Analyzing Molecular Simulations Trajectories by Utilizing CUDA on GPU Architecture

Gourav Shrivastav†,* and Manish Agarwal†,‡,*

† Department of Chemistry, Indian Institute of Technology Delhi, Hauz Khas, New Delhi 110016.
‡ Computer Services Centre, Indian Institute of Technology Delhi, Hauz Khas, New Delhi 110016.
* Corresponding author: E-mail: gourav.shri7@gmail.com, zmanish@cc.iitd.ac.in Tel: +91-11-26597170

Abstract

With the advent of high-performance computing techniques, the data for analysis has grown significantly. Here, graphic processing unit (GPU) based program kernels are discussed to exploit parallelism in the analysis codes specific to molecular simulations trajectories and data, hence reducing the time consumption. Commonly computed properties of systems which utilize static and dynamic correlations are considered. In static properties, order metrics based on two-particle correlations are shown to exhibit 10-50× speedups relative to conventional serial CPU codes. Efficiency in finding three-particle correlations, which are relatively more time consuming calculations, are also shown to be 10-45× faster depending on system size. In the case of dynamic properties, the viscosity is computed using Green-Kubo formalism at a 10-25× faster rate depending on the correlation time and total trajectory length. Similarly, for mean square displacement calculations, the speedup of ≈ 40 − 80× is achieved for all system sizes and simulation times. Particularly for these embarrassingly parallel computationally intensive problems, the compute unified device architecture application programming interface affords maximum scale up with minimum implementation time. It is shown that all the GPU codes render 1-2 order of magnitude reduction in the analyses times. Hence these codes, in conjunction with GPU accelerated molecular simulations, can lead to an overall improved and efficient performance.

1 Introduction

Recent emergence of high-performance computing (HPC) techniques and relatively easy access to large clusters has resulted in development and use of computational methods as a recurrent tool in many interdisciplinary scientific research areas such as quantum mechanics [1–3], molecular mechanics [4–7], molecular modeling [8], medical imaging [9], and cosmology [10,11]. These HPC techniques are harnessing parallelism by the introduction of integrated many core/thread architecture of Nvidia GPUs and Intel Xeon Phi cards while keeping the economic and environmental footprint low. More specific to molecular dynamics (MD) studies, various modern simulation packages LAMMPS [12–14], GROMACS [15], DL_POLY [16], NAMD [17], CHARMM [18], Gaussian [19] etc. have employed these accelerating cards for compute-intensive calculations, such as generation of neighbour lists and force calculations. MD simulation packages like HOOMD [4], RUMD [20], and ACEMD [21] which perform entire calculations on GPUs are also available. Consequently, massive size simulations, such as complex biomolecular compounds and multi nano-particle systems under solvent conditions, are now feasible. Low to
mid range consumer GPUs have enough compute power and memory to accelerate simulations 2-3 times faster as opposed to that on only CPUs. Hence the data generated and required calculations for analyses also have grown tremendously. Although the growth in capabilities of simulation packages is remarkable and evident from literature, GPU-based analysis codes of the resultant data are not easily available, with few exceptions [4, 22].

Algorithms used for analyzing molecular dynamics data are traditionally run as serial codes mostly written in FORTRAN or C/C++. Recently, time taken by such analysis codes has reached similar or more than the time taken by the simulation itself. Even though in some cases, the analysis can be made during the simulation itself, it is often preferred to obtain the raw data so as to facilitate extraction of more meaningful and varied analysis. Thus, it is essential to port these analysis codes on parallel architectures to gain overall performance. There are many shared memory and distributed memory parallelization application programming interfaces (APIs) and standards available. In terms of increasing ease of coding some of these are Message Passing Interface (MPI), Common Unified Device Architecture (CUDA), Open Multi-Processing (OpenMP) and Open Accelerators (OpenACC). Herein, GPU-accelerated versions of common analyses for MD simulation trajectories are presented. Distributed memory APIs such as MPI involve a full overhaul of the analysis code as well as needs a thorough understanding of the machine architecture. Compiler directive driven standards such as OpenMP and OpenACC rely on the compiler’s capabilities for parallelization. CUDA, on the other hand, affords a good balance between programmer control and process parallelization while requiring device level control rather than process level control.

MD simulations give access to both static and dynamic properties of the system. The radial distribution function is one of the most common algorithms and has been utilized in many areas of scientific research, and has been parallelized and optimized [22]. Its calculation involves the compute-intensive, but extremely parallel, search for possible pair distances which depend quadratically on the system size. The present study exploits the parallelism in two static properties, tetrahedral order parameter ($q_{tet}$) [23] and crystalline order parameter ($P_2$) [24], for trajectory data obtained from molecular dynamics simulations of systems such as water and polyethylene, respectively. Both these order parameters utilize similar core structure of calculations as that of radial distribution functions (RDFs), and therefore, have a highly parallelizable algorithm. Another investigated static property is triplet correlation function (TCFs) [25], an important quantity in systems where two-body correlations are not sufficient to probe correlations. TCFs extend the calculation of pair distances to a next level and therefore can reveal information about the angle resolved local ordering. It first calculates all the possible pair distances, like RDFs, subsequently, distances of all the particles are calculated for each pair. Hence, analysis time varies cubically with system size and makes it a compute-expensive analysis scheme. Among dynamic properties, which are discussed here, viscosity and diffusivity are computed by utilizing the Green-Kubo and Einstein formalism. The Green-Kubo formalism evaluates the viscosity as the time integral of off-diagonal terms of pressure tensor. The calculation of diffusivity via Einstein method is based on the average mean square distance travelled by particles in a given time, and like time-correlation functions, it utilizes all the possible time origins. Both these algorithms exhibit embarrassing parallelism. Since the creation of histograms generates race conditions, such updates are carried out by utilizing the appropriate atomic operation. Since these codes offer inherent embarrassing parallelism, CUDA on GPUs is an appropriate choice. One of the intentions of current work is to achieve considerable scale up from minimal knowledge and learning time of CUDA and implementing the parallel code. Also, consumer GPUs offer considerable compute capabilities for the execution of these codes.

The paper is organized as follows. Section 2 contains computational details which includes simulation details, hardware details, and introduction to CUDA kernel. Section 3 provides
discussion on the parallel pseudocodes of utilized algorithms along with the results obtained. Section 4 discusses the conclusions.

2 Systems and Simulation Details

In order to describe the parallelization of the common analysis codes, we use the following simulations of representative systems.

2.1 Simulation Details

We have considered three varieties of systems for this study, namely n-octane, water, and polyethylene. Octane was modeled using the united atom version of TraPPE [27], as implemented in Refs [28][29]. The three sites single point charge (extended) (SPC/E) model of water was used [30], and all-atom version of OPLS force-field [31] was used for polyethylene. The corresponding trajectories were used for further analysis. For octane and water, simulations were performed using the LAMMPS [12][14] MD simulation package to obtain the trajectories. The equations of motion were integrated using the velocity-Verlet scheme with a time step of 1 fs. Orthogonal periodic boundary conditions were employed to ensure the bulk limit and a global cut-off of 14 Å was used for short range interactions. Temperature and pressure were maintained using Nosé-Hoover thermostat and barostat, respectively. Particle-particle-particle mesh was used for the k-space calculations. Rigid constraints associated with water molecules were maintained by the use of SHAKE algorithm. For both systems, initial random configuration of 6720 molecules was obtained by PACKMOL. [32] NPT simulations were performed for 2 ns at 300 K and 1 atm pressure. The resulting configuration was replicated to obtain the configurations for the system containing 13440 and 26880 molecules. Subsequently, systems were allowed to equilibrate in NVT ensemble for 1 ns followed by a production run of 1 ns. Trajectories were saved for every 100 steps. For polyethylene, data from our previous study for the system containing 40 chains of 50mers, i.e., 2000 carbon atoms with 2000 frames was used [33].

2.2 Hardware Details

For testing and deploying analysis codes, we have used two kinds of machines with K40m (2880 cores, 12 GB memory, 0.745 GHz clock rate) and GTX 780 (2304 cores, 3 GB memory, 0.993 GHz clock rate) GPUs. The K40 machine contains Intel(R) Xeon(R) CPU E5-2680 v3 with the clock rate of 2.50 GHz and 64 GB memory while the GTX machine contains Intel(R) Xeon(R) CPU E5-1650 v2 with clock rate of 3.50 GHz and 32 GB memory. All MD simulations were done on the K40 machine.

2.3 CUDA Kernels

CUDA is a programming paradigm introduced by Nvidia to leverage parallelism on multi-threaded GPU devices. It also manifests optimized libraries, both compile and running time, of certain algorithms for faster computations such as fast Fourier transform library, sparse matrix library, etc. A conventional CUDA program accommodates both serial and parallel structure. For this, CUDA may utilize both C/C++ and FORTRAN language. In general, serial part executes on CPU and the parallel part, also known as CUDA kernel or kernel, is offloaded to GPU. For the processing of data, CUDA implements single instruction multiple thread (SIMT) model which implies that every thread in a block executes a single instruction on different data. CUDA hides memory latency in order to improve parallel performance.
Algorithm 1 is a representative CUDA program for computing a specified parameter. Here, we have discussed the mandatory ingredients of a parallel CUDA program. At first, a kernel CUDA parameter is defined which utilizes the input to update d\text{parameter} (line 4-14). It is then followed by the main program which defines the required variables (line 19-25), allocates memory on host and device appropriately (line 29-36), and read input files (line 45-46). Subsequently, the data is transferred to GPU device (line 49) and a kernel for parallel execution is launched using ‘nblock’ blocks with ‘nthread’ threads each(line 54). Once the calculation in kernel is over, data is copied back from device to host (line 64) and allocated memory is freed (line 69-74).

Algorithm 1: Representative CUDA program

```cpp
// include required files
using namespace std;

// Define kernel
__global__ void CUDA\_parameter(double *d\_x, double *d\_y, double *d\_z, double *d\_parameter, double \_xbox, double \_ybox, double \_zbox, int numatm)
{
    int i;
    i=blockIdx.x*block\_id\_x+thread\_id\_x; // atom index is the thread\_id
    if (i < numatm)
    { //Compute parameter
        //update d\_parameter
    }
}

//Define main program
void main()
{
    //define device variables
    double *d\_x, *d\_y, *d\_z;
    double *d\_dx, *d\_dy, *d\_dz, *d\_dr;
    double *d\_param;
    //define host variables
    double *h\_x, *h\_y, *h\_z;
    double *h\_param;
    int nthreads = 128, nblock;

    //Read numatm = number of atoms
    unsigned long long int sized = numatm * sizeof(double);
    cudaMalloc((void**)&d\_x, sized);
    //allocate other variables
    //allocate host memory e.g.
    h\_x = (double*) malloc(sized);
    //allocate other variables
    //Check for valid allocation e.g. cudaGetLastError()
    //compute number of blocks
    nblock = (int)(numatm/nthreads) + 1;
    //Start loop(1) to read data from file
    //e.g. read data for one frame
    //transfer data from host to device e.g.
    cudaMemcpy(d\_x, h\_x, sized, cudaMemcpyHostToDevice);
    //transfer other data

    //invoke kernel
    CUDA\_parameter<<<nblock, nthreads>>>(
```
Since C and FORTRAN have different ways of storing 2 dimensional arrays, we use linear arrays extensively all throughout the codes.

3 Results and Discussions

Prior to comparing the performance achieved from GPUs, it is confirmed that (i) the computed concerned quantities are in accordance with the reported values in literature and (ii) the outputs obtained from GPU programs are equivalent to those obtained from conventional CPU programs.

3.1 Two Particle Parameters

The RDF or pair correlation function is one of the most common static analysis scheme for molecular simulations. For a given trajectory frame, it depicts the complexity of $O(N_{atom}^2)$ where $N_{atom}$ is the number of particles. However, data from a single frame is not sufficient to satisfy statistical accuracy, an additional loop over the number of frames is also required. Consequently, it exhibits a complexity of $O(N_{frame}(N_{atom})^2)$. Note that calculations in a given trajectory frame are independent on the rest of trajectory frames. Since RDF has been discussed before [22] we do not describe it in detail here. However, both the considered order metrics, $q_{tet}$ and $P_2$, illustrate a common algorithmic feature of finding pair distances, a nested loop over the number of particles, as in the case of RDF. The only difference comes in final processing where resulting pair distances are further utilized to give the respective outcomes. It clearly conveys that similar parallel and logical structures can be utilized to perform accelerated computations for these metrics. It must be noted that the time taken for such a calculation will also depend on the state point, i.e. the temperature and pressure of the system. The time taken can vary as much as 30% is some cases, for both CPU and GPU algorithms. While the additional cost of calculation per particle is accumulated in the CPU code, the same is amortized over the parallel CUDA cores, hence resulting in improved scale ups.
3.1.1 Tetrahedral order parameter

Tetrahedral fluids are the simplest complex fluids which are known to form networks. [34] The $q_{tet}$ parameter has been extensively used to understand the local orderings in water molecules and other tetrahedral atomic fluids, viz. beryllium fluoride and silica [35–37]. In such systems, $q_{tet}$ is used to quantify the extent to which particles are organized in a tetrahedral manner. Mathematically, for a single particle, it is given as

$$ q_{tet} = 1 - \frac{3}{8} \sum_{j=1}^{3} \sum_{k=j+1}^{4} (\cos \psi_{jk} + 1/3)^2 $$

(1)

where $\psi_{jk}$ is the angle between the bond vectors $r_{ij}$ and $r_{ik}$ where $j$ and $k$ label the four nearest neighbour atoms of the same type in the vicinity of atom $i$. [23, 38] The pseudocode of a conventional program for calculation of $q_{tet}$ is given in Algorithm 2.

Algorithm 2: Serial logic for tetrahedral order parameter

1. Allocate memory for array of positions on CPU
2. Allocate memory for arrays of neighbour vectors and $q_{tet}$ on CPU
3. Do frame = 1 to Nframe ! Loop 1 over Nframe frames
4. \hspace{1em} Do i = 1 to Natom ! Loop 2 over Natom particles
5. \hspace{2em} Do j = 1 to Natom ! Loop 3 over Natom particles
6. \hspace{3em} Calculate the minimum image distance vector between particle i and j
7. \hspace{2em} store all distances
8. \hspace{1em} End Do ! End of Loop 3
9. \hspace{1em} Sort the distances to get the four nearest neighbours
10. \hspace{1em} Do j = 1 to 3 ! Loop 4
11. \hspace{2em} Do k = (j+1) to 4 ! Loop 5
12. \hspace{3em} Calculate the $\cos \psi_{jk}$
13. \hspace{2em} Solve Eq. 1
14. \hspace{1em} End Do ! End of Loop 5
15. \hspace{1em} End Do ! End of Loop 4
16. \hspace{1em} store $q_{tet}$ ! End of Loop 2
17. \hspace{1em} End Do ! End of Loop 1
18. Update histogram with $q_{tet}$
19. \hspace{1em} End Do ! End of Loop 1
20. Free memory

As shown in the Algorithm 2, after searching all the neighbour distances, 4 nearest neighbours are obtained and $q_{tet}$ is computed for each particle in each frame of the trajectory in a sequential manner. Since calculations for every atom in each frame are entirely independent, parallel threads on GPU device can be launched for every pair of frame and atom. However, in order to account for memory constraints for larger systems, data only for a single frame can also be sent to the GPU. Now, Loop 2 (Algorithm 2) can be launched in parallel where each thread will perform Loop 3 (Algorithm 2) to compute $q_{tet}$ for one particle. The execution on these threads is scheduled internally by the GPU architecture. The pseudocode of parallel GPU program for the $q_{tet}$ is given in Algorithm 3.

Algorithm 3: Parallel logic for tetrahedral order parameter

1. Allocate memory for arrays of positions and $q_{tet}$ on CPU and GPU
2. Allocate memory for array of neighbour vectors on GPU
3. Do frame = 1 to Nframe ! Loop 1 over Nframe frames
4. \hspace{1em} Transfer array of positions to GPU ! Kernel call. cf. Algorithm 1 line 54–56
5. \hspace{2em} Call CUDA $q_{tet}$ over all atoms
6. \hspace{2em} Transfer $q_{tet}$ histogram to CPU
7. \hspace{2em} End Do ! End of Loop 1
8. \hspace{1em} Free memory
9. \hspace{1em} function CUDA $q_{tet}$
If (thread.id < Natom)
Compute qtet on device by performing Loop 3, 4, 5 of Algorithm 2
Update qtet histogram using atomic operation
End function

It may also be noted that the calling program and the CUDA kernel (CUDA_qtet) are entirely independent and hence can be written in different languages. We use FORTRAN for the former to read the trajectory and writing outputs while CUDA-C is used for the latter to compute qtet. The speedups gained from this approach of only parallelizing over number of particles for different system sizes and available GPU machines are shown in Table 1. Typical speed ups are $\approx 20 \times$ to $\approx 50 \times$ compared to the serial code. This illustrates that considerable performance improvement can be achieved by minimal changes in the program.

| System Size | CPU Time (s) | GTX Time (s) | Scale up | K40 Time (s) | Scale up |
|-------------|--------------|--------------|----------|--------------|----------|
| 6720        | 184          | 7            | 26       | 8            | 23       |
| 13440       | 994          | 19           | 52       | 23           | 43       |
| 26880       | 3560         | 63           | 56       | 60           | 59       |

Table 1: Performance comparisons for computation of qtet using conventional serial CPU and parallel CUDA codes, on GTX and K40 machines. Computation is performed for 100 frames and relative speed up is also given. Note that time of file reading is not included in CPU time ($< 1$ s).

3.1.2 Bond Orientational Parameter

Another order metric which involves discovery of nearest pairs is $P_2$. This local order parameter quantifies the extent to which chain segments are aligned in the vicinity of a given particle and have been used extensively for long chain compounds/polymers. Mathematically, local $P_{2,i}$ order parameter of the $i$th $C$ atom on a chain is defined as

$$P_{2,i} = \frac{1}{n} \sum_{j=1}^{n} \frac{3 \cos^2(\theta_{ij}) - 1}{2}$$  \hspace{1cm} (2)

where $\theta_{ij}$ is the angle between the vector from the $(i-1)$th to the $(i+1)$th $C$ atom and the vector from the $(j-1)$th to the $(j+1)$th $C$ atom that lie within a certain cutoff distance. The value of $P_2$ ranges from 1 to -0.5 for a perfect crystalline arrangements to completely random configurations, respectively. The logical structure for the calculation of $P_2$ is described in Algorithm 4

Algorithm 4: Bond order parameter

1. Allocate memory for arrays of positions and $P_2$ on CPU
2. Do frame = 1 to Nframe  \hspace{1cm} / Loop 1 over Nframe frames
3. Do i = 1 to Natom  \hspace{1cm} / Loop 2 over Natom atoms
4. If (carbon atom i is bonded to two carbon atoms)  \hspace{1cm} / If condition 1
5. Do j = 1 to Natom  \hspace{1cm} / Loop 3 over Natom atoms
6. If (i.ne.j and j is bonded to two carbon atoms)  \hspace{1cm} / If condition 2
7. Calculate the minimum image distance between i and j
8. If (distance is within the cut-off)  \hspace{1cm} / If condition 3
9. Find the vector $(i-1)$th to the $(i+1)$th atom
10. Find the vector $(j-1)$th to the $(j+1)$th atom
11. Solve dot product of vectors to get angle
12. Solve the Eq. 2
13. Assign $P_2$ value to atom i
Just like in $q_{tet}$, it can be parallelized over the number of atoms in each frame. However, in order to efficiently use the GPU architecture, additional parallelism can be exploited over frames. To accomplish this goal, threads are launched for each pair of frame-atom in a kernel call. This ensures that time between the launch of subsequent kernels in the previous case of $q_{tet}$ is eliminated, and all data can be processed in a single kernel call, as described in Algorithm 5. The caveat, of course, is that the full data must fit on the GPU. Since these are embarrassingly parallel algorithms, data can always be designed to fit on a single device e.g. a single trajectory can be split before processing and the results can be merged afterward.

Using these strategies for a system of 2000 carbons [33] with 2000 configurations in the trajectory, serial CPU computation takes 162 seconds. On the other hand, the parallel computation on K40 and GTX machines takes 39 and 14 seconds, respectively. It is not surprising that the scale up achieved is less ($\approx 4\times$) since the calculation is itself very small. Better scale up is expected with the large number of atoms and frames.

### 3.2 Three-Particle Parameter

$$g^{(3)}(r,r') = \left\langle \sum_{i=1}^{N} \sum_{j=1, i\neq j}^{N} \delta(r-r_i)\delta(r'-r_j) \right\rangle$$

Three-particle correlation function (TPCF) $g^{(3)}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)$ can be described as the probability of finding three particles simultaneously at positions $\mathbf{r}_1$, $\mathbf{r}_2$, and $\mathbf{r}_3$ in the system irrespective of the configuration of the remaining $(N-3)$ particles [29,39]. Unlike RDFs, which gives angle-averaged data, TPCF can be used to obtain angle-resolved information. The computation of $g^{(3)}$ involves 3D histogram binning of distances with imposed conditions. In comparison to two-particle correlation, it involves an additional loop over the number of particles reduces the performance and enhances analysis time. The serial logic for TPCF is shown in Algorithm 6.

**Algorithm 5: Parallel bond order parameter**

```plaintext
Allocate memory for arrays of positions and $P_2$ on GPU and CPU
Transfer all data to GPU
Call CUDA_p2! Calling parallel calculation for all the atoms of all frames
Transfer results to CPU
Free memory
```

```plaintext
function CUDA_p2
If (thread_id < Nframe*Natom) ! If condition 1
Frame index = int(thread_id/Natom) ! Fetching frame index
Atom index = (thread_id %Natom) ! Fetching atom index for atom i
If (carbon atom i is bonded to two carbon atoms) ! If condition 2
Compute $P_2$ value for each carbon using Loop 3 of Algorithm 4
End If
End If
End function
```

Using these strategies for a system of 2000 carbons [33] with 2000 configurations in the trajectory, serial CPU computation takes 162 seconds. On the other hand, the parallel computation on K40 and GTX machines takes 39 and 14 seconds, respectively. It is not surprising that the scale up achieved is less ($\approx 4\times$) since the calculation is itself very small. Better scale up is expected with the large number of atoms and frames.

**Algorithm 6: Serial logic for triplet correlation function**

```plaintext
Allocate memory for arrays of positions, updated positions on CPU
Allocate memory for $g_2$ and $g_3$ histograms on CPU
```

3
Do frame = 1 to Nframe  
! Loop 1 over Nframe frames
Do i = 1 to Natom  
! Loop 2 over Natom atoms
Do j = (i+1) to Natom  
! Loop 3
Calculate minimum image distance between atom i and j
Compute and store new coordinates of neighbour particles
Assign two particle histogram
End Do  
! End of Loop 3
Do j = 1 to Natom−1  
! Loop 4 over Neighbour particles
If (distance of i and j is within Boxlength/4 )  
! If condition 1
Do k = (j+1) to Natom−1  
! Loop 5
If (distance of i and k is within Boxlength/4 )  
! If condition 2
Calculate the distance using updated positions
If (distance of j and k is within Boxlength/2 )  
! If condition 3
Update g3 histogram
End If  
! End of if condition 3
End If  
! End of if condition 2
End If  
! End of if condition 1
End Do  
! End of loop 5
End Do  
! End of loop 4
End Do  
! End of loop 2
End Do  
! End of loop 1
End Do  
! End of loop 1
Free memory

In the case of CUDA program, slightly different logic is used. First, pair histograms and pair vectors are computed in parallel for all particles in a configuration. Subsequently, kernels are launched for each particle pair asynchronously. Each thread computes and updates the global $g^{(3)}$ 3D histogram using atomic operations, as shown in Algorithm 7. n.b. $g^2$ histogram is computed with cut-off of 1/2 of box length while neighbour vectors are stored within a cut-off of 1/4 of box length. Note that the neighbor vectors computed in kernel CUDA_g2 (Algorithm 7) are utilized while computing g3 histogram.

Algorithm 7: Parallel logic for triplet correlation function

Allocate memory for arrays of positions on GPU and CPU
Allocate memory for arrays of neighbour vectors on GPU only
Allocate memory for g2 and g3 histograms on CPU and GPU
Do frame = 1 to Nframe  
! Loop 1 over Nframe frames
Transfer data for this frame to GPU
Do i = 1 to Natom  
! Loop 2 over Natom atoms
Call CUDA_g2  
! Calling parallel calculations over Natom
Do j = 1 to Natom−1  
! Loop 3 over Neighbour particles
Call CUDA_g3  
! Calling parallel calculation over neighbour vectors
End Do  
! End of loop 2
End Do  
! End of loop 1
Transfer g2 and g3 histogram to CPU
Free memory

function CUDA_g2
If (thread_id < Natom)  
! If condition 1
Do j = 1 to Natom  
! Loop 3 over Natom atoms
Compute and store g2 histogram for atom i
Compute and store minimum image neighbour vectors for atom i.
End Do  
! End of Loop 3
End If  
! End of if condition 1
End function

function CUDA_g3
Identify current neighbour j of atom i as thread_id
Do k = 1 to number of neighbours of i  
! Loop 4 over neighbours
Compute j−k distance
If distance j−k < boxlength/2  
! If condition 2
Update g3 histogram
End If  
! End of if condition 2
End Do  
! End of Loop 4
End function
Table 2 shows the time consumed for the computation of TPCF. We note here that serial computation of \(g^{(3)}\) is extremely slow and usually requires large system sizes as well as a long trajectory for convergence and good ensemble averages. For example, the serial code (unoptimized) takes \(\approx 3\) hours to compute \(g^{(3)}\) for 10 frames only in case of largest system size considered here (Table 2). In contrast, the same computation on a consumer GTX780 takes \(\approx 13\times\) less time, i.e. \(\approx 14\) minutes.

| System Size | CPU Time (s) | GTX Time (s) | Scale up | K40 Time (s) | Scale up |
|-------------|--------------|--------------|----------|--------------|----------|
| 6720        | 596          | 13           | 45       | 16           | 37       |
| 13440       | 2089         | 104          | 20       | 132          | 16       |
| 26880       | 11515        | 860          | 13       | 1124         | 10       |

Table 2: Performance comparisons for computation of \(g^{(3)}\) using conventional serial CPU and parallel CUDA codes, on GTX and K40 machines. Computation is performed for 10 frames and relative scale up is given. Time taken for file reading is included in the data. Time taken for writing output is 2-3s on GTX and 12-14s on K40, not included in the data above.

3.3 Correlation functions

3.3.1 Viscosity

In equilibrium molecular dynamics, viscosity can be computed by using Green-Kubo formalism. This formalism utilizes auto-correlation function of the off-diagonal terms of pressure tensor in Equation 4. Since correlation of pressure tensor only depends on data points, rather than system size, these calculations were performed only for the system containing 6720 water molecules. For the computation, a total of \(2 \times 10^6\) data points are considered with correlation time corresponding to \(5 \times 10^5\) data points. The corresponding logic for computing viscosity is described in Algorithm 8.

\[
\eta(dt) = \frac{V}{k_B T} \int_0^\infty dt \langle P_{\alpha\beta}(t) P_{\alpha\beta}(t + dt) \rangle 
\]  

Algorithm 8: Serial logic for auto-correlation function

1. Do \(dt = 0\) to \(N\)frame−1  
2. Do \(t = 1\) to \(N\)frame−dt  
3. Compute product of pressures at different times  
4. Update correlation array  
5. End Do  
6. End Do  
7. Do \(dt = 0\) to \(N\)frame−1  
8. Perform integration using quadrature method for Eq. 4  
9. End Do

Since all computations of the product are independent, we parallelize over correlation times as shown in Algorithm 9.

Algorithm 9: Parallel logic for auto-correlation function

1. Allocate memory for time data and correlation array on CPU and GPU  
2. Transfer time data to GPU  
3. call CUDA_acf  
4. Transfer correlation array to CPU  
5. Do \(dt = 0\) to \(N\)frame−1  
6. Perform integration using quadrature method for Eq. 4
Using this strategy, the computing times obtained for CPU, GTX, and K40 are 1534 s, 76.2 s, and 84.1 s which indicate the scale of $\approx 18-20 \times$ of GPUs over CPU.

### 3.3.2 Mean Square Displacement

Mean square displacement (MSD) averaged over all the atoms can be used to obtain self-diffusion constant by utilizing Einstein relation, given as [26]

$$D = \lim_{t \to \infty} \frac{1}{2dt} \langle |r(t) - r(0)|^2 \rangle$$

(5)

where $d$ is the dimensionality of the system, while the angular bracket $\langle \cdots \rangle$ denotes the MSD with averaging done over all the atoms (or all the atoms in a given subclass), to improve statistical accuracy. As shown in Algorithm 10, MSD involves looping over all atoms, all time origins and correlation times. The pseudocode for serial calculation is given in Algorithm 10.

**Algorithm 10: Serial logic for mean square displacement**

```
1 ! Starting from unwrapped trajectory
2 Do i = 1 to Natom ! Loop 1 over the particles
3    Do dt = 0 to Nframe−1 ! Loop 2 over time interval
4       Do t = 1 to Nframe−dt ! Loop 3 over number of frames
5          Compute MSD of particle i between t and t+dt
6          Accumulate MSD array
7     End Do ! End of Loop 1
8    End Do ! End of Loop 2
9 ! End of Loop 3
```

Just like the ACF described above (Algorithm 8), MSD can be parallelized over the correlation time (dt). It can be, additionally, parallelized over all atoms. This also affords division of data over multiple GPU, if required. We describe this in Algorithm 11.

**Algorithm 11: Parallel logic for mean square displacement**

```
1 ! Starting from unwrapped trajectory
2 Allocate memory for arrays of positions and msd on CPU and GPU
3 Transfer unwrapped position arrays to GPU
4 Call CUDA_msd
5 Transfer msd array from GPU
6 Free memory
7
8 function CUDA_msd
9 dt and atom index is obtained from thread−id
10 Do t = 1 to Nframe−dt ! Loop 1 over number of frames
11    Compute MSD of particle i between t and t+dt
12    Accumulate local MSD data
13 End Do ! End of Loop 1
14 Update global MSD array
15 End function
```

MSD computation is among the most memory intensive calculation in this study and hence is considerably aided by splitting data sets over atoms. On a single GPU, scale up of $\approx 80 \times$ are
Table 3: Performance comparisons for computation of mean square displacement using conventional serial CPU and parallel CUDA codes, on GTX and K40 machines. Computation is performed for 3500 frames and relative speed up is also given. Note that time of file reading is not included in CPU time.

| System Size | CPU Time (s) | GTX Time (s) | Scale up | K40 Time (s) | Scale up |
|-------------|--------------|--------------|----------|--------------|----------|
| 6720        | 664          | 8            | 83       | 10           | 66       |
| 13440       | 1310         | 15           | 87       | 21           | 62       |
| 26880       | 2600         | 31           | 83       | 43           | 40       |

easily achieved for systems with $\approx 26000$ particles, as shown in Table 3, i.e. from 43 minutes to 31 seconds on a GTX780 for 2600 particles. n.b. 3500 frames were chosen so as to completely utilize memory of a GTX780 card. The time saved by such an implementation is evident in computations where there are a larger number of state points, e.g. diffusivity calculations for more than 300 state points for n-alkanes in Ref [29].

### 3.4 Parallelization over CPUs

In the case of trajectory analysis algorithms, it is possible to devise shell script wrappers for serial codes which would make the calculation parallel over the available hardware. For example, a serial code can be converted into a parallel code over $n$ cores by breaking the trajectory into multiple sets and using a wrapper script or program to run control the execution. A separate program would be needed to collate the results from the multiple serial runs. Assuming near 100% efficiency and ignoring file reading/writing overheads, the best scale up with this method would be $n \times$. For example, in this study, the best scale up we can expect is $24 \times$ for the K40 machine and $6 \times$ for the GTX machine. Considering that each machine has 2 and 1 GPUs respectively, a similar GPU scale up, e.g. for computation of MSD, would be $\approx 80 \times$ (on both machines). It must be noted that if a CPU parallel scheme is used, the entire machine would be unavailable for other programs. On the other hand, if a GPU code is used, the other CPU cores would be available for processes. As shown in the results, there are very few cases where the GPU scale up over the serial code is less than $10 \times$, and hence would be more efficient than using a parallel CPU code.

Studies involving large number of state points, for example Ref [29], the combined time saved is of considerable importance, considering the cost of high performance computing. In Ref [29], $\approx 300$ state points have been simulated, and $g^{(3)}$ was computed with 5000 frames for each state point. With a conventional CPU code, the estimated computation time would be $\approx 43-44$ days parallelized over 24 cores (see Table 2). In comparison, the task would be completed in $\approx 28$ days on 2 K40 GPUs, or approx 16-17 days on 2 GTX 780 GPUs including all file I/O.

### 4 Conclusion

MD simulations are routinely used as investigating tool for a range of systems. Large system sizes in MD simulations result in large trajectories which demand swift analysis of multiple varieties. Like the simulations themselves, the analysis too have some parallelizable regions which can be easily implemented on NVIDIA GPUs, and hence it is imperative to port these codes from CPU to GPU. Multiple algorithms, with low to moderate complexities, are described with ease of implementation in mind. Although the corresponding OpenMP and MPI based algorithms can be coded, the scale up demonstrated above can be achieved only after significant
optimization and code modification. Even though the usage cost of GPUs is $\approx 10 \times$ that of a single CPU core in a typical commercial HPC setup, the effective cost of analysis on GPU would be considerably lower than that on CPUs. Since these codes do not use any advanced GPU architecture based functionality, they would be compatible with future generations of NVIDIA devices.

Static properties such as $q_{tet}$, $P_2$, and $g^{(3)}$ can be easily parallelized over atoms as well as frames while the trajectory can be partitioned to conform to memory limitations and make use of multiple devices. Using the approaches described in this work, the process of discovery of both pairs and triplets can be accelerated up to a significant extent. For dynamic quantities, such as MSD, all trajectory data for a single species are required to be present on the device and, therefore, data is needed to be divided over specific atoms or species. Hence, parallelization can be done over trajectory frames or atoms as per requirement. Typical scale up obtained are $10\times$–$80\times$ efficient than conventional CPU codes, allowing researchers to quickly analyze data on both server class NVIDIA GPUs as well as Desktop and workstation level consumer GPUs. Since similar approaches such as autocorrelation functions and neighbour search are also used in other scientific fields, these algorithms can be easily extended.

Acknowledgement

This paper is dedicated to the memory of Late Prof. Charusita Chakravarty. The work described above started in January 2014 with the aim of reducing time taken for computation of three particle correlation function in the group. G.S. thanks the CSIR, India for granting the Senior Research Fellowship (F. No. - 09/086(1141)/2012-EMR-I). Authors thank the IIT Delhi HPC facility for computational resources. Authors appreciate discussions with NVIDIA development team.

References

[1] Yasuda K. *J. Chem. Theory Comput.* **2008**, *4*, 1230–1236.

[2] Yasuda K. *J. Comput. Chem.* **2008**, *29*, 334–342.

[3] Ufimtsev IS, Martinez TJ. *Comput. Sci. Eng.* **2008**, *10*, 26–34.

[4] Anderson JA, Lorenz CD, Travesset A. *J. Comput. Phys.* **2008**, *227*, 5342–5359.

[5] Liu W, Schmidt B, Voss G, Müller-Wittig W. *Comput. Phys. Comm.* **2008**, *179*, 634–641.

[6] Sunarso A, Tsuji T, Chono S. *J. Comput. Phys.* **2010**, *229*, 5486–5497.

[7] Spellings M, Marson RL, Anderson JA, Glotzer SC. *J. Comput. Phys.* **2017**, *334*, 460–467.

[8] Anandakrishnan R, Daga M, Onufriev A, Feng WC. Multiscale Approximation with Graphical Processing Units for Multiplicative Speedup in Molecular Dynamics. *Proc. 7th ACM Int. Conf. Bioinformatics, Comput. Biol. Heal. Informatics - BCB ’16*, ACM Press: New York, New York, USA, 2016.

[9] John E Stone, David J Hardy, Ivan S Ufimtsev, Schulten K. *J. Mol. Graph Model.* **2011**, *29*, 116–125.

[10] Ponce R, Cárdenas-Montes M, Rodriguez-Vazquez JJ, Sánchez E, Sevilla I. *arXiv Prepr. arXiv:1204.6630* **2012**, **.”
[11] Bard D, Bellis M, Allen M, Yepremyan H, Kratochvil J. Astron. Comput. 2013, 1, 17 – 22.
[12] Plimpton S. J. Comput. Phys. 1995, 117, 1–19. URL http://lammps.sandia.gov
[13] Brown WM, Wang P, Plimpton SJ, Tharrington AN. Comput. Phys. Comm. 2011, 182, 898–911.
[14] Brown WM, Kohlmeyer A, Plimpton SJ, Tharrington AN. Comput. Phys. Comm. 2012, 183, 449–459.
[15] Spoel DVS, Lindahl E, Hess B, Groenhof G, Mark AE, Berendsen HJC. J. Comput. Chem. 2005, 26, 1701–1718.
[16] Todorov IT, Smith W, Trachenko K, Dove MT. J. Mater. Chem. 2006, 16, 1911–1918.
[17] Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kale L, Schulten K. J. Comput. Chem. 2005, 26, 1781–1802.
[18] Brooks BR, Brooks CL, MacKerell AD, Nilsson L, Petrella RJ, Roux B, Won Y, Archontis G, Bartels C, Boresch S, et al. J. Comput. Chem. 2009, 30, 1545–1614.
[19] Frisch MJ, Trucks GW, Schlegel HB, Scuseria GE, Robb MA, Cheeseman JR, Scalmani G, Barone V, Petersson GA, Nakatsuji H, et al. Gaussian16 Revision A.03. 2016. Gaussian Inc. Wallingford CT.
[20] Bailey NP, Ingebrigtsen TS, Hansen JS, Veldhorst AA, Bøhling L, Lemarchand CA, Olsen AE, Bacher AK, Costigliola L, Pedersen UR, et al. arXiv Prepr. arXiv1506.05094 2015, .
[21] Harvey MJ, Giupponi G, Fabritiis GD. J. Chem. Theory Comput. 2009, 5, 1632–1639.
[22] Levine BG, Stone JE, Kohlmeyer A. J. Comput. Phys. 2011, 230, 3556–3569.
[23] Chau PL, Hardwick AJ. Mol. Phys. 1998, 93, 511.
[24] Yi P, Locker CR, Rutledge GC. Macromolecules 2013, 46, 4723–4733.
[25] Baranyai A, Evans DJ. Phys. Rev. A 1989, 40, 3817–3822.
[26] Frenkel D, Smit B. Understanding molecular simulation: From algorithms to applications. Academic Press, 2002.
[27] Martin MG, Siepmann JL. J. Phys. Chem. B 1998, 102, 2569−2577.
[28] Chopra R, Truskett TM, Errington JR. J. Phys. Chem. B 2010, 114, 16487–93.
[29] Shrivastav G, Agarwal M, Chakravarty C, Kashyap HK. J. Mol. Liq. 2017, 231, 106 – 115.
[30] Berendsen HJC, Grigera JR, Straatsma TP. J. Phys. Chem. 1987, 91, 6269.
[31] Jorgensen WL, Maxwell DS, Tirado-Rives J. J. Am. Chem. Soc. 1996, 118, 11225–11236.
[32] Martínez L, Andrade R, Birgin EG, Martínez JM. J. Comput. Chem. 2009, 30, 2157–64.
[33] Shrivastav G, Agarwal M. J. Phys. Chem. B 2016, 120, 7598–7605.
[34] Nayar D, Chakravarty C. Phys. Chem. Chem. Phys. 2013, 15, 14162–14177.
[35] Agarwal M, Chakravarty C. J. Phys. Chem. B 2007, 111, 13294.
[36] Agarwal M, Chakravarty C. *Phys. Rev. E* **2009**, *79*, 030202(R).

[37] Agarwal M, Singh M, Sharma R, Alam MP, Chakravarty C. *J. Phys. Chem. B* **2010**, *114*, 6995.

[38] Errington JR, Debenedetti PG. *Nature* **2001**, *409*, 318.

[39] Shrivastav G, Gupta A, Rastogi A, Dhabal D, Kashyap HK. *J. Chem. Phys.* **2017**, *146*, 064503.