Improving Numerical Reproducibility and Stability in Large-Scale Numerical Simulations on GPUs

Michela Taufer*, Omar Padron†, Philip Saponaro* and Sandeep Patel‡
*Dept. of Computer & Inf. Sciences
University of Delaware
Email: {taufer, saponaro}@udel.edu
† New Jersey Center for Science, Tech., and Math. Education
Kean University
Email: padrono@kean.edu
‡Dept. of Chemistry
University of Delaware
Email: patel@udel.edu

***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, comparable to CPU performance.**

To address this problem, we extend the approaches of previous literature to improve numerical reproducibility and stability in MD simulations, while assuring efficiency and performance comparable to that when using the GPU hardware implementation of single precision arithmetic. We present development of a library of mathematical functions that use fast and efficient algorithms to fix the error produced by the equivalent operations performed by GPU. We successfully validate the library with a suite of synthetic codes emulating the MD behavior on GPUs.

***Keywords*-Parallel programming; GPU programming; Molecular Dynamics; Floating-point arithmetic.**

I. Introduction

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). NVE dynamics is also the original method of molecular dynamics, corresponding to the microcanonical ensemble of statistical mechanics [5]. With single precision GPUs, we observe significant drift of the total energy with time over a 30 nanosecond (ns) molecular dynamics simulation of a box of water molecules representing a bulk system at ambient conditions. The components of the potential energy, the electrostatic and Van der Waals (dispersion) 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 [9], [10]. This phenomenon is also observed when round-toward-even operations are used 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.

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 simulations are parallelized: final results can differ significantly among platforms and number of parallel units used (threads or processes). Overall, numerical reproducibility and stability of results (where by "reproducibility and stability" we mean that results of the same simulation running on GPU and CPU lead to the same scientific conclusions) cannot be guaranteed in large-scale simulations. 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 [12], [1], [7], [6]. However, this comes at a higher cost in numerical reproducibility and stability. Threads 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.

In this paper we show how energy drifts observed in MD simulations can be substantially reduced, while maintaining performance comparable to single precision GPUs, by making use of a separate set of mathematical functions for floating-point arithmetic that 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". We present a library of operations that handle this new data type. Since MD codes are very complex to deal with, validation of accuracy and measurement of performance are 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.

The paper is organized as follows: Section II provides a short overview of GPU programming and accuracy issues in GPU calculations; Section III shows the energy drifting in MD simulations; Section IV describes our composite floating-point arithmetic; Section V presents the synthetic suite used for assessing accuracy and performance of our approach; Section VI discusses the state of the art in the field; and Section VII concludes the paper and presents future work.

II. Background

A. 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 such 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 cores of a single multiprocessor (shared memory). Clearly, this constitutes a great deal of raw computing power.

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 than a traditional CPU, code optimization for the GPU involves different approaches, which are described in detail elsewhere [9], [10].

B. Issues with GPU Precision

As pointed out in the CUDA Programming Guide [9], [10], 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) [1] 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. Of course, some of the drift in MD simulations can be eliminated by making use of CUDA functions for floating-point operations that avoid the nonstandard truncation.

III. Energy Drifting in MD Simulations

Molecular Dynamics simulations, being chaotic applications [4], 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. To study the energy behavior of NVE MD simulations on GPUs, we measured and plotted the total energy profile over the course of a 30 ns simulation for the 988 water system using the MD code for GPUs that we presented in other work [6]. We used a time step size of 1 fs, so this test simulation is 30 million MD steps long. The results are shown in Figure 1. 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

[1] On GPUs, the maximum error is stated as the absolute value of the difference in ULPs between a correctly rounded single precision result and the result returned by the CUDA library function

fourth profile (double precision) in Figure 1 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. We attribute the drifting to the lack in numerical reproducibility and stability already observed in conventional distributed systems such as clusters [8]. Here, the effect is significantly enhanced since the simulation is effectively performed on a "cluster" of greater than 32 or 64 cores (processors).

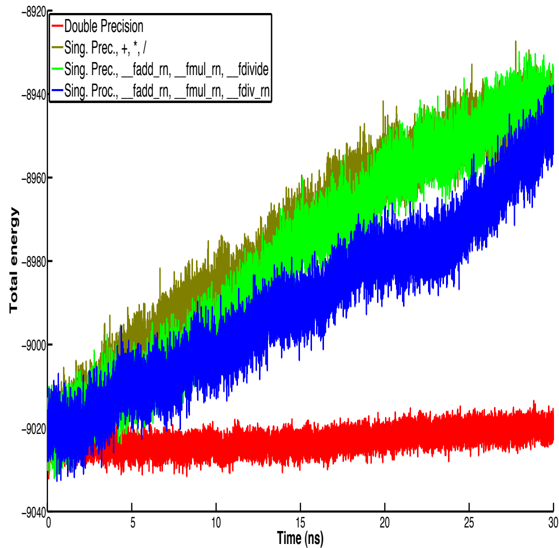


Figure 1. Time profiles of the total energy for simulations of the 988 water, 18 NaI system. Results are shown using default single precision, single precision with correction for nonstandard rounding and truncation, and double precision.

Using double precision does not reduce drifting on current GPU systems for very long MD simulations, longer than 100ns. As also summarized in Table I, double precision arithmetic on currently available GPUs (e.g., Tesla S1070) dramatically reduces performance to levels comparable to that of CPUs (12 times slower).

IV. Redefining Floating-Point Arithmetic on GPUs

A. Composite Precision 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. 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:

$$n = n_{value} + n_{error} \quad (1)$$

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 2). Errors in each calculation are carried through

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

Figure 2. Data structure of float2.

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 totally truncate and 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.

B. 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

Table I
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., __fadd_rn, __fmul_rn, /	457.40
CHARMM-GPU	Tesla S1070	Sing. Prec., __fadd_rn, __fmul_rn, __fdiv_rn	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

for the addition and multiplication are based on algorithms proposed in [13], [8].

The implementation of the composite precision floating-point addition is presented in Figure 3 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.

```
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 3. Algorithm for the composite precision floating-point addition.

For the composite precision floating-point multiplication presented in Figure 4, 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

```
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 4. Algorithm for composite precision floating-point multiplication.

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.

The composite precision floating-point division implementation in Figure 5 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* $\frac{1}{b} - a = 0$. 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 5, extends the algorithm in [13].

```
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 5. Algorithm for the composite precision floating-point division.

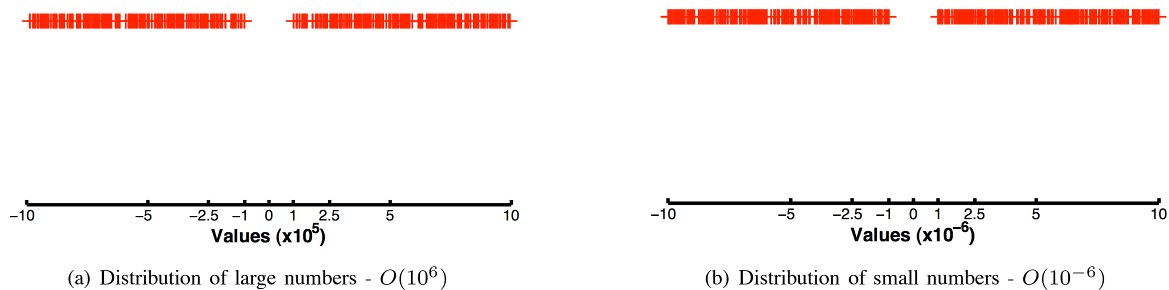
V. EVALUATION

A. Synthetic Suite

MD codes are very complex, thus we developed a suite of synthetic codes that reproduce rounding errors in MD. The suite comprises of several 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. This 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.

B. 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.



(a) Distribution of large numbers - $O(10^6)$

(b) Distribution of small numbers - $O(10^{-6})$

Figure 6. Value distribution used for assessing numerical reproducibility and stability in global summations.

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 numbers filled with very large $O(10^6)$ and very small $O(10^{-6})$ values. The distribution of the values were purposefully made symmetric: whenever we generated a number, the next number generated was its opposite. 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 6(a) and 6(b) show the distribution of the numbers used for the validation: Figure 6(a) shows the numbers with absolute value larger than 1 (up to 10^6) and Figure 6(b) shows the numbers with absolute value smaller than 1 (on the order of 10^{-6}).

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 II 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 III 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 IV and Table V 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.

C. Do/Undo Programs

In the do/undo programs, we consider multiple kernels to handle different operations and their inverses. The programs 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 7(a) shows the general program framework and Figure 7(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.

For our assessment, we generated a random x, then repeatedly multiply and divide it by a random y each iteration. We performed 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 8(a) and 8(b)
- Trial 2: $x = (1e5, 1e6)$, $y = (1e-6, 1e-5)$ - Figures 8(c) and 8(d)
- Trial 3: $x = (1e-6, 1e-5)$, $y = (1e5, 1e6)$ - Figures 8(e) and 8(f)

We performed our tests on one GPU of the Tesla S1070 system with single and double precision as well as with our

Table II
RESULTS OF A GLOBAL SUMMATION FOR AN ARRAY OF 1,000 ELEMENTS SUMMED USING A *single thread on GPU*. DIFFERENT SORTING TECHNIQUES ARE USED. THE EXPECTED RESULT IS 0.

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 III
RESULTS OF A GLOBAL SUMMATION FOR AN ARRAY OF 1,000 ELEMENTS SUMMED USING *1000 threads on GPU*. DIFFERENT SORTING TECHNIQUES ARE USED. THE EXPECTED RESULT IS 0.

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 IV
RESULTS OF A GLOBAL SUMMATION FOR AN ARRAY OF 1,000 ELEMENTS SUMMED USING *100 threads on GPU*. DIFFERENT SORTING TECHNIQUES ARE USED. THE EXPECTED RESULT IS 0.

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 V
RESULTS OF A GLOBAL SUMMATION FOR AN ARRAY OF 1,000 ELEMENTS SUMMED USING *10 threads on GPU*. DIFFERENT SORTING TECHNIQUES ARE USED. THE EXPECTED RESULT IS 0.

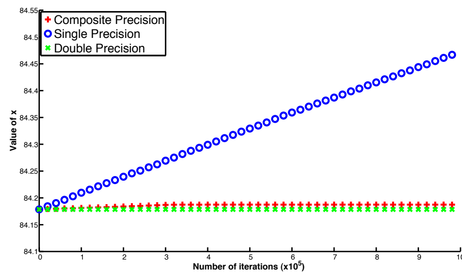
Sorting	float	double	float2
Unsorted, shuffled (1)	-1.0000e+00	0.0000e+00	-1.2207e-04
Unsorted, shuffled (2)	-6.2500e-01	1.2515e-09	1.2207e-04
Unsorted, shuffled (3)	-7.5000e-01	-4.6566e-10	1.2207e-04
Unsorted, shuffled (4)	5.0000e-01	-1.8626e-09	-9.1553e-05
Sorted Descending	8.0000e+00	4.4703e-08	3.0518e-04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

```
General Code

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

(a)
```

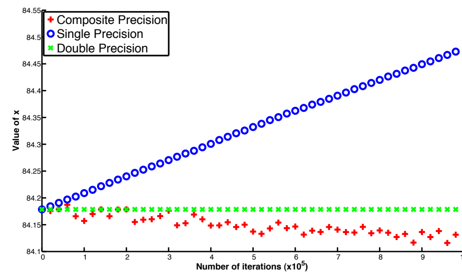
```
Example op = * and op^{-1} = /

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

(b)
```

Figure 7. General framework of the suite program (a) and one simple example with * and / (b).

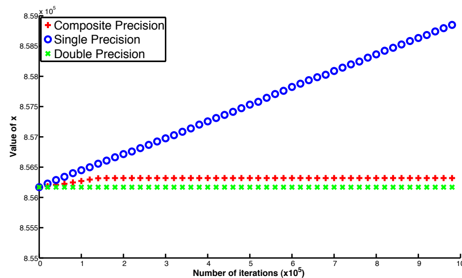
composite precision. We measured both accuracy (in terms of drifting as the simulations evolve) and performance (in terms of the total time needed to execute the 1,000,000 iterations on the GPU). The iterations were performed using a single thread. We also considered two different scenarios: in a first scenario x was multiplied by y and then divided; in a second scenario x was divided by y and then multiplied. Figure 8 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



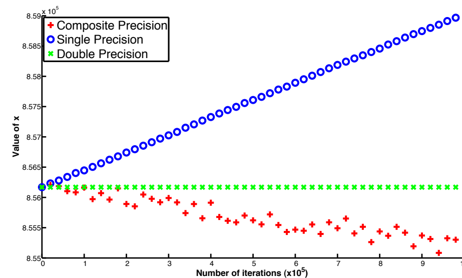
(a) $Op = *, Op^{-1} = \div$ Range: $x = (1, 100), y = (1, 100)$



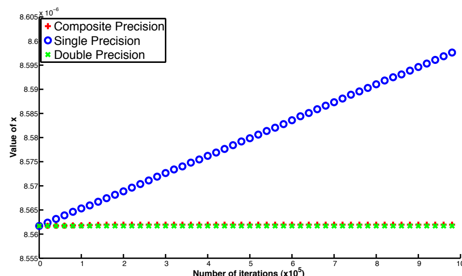
(b) $Op = \div, Op^{-1} = *$ Range: $x = (1, 100), y = (1, 100)$



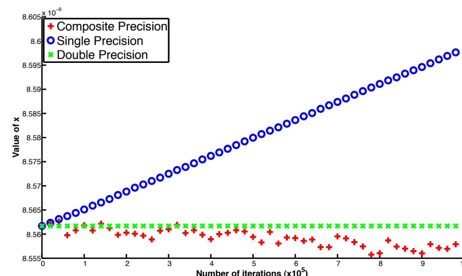
(c) $Op = *, Op^{-1} = \div$ Range: $x = (1e5, 1e6), y = (1e-6, 1e-5)$



(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 8. Accuracy of do/undo program with iterative execution of multiplications and divisions.

simulations shown in Figure 1. For double precision, we do not observed 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.

An important aspect of our approach is the cost of improving numerical reproducibility and stability. For the three trials in Figure 8, 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 VI. 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

Table VI
PERFORMANCE OF THE DO/UNDO PROGRAM WITH ITERATIVE EXECUTION OF MULTIPLICATIONS AND DIVISIONS.

Trial 1	$Op = *, Op^{-1} = \div$		$Op = \div, Op^{-1} = *$	
	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	$Op = *, Op^{-1} = \div$		$Op = \div, Op^{-1} = *$	
	Avg (s)	Stdv (s)	Avg (s)	Stdv (s)
Single Precision	15.40	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	$Op = *, Op^{-1} = \div$		$Op = \div, Op^{-1} = *$	
	Avg (s)	Stdv (s)	Avg (s)	Stdv (s)
Single Precision	15.40	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

combine double precision accuracy with single precision performance. The values in the table are average values and each test was repeated three times.

VI. RELATED WORK

Numerical reproducibility and stability for chaotic applications was addressed for massively parallel CPU-based architectures in [8]. The work in [8] does not address emerging high-performance paradigms such as GPU programming and their novel architectures. An approach similar to ours was theoretically suggested in [13]. We build our work upon these two contributions with MD simulations as the targeted large-scale numerically intensive applications.

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 [11], [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. 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.

VII. CONCLUSION AND FUTURE WORK

In this paper 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 GPU systems. We propose to solve this problem using composite precision floating-point arithmetics. In particular, we present the implementation of a composite precision floating-point library and we show how our library allows scientists to successfully and easily combine double precision accuracy with single precision performance for a suite of synthetic codes emulating the behavior of MD simulations on GPU systems.

Overall, our tests on synthetic codes reproducing MD behavior outline more accurate results than with simple single precision (improving the accuracy of up to 5 orders of magnitude) with almost no loss in performance (7% with our library versus 182% with double precision). Work in progress includes the integration of our library in MD codes.

ACKNOWLEDGMENT

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 NVIDIA University Professor Partnership Program, and by the Computing Research Association through the Distributed Research Experiences for Undergraduates (DREU).

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. E. Davis, A. Ozsoy, S. Patel, and M. Taufer. Towards Large-Scale Molecular Dynamics Simulations on Graphics Processors. In *Lecture Notes in Bioinformatics*, 5462:176–186, 2009.

[7] 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.

[8] 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.

[9] H. Nguyen. GPU Gems 3. *Addison-Wesley Professional*. 2007.

[10] NVIDIA. *NVIDIA CUDA - Programming Language*. 2008.

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

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

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