GPU IMPLEMENTATION OF ATOMIC FLUID MD SIMULATION

Aleksander Dawid

Department of Transport and Computer Science,
The University of Dąbrowa Górnicza,
1c Cieplaka St., 41-300 Dąbrowa Górnicza, Poland

ABSTRACT

A computer simulation of an atomic fluid on a GPU was implemented using the CUDA architecture. It was shown that the programming model for efficient numerical computing applications was changing with the development of the CUDA architecture. The introduction of the L2 cache decreased the latency between the global GPU memory and the registers. The performed MD simulation using the global memory and registers showed that the average acceleration relative to the CPU reached 80 times for single-precision calculations. Usually, the shared block memory gives much better results for this kind of calculation. We have found that using the shared memory gives acceleration over 116 times in comparison to the CPU. It is about 49% faster than using the global memory and registers. It is shown here that the performance of generally available graphics cards for double-precision calculations is significantly lower than for single-precision calculations. The recorded double-precision acceleration relative to the CPU in our experiment averaged 6 and 7 times for the global and shared memory, respectively. We performed these calculations on two different CUDA enable device systems.

Keywords: MD simulation, GPU; atomic fluid, MD parallel algorithm

DOI: https://doi.org/10.17466/tq2022/26.1/b

1. Introduction

Fluid dynamics in closed tanks has been intensively researched in recent years due to its potential applications as nanoelectronic devices, nanoscale particle sensors [1] and gas storage devices [2,3]. The most frequently used tool for conducting preliminary research in this field are classical computer simulations using the molecular dynamics method. There are many studies investigating the properties of atomic fluids with this method in small clusters [4–6] and bulk atomic systems [7–9]. There are many studies in the field of computer simulations examining atomic fluids in narrow graphene or graphite gaps, where such an atomic fluid exhibits a two-dimensional structure. These solutions are used in the search for materials with low thermal permeability [10–12] and better lubricity [13]. Another area of research where computer simulations of atomic fluids are of particular importance due to the costs of real experiments are studies of the interaction of atomic fluids with carbon nanotubes. The movement of atoms inside a carbon nanotube is heavily limited by its geometry. In extreme cases, a one-dimensional array of atoms can form inside a nanotube. Investigating the dynamics of such a series of atoms, e.g. noble metals, may give an answer to whether such a series of atoms can be a good conductor of electricity. Currently, there are many publications on the interaction of thin atomic layers with carbon nanotubes [14–17]. The most promising in terms of applications for the construction of memory elements are studies on nanotubes filled with fullerenes with potassium ion inclusion. [18,19]. Such a system can be controlled by an electric or magnetic field by changing its state by shifting the doped fullerene, which corresponds to 1 or 0 in a binary system. Due to the small size of such systems, as well as the possibility of building structures in three dimensions, they can be widely used in the construction of capacious memory systems. The difficulty level of carrying out classical computer simulations increases with the number of atoms and their chemical bonds. Classically, i.e. using Newtonian dynamics, it requires the use of complex empirical potentials modeling the structure and dynamics of atomic systems. Currently, most computer simulations are performed using the so-called force field. A good example is the publication examining the properties of fullerenols in an aqueous solution. The atomic force field is sufficient to determine whether the fullerenol is hydrophilic or hydrophobic [20]. All these molecular dynamics simulations are computationally intensive. Accelerating them even twice on the GPU is already a big qualitative leap. Speed is especially important when simulation is to be interactive. What does it mean? Let us assume that we want to check how an atomic fluid will behave in a container with various obstacles. This kind of calculation is very expensive on CPU processing. We probably need a few hours to see what will happen in a few seconds. It is impossible to interact in real-time with the simulation. At high computing speeds, we can create an interactive simulation program where the user can change the obstacles in the container and observe in real-time how the fluid flows. This allows observing changes in the behavior of an atomic fluid on a computer screen. Such a solution can significantly speed up the design of tanks and installations for storing atomic gases in the liquid state. The graphics card is a device that helps in achieving the required computing speed for computer simulations. Many solutions in the form of parallel MD algorithms for the use on the GPU have already been developed [21–

24]. The purpose of this paper is to present a simple example of replacing the serial algorithm used previously on the CPU with the concurrent algorithm for the GPU. The code was written for the NVIDIA CUDA architecture [25]. All the algorithms presented in the paper are written in C++ and presented in the form of easy-to-use listings. This paper addresses the question of how to move from serial CPU computing to parallel computing in molecular dynamics simulations. Experimental simulations of the fluid of argon atoms show the advantage of calculations on modern graphics systems over calculations performed on the central processing unit.

2. Fitting the problem to the multi-threaded GPU model

The CUDA programming model requires the problem to be broken down into multiple threads, whose identifiers usually number a certain iteration of a given problem. If we look at the serial code of a typical MD simulation, we notice that the processing stream is concentrated around a single particle. The task of the simulation is to determine the position of a given particle in the next time step. This new position, in principle, will depend mainly on the force field that acts on this particle. If we now connect the thread with the dynamics of a single particle, we will get a system of dependent threads. This relationship will only manifest itself in the case of calculating the forces or accelerations acting on a given particle. Here, a global thread synchronization will be needed before refreshing the position of any particle. Microscopic quantities such as the position, velocity and, acceleration will be found in arrays of three-dimensional vectors allocated in the global memory.

2.1 Allocation of resources in the GPU

The example of fluid dynamics presented here assumes that the atoms are located at the nodes of a simple cubic (SC) lattice (Fig. 1). Such a network is characterized by three parameters defining the network size (Nx, Ny, Nz) and one parameter defining the distance from the nearest neighbors denoted as *d*. The last parameter is also referred to as the lattice constant in the case of the SC lattice. The constants N_x, N_y, and N_z define the number of times the network is duplicated in a given direction. The procedure that is used in this example assumes that the center of such a crystal will be at the origin of the coordinate system.

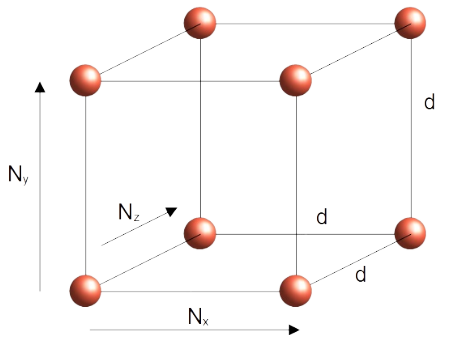

Figure 1. Simple cubic lattice.

In order to create such a crystal, we need an appropriate size table to store the positions of the atoms. The size of such an array is calculated as the product of all the three dimensions $N = N_x*N_y*N_z$, *where* $N_x = 1,2,\ldots,i$, $N_y = 1,2,\ldots,j$, $N_z = 1,2,\ldots,k$. The microscopic description of atoms requires the storage of data in the form of vector quantities of positions and its successive derivatives. We have created a special Vector class for this purpose with the property fields labeled *x*, *y*, and *z* with a floating-point type, it describes the three-dimensional (3D) Euclidean space. We have also defined in this class all operators that operate on 3D vectors. Now we can create an array of r = new Vector[N] objects in which the positions of the atoms will be stored. To build a simple cubic lattice, we can use an algorithm written in C++ :

Listing 1

```
void SC_lattice(double d, int Nx, int Ny, int Nz)
{
   for (int i = 0; i < Nx; i++)
      {
        for (int j = 0; j < Ny; j++)
         {
           for (int k = 0; k < Nz; k++)
            {
              int n = i * Nz * Ny + j * Nz + k;
              r[n].x = d * i;
              r[n].y = d * j;
              r[n].z = d * k;
            }
         }
      }
}
```

To perform MD simulations, we also need Vector object arrays that store velocities v = new Vector[N] and accelerations a = new Vector[N]. Equivalents of these tables must also be created for the GPU memory. Then, we must copy such data from the host to the device memory. We assume that the initial velocity and acceleration of the atoms are zero. We practically do not need to take any extra action to set the values here, because the the Vector constructor always sets the variables to zero. Now, it is sufficient to copy this zeroed v and a tables to the device memory. Now, the GPU buffers are ready to perform the calculations related to solving the equations of motion.

2.2 Kernel for concurrent computing

An algorithm that performs the MD simulation with the velocity Verlet method [26] requires three steps. We have enclosed all of these steps in one CUDA kernel routine. The input parameters for this routine are the positions, velocities, and accelerations of the atoms grouped in arrays of the structure (AOS) located in the global GPU memory. The first step of the routine does a half refresh of positions and velocities (Eq.1).

$$\begin{aligned} \vec{r}_i(t+1) &= \vec{r}_i(t) + \vec{v}_i(t)\,dt + \vec{a}_i(t)\,0.5\,dt^2 \\ \vec{v}_i(t+1/2) &= \vec{v}_i(t) + \vec{a}_i(t)\,0.5\,dt \end{aligned} \qquad (1)$$

The most important element of this procedure, in the case of CUDA programming, is the index i that enumerates atoms. This is the basis of choosing the number of threads involved in the calculation. In this work, we have attached one thread to one atom to solve its trajectory evolution. To start the calculations we need to copy the data from the global memory to local registers. The calculations themselves are performed on local registers, which gives the optimal time of their execution. The next procedure is to update the accelerations of each atom. We have to calculate the net force on each particle. In general, the problem is of O(n2) complexity. The critical part of the effective execution of MD simulations is the formula of the net force calculation (Eq. 2).

$$\vec{f}_i(t) = \sum_{j=1}^{N} \vec{f}\left(r_{ji}(t)\right) \qquad (2)$$

The simplest approach to this problem is to work directly on global tables from the level of the computational kernel. It is about reading directly from the GPU buffer. It is known from Newton's second law of motion that acceleration is proportional to the force acting between the atoms. Usually, we add the forces acting on one atom, but we can also add up the accelerations, especially when the masses of the individual atoms are the same. The inputs to this procedure are AOS describing positions. The output from this procedure is the acceleration of each atom. As one can see, the procedure loops through all n atoms in the system. We know that this can be simplified by using a truncation radius or a Verlet list [26]. Here, however, we give up these solutions for both the CPU and the GPU to show the full calculations. In this computational kernel, we only use fast local variables to perform calculations. The loop through all the atoms is broken into two parts to eliminate the condition that checks whether the atom sometimes does not interact with itself. We calculate the force as a negative potential gradient.

$$\vec{v}_i(t+1)=\vec{v}_i(t)+\vec{a}_i(t)\,0.5\,dt \qquad (3)$$

The basis for calculating the force in our work is the Lennard-Jones potential [27].

$$f(\vec{r}_{ji}(t))=-\nabla V(\vec{r}_{ij}(t)) \qquad (4)$$

We calculated the forces from the potential analytically, and then we applied it to the computational routine. In the CPU algorithm we took the general assumption from the 3rd Newton’s law that it reduces the iteration of the calculation by half. The same trick is hard to apply in the CUDA architecture. Precalculations of individual pairs will take extra time, especially if we need *(n-1)2/2* extra threads. In addition, two dimensional-arrays are needed to access these values. Our solution is more straight. Each thread calculates the net force acting on the i-th atom. We named this method GLOBAL. According to the Nvidia CUDA documentation [28], the global GPU memory is marked as the slowest. The overall performance of this memory improves in the new GPU architecture, the L2 cache memory. It works on the hardware level and does not need any extra actions from the programmer. Nevertheless, the good practice of the CUDA programming encourages us to use the block shared memory. This type of memory is much faster than the global memory but slower than registers. The main problem of this memory is its range, reduced to a single block. We cannot access the shared memory from another block. The size of this memory is usually between 32 and 64 kB. The block size is expressed by the number of threads that it can contain. The flow diagram of the algorithm that calculates the net force on *i*-th atom using the shared memory is shown in Figure 2. We named this version of the algorithm SHARED. We start processing by filling up the shared memory by the portion of positions data from the global memory equal to the block size. After this step, we have to synchronize all calculation threads before we go to the next step. The synchronization prevents the calculation of an old set of data. Next, we can calculate a partial force acting on the *i*-th atom. After that step, we must again synchronize the threads to obtain correct results. We repeat the process of copying data from the global memory and calculating the partial net force until the data runs out. The procedure is based on the assumption that, despite the limited cache, the forces in the chunks that fit in this cache can be added up. The last step in Verlet’s algorithm is to update the velocities in the *t+1* time step.

$$V(\vec{r}_{ij})=4\epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12}-\left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \qquad (5)$$

The formula for updating velocities (Eq. 5) is exactly the same as the formula from Equation 1. Now, we can go to the detailed CUDA implementation of the algorithms and discuss the bottlenecks of these solutions.

We start our implementation with the acceleration calculations. In Listing 2 we have the procedure with the shared memory. The *extern __shared__ Vector rPos[];* statement specifies a buffer in the shared memory the size of which is determined before the given compute kernel is started.

Listing 2

```
1. __global__ void Accel(Vector *R, Vector *A)
2. {
3.          extern __shared__ Vector rPos[];
4.
5.           unsigned int gid = threadIdx.x + blockDim.x*blockIdx.x;
6.          if (gid>gN) return;
7.
8.          int i, portion;
9.          Vector mPos,f;
10.         mPos = R[gid];
11.
12.         for (i = 0, portion = 0; i < gN; i += blockDim.x, portion++)
13.         {
14.             int bli = porcja * blockDim.x;
15.             int idx = bli + threadIdx.x;
16.             rPos[threadIdx.x] = R[idx]; //copy to block
17.             __syncthreads();
18.             f = CalcForcePart(bli, gid, mPos, f);
19.             __syncthreads();
20.         }
21.         A[gid] = f;
22. }
```

The for loop in this procedure is limited by the number of atoms but also changes by the entire block size because the force is calculated for the entire block (Listing 2). The condition inside the block loop prevents the atom from interacting with itself. The same procedure as before is used to calculate the forces acting on the selected atom. Variables beginning with g in the procedures described above represent variables of type __constant__. They are used to store constant information that does not change during the simulation.

Listing 3

```
1. inline __device__ Vector CalcForcePart(int bli, unsigned int gbli,
Vector r, Vector a)
2. {
3.          int i;
4.          extern __shared__ Vector rPos[];
5.          for (i = 0; i<blockDim.x; i++) {
6.              if (bli + i != gbli) {
7.                  a = LJ_Pairs(r, rPos[i], a);
8.              }
9.          }
10.         return a;
11. }
```

The main intensive algebraic operation is present in the function *LJ_Pairs()*. If we look at the code below, we can find the most computational expensive line of this code: Yes, it is reciprocal of the distance square calculations in line 6. If we replace this line with a well-optimized CUDA library function *Rijs2=__fdivdef(1.0f,Rij2)* we can speed up our calculation significantly. If we eliminate a mix of single-precision and double-precision numbers we can again speed up our calculations. For example, number 2.0 is treated internally as a

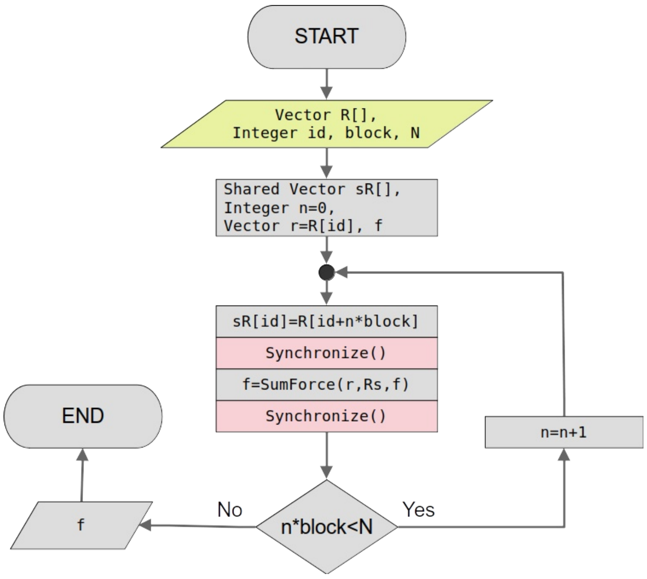

Figure 2. Single threaded force calculation using the shared memory.

double-precision number.

Listing 4

```
inline __device__ Vector LJ_Pairs(Vector ri, Vector rj, Vector a) {
        real Rij2, Rijs2, Rijap, Rijrp, PZ;
        Vector Rij, Fij;
         Rij = ri - rj;
        Rij2 = Rij.x*Rij.x + Rij.y*Rij.y + Rij.z*Rij.z;
        Rijs2 = 1.0 / Rij2;
        Rijap = Rijs2*Rijs2*Rijs2*g_Sigma6;
        Rijrp = 2.0 * Rijap*Rijap;
        PZ = g_ALJ*(Rijrp - Rijap)*Rijs2;
        Fij = Rij*PZ;
        a += Fij;
        return a;
}
```

How can we start the MD simulation using the computational kernels? In the beginning, two important parameters must be defined. First, the size of the block, i.e. how many threads the block will contain. In the case of the reference device used in this work, it is best to choose the number of threads equal to the number of threads processed at a time in a single multiprocessor. We set the number of threads in the block in variable *threadsPerBlock*. Now we need to determine how many blocks we need to accommodate this given *N* number of atoms. The algorithm for calculating the number of blocks based on the variables N and threadsPerBlock will look as follows.

Listing 5

```
 test = N % threadsPerBlock;
if (test == 0) {
   blocksPerGrid = N / threadsPerBlock;
}
else {
   blocksPerGrid = N / threadsPerBlock + 1;
}
```

After setting the variables *threadsPerBlock* and *blocksPerGrid*, we can call computational kernels for molecular dynamics simulation. First, let us see what the computation will look like without shared memory declaration:

Listing 6

```
for(unsigned int step=0; step<max_step; step++){
    VerletHalf<<<blocksPerGrid, threadsPerBlock>>>(dev_r, dev_v, dev_a);
    GlobalAccel <<<blocksPerGrid, threadsPerBlock>>>(dev_r, dev_a);
    VerletAll <<<blocksPerGrid, threadsPerBlock >>>(dev_v, dev_a);
}
```

To use the shared memory in a block, we need to define its size. It will be related to the data type that we will use in this memory. In the case of simulation, this is the position information in the form of a Vector class, so we can determine the cache size from the number of threads in the block as follows: *sharedSize = threadsPerBlock * sizeof(Vector)*;

If we take a maximum block size value to be 1024, the shared memory size will be 1024 * 24 = 24576 bytes. This is approximately 39% of the maximum memory per block for the reference device. The execution of the procedures for the simulation using shared memory will be as follows:

Listing 7

```
for(unsigned int step=0; step<max_step; step++){
    VerletHalf <<<blocksPerGrid, threadsPerBlock>>>(dev_r, dev_v, dev_a);
    Accel <<<blocksPerGrid, threadsPerBlock, sharedSize>>>(dev_r, dev_a);
    VerletAll <<<blocksPerGrid, threadsPerBlock>>>(dev_v, dev_a);
}
```

In a modern CUDA architecture, we can use streams technology. We can run not one kernel but several concurrent kernels. For example, if we want to use four streams, then our kernel will look as in Listing 8. The variables *cuda1*, *cuda2*, *cuda3*, *cuda4* are the handlers to the CUDA streams:

Listing 8

```
int blocks= blocksPerGrid/4;
int chunk =  blocks*threadsPerBlock;
for(unsigned int step=0; step<max_step; step++){
    MD<<<blocks, threadsPerBlock, sharedSize, cuda1>>>(R, V, A, 0);
    MD<<<blocks, threadsPerBlock, sharedSize, cuda2>>>(R, V, A, chunk);
    MD<<<blocks, threadsPerBlock, sharedSize, cuda3>>>(R, V, A, 2*chunk);
    MD<<<blocks, threadsPerBlock, sharedSize, cuda4>>>(R, V, A, 3*chunk);
}
```

2.4 CPU vs GPU precision of calculation

After the MD simulation in the GPU is finished, we need to assert that the result is correct. Before copying the result, we need to make sure whether any other threads in the device are not performing the calculations. We have to synchronize the threads again. As the control parameters, we use the velocities of all particles. We have to copy the results from the GPU to the host memory. Copying is performed to a new table in order to avoid losing data from the simulation performed on the main processor. There are a number of ways to compare the results between the CPU and the GPU. The solution used here is to determine the sum of the absolute values from the difference between the speed vectors calculated in the GPU and the CPU (Eq. 6).

$$s=\sum_{i=0}^{N}\left|\overrightarrow{v_{GPU}}-\overrightarrow{v_{CPU}}\right| \qquad (6)$$

3. Results

Graphics cards are mainly focused on quick calculations related to the processing of polygon graphics. The accuracy of these calculations is often of secondary importance. The only thing that matters is that the objects are properly transformed into 3D spaces and that there are no gross irregularities in their matching. The race to increase the frame rate in computer games has recently resulted in the introduction of half-precision calculations, where a floating-point number is described on 2 bytes. However, in serious scientific applications, even a 4-byte description of a floating-point number is not enough. These are the topics of computer simulations of molecular systems. Let us illustrate this with an example. Two atoms come closer to each other. There is no such thing as an infinitesimally small shift in computer calculations using floating-point numbers. The precision of floating-point numbers is significant in MD simulations. If the calculations are not too precise, they may cause an increase in the kinetic energy of the system, although no external forces act on the system. The research done here focuses on determining the GPU acceleration provided by both single and double-precision calculations.

In our experiment, the calculations were made on two different GPU devices. The first unit is an HP Omen laptop with the CPU: Intel i5-6300HQ clocked with a clock frequency of 2.3 GHz and the GPU: Nvidia Geforce GTX 960M (processor GM-107A) clocked with a clock frequency of 1.176 GHz. This GPU is for mobile computers. The second unit is Jetson Nano with the CPU: Quad-core ARM Cortex-A57 MPCore processor and the GPU: NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores. This class of the devices belongs to single-board computers. Our simulations were performed on Linux UBUNTU 18.04 systems. We compiled our source code with the CUDA toolkit 11.4 (HP Omen) and CUDA toolkit 10.3 (Jetson Nano). In both cases, we switched on the O2 level optimization for the CPU and the GPU. We utilized switches for the CUDA 5.0 (HP Omen) and CUDA 5.3 (Jetson Nano) architectures. In our calculations, the number of atoms was the power of two. We performed the calculations for lattices of *N* from

2^{10} to 2^{19}. We first looked at accelerating single-precision calculations (Fig. 3). The calculations were performed using one stream with the force calculations in the shared memory.

These calculations were performed for one MD simulation step with different block size values. The MD step time is equal $dt = 10^{-15}$ s. Figure 3 shows the acceleration saturation for the number of atoms greater than 80,000. The number of atoms in the system directly corresponds to the number of threads involved in the calculation.

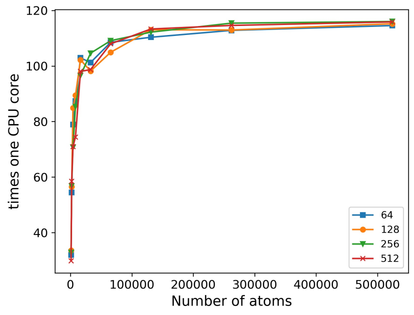

Figure 3. Dependence of acceleration of single-precision computation on the number of atoms in the system.

With low numbers of atoms, from a few to several thousand, the speed of computation increases very quickly. After that, this increase continues, but it is much slower. In this first period, the threads usually fit increasingly better with the graphics processor. After the processor is saturated with threads, they must be queued for further execution. However, floating-point calculations are so fast in the graphics system that we can still see an increase in performance. There are also small differences in performance for different block sizes. The maximum acceleration achieved in this case was 116 times compared to the single CPU core. Let us see now how the introduction of the shared memory improves the MD calculations in single precision. In Figure 4 we can see the difference in performance when using global and shared memory kernels. If we look closer at this plot, we can see the maximum of global memory calculations for the number of atoms equal to 65536 (2^{16})). It is true for block sizes equal to both 128 and 256 threads. We cannot observe any maxima at this point for shared memory calculations. In the case of shared memory calculations, the plot of 256 threads increases smoothly until saturation at 262144 simulated atoms. The situation is a bit different in the case of double-precision calculations (Fig. 5). The accelerations of a double-precision calculation are smaller than accelerations of single-precision. It is only on the order of 7.0 times compared to the one CPU core. It means that for all processor cores, it can give only a percentage advantage of the GPU. Moreover, this speed-up is achieved only above 50,000 atoms. Faster saturation may indicate inferior handling of double-precision floating-point numbers. Using direct global memory in the calculation does not fix the slow speed-up of calculations (Fig. 6).

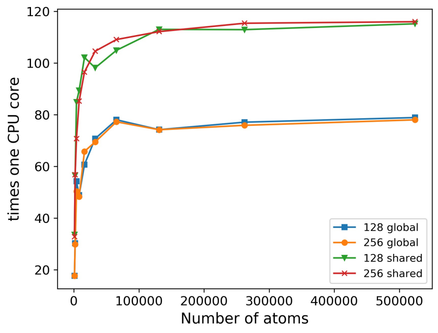

Figure 4. Comparison of global and shared memory single precision MD calculations.

We see that use of the global memory is slower than using the shared memory. In the plot for both algorithms, we can observe a peak at 65536 atoms for a block size equal to 256 threads. In Table 1, we can see the times of calculation on two GPU devices. The table label "Global" means a global memory algorithm, and the label "Shared" means a shared-memory algorithm. The integer after the label named "Shared" means the number of streams in the CUDA technology.

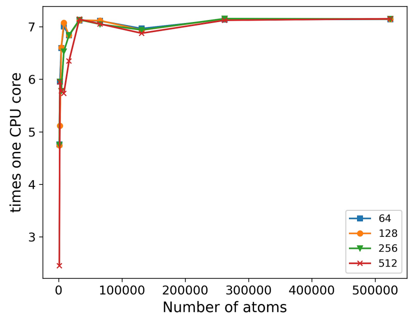

Figure 5. Dependence of acceleration of double-precision computation in relation to one CPU core on the number of atoms in the system.

The ratio global to the shared memory single-precision calculation is equal to 1.49 for GTX960M and 1.62 for Jetson Nano. Both GPUs are of the same "Maxwell" architecture. GTX960M uses CUDA 5.0 and Jetson Nano CUDA 5.3. It seems that the Jetson Nano GPU is less cached. Introducing more streams to the calculations does not affect the MD simulation acceleration. Based on these results, we can see that the procedure that uses the shared memory is faster than the procedures that use the global memory. The graphics card used for testing is a card typically designed for polygon processing. GPUs used in computing centers can perform both single and double-precision floating-point calculations with equal

the service "TechPowerUp" [29] RTX3090 is over 10 times faster than GTX960M. Unfortunately, we have no access to this graphic card to test the speed-up of our MD simulation.

Another issue when it comes to concurrent computing is its accuracy and how much it differs between the CPU and the GPU. The molecular dynamics method is based on solving the equations of motion. It allows us to follow the trajectory of a single particle. If we assume that we are dealing with a finite precision of calculations, then this trajectory is one of many possibilities. An example are the calculations performed here.

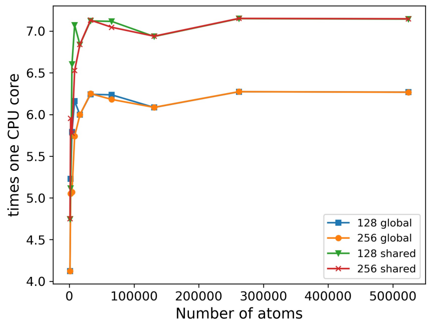

Figure 6. Comparison of global and shared memory double precision MD calculations.

A slight difference in the precision of the calculations may result in a completely different position of the particle after millions of steps. Does this mean that one trajectory is true

Device	Precision	Global [s]	Shared 1	Shared 4	Shared 8
GTX960M	float	21.45	14.31	14.40	14.46
	double	244.40	214.28	214.27	214.26
Jetson nano	float	119.27	73.36	73.21	73.21
	double	1579.47	1388.18	1389.10	1389.04

Table 1. Calculation times. The data was taken from our MD simulation with the 256 thread block size.

speed. However, it was more about showing acceleration on popular and cheap to buy equipment. On the other hand, the thermal design power (TDP) of GTX960M is 75 W, and Jetson nano is 10 W. The top gaming card RTX3090 has a TDP equal to 350 W. According to

and the other is not? It all depends on the numerical libraries and the equipment on which the calculations will be performed. It turns out that despite the differences in the positions of individual particles, macroscopic values, such as pressure and temperature, are the same in both systems. However, we will be interested in the microscopic differences between CPU and GPU computing. Figure 7 shows the relationship between the relative error of the GPU and the CPU calculated based on the final values of the velocity vector in one step and the number of atoms. It is generally an increasing linear relationship. In single-precision calculations (Fig. 7) the error values are 9 orders higher than for double-precision calculations (Fig. 8). The differences may be due to the rounding applied to the floating-point number in the CPU and the GPU.

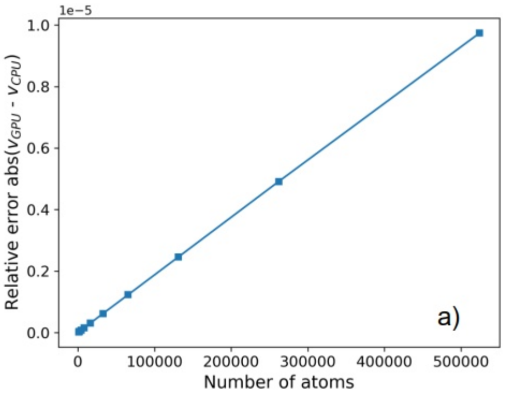

Figure 7. Dependence of the relative velocity error on the number of atoms for operations on single precision numbers.

4. Conclusions

The aim of this study is to indicate the growing role of concurrent programming in the construction of applications based on computer simulations using the molecular dynamics method. The current direction of processor development can be characterized by one slogan -“multi-core”. The leader in introducing the multi-core technology are graphics processors, where we are dealing with thousands of cores. On the other hand, properly written applications are still needed to use these numbers of cores. This study is intended to help create concurrent software using well-known computer simulation algorithms. This work gives us a brief overview of the techniques used in molecular dynamics simulations to increase their efficiency. The results presented in this paper show the acceleration of MD simulation calculations on the GPU.

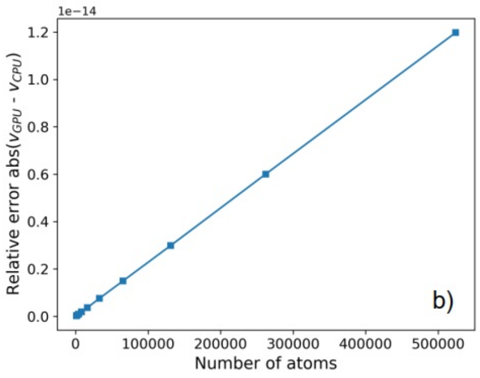

Figure 8. Dependence of the relative velocity error on the number of atoms for operations on double precision numbers.

For operations carried out on single-precision numbers, the acceleration reached 116 times. The result for double precision numbers, 7.1 times, may be a bit disappointing compared with the previous results. It must be remembered that these calculations were performed on the mobile graphics card version, where its full potential was not unleashed due to the power consumption.

References

1. Huang B, Li Z, Liu Z, Zhou G, Hao S, Wu J, Gu B L, Duan W 2008 *Adsorption of Gas Molecules on Graphene Nanoribbons and Its Implication for Nanoscale Molecule Sensor* J. Phys. Chem. C. **112** pp. 13442-13446

2. Zhang F, Yang F, Huang J, Sumpter B G and Qiao R 2016 *Thermodynamics and Kinetics of Gas Storage in Porous Liquids* J Phys. Chem B **120** pp. 7195-7200

3. Dawid A, Raczyński P and Gburski Z 2014

Depolarised Rayleigh light scattering in argon layer confined between graphite plains: MD simulation Mol. Phys. pp. 1-6

4. Dawid A and Gburski Z 1997 *Dynamical properties of the argon-krypton clusters: molecular dynamics calculations* J Mol. Struct. **410** pp. 507-511

5. Blaisten-Barojas E and D. Levesque 1986 *Molecular-dynamics simulation of silicon clusters* Phys Rev. B. **34** pp. 3910-3916

6. Dawid A and Gburski Z 1998 *Interaction-induced absorption in argon-krypton mixture clusters: Molecular-dynamics study* Phys Rev. A. **58** pp. 740-743

7. Sokol M, Dawid A, Dendzik Z and Gburski Z 2004 *Structure and dynamics of water - molecular dynamics study* J Mol. Struct. **704** pp. 341-345.

8. Akbarzadeh H, Abroshan H, Taherkhani F, Izanloo C and Parsafar G 2011 A *Size dependence and effect of potential parameters on properties of nano-cavities in liquid xenon using molecular dynamics simulation* Chem Phys. **381** pp. 44-48

9. Devarajan D, Liang L, Gu B, Brooks S C, Parks J M and Smith J C 2022 *Molecular Dynamics Simulation of the Structures, Dynamics, and Aggregation of Dissolved Organic Matter* Environ Sci. Technol. **54** pp. 13527-13537

10. Dawid A and Gburski 2017 Z *Molecular dynamics simulation of collision-induced absorption spectra of neon-krypton mixture thin layer confined between graphite walls* J Mol. Liq 2017 **245** pp. 85-90

11. Dawid A and Gburski Z 2017 *Interaction-induced light scattering in thin neon film confined between graphite slabs: MD study* J Mol. Liq **245** pp. 71-75

12. Dawid A and Gburski Z 2017 *Structural and Dynamical Properties of Argon-Krypton Binary Mixture Confined Between Graphite Slabs: Molecular Dynamics Simulation* Interface Stud. Appl **195**

13. Piatek A, Dawid A and Gburski Z 2006 *The existence of a plastic phase and a solid–liquid dynamical bistability region in small fullerene cluster (C60)7: molecular dynamics simulation* J Phys. Condens. Matter **18** pp. 8471.

14. Dawid A and Gburski 2007 Z *Dielectric relaxation of 4-cyano-4-n-pentylbiphenyl (5CB) thin layer adsorbed on carbon nanotube - MD simulation* J Non-Cryst. Solids **353** pp. 4339-4343.

15. Dawid A and Gwizdała W 2009 *Dynamical and structural properties of 4-cyano-4-n-pentylbiphenyl (5CB) molecules adsorbed on carbon nanotubes of different chiralities: Computer simulation* J Non-Cryst. Solids **355** pp. 1302-1306.

16. Raczyński P, Dawid A, Piętek A and Gburski Z 2006 *Reorienatational dynamics of cholesterol molecules in thin film surrounded carbon nanotube: Molecular dynamics simulations* J Mol. Struct. **792** pp. 216-220.

17. Raczyński P, Dawid A, Sokoł M and Gburski Z 2007 *The influence of the carbon nanotube on the structural and dynamical properties of cholesterol cluster* Biomol Eng. **24** pp. 572-576.

18. Dawid A, Piątek A, Sokoł M and Gburski Z 2008 *Dynamical properties of potassium ion K+ trapped in a fullerene C60 cage: An MD simulation* J Non-Cryst. Solids **354** pp. 4296-4299.

19. Dawid A and Gorny K 2007 *Dynamics of Endohedral Fullerene K+@C60 inside single walled carbon nanotube: MD simulation* **10**

20. Dawid A, Gorny K and Gburski Z 2011 *The structural studies of fullerenol C60(OH)24 and nitric oxide mixture in water solvent – MD simulation* Nitric Oxide **25** pp. 373-380.

21. Liu W, Schmidt B, Voss G and Muller-Wittig W 2008 *Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA* Comput Phys. Commun. **179** pp. 634-641.

22. Kondratyuk N, Nikolskiy V, Pavlov D and Stegailov V 2021 *GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP* Int J. High Perform. Comput. Appl. **35** pp. 312-324.

23. Dawid A 2019 *GPU-Based Parallel Algorithm of Interaction Induced Light Scattering Simulations in Fluids* TASK Q. **23** 5-17
24. Dawid A 2020 *GPU Implementation of the Parallel Ising Model Algorithm Using Object-Oriented Programming* Springer International Publishing
25. NVIDIA 2013 *CUDA Toolkit*
26. Rapaport D C 2004 *The Art of Molecular Dynamics Simulation* Cambridge University Press
27. Frenkel D and Smit B 2001 *Understanding Molecular Simulation From Algorithms to Applications* Elsevier
28. Cuda C++ Best Practices Guide *http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. 2021*
29. TechPowerUp *Nvidia GeForce Gtx 960m Specs https://www.techpowerup.com/gpu-specs/*geforce-gtx-960m.c2635

Conflicts of interests

The author(s) declare(s) that there is no conflict of interest.

Aleksander Dawid Received his M.Sc and Ph.D. degrees in a computer simulation in molecular physics at the University of Silesia, Poland, in 1995 and 2000, respectively. Worked at the University until 2017. In the same year, joined the computer science department of the WSB University in Dąbrowa Górnicza. Currently, Professor at the WSB University. His research interests include molecular dynamicsimulation, molecular physics and chemistry, programming, computational intelligence, parallel processing, machine learning, signal processing, and brain-computer interface. Published over 50 articles in refereed journals in the areas of computational physics, algorithms, and signal processing.