Mayer Sampling Monte Carlo Calculation of Virial Coefficients on Graphics Processors

Andrew J. Schultz,[†] Vipin Chaudhary,[‡] David A. Kofke,[*,†] and Nathaniel S. Barlow[†]

Department of Chemical and Biological Engineering, and Department of Computer Science and Engineering, University at Buffalo, The State University of New York, Buffalo, NY 14260

E-mail: kofke@buffalo.edu

Abstract

Virial coefficients for various molecular models are calculated up to B_8 using the Mayer Sampling Monte Carlo method and implemented on a graphics processing unit (GPU). The execution time and performance of these calculations is compared with equivalent computations done on a CPU. The speedup between virial coefficient computations on a CPU (w/ optimized C code) and a GPU (w/ CUDA) is between 1 to 3 orders of magnitude. We report values of B_6, B_7, and B_8 of the Lennard-Jones (LJ) model, as computed on the GPU, for temperatures T=0.6 to 40 (in LJ units).

1. Introduction

Graphics processing units (GPUs) were originally developed to facilitate rendering of images to meet the demands of gaming software and other graphics-intensive applications. Computational

[*]To whom correspondence should be addressed
[†]Department of Chemical and Biological Engineering
[‡]Department of Computer Science and Engineering

scientists noticed that the highly parallel structure of these devices makes them very effective at a broader range of modeling applications, and in recent years their use in this manner has exploded. Many modeling groups are now rewriting codes almost from scratch to exploit the capabilities offered by GPUs, which often are found to speed up calculations by factors of 10 to 100, even without special efforts to revise the algorithms to exploit the architecture of the GPU. The devices are particularly effective at loosely-coupled or uncoupled computations having relatively low memory requirements. In this work, we examine application of GPUs to Mayer Sampling Monte Carlo (MSMC), an emerging computational methodology that seems tailor-made for such devices.

Mayer Sampling Monte Carlo (MSMC)[1] describes a set of methods developed for the calculation of cluster integrals that appear in statistical mechanical theories. Probably the most well-known cluster integrals are those relating to the virial equation of state[2] (VEOS), written

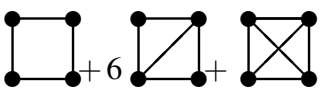
$$Z = \frac{p}{\rho kT} = 1 + B_2\rho + B_3\rho^2 + B_4\rho^3 + B_5\rho^4 + \ldots \tag{1}$$

where Z is the compressibility factor, p is the pressure, ρ is the number density, k is the Boltzmann constant and T is the absolute temperature. The n^{th} virial coefficient (B_n) can be expressed in terms of integrals over the positions of n molecules,[3] conveniently expressed as diagrams.[4] In particular, the first three coefficients are written

$$B_2 = -\frac{1}{2V}\int\int f_{12}dr_1dr_2 = -\frac{1}{2V}$$

$$B_3 = -\frac{1}{3V}\int\int f_{12}f_{13}f_{23}dr_1dr_2dr_3 = -\frac{1}{3V}$$

$$B_4 = -\frac{1}{8V}\left[3 \quad + 6 \quad + \quad\right]. \tag{2}$$



In each diagram, the black bonds represent the Mayer$-f$ function, $f_{ij} = \left[\exp\left(-u_{ij}/kT\right) - 1\right]$

where u_{ij} is the intermolecular potential between molecule i and molecule j with separation distance r_{ij}. For instance, the LJ model is defined by

$$u(r_{ij}) = 4\varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right], \tag{3}$$

where σ and ε are model parameters that define the size of the atom and the depth of the attractive energy well, respectively. The black points in Eq. (2) represent an integral of the molecule's position over all space coordinates r,[3,5] and simple extensions can be written for molecules with orientational or internal degrees of freedom. Cluster series appear in many applications other than the bulk-phase equation of state. Examples include inhomogeneous fluids,[6] surfaces,[7–9] random media,[10] associating fluids,[11–14] charged particles,[15] kinetic phenomena,[16] and more. Usually, the series are employed as an algebraic construct that supports development of integral equations and other mathematical formulations that can be solved for statistical mechanical quantities of interest. Much less frequently are attempts made to actually evaluate numerically the integrals represented by the cluster diagrams. This is due in no small part to the difficulty of evaluating such highly multidimensional integrals, a problem that is compounded by the proliferation of diagrams that appear when one moves to increasing order in the expansion variable (typically, the density). One of the consequences of this lack of information is that we have little understanding of the convergence behavior of the series that are formed by the diagrams, and indeed the extent to which the series are (by themselves) a viable means for quantifying the behavior of interest.

MSMC helps to remedy this problem by borrowing methods developed for free energy calculations.[17–19] The free energy is directly related to the partition function,[3] which for off-lattice systems is given in terms of a multi-dimensional configurational integral over all the arrangements of the atoms constituting the system. The integrand of the partition function is typically the Boltzmann factor of the energy, which is non-negative and thus can serve as the weighting distribution for an importance-sampling Monte Carlo scheme. In contrast, the configurational integrals appearing in the diagrammatic series have integrands that can have positive or negative values for different

configurations, and so they cannot serve directly as sampling weights. Consequently, some adjustments to the basic free-energy methods are required, but the process is still quite straightforward.

The basic formulation of MSMC is direct sampling, for which the working equation is

$$\Gamma(T) = \Gamma_0 \frac{\langle \gamma/\pi \rangle_\pi}{\langle \gamma_0/\pi \rangle_\pi}. \tag{4}$$

Here, Γ is the "target" cluster integral, or sum of cluster integrals, that we want to calculate; γ is the corresponding integrand or sum of integrands in Γ; π is the distribution function for weighting the configurations; and the angle brackets specify an ensemble average weighted by the subscripted distribution functions. The subscript 0 in Eq. (4) indicates a value for the reference system, for which the evaluated integral Γ_0 is known. Usually, the reference is selected as a system of hard spheres, although other choices are sometimes useful. Note, that the familiar free-energy perturbation formulation is recovered if $\pi = \gamma_0$, although this choice is not considered, since convenient choices of reference systems have γ_0 taking on both positive and negative values. Instead, π=$|\gamma|$ is used for the importance-sampling formulation.

Direct sampling methods are prone to severe bias,[20–26] which arises when the configurations that contribute most to the reference integral Γ_0 are not among those that are sampled by the distribution π. A reliable remedy is found in overlap sampling,[27–29] particularly in the optimum form given by Bennett.[30] The working equation is

$$\Gamma(T) = \Gamma_0 \frac{\langle \gamma/\pi \rangle_\pi / \langle \gamma_{OS}/\pi \rangle_\pi}{\langle \gamma_0/\pi_0 \rangle_{\pi_0} / \langle \gamma_{OS}/\pi_0 \rangle_{\pi_0}} \tag{5}$$

where $\pi = |\gamma|$, $\pi_0 = |\gamma_0|$, and γ_{OS} is defined in terms of γ and γ_0 such that bias is shifted towards configurations in the overlap region (i.e. those contributing significantly to both the target and reference system). The optimized form specified by Bennett[30] is

$$\gamma_{OS} = \frac{|\gamma_0||\gamma|}{\alpha|\gamma_0| + |\gamma|} \tag{6}$$

where α is a parameter optimized for the convergence of the calculation. Additionally, the distribution of computational effort expended between sampling the target and reference systems is balanced to ensure that their marginal contribution to the stochastic error of the result is equalized. By using overlap sampling, we need be less concerned that the configurations of the reference system form a wholly contained subset of the target system that governs sampling.[17–19,25–27] The best choice for α presumes knowledge of the quantity being calculated, and so some iteration is needed to obtain a self-consistent result. Alternatively, the averages in Eq. (5) can be recorded for a range of α (its choice does not affect sampling), and the optimum self-consistent α is then determined after a short initialization period. For molecular models with intramolecular degrees of freedom, the calculation of terms arising from molecular flexibility requires a variation on the MSMC method, but the overall approach is the same; further details are given in Shaul et. al.[31]

MSMC has several features that make it particularly appropriate for implementation on a GPU. Most importantly, it is trivial to parallelize the calculations. With so few molecules in the simulation, there is no need to apply a domain decomposition or consider other parallelization strategies. Instead, independent calculations of the same cluster sum can be launched on multiple processors, and the results combined at the end. This yields a result no different than if they had all been run in the same sequence. Unlike standard Monte Carlo or molecular dynamics simulations, there is no significant equilibration period that needs to occur before averages are collected. Also, since the parallel calculations can proceed without any communication required between processors, perfect scaling is trivial to achieve when parallelizing MSMC calculations. Moreover, the memory requirements for each process are quite modest. Multicore processors, and GPUs in particular, present an ideal platform to conduct these calculations.

In the next section, we describe how the MSMC calculation is implemented, with special attention to how the algorithm is ported to the GPU. In Section 3, we present performance results for the calculations and provide new values for Lennard-Jones virial coefficients that are significantly more precise and extensive than any reported previously. We finish with concluding remarks in Section 4.

2. Methods and Simulation Details

Computing with GPUs presents unique challenges and limitations that must be addressed in order to achieve high performance. In this section we describe the NVIDIA 295 GTX-based GPU that is used in our calculations. We also explain the unique features of the GPU that make programming them a challenge. The NVIDIA 295 GTX GPUs used here have 1 GB RAM and are composed of 16 stream multiprocessors, each of which is itself composed of 8 stream processors for a total of 128 stream processors. Each multiprocessor has 8192 32-bit registers, which in practice limits the number of threads (and therefore, performance) of the GPU kernel. The GPU is programmed using NVIDIA's CUDA programming model.[32] Each multiprocessor can manage 768 active threads. Threads are partitioned into thread blocks of up to 512 threads each, and thread blocks are further partitioned into warps of 32 threads. Each warp is executed by a single multiprocessor. Warps are not user-controlled or assignable, but rather are automatically partitioned from user-defined blocks. At any given clock cycle, an individual multiprocessor (and its stream processors) executes the same instruction on all threads of a warp. Consequently, each multiprocessor should most accurately be thought of as a SIMD processor.

Programming the GPU is not a matter of simply mapping a single thread to a single stream processor. Rather, with 8192 registers per multiprocessor, hundreds of threads per multiprocessor and thousands of threads per board should be used in order to fully utilize the GPU. Memory access patterns, in particular, must be carefully studied in order to minimize the number of global memory reads. Where possible, an application should make use of the 16 KB of shared memory per multiprocessor, as well as the texture and 64 KB of constant memory, in order to minimize GPU kernel access to global memory. When global memory must be accessed, it is essential that memory be both properly aligned, and laid out such that each SIMD thread accesses consecutive array elements in order to combine memory reads into larger 384-bit reads. In the following, we discuss these optimizations in more detail.

The MSMC simulation is written to perform a given number of Monte Carlo steps. In each step n, a new configuration is chosen by randomly moving the molecules from the configuration in the

previous step $(n-1)$. Using the Metropolis criteria, the trial configuration is accepted if $\pi_n \geq \pi_0$ or (if $\pi_n < \pi_0$), is accepted with probability π_n/π_0. After each step, γ and γ_0 are calculated and the quantities described in section 1 are computed.

We consider Mayer Sampling Monte Carlo in CUDA for three models: Lennard-Jones (LJ) sphere, TraPPE-UA ethane, and TraPPE-UA propane. TraPPE-UA ethane includes two LJ spheres (each representing a CH3 group) with a fixed bond length. TraPPE-UA propane includes three LJ spheres (one CH2 group between two CH3 groups) with fixed bond lengths and a harmonic potential for the bond angle.[33] For each model, we use an MC move that translates each molecule (except for the first one) by some random displacement. For ethane and propane, we also include an MC move that rotates all molecules by some angle about a random axis. For propane, we include a third MC move that takes the bond vector from the central atom to the first atom on each molecule and rotates it within the plane of that molecule.

Because the MSMC simulation considers only a few molecules interacting at a time, we are able to run many simulations simultaneously on a GPU, each running independently as a separate thread. However, several changes are necessary to the basic structure of the MSMC calculation as implemented for a CPU. For example, we run the reference and target simulations in a single process on the CPU, allowing focus to be shifted towards one or the other, based on the standard errors from each simulation. When running on the GPU, we use separate CUDA kernels for the target and reference system and manually optimize the fraction of time spent running the reference and target systems. Our MSMC code for CPUs is also able to tune the MC step size (to yield 50 % acceptance) during the initialization period, while the GPU kernel takes the step size as a parameter. We have an additional special-purpose kernel that reports on the acceptance of each type of move, and we use this to manually tune the step size. Our CPU version of MSMC can also determine an appropriate value for α during initialization. For the GPU, we split this functionality into separate kernels (one for the reference and one for the target) that are only used to determine α. Finally, our CPU version of MSMC takes block averages of the data being collected and estimates uncertainty in the averages based on the standard deviation of the block averages. For

the GPU version, we compute average values in each simulation and estimate uncertainty based on the standard deviation of the averages calculated from each separate simulation.

In order to avoid the use of local memory while running on the GPU, array indices are used only when the compiler can determine the index at compile-time. The primary effect of this is that a loop variable cannot be used as an array index. It is then necessary to replace looping structures with preprocessor directives to perform tasks such as computing a value for each Mayer bond or even initializing each molecule's position. Also, we cannot use random numbers as array indices to (for instance) move a random molecule or move a random atom on a molecule. For generating random numbers, we use a linear congruential generator with each thread generating its own random numbers. Although a Mersenne twister generator works faster on a CPU, the additional memory requirements make it unsuitable for the GPU.

One additional change to the CUDA code is necessary to achieve optimal performance. When performing a rotation or bond bending move, we need to first generate a random unit vector. This requires that we first generate trial x and y coordinates whose squared values do not sum to a value greater than one;[34] this is typically implemented via a while-loop. However, using a while loop within our CUDA code has a substantial negative effect on performance (overall performance is slower by about 33%). The reason for this is unclear, but we avoid the problem by explicitly trying twice and returning from the rotation routine if suitable x and y values are not found (which should only happen about 4.6% of the time).

Initially, our implementation of MSMC on the GPU was limited to single precision floating point calculations because of the large performance penalty for using double precision variables. However, we found that the accuracy of the final result suffered because the sum used to calculate $\langle \gamma_{OS}/\pi \rangle$ became too large and successive additions were severely truncated. Consequently, we now use a double precision number for this sum. We avoid a similar problem for $\langle \gamma/\pi \rangle$ because γ/π for each configuration must be +1 or -1 (since π=$|\gamma|$). Consequently, we use an integer for this sum and limit ourselves to 2 billion steps (to avoid integer overflow).

Another consequence of using single precision variables is that the bond lengths drift due to

roundoff. This problem also affects double precision calculations, but the drift is generally negligible for the time scale on which simulations are typically run. Using single precision, we maintain fixed bond lengths for ethane by performing rotations in a way that restores the bond length and explicitly relaxes the propane bond lengths back to their nominal values periodically during the simulation (this is done every 256 steps).

3. Results and discussion

In this section, the performance benefit for implementing MSMC on a GPU is demonstrated. Also, new values of the virial coefficients B_6, B_7, and B_8 are reported for the LJ system (defined by Eq. (3)), given respectively in Table 1, Table 2, and Table 3 for temperatures T=0.6 to 40 (in LJ units). The new values (computed on the GPU) are given in column 2 of each table. For comparison, the virial coefficients reported in Schultz and Kofke[35] are shown in column 3 of each table, for available temperatures. This serves as a consistency check between virial coefficient calculations using a CUDA implementation on the GPU (described in section 2), and the CPU implementation used in Schultz and Kofke.[35] When making this comparison, it should be noted that some coefficients have non-overlapping deviations. Note however, that these outliers are statistically permissible. When comparing all reported virial coefficients in Table 1, Table 2, and Table 3 for which the literature data exists, most differences remain close to the standard deviation of the mean, and no systematic deviation is evident.

The performance of implementing MSMC on a GPU is examined for three models: the LJ sphere, TraPPE-UA ethane, and TraPPE-UA propane. Although no coefficients are reported here for TraPPE-UA ethane and TraPPE-UA propane, the high computational expense of calculating coefficients for these models allows us to elucidate the GPU performance benefit, and to compare between systems of increasing complexity. In Figure 1, the performance (measured in Monte-Carlo steps per second) is plotted versus virial coefficient for three separate implementations of MSMC: the CUDA implementation described in section 2, a comparable implementation (written in C)

optimized and run on a CPU, and Etomica[36] (written in Java) also run on a CPU. The CUDA implementation on the GPU out-performs the C and Java implementations on the CPU by at least two orders of magnitude. For each implementation and virial coefficient shown in Figure 1, the computational expense is greatest for theTraPPE-UA propane model ($\times$), which is expected. The LJ model ($\square$) is the least expensive of the three models, although it's CUDA implementation performs only slightly better than the TraPPE-UA ethane ($\circ$) CUDA implementation for B_8. As expected, the performance decreases with the order of virial coefficient for all models and implementations.

The benefit of using the GPU for MSMC calculations is shown in Figure 2, where the CUDA speedup versus virial coefficient is plotted for each model. The CUDA speedup is defined here as the ratio of GPU-to-CPU performance. For the LJ model , the CUDA speedup is nearly constant for lower order virial coefficients, with a performance that is $\sim 200\times$ that of the C implementation and $\sim 800\times$ that of the Java implementation on the CPU. For all CPU computations, we use an Intel "Westmere" Xeon 2.66GHz (X5650) processor.

The memory usage during the MSMC computations on the GPU is shown in Figure 3 for each model. As expected, the memory use increases with the order of virial coefficient. For most computations, memory is accessed solely through the registers of the GPU, each providing 4 bytes of memory. The number of registers are indicated on the rightmost vertical axis of Figure 3 and shown by solid curves in the plot. When calculating B_8 for the three models, all registers are utilized, leading to local memory access on the GPU; this is also true for B_7 (TraPPE-UA ethane, propane) and B_6 (TraPPE-UA propane), as indicated in Figure 3.

As mentioned in section 2, each multiprocessor on the GPU has 8192 registers available for use by CUDA program threads. The CUDA compiler attempts to minimize register usage in order to maximize the number of thread blocks that can be simultaneously active on the GPU. Upon compilation of each individual virial coefficient program, the CUDA compiler reports the minimum number of registers needed. Using this number of registers, the number of total threads and the size of each thread block is then calculated such that the occupancy is maximized; this is done for each virial coefficient program. On a GPU, the occupancy is defined as the ratio of active warps to

the maximum number of warps supported. The number of threads for maximum occupancy is plotted versus virial coefficient in Figure 4; these are the number of threads used in the computations. As the problem size increases, fewer threads are used by the program to maximize occupancy. The performance-per-thread is plotted versus virial coefficient in Figure 5. There is a noticeable lack of performance increase between the computation of B_6 and B_2, compared with the higher order virial coefficients for the LJ model ($\square$); B_7 and B_8 calculations are substantially slower. To examine this further, the computations for B_2 and B_6 were run again, for the same thread block size (used for maximum occupancy) but for fewer total threads. Using fewer threads, the calculation of B_2 runs up to $4\times$ faster, while the performance of the B_6 calculation remains the same. This may explain the slow decrease in performance between B_2 and B_6 (shown in Figure 5) when we fix the number of threads and thread block size for maximum occupancy.

4. Conclusions

Despite necessary modifications needed to allow a Mayer Sampling Monte Carlo calculation to be performed on a GPU, such as replacing looping structures, generating random unit vectors, and (in some cases) using single precision variables, the performance benefit is well worth the effort. The CUDA speedup between both Java and C implementations is such that we are now able to compute virial coefficients with significantly higher precision.

GPUs are not the only multi-core computing platform. While the core count in today's GPU is very high (in the hundreds), the general-purpose processors from Intel and AMD are also increasing their core counts substantially and Intel has suggested that they will have many-core general-purpose processors in the next two years. Although the architectures will be different from NVIDIA GPUs, the basic challenges of many-core processors will remain the same. Thus, the experience of specific optimizations on GPUs will be useful for next generation GPUs (with five hundred or a thousand cores) as well as many-core general-purpose processors.

Table 1: Sixth virial coefficients for the Lennard-Jones model (defined by Eq. (3)). The values calculated using the GPU (column 2) are compared with those found in Schultz and Kofke[35] (column 3). Numbers in parentheses are the 67% confidence limits in the rightmost digit(s) of the tabulated value.

kT/ε	$B_6\sigma^{-15}$	$B_6\sigma^{-15}$ (ref. 35)
0.62	$-2.0314(17)\times10^6$	
0.64	$-1.1766(10)\times10^6$	
0.66	$-6.958(6)\times10^5$	
0.68	$-4.205(4)\times10^5$	
0.7	$-2.589(2)\times10^5$	
0.72	$-1.6131(16)\times10^5$	
0.74	$-1.0241(11)\times10^5$	
0.76	$-6.571(7)\times10^4$	
0.78	$-4.261(5)\times10^4$	
0.8	$-2.798(4)\times10^4$	
0.82	$-1.853(3)\times10^4$	
0.84	$-1.233(2)\times10^4$	
0.86	$-8.331(15)\times10^3$	
0.88	$-5.611(11)\times10^3$	
0.9	$-3813(9)$	
0.92	$-2612(7)$	
0.94	$-1777(5)$	
0.96	$-1213(4)$	
0.98	$-836(3)$	
1	$-580(3)$	$-5.8(2)\times10^2$
1.02	$-395(2)$	
1.04	$-270.6(18)$	
1.06	$-187.0(14)$	
1.08	$-127.0(12)$	
1.1	$-88.5(7)$	$-97.(5)$
1.12	$-61.5(5)$	
1.14	$-41.8(4)$	
1.16	$-28.8(3)$	
1.18	$-20.7(2)$	
1.2	$-14.9(2)$	$-15.3(9)$
1.22	$-11.31(17)$	
1.24	$-9.11(14)$	
1.26	$-7.23(11)$	
1.28	$-6.40(10)$	
1.3	$-6.09(8)$	$-5.94(15)$
1.32	$-5.77(7)$	
1.34	$-5.58(7)$	
1.36	$-5.35(6)$	
1.4	$-5.25(5)$	$-5.43(12)$
1.44	$-4.90(5)$	
1.48	$-4.63(5)$	
1.52	$-4.26(4)$	
1.56	$-3.73(3)$	
1.6	$-3.28(3)$	$-3.11(10)$
1.64	$-2.74(3)$	
1.68	$-2.23(2)$	
1.72	$-1.766(19)$	
1.76	$-1.29(2)$	
1.8	$-0.88(2)$	
1.84	$-0.510(17)$	
1.88	$-0.142(15)$	
1.92	0.131(13)	
1.96	0.420(11)	
2	0.669(10)	0.63(7)
2.1	1.135(8)	
2.2	1.461(6)	1.41(5)
2.3	1.688(5)	
2.4	1.829(4)	
2.5	1.901(3)	1.90(3)
2.6	1.941(3)	
2.7	1.943(2)	1.91(3)
2.8	1.9260(18)	
2.9	1.8934(15)	
3	1.8547(13)	1.85(3)
3.1	1.8047(12)	
3.2	1.7535(10)	
3.3	1.6995(9)	
3.4	1.6475(8)	
3.5	1.5921(8)	1.604(18)
3.6	1.5402(7)	
3.7	1.4887(6)	
3.8	1.4384(6)	
3.9	1.3909(6)	
4	1.3458(5)	
4.2	1.2586(11)	
4.4	1.1804(10)	
4.8	1.0429(8)	
5.2	0.9292(6)	
5.6	0.8347(5)	
6	0.7550(4)	
7	0.6035(3)	0.603(4)
8	0.4972(2)	0.497(3)
9	0.41994(19)	0.420(2)
10	0.36073(16)	0.361(2)
12	0.27814(12)	0.2780(13)
14	0.22375(9)	
18	0.15705(6)	
24	0.10531(4)	
30	0.07738(3)	
40	0.05225(2)	

Table 2: Seventh virial coefficients for the Lennard-Jones model (defined by Eq. (3)). The values calculated using the GPU (column 2) are compared with those found in Schultz and Kofke[35] (column 3). Numbers in parentheses are the 67% confidence limits in the rightmost digit(s) of the tabulated value.

kT/ε	$B_7\sigma^{-18}$	$B_7\sigma^{-18}$ (ref. 35)
0.6	$-2.13(3)\times10^{8}$	
0.64	$-5.44(8)\times10^{7}$	
0.68	$-1.51(2)\times10^{7}$	
0.72	$-4.78(7)\times10^{6}$	
0.76	$-1.62(3)\times10^{6}$	
0.8	$-5.87(10)\times10^{5}$	
0.84	$-2.13(4)\times10^{5}$	
0.88	$-8.6(2)\times10^{4}$	
0.92	$-3.34(10)\times10^{4}$	
0.96	$-1.41(5)\times10^{4}$	
1	$-5.4(2)\times10^{3}$	$-5.7(4)\times10^{3}$
1.04	$-2.28(12)\times10^{3}$	
1.08	$-8.8(8)\times10^{2}$	
1.12	$-2.6(5)\times10^{2}$	
1.16	$-1.6(3)\times10^{2}$	
1.2	$-6(2)\times10$	$-67.(17)$
1.24	$-1.1(14)\times10$	
1.28	15(10)	
1.32	$-2(7)$	
1.36	3(5)	
1.44	$-3(3)$	
1.52	4.4(16)	
1.6	2.5(10)	3.3(8)
1.68	4.8(7)	
1.76	5.6(4)	
1.84	5.3(3)	
1.92	5.2(2)	
2	5.5(2)	4.97(19)
2.2	4.37(11)	
2.4	3.50(6)	
2.6	2.74(4)	
2.8	2.20(2)	
3	1.768(15)	
3.2	1.441(11)	
3.4	1.193(8)	
3.6	1.003(6)	
3.8	0.842(5)	
4	0.728(4)	
4.4	0.555(2)	
5.2	0.3507(13)	
6	0.2437(8)	
8	0.1201(3)	
10	0.06982(19)	0.0697(5)
14	0.03187(8)	
24	0.00962(3)	
40	0.003448(12)	

Table 3: Eighth virial coefficients for the Lennard-Jones model (defined by Eq. (3)). The values calculated using the GPU (column 2) are compared with those found in Schultz and Kofke[35] (column 3). Numbers in parentheses are the 67% confidence limits in the rightmost digit(s) of the tabulated value.

kT/ε	$B_8\sigma^{-21}$	$B_8\sigma^{-21}$ (ref. 35)
0.6	$-1.3(5)\times 10^{10}$	
0.8	$-1.48(19)\times 10^{7}$	
1	$-6.4(19)\times 10^{4}$	
1.3	$0(4)\times 10^{2}$	
2	5(2)	
5	−0.101(8)	−0.16(6)
40	−0.00229(6)	

Acknowledgement

This work is supported by the National Science Foundation under grants CBET-0854340, CHE-1027963, and CITraCS award 1048579.

References

(1) Singh, J. K.; Kofke, D. A. *Phys. Rev. Lett.* **2004**, *92*, 220601/1–220601/4.

(2) Mason, E. A.; Spurling, T. H. In *The International Encyclopedia of Physical Chemistry and Chemical Physics, Topic 10: The Fluid State*; Rowlinson, J. S., Ed.; Pergamon Press, New York, 1969; Vol. 2; Chapter The Virial Equation of State.

(3) McQuarrie, D. A. *Statistical Mechanics*; University Science Books, California, 1973.

(4) Mayer, J. E.; Mayer, M. G. *Statistical Mechanics*; Wiley, New York, 1940.

(5) Hansen, J. P.; McDonald, I. R. *Theory of Simple Liquids*, 3rd ed.; Academic Press, London, 2006.

(6) Rowlinson, J. S. *P. Roy. Soc. A-Math. Phy.* **1985**, *402*, 67–82.

(7) Bellemans, A. *Physica* **1962**, *28*, 493–510.

(8) Bellemans, A. *Physica* **1962**, *28*, 617–632.

(9) Bellemans, A. *Physica* **1963**, *29*, 548–554.

(10) Madden, W. G.; Glandt, E. D. *J. Stat. Phys.* **1988**, *51*, 537–558.

(11) Wertheim, M. *J. Stat. Phys.* **1984**, *35*, 19–34.

(12) Wertheim, M. *J. Stat. Phys.* **1984**, *35*, 35–47.

(13) Wertheim, M. *J. Stat. Phys.* **1986**, *42*, 459–491.

(14) Kim, H. M.; Schultz, A. J.; Kofke, D. A. *J. Phys. Chem. B* **2010**, *114*, 11515–11524.

(15) Stell, G.; Lebowitz, J. L. *J. Chem. Phys.* **1968**, *49*, 3706–3717.

(16) Anderson, H. C. *J. Phys. Chem. B* **2002**, *106*, 8326–8337.

(17) Kofke, D. A. *Mol. Phys.* **2004**, *102*, 405–420.

(18) Kofke, D. A. *Fluid Phase Equil.* **2005**, *228-229*, 41–48.

(19) Kofke, D. A.; Frenkel, D. In *Handbook of Molecular Modeling*; Yip, S., Ed.; Kluwer Academic Publishers, Dordrecht, 2004; Chapter Free energies and phase equilibria.

(20) Kofke, D. A.; Cummings, P. T. *Mol. Phys.* **1997**, *92*, 973–996.

(21) Lu, N. D.; Kofke, D. A. *AIChE Symp. Ser.* **2001**, *97*, 146–149.

(22) Lu, N. D.; Kofke, D. A. *J. Chem. Phys.* **2001**, *114*, 7303–7311.

(23) Lu, N. D.; Kofke, D. A. *J. Chem. Phys.* **2001**, *115*, 6866–6875.

(24) Wu, D.; Kofke, D. A. *Phys. Rev. E* **2004**, *70*, 066702/1–11.

(25) Wu, D.; Kofke, D. A. *J. Chem. Phys.* **2005**, *123*, 054103/1–10.

(26) Wu, D.; Kofke, D. A. *J. Chem. Phys.* **2005**, *128*, 084109/1–10.

(27) Lu, N. D.; Kofke, D. A.; Woolf, T. *J. Comp. Chem.* **2004**, *25*, 28–39.

(28) Lu, N. D.; Singh, J. K.; Kofke, D. A. *J. Chem. Phys.* **2003**, *118*, 2977–2984.

(29) Shirts, M. R.; Blair, E.; Hooker, G.; Pande, V. S. *Phys. Rev. Lett.* **2003**, *91*.

(30) Bennet, C. H. *J. Comput. Phys.* **1976**, *22*, 245–268.

(31) Shaul, K. R. S.; Schultz, A. J.; Kofke, D. A. *J. Chem. Phys.* **2011**, *135*, 12401:1–8.

(32) Kirk, S. B.; Hwu, W. W. *Programming Massively Parallel Processors: A Hands-on Approach*; Morgan Kaufman Publishers, 2010.

(33) Martina, M. G.; Siepmann, J. I. *J. Phys. Chem. B* **1998**, *102*, 2569–2577.

(34) Allen, M. P.; Tildesley, D. J. *Computer Simulation of Liquids*; Clarendon Press, Oxford, 1987.

(35) Schultz, A. J.; Kofke, D. A. *Mol. Phys.* **2009**, *107*, 2309–2318.

(36) Kofke, D.; Schultz, A. http://www.etomica.org/.

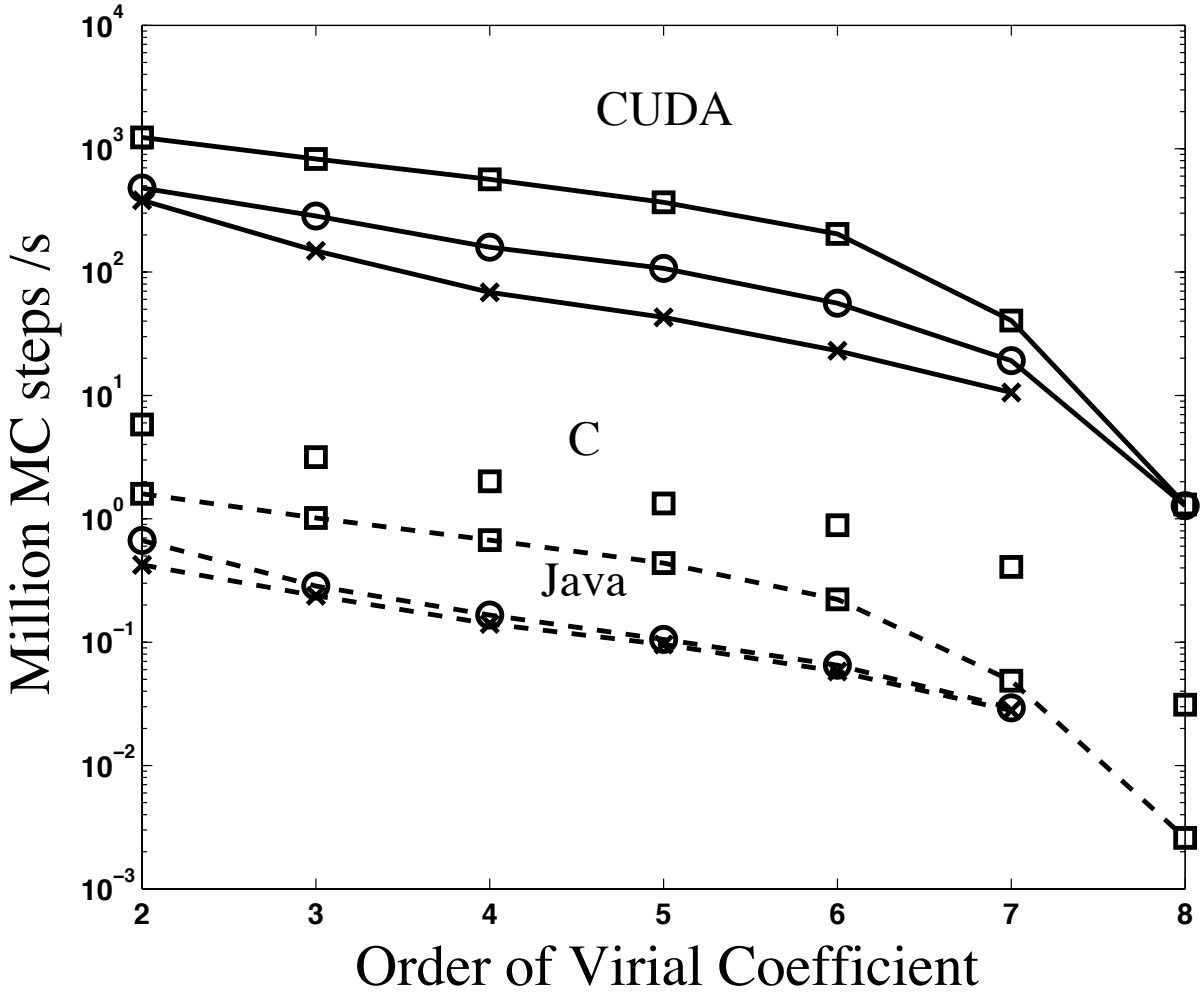


Figure 1: Performance of virial coefficient computations for the Lennard-Jones (□), TraPPE-UA ethane (○), and TraPPE-UA propane (×) models, measured in Monte Carlo steps per second. A comparison is made between the CUDA implementation on the GPU (solid curves), the C code implemented on a CPU (unconnected symbols), and Etomica[36] (written in Java) implemented on a CPU (dashed curves).

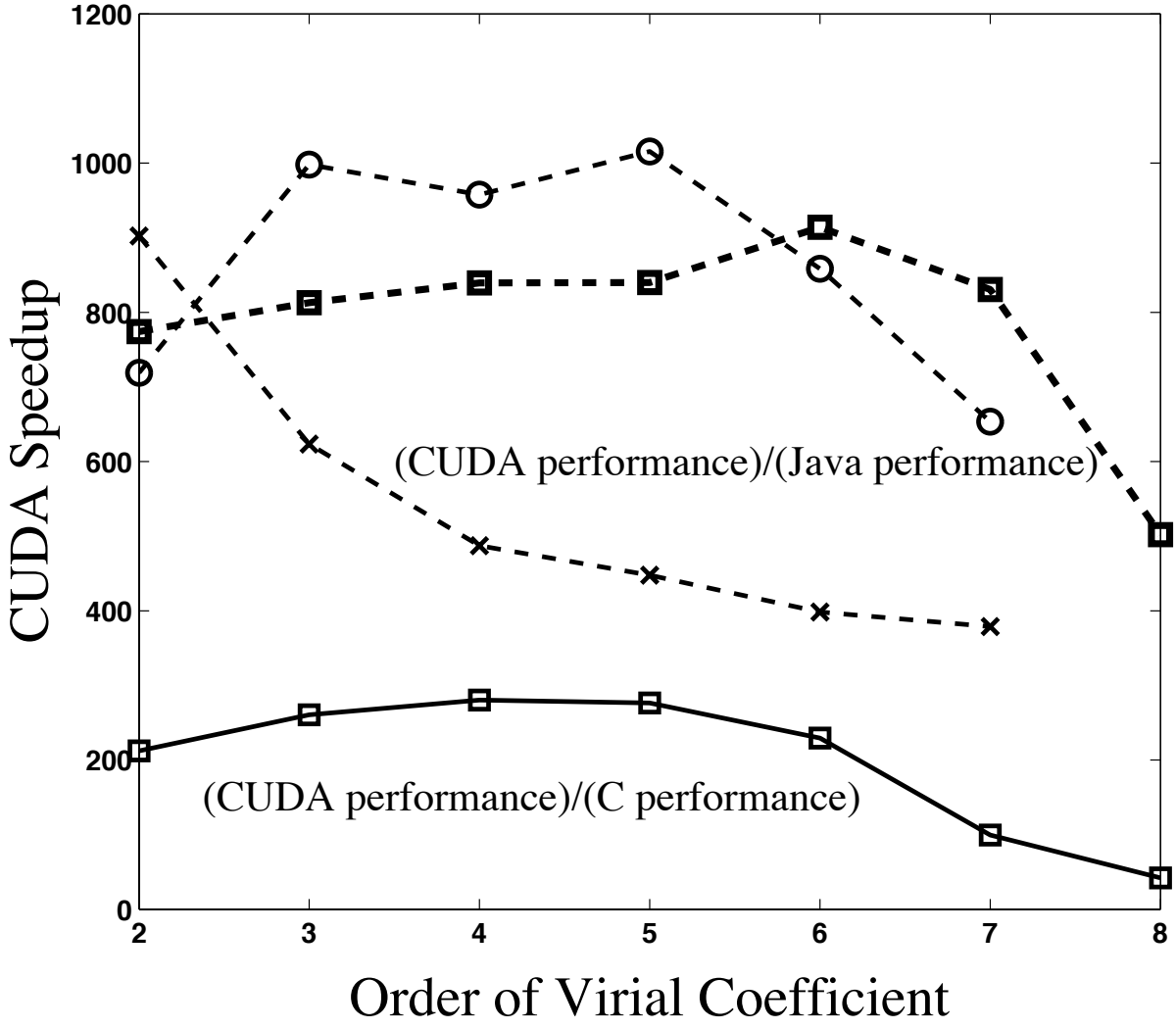


Figure 2: The ratio of GPU-to-CPU performance for the computation of virial coefficients for the Lennard-Jones (□), TraPPE-UA ethane (○), and TraPPE-UA propane (×) models. A comparison is made between the CUDA/(C code) speedup (solid curve) and CUDA/(Java: Etomica) speedup (dashed curves).

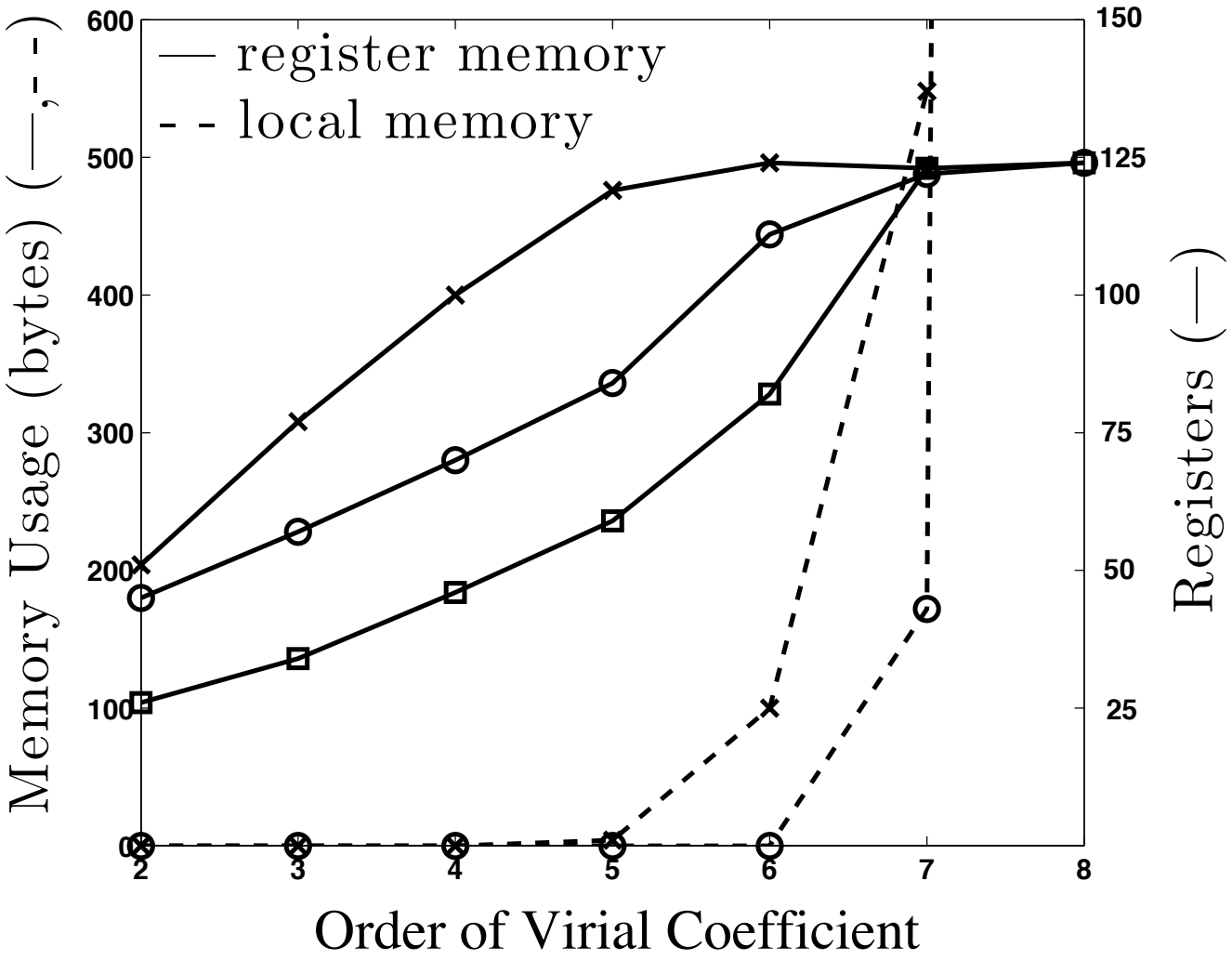


Figure 3: GPU register (—) and local (- -) memory usage during the computation of virial coefficients on the for the Lennard-Jones (□), TraPPE-UA ethane (○), and TraPPE-UA propane (×) models. Here, one register on the GPU provides 4 bytes of memory. The local memory used in computing B_8 for the Lennard-Jones and TraPPE-UA ethane models are not shown in the figure; they are respectively 14740 bytes (3685 registers) and 15016 bytes (3754 registers).

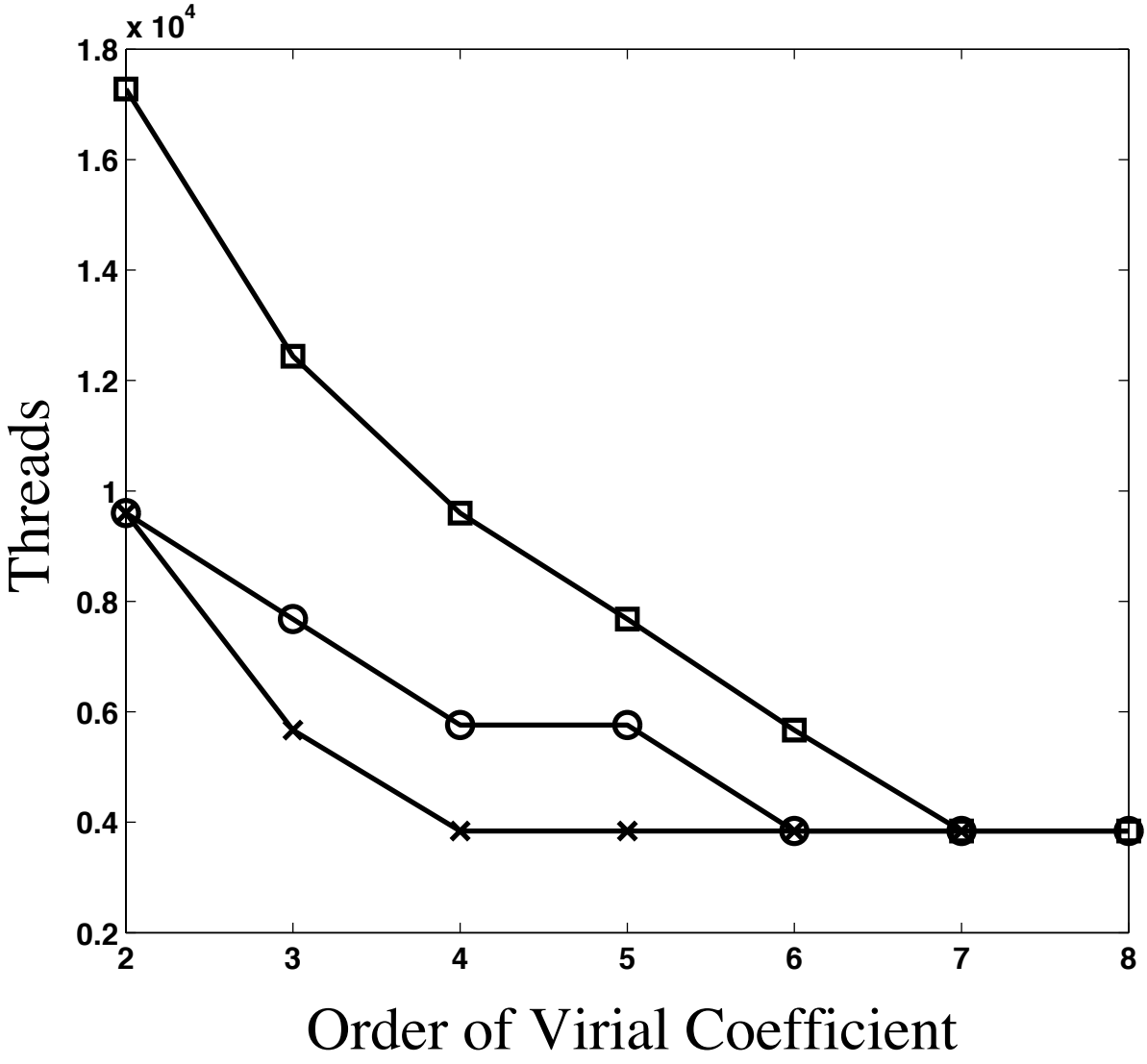


Figure 4: The number of threads (processors) used in the computation of virial coefficients on the GPU for the Lennard-Jones (□), TraPPE-UA ethane (○), and TraPPE-UA propane (×) models.

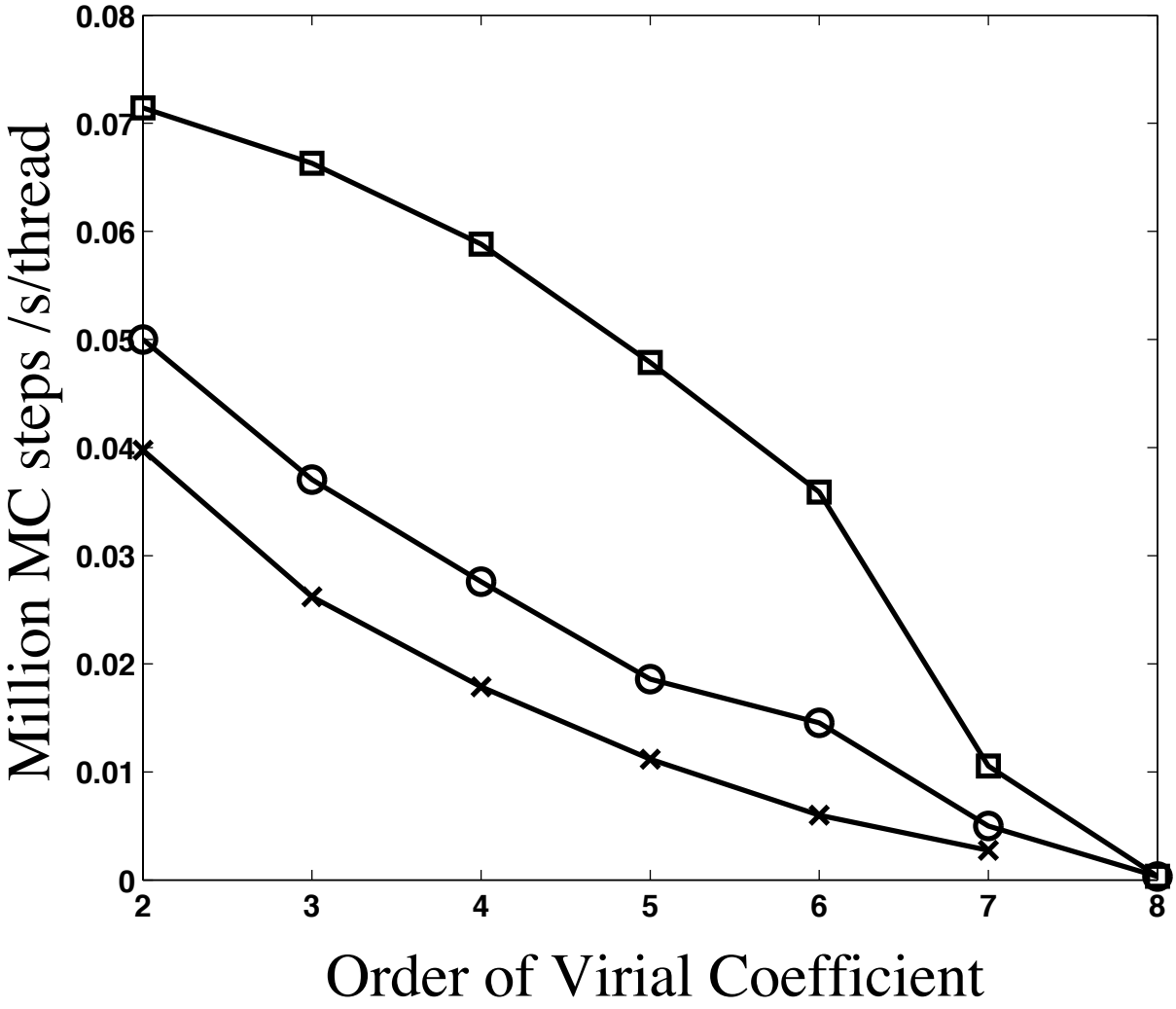


Figure 5: Performance (measured in Monte Carlo steps per second) per thread for the computation of virial coefficients on the GPU for the Lennard-Jones (□), TraPPE-UA ethane (○), and TraPPE-UA propane (×) models.