Mixed-Precision Models for Calculation of High-Order Virial Coefficients on GPUs

Chao Feng[*], **Andrew Schultz**[†], **Vipin Chaudhary**[*], **David Kofke**[†]
[*]*Department of Computer Science and Engineering*
[†]*Department of Chemical and Biological Engineering*
University at Buffalo, The State University of New York
Buffalo, NY 14260
Email: {chaofeng, ajs42, vipin, kofke}@buffalo.edu

***Abstract*—The virial equation of state (VEOS) is a density expansion of the thermodynamic pressure with respect to an ideal-gas reference. Its coefficients can be computed from a molecular model, and become more expensive to calculate at higher order. In this paper, we use GPU to calculate the 8th, 9th and 10th virial coefficients of the Lennard-Jones (LJ) potential model by the Mayer Sampling Monte Carlo (MSMC) method and Wheatley's algorithm. Two mixed-precision models are proposed to overcome a potential precision limitation of current GPUs while maintaining the performance benefit. On the latest Kepler architecture GPU Tesla K40, an average speedup of 20 to 40 is achieved for these calculations.**

***Keywords*-Virial Equation of State; Lennard-Jones Potential; Mayer Sampling Monte Carlo; Wheatley's Algorithm; Mixed-Precision; GPU**

I. Introduction

Nowadays one graphic processing unit (GPU) could contain thousands of cores and a complicated memory layout where thousands of threads could be organized and launched in a parallel way. General-purpose computing on graphic processing units (GPGPU) is the application of GPUs to perform general purpose computation traditionally done by CPUs [1]. Compute Unified Device Architecture (CUDA) [2] is a parallel computing platform and programming model introduced by NVIDIA in 2006, with which the GPUs can be utilized for general purpose processing. In the CUDA programming model [3], data parallelism is realized by launching a kernel which could consist of a large number of CUDA thread blocks and executes on many GPU cores to process different data elements. Data can be accessed in multiple memory types, ranging from fast register to shared memory to global memory, which is comparatively slow.

In recent years, CUDA has been applied across diverse research areas such as physics, chemistry, biology, and finance, for acquiring high computational power. Many problems have been ported to GPUs, and remarkable computational performance has been achieved. Molecular dynamics (MD) and Monte Carlo (MC) simulations on GPU have been two particularly successful examples, because large amounts of sequential time is always required in these areas [4] [5] [6] [7] [8] [9] [10]. For example, Friedrich *et al.* [4] developed an all-atom protein MD simulation on GPU that can be more than 700 times faster than a conventional implementation on a single CPU core. Hou *et al.* [5] developed a scalable algorithm for GPU MD simulation of solid covalent crystals using sophisticated many-body potentials, with an acceleration of 650 over a contemporary CPU. Stone *et al.* [6] have given an analysis of several MD algorithms developed on GPU. As an example of an MC implementation, Preis *et al.* [7] calculated the critical temperatures of phase transitions in the two- and three-dimensional Ising models and obtained substantial acceleration. Levy *et al.* [8] developed two different algorithms for lattice spin models that led to speedup of 70- to 150-fold. Quinn and Abarbanel [9] did research on data assimilation by GPU accelerated Path Integral Monte Carlo method and reported a speedup of about 300. Anderson *et al.* [10] presented a massively parallel method obeying detailed balance and implemented it for a system of hard disks on the GPU with a maximum speedup of 100.

In this paper, we use GPU to calculate the coefficients appearing in the virial equation of state (VEOS), one of very few equations which provide an exact and tractable connection between molecular and macro-scale descriptions in statistical mechanics [11]. The simplicity of the VEOS makes it very easy to apply in engineering calculations. It can be applied to compressed gases and supercritical fluids at conditions of considerable practical interest. The virial coefficients B_{N} are the temperature-dependent coefficients of the number density appearing in the VEOS. They are formally calculated as integrals over sums of products of Mayer functions for a system. The order N of coefficients relates to the dimension of the integral. Singh and Kofke [12] first proposed the Mayer Sampling Monte Carlo (MSMC) method to calculate virial coefficients, and using MSMC they evaluated for several temperatures the virial coefficients of Lennard-Jones (LJ) potential model up to 6th order (wherein B_6 is given as a 15-dimensional integral). Schultz and Kofke [13] then calculated the 6th, 7th and 8th virial coefficients using MSMC. Schulz *et al.* [14] further implemented the MSMC on GPU to calculate values of the 6th, 7th and 8th virial coefficients at more temperatures, and achieving speedup of two orders of magnitude compared to the CPU implementation. However, they did not find it feasible to calculate the coefficients at even higher orders

Table I
SEQUENTIAL TIME OF CALCULATION OF VIRIAL COEFFICIENTS OF LJ POTENTIAL WITH 1% UNCERTAINTY, FOR A REDUCED TEMPERATURE T = 1.0.

Order, N	Sequential Time
2	0.05 seconds
3	16.08 seconds
4	2.16 hours
5	8.27 hours
6	6.41 days
7	2.15 months
8	4.11 years
9	62.26 years

because the direct evaluation of the integrand value is very inefficient when order is greater than 8.

In an important advance, Wheatley [15] recently proposed an algorithm to evaluate the integrand value more efficiently than the direct method. This algorithm scales exponentially with N in time and memory. With this new approach, Wheatley evaluated the 11^{th} and 12^{th} virial coefficients of the hard-sphere potential model, and 9^{th} and 10^{th} virial coefficients of a soft-sphere model. Still, even with this advance, sequential calculation of high-order virial coefficients with a useful precision is very time consuming. For instance, to calculate the virial coefficients of LJ potential model with 1 percent relative uncertainty at a reduced temperature of 1.0 using MSMC, the sequential time required according to the coefficient order is shown in Table I. For B_8, four years are required to get the result with required precision, while for B_9, more than sixty years of sequential time are needed. Therefore, it is quite necessary and helpful to accelerate calculation of high-order virial coefficients. GPU CUDA offers an opportunity to achieve this goal.

Accordingly, in the present paper we use a GPU implementation of Wheatley's algorithm in MSMC calculations of 8^{th}, 9^{th}, 10^{th} virial coefficients of the LJ potential model. However, in this application a precision problem is encountered in evaluation of the integrand for configurations where particles are far apart and the integrand value is near zero. Wheatley mentioned this precision problem in his paper [15] for the soft-sphere calculation, and he was able to circumvent it through a simple truncation scheme. To resolve the precision problem and get speedup on GPUs, we propose two mixed-precision models which utilize both GPU and CPU.

Obtaining an accurate and precise result while achieving best performance has been a major concern for many researchers in multi-core computing, where high-precision data types are unavailable or expensive. Göddeke and Strzodka first [16] presented a mixed-precision defect correction approach to accelerate vector and mtrix operations in Finite Element (FEM) simulations using GPU. They [17] further followed the track to adapt the mixed-precision iterative refinement methods to FPGAs. Langou *et al.* [18] exploited single-precision operations and resorted to double-precision at critical stages of iterative refinement method while attempting to provide the full double-precision results. Their research group later [19] [20] applied the mixed-precision method in sparse-matrix computations and some linear algebra algorithms. In addition, emulating higher precision value with lower precision numbers is another way to resolve precision problem. Thall [21] introduced emulated-precision method to GPU by demonstrating 'doublefloats', a software-based emulated-precision floating-point numbers for GPU computation. Lu *et al.* [22] designed and implemented a GPU-based emulated-precision library to enable applications with high precision requirement to run on GPUs.

In section II we briefly describe the methods to calculate the virial coefficients-MSMC and Wheatley's algorithm. In section III we introduce a parallel algorithm design of the methods. In section IV we propose two different mixed-precision models to achieve accurate result on GPUs. Then in section V we present experiment results and discuss them before concluding in section VI.

II. METHOD

The VEOS can be expressed as follows [11]:

$$\beta P/\rho = 1+B_2(T)\rho+B_3(T)\rho^2+\cdots+B_n(T)\rho^{n-1}+\ldots \quad (1)$$

where P and ρ are the pressure and number density respectively, $\beta = 1/k_{\text{B}}T$ with T the absolute temperature and k_{B} the Boltzmann constant. The classical virial coefficients B_{N} are calculated by an 3(N-1)-dimensional integral (for models defined in a 3-dimensional space) as [23]

$$B_N = \frac{1-N}{N!}\int\ldots\int f_B(\mathbf{r}^N)d\mathbf{r}_{12}d\mathbf{r}_{1N} \quad (2)$$

The integrand of N^{th} order virial coefficients is defined as a summation over all the biconnected graphs G on N vertices, with each vertex corresponding to a molecule. Each biconnected graph is represented as the product of Mayer functions $f(r_{ij})$:

$$f_B(\mathbf{r}^N) = \sum_G [\prod_{ij\in G} f(r_{ij})] \quad (3)$$

The Mayer function $f(r_{ij})$ for any pair of particles labeled i and j is formally defined by

$$f_{ij} = e^{-\beta u(r_{ij})} - 1 \quad (4)$$

where $u(r_{ij})$ is the pair potential between particle i and j separated by a distance r_{ij} (where we now assume a spherically-symmetric potential). The expression for $u(r_{ij})$ is different for different potential models. In particular,

the LJ potential model is defined by the potential energy function:

$$u(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}] \quad (5)$$

with ϵ the depth of the attractive energy well and σ the size of the particle. From this point on, all temperatures will be given in units of ϵ, so T represents $k_B T/\epsilon$, and all lengths will be expressed in units of σ.

A. Mayer Sampling Monte Carlo Method

The MSMC method is an efficient method to evaluate the configurational integral given in Eq. (2) via Monte Carlo importance sampling [12]. In MSMC, the simulation is performed in an infinite volume without any periodic boundaries. The configurations of the particles are generated as a Markov chain via an importance-sampling Metropolis Monte Carlo process [12] [13] [24]. The virial coefficient is calculated as a ratio of the desired target integral (LJ potential model) over a known reference integral using overlap sampling [13]. We adopt the hard-sphere potential model as the reference system because the virial coefficients of hard spheres are known. The working equation of overlap sampling in MSMC is

$$B_N(T) = B_{N,0}\frac{<\gamma/\pi>_{\pi}/<\gamma_{OS}/\pi>_{\pi}}{<\gamma_0/\pi_0>_{\pi_0}/<\gamma_{OS}/\pi_0>_{\pi_0}} \quad (6)$$

where γ is the integrand of the target integral, π is the absolute value of γ, the subscript 0 indicates a value for the hard-sphere reference integral, the angle brackets specify an ensemble average weighted by π, and γ_{OS} is the overlap function in terms of the absolute value of target and reference integrands:

$$\gamma_{OS} = \frac{\pi\pi_0}{\alpha\pi_0 + \pi} \quad (7)$$

where α is the optimization parameter selected for the convergence of the calculation.

B. Wheatley's Algorithm

One significant step in MSMC is the evaluation of the integrand γ for each sampled configuration. For N particles, there are in total $2^{N(N-1)/2}$ possible graphs on them (including singly and unconnected graphs). Clearly, the number of total graphs increases rapidly when N becomes large (see Table II), and the fraction of biconnected graphs (which are those summed to evaluate f_{B}) also continues to grow. It is prohibitive to calculate the integrand directly for $N > 8$.

Wheatley's algorithm [15] employs an indirect way to evaluate integrand of the virial coefficients, and it is especially efficient at high order. It starts by calculating the sum of all graphs, which is easily given as the product of $(1 + f_{ij})$ over all i and j. Subsequently, the biconnected graphs are obtained by subtracting first, contributions from the disconnected graphs, and then contributions from graphs with articulation points (singly-connected graphs). Let f_Q denote for a given configuration the sum of all $2^{N(N-1)/2}$ graphs, f_C be the corresponding sum of all connected graphs, and f_A be the sum for all the articulated graphs. This process is demonstrated in Figure 1. The subtracted terms are formed as products of appropriate partitions of the full set of points, and the procedure relies on evaluation of f_Q, f_A, f_C, and f_B for all subsets of the points. This is accomplished via a recursive scheme that culminates in evaluation of f_B for the full set.

Table II
TABLE2. NUMBER OF TOTAL GRAPHS AND BICONNECTED GRAPH PERCENTAGE

N	Total Graphs $2^{N(N-1)}$	Biconnected Graph Percentage
2	2	50%
3	8	13%
4	64	16%
5	1,024	23%
6	32,768	34%
7	2,097,152	48%
8	268,435,456	62%
9	68,719,476,736	74%
10	35,184,372,088,832	83%
11	36,028,797,018,963,968	89%
12	73,786,976,294,838,206,464	93%
13	302,231,454,903,657,293,676,544	96%

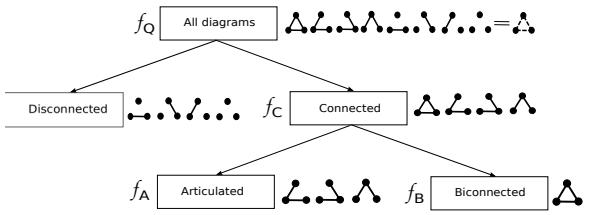

Figure 1. The basic idea of Wheatley's algorithm

III. PARALLEL ALGORITHM DESIGN

In the CUDA model, each CUDA thread is designed to execute one Monte Carlo simulation in which the sum of the integrand function γ and the sum of the overlap function γ_{OS} are calculated repeatedly for many configurations of the LJ particles. All simulations are independently executed to take the same number of Monte Carlo steps on GPU. The results of all Monte Carlo simulations are transferred back from GPU to CPU after completion. The final results are evaluated on CPU via Eq. (6). The reference system and the target system are implemented separately for the purpose of running the reference and target system with a different number of Monte Carlo steps. The target system typically requires more steps to reach the desired uncertainty.

In our algorithm, the main factor limiting performance is

the memory usage per thread. The primary memory usage is the arrays needed for Wheatley's algorithm. Three arrays are used for storing f_Q, f_C, f_B values of all subsets of N particles; each array requires 2^N memory units. f_Q is needed only for calculating f_C, therefore the array used for storing f_Q can be reused for f_A. We use bitmaps to index all subsets of the particles. For example, to calculate virial coefficient B_3, we first number the particles as 0, 1 and 2. Then 3 bits are used to represent 8 subsets formed by 3 particles. For example, '001' represents the set which include only particle 0, '010' has only particle 1 and '101' has particles 2 and 0. With the bit manipulation, we could loop over all the subsets of N particles, and do other manipulations needed to generate and operate on partitions of each subset.

Additional memory usage is required for the 2-dimensional array used for storing all current particle 3D coordinates. There are (3*N*-1) particle coordinates to be stored (with one particle always located at the origin). In addition, another (3*N*-1) particle positions are required for storing previous particle coordinates, which are needed when an MC trial is rejected. The Mayer function values f_{ij} between all pairs needs and additional $N(N-1)/2$ memory units. All these data are declared as 'float' with 32-bit floating point data type.

Because our parallel algorithm design is simple, with each thread executing independently, we do not use a number of GPU features. The shared memory is not utilized in the implementation because first, shared memory with only 48KB per block currently is too small to be helpful, and second, independent threads do not require use of shared memory. In addition, there is no need for synchronization or communication between threads.

IV. Mixed-Precision Model

In section II-B, we mentioned the procedures in Wheatley's algorithm in which there is subtraction. The numerous subtractions cause a precision problem due to strong cancellation between terms. The cancellation is most severe when the particles are far apart, due to the cancellation between $(f + 1)$ and 1 factors in the terms being subtracted. For high-order coefficients and high temperature, 64-bit floating point data type is not enough to acquire an accurate result, so GPU is inadequate.

The direct resolution to this problem is to use a data type with more precision than 'double'. Therefore, we need the help from CPU. The gcc compiler offers two kinds of floating point data types with more precision digits on CPU architecture than 'double' : 'long double'and '__float128'. They each use 16 Bytes of memory. 'long double' could provide 20 decimal precision digits while '__float128' has 35 decimal precision digits.

CPU and GPU computations proceed independently, handling disjoint subsets of the configurations. The GPU handles the configurations which can be accurately processed with 64-bit precision data type; the CPU handles all other configurations. The final result is obtained as the sum of GPU integral with CPU. We note that the mixed-precision model is very successful because the time spent on CPU computation is trivial compared with the GPU computation, because the number of configurations requiring the high-precision calculation is much less than those that do not require it. A key feature of the approach is in focusing the CPU calculation just on those configurations that require its higher precision.

So the next issue to discuss is how to effectively partition configurations between the GPU and CPU just using the 64-bit data type. There are two approaches developed here to achieve this goal: partitioning by critical biconnected distance, and by integrand value.

A. Partition by Critical Biconnected Distance

This approach is based on biconnected distance (BD), defined as follows. Given a configuration, we construct a graph by checking whether the distance between each pair is greater than a cutoff value *X*. If it is greater, then there is no edge between two vertices in the corresponding graph representing the configuration; otherwise, there is an edge between them. Then the BD is the smallest value of X that results in a biconnected graph for the configuration when this construction is performed.

When the configuration is spread out, the integrand f_B is very small, and also the BD is relatively large. Hence, probability of visiting a configuration should decrease with increasing BD. However, when there is a precision problem, the probability of accessing the configurations which are spread out starts to grow inappropriately. The smallest non-zero magnitude of the integrand that can be resolved by the 'double' data type is approximately 10^{-16}, and any configuration having a π value that is in fact less than this will instead compute (incorrectly) to a value of order 10^{-16}. Once this point is reached, π becomes flat with increasing BD, and the probability to visit a configuration below this precision limit will (again, incorrectly) increase with BD. We must avoid including such configurations in the MC averages. Accordingly, we define the critical biconnected distance (CBD) as the turning point where this inaccuracy takes hold and the trend of the probability begins to change. We use this criterion to screen out configurations that are not suitable for processing with the GPU, and we relegate to the CPU the evaluation of their contribution. Identification of the CBD can be done by making the histogram of BD. Figure 2 shows the relationship between the BD and its probability for B_8 with $T = 2$. To generate this histogram, we force the biconnected distance of the configurations to fall into the region near the minimum, in order to collect data without spending too much time on configurations not relevant to the plot. From Figure 2 we can see that, as expected, the CBD

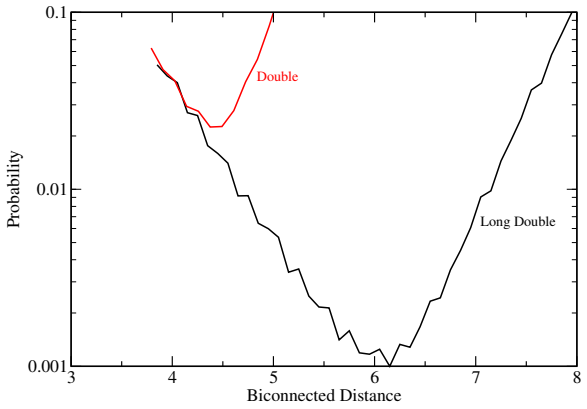

Figure 2. Probability density to observe a configuration, as a function of its biconnected distance (BD). The lines should decrease monotonically with BD, but precision errors cause them to increase for large BD, as shown. The critical biconnected distance is identified as the location where the lines no longer decrease with BD.

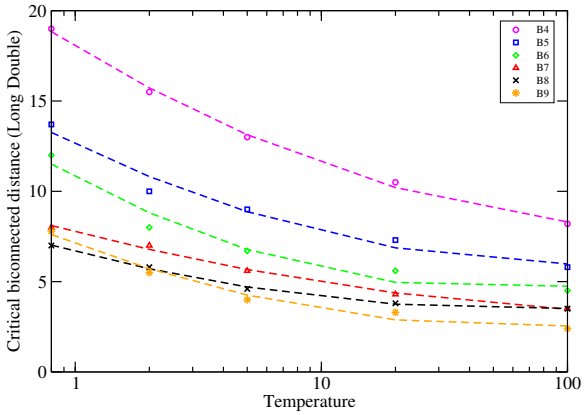

Figure 3. Critical biconnected distances for 'long double'. Points show values from experiment, and dashed lines are the fit to a quadratic polynomial in $\ln(T)$.

with 'long double' is greater than the CBD with 'double'(6.0 vs. 4.5, respectively).

We can in this manner determine the CBDs for several different temperatures at each order, and then generalize them by fitting the points to a quadratic polynomial. From the fitted function, we could calculate the CBD of any temperature and coefficient order. Figure 3 and 3 show the points and fitted functions of CBDs from virial coefficients B_2 to B_9.

Taking B_8 as an example, the fitted functions to calculate the CBDs of both 'double' and 'long double' are

$$CBD_{double} = 0.087[ln(T)]^2 - 0.987ln(T) + 4.772 \quad (8)$$

$$CBD_{long\ double} = 0.180[ln(T)]^2 - 1.515ln(T) + 6.676 \quad (9)$$

where T is again the reduced temperature.

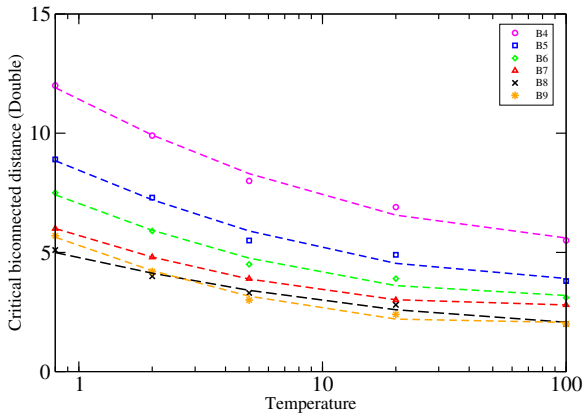

Figure 4. Critical biconnected distance for 'double'. Points show values from experiment, and dashed lines are the fit to a quadratic polynomial in $\ln(T)$.

In the GPU implementation, GPU is first used to compute the major part of integral with data type 'double' for configurations where $BD < CBD_{\text{double}}$. CPU further computes the integral with 'long double' for configurations where $CBD_{\text{double}} \leq BD < CBD_{\text{long double}}$. In particular, for the CPU calculations, if $BD < CBD_{\text{double}}$ or $BD > CBD_{\text{long double}}$ for a configuration, then the integrand value equals to 0 and attempts to sample the configuration are rejected; otherwise, the integrand is calculated with 'long double'.

B. Partition by Integrand Value

For the second approach, the integrand value itself is used for partitioning between GPU and CPU. This second approach is simpler than the first one, and it does not require extra effort during the computation. It is the method adopted by Wheatley to screen inaccurately weighted configurations. His calculations were performed on a CPU, and for the soft-sphere model that he studied, he found that no correction was required for neglect of these configurations.

The precision problem appears when the integrand value is sufficiently small, so this makes a natural basis for the screening decision. To find a good truncation value for the data type 'double', we calculate integrand values using both the data type 'double' and the more precise '__float128'. Ideally, if there were no precision problem, the integrand value calculated by 'double' and '__float128'would be the equal for the same configuration. A plot of the integrand values calculated by 'double' versus those computed via '__float128' clearly shows the region where the precision problem appears. Figure 5 presents this construction for configurations encountered in calculations of B_6, B_8, and B_{10}, respectively. From Figure 5, we can see that over the range of π values considered, the majority of the points are located the upper region of $y = x$, which means that the value calculated by '__float128' reaches values smaller than

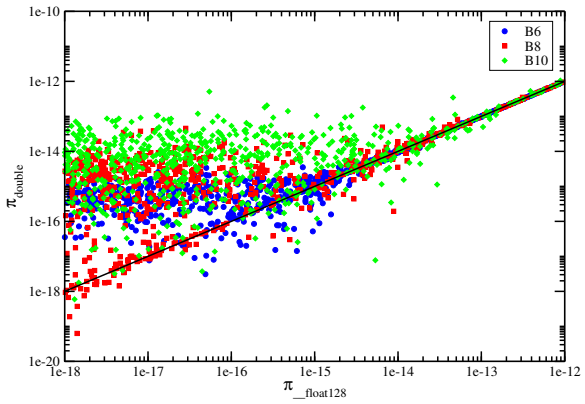

Figure 5. Construction used to determine proper truncation for integrand value calculated by 'double'. Solid black line is $y = x$.

can be resolved by 'double'. For B_6, when π_{double} is bigger than 3.6×10^{-15}, the difference between it and the accurate value is small; for B_8, this threshold occurs at about of about 8×10^{-14}; for B_{10}, it is about 10^{-13}. We conclude from this analysis that π_{double}=10^{-12} is a safe truncation value for all order of virial coefficients examined here.

Thus, in the GPU implementation, using data type 'double' with 16 digits of precision, we reject any configuration where the calculated integrand is less than 10^{-12}. In the CPU implementation, we evaluate the contribution of these neglected configurations by performing calculations with the data type '__float128' having 35 digits of precision. For the CPU calculations, a configuration is rejected if the calculated integrand is less than 10^{-30} or greater than 10^{-12}. The final integral is the sum of both the GPU and CPU results.

V. Experiment and Discussion

The experiments to evaluate the performance of the virial coefficients computation are conducted on NVIDA GPU Tesla K40, GTX580, Tesla M2050 and Intel CPU Xeon E5645 (6 cores) with 2.40GHz clock rate at the University at Buffalo's Center for Computational Research.

The major device parameters of Tesla K40, GTX580 and Tesla M2050 are shown in Table V, VI and VII. Tesla K40 is the latest Kepler architecture GPU with the compute capability 3.5. Both GTX580 and Tesla M2050 belong to the Fermi architecture GPU with the compute capability 2.0.

We verify the accuracy of the mixed-precision model in the first experiment. The result of partition by integrand value is shown in Tables III and IV. The results for B_8 and B_9 at $T = 5.0$ are tested. The truncation values 10^{-5} and 10^{-12} are examined in the experiment. In Tables III and IV, the 'Sequential' rows show the result and time when the coefficient is completely calculated on CPU. At the 'Truncation' rows, the final result is calculated by summing the GPU and CPU results. So is the time. We find that the results with different truncation values are consistent with

Table III
Accuracy Verification for B_8 at $T = 5.0$

	Result	Time(days)
Sequential	-0.087 (7)	152.6
Truncation at 10^{-5}	GPU -0.625(10)	5.8
	CPU 0.55(8)	4.7
	GPU+CPU -0.08(8)	10.5
Truncation at 10^{-12}	GPU -0.097(7)	9.3
	CPU 0.00030(15)	1.6
	GPU+CPU -0.097(7)	10.9

Table IV
Accuracy Verification for B_9 at T = 5.0

	Result	Time(days)
Sequential	-0.21(3)	551.6
Truncation at 10^{-5}	GPU 1.58(9)	7.0
	CPU -2.1(3)	13.1
	GPU+CPU -0.6(3)	20.1
Truncation at 10^{-12}	GPU -0.16(5)	9.6
	CPU 0.005(4)	1.5
	GPU+CPU -0.15(5)	11.1

the sequential results. For truncation at 10^{-12}, the value of CPU correction is much less than the uncertainty of the GPU result. Therefore, the CPU correction is negligible in the actual computation. If we run GPU part much longer, the CPU correction would become important. In addition, the result from truncation at 10^{-5} proves the approach is working. The GPU result itself does not agree with the sequential result, but with adding the CPU error correction, the result is consistent.

On GPUs, the number of registers per thread and the size of shared memory per block used are two key resources which determine the active threads per multiprocessor [25]. Once these two parameters are decided, with optional number of threads per block, the number of active threads per multiprocessor could be further decided, and the total number of threads will be the product of threads per multiprocessor and number of multiprocessors. In current CUDA implementation, GPU occupancy is a major factor to affect the performance since all the threads are independently executed. To get the best occupancy, we investigate the relationship between the number of total threads and the speed of the program. Figure 6, 7 and 8 plot the speed vs. number of threads for B_8, B_9, and B_{10} separately at $T = 1$ for the target system simulation. The speed is measured by both number of MC steps per second and GFLOPS. The GFLOPS is obtained by counting double precision instructions within integrand calculation by Wheatley's algorithm since that is the most expensive part for the whole program. The maximum speed measured in GFLOPS for Figure 6, 7 and 8 is less than 14 which is much below than the

Table V
TESLA K40 DEVICE PARAMETERS

#. of Streaming Multiprocesssors	15
#. of Streaming processsors	192
#. of cores	$15 \times 192 = 2880$
GPU Clock Rate	0.88GHz
Max #. of Registers per thread	255
Max #. of Registers per Block	65536
Shared memory per block	48KB
Global Memory	11520MB
Constant Memory	64KB
Max #. of threads per block	1024

Table VI
GTX580 DEVICE PARAMETERS

#. of Streaming Multiprocesssors	16
#. of Streaming processsors	32
#. of cores	$16 \times 32 = 512$
GPU Clock Rate	1.54GHz
Max #. of Registers per thread	63
Max #. of Registers per Block	32768
Shared memory per block	48KB
Global Memory	1535MB
Constant Memory	64KB
Max #. of threads per block	1024

Table VII
TESLA M2050 DEVICE PARAMETERS

#. of Streaming Multiprocesssors	14
#. of Streaming processsors	32
#. of cores	$14 \times 32 = 448$
GPU Clock Rate	1.15GHz
Max #. of Registers per thread	63
Max #. of Registers per Block	32768
Shared memory per block	48KB
Global Memory	2687MB
Constant Memory	48KB
Max #. of threads per block	1024

theoretical GFLOPS in [3] because a large proportion of instructions are used for bit manipulation to search partitions of each subset and memory access but not floating point operations in Wheatley's algorithm. In measuring the speed of GPU, the extra time for equilibration of MC process is included without counting corresponding steps, so the speed of GPU measured by the number of MC steps per second could be faster than the ones shown in all the plots.

Tesla K40 is used in this experiment. For B_8, speed shown in Figure 6, 69 registers are used by each thread, with no shared memory used. When the number of threads per block is set to 64 or 128, the number of threads per multiprocessor is limited to 896 with 44% of multiprocessor

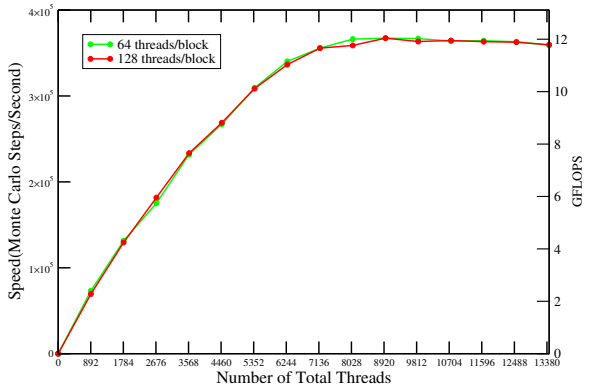

Figure 6. Occupancy of K40 for B_8

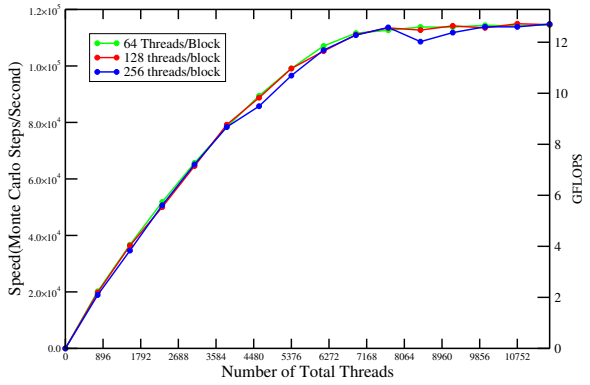

Figure 7. Occupancy of K40 for B_9

occupancy. From Figure 6 we can see that the speed does not always increase with increasing number of total threads. The maximum speed appears at the point where the total number of threads equals 8960. After that point no more speed is gained. The speed with the maximum number of threads is 98% of the maximum speed.

Figure 7 and 8 show the GPU occupancy when calculating B_9 and B_{10}. For B_9, the maximum speed is obtained when the total number of threads is 10752 and the threads per block are 128. When the number of threads goes up to the maximum, its speed is 99% of the maximum speed. A similar situation happens for B_{10}.

Figure 9 demonstrates the speedup of virial coefficients of all order for both reference system and target system. All CUDA speeds are tested under the maximum occupancy of the threads. We can see that the target system can get higher speedup than reference system for any order. Generally, the speedup falls down when the order goes high except that at B_4 the speedup becomes lowest. When the order is higher, on one hand the GPU occupancy will go down because the limited resources (such as fast registers) are used more; on the other hand more data such as the arrays using 'double' are accessed from slower memory. Therefore, the speed of GPU implementation naturally goes down. This

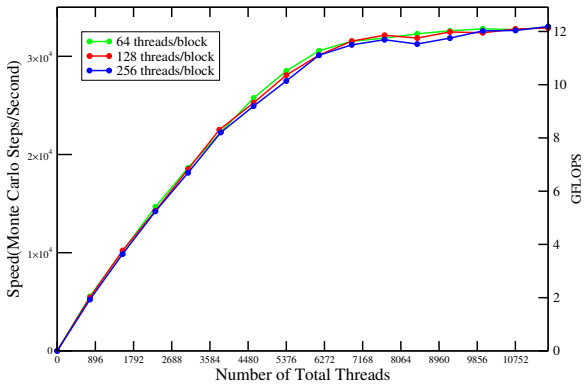

Figure 8. Occupancy of K40 for B_{10}

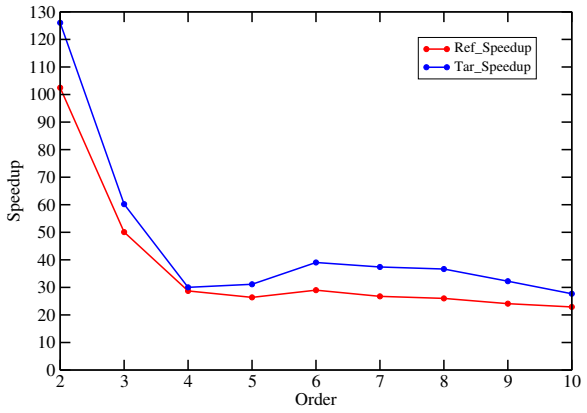

Figure 9. The speedup of virial coefficients of all orders.

mainly affects the trend of the speedup curve. In addition, the speed of single core of CPU measured in GFLOPS is calculated by counting the double precision instructions in Wheatley's algorithm. For example, when calculating $B4$, the GPU and single core of CPU achieve 4.83 GFLOPS and 0.15 GFLOPS, respectively, where the speed of single core of CPU is also far below its peak GFLOPS. As with the GPU, this slow performance reflects the fact that the floating point instructions occupy a small part of the overall computational effort.

Figure 10, 11 and 12 illustrate the speedup for B_8, B_9, and B_{10} among temperatures ranging from very low to very high values. The speedup of both reference system and target system are plotted. For B_8 and B_9, the speedup among Tesla K40, GTX580 and Tesla M2050 are compared. In Figure 11, we could see that the speedup of B_8 reference system on K40 could be more than twice as fast as M2050, whereas the speedup of target system is less than 2. GTX580 could obtain almost the same speedup as K40. B_9 speedup comparison on three devices shows a similar situation in Figure 11. For temperatures above 20, the reference system could gain much higher speedup. For B_{10}, the K40 could offer the program with the speedup average between 20 and

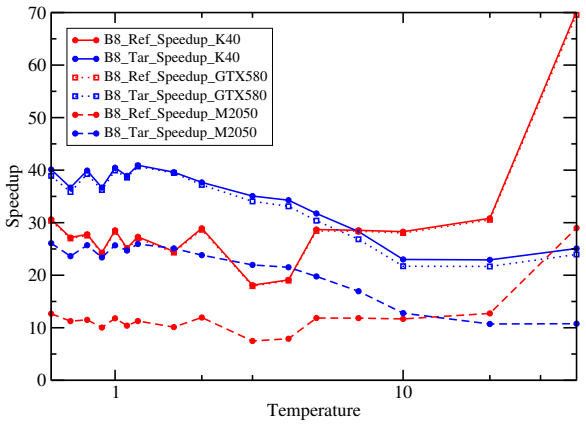

Figure 10. B_8 speedup as a function of temperature.

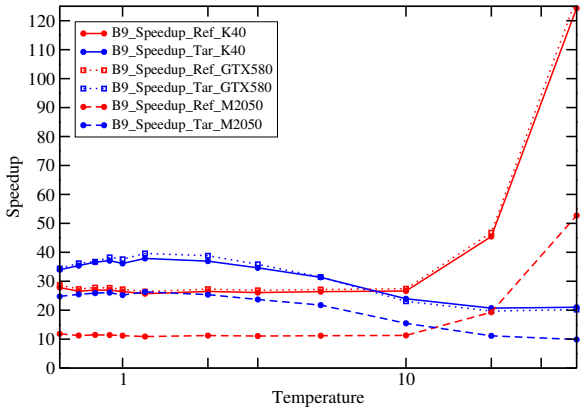

Figure 11. B_9 speedup as a function of temperature.

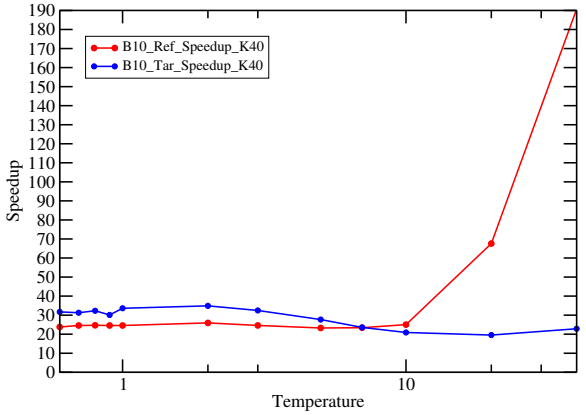

Figure 12. B_{10} speedup on Tesla K40 as a function of temperature.

40 shown in Figure 12, yet both GTX580 and Tesla M2050 would fail.

From Table V, VI and VII, we have clearly seen that the number of GPU cores in Tesla K40 is 6.4 times of the cores in Tesla M2050 and 5.6 times of the cores in GTX580. However, we do not receive 6 times more speedup on K40

compared with TeslaM2050. At some temperatures K40 is even slower than GTX580. This problem is mainly from the memory bottleneck and speed limitation of K40.

For the memory bottleneck, we can take the target system of B_8, B_9, and B_{10} as an example. For the NVIDIA GPU with the compute capability 2.0, there are maximum 63 registers with 32 bits which could be allocated to each CUDA thread by NVCC compiler. When calculating B_8 and B_9 on Tesla M2050 and GTX580, all registers are used up. For Tesla K40 with compute capability 3.5, there are maximum 255 registers with 32 bits which could be allocated to each thread. However, actually only 69 registers for B_8, 76 registers for B_9 and B_{10} are assigned by the compiler, far below the maximum number of registers which could be allocated.

Why does the NVCC compiler not allocate all registers? We first know that in each thread the Wheatley's algorithm has many operations and consumes a large amount of memory. There are three major arrays in Wheatley's algorithm used to store f_Q or f_A, f_C and f_B, each of which needs 2^N memory unit. Declaration of 'double' is also required for enough precision. So each array takes up $2^8 \times 8\text{B} = 2\text{MB}$ for B_8 per thread, 4MB for B_9, and 8MB for B_{10}. Apparently these data declared with automatic variables will be pushed to slower local memory but not the registers because they are 32 bits with limited amount. Beside, it is also not possible to store these data into shared memory with only 48KB per block. Therefore, the speed to access f_Q or f_A, f_B and f_C does not improve even though the code is ported to K40.

For the speed limitation, we first notice that the GPU clock rate of K40 is slower than GTX580 and M2050. Then take the target system of B_8 at $T = 1.0$ as an example. Figure 13 shows the speed comparison among K40, GTX580 and M2050 with the increasing number of total threads. The speed is also measured by two measurements-number of MC steps and GFLOPS. The number of threads per block is set to 64 for K40, GTX580 and M2050. In Figure 13, the single thread speed of K40 is 127 steps/sec while the single thread speed of GTX580 and M2050 are 160 steps/sec and 117 steps/sec. When the number of total threads increases, the speed of K40 initially increases linearly. When the number of total threads goes up to 8960, the speed almost has no increase before reaching the maximum number of threads (13440). The highest speed of K40 is 376100 steps/sec when the number of threads is equal to 10752, about 97% of GTX580. For GTX580, the faster single-core speed and approximate linear growth mainly accounts for its better speedup than K40, although there are fewer cores. For M2050, the speed could also keep an approximate linear growth rate. The speed is 233786 steps/sec when the number of threads reaches 7168-the maximum number of threads. So K40 is only 1.61 times faster than M2050. Therefore, being unable to keep the linear growth rate with increasing number of threads leads to the result that K40 cannot be 6 times faster than M2050.

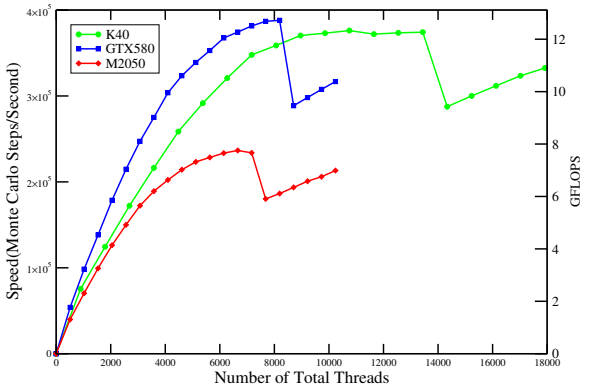

Figure 13. K40, GTX580 and M2050 Speed comparison with increasing number of threads.

VI. Conclusion and Future Work

Sequentially, the computation of high order virial coefficients of LJ potential model could cost years. In this work, we target accelerating the computation by developing the efficient algorithms on GPU. For the precision of our calculations of B_8, B_9, and B_{10} described here, we could obtain accurate results using the precision available on the GPU, while obtaining a speedup of 20 to 40 on the latest Tesla K40 GPU. However, calculations made to higher precision, or perhaps for other model potentials or cluster integrals, it may be necessary to employ the mixed-precision GPU/CPU approach outlined here. The experiment shows that the limited shared memory size, and nonlinear speed growth rate with increasing number of threads, are the bottlenecks of K40 that prevent the application from achieving further speedup.

In the future, we will continue focusing on how to further improve the performance of computing virial coefficients, trying more optimization strategies on GPU. In addition, we will spend more time on generalizing the mixed-precision model and applying it to other related areas having precision limitations.

Acknowledgment

Funding for the research was provided by the grant CHE-1027963 from the U.S. National Science Foundation. Computational resource was provided by the University at Buffalo Center for Computational Research. The latest GPU Tesla K40 was provided by NVIDIA corporation.

References

[1] Gpgpu. [Online]. Available: http://en.wikipedia.org/wiki/GPGPU

[2] Nvidia cuda zone. [Online]. Available: https://developer.nvidia.com/category/zone/cuda-zone

[3] NVIDIA. (2014, Feb.) cuda c programming guide, v5.5. CUDA_C_Programming_Guide.pdf. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[4] M. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houson, S. Legrand, A. Beberg, D. Ensign, C. Bruns, and V. Pande, "Accelerating molecular dynamic simulation on graphics processing units," *Journal of Computational Chemistry*, vol. 30, pp. 864–872, Apr. 2010.

[5] C. Hou, J. Xu, P. Wang, W. Huang, and X. Wang, "Efficient gpu-accelerated molecular dynamics simulation of solid covalent crystals," *Computer Physics Communications*, vol. 184, pp. 1364–1371, May 2013.

[6] J. Stone, D. Hardy, I. Ufimtsev, and K. Schulten, "Gpu-accelerated molecular modeling coming of age," *Journal of Molecular Graphics and Modelling*, vol. 29, pp. 116–125, Sep. 2010.

[7] T. Preis, P. Virnau, W. Paul, and J. Schneider, "Gpu accelerated monte carlo simulation of the 2d and 3d ising model," *Journal of Computational Physics*, vol. 228, pp. 4468–4477, Jul. 2009.

[8] T. Levy, G. Cohen, and E. Rabani, "Simulating lattice spin models on graphics processing units," *Journal of Chemical Theory and Computation*, vol. 6, pp. 3293–3301, Oct. 2010.

[9] J. Quinn and H. Abarbanel, "Data assimilation using a gpu accelerated path integral monte carlo approach," *Journal of Computational Physics*, vol. 230, pp. 8168–8178, Sep. 2011.

[10] J. Anderson, E. Jankowski, T. Grubb, M. Engel, and S. Glotzer, "Massively parallel monte carlo for many-particle simulations on gpus," *Journal of Computational Physics*, vol. 254, pp. 27–38, Dec. 2013.

[11] E. Mason and T. Spurling, *The Virial Equation of State*. Oxford: Pergamon Press, 1969.

[12] J. Singh and D. Kofke, "Mayer sampling: Calculation of cluster integrals using free-energy perturbation methods," *Phys. Rev. Lett.*, vol. 92, no. 22, p. 22061, 2004.

[13] A. Schultz and D. Kofke, "Sixth, seventh and eighth virial coefficients of the lennard-jones model," *Journal of Molecular Physics*, vol. 107, pp. 2309–2318, Nov. 2009.

[14] A. Schultz, N. Barlow, V. Chaudhary, and D. Kofke, "Mayer sampling monte carlo calculation of virial coefficients on graphics processors," *Journal of Molecular Physics*, vol. 111, pp. 535–543, 2013.

[15] R. Wheatley, "Calculation of high-order virial coefficients with applications to hard and soft spheres," *Phys. Rev. Lett.*, vol. 110, p. 200601, 2013.

[16] D. Göddeke, R. Strzodka, and S. Turek, "Accelerating double precision FEM simulations with GPUs," in *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, Sep 2005.

[17] R. Strzodka and D. Göddeke, "Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006)*, Apr. 2006, pp. 259–268.

[18] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy," in *SC 2006 Conference, Proceedings of the ACM/IEEE*, Tampa, FL, USA, Nov. 2006, p. 50.

[19] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Transactions on Mathematical Software*, vol. 34, no. 17, Jul. 2008.

[20] M. Baboulin, A. Buttari, J. Dongarra, J., J. Kurzak, Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, pp. 2526–2533, Dec. 2009.

[21] A. Thall, "Extended-precision floating-point numbers for gpu computation," in *ACM SIGGRAPH 2006 Research Posters*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006.

[22] M. Lu, B. He, and Q. Luo, "Supporting extended precision on graphics processors," in *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, ser. DaMoN '10. New York, NY, USA: ACM, 2010, pp. 19–26.

[23] T. Hill, *Statistical Mechanics*. New York: McGraw-Hill, 1956.

[24] N. Metropolis, A. Rosenbluth, M. Rosenbluth, and A. Teller., "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, pp. 1087–1092, Jun. 1953.

[25] NVIDIA. cuda gpu occupancy calculator. CUDA_Occupancy_calculator.xls. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls