IMPROVING NUMERICAL REPRODUCIBILITY AND STABILITY IN LARGE-SCALE NUMERICAL SIMULATIONS ON GPUS

by

Philip Saponaro

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of B.S of Computer Science with Distinction.

Spring 2010

IMPROVING NUMERICAL REPRODUCIBILITY AND STABILITY IN LARGE-SCALE NUMERICAL SIMULATIONS ON GPUS

by

Philip Saponaro

Approved: _____

Michela Taufer, PhD
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Lori Pollock, PhD
Committee member from the Department of Computer Science

Approved: _____

Pak-Wing Fok, PhD
Committee member from the Board of Senior Thesis Readers

Approved: _____

Ismat Shah, PhD
Chair of the University Committee on Student and Faculty Honors

ACKNOWLEDGMENTS

I wish to thank my advisor, Michela Taufer, and my research partner, Omar Padron, without whom none of this project would have been possible. I would also like to thank my friends and family, who have supported me throughout my entire college career.

This work was supported by the National Science Foundation grant #0941318 ``CDI-Type I: Bridging the Gap Between Next-Generation High Performance Hybrid Computers and Physics Based Computational Models for Quantitative Description of Molecular Recognition", and grant #0922657 ``MRI: Acquisition of a Facility for Computational Approaches to Molecular-Scale Problems"; by the U.S. Army, grant #YIP54723-CS ``Computer-Aided Design of Drugs on Emerging Hybrid High Performance Computers", by the Computing Research Association through the Distributed Research Experiences for Undergraduates (DREU), and by the NVIDIA University Professor Partnership Program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The advent of general purpose graphics processing units (GPGPU's) brings about a whole new platform for running numerically intensive applications at high speeds. Their multi-core architectures enable large degrees of parallelism via a massively multi-threaded environment. Molecular dynamics (MD) simulations are particularly well-suited for GPU's because their computations are easily parallelizable. Significant performance improvements are observed when single precision floating point arithmetic is used. However, this performance comes at the cost of accuracy: it is widely acknowledged that constant-energy (NVE) MD simulations accumulate errors as the simulation proceeds due to the inherent errors associated with integrators used for propagating the coordinates. A consequence of this numerical integration is the drift of potential energy as the simulation proceeds. Double precision arithmetic partially corrects this drifting but is significantly slower than single precision and is comparable to CPU performance.

To address this problem, we present development of a library of mathematical functions that use fast and efficient algorithms to improve numerical reproducibility and stability of large-scale simulations. We test the library in terms of its performance and accuracy with a synthetic code that emulates the behavior of MD codes on GPU, and then we present results of a first integration of our library in a MD code. These first results show correction of the drifting with a performance much better than double precision.

Chapter 1

INTRODUCTION

1.1 Motivation

Large scale simulations are being moved to parallel platforms such as the GPU. These simulations can be run at longer time scales with a much better performance; however, small errors accumulate over time and skew the results.

Molecular Dynamics (MD) simulations are excellent targets for GPU accelerators since most aspects of MD algorithms are easily parallelizable. Enhancing MD performance can allow the simulation of longer times and the incorporation of multiple scale lengths. Constant energy (NVE) dynamics is performed in a closed environment with a constant number of atoms (N), constant volume (V), and constant energy (E). With single precision GPUs, constant energy simulations present significant drift of the energy values for long simulations (of the order of 30 nanoseconds) [7]. So for example, the components of the potential energy, i.e., the electrostatic and Van der Waals energies, converge towards zero (e.g., the negative electrostatic energy increases towards zero and the positive Van der Waals energy decreases towards zero) rather than remaining constant as expected.

The problem is not simply due to the fact that some operations on GPU are not IEEE compliant [10, 11]. This phenomenon is also observed when IEEE complaint operations are used on GPUs, and for the same simulations when performed on double precision GPUs. In the latter case the divergence is very small and in all cases it is not related to an erroneous implementation of the MD algorithm [6].

Furthermore, MD simulations are among other large-scale numerical simulations that, when performed on parallel systems, suffer from being very sensitive to cumulative rounding errors. These errors depend both on the implementation of floating point operations and on the way calculations are performed in parallel: final results can differ significantly among platforms and number of parallel units used (threads or processes) [9]. Overall, numerical reproducibility and stability of results cannot be guaranteed in large-scale simulations. By "reproducibility and stability" we mean that results of the same simulation running on GPU and CPU lead to the same scientific conclusions. Over time, these small errors accumulate and skew the final results; the longer the simulation, the larger the error.

Because of their parallelism and power, GPUs are able to run longer simulations in a shorter amount of time than CPUs [13, 1, 8, 7]. However, this comes at a higher cost in numerical reproducibility and stability. Threads, which are single lightweight processes that run in parallel, can be scheduled at different times, leading to different errors, and ultimately, different final results. This, combined with longer simulations and lack of IEEE compliance in some hardware operations, can lead to erroneous conclusions.

1.2 Thesis Contribution

The contribution of this thesis is as follows:

1) We developed a mathematical library, built upon previous literature [14], by implementing a set of mathematical operations for floating point arithmetic to improve numerical reproducibility and stability of large-scale parallel simulations on GPU systems. Our proposed approach uses a new numeric type composed of multiple

single precision floating point numbers. We call numbers of this type "composite precision floating point numbers".

2) Since MD codes are very complex to deal with, the library validation in terms of accuracy and performance of our library is performed on a suite of synthetic codes that simulate the MD behaviors on GPU systems. The suite includes a global summation that reproduces errors in total energy summations and a do/undo set of programs that reproduces drifting in single energy computations. We present results that show the accuracy of composite precision arithmetic is comparable to double precision, and the performance comparable to single precision.

3) After validating the library with the suite of synthetic codes, we integrate the library with the MD code. We present preliminary results of the accuracy and performance of the MD code using composite precision arithmetic.

1.3 Thesis Outline

The paper is organized as follows: Chapter 2 provides a short overview of GPU programming and accuracy issues in GPU calculations; Chapter 3 discusses the state of the art in the field; Chapter 4 shows the energy drifting in MD simulations; Chapter 5 describes our composite floating point arithmetic; Chapter 6 presents the synthetic suite used for assessing accuracy and performance of our approach; Chapter 7 shows the results of integrating our composite precision with the MD code; and Chapter 8 concludes the paper and presents future work.

Chapter 2

BACKGROUND

2.1 GPU Programming

GPUs are massively parallel multithreaded devices capable of executing a large number of active threads concurrently. A GPU consists of multiple streaming multiprocessors, each of which contains multiple scalar processor cores. For example, NVIDIA's G80 GPU architecture contains 16 multiprocessors, each of which contains 8 cores, for a total of 128 cores which can handle up to 12,288 active threads in parallel. In addition, the GPU has several types of memory, most notably the main device memory (global memory) and the on-chip memory shared between all threads in a block.

The CUDA language library facilitates the use of GPUs for general purpose programming by providing a minimal set of extensions to the C programming language. From the perspective of the CUDA programmer, the GPU is treated as a coprocessor to the main CPU. A function that executes on the GPU, called a kernel, consists of multiple threads each executing the same code, but on different data, in a manner referred to as "single instruction, multiple data" (SIMD). Further, threads can be grouped into thread blocks, an abstraction that takes advantage of the fact that threads executing on the same multiprocessor can share data via the on-chip shared memory, allowing a limited degree of cooperation between threads in the same block. Finally, since GPU architecture is inherently different from a traditional CPU, code

optimization for the GPU involves different approaches, which are described in detail elsewhere [10, 11].

2.2 Issues with GPU Precision

As noted in the CUDA Programming Guide [10, 11], CUDA implements single precision floating-point operations e.g., division and square root operations, in ways that are not IEEE-compliant. Their error, in ULP(Units in the Last Place) is nonzero. While addition and multiplication are IEEE-compliant, combinations of multiplication and addition are treated in a nonstandard way that leads to incorrect rounding and truncation.

Chapter 3

RELATED WORK

Numerical reproducibility and stability for chaotic applications was addressed for massively parallel CPU-based architectures in [9]. The work in [9] does not address emerging high performance paradigms such as GPU programming and their novel architectures. An approach similar to ours was theoretically suggested in [14]. We build our work upon these two contributions with MD simulations as the targeted large-scale numerically intensive application. Specifically, in [14] a theoretical method for capturing error in computation on GPU was described, but was never implemented. We implement and validate these methods on the GPU.

Arbitrary precision mathematical libraries are a valuable approach used in the 70s and 80s to address the acknowledged need for extended precision in scientific applications. As outlined in [12, 3, 2], high precision calculations can indeed be achieved using arbitrary precision libraries and these libraries can solve several problems, e.g., correct numerically unstable computation when even double precision is not sufficient. Existing libraries target CPU platforms, not GPUs. Most libraries are open source, e.g., MPFR C library for multiple precision floating point computations with correct rounding under LGPL (http://www.mpfr.org/) and the ARPREC C++/Fortran-90 arbitrary precision package from LBNL (http://crd.lbl.gov/dhbailey/mpdist/). One critical aspect of these libraries is their complexity. Our approach targets GPUs, is simpler to implement, and can be easily integrated in existing CUDA codes.

Chapter 4

ENERGY DRIFTING IN MD SIMULATIONS

4.1 MD Code Organization

Molecular Dynamics simulations, being chaotic applications, make perfect examples of unstable applications when executed on parallel computers. Small changes during intermediate computations (such as Van der Waals and electrostatic energies or global summations of the various energies) accumulate to yield substantially different final results [4].

The MD code we used for GPUs, which was presented in other work [7], is organized as follows in Figure 1. The program reads initial input parameters, such as length of the simulation and output file names, loads any check pointing information, and then enters the main loop. This main loop performs a single step of the MD simulation, and then prints the results to a file at regular intervals defined by the user. After the simulation has completed the number of steps as described in the input parameters, the program frees any allocated memory and then ends.

A single step of the MD simulation is composed of calculating energies due to different types of interactions, seen below in Figure 2. There are five types of interactions: bond, angle, dihedral, Van Der Waal's, and electrostatic. The energy due to each interaction is calculated, stored, and are later summed together to give the total energy of the simulation. In this thesis, we consider the non-bond energy (ie Van Der Waal's and electrostatic energies). Figure 3 shows the high level view of calculating the non-bond energy.

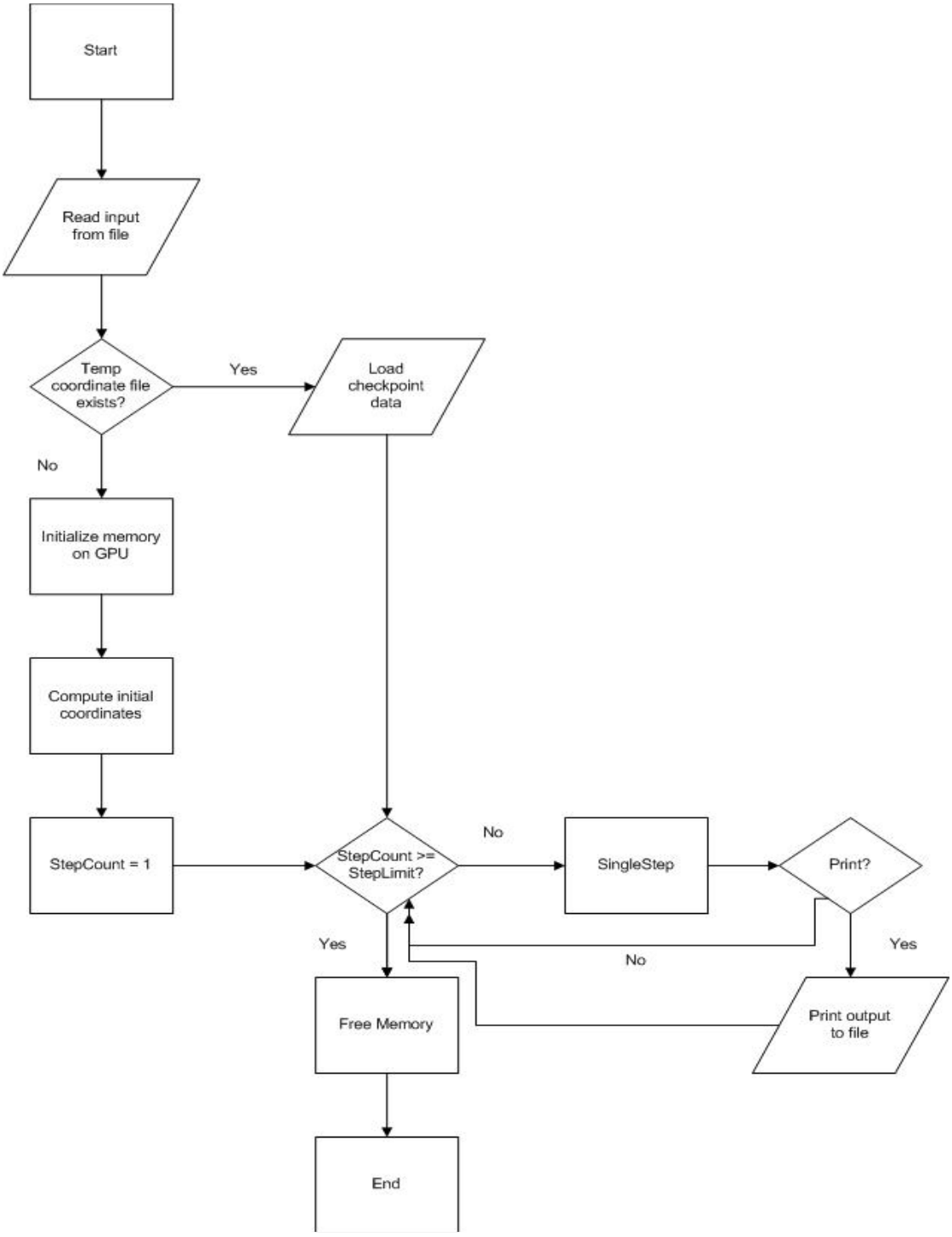


Figure 1 Flow chart of main MD code

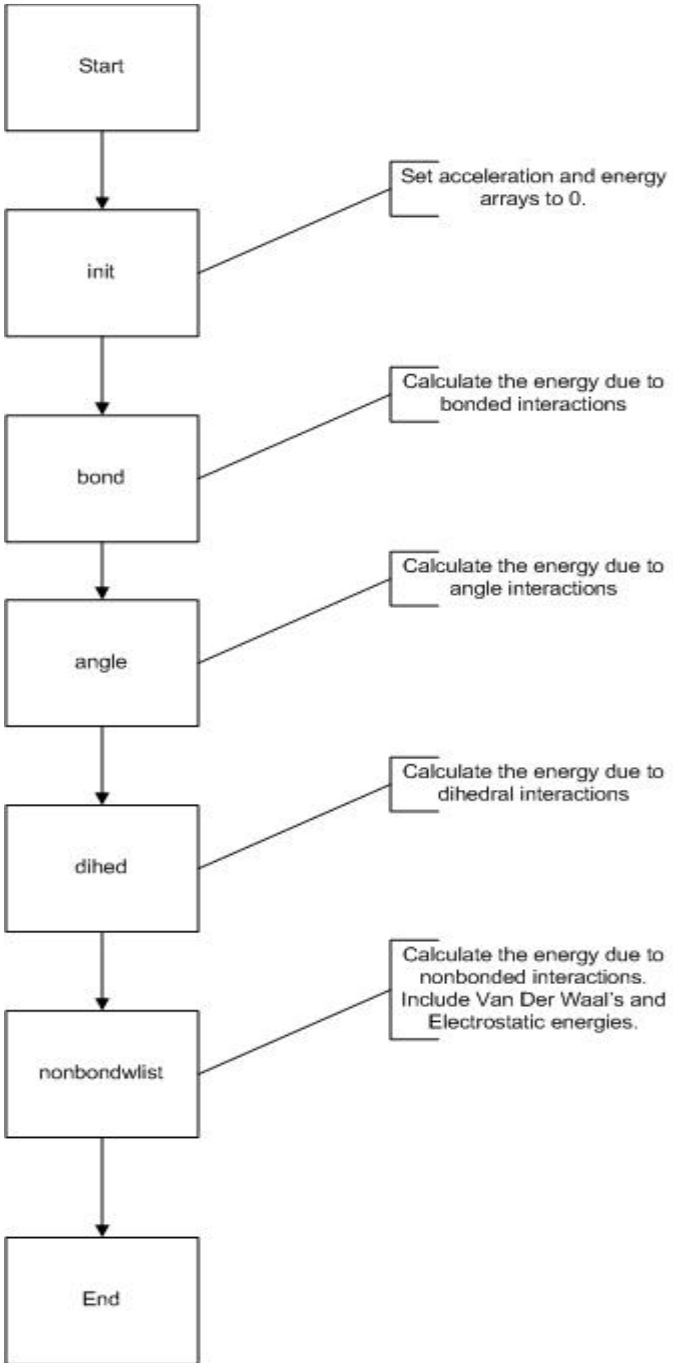


Figure 2 Flow chart of single step of the MD simulation, which calculates the energy due to each interaction

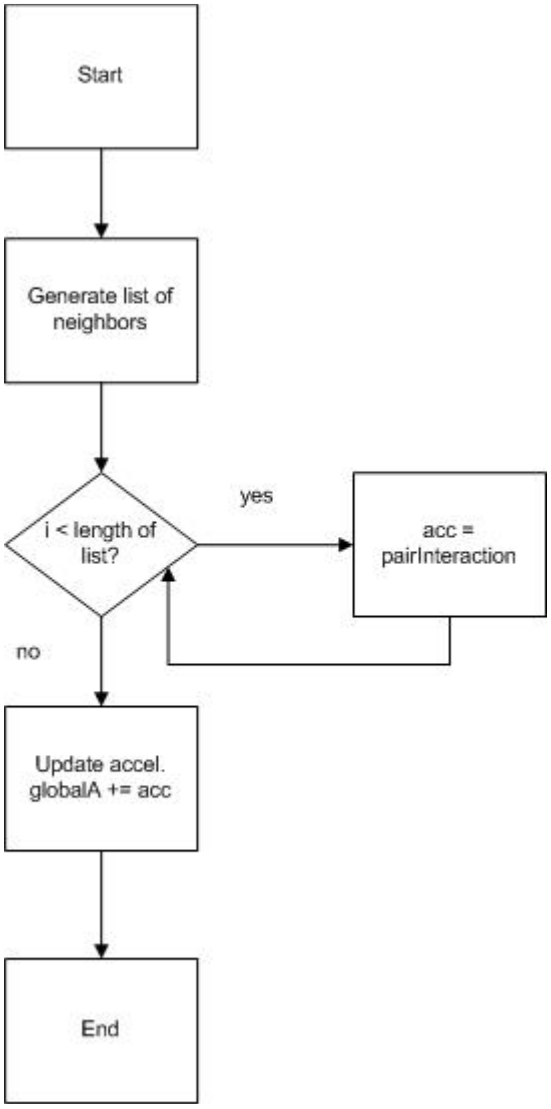


Figure 3 Flow chart of nonbondwlist, which calculates the energy due to nonbonded interaction

4.2 Drifting

In constant energy MD simulations, the total energy of the simulation should stay the same. However, we observe that the total energy "drifts" over time. By drifting we mean the total energy slowly increases or decreases over time, instead of staying constant.

To study the energy behavior of NVE MD simulations on GPUs, we examined previous work in which the total energy was measured over the course of a 30 ns simulation for the 988 water system using the MD code for GPUs. A time step size of 1 fs was used, so this test simulation is 30 million MD steps long. The results are shown below in Figure 4.

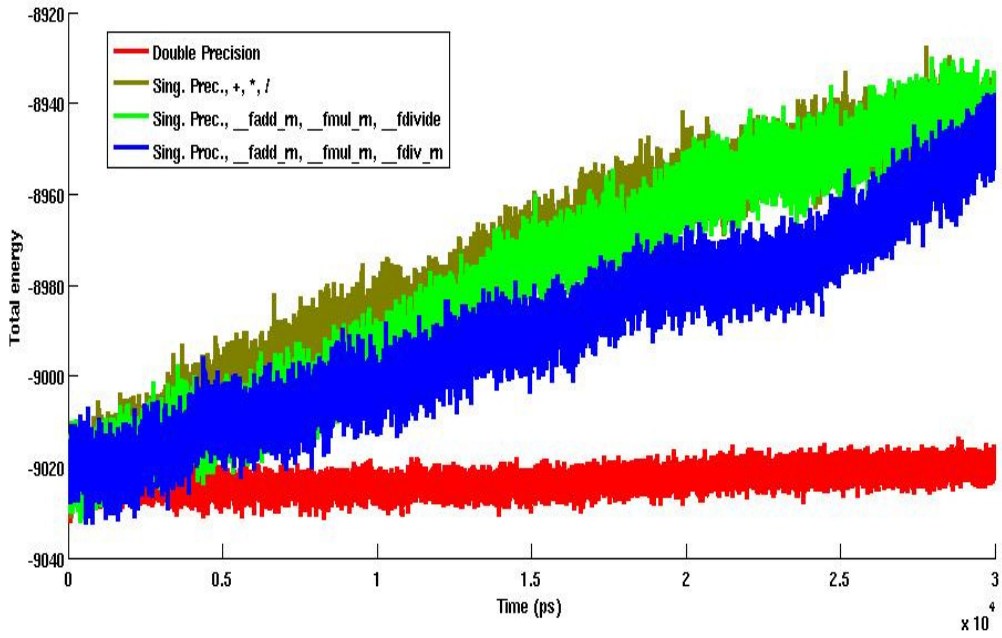


Figure 4 **Time profiles of the total energy for simulations of the 988 water, 18 NaI system. Results are shown using default single precision, single precision with correction for nonstandard rounding and truncation, and double precision.**

Four profiles with different types of precision (single and double precision) and different implementations of the single precision operations sum, multiplication, and division, are shown. The first (single precision with +, *, and /) demonstrates that the use of default single precision arithmetic leads to a very large drift over the 30 ns simulation. CUDA implements these operations in ways that are not IEEE-compliant. The second (single precision with _fadd_rn, _fmul_rn, and _fdividef) still demonstrates the same drifting despite addition and multiplication which are IEEE compliant. The third (single precision with _fadd_rn, _fmul_rn, and _fdiv_rn) exhibits drifting similar to the other two profiles despite the introduction of an IEEE-compliant division, suggesting that the cause of drifting goes beyond the implementation of single operations. The fourth profile (double precision) in Figure 4 is the result of using double precision arithmetic and shows no significant drift, except for the very small amount expected normally in long NVE simulations.

For longer simulations, longer than 100 ns, even double precision GPUs start showing a drifting behavior. Previous work shows the drifting to the lack in numerical reproducibility and stability already observed in conventional distributed systems such as clusters [9]. Here, the effect is significantly enhanced since the simulation is effectively performed on a ”cluster” of greater than 32 or 64 cores (processors).

Using double precision does not completely eliminate drifting on current GPU systems as we noted in our previous work [7]. Moreover, double precision arithmetic dramatically reduces performance to levels comparable to that of CPUs (12 times slower), as shown in Table 1. Therefore, simply using double precision is not an acceptable solution.

Table 1 Performance measured in MD steps per second for a 988 water, 18 NaI solvent system using different types of precision

MD code	Platform	Precision	steps/s
CHARMM-GPU	Tesla S1070	Doub. Prec., +, *, /	35.23
CHARMM-GPU	Tesla S1070	Sing. Prec., +, *, /	377.92
CHARMM-GPU	Tesla S1070	Sing. Prec.,faddrn,fmulrn, /	423.54
CHARMM-GPU	Tesla S1070	Sing. Prec.,faddrn,fmulrn,fdivrn	129.87
CHARMM CPU	1 CPU		34.34
CHARMM CPU	2 CPUs		64.95
CHARMM CPU	4 CPUs		116.62
CHARMM CPU	8 CPUs		186.05

Chapter 5

REDEFINING FLOATING POINT ARITHMETIC ON GPUS

5.1 Composite Floating Point Numbers

The major flaw in traditional floating point numbers is that an accurate representation of values with many significant bits is not possible as the less significant bits may be truncated. However, if the value considered has "clusters" of contiguous significant bits with a large number of zeros separating them, a more accurate representation can be achieved with separate floating point numbers (intuitively one for each cluster, although not necessarily) for which each cluster of bits corresponds to a portion of the mantissa, which are the bits that contain the significant digits of the number. Note that each cluster is significant in different orders of magnitude. We propose to represent a value as the sum of two floating point numbers of arbitrarily varying orders of magnitude. This allows us to capture the significant parts of the value for numbers that exhibit these properties and affords scientists a better compromise between performance and reliability on GPU systems. In particular, we propose that numerical reproducibility and stability of large-scale simulations are achievable on GPUs with the use of composite precision floating point arithmetic. The composite precision floating point number is a data structure consisting of two single precision floating point numbers, *value* and *error*. The value of a floating point number, *n*, is expressed as the sum of the two floats, with $n_{error} << n_{value}$:

$$n = n_{value} + n_{error}$$

When calculating the sum or product of two numbers, the approximation of the error in their result is much lower in magnitude when compared to the result itself. Both result components can be preserved by representing the final result as the sum of the truncated result and the approximation of its error. In other words, we can think of the *value* component of the number as the result of a calculation and the *error* component as an approximation of the error carried in the calculation. For this representation on GPUs, we used the float2 data type that is available in CUDA (Figure 5).

```
struct float2 {
    float x; //x2.value
    float y; //x2.error
} x2;
 ...
float x2 = x2.x + x2.y; //x2.value + x2.error
```

Figure 5 Data structure of float2

Errors in each calculation are carried through operations on GPUs. The conversion from float2 structures back to float structures is a simple matter of adding the *value* and *error* terms. In large-scale simulations, we observe how errors accumulate so that when converting float2 back to float, the final result does not neglect the error component. The individual errors that would have been truncated under traditional single precision floating point operations add up and ultimately impact the final reported value, resulting in more stable numerics.

5.2 Redefining Floating Point Operations

The algorithms used for performing composite precision floating point addition, multiplication, and division are defined in terms of multiple single precision additions, subtractions, and multiplications as well as a single precision floating point reciprocal. These algorithms are referred to as being "self compensating" - they perform the calculation as well as keep track of inherent error. The algorithm used for the addition and multiplication are based on algorithms proposed in [14, 9].

The implementation of the composite precision floating point addition is presented in Figure 6 and requires four single precision additions and four subtractions. The subtraction is implemented the same as the addition, with the exception that the signs of y2.*value* and y2.*error* are reversed before performing the sum. The error equation involves calculating the lower bits of the sum.

```
Pseudo Code          Implementation

float2 x2,y2,z2      float2 x2,y2,z2
z2 = x2 + y2         float t
                     z2.value = x2.value
                              + y2.value
                     t = z2.value
                       - x2.value
                     z2.error = x2.value
                              - (z2.value - t)
                              + (y2.value - t)
                              + x2.error
                              + y2.error
```

Figure 6 Algorithm for composite precision floating point addition

For the composite precision floating point multiplication presented in Figure 7, each operand is expressed as the sum of their value and error components and the resulting product is symbolically expanded into a sum of four terms. The first is the value stored in z2.*value* and the sum of the others is stored in z2.*error*. For this multiplication, four single precision multiplications and two single precision addition operations are required.

```
Pseudo Code       Implementation

float2 x2,y2,z2   float2 x2,y2,z2
z2 = x2 * y2      z2.value = x2.value * y2.value
                  z2.error = x2.value * y2.error
                           + x2.error * y2.value
                           + x2.error * y2.error
```

Figure 7 Algorithm for composite precision floating point multiplication

The composite precision floating point division implementation in Figure 8 represents the ratio of two numbers as the product of the dividend and the reciprocal of the divisor. The problem of calculating a reciprocal is, in turn, posed as a root finding problem: Given a floating point number a, its reciprocal is another floating point number b such that $b^{-1} - a = 0$. The process of finding the root of this function is based on Karps method, an extension of the Newton-Raphson method. Our algorithm, presented in Figure 8, extends the algorithm in [14].

```
Pseudo Code          Implementation

float2 x2,y2,z2      float2 x2,y2,z2
z2 = x2 / y2         float t,s,diff
                     t = (1 / y2.value)
                     s = t * x2.value
                     diff = x2.value
                          - (s * y2.value)
                     z2.value = s + t * diff
                     z2.error = t * diff
```

Figure 8 Algorithm for composite precision floating point division

Chapter 6

EVALUATION OF LIBRARY WITH SYNTHETIC CODES

6.1 Synthetic Suite

Errors within the MD simulation are caused by two main parts. The first is the summation of all different energies. The second source of error comes from calculating the energies themselves. To validate the composite precision library, we need to verify that it does capture the correct amount of error over many performed calculations.

MD codes are very complex, thus we developed a suite of synthetic codes that reproduce rounding errors in MD. The suite is comprised of two programs emulating iterative calculations of energy terms with their energy fluctuations typical of MD simulations and the observed drifting. The first program is a global summation program that reproduces errors in total energy summations in MD. The second program is a do / undo program that produces drifting in single energy computations in MD. The do/undo is done by performing an operation on a value, and then applying its inverse (e.g., multiplication and division, or self multiplication and sqrt). The truncation of intermediate results produce the drifting behavior observed.

6.2 Global Summation

The global summation program calculates the sum of a large set of numbers with a high variance in magnitude. Since computers can only store a fixed

amount of significant digits, when adding very small numbers with very large numbers, the small numbers may be neglected. In other words, the small number contributes too small a portion to the result and the number of significant digits needed to represent it is more than what is available. The final result is very sensitive to the order in which the numbers are summed.

To assess how our composite precision floating point arithmetic library improves the numerical reproducibility and stability in a global summation calculation, we randomly generated an array of 1000 numbers filled with very large $O(10^6)$ and very small $O(10^{-6})$ values. The distribution of values was purposefully made symmetric: whenever we generated a number, the next number generated was its negative. This gave us a numerical benchmark from which to judge the effectiveness of our algorithms: the advantage of knowing, a priori, the correct sum to be zero. Figures 9 and 10 show the distribution of the numbers used for the validation: Figure 9 shows the numbers with absolute value larger than 1 (up to 10^6) and Figure 10 shows the numbers with absolute value smaller than 1 (on the order of 10^{-6}).

Figure 9 Distribution of large numbers

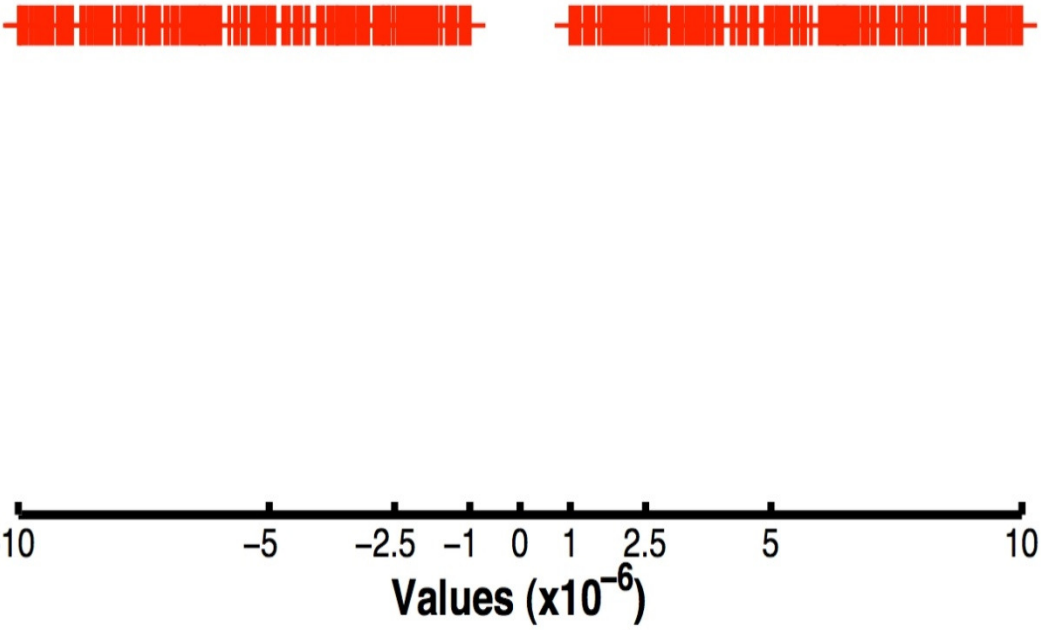


Figure 10 Distribution of small numbers

The sum of the array was computed multiple times on a GPU, each time with a different sorting order. We considered a sum in increasing order, a sum in decreasing order, and four independent trials in which the array was shuffled into random configurations. We summed the array using different representations of the numbers, i.e., single (float) and double precision as well as our software composite precision (float2), and with different numbers of threads. The tests were performed on one GPU of the Tesla S1070 system. Table 2 shows results of our global summation for an array of 1,000 elements summed using a single thread on GPU. In other words, the sum was performed sequentially by a single thread on the GPU. Table 3 shows results of our global summation for the same array of 1,000 elements summed using 1,000 threads on GPU. In this case, the summation is done by the CPU when the array values are returned to the host. Table 4 and Table 5 show results of our global summation for the same array of 1,000 elements summed using 100 threads on GPU and 10 threads on GPU, respectively. In this case, the summations were partially performed on GPU and partially on CPU.

In all cases, because of the way the array of values is built, we expected the result to be zero. However, in only a few cases was this actually observed, even with double precision. If compared with the float representation (single precision), our composite representation is able to correct the results significantly (i.e., between 4 and 5 orders of magnitude) and provides results closer to the double precision solution than the single precision representation. On average, our float2 implementation is having errors on the order of $1e-5$ to $1e-7$, which are far better than using regular floats. Moreover, the standard deviation for float2 is also much lower (i.e., the standard deviation for double is on the order of $1e{-}8$ to $1e{-}9$, for float2 of $1e{-}4$ to

1e−5, and for float of 1e+0). Thus, our implementation is getting more stable results with tighter bounds on the error than regular floating point numbers.

Table 2 Results of a global summation for an array of 1,000 elements summed using a single thread on GPU

Sorting	float	double	float2
Unsorted, shuffled (1)	-4.8750e+00	6.1521e-09	-1.9423e-05
Unsorted, shuffled (2)	-2.1250e+00	6.8585e-10	5.2670e-05
Unsorted, shuffled (3)	1.6250e+00	8.4459e-09	-4.3361e-05
Unsorted, shuffled (4)	-5.0000e-01	-1.5134e-09	1.1444e-05
Sorted descending	-7.0000e+00	9.3132e-09	0.0000e+00
Sorted ascending	7.0000e+00	-9.3132e-09	0.0000e+00

Table 3 Results of a global summation for an array of 1,000 elements summed using 1000 threads on GPU

Sorting	float	double	float2
Unsorted, shuffled (1)	-4.8750e+00	6.1521e-09	-1.9423e-05
Unsorted, shuffled (2)	-2.1250e+00	6.8585e-10	5.2670e-05
Unsorted, shuffled (3)	1.6250e+00	8.4459e-09	-4.3361e-05
Unsorted, shuffled (4)	-5.0000e-01	-1.5134e-09	1.1444e-05
Sorted descending	-7.0000e+00	9.3132e-09	0.0000e+00
Sorted ascending	7.0000e+00	-9.3132e-09	0.0000e+00

Table 4 Results of a global summation for an array of 1,000 elements summed using 100 threads on GPU

Sorting	float	double	float2
Unsorted, shuffled (1)	-2.1250e+00	-5.1223e-09	0.0000e+00
Unsorted, shuffled (2)	0.0000e+00	3.1432e-09	6.9618e-05
Unsorted, shuffled (3)	1.0000e+00	-1.3970e-09	7.6294e-05
Unsorted, shuffled (4)	7.5000e-01	-1.8626e-09	-7.6294e-06
Sorted descending	-3.0000e+00	0.0000e+00	0.0000e+00
Sorted ascending	3.0000e+00	0.0000e+00	0.0000e+00

Table 5 Results of a global summation for an array of 1,000 elements summed using 10 threads on GPU

Sorting	float	double	float2
Unsorted, shuffled (1)	-1.0000e+00	0.0000e+00	-1.2207e-04
Unsorted, shuffled (2)	-6.2500e-01	1.2515e-09	1.2207e-04
Unsorted, shuffled (3)	-7.5000e-01	-4.6566e-10	1.2207e-04
Unsorted, shuffled (4)	5.0000e-01	-1.8626e-09	-9.1553e-05
Sorted descending	8.0000e+00	4.4703e-08	3.0518e-04
Sorted ascending	-8.0000e+00	-4.4703e-08	-3.0518e-04

6.3 Do/Undo

In the do/undo program, we consider multiple kernels to handle different operations and their inverses. The program consist of the iterative execution of an operation followed by its inverse using random numbers, e.g., the randomly generated operand x (or array of operands X) is iteratively multiplied and divided by a series of randomly generated operands y (or an array of randomly generated operands Y). The randomly generated operands x and y (or array of operands X and Y) can be either positive or negative and are randomly chosen within an interval whose maximum absolute value is defined by a seed. Figure 11(a) shows the general program framework and Figure 11(b) shows an example of our synthetic program for the multiplication and division. The randomly generated values help to emulate the energy fluctuations in MD simulations.

```
General Code

x = (+/-)rand(seed)
loop
  y = (+/-)rand(seed)
  x = (x op y) op^{-1} y
  print x
end loop
```

```
Example op = * and op^{-1} = /

x = rand(seed)
loop
  y = rand(seed)
  x = (x * y) / y
  print x
end loop
```

Figure 11 General framework of the suite program (a) and one simple example with * and / (b)

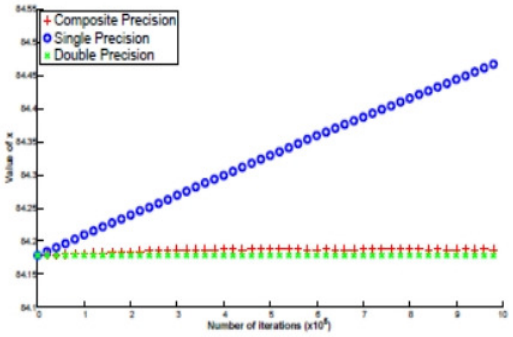
For our assessment, we generate a random x, then repeatedly multiply and divide it by a random y each iteration. We perform this computation with 1,000,000 iterations and we considered different ranges of x and y:

- Trial 1: x = (1, 100), y = (1, 100)
  - Figures 12(a) and 12(b)
- Trial 2: x = (1e5, 1e6), y = (1e−6, 1e−5)
  - Figures 12(c) and 12(d)
- Trial 3: x = (1e−6, 1e−5), y = (1e5, 1e6)
  - Figures 12(e) and 12(f)

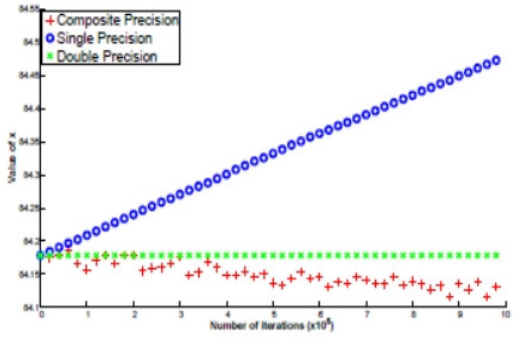
We performed our tests on one GPU of the Tesla S1070 system with single and double precision as well as with our composite precision. We measured both accuracy (in terms of drifting as the simulations evolve) and performance (in terms of the total time needed to execute the 1,000,000 iterations on the GPU). The iterations were performed using a single thread. We also considered two different scenarios: in a first scenario x

was multiplied by y and then divided; in a second scenario x was divided by y and then multiplied.

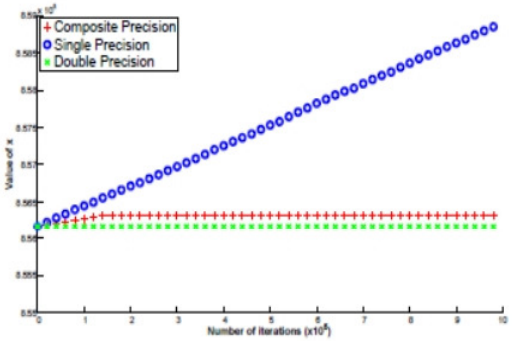
Figure 12 shows the results and associated drifting. Independently from the range of the x and y values and from the order of the operations (multiplication followed by division or vice versa), for single precision computations, we observed the same drifting as in MD simulations shown in Figure 4. The linear growth of the do/undo single precision errors is caused by the lack of bits to represent the real number. The representation is simply cut off at a certain number of digits, and thus the errors are always in the same direction. For double precision, we do not observe any drifting, probably because of the too small number of iterations and the larger number of bits used to represent the values. In all cases, our composite precision significantly corrects the drifting. However, our composite precision multiplication and division operations are still not commutative; indeed, there are different results depending on the ordering of these operations. This is caused by the error calculations in the multiplication and division codes. To find the error for divisions, we calculate the difference between the initial parameter x, and x after one iteration of y*x/y. For multiplications, on the other hand, we multiply the errors together from the previous run. Since the division code scales down the error, while the multiplication scales up the error, we get different results depending on the ordering. Note that the error itself has errors, and therefore scaling in different directions can still affect the final result.



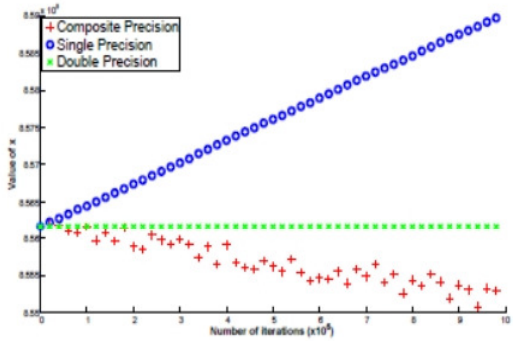
(a) $Op = *, Op^{-1} = \div$ Range: $x = (1, 100), y = (1, 100)$



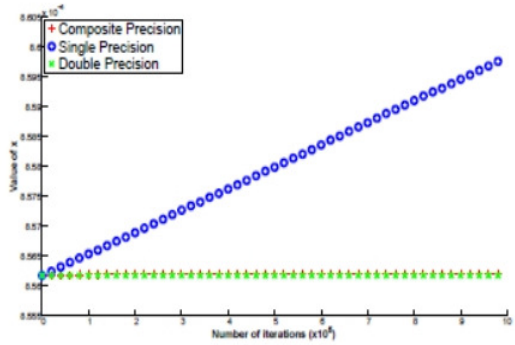
(b) $Op = \div, Op^{-1} = *$ Range: $x = (1, 100), y = (1, 100)$



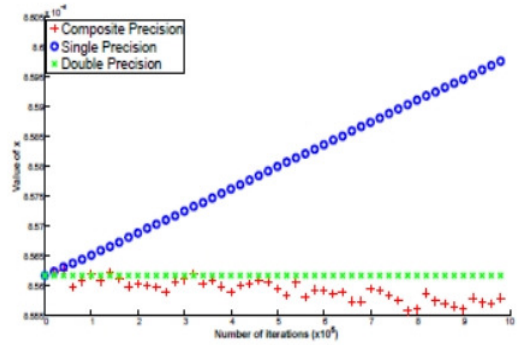
(c) $Op = *, Op^{-1} = \div$ Range: $x = (1e5, 1e6), y = (1e-6, 1e-5)$



(d) $Op = \div, Op^{-1} = *$ Range: $x = (1e5, 1e6), y = (1e-6, 1e-5)$



(e) $Op = *, Op^{-1} = \div$ Range: $x = (1e-6, 1e-5), y = (1e5, 1e6)$



(f) $Op = \div, Op^{-1} = *$ Range: $x = (1e-6, 1e-5), y = (1e5, 1e6)$

Figure 12 Accuracy of do/undo program

An important aspect of our approach is the cost of improving numerical reproducibility and stability. For the three trials in Figure 12, we measured and compared the time to run the 1,000,000 iterations with different precision, i.e., single precision, double precision, and composite precision. The results of these tests are shown in Table 6. As expected, for our synthetic do/undo programs, double precision is, on average, 182% slower than single precision floating point arithmetic. This is even worse, as seen in Figure 1, in actual applications such as our MD codes. The prohibitive cost of double precision computations (three times slower than single precision calculation) does not justify the associated accuracy for routine scientific applications. On the other hand, the reduced computational efficiency due to our composite precision is marginal (7% in average) while the accuracy is comparable to the double precision accuracy, demonstrating that our approach allows us to combine double precision accuracy with single precision performance. The values in the table are average values and each test was repeated three times.

Table 6 Performance of the do/undo program

Trial 1	Op1 = *, Op2 = /		Op1 = /, Op2 = *	
	Avg(s)	Stdv(s)	Avg(s)	Stdv(s)
Single Precision	15.4	0.04	15.95	0.04
Double Precision	44.2	0.26	44.25	0.21
Composite Precision	16.71	0.03	16.97	0.02
Trial 2	Op1 = *, Op2 = /		Op1 = /, Op2 = *	
	Avg(s)	Stdv(s)	Avg(s)	Stdv(s)
Single Precision	15.4	0.03	15.95	0.04
Double Precision	44.16	0.08	44.10	0.02
Composite Precision	16.71	0.04	16.96	0.01
Trial 3	Op1 = *, Op2 = /		Op1 = /, Op2 = *	
	Avg(s)	Stdv(s)	Avg(s)	Stdv(s)
Single Precision	15.4	0.03	15.94	0.04
Double Precision	44.06	0.01	44.06	0.09
Composite Precision	16.74	0.01	16.94	0.03

Chapter 7

EVALUATION OF LIBRARY WITH MD CODE

7.1 Evaluation Goals

By integrating the library into the MD code, we expect to evaluate the library's effect on the drifting described in section 4.2. We expect the energy values to become more constant, with a performance better than double precision.

7.2 Reengineering MD code

Integrating the library into the MD code is not as simple as replacing regular mathematical operations with function calls. Since the library requires we store both the value and error of a single-precision floating point number, we need to allocate additional space in memory for the error. Moreover, the library functions take float2 numbers and return float2 numbers; therefore, we must convert regular floating point numbers to float2 composite precision numbers. Finally, at the end of the program, we need to add in the error to the printed results.

The process of integrating the library with the MD code is shown in Figures 13-15. The underlined red text designates what was added or changed in the MD code for the library to work.

In Figure 13, we see that the main flow of the MD code stays the same; however, there are two main changes to the program. First, for each variable, v, that is involved with calculating the energy (the values we are trying to fix), we add a new variable, v_err, and allocate space for it. At the end of the program, we free that

memory. Secondly, when the program is printing the results, we must add the error variables into the results. This is as easy as it sounds; for each energy that we are printing, we can simply say energy_tot = energy_val + energy_err.

None of the functions described in Figures 1-3 are changed; the flow remains the same. The majority of changes in the MD code are within the function nonbondwlist, which calculates the energies due to nonbonded interactions (Van der Waal's and electrostatic energies). The changes to that function can be seen in Figure 14. The error variables must be passed as parameters, which are then passed down into the pairInteraction function. The results of the calculations are then stored in the globalA array (acceleration array), with the values and associated errors stored separately.

The pairInteraction function is what actually calculates the electrostatic energy, the Van Der Waal's energy, and acceleration of each atom. The changes to pairInteraction are shown in Figure 15. For each single precision floating point variable x, we must create a corresponding float2 variable so that the library functions can work. So, we have written two functions to easily convert floating point numbers to float2. The first takes two floating point numbers – a value and an error – and combines them into one float2 variable. The latter takes a single floating point number and returns a float2. Note that since a single precision floating point number is not perfectly accurate, there is an associated error that is calculated and put into the error slot. The rest of the function is converted in a straightforward way – each operation is replaced with the corresponding library function call. For example, x + y is replaced with my_add(x2,y2). Finally, the only difference at the end of the function is that we

must remember to store the error using the space allocated at the very beginning of the program.

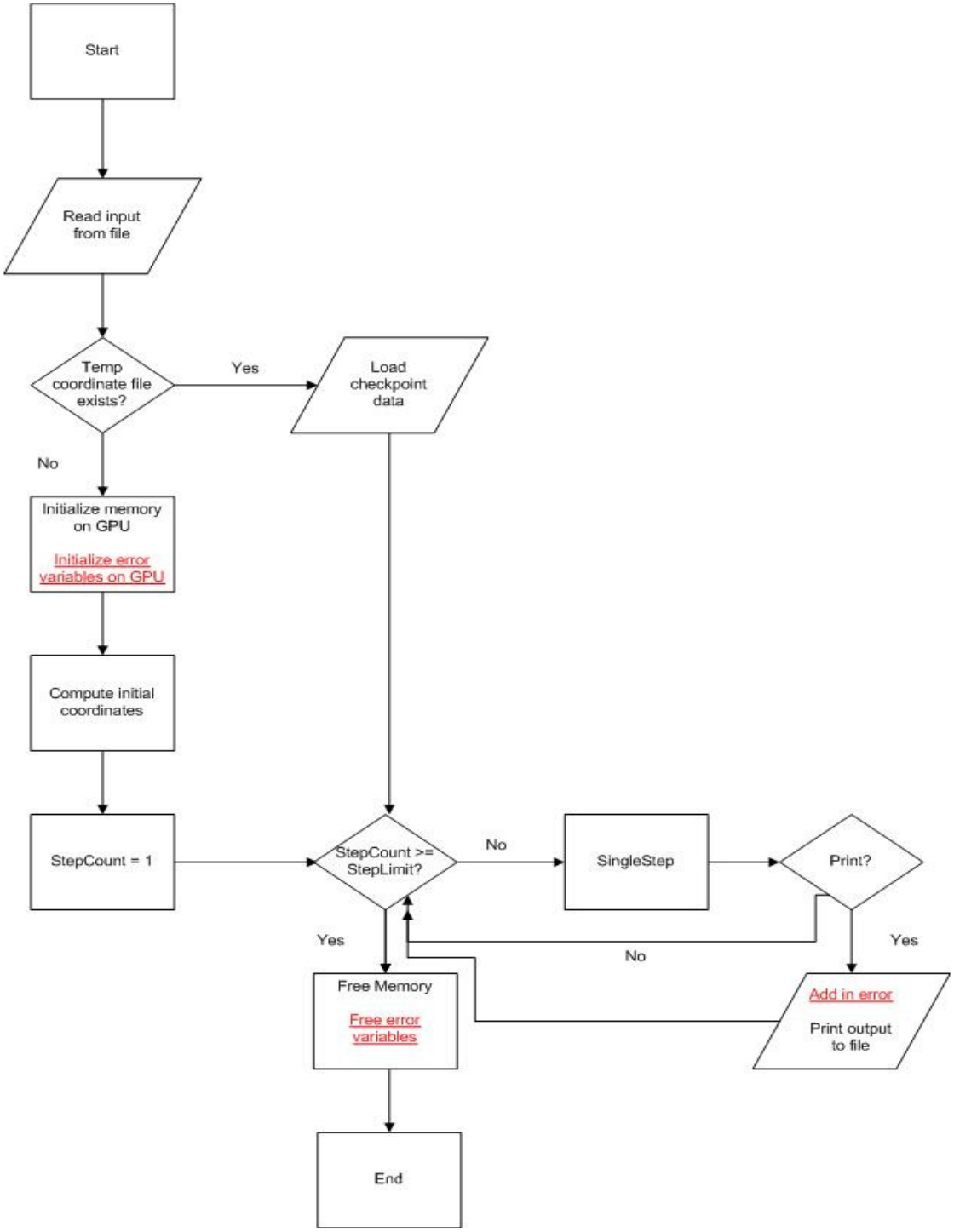


Figure 13 Flow chart of main MD code with library, an extension of Figure 1

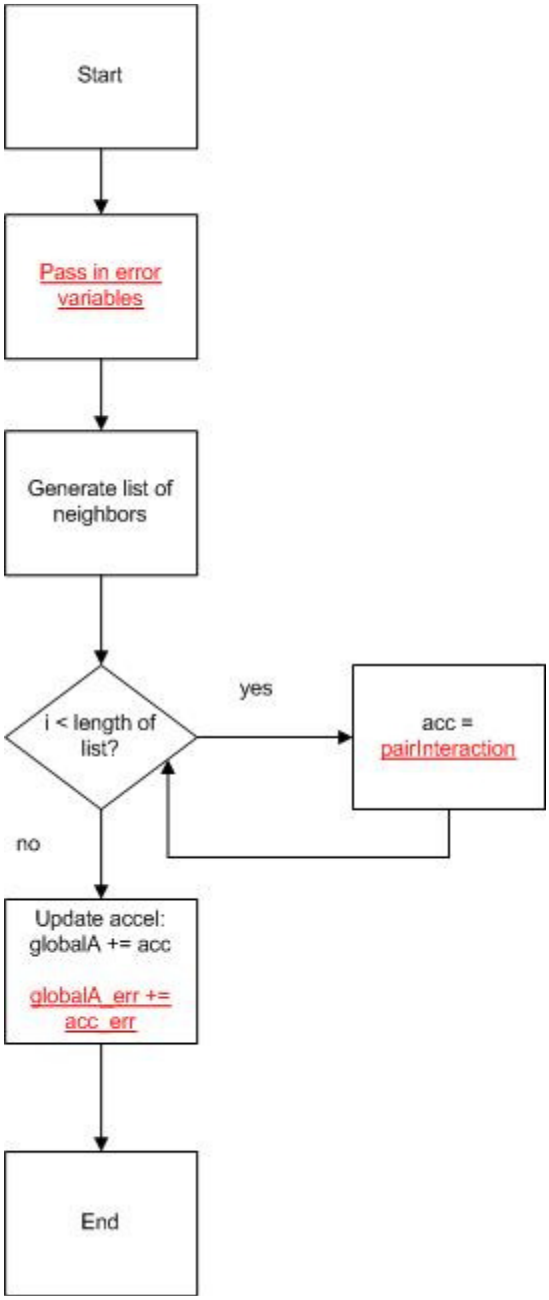


Figure 14 Flow chart of nonbondwlist, which calculates the energy due to nonbonded interaction. An extension of Figure 3

Original Pair Interaction

```
Calculate rij, sigma, and
epsilon

Calculate the force from
Lennard-Jones interaction

Calculate the energy from
Lennard-Jones interaction

Calculate the force from Coulomb
interaction

Calculate the energy from
Coulomb interaction

Zero out force and energy
contributions that are redundant
or impossible

Calculate the acceleration

Store energies

Return acceleration
```

```
Pair Interaction with library

Convert all inputs to float2.
Combine all variables x with
x err

Calculate rij, sigma, and
epsilon using library operations

Calculate the force from
Lennard-Jones interaction using
library operations

Calculate the energy from
Lennard-Jones interaction using
library operations

Calculate the force from Coulomb
interaction using library
operations

Calculate the energy from
Coulomb interaction using
library operations

Zero out force and energy
contributions that are redundant
or impossible

Calculate the acceleration using
library operations

Store energies and errors

Convert acceleration values from
float2 back to one float8

Return acceleration and
acceleration errors
```

Figure 15 Pseudo code for pairInteraction function

7.3 Molecular Systems and Computing Environment

For the study of accuracy and performance, we conducted tests on two different sized molecular systems. The first system consists of 988 water molecules, 18 Na+ ions, and 18 I– ions.. The second, larger system consists of 3665 water molecules, 70 Na+ ions, and 70 I– ions. Both tests were run on a GPU of Nvidia's Tesla S1070 system.

The 988 system was chosen because it is the same system used in 4.2 to describe the drifting. The 3665 system was chosen to have another, larger system for comparison.

7.4 Accuracy Study

To assess how our composite precision floating point arithmetic library improves the numerical reproducibility and stability in the actual MD code, we tested how our library affected the electrostatic and Van Der Waal's energies. We expect that with single precision, as described in section 4.2, the energies will drift instead of staying constant. However, when we apply our library, we expect the values to be constant.

All simulation tests were run for 10,000,000 MD steps, with each MD time step being 1 fs. Therefore the total MD simulation time for each run is 10ns.

Below are the results from the 988 system.

Figures 16-18 show the Van Der Waal's energy. Figure 16 shows the energy without using the library. Figure 17 shows the error calculated by the library, and Figure 18 shows the corrected energy using the library. Notice that the Van Der

Waal's energy drifts about 100 from where it started, and is moving towards zero. The error calculated at each step corresponds to the amount the energy has drifted since the beginning. After adding this error to the original value, the energy remains constant over the 10ns simulation.

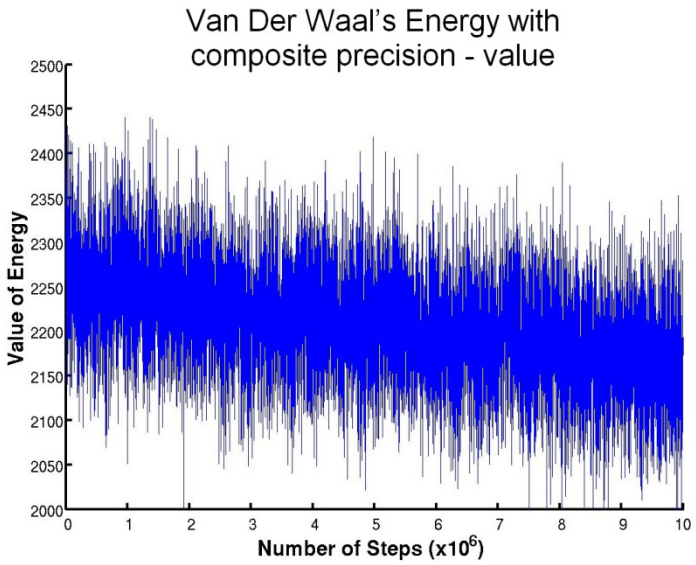


Figure 16 Van Der Waal's energy without adding in error, 988 system

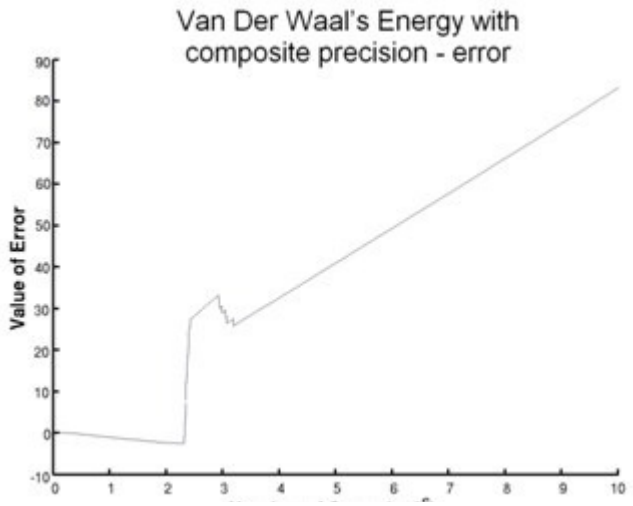


Figure 17 Error calculated for Van Der Waal's energy using library, 988 system

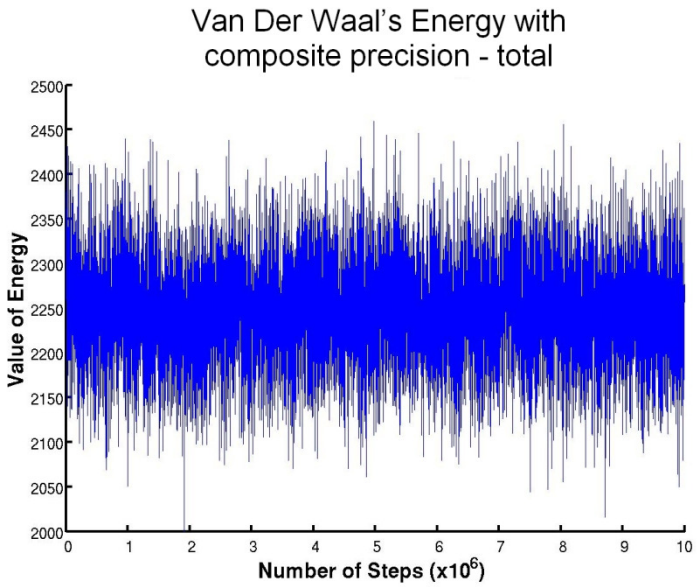


Figure 18 Corrected Van Der Waal's energy using library, 988 system

Figures 19-21 show the electrostatic energy. Figure 19 shows the energy without using the library, Figure 20 shows the error calculated by the library, and Figure 21 shows the corrected energy using the library. Again, the energy is drifting towards zero (notice the values are negative in this case). After adding in the error, we can see from 3ns to 10ns the energy is constant. However, from 0 to 3ns, the amount of error calculated is too high.

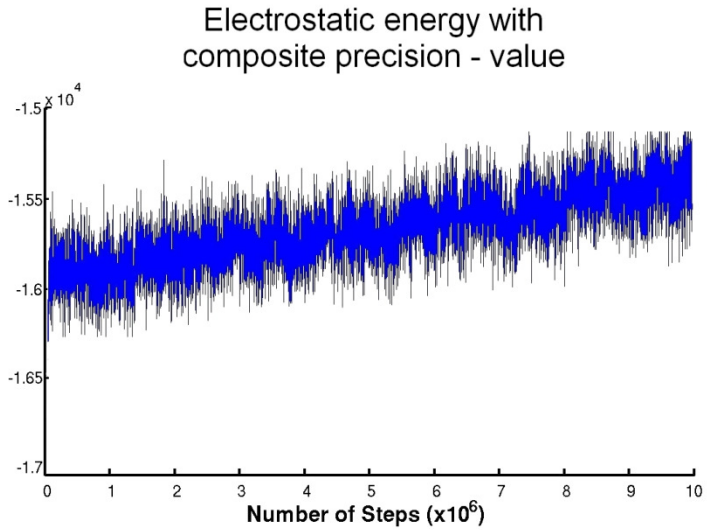


Figure 19 Electrostatic energy without adding in error, 988 system

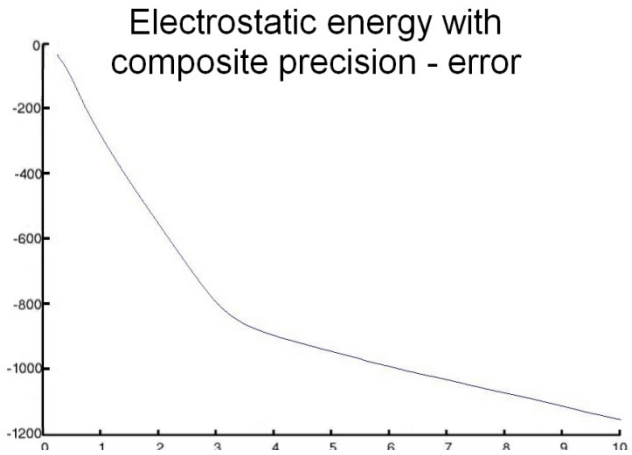


Figure 20 Error calculated for electrostatic energy using library, 988 system

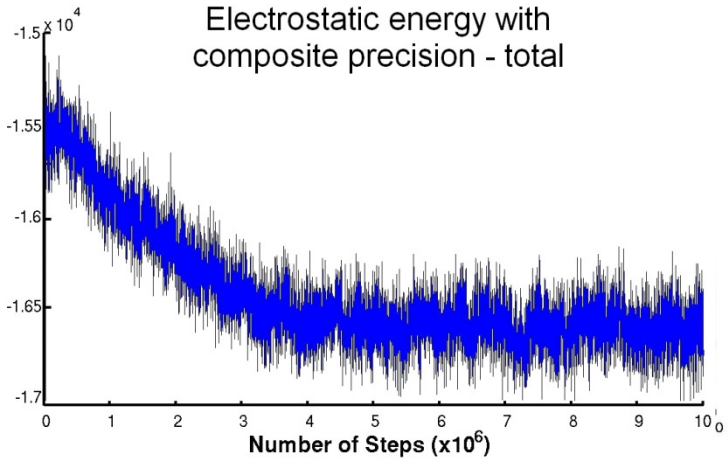


Figure 21 Corrected Electrostatic energy using library, 988 system

Next, we will examine the 3665 system.

Figures 22-24 show the Van Der Waal's energy. Even in this larger system, the energy is still drifting towards zero. After adding in the error, we can see

the energy remain constant over the entire simulation, which verifies what we examined earlier in the smaller system.

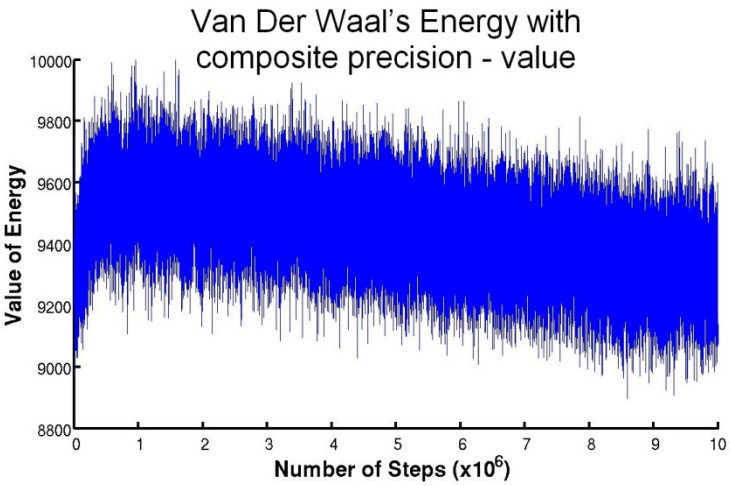


Figure 22 Van Der Waal's energy without adding in error, 3665 system

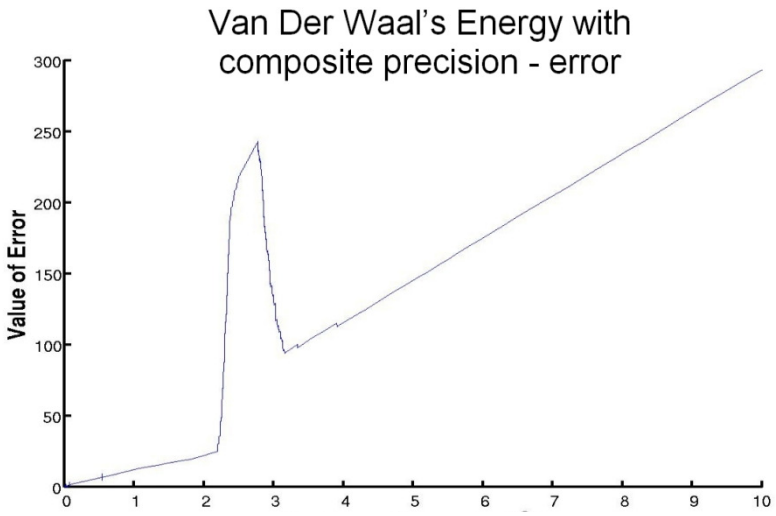


Figure 23 Error calculated for Van Der Waal's energy using library, 3665 system

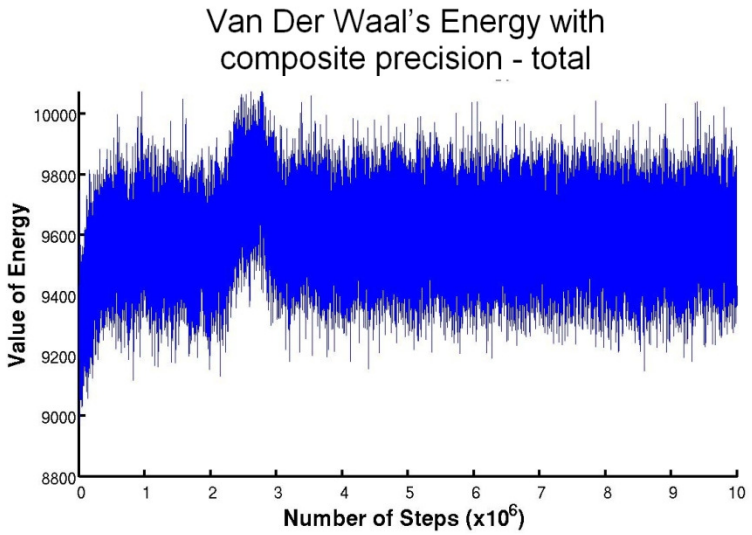


Figure 24 Corrected Van Der Waal's energy using library, 3665 system

Figures 25-27 show the electrostatic energy. The energy is still drifting towards zero, as in the smaller system (notice the values are negative). And again, after adding in the error, we can see from 3ns to 10ns the energy is constant, and from 0-3ns the error calculated is too high. So this confirms what we observed in the smaller system.

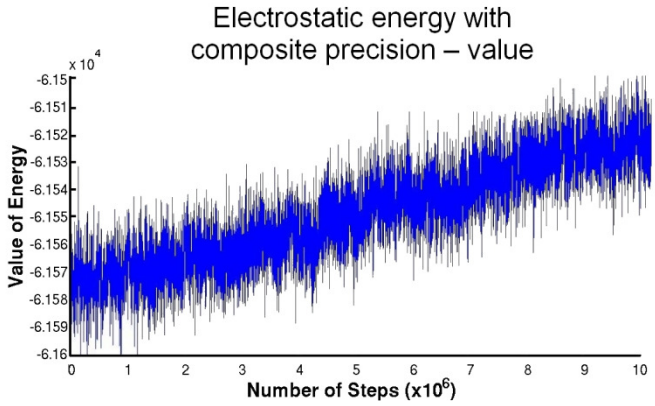


Figure 25 Electrostatic energy without adding in error, 3665 system

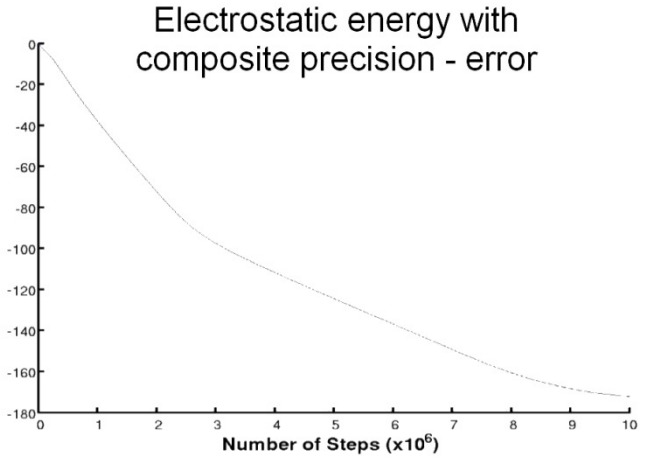


Figure 26 Error calculated for electrostatic energy using library, 3665 system

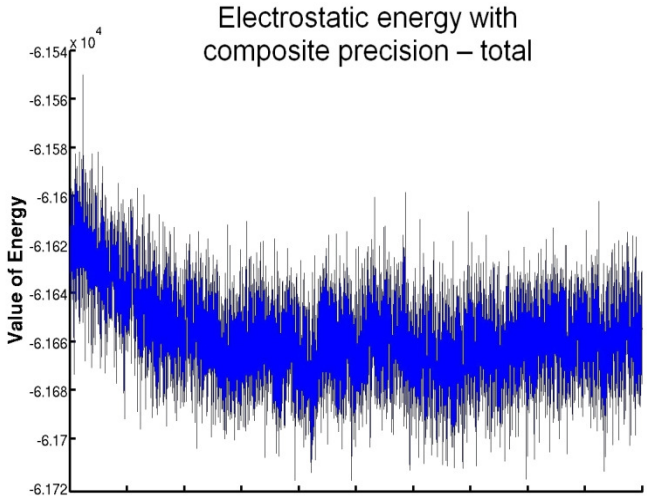


Figure 27 Corrected electrostatic energy using library, 3665 system

We observed that in both 988 and 3665 systems, the Van Der Waal's energy was corrected by the library. However, in both cases there is an anomaly in the error calculated around 3ns. And in both electrostatic calculations, we see that from 0-3ns the error calculated is too high. These anomalies could be explained by recording error that is too high for a few of the atoms, which could be caused by errors in the boundary condition calculation between those atoms. Or the anomalies could simply be the result of mistakes made when integrating the library. In either case, more testing could be done to find the cause, and until the cause is found the error can be scaled down for the offending atoms.

7.4.1 Accuracy Study – Effect of Library on Total Energy

Below are Figures 28 and 29, which show the results of using the corrected Van Der Waal's and electrostatic energies in the 988 and 3665 systems, respectively.

Within the 988 system, we can see that the total energy drifts without the library (single precision). However, when the Van Der Waal's and electrostatic energies are corrected, we can see the total energy looks very similar to the electrostatic graph seen in Figure 21. This is due to the fact that the electrostatic energy is the largest value, and thus contributes the most to the total energy.

Within the 3665 system, we see a similar pattern. The total energy clearly is drifting without the library. And when we use the corrected Van Der Waal's and electrostatic energies, the total energy follows the electrostatic graph seen in Figure 27. Notice that for both the 988 and 3665 systems, the total energy appears to be constant after about 3ns.

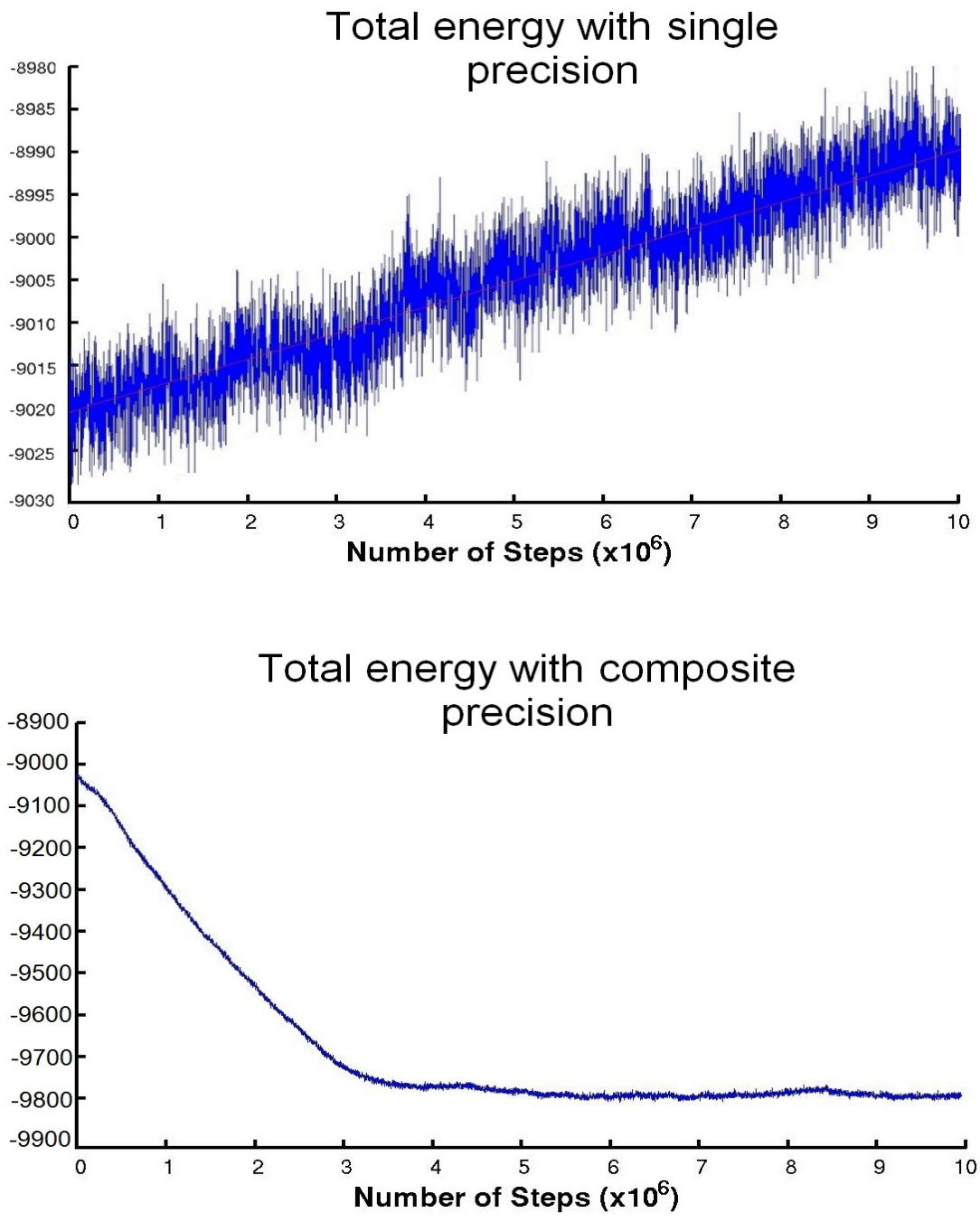


Figure 28 Total Energy, 988 system. a) Without library. b) With library

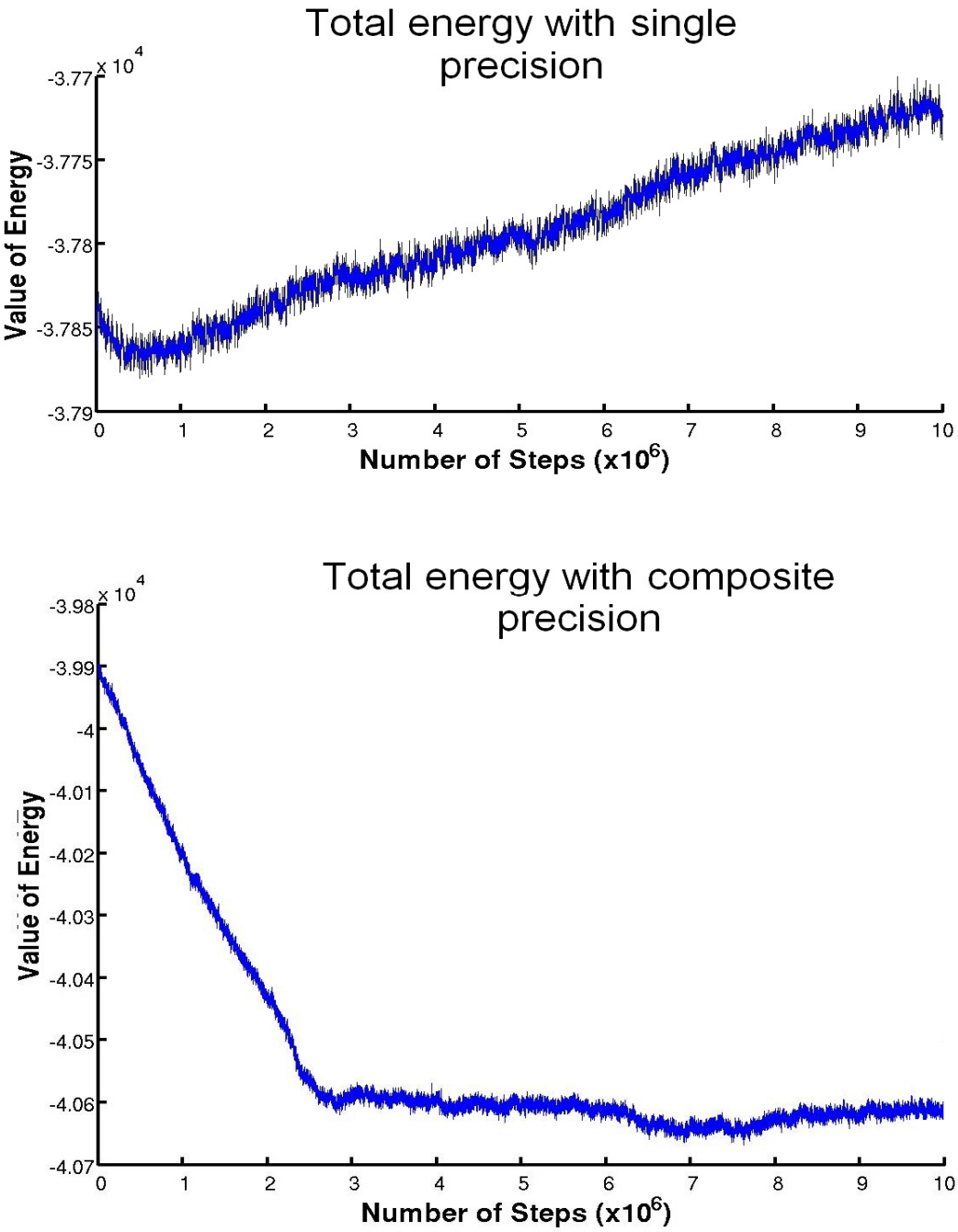


Figure 29 Total Energy, 3665 system. a) Without library. b) With library

7.5 Performance Study

We measure the cost of using the composite precision library with the MD code compared to using single or double precision. To test the performance, we ran simulations of both the large and small systems with single precision, double precision, and composite precision. The length of the simulations was 100,000 MD steps (.1ns). Below is a table summarizing the results. Note that the current implementation of the code does not include any optimization and thus the performance results are preliminary,

Table 7 Performance of library in MD code

Molecular System Size	Precision	Total time	MD steps/sec
3000 atoms	Single	250.8	400
3000 atoms	Double	2742.7	36.46
3000 atoms	Composite	925.4	111.1
11135 atoms	Single	672.3	148.7
11135 atoms	Double	5310.4	18.8
11135 atoms	Composite	2043.9	48.9

In the case of the smaller system, double precision is 10.97 times slower than single precision, while the composite precision is 3.60 times slower. In the larger system, double precision is 7.90 times slower than single precision, while the composite precision is 3.04 times slower. In both cases, although the composite precision library is slower than single precision, it always outperforms double precision.

Chapter 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

In this thesis we show how numerical reproducibility and stability of large-scale numerical simulations with chaotic behavior such as MD simulations is still an open problem when these simulations are performed on multi-threaded systems such as GPUs. We propose to solve this problem using composite precision floating point arithmetic. In particular, we present the implementation of a composite precision floating point library and we show how our library allows scientists to successfully combine double precision accuracy with single precision performance for a suite of synthetic codes emulating the behavior of MD simulations on GPU systems. We also show how the library can be integrated and used within real applications to improve numerical reproducibility and stability.

8.2 Future Work

As discussed in chapter 6.1, integrating the library into an application is not trivial. New variables must be created and allocated memory, while operations must be replaced with function calls to the library. The manual integration of the composite precision library is error-prone and tedious; however the process can be described in terms of a pattern that can be automated. Future work includes the design and implementation of a parser that identifies code sections in which the composite

precision can be beneficial as well as a converter that integrates the library automatically. This converter should automatically optimize the code for performance.

References

[1] J. A. Anderson, C. D. Lorenz, and A. Travesset. General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units. J. Comput. Phys., 227:5342–5359, 2008.

[2] D. H. Bailey. High-precision Floating-point Arithmetic in Scientific Computing. IEEE Computing in Science and Engineering, pages 54–61, 2005.

[3] D. H. Bailey, D. Broadhurst, Y. Hida, X. S. Li, and B. Thompson. High Performance Computing Meets Experimental Mathematics. In Supercomputing ’02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pages 1–12, 2002.

[4] M. Braxenthaler, R. Unger, D. Auerbach, J. Given, and J. Moult. Chaos in Protein Dynamics. Proteins, 29:417425, 1997.

[5] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. J. Comp. Chem., 4:187–217, 1983.

[6] J. Davis, B. Bauer, M. Taufer, and S. Patel. Molecular Dynamics Simulations of Aqueous Ions at the Liquid-Vapor Interface Accelerated Using Graphics Processors. In Submitted to Review, 2009.

[7] J. E. Davis, A. Ozsoy, S. Patel, and M. Taufer. Towards Large-Scale Molecular Dynamics Simulations on Graphics Processors. In BICoB ’09: Proceedings of the 1st International Conference on Bioinformatics and Computational Biology, pages 176–186, 2009.

[8] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. LeGrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande. Accelerating molecular dynamic simulation on graphics processor units. J. Comput. Chem., 30:864–872, 2009.

[9] Y. He and C. H. Q. Ding. Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications. In ICS ’00: Proceedings of the 14th international conference on Supercomputing, 2000.

[10] H. Nguyen. GPU Gems 3. 2008.

[11] NVIDIA. NVIDIA CUDA - Programming Language. 2008.

[12] D. M. Smith. Using Multiple-precision Arithmetic. Computing in Science and Engineering, 5:88 – 93, 2003.

[13] J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, and K. Schulten. AcceleratingMolecular Modeling Applications with Graphics Processors. J. Comput. Chem., 28:2618–2640, 2007.

[14] A. Thall. Extended Precision Floaing Point Numbers for GPU Computation. In Poster at ACM SIGGRAPH, Annual Conference on Computer Graphics, 2006.