GPU-BASED PARALLEL ALGORITHM OF INTERACTION INDUCED LIGHT SCATTERING SIMULATIONS IN FLUIDS

ALEKSANDER DAWID

Department of Transport and Computer Science
University of Dąbrowa Górnicza
Cieplaka 1c, 41–300 Dąbrowa Górnicza, Poland

(received: 9 November 2018; revised: 4 December 2018; accepted: 20 December 2018; published online: 3 January 2019)

Abstract: We parallelized the sequential algorithm of the four-body correlation function if each combination of two pairs (i,j) and (k,l) was averaged over the time in a separate calculation thread. The generator of pairs used as the input for this algorithm was also parallelized and connected with the 4-body correlation function calculations. We used our algorithm to accelerate extremely intensive calculations of the 4-body polarizability anisotropy correlation functions, which were very important to estimate the interaction induced light scattering spectrum. The resulting C code was used to test our algorithm on Graphics Processing Units (GPUs) with the Compute Unified Device Architecture (CUDA) technology from NVIDIA® Corporation. As a result, we achieved 12 times the acceleration of the 4-body correlation function calculations in comparison to the Central Processing Unit (CPU) core. The peak performance of the GPU calculations was registered at the level of 19 times faster than the CPU core. We also found that acceleration depended on the memory consumption. In the single precision mode, the relative error between the CPU and GPU calculations was found to be within 0.1%.

Keywords: GPGPU, CUDA, interaction induced phenomena, many body correlation function, parallel algorithm

DOI: https://doi.org/10.17466/tq2019/23.1/a

1. Introduction

The calculation of many-body correlation functions (MBCF) is very important in many fields of physics, chemistry and material sciences. The MBCFs are applied to the theoretical study of liquids spectra [1], to investigate the properties of heavy nuclei [2], to describe the dynamics of distributions of ion pairs as the issue of electric conductivity of electrolytic solutions [3], to study the structural relaxation of liquids near the glass transition [4, 5], and to many other applications where many-body interactions are considered. One of the areas where the

many-body correlation functions are used is the interaction-induced light scattering process. It comes from the fact that two colliding atoms induced short-lived dipole moment capable of interacting with an incident light beam. The polarizability anisotropy created in the system can be described by the dipole-induced-dipole mechanism (DID) [6]. The Fourier transform of the polarizability anisotropy correlation function is known as the Rayleigh light scattering spectrum. Although the DID mechanism is a two-body interaction, it gives rise to two-, three-, and four-body correlations contributing to the total intensity of scattered light. The depolarized light scattering spectra can be measured experimentally. On the other hand, we can calculate the spectra using the computer simulation technique. In this way, we are able to assign the microscopic mechanism in the studied system to the experimental data. The calculations of these many-body correlation functions consume a lot of time and in practice they are manageable only for small physical systems, like clusters [7–10] and ultrathin layers [11–14]. The acceleration of these calculations would help scientists analyze the simulation results for larger systems, like biological ones [15–17]. The new possibility of fast, scientific calculations has been opened due to the game market, which caused the rapid development of graphics accelerators. Modern GPUs are highly parallel, they have a high bandwidth memory system, containing more than a thousand cores and they are programmable in the sense of general computations. The GPU provides much faster floating-point performance than a typical CPU, at a comparable price. The top supercomputers in the world are made of GPU-based clusters. The programming platform widely used in scientific computations is the CUDA proposed by NVIDIA Corporation [18–20]. It works as an extension to the C/C++ library and allows entering the output from a graphic card as a character code. Before the release of the CUDA technology the whole output of GPU calculations was sent to the frame buffer in the form of a pixel matrix. Using this library on CUDA enabled devices on which the programmer can make all input/output operations in a terminal mode. There are a lot of publications using the CUDA technology in different fields of scientific calculations [21–27]. The achieved acceleration of calculations varies from ten to over a thousand times compared to calculations on one core of the CPU. The most efficient calculations on the GPU have been reported for the fluid flow algorithms [28–32]. In this report, we present a parallel algorithm for the calculation of a computationally intensive 4-body correlation function based on the input data taken from a molecular dynamics (MD) simulation using the CUDA programming technology.

2. Problem background

One of the examples where many body correlation functions are calculated is the interaction-induced light scattering. The whole process can be described as follows. In the case when two electron clouds overlap with each other the polarizability of such system is no longer isotropic. This anisotropy generates a short-time dipole moment that is able to interact with the electromagnetic

radiation. The computer simulation of the phenomena needs a physical model of polarizability anisotropy. The common approach is to use the DID model [6], where the pair anisotropy β_{ij} is described by the following equation

$$\beta_{ij}(t) = \sigma^3 \left[3x_{ij}(t)z_{ij}(t)/r_{ij}^5(t)\right] \tag{1}$$

where x_{ij} and z_{ij} are components of the separation vector r_{ij} between the ith and j^{th} atoms. The depolarized Rayleigh spectrum is the Fourier transform of the polarizability anisotropy autocorrelation function $G(t)$, which for a monatomic sample of N atoms (Figure 1) is

$$G(t) \propto \left\langle \sum_{i,j,k,l=1, i\neq j, k\neq l}^{N} \beta_{ij}(t)\beta_{kl}(0) \right\rangle \tag{2}$$

where i,j,k,l identify different atoms. The total correlation function $G(t)$ can be decomposed into pair, triplet, and quadruplet contributions

$$G(t) = G_2(t) + G_3(t) + G_4(t) \tag{3}$$

where

$$G_2(t) \propto \left\langle \sum_{i,j=1, i\neq j}^{N} \beta_{ij}(t)\beta_{ij}(0) \right\rangle \tag{4}$$

$$G_3(t) \propto \left\langle \sum_{i,j,k=1, i<j, i\neq k, i\neq j}^{N} \beta_{ij}(t)\beta_{ik}(0) \right\rangle \tag{5}$$

$$G_4(t) \propto \left\langle \sum_{\substack{i,j,k,l=1, i<j, \\ i\neq k, k<l, i\neq l, j\neq l, j\neq k}}^{N} \beta_{ij}(t)\beta_{kl}(0) \right\rangle \tag{6}$$

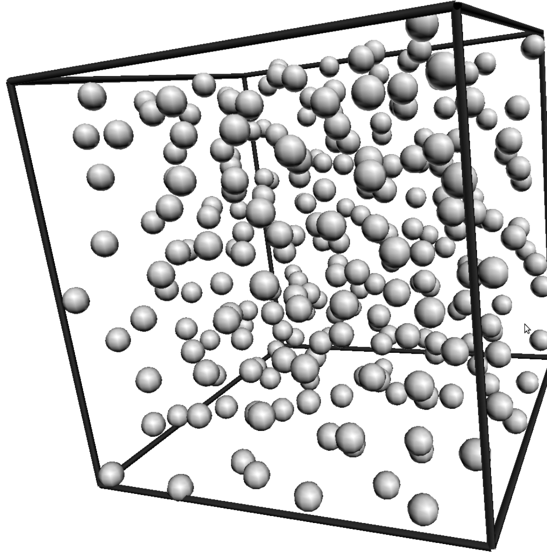

Figure 1. Monoatomic sample of N atoms

As we can see, the most computationally demanding is the four-body correlation function described by Equation (6). It contains four loops over particles and two loops over time. We have used this function as an example to present our GPU-based parallel algorithm. All contributions to the $G(t)$ function were tested and validated in the calculation of the interaction induced spectra of argon between graphite walls [33].

3. Algorithm and implementation

3.1. CPU

The sequential algorithm for solving the time-dependent $G_4(t)$ function is straightforward. As the analog to the two-body correlation function [34], we can build the algorithm with total six loops, two loops over the time and four loops over particles (Algorithm 1).

Algorithm 1. The four-body correlation function – sequential CPU code

```
 1. SET N to be equal to number of atoms
 2. SET TMAX to be the total simulation time
 3. SET input beta to be the polariazability anisotropy
        3D matrix of dimensions N,N,TMAX
 4. FOR t=1 to TMAX
 5. SET output AVER to zero
 6. SET TORIG = TMAX - t
 7.  FOR i=1 to N
 8.   FOR j=i+1 to N
 9.    FOR k=1 to N
10.     FOR l=k+1 to N
11.      IF i≠k AND j≠k AND i≠l AND j≠l THEN
12.        SET ATA to zero
13.       FOR tau=1 to TORIG
14.        ATA = ATA + beta[i][j][tau]*beta[k][l][tau+t]
15.       END FOR
16.       ATA = ATA / TORIG
17.       AVER = AVER + ATA
18.      END IF
19.     END FOR
20.    END FOR
21.   END FOR
22.  END FOR
23. print t, AVER
24. END FOR
```

The most efficient way of calculating the 4-body polarizability anisotropy correlation function is to put the values of $\beta_{ij}(t)$ (Equation (1)) into the three dimensional table `beta[N][N][Tmax]` (line 14). Although this method is the fastest, it needs enormous amount of random-access memory (RAM), thus limiting the trajectory lengths. The compromise between speed and memory usage assumes that calculations of the $\beta_{ij}(t)$ functions are inside the loops. In our CPU algorithm we used the fastest, memory intensive, version with the pre-calculated values of $\beta_{ij}(t)$ to compare the obtained results with the GPU algorithm. The resulting CPU

code was written in a clear C language and compiled using the GNU C compiler on a Linux system.

3.2. GPU

In the development of a parallel version of algorithm 1, we used the CUDA programming technique. The threads in this model have their own unique identifiers. Each identifier represents a separate thread in the computation process. A common way of implementing any algorithm on CUDA enabled devices is to replace the outer loop in multi-loop calculations by a thread index. The main problem in constructing the algorithm for CUDA enabled devices is to find the best decomposition of the outer loop to fulfill the load balance over the available cores. The number of threads should not be greater than the maximum number of threads for the specific device. In other words, the number of threads should not be too small because of the idle threads in the device. The best practice of CUDA programming is to keep all cores busy on the device [35]. In our parallelization of the algorithm, we used a thread identifier to enumerate a set of different pairs of atoms (i,j) and (k,l). The schematic calculation stream of the GPU algorithm is shown in Figure 2. Each identifier `idx` is associated with a pair of atoms (i,j) and represents a single thread executed in a separate core. The initial values of k and l

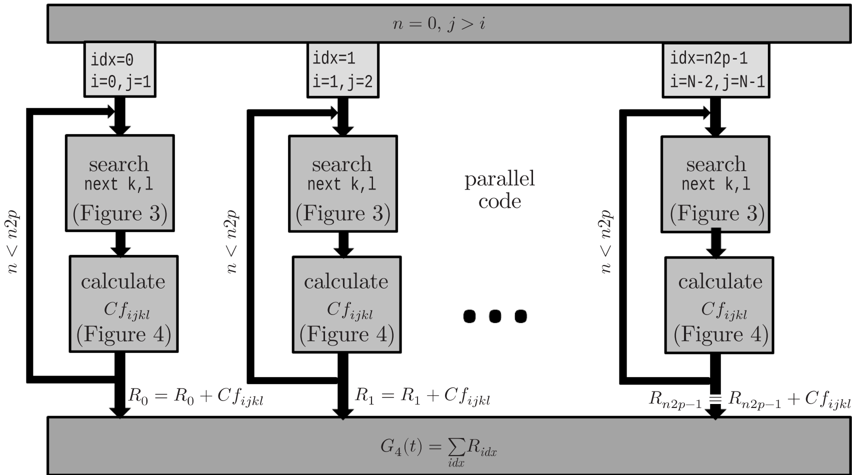

Figure 2. The flowchart of GPU calculations

indices for each thread were set as to fulfill the conditions $k \neq i$ and $k \neq j$ and $l > k$ and $l \neq i$ and $l \neq j$. The part of the algorithm in a single thread starts from finding the next (k,l) indices according to the conditions in Equation (6) (Figure 3). Next, the partial results of the $G_4(t)$ function in a single thread are calculated and summed up. These two steps are repeated until the last pair of indices (k,l). The process takes almost the same amount of time for each starting values of pairs (i,j) (see Figure 2). After that, all threads are synchronized and the values of the

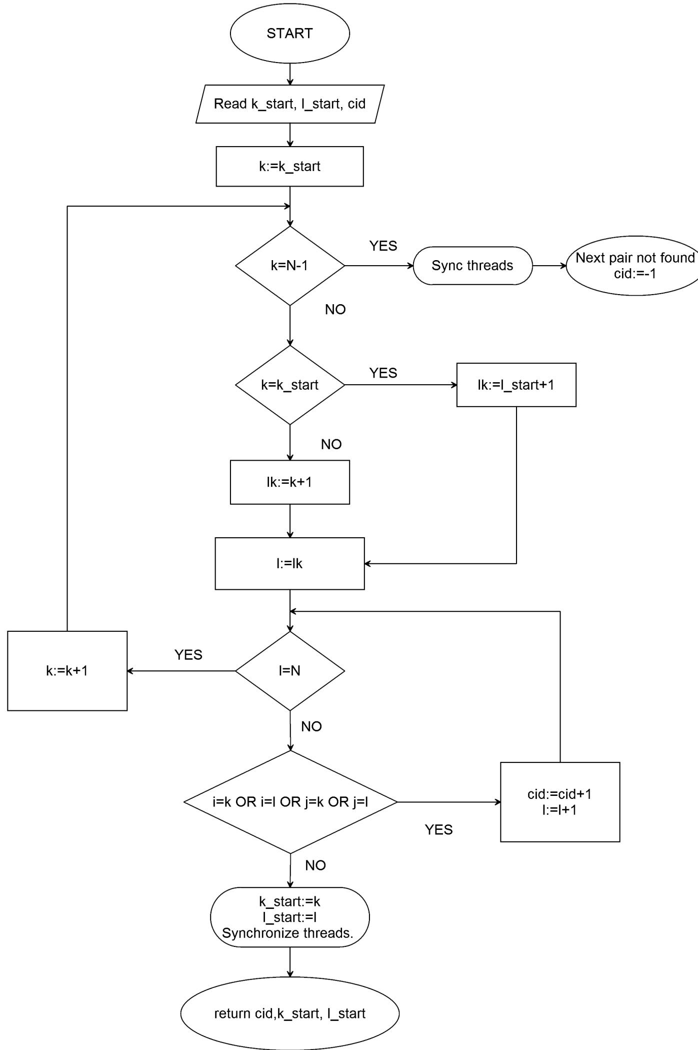

Figure 3. Flowchart of next pair search algorithm

partial $G_4(t)$ function are summed between parallel threads using the reduction algorithm [18]. The flowchart of the searching algorithm is shown in Figure 3. The algorithm starts from $k=0$ and $l=0$ for each GPU thread. The first loop over k ends if the value of k is greater than $N-1$, where N denotes the number of atoms, or if new $k \neq i$ and $k \neq j$ is found. The symbol N denotes the number of atoms. In the next loop the value of l index is searched under the condition that l is bigger than k and l is not equal i and j. The most important index in this algorithm is *cid*. It is responsible for the particular pair of atoms. For example; $cid=0 \rightarrow i=0$ and $j=1$, $cid=1 \rightarrow i=0$ and $j=2, ...cid=N(N-1)/2 \rightarrow i=N-1$ and $j=N$. This sequence satisfies the condition $j>i$. If we now replace the i and j indices by k and l we can obtain the particular *cid* representing our pairs. If the next pair of indices

is not found, the $cid = -1$ indicates the end of calculations for this GPU kernel. The main target of the algorithm is to find the *cid* index for the pair of atoms. The successive values of pairs (k,l) are stored in the local memory of the CUDA device and are used to set the starting values for the next search. The 4-body polarizability anisotropy correlation function is calculated with respect to the `idx` and *cid* values (Figure 4). The algorithm uses the values of $\beta_{ij}(t)$ previously stored in the table. The loop over `tau` sums `Beta[ij_tau]*Beta[kl_tau_t]` over the whole stored trajectory for specific combinations of pairs. The variable `ij_tau` is associated with `idx` and represents a GPU thread identifier. The variable `kl_tau_t` is associated with a calculated *cid* number. Finally, the result is multiplied by the reciprocal of the total simulation time.

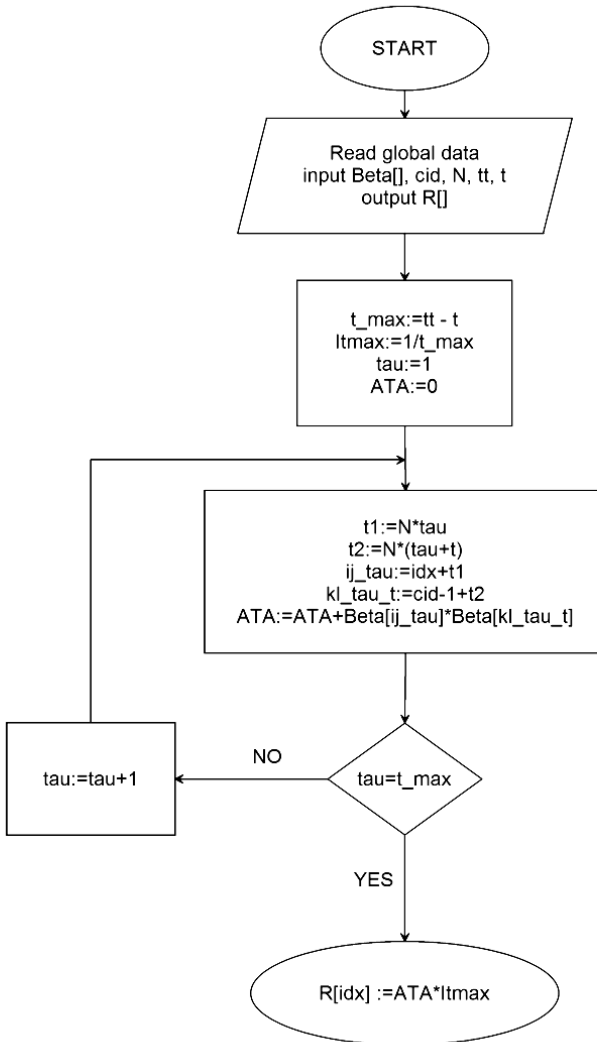

Figure 4. Flowchart of correlation function calculation algorithm for single thread

3.3. CUDA kernels

Our calculations of the $G_4(t)$ function were performed on five CUDA kernels. In Listing 1, we can see the sequence of GPU kernels invoked by the CPU code. The parameter `grid` represents the topology of a network of threads. In our case the grid is one dimensional. The second parameter TPB represents the number of threads per block [18]. The kernel number 1, called `PairIndex()`, according to the flowchart in Figure 3, calculates the successive values of (k,l) pairs and stores its index in a `d_PairTab` structure, consisting of four unsigned 16 bit values. The next kernel called `CalcFuncPair()` calculates the $G_4(t)$ function for one set of pairs and stores the result in the `d_ArTab` table according to the flowchart in Figure 4. The `d_ArTab` and `d_PairIndex` tables are allocated in the global GPU memory. The loop over n calculates the partial $G_4(t)$ function as the sum of pair sets per each GPU thread. The value of `n4p` represents the number of (k,l) pairs. The values of the partial $G_4(t)$ function are summed inside each calculation thread and stored in the `d_ArTab[idx]` variable. Finally, we have to sum up over all values in the `d_ArTab` table to obtain the total $G_4(t)$ function. For this we are using the reduction procedure conducted in two steps. The `Reducto` GPU kernel is the main procedure of calculating the final $G_4(t)$ function and `ReductoEnd` only stores the result in the `d_Cvv` table depending on the current time step `t`. In order to produce the $G_4(t)$ function for the next time step we need to reset the `d_PairTab` table in procedure `ZeroPairs()`. The code of this solution is available on GitHub for the purpose of reproducing the results [36]. The source code was compiled and tested in UBUNTU 16.04 with CUDA toolkit 10.1

Listing 1. CUDA GPU kernels.

```
   for(n=0;n<n4p;n++)
   {
1.  PairIndex<<<grid, TPB>>>(d_P2T, d_PairTab);
2.  CalcFuncPair<<<grid, TPB>>>(d_Beta, d_ArTab, d_PairTab, t);
   }
3. Reducto<<<grid, TPB, sharedSize>>>(d_ArTab, d_ArTab);
4. ReductoEnd<<<grid2, TPB,
                sharedSize>>>(d_ArTab, d_ArTab, d_Cvv, t);

5. ZeroPairs<<<grid, TPB, sharedSize>>>(d_PairTab);
```

4. Results

In order to test the GPU acceleration of calculations, we prepared several molecular dynamics (MD) simulations of argon atoms with orthogonal periodic boundary conditions in an NVT ensemble. The equation of motion in these simulations was solved by using a Velocity Verlet algorithm [37] with the integration time step equal to 2.5 fs. The temperature was controlled by a Berendsen ther-

mostat [37]. The interaction potential between argon pairs is taken to be the Lennard-Jones (LJ) potential with the usual form

$$V(r_{ij}) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{7}$$

where r_{ij} is the distance between atoms, $\varepsilon = 10.34\,\mathrm{meV}$ and $\sigma = 3.4\,\mathrm{A}$ are the LJ potential parameters for argon (19). The system of argon atoms was equilibrated for 10^6 MD steps. The total time of a single simulation was 2 ns. All simulations were performed using the author's simulation program, named RIGMD [38]. An example of the instantaneous configuration of the system is shown in Figure 1. The trajectory results from the simulations were stored in text files using the xyz format. We prepared 17 simulations with the number of atoms N ranging between 60 and 256. After equilibration, we performed a production run of 10^6 time steps. The resulting trajectory was recorded every 50 steps. We obtained a time dependent array of 20 000 locations of each atom in the simulation. Next, we used this data to calculate the $G_4(t)$ function on both the CPU and GPU units. The speedup factor was calculated as the ratio of the CPU to GPU time needed to complete the calculation task.

$$speedup = \frac{t_{\mathrm{CPU}}}{t_{\mathrm{GPU}}} \tag{8}$$

The specification of the computer used in the tests is shown in Table 1. The GPU and CPU used in the tests come from the same period of time. The CPU program was compiled using the GNU gcc compiler version 4.4.7 with the optimization parameter O2 and the architecture parameter native. The program for the CUDA device was compiled using the nvcc compiler from CUDA toolkit 4.1. The CUDA program was compiled with `-use_fast_math` and `-arch sm_20` switches. We used the CUDA 2.0 architecture as it provides better solutions in case of transfer between local memory and global memory. The grid of our calculations was set to `grid(256,1,1)`. This means that we had 256 blocks of threads. The number of threads per block (TPB) was set to 256. The total number of threads used in our calculations was declared by the following grid of blocks $256 \cdot 1 = 256$, each block containing 256 threads, what in summary gave 65 536 threads. In fact, not all these threads were still active in the data processing. The real number of active threads depends on the number of atom pairs in simulations. If the number of threads is greater than the number of atom pairs, then the thread finishes its work. The plot of the speedup test against a number of pairs or active threads N_t is shown in Figure 5. At the beginning the speedup rise is linear. The starting number of active threads is equal to 2016. This value is lower than the total number of threads per cores in NVIDIA GPU TESLA C2075. The speedup function reaches its first maximum at $N_t = 9180$. Next the value of the speedup factor begins to

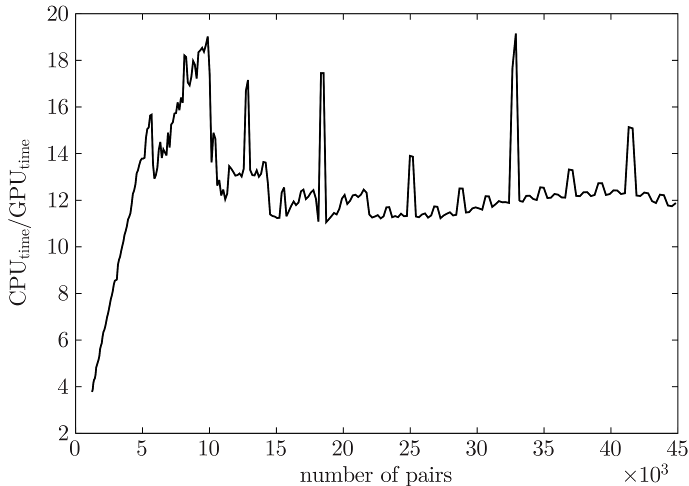

Figure 5. Speedup dependence of atomic pair numbers

oscillate and slowly decays to the average factor of 12. In this plot we can observe some spots where the GPU performance is substantially increased. The locations of peeks are correlated with the 4-byte data alignment in the GPU processor. The maxima of this plot show that calculations on the GPU are 19 times faster than on a single CPU core or 4.75 times faster than on a 4 core CPU unit. Another important parameter in the scientific calculation is the memory load. We made the memory load up test by increasing the number of the time origins included in the averaging $G_4(t)$ function, leaving the same number of threads in case of the GPU calculations. The number of constant threads used in our calculations of $G_4(t)$ function was equal to 32 640. We made the plot of the speedup factor against the memory load (Figure 6). The speedup factor decreases its value with an increase in the memory load. We noticed that the decrease was generally linear, but divided into two separate regions of different decrease acceleration. In the range from 50 up to 400 MB, the speedup factor decreases faster than in the region above 400 MB. The decrease in the speedup in the range of 50–1300 MB is equal to three times the CPU speed. Ultimately, the most important question is the quality of calculations of the GPU parallel algorithm in comparison to the CPU sequential algorithm. In our test, both calculations were performed with single precision floating point numbers. The appropriate four-body correlation functions were averaged over 15 000 time origins, up to 6 ps. The relative error between CPU and GPU calculations was calculated as the difference between the values of the $G_4(t)$ functions calculated on the CPU and the GPU. The plot of the relative error against the time shows small deviations that do not depend on the total simulation time (Figure 7). The maximal deviation of the relative error is within 0.1%. The average relative error value is equal to 0.039%. Most of the deviations in the floating point calculations come from different hardware representations of single precision floating point numbers.

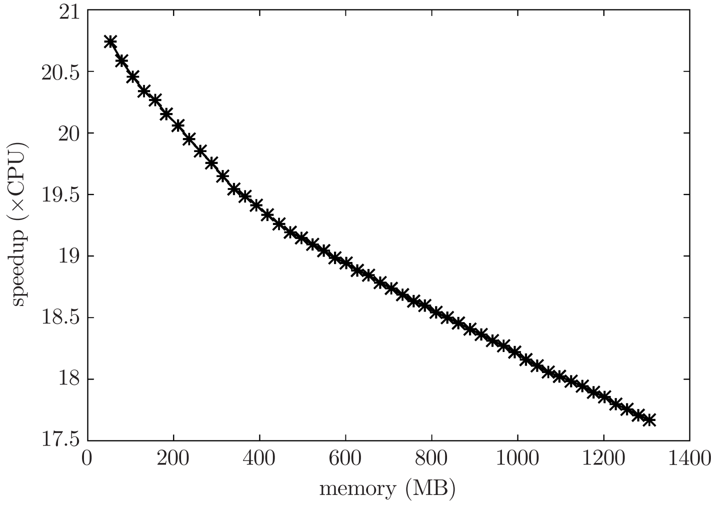

Figure 6. GPU speedup dependency on memory consumption

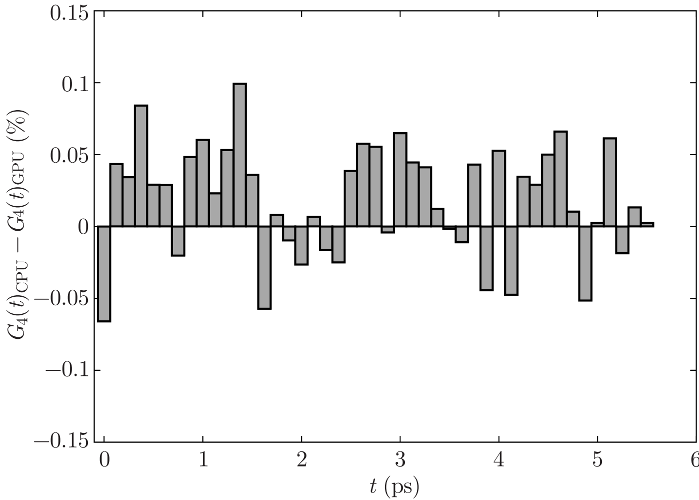

Figure 7. Time dependence of relative error in percentage

5. Conclusions and future work

A parallel algorithm of the MBCF calculations is a difficult task. As far as we know there are no publications about acceleration of many body correlation functions using a GPU. This work has shown that even for many body correlation functions we can create a parallel algorithm suitable to run about 12 times faster on the GPU than on a single core CPU, giving good results at the same time. We have also observed that at the peak performance of the GPU calculations are 19 times faster than CPU single core calculations. Our calculations show that the number of active threads is a very important parameter in the case of calculation

Table 1. The test computer configuration

processor	Intel® Core™ i7–950 Processor (8 M Cache, 3.06 GHz, 4.80 GT/s Intel® QPI)	NVIDIA Tesla C2075 GPU
launch date	Q2'09	Q3'11
frequency, graphics clock		575 MHz
frequency, processor clock	3.33 GHz	1.15 GHz
memory size	4096 MB (computer memory)	6144 MB
memory type	DDR3–800/1066	GDDR5
memory clock	1066 MHz	750 MHz
memory clock (effective)		3000 MHz
memory interface width:	64 bits	384-bit
memory bandwidth	25.6 GB/s	144 GB/s
# of cores	4	14 streaming multiprocessors (SM)32 CUDA cores/SM => total of 448 CUDA cores
# of threads	8	32 threads/core => total of 14 336 threads

acceleration on the GPU. The difference between peak performance and average speedup can be the subject of future research of better parallel algorithms of the $G_4(t)$ function for the GPU architecture.

Acknowledgements

The author wants to greatly acknowledge NVIDIA® Corporation for supporting with CUDA enabled GPU cards (Professor Partnership program).

References

[1] Oxtoby D W 1977 *Mol. Phys.* **34** 987 doi: 10.1080/00268977700102291
[2] Bisconti C, de Saavedra F A, Co' G and Fabrocini A 2006 *Phys. Rev. C.* **73** 54304 doi: 10.1103/PhysRevC.73.054304
[3] Yamaguchi T, Matsuoka T and Koda S 2007 *J. Chem. Phys.* **127** 234501 doi: 10.1063/1.2806289
[4] Kirkpatrick T R and Thirumalai D 1988 *Phys. Rev. A.* **37** 4439 doi: 10.1103/PhysRevA.37.4439
[5] Grzybowski A, Koperwas K, Kolodziejczyk K, Grzybowska K and Paluch M 2013 *J. Phys. Chem. Lett.* **4** 4273 doi: 10.1021/jz402060x
[6] Frommhold L 1994 *Collision-induced Absorption in Gases*, Cambridge University Press doi: 10.1017/CBO9780511524523
[7] Dawid A and Gburski Z 2002 *J. Mol. Struct.* **614** 183 doi: 10.1016/S0022-2860(02)00245-4
[8] Kosmider M, Dendzik Z, Palucha S and Gburski Z 2004 *J. Mol. Struct.* **704** 197 doi: 10.1016/j.molstruc.2004.02.050
[9] Martins M M and Tassen H 2003 *J. Chem. Phys.* **118** 5558 doi: 10.1063/1.1555632
[10] Dawid A and Gburski Z 1999 *J. Mol. Struct.* **482–483** 271 doi: 10.1016/S0022-2860(98)00668-1
[11] Dawid A and Gburski Z 2003 *J. Phys. Condens. Matter.* **15** 2399 doi: 10.1088/0953-8984/15/14/315

[12] Dendzik Z, Kosmider M, Dawid A and Gburski Z 2005 *J. Mol. Struct.* **744** 577 doi: 10.1016/j.molstruc.2004.12.049
[13] Skrzypek M and Gburski Z 2002 *Europhys. Lett.* **59** 305 doi: 10.1209/epl/i2002–00242–8
[14] Dawid A and Gburski Z 2017 *J. Mol. Liq.* **245** 71 doi: 10.1016/j.molliq.2017.06.040
[15] Raczyński P, Dawid A and Gburski Z 2006 *J. Mol. Struct.* **792** 212 doi: 10.1016/j.molstruc.2006.01.063
[16] Amani M, Amjad-Iranagh S, Golzar K, Sadeghi G M M and Modarress H 2014 *J. Membr. Sci.* **462** 28 doi: 10.1016/j.memsci.2014.03.018
[17] Raczyński P, Dawid A, Sokół M and Gburski Z 2007 *Biomol. Eng.* **24** 572 doi: 10.1016/j.bioeng.2007.08.010
[18] Kirk D B and Hwu W W 2010 *Programming Massively Parallel Processors: A Hands-on Approach, 1 edition*, Morgan Kaufmann
[19] Wilt N 2013 *CUDA Handbook: A Comprehensive Guide to GPU Programming, The 1st edition*, Addison-Wesley Professional
[20] Ruetsch G and Fatica M 2013 *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming, 1 edition*, Morgan Kaufmann
[21] Brodtkorb A R, Sætra M L and Altinakar M 2012 *Comput. Fluids.* **55** 1 doi: 10.1016/j.compfluid.2011.10.012
[22] Liu W, Schmidt B, Voss G and Müller-Wittig W 2008 *Comput. Phys. Commun.* **179** 634 doi: 10.1016/j.cpc.2008.05.008
[23] Rybakin B P 2013 *Comput. Fluids.* **80** 403 doi: 10.1016/j.compfluid.2012.01.016
[24] Zaspel P and Griebel M 2013 *Comput. Fluids.* **80** 356 doi: 10.1016/j.compfluid.2012.01.021
[25] Jacques R, Taylor R, Wong J and McNutt T 2010 *Comput. Methods Programs Biomed.* **98** 285 doi: 10.1016/j.cmpb.2009.07.004
[26] Goldsworthy M J 2014 *Comput. Fluids.* **94** 58 doi: 10.1016/j.compfluid.2014.01.033
[27] Komatsu K, Soga T, Egawa R, Takizawa H, Kobayashi H, Takahashi S, Sasaki D and Nakahashi K 2011 *Comput. Fluids.* **45** 122 doi: 10.1016/j.compfluid.2010.12.019
[28] Januszewski M and Kostur M 2010 *Comput. Phys. Commun.* **181** 183 doi: 10.1016/j.cpc.2009.09.009
[29] Ye Y and Li K 2013 *Comput. Fluids.* **88** 241 doi: 10.1016/j.compfluid.2013.08.005
[30] Dom Mínguez J, Crespo A J C and Gómez-Gesteira M 2013 *Comput. Phys. Commun.* **184** 617 doi: 10.1016/j.cpc.2012.10.015
[31] Januszewski M and Kostur M 2014 *Comput. Phys. Commun.* **185** 2350 doi: 10.1016/j.cpc.2014.04.018
[32] Liang S, Liu W and Yuan L *Comput. Fluids.* **99** 156 doi: 10.1016/j.compfluid.2014.04.021
[33] Dawid A, Raczyński P and Gburski Z 2014 *Mol. Phys.* **112** 1645 doi: 10.1080/00268976.2013.853111
[34] Allen M P and Tildesley D J 1989 *Computer Simulation of Liquids*, Oxford University Press
[35] CUDA C Best Practices Guide 2016 [online] `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3DOAuFZqI` (accessed September 11, 2016)
[36] Dawid A 2019 *GPU implementation of 4-body correlation function. Contribute to alex386/GPGPU-4-body-correlation-function development by creating an account on GitHub* [online] `https://github.com/alex386/GPGPU-4-body-correlation-function` (accessed January 25, 2019)
[37] Rapaport D C 2004 *The Art of Molecular Dynamics Simulation*, Cambridge University Press
[38] Dawid A 2017 *RIGid Molecular Dynamics (RIGMD) – simualation software, RIGid Mol. Dyn. RIGMD – Simualation Softw.* [online] `http://www.wsb.edu.pl/nauka-aleksander-dawid/RIGMD/rigmdUS.html`