

Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware

Ronald Scrofano

Department of Computer Science
University of Southern California
Los Angeles, CA

Viktor K. Prasanna

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA

Abstract—Technological advances have made FPGAs an attractive platform for the acceleration of complex scientific applications. These applications demand high performance and high-precision floating point arithmetic. In this paper, we present a design for calculating the Lennard-Jones potential and force as is done in molecular dynamics simulations. This architecture employs IEEE 754 double precision floating point units, including a square root unit developed for this kernel. The design presented is a modular, very deeply pipelined architecture that exploits the fine-grained parallelism of the calculations. With the Xilinx Virtex-II Pro as a target device, an implementation using two pipelines operating in parallel achieves 3.9 GFLOPS.

Index Terms—molecular dynamics, square root, supercomputing

I. INTRODUCTION

Scientific computing applications are very computationally intensive and thus good candidates for hardware acceleration. However, these applications also often require high precision, floating point arithmetic. Until recently, reconfigurable computing devices, such as FPGAs, did not have the logic resources necessary to implement even a single floating point unit, let alone the multiple floating point units that would be needed to effectively accelerate scientific applications. However, recent advances in FPGA technology, such as drastic increases in the amount of logic resources and the inclusion of embedded hardware multipliers, have made FPGAs a viable platform for accelerating scientific computing applications.

In this paper, we investigate the acceleration of Lennard-Jones force and potential calculations for molecular dynamics (MD) simulations. Force calculation is the most computationally intensive part of an MD simulation, so its acceleration will lead to the greatest overall speedup of a simulation. Force and potential calculations are also difficult because they require many floating point operations. Lennard-Jones force and potential calculations are relatively simple and they still require 16 floating point operations, including two divisions and a square root. However, there are also many operations that can be performed in parallel and the whole calculation can be pipelined. In this paper, we present such a parallel, pipelined architecture for the force and potential calculation and analyze its performance. The pipelined design accepts one input per cycle and produces two results—a potential and a force—per cycle. This architecture performs all computations with floating point units that comply with IEEE standard 754

for double precision floating point arithmetic. While we were able to make use of existing adders, multipliers, and dividers, we developed our own square root unit that meets the needs of this kernel.

In the next section, we give background information about MD simulations. In Section III, we describe existing work that is related to ours. In Section IV, we describe our design for the calculation of Lennard-Jones forces and potentials. Section V describes the pipelined floating point units in our design, especially the square root unit developed specifically for this design. In Section VI, we describe our design's performance and compare it with the performance of the same calculations on a general purpose processor. Finally, Section VII concludes the work and presents areas for future study.

II. BACKGROUND

MD is a simulation technique that is used in many different fields, from materials science to pharmaceuticals. Here, we give some background information about MD. In so doing, we reference [1] and [2].

MD is a technique for simulating the movements of particles in a system over time. Each of the particles i in the system has an initial position $\vec{r}_i(t_0)$ and an initial velocity $\vec{v}_i(t_0)$ at time $t = t_0$. Given the number of particles in the system, the initial temperature, the initial density, and the volume of the system, the MD simulation determines the *trajectory* of the system from time $t = t_0$ to some later time $t = t_f$. The trajectory is basically the positions of the particles in the system as time advances. The simulation also keeps track of properties of the system such as total energy, potential energy, and kinetic energy.

In order to compute the system's trajectory, the positions of all the molecules at time $(t + \Delta t)$ are calculated based on the positions of all the molecules at time t , where Δt is a small time interval. There are many methods for calculating new positions; the most popular is the velocity Verlet algorithm. The steps in this algorithm are

- 1) calculate the velocity of each molecule at time $(t + \frac{\Delta t}{2})$ based on the acceleration of each molecule at time t ;
- 2) using the newly calculated velocities, calculate the molecular positions at $(t + \Delta t)$;
- 3) based on the new positions, calculate accelerations $a_i(t + \Delta t)$;

- 4) based on these newly calculated accelerations, calculate the velocities at time $(t + \Delta t)$.

The literature is in agreement that the most time-consuming step of an MD simulation is step 3. The acceleration is computed using Newton’s Second Law of Motion: $\vec{F}_i(t) = m_i \vec{a}_i(t)$, where m_i is the mass of particle i and $\vec{F}_i(t)$ is the force acting on particle i at time t . There are three general types of forces: van der Waals (intermolecular), intramolecular, and electrostatic. The force studied in this paper is a van der Waals force. In this paper, we also will assume a system in which all the molecules are the same.

We develop an architecture for calculating the scalar component of the Lennard-Jones force. The Lennard-Jones force is one of the simplest types of van der Waals forces to calculate. It arises due to the potential between two molecules i and j that are a distance r_{ij} apart, as shown in Equation 1. This potential is also calculated by our pipeline. Equation 2 gives the equation for the Lennard-Jones force.

$$u(r_{ij}) = 4 \left(\frac{1}{r_{ij}^{12}} - \frac{1}{r_{ij}^6} \right) - U_c + dU_c (r_{ij} - r_c) \quad (1)$$

$$\vec{F}_i(t) = \sum_{j \neq i} \vec{r}_{ij} \frac{48}{r_{ij}^2} \left(\frac{1}{r_{ij}^{12}} - \frac{1}{2r_{ij}^6} \right) + \frac{dU_c}{r_{ij}} \quad (2)$$

where $\vec{r}_{ij} = \vec{r}_i(t) - \vec{r}_j(t)$ and r_{ij} is the magnitude of \vec{r}_{ij} . Both of the equations are in terms of normalized units. Normalized units are dimensionless quantities that allow the simulation to use larger numbers so that underflow is much less likely. r_c in Equation 1 is the *cutoff distance*. The potential between molecule i and molecule j and the forces that these molecules exert on one another are only calculated if the distance between the two molecules is less than some cutoff distance r_c . The use of the cutoff distance leads to discontinuities; the simulation must “shift” the potential and the force in order to compensate for these discontinuities. U_c and dU_c are constants in the simulation, related to r_c , used in this shift.

III. RELATED WORK

A. Hardware Acceleration of Molecular Dynamics

We begin by studying previous efforts in hardware acceleration of MD simulations. The most prominent example of hardware acceleration for MD simulations is the MD-GRAPe project [3].

MD-GRAPe (and its successors, MD-GRAPe-2 and MD-GRAPe-3) is an ASIC chip that is capable of calculating a wide variety of forces and potentials, including electrostatic and Lennard-Jones forces and potentials. The MD-GRAPe chip uses a 1024-piece, fourth order polynomial to approximate the calculation of the force or potential. The coefficients in this polynomial determine which force or potential is calculated. The chip also keeps a sum of the forces on or potential due to each particle. MD-GRAPe only accelerates the force and potential calculation, leaving the rest of the MD simulation to a host processor. We follow a similar

computational model in our work. Between our work and MD-GRAPe, there are, however, differences that go beyond our use of reconfigurable hardware instead of an ASIC. Rather than use a polynomial approximation, we calculate the force and potential using their respective equations. Further, our entire design uses double precision floating point numbers, whereas MD-GRAPe employs a combination of floating and fixed point numbers.

An FPGA-based system for accelerating MD simulations is presented in [4]. In this system, the velocity and position calculations of the Velocity Verlet algorithm have been mapped to an FPGA. The calculations are done with IEEE 754 32-bit floating point arithmetic. The performance for two types of platform FPGAs is presented. The implementation on an Altera Stratix achieves 5.69 GFLOPS while the implementation on a Xilinx Virtex-II Pro achieves 4.47 GFLOPS. This effort is complimentary to ours in that it accelerates a different portion of MD simulations than we are targeting. It is also one of the early examples of using reconfigurable hardware with floating point units for scientific computations.

B. Reconfigurable Computing for Scientific Applications

An earlier example of the use of reconfigurable hardware in the acceleration of scientific computations is PROGRAPE-1 (PROgrammable GRAPE-1) [5]. PROGRAPE-1 is an FPGA-based, programmable, multipurpose computer for many-body simulations. It is especially targeted for scientific techniques that have many different algorithmic implementations and for which new algorithms are still being developed. The example cited in [5] is the SPH force calculation algorithm. PROGRAPE-1 is implemented with two Altera EPF10K100 FPGAs, each of which contains about 100,000 gates. With this now somewhat outdated technology, the developers were able to achieve a throughput equivalent to 0.96 GFLOPS for a gravitational interaction calculation. Note that they did not implement IEEE standard floating point units due to size limitations of the FPGAs.

[6] is recent work that uses reconfigurable hardware to accelerate three kernels in computational fluid dynamics: the Euler, Viscous, and Smoothing algorithms. The design employs Nallatech boards comprised of multiple FPGAs. Using single precision floating point numbers, the authors achieve 10.2, 23.2, and 7.0 GFLOPS, respectively, for the kernels listed above.

C. FPGA Floating Point Units

[7] describes a parameterized floating point library and uses this library to implement the SPH force calculation on a Virtex-II FPGA. The library has addition, multiplication, division, and square root units. For each floating point unit, performance up to 24 bits of precision is listed. The FPGA implementation of SPH using the floating point units with 24-bit precision was able to achieve a performance of 3.9 GFLOPS.

[8] describes adders and multipliers that follow the IEEE 32- and 64-bit formats, as well as a 48-bit format. [9] uses these

adders and multipliers in a high performance floating point matrix multiplication design. This design is able to achieve a performance of 8.3 GFLOPS. In the implementation of our design, we employ the same 64-bit adders and multipliers as are used in this work and we will describe these units more thoroughly in Section V.

In [10], a parameterized floating point library is presented. The library contains addition and multiplication units as well as units for the conversion from (to) floating point format to (from) fixed point format.

Clearly, there has been much work that is related to ours. However, to the best of our knowledge, we are the first to use IEEE 754 double precision floating point units on an FPGA and apply these units to the acceleration of MD simulations.

IV. DESIGN

In this section, we describe our design for computing the Lennard-Jones potentials and forces. We develop a deeply pipelined design based on Equations 1 and 2. In the context of an MD simulation, our design would accelerate the force and potential calculations that occur immediately following the determination that the distance between two molecules is less than the cutoff distance r_c . The output of our design would then be used as part of a calculation of the total potential energy of the system and a calculation of the vector components of the force acting on molecules i and j . Specifically, our design calculates

$$u = 4 \left(\frac{1}{r^6} \right) \left(\frac{1}{r^6} - 1 \right) + (dU_c r_c - U_c) + dU_c r \quad (3)$$

and

$$F = 48 \left(\frac{1}{r^2} \right) \left(\frac{1}{r^6} \right) \left(\frac{1}{r^6} - 0.5 \right) + \frac{dU_c}{r} \quad (4)$$

Equation 3 is simply Equation 1 rearranged and with $r = r_{ij}$. Note that $(dU_c r_c - U_c)$ is a constant. Equation 4 is a rearrangement of Equation 2 for molecules i and j where $r = r_{ij}$. Also, Equation 4 does not contain the multiplication by vector \vec{r}_{ij} because this multiplication will not be performed in our pipeline.

In practice, rather than compare r with r_c , MD simulations compare r^2 with r_c^2 . Thus, we assume that the input to our pipeline will be r^2 . We design the pipeline such that it accepts new data on every clock cycle and outputs both u and F on every clock cycle once the pipeline is full.

Figure 1 shows the dataflow graph for our pipeline. The horizontal, dashed lines show the places at which it appears natural to break the design into pipeline stages. Anywhere that an arrow crosses a stage without going through a computation, the data represented by that arrow will have to be stored for that stage.

We see from the data flow diagram that each set of force and potential calculations requires 16 floating point operations, including two divisions and one square root. Each of these computations will be performed by *pipelined* floating point units (see Section V). Thus each stage denoted in Figure 1 will actually consist of many pipeline stages. Moreover, each

type of floating point unit may require a different number of stages. Thus, the graph does not break up as cleanly as is shown in Figure 1.

We also note that the square root can be performed at the very beginning of the pipeline but its results are not needed until almost the end of the pipeline. Thus, it would be acceptable for the square rooter to have many more stages than the other operations. To put it more formally, let s , a , m , and d be the number of pipeline stages in the square rooter, adder/subtractor, multiplier, and divider, respectively. Then as long as

$$s + \max(m, d) \leq d + \max((4m + 2a), (3m + 3a))$$

the square root will finish its computations early enough so that it does not add extra latency to the force and potential calculation pipeline. We take this fact into account in designing the floating point square root unit.

Also, notice from the data flow graph that there are many instances in which more than one computation is happening simultaneously in the same stage. By exploiting this fact, the force and potential calculation pipeline takes advantage of the fine grained parallelism present in the calculations.

There are several benefits to this force and potential calculation pipeline. First, it produces two results, a force and a potential, for every one input that it receives. Second, there is no control overhead. Any synchronization due to the varying latencies in floating point units is accomplished through the addition of delay stages. Additionally, there are no data hazards in the pipeline, so it never needs to stall or to forward data. Thus, early stages never need to communicate with later ones. This fact leads to the third benefit: the design is very modular. Each of the floating point units is self-contained. The force and potential pipeline could be split over multiple chips, if necessary. Further, multiple pipelines could be implemented without any dependencies between the two pipelines, thus taking advantage of coarse-grained parallelism.

V. FLOATING POINT UNITS

As seen in the dataflow diagram, the force and potential calculations require four different types of floating point operations: addition/subtraction, multiplication, division, and square root. We briefly describe the floating point adder, multiplier, and divider used in this design. We then describe in greater detail the floating point square root unit, which was developed specifically for our implementation of the force and potential calculations.

For our floating point adders and multipliers, we chose the 64-bit adder and multiplier described in [8]. These floating point units follow the IEEE 754 double precision floating point format, except for that they do not handle denormal numbers or NaNs. However, because we are using normalized units in our MD simulations, it is very unlikely that any denormal or special numbers will occur. Thus, the adder and multiplier described in [8] are sufficient for our design. Currently, the multiplier also supports only the truncation mode of rounding.

These adders and multipliers are parameterized by the number of pipeline stages. They have been analyzed to determine

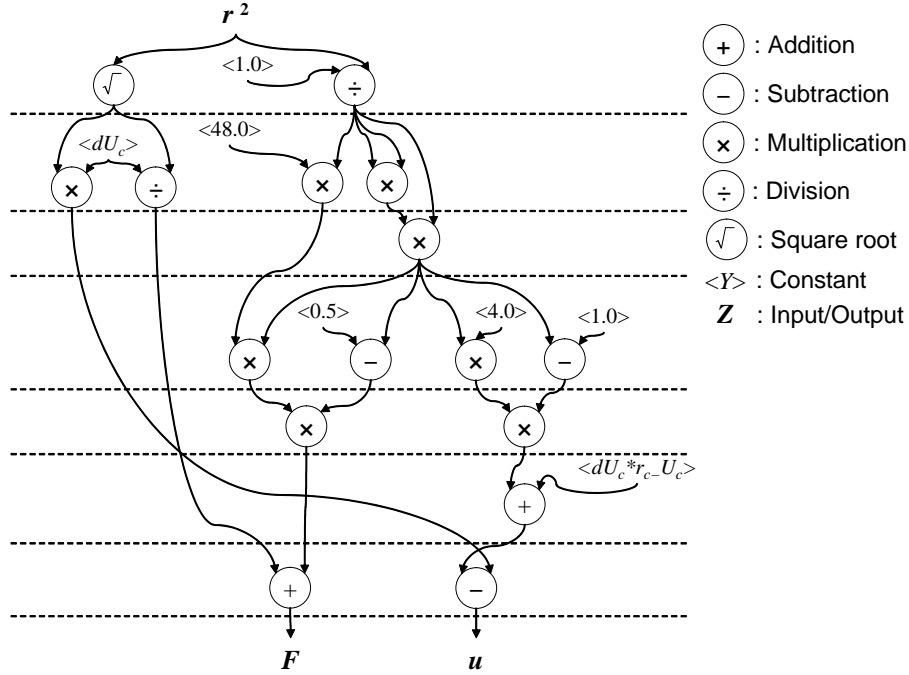


Fig. 1. Dataflow graph for the MD force and potential calculation kernels

the number of pipeline stages required to achieve the highest frequency/area ratio when these devices are implemented in a Xilinx Virtex-II Pro XC2VP125. The frequency/area ratio relates the throughput of the floating point unit to the area that it occupies on the FPGA. Pipelining a floating point unit increases the frequency at which the unit can run. However, after a certain number of pipeline stages, adding additional pipeline stages leads to only small increases in frequency but large increases in area. Thus, the number of pipeline stages leading to the highest frequency/area ratio is considered the optimal number of pipeline stages for the unit. By experiment, it has been determined that the optimal number of pipeline stages for the adder is 17 while the optimal number of pipeline stages for the multiplier is 12. The performance of these units, as well as that of the divider and square rooter, are given in Table I.

The divider is described in [11]. Currently, it requires 32 pipeline stages and can achieve a frequency of 125 MHz.

A. Square Root

For our force and potential calculations, we developed a 64-bit square rooter that follows the IEEE standard 754 for double precision floating point arithmetic with two exceptions. First, as with the adder, multiplier, and divider, denormal numbers are not supported. As stated previously, for our particular application, denormal number are unlikely to arise. Second, the square root unit only supports the round-to-the-nearest (tie to even) method of rounding. This is the most popular rounding mode and is thus sufficient for our design.

In IEEE format, a double precision floating point number consists of three parts: a sign bit (s), an 11-bit biased exponent (e), and a 52-bit mantissa (m). The floating point

TABLE I
PROPERTIES OF THE FLOATING POINT UNITS

	Number of Pipeline Stages	Frequency (MHz)	Area (slices)	Frequency/Area
Adder	17	160	884	0.180
Multiplier	12	205	910	0.225
Divider	32	125	4132	0.030
Square Rooter	47	143	1730	0.078

numbers are assumed to be normalized, so there is also an implied 1 (the “hidden bit”). The number represented is $(-1)^s \times 1.m \times 2^{e-b}$, where b is the exponent’s bias.

Let e_s and m_s be the exponent and mantissa, respectively, of the result of the square root operation. Then, as described in [12], the following is the basic algorithm for floating point square root.

- 1) If the e is even, find $e_s = e/2$ and $m_s = \sqrt{1.m0}$. If e is odd, find $e_s = (e + 1)/2$ and $m_s = \sqrt{0.1m}$.
- 2) Normalize m_s and update the exponent e_s .
- 3) Round.
- 4) Determine exception flags and special values.

We developed a pipelined design to perform these steps. Figure 2 shows the stages in our square root pipeline. In the first stage of the pipeline, the hidden bit is prepended to m , forming the significand ($1.m$), and the exponent and significand are adjusted if the exponent is odd.

In the first stage of the pipeline, the pipeline also begins to determine if the input received requires that a special number or zero be output. The following four cases are looked for.

- 1) If the input was positive or negative zero, a flag is set indicating that positive zero or negative zero, respectively, should be output.

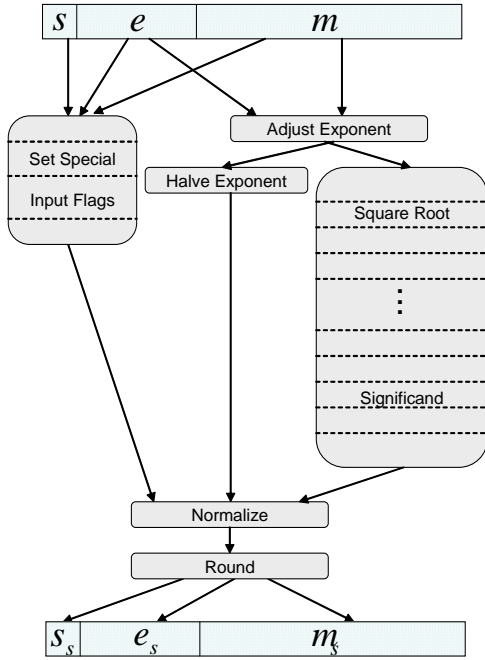


Fig. 2. Operations in the Square Root Pipeline

- 2) If the input was a NaN, a nonzero negative number, or negative infinity, a flag is set to indicate that NaN should be output.
- 3) If the input was positive infinity, a flag is set to indicate that positive infinity should be output.
- 4) If none of the above conditions apply, a flag is set to indicate that the computed result should be output.

The determination of the type of input and the special-case flag to be set takes four pipeline stages. After that, the flag is passed unchanged from stage to stage of the pipeline until it is used in the final stage.

In parallel with the special output determination, the pipeline computes e_s . This requires only one stage as it is only a simple right shift of e . The result is then passed from stage to stage of the pipeline until m_s has been computed and e_s is needed.

In parallel with the special output determination and finding e_s , starting in the second stage of the square root pipeline, the unit begins taking the square root of the significand (calculating m_s). The output of a square root operation has only half as many bits as the input to the square root operation. Our unit must calculate a 55-bit output: 52 bits for the mantissa, 1 bit for the hidden bit, 1 bit for normalization (the guard bit) and 1 bit for rounding (the round bit). The input is only 54 bits: 52 from the mantissa, 1 from the hidden bit, and 1 from the exponent adjustment. Thus, the unit appends 56 bits with value 0 to the end of the significand, giving a 110-bit number. Taking the square root of this 110-bit number yields a 55-bit output, as desired.

To perform the square root, the unit employs the non-restoring square root algorithm described in [13]. In this algorithm, one bit of the square root is determined at a time, starting with the most significant. The k th bit of the square

root is determined by the $(k - 1)$ bits of the square root that have been calculated so far, the remainder calculated in stage $(k - 1)$, and bits $(110 - k)$ and $(110 - k - 1)$ of the extended significand. The k th bit of the square root is found by performing either an add or subtract, depending upon the $(k - 1)$ th bit. Every step of the algorithm uses two bits of the extended significand to produce one bit of the result and a $(k + 1)$ -bit remainder.

A naive pipelining of this design would require 55 stages to calculate the square root of the 110-bit extended significand. However, we note that the early stages of such a pipeline would compute far less than the late stages. For example, the first stage would perform a 3-bit addition while the last stage would perform a 57-bit addition. Clearly, the 57-bit addition will be the critical path in the design. Hence, early stages of the naive pipeline can be combined without affecting the achieved frequency, so long as their combined delay is not greater than that of the final stage. We have been able to combine stages so that the square rooting of the extended significand requires 43 pipeline stages. If a smaller pipeline latency were needed, more stages could be combined with the penalty of a lower clock frequency.

After the 55-bit m_s has been computed, it is normalized: if the most significant bit is a 0, m_s is shifted left one bit and e_s is decremented by 1; otherwise, no change is made. The most significant 1 becomes the hidden bit and the next most significant 53 bits are then passed to the rounding stage.

In the rounding stage, the 53-bit normalized result is incremented by 1. The 52 most significant bits of this addition as well as the carry out are passed to the final stage.

In the final stage of the pipeline, the special case flags are examined. If one of them is set, the corresponding value is written to the output register. Otherwise, there are two possibilities.

- 1) The carry out of the rounding stage was 0. In this case, the exponent and the unchanged output from the rounding stage are written to the output register, along with a sign bit of 0.
- 2) The carry out of the rounding stage was 1. This carry would be added to the hidden bit, giving $1 + 1 = 10$. The 1 from this sum is considered the hidden bit. The 0 from this sum is prepended to the output of the rounding stage. The most significant 52 bits of this value are written to the output register. The exponent is incremented and the result is written to the output register. A sign bit of 0 is also written to the output register.

In total, there are 47 stages in the square root pipeline. This large number of pipeline stages is perfectly acceptable for our force and potential calculation design. As described in Section IV, the square root can have up to $d + \max((4m + 2a), (3m + 3a)) - \max(m, d)$ stages for the square root data to be available soon enough to not delay the pipeline. Since our multiplier, adder, and divider have 12, 17, and 32 stages, respectively, the square root could have as many as 87 stages and still finish in time to do the force and potential

TABLE II

PERFORMANCE FOR ONE AND TWO PIPELINES IN THE XC2VP125-7

Number of Pipelines	Frequency (MHz)	Area (slices)	No. of Embedded Multipliers	Throughput (MFLOPS)
1	122	21113	128	1952
2	122	43133	256	3904

calculations. Therefore, our square rooter is well-within our allowed range.

To verify the functionality of our square rooter, we coded it in VHDL, synthesized it with Xilinx XST, and placed-and-routed the design using Xilinx PAR. We then configured the Virtex-II Pro XC2VP7 FPGA on Xilinx ML300 board with the design using iMPACT. To provide data to the design and read the output results, we used the Xilinx Microprocessor Debug (XMD) engine utility in the Xilinx EDK. With XMD, we could tell the embedded PowerPC in the Virtex-II Pro to write to the square root design’s input register and to read its output register. The communication between the PowerPC and the square rooter was accomplished with an OPB bus. We provided several test inputs to the design and, from the debugger output, verified that the output was indeed correct.

VI. PERFORMANCE

We used the floating point units described in the last section to compose the force and potential pipeline described in Section IV. Given the delays of the various floating point units, the entire force and potential pipeline calculation has 119 stages; it is very deeply pipelined. However, the pipeline never has to stall and there is never an occasion for the pipeline to be flushed. It can accept input in the form of one 64-bit floating point number every clock cycle and, once the pipeline is full, output *two* 64-bit results (one potential and one force) every clock cycle. For a typical 10000-molecule simulation, there will be approximately 5800 force and potential calculations per position update. Including the pipeline fill-up latency, our design will require 5919 cycles to perform these computations, for an average of 1.02 cycles/(2 results) or *0.51 cycles/result*.

The entire design was coded in VHDL, including the appropriate synchronization logic to deal with the fact that the floating point units all have different numbers of pipeline stages. In the current design, this synchronization logic was coded as shift registers. The design was then synthesized using Synplicity Synplify Pro 7.2. We then used the place-and-route tools in Xilinx ISE 5.2. We targeted the Xilinx Virtex-II Pro XC2VP125-7. Table II shows the frequency, area, number of embedded multipliers, and throughput for up to two pipelines.

A. Comparison with General Purpose Processor

We compared our design against implementations on two general purpose processor systems. One system has an AMD Athlon XP 1700+ processor, has 512 MB of memory, and runs the Microsoft Windows 2000 operating system. The other system has dual 900 MHz Intel Itanium 2 processors, 8.3 GB of memory, and runs the Red Hat Linux Advanced Server 2.1 operating system. To obtain the performance results for the general purpose processors, we first we began with the C code from [2]. We made minor modifications to the code so

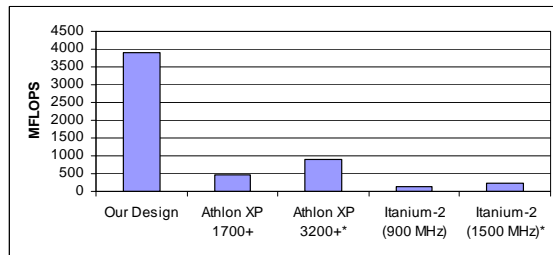


Fig. 3. Comparison of the throughput of two pipelines of our design with Athlon XP 1700+, Athlon XP 3200+*, 900 MHz Itanium 2, and 1500 MHz Itanium 2* implementations for force and potential calculation within a simulation (* indicates estimated results)

that we would be able to measure the performance of the force and potential calculations using AMD CodeAnalyst version 2.26.1.0 and Intel VTune version 2.0 for Linux [14], [15]. For the Athlon system, we compiled the code with the Microsoft Visual C/C++ compiler, version 6, using full optimization. The throughput, in MFLOPS, of this implementation is presented in Figure 3 against the throughput of our design with two pipelines. We also scale the Athlon XP 1700+ performance by 3200/1700 to estimate the performance of the program on an Athlon XP 3200+, the fastest Athlon processor. This estimated performance is also shown in Figure 3. For the Itanium system, we compiled the code with the Intel C++ Compiler 7.0 for Linux, using the `-fast` (highest optimization, static linking, interprocedural optimization) and `-Ob2` (inline any suitable function) optimizations. Note that the compiled code used only one of the two processors in the Itanium system while running. The results for the Itanium system, as well as estimated results for a 1500 MHz Itanium system, are shown in Figure 3. We can see that, in this case, our force and potential calculation pipeline significantly outperforms even the fastest Athlon XP and Itanium processors.

As another comparison, we isolated the force and potential calculation in a separate program and measured the general purpose processors’ performances in executing this program. That is, we made a program that just contained the computations for force and potential calculation. The input data for the computations was randomly generated squares of distances $0 < r^2 < r_c^2$. As seen in Figure 4, the general purpose processors perform much better in this scenario. However, our design still outperforms the fastest of the general purpose processors by over 1100 MFLOPS.

B. Contrasting Our Work with Existing FPGA-based Acceleration for MD

It is difficult to compare our design’s performance with that of the design in [4]. Instead, we state the key differences between our work and theirs. First, the two designs accelerate different portions of the simulation. Second, our design uses 64-bit precision while that in [4] uses 32-bit precision. Third, our design employs area-hungry and comparatively low-frequency dividers and square rooters while the other design does not. And finally, for the Virtex-II Pro implementation, our design is targeted to a much larger device than that in [4].

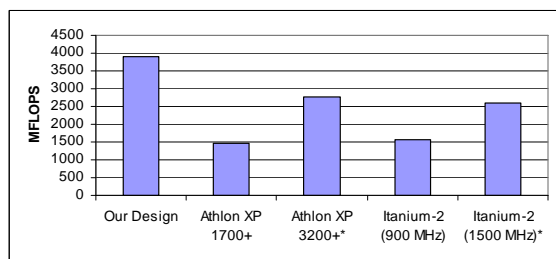


Fig. 4. Comparison of the throughput of two pipelines of our design with Athlon XP 1700+, Athlon XP 3200+*, 900 MHz Itanium 2, and 1500 MHz Itanium 2* implementations for force and potential calculation alone (* indicates estimated results)

VII. CONCLUSION

In this paper, we have presented a pipeline for the calculation of the Lennard-Jones forces and potentials that are used in MD simulations. This pipeline uses 64-bit, IEEE standard 754 floating point arithmetic. A square root unit has been developed especially for this force and potential calculation pipeline. It has 47 pipeline stages and can achieve up to a 143 MHz clock frequency.

The force and potential pipeline overall has 119 pipeline stages. It can achieve a frequency of 122 MHz with either one or two pipelines present on the FPGA. With two pipelines present, a throughput of 3.9 GFLOPS is achieved. This throughput is a significant improvement over even the fastest general purpose processor implementation.

A. Future Work

There are many areas for future work, starting with the floating point units. The square root and divider both achieve low frequencies when compared to the multiplier and the adder, so we would like to investigate ways for improving the speed. We would also like to analyze these floating point units to determine the optimal frequency/area ratio.

We also believe that the area of the floating point units and the area of the entire force and potential pipeline can be reduced by utilizing the on-chip, embedded memory. Any data that is just stored for multiple pipeline stages could be stored in a Block RAM until the stage in which it is needed. This will eliminate the need for shift registers and free up their logic for other uses, or allow us to use a smaller FPGA for the same pipeline.

There are many future directions to pursue in the area of reconfigurable hardware acceleration for MD simulation. There are many van der Waals forces beyond the Lennard-Jones forces. The approach used in this paper, in which we break the force and potential calculation into stages and exploit both fine- and coarse-grained parallelism, can be used for other van der Waals forces whose calculation is similar to that of the Lennard Jones force. Such forces include those arising from the 12-6, n-m, and hydrogen bond potentials [16].

There are also forces that have much more complicated formulas for their calculation, including the use of trigonometric functions [1]. To implement the calculation of these

forces and their associated potentials may require an approximation method such as table lookup and interpolation. There are also the other two classes of forces—intramolecular and electrostatic—to be investigated. Finally, we would like to investigate the reconfigurable hardware acceleration of multiple kernels for a given simulation.

ACKNOWLEDGMENTS

The authors wish to thank Gokul Govindu; Ling Zhuo; Cong Zhan; and Ronald Scrofano, Sr. for assistance with and discussion regarding the floating point units and their implementation. The authors also wish to thank Maya Gokhale, Frans Trouw, Aiichiro Nakano, and Priya Vashishta for their valuable insights into molecular dynamics and the application of reconfigurable computing to it.

REFERENCES

- [1] M. Allen and D. Tildesley, *Computer Simulation of Liquids*. New York: Oxford University Press, 1987.
- [2] A. Nakano, "Class notes for CSCI 599: High performance scientific computing," University of Southern California, Fall semester, 2003.
- [3] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya, "Protein explorer: A petaflops special-purpose computer system for molecular dynamics simulations," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. New York: ACM Press, November 2003.
- [4] C. Wolinski, F. Trouw, and M. Gokhale, "A preliminary study of molecular dynamics on reconfigurable computers," in *Proceedings of the 2003 International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2003.
- [5] T. Hamada, T. Fukushige, A. Kawai, and J. Makino, "PROGRAPE-1: A programmable, multi-purpose computer for many-body simulations," *Publications of the Astronomical Society of Japan*, vol. 52, pp. 943–954, October 2000.
- [6] W. D. Smith and A. R. Schnore, "Towards an RCC-based accelerator for computational fluid dynamics applications," in *Proceedings of the 2003 International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2003.
- [7] G. Lienhart, A. Kugel, and R. Manner, "Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society Press, April 2002.
- [8] G. Govindu and V. K. Prasanna, "Analysis of high performance floating point arithmetic on FPGAs," in *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW 2004)*, April 2004.
- [9] L. Zhuo and V. K. Prasanna, "Scalable modular algorithms for floating-point matrix multiplication on FPGAs," in *Proceedings of the International Parallel and Distributed Processing Symposium 2004 (IPDPS 2004)*, April 2004.
- [10] P. Belanovic and M. Leeser, "A library of parameterized floating point modules and their use," in *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, Berlin: Springer-Verlag, September 2002, pp. 657–666.
- [11] V. Daga, G. Govindu, V. K. Prasanna, S. Gangadharipalli, and V. Sridhar, "Floating-point based block LU decomposition on FPGAs," in *Proceedings of the 2004 International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2004.
- [12] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco: Morgan Kaufman, 2004.
- [13] Y. Li and W. Chu, "A new non-restoring square root algorithm and its VLSI implementations," in *Proceedings of the 1996 International Conference on Computer Design*, October 1996.
- [14] AMD. Inc., <http://www.amd.com>.
- [15] Intel Corp., <http://www.intel.com>.
- [16] W. Smith, M. Leslie, and T. R. Forester, *The DL-POLY.2 User Manual*, <http://www.cse.clrc.ac.uk/msi/software/DL-POLY/USRMAN2/USRMAN.html>, April 2003.
- [17] Xilinx. Inc., <http://www.xilinx.com>.