# An Efficient Arbitrary Precision Mathematical Library for Accurate and Fast MD Simulations on Single Precision GPUs – The *sqrt* function

**Student: Omar Padron**
**Supervisor: Michela Taufer**
**Computer and Information Sciences, University of Delaware**

## 1. Project Description

Molecular Dynamics (MD) simulations are an excellent target for GPU acceleration since most aspects of MD algorithms are easily parallelizable. Enhancing the performance of MD can allow the simulation of both larger time scales and larger length scales. Figure 1 illustrates some types of calculations that are possible at differing length and time scales [1]. Ideally, simulations would be able to attain all-atom resolution on time scales of microseconds to milliseconds. With coarse-grained resolution, even longer time scales approaching seconds may be possible. However, the ultimate goal of any simulation is atomistic resolution of very large length scales over very long time scales, i.e., essentially continuum physics with atomic detail. The utilization of parallel architectures such as GPUs constitutes a step towards this goal.

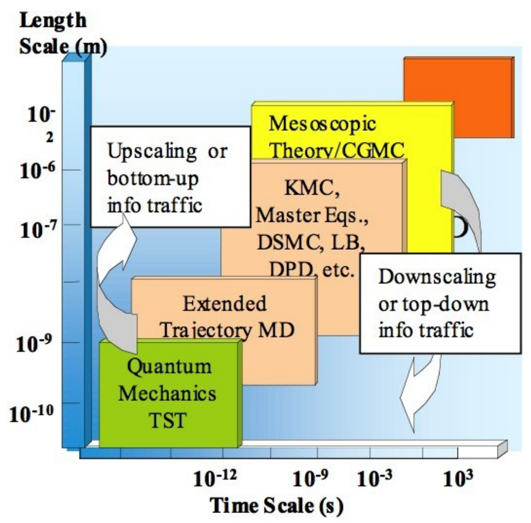


**Figure 1:** An illustration of some of the types of theoretical physical calculations that are possible at varying time and length scales (Figure reproduced from Reference [1]).

In the past year, the Global Computing Lab at the University of Delaware has designed and implemented a CUDA implementation of MD code modeled after CHARMM [2] in terms of the force field functional forms, neighbor list structure, and measurement units. Our GPU-CHARMM code includes bonded and nonbonded atom interactions. Results of MD simulations with our GPU-CHARMM show that several floating-point operations (i.e., division and sqrt) are not IEEE compliant and lack in accuracy when executed on single precision GPUs. Our results are also supported by the CUDA documentation [3]. At the same time, such compliance and accuracy are provided on double precision GPUs but at a too high performance cost. Table 1 shows this cost in terms of MD steps/sec for a NaI solution system (the higher, the better). The total execution time for 10 million MD steps (10 nanoseconds simulation time) was used to obtain this data. The performance of the same simulations on double precision GPUs is 8 times slower and comparable to CPU performance [4, 5]. Another aspect that affects performance on double precision GPUs is the fact that performance optimizations available for single precision GPUs, e.g., using the texture unit as a means of improving random access performance for table lookups, are not available in double precision devices yet. So for example, for benefiting from the texture unit on double precision GPUs, turnaround solutions are needed in which floating-point numbers are embedded in C union types.

An efficiently implemented arbitrary-precision mathematical library for GPUs can provide the scientist with a solution to this problem because it would allow the execution of accurate floating-point operations on faster single precision GPUs.

This proposal has two goals:

- Implement an arbitrary precision mathematical library for floating point operations on GPUs.
- Study the library's accuracy and efficiency trade-offs with real MD applications using our GPU-CHARMM code.

| GPU-CHARMM | MD step/sec |
|---|---|
| **GeForce 9800 GX2** | 260.88 |
| **Quadro FX 5600** | 246.37 |
| **GTX 280 (double precision)** | 31.79 |

**Table 1:** Performance of our GPU-CHARMM code for the NaI solution system compared among different GPUs. The total execution time for 10 million MD steps (10 nanoseconds simulation time) was used to obtain this data.

Arbitrary precision mathematical libraries are not new. In the 70s' and 80s', several open-source libraries for CPUs were developed. The lessons learned in implementing the CPU libraries in the past will help us in delivering an open-source arbitrary precision mathematical library to the community of scientists using GPUs.

The rest of this research proposal is organized as follow: Section 2 presents preliminary results motivating the need for an arbitrary precision library and presenting the knowledge built in the past year by the Global Computing Lab at the University of Delaware. Section 3 describes the research goals for this summer internship. Section 4 presents the summer timeline and the starting bibliography. Finally, Section 5 presents the equipment available to the group members for this project.

## 2. Project Methodology

### 2.1 GPU-CHARMM Code for MD Simulations

The GPU-CHARMM code implemented at the Global Computing Lab emulates the CHARMM code with its force field and parameter settings [4, 5]. The code includes both bonded and nonbonded interactions. Bond, angle, and dihedral potentials are computed using a similar list approach. For the bonds, each thread iterates through all atoms bonded to an atom *i* and accumulates the total bond forces. For the angles and dihedrals, each thread iterates through the atoms that are involved in an angle or a dihedral with *i* and calculates the appropriate interactions. Unlike the nonbonded lists, these lists are constructed once on the CPU at the beginning of the simulation, copied to the GPU, and never need to be modified.

Nonbonded interactions (Lennard-Jones and electrostatics) are calculated by a single kernel in which each thread iterates through the neighbor list for a single atom *i* and accumulates the interactions between atom *i* and all its neighbor list entries. The texture cache is used for reading the coordinates of the neighbor atoms since they are not contiguous in memory. Shifted force forms are used for the electrostatic and Lennard-Jones potentials so that both energies and forces go smoothly to zero at the cutoff $r_{cut}$. The Verlet list approach [6] is used to construct the neighbor list. Briefly, a list is constructed for each atom containing all atoms within a cutoff $r_{list}$, where $r_{list} > r_{cut}$. This way, the list only needs updating whenever an atom has moved more than $0.5( r_{list} - r_{cut})$. The list is constructed on the GPU as follows. Each thread checks the distance between an atom *i* and all other atoms, and adds to *i*'s neighbor list those atoms that are within $r_{list}$ of *i*. This process is accelerated by having each block take advantage of shared memory using a previously described tiling approach [7].

### 2.2 Study of Constant Energy MD Simulations

NVE dynamics is the original method of molecular dynamics, corresponding to the microcanonical ensemble of statistical mechanics [8]. NVE MD simulations are also called constant energy simulations because they are performed in a closed environment with constant number of atoms, constant volume, and constant energy.

We observed a major problem with the total energy of NVE MD simulations executed on single precision GPUs. The total energy, which should remain constant as the simulation evolves, systematically diverged from its initial value, converging toward zero both for negative and positive energies. This phenomenon is not observed on double precision GPUs. To study the phenomenon and isolate the causes, we considered different GPUs and MD parameters. Table 2 summarizes the characteristics of the GPUs considered in this study: single-precision GPUs with CUDA 1.1 - without round-to-nearest-even rounding (SPO); single-precision GPUs with CUDA 2.0 - with round-to-nearest-even rounding for *, + and - (SPW); and double precision GPUs with CUDA 2.0 (DPW). The MD simulations are performed with different parameters summarized in Table 3.

| **GPU characterization** | |
|---|---|
| **Single precision, CUDA 1.1 (SPO)** | All floating-point operations do not uses round-to-nearest-even rounding and are not IEEE compliant |
| **Single precision, CUDA 2.0 (SPW)** | +/- and * **only** use round-to-nearest-even rounding and are IEEE compliant |
| **Double precision (DPW)** | All floating-point operations use round-to-nearest-even rounding and are IEEE compliant |

§

**Table 2:** Type of GPUs

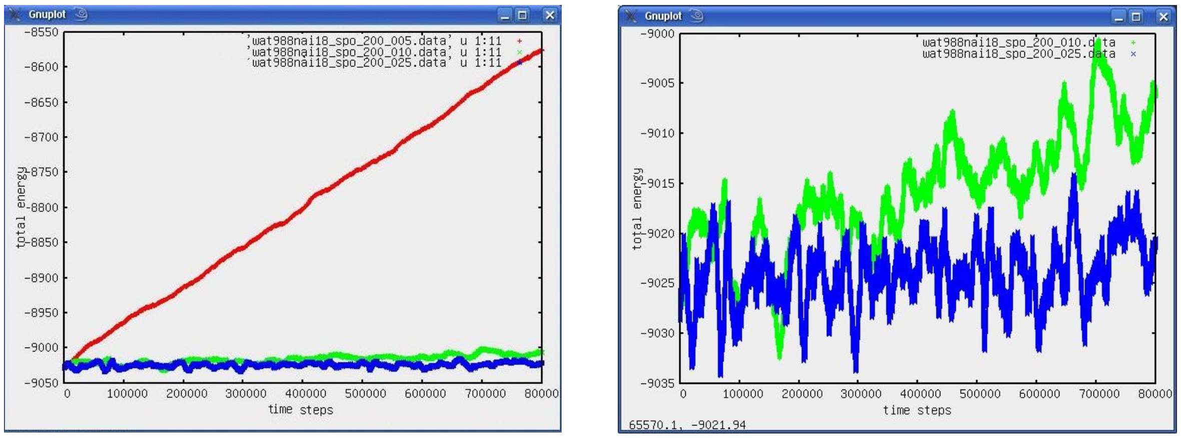
| **MD simulation characterization** | |
|---|---|
| **Step size (fs)** | 0.05, 0.10, 0.25, 0.50, 0.80, 1.00, 1.60, 2.00 |
| **Initial system configuration** | Different random seeds, e.g., 100, 768.5, 1200 |
| **Total MD steps** | 80K, 2,000K |

**Table 3:** MD parameter setting

We applied an n-factor analysis: we took into consideration one single parameter in Table 3 at a time. To study the effect of the *step size*, we ran the same MD simulation with the same number of steps but different step sizes on a single precision GPU and CUDA 2.0 (SPW). Figure 2a shows the total energy over 80K steps with step size equal to 0.05fs (red line), 0.10 fs (green line), and 0.25fs (blue line). Figure 2.b zooms in on Figure 2.a and shows that the deviation from the constant energy is also present for larger step sizes than 0.05fs, i.e., 0.10fs, but is not visible with a step size of 0.25fs or larger. Overall, Figure 2 shows that, even for short simulated intervals, MD simulations with small step sizes present significant total energy drifting.

The random seed in MD simulations affects the initial system configuration. To study the potential impact of the *initial system configuration*, we ran the same MD simulation with different seeds on a single precision GPU and CUDA 2.0 (SPW). Figure 4 shows the total energy for two of the three seeds in Table 3. In all cases the simulation shows the same energy drifting. Therefore, we can conclude that the initial configuration of the MD system is not responsible for the drift.

Figure 4 shows that *longer simulations* present energy drift even for larger step sizes (2fs) when executed on a single precision GPU with CUDA 1.1 (red line) and CUDA 2.0 (green line) but have constant total energy when executed on a double precision GPU. The MD simulation in Figure 4 consists of 20,000K MD steps (40ns). The energy drift was not visible for the same MD step size when the same simulation was executed over a shorter simulated time of 80K MD steps. The figure also provides us with a first indication of the cause of the energy drifting. On single precision GPUs with CUDA 1.1, all the floating-point operations including multiplications, additions, and subtractions do not use the round-to-nearest-even rounding and the drifting is the largest observed. With CUDA 2.0 the latter three operations use the round-to-nearest-even rounding but none of the other operations do (in particular division and sqrt); a slightly smaller drift is still observed. On double precision GPUs all the operations are IEEE complaint and no drift is measured. This suggests that the cause of the energy drift must be related to the way the IEEE floating-point operations are performed on the single precision GPUs.



(a) (b)

**Figure 2:** Effect of different step size on divergence of total energy in MD simulations

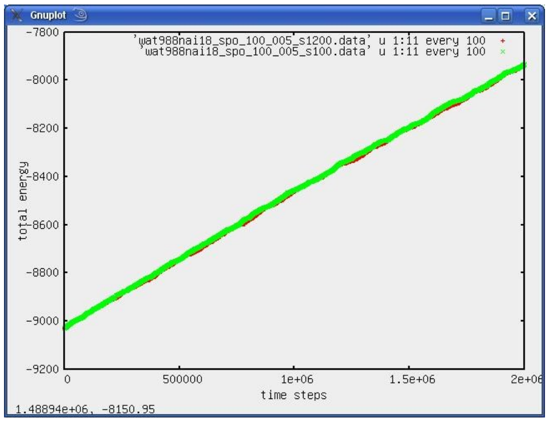


**Figure 3:** Effect of different random seeds on divergence of total energy in MD simulations

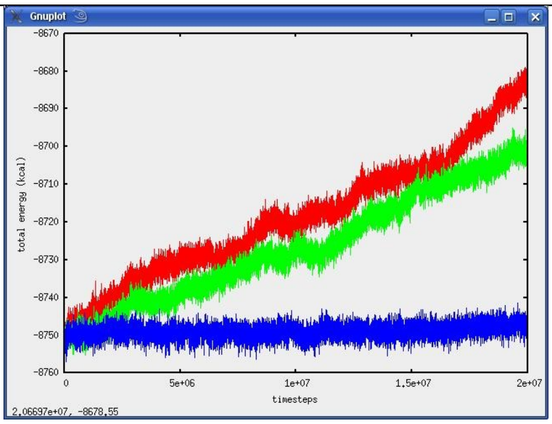


**Figure 4:** Long MD simulation on single prec. GPUs (red and green) and double prec. GPU (blue)

The reviewer could argue that the energy deviations are due to an incorrect MD algorithm. It is known that, to be correct, the fluctuations in total energy of MD simulations should be proportional to the time step size. To evaluate the correctness of the MD code, we considered the constant energy simulation and we plot the standard deviation of the total energy as a function of time step size for short simulations of 50,000 steps.

Figure 6 shows the standard deviation of total energy values for different step sizes (0.05fs, 0.1fs, 0.25fs, 0.5fs, 0.8fs, 1fs, 1.6fs and 2fs) on the three GPUs in Table 2. We know from the results in Figure 2 that for a short simulation, the energy drifts only for small step sizes. From Figure 5 we know that the drift is larger on SPO. SPO and SPW show a different behavior from DPW for step sizes less than 0.2fs (x-axis equal to 20). This difference is more accentuated for SPO than SPW.

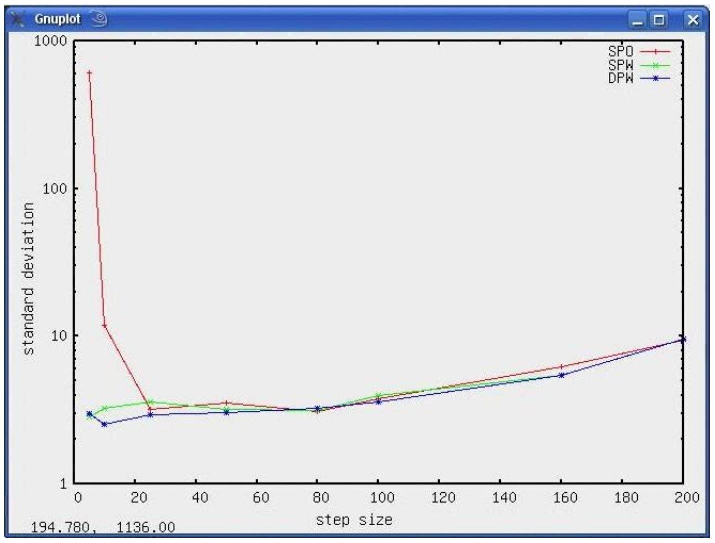


**Figure 6:** Standard deviation of total energy values in MD simulations with different step sizes and types of GPU

## 2.3 Reproducing Drift in Synthetic Code

Since MD codes are normally very complex, we designed and implemented a suite of synthetic programs that show similar drift as our GPU-CHARMM code. Because they are less sophisticated, the synthetic programs allow us to search for effective solutions in a controlled testing environment. The programs consist of the iterative execution of an operation followed by its inverse using random numbers, e.g., the randomly generated operand x (array of operands X) is iteratively squared and then square-rooted. The randomly generated operand x (array of numbers X) is taken to be nonnegative and are randomly chosen within an interval whose maximum value is defined by a seed. Figure 7.a shows the general program framework and Figure 7.b shows an example of a synthetic program. The random generated values emulate fluctuation in MD simulations.

*General code:*

```
x = (+/-) rand (seed)
loop
      x = op⁻¹ (op (x))
      print x
end loop
```

(a)

*Example of code with op = sq and op$^{-1}$ = sqrt* (*i.e.,* sq($x$) = $x^2$ = $x$*$x$)

```
x = rand (seed)
loop
      x = sqrt(x * x)
      print x
end loop
```

(b)

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

Our suite of synthetic programs includes these operations: division, multiplication with reciprocal, and *sqrt*. The programs can be executed on both CPUs and GPUs. Table 4 summarizes the equations considered in our first version of the suite, the values for x and y used for our testing, and the number of iterations per loop.

| Equation | x values | y values (range) | Num. iterations |
|---|---|---|---|
| NEED TO UPDATE WITH SQRT | NEED TO UPDATE WITH SQRT | NEED TO UPDATE WITH SQRT | NEED TO UPDATE WITH SQRT |

**Table 4:** Equations and operands used for studying the drifting in synthetic programs

When running the programs on single precision GPUs using CUDA 2.0 with different operand settings, we always observe a drift from the CPU results that can converge toward 0, +Inf, or –Inf based on the value signs (positive or negative) and value ranges (close to zero or very large values). Figures 8-10 shows three examples of our testing. Figure 8 **[MAKE THIS FIGURE]** presents results with *sqrt* as the main operation in the body of the loop and multiple threads contributing to the final constant positive value x, which is the sum of an array of values $x_i$. Overall we observe that the x values do not remain constant (as expected) but drift from their original values depending on the sign of x, the range of the randomly generated numbers y used in the body of the loops, and the number of loop iterations.

## 3. Goals for Summer Internship

The GCLab group has been working on a CPU version of an arbitrary precision division based on a variable-length array of digits. For the summer, another student in the group will implement a first version of this division for GPUs. I will work on the implementation of a software sqrt for CPU and GPU. Under the supervision of Dr. Taufer, we will address hardware optimizations for my code that are GPU dependent, e.g., use of shared memory for more efficient memory access of data. Therefore, the goals for the summer are to learn CUDA programming, and then convert my CPU sqrt function to work for the GPU efficiently. A testing component will conclude my summer internship. The testing will include two main aspects: accuracy and performance. First we will make sure that the software sqrt function, contrary to the default sqrt currently available on the GPU, does not cause divergence in GPU simulations. Then, we will measure the cost of the software sqrt in terms of performance. Ideally we would like to keep the cost of our function below 10%.

## 4. Timeline and Starting Bibliography

The detailed plan of the summer internship is as follows:

Week 1: Work on a web page for the research; learn about the other members in the lab
Week 2: Run tests on CPU and GPU to confirm the presence of divergences with sqrt
Week 2 – 4: Learn CUDA programming on the GPU
Week 5 – 7: Implement the sqrt function on CPU and GPU
Week 7: Prepare a poster detailing the project
Week 8: Present poster, debug and optimize GPU sqrt
Week 9 – 10: Continue debugging and optimizing GPU sqrt
Week 10: Prepare and give a presentation for the group summarizing my summer experience
Write final report of the research and the achievements.

The proposed research is supported by this related work [9-20]. During the internship, we will critically read and discuss this work.

## 5. Equipment

This project is supported by research equipment at the Global Computing Lab, led by Dr. Michela Taufer. The lab includes:

- A homogenous cluster of three high-end workstations (Dual Quad Core Intel(R) Xeon 2.66GHz) each hosting two different types of GPUs: 2 Nvidia Graphics Card GeForce 9800GX2, 2 Nvidia Graphics Card GeForce GTX280, and 2 Nvidia Graphics Card Quadro FX5600
- A hybrid cluster composed by: 6 quad-core computing nodes, 3 Tesla S1070, each connected to two computing nodes, and a front-end node for compilation and job submissions. The nodes are interconnected by Infiniband and Gigiabit Ethernet.

## 6. References

[1] D.G. Vlachos: A Review of Multiscale Analysis: Examples from Systems Biology, Materials Engineering, and other Fluid-surface Interacting Systems. Adv. Chem. Eng. 30, 1--61, invited (2005).

[2] 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. Comput. Chem. 4:187 (1983).

[3] NVIDIA CUDA Programming Guide 2.0, NVIDIA, 2008.

[4] J.E. Davis, A. Ozsoy, S. Patel, and M. Taufer: Towards large-scale molecular dynamics simulations on graphics processors. In Proceedings of the International Conference on Bioinformatics and Computational Biology (BICoB), April 2009, New Orleans, Louisiana, USA.

[5] J.E. Davis, B.A. Bauer, M. Taufer, and S. Patel: Molecular Dynamics Simulations of Aqueous Ions at the Liquid-Vapor Interface Accelerated Using Graphics Processors. Journal of Computational Physics, 2009 (Submitted).

[6] M.P. Allen and D.J. Tildesley: Computer Simulation of Liquids. Oxford: Clarendon Press, 1987.

[7] H. Nguyen (ed.): GPU Gems 3, Boston: Addison-Wesley, 2008, chapter 31.

[8] Y. Hida, X. S. Li and D. H. Bailey: Algorithms for Quad-Double Precision Floating Point Arithmetic. IEEE Symposium on Computer Arithmetic, 2001.

[9] D.M. Priest: Algorithms for Arbitrary Precision Floating Point Arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic,* 1991.

[10] J.R. Shewchuk: Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Technical Report CMU-CS-96-140, Carnegie Mellon University, 1996.

[11] J.R. Shewchuk: Robust Adaptive Floating-Point Geometric Predicates. In Proceeding of the 12th Annual ACM Symposium Comput. Geom, 1996.

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

[13] MPFR C library for multiple-precision floating-point computations. http://www.mpfr.org/

[14] D. B. Bailey *et al.*: ARPREC - High-Precision Software Directory. http://crd.lbl.gov/~dhbailey/mpdist/

[15] NVIDIA CUDA Compute Unified Device Architecture: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[16] Online book from CUDA programming course at UIUC. http://courses.ece.illinois.edu/ece498/al/Syllabus.html

[17] D. H. Bailey: High-precision Floating-point Arithmetic in Scientific Computation. Computing in Science and Engineering, May–June 2005.

[18] Y. He and C.H.Q. Ding: Using Accurate Arithmetics to Improve Numerical Reproducibility And Stability. Journal of Supercomputing, 2001.

[19] R. Howell: Comparison of Two Arbitrary-precision Arithmetic Packages. http://people.cis.ksu.edu/~rhowell/calculator/comparison.html, 2001.

[20] R. Howell: Arbitrary-Precision Division. http://www.cis.ksu.edu/~howell/calculator/division.pdf, 2000.