to performance restrictions, we were unable to run the software on the lesser system for 768 MB.

# Of F.P. Ops	Pipeline Stages	Slices	Clock Speed
27	125	31292	100 MHz

Table 1. Hardware accelerator details. The FPGA circuit can sustain 2.7 GFLOPS assuming full bandwidth.

Input Size (MB)	48	96	192	768
FPGA on RASC blade	0.151s	0.243s	0.382s	1.278s
FPGA Dual Clock Scheme	0.1s	0.162s	0.254s	0.852s
FPGA Full Bandwidth	0.05s	0.081s	0.127s	0.426s
Xeon 3.0 GHz (4MB cache)	0.061s	0.132s	0.235s	0.931s
Xeon 2.8 GHz (512KB cache)	0.664s	2.269s	4.739s	n/a

Table 2. Execution time with varying amount of data. The first row is the measured time on the RASC blade at 100 MHz. The second row is on the RASC blade with separate clock domain for the memory interface running at 200 MHz. Row three is the FPGA implementation assuming sufficient bandwidth to accommodate all the data needed per iteration. The last two rows are the software implementation on a high end and a desktop CPU.

4. RELATED WORK

Accelerating molecular dynamics code has been examined in detail by several researchers.

In [9], an approach very similar to ours was taken. The difference between our approach and theirs is that our profiling went into more depth to identify the loop body that was most frequently executed rather than the function, and with this information we didn't rewrite entire functions, but only that loop body. They also used Map C, which is specific to a particular architecture, whereas our high level language is C and the hardare generated could be used in many platforms.

In [10], the authors take a particular set of equations, the smooth particle mesh Ewald summation technique, and create hardware at a low level by hand. They do not include the Lennard-Jones equations, which we observed to be part of the most frequently executed section in NAMD.

The authors of [11] create a pipeline similar to the one that we compile for just the Lennard-Jones potential. Again, the pipeline they create is done by hand. Included in their pipeline is a square root operation. While the square root operation is mathematically valid, code implementations like NAMD skip this step and the most frequently executed section of code does not contain this operation.

The MDGRAPE-3 project [7] 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.

5. CONCLUSIONS

Molecular dynamics is an important class of high-performance computing applications that can potentially be a good candidate for FPGA-based code acceleration. We report on an extensive effort to accelerate MD computations. We have explored a commonly used MD software package. Our first result is that simply replacing the most frequently executed section of code with a call to an FPGA-based hardware accelerator resulted in performance much worse than the software version. We have investigated the causes of this slowdown and show that the structure of the software optimized code defeats the acceleration advantages of the FPGA. Using this knowledge, we rewrote the most computationally intensive section of code to support the streaming of data to the FPGA accelerator. We compiled the C code to VHDL using the ROCCC compiler and mapped the hardware on the Xillinx Virtex 4 LX200 on the SGI RASC RC100 blade. The resulting hardware runs at 100 MHz, consists of 125 pipeline stages with 27 floating point operations and occupied about a third of the FPGA. Our results show that a speedup, over a software execution, can be obtained for large amounts of data and when the bandwidth in and out of the FPGA is large enough to support the initiation of one iteration per cycle. We also report on the performance of the software version on two CPUs: one desktop class and the other high-end with 4 MB cache. The difference in execution time varies by an order of magnitude.

6. REFERENCES

[1] B. Buyukkurt, Z. Guo, and W. Najjar. Impact of Loop Unrolling on Area, Throughput and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs. International Workshop On Applied Reconfigurable Computing, Delft, The Netherlands, March 2006.

[2] The GROMACS MD System. www.gromacs.org, 2008.

[3] Y. Gu and M. Herbordt. FPGA-Based Multigrid Computation for Molecular Dynamics Simulations. FCCM 2007.

[4] Y. Gu, T. VanCourt and M. Herbordt. Improved Interpolation and System Integration for FPGA-based Molecular Dynamics Simulations. International Conference on Field Programmable Logic and Applications (FPL), 2006.

[5] T. Matthey, et al. ProtoMol, An Object-Oriented Framework for Prototyping Novel Algorithms for Molecular Dynamics. ACM Transactions on Mathematical Software, vol. 30, pp. 237-265, 2004.

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

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

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

[9] V. Kindratenko and D. Pointer. A case study in porting a production scientic supercomputing application to a recongurable computer. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2006.

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

[11] R. Scrofano and V. Prasanna. Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware. Engineering of Reconfigurable Systems and Algorithms, 2004.

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

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

[14] J. Villarreal, J. Cortes, and W. Najjar. Compiled Code Acceleration of NAMD on FPGAs. Reconfigurable Systems Summer Institute (RSSI), 2007.