High Performance Computing for Science and Engineering I
Autumn semester 2013

Prof. Dr. P. Koumoutsakos
ETH Zentrum, CLT F12
CH-8092 Zürich

Project 1

Issued: October 1, 2013
Hand in: October 29, 2013

The N-body problem

The N-body problem consists of computing the motion of N interacting objects/bodies. It is fundamental to several applications in Computational Science, ranging from Electrostatics and Fluid Dynamics to Gravitational Dynamics and Chemistry.

Formulation

In this exercise we consider the motion of n bodies (e.g. planets and molecules), represented as point masses $m_1, m_2, ..., m_n$, that interact with each other via central forces. The associated positions of the point masses are $\mathbf{q}_1(t), ..., \mathbf{q}_n(t)$. The values $\mathbf{q}_1(0), ..., \mathbf{q}_n(0)$ and $\dot{\mathbf{q}}_1(0), ..., \dot{\mathbf{q}}_n(0)$ represent the initial condition of the N-body problem.

The motion of $\mathbf{q}_j$ is described with a second-order Ordinary Differential Equation (ODE):

$$m_j \ddot{\mathbf{q}}_j = \sum_{k \neq j} \mathbf{F}(\mathbf{q}_k, \mathbf{q}_j), \tag{1}$$

By introducing the variables $\mathbf{x}_j = \mathbf{q}_j$ and $\mathbf{v}_j = \dot{\mathbf{q}}_j$, we can convert the second-order ODE into an equivalent system of first-order ODEs:

$$\begin{pmatrix} \dot{\mathbf{x}}_j \\ m_j \dot{\mathbf{v}}_j \end{pmatrix} = \begin{pmatrix} \mathbf{v}_j \\ \sum_{k \neq j} \mathbf{F}(\mathbf{x}_k, \mathbf{x}_j) \end{pmatrix}. \tag{2}$$

Numerical integration

The numerical integration of the particle motion depends on the integration time step δt, that here for simplicity is considered as a pre-specified constant.

The discretized solution of the N-body problem can be expressed as:

$$\mathbf{x}_j^i = \mathbf{q}_j(i\ \delta t), \tag{3}$$

$$\mathbf{v}_j^i = \dot{\mathbf{q}}_j(i\ \delta t). \tag{4}$$

At any instant $t = i\ \delta t$, the acceleration of the particle j can be recovered as follows:

$$\begin{aligned} \mathbf{a}_j^i &= \ddot{\mathbf{q}}_j(i\ \delta t) && (5)\\ &= \frac{1}{m_j}\sum_{k\neq j}\mathbf{F}(\mathbf{x}_k,\mathbf{x}_j). && (6) \end{aligned}$$

The motion of the bodies can be computed by numerically integrating Equation 2; this can be done in several ways which trade accuracy for computational speed. In this exercise we employ the Velocity-Verlet scheme.

In the Velocity-Verlet scheme the bodies position and velocity at time $t = (i+1)\ \delta t$ from $t = i\ \delta t$ are computed as follows:

$$\begin{aligned} \mathbf{x}_j^{i+1} &:= \mathbf{x}_j^i + \delta t\ \mathbf{v}_j^i + \frac{1}{2}\delta t^2 \mathbf{a}_j^i, && (7)\\ \mathbf{v}_j^{i+1} &:= \mathbf{v}_j^i + \frac{\delta t}{2}\left(\mathbf{a}_j^{i+1} + \mathbf{a}_j^i\right). && (8) \end{aligned}$$

Molecular Dynamics

An example of N-body problem comes from chemistry and in particular from Molecular Dynamics.

A simple force used in this field is the Lennard-Jones force:

$$\mathbf{F}_{LJ}(\mathbf{r}) = -24\varepsilon\left[2\left(\frac{\sigma}{|\mathbf{r}|}\right)^{12} - \left(\frac{\sigma}{|\mathbf{r}|}\right)^{6}\right]\frac{\mathbf{r}}{|\mathbf{r}|^2}, \tag{9}$$

where ε is the depth of the potential well, σ is the minimum distance for which the potential between two particles is zero and $\mathbf{r}$ is the distance vector. The total force on a particle j would then be $\sum_{k\neq j}\mathbf{F}_{LJ}(\mathbf{x}_k - \mathbf{x}_j)$.

As can be seen from Figure 1, this force has a very rapid decay and is thus considered a short range interaction.

To decrease the computational costs, a truncated Lennard-Jones potential is introduced. The idea is to set the potential to exactly zero for $|\mathbf{r}| > r_{cut}$, where r_{cut} is the cutoff distance. If the cutoff is sufficiently large (r_{cut} commonly used is $2.5 \cdot \sigma$), the introduced error will be very small. Therefore the force now must be calculated only for the particle pairs, that are within the cutoff range.

Note that care should be taken to avoid jump discontinuity in the potential, which results in infinite force. So the additional shift is introduced in the potential such that it becomes 0 at r_{cut}:

$$U_{LJ_{cut}}(r) = \begin{cases} U_{LJ}(r) - U_{LJ}(r_{cut}), & \text{for } r < r_{cut}\\ 0, & \text{for } r \geqslant r_{cut} \end{cases} \tag{10}$$

Here $U_{LJ} = 4\varepsilon\left(\frac{\sigma^{12}}{r^{12}} - \frac{\sigma^6}{r^6}\right)$ is the original Lennard-Jones potential. It is clear that the force derived from the modified potential remains the same as in Equation 9 within the cutoff.

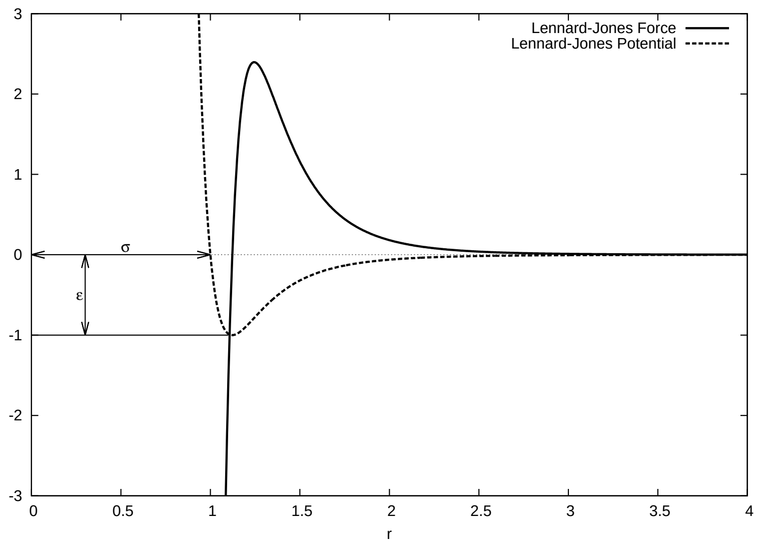

Figure 1: The Lennard-Jones potential and force ($\sigma = 1$, $\varepsilon = 1$).

Question 1: A-Priori Performance Analysis

In Equations (7-8) we identify two kernels, *i.e.* two distinct computational operations:

K1: Given a scalar value α and two vectors $\mathbf{y}, \mathbf{z}$ we perform $y_i := y_i + \alpha\ z_i$.

K2: Given the scalar value β, the velocity vector $\mathbf{v}$ and position vector $\mathbf{x}$, we compute all the accelerations $\mathbf{a}$ and update $\mathbf{v}$. Specifically, we compute: $v_i := v_i + \beta\ a_i(\mathbf{x})$

a) In pseudo code, outline the general N-body algorithm according to the discussed discretization. The computational parts of the algorithm should rely on K1 and K2.

Listing 1 shows the pseudocode for the n-body algorithm. Note that the accelerations are computed in the kernel K2.

```
// first step:
t = 0;
K2(x, v, deltat/2);
K1(x, v, deltat);
K2(x, v, deltat);
// note that we will have x(t) and v(t+deltat/2) after each
  iteration
t += deltat;
// remaining steps:
while(t < tmax)
{
    K1(x, v, deltat);
    K2(x, v, deltat);
    t += deltat;
}
```

Listing 1: Pseudo code of the n-body algorithm.

If we are interested in having position and velocity quantities at the end of each time step, the pseudocode shown in Listing 2 shows an alternative where the velocity is synchronized with the position information.

```
// first step:
t = 0;
K2(x, v, deltat/2);
K1(x, v, deltat);
K2(x, v, deltat/2);
t += deltat;
// remaining steps:
while(t < tmax)
{
    K2(x, v, deltat/2);
    K1(x, v, deltat);
    K2(x, v, deltat/2);
    t += deltat;
}
```

Listing 2: Pseudo code of the n-body algorithm with synchronized position and velocity information.

b) The following quantities determine the performance of a kernel:

- the total amount (T) of transfers (read and writes in number of real components)
- total number (C) of floating point operations (FLOP)
- the amount of memory (M) required to store the data for the problem.

Compute such quantities for K1 and K2 for n bodies in 3D as well as for the Lennard-Jones force on a single particle.

K1. We assume α to be a *literal*, i.e. we do not need to store them as their values will be substituted in the code. If we assume that there is no caching and that we work with `float` values, the quantities associated with K1 are:

$$\begin{aligned}
T_1 &= 3n \cdot (1 \text{ read } z_i + 1 \text{ read } y_i + 1 \text{ write } y_i) = 9n \text{ [Transfer]},\\
C_1 &= 3n \cdot (1 \text{ multiply} + 1 \text{ add}) = 6n \text{ [FLOP]},\\
M_1 &= (3n\ z \text{ values} + 3n\ y \text{ values}) \cdot 4 = 24n \text{ [Bytes]}.
\end{aligned}$$

K2. We assume β, the mass m, ϵ and σ to be a *literal*. Furthermore we assume that no caching occurs, that the local values are stored into registers and we base our computation on the optimized algorithm for K2 shown in Listing 3.

```
// K2 for particle i
c1 =  − 24 * beta * epsilon / m; // 4 FLOP
f = 0;
for (int j = 0; j < N; j++)
{
    if (i != j)
    {
        dx = p[j].x − p[i].x; // 1 FLOP, 2 reads
        dy = p[j].y − p[i].y; // 1 FLOP, 2 reads
        dz = p[j].z − p[i].z; // 1 FLOP, 2 reads

        IrI2 = dx * dx + dy * dy + dz * dz; // 5 FLOP

        invIrI2 = 1 / IrI2; // 1 FLOP
        sr2 = sigma * sigma * invIrI2; // 2 FLOP
        sr6 = sr2 * sr2 * sr2; // 2 FLOP

        c2 = (2 * sr6 * sr6 − sr6) * invIrI2; // 4 FLOP

        fx = fx + dx * c2; // 2 FLOP
        fy = fy + dy * c2; // 2 FLOP
        fz = fz + dz * c2; // 2 FLOP
    }
}

v[i].x = v[i].x + c1 * fx; // 2 FLOP, 1 read, 1 write
v[i].y = v[i].y + c1 * fy; // 2 FLOP, 1 read, 1 write
v[i].z = v[i].z + c1 * fz; // 2 FLOP, 1 read, 1 write
```

Listing 3: Pseudocode for the optimized K2 kernel.

Under these assumptions we compute the following values for T_2, C_2 and M_2:

$$\begin{aligned} T_2 &= n \cdot (3 \text{ read } v_i + 3 \text{ write } v_i + (n-1) \cdot (3 \text{ read } p_j + 3 \text{ read } p_j)) = 6n^2 \text{ [Transfer]}, \\ C_2 &= n \cdot (10 + (n-1) \cdot 23) = 23n^2 - 13n \text{ [FLOP]}, \\ M_2 &= (3n \text{ values } v + 3n \text{ values } p) \cdot 4 = 24n \text{ [Bytes]}. \end{aligned}$$

Question 2: N-Body Problem on CPU

Throughout this project we will consider a computational cubic domain $[-1, 1[^3$ with periodic boundary conditions.

In order to verify the correctness of the code, we introduce the *first integrals*, which are functions that remain constant in time. We are interested in the following quantities:

- the total energy in the system $E^i = E^i_{kin} + E^i_{pot} = \sum_j \frac{m_j}{2} \|\mathbf{v}^i_j\|^2 + \sum_j V(x_i, x_j)$
  (note that each pairwise interaction is only counted once for the potential energy)

- the center of mass $\mathbf{R}^i = \frac{\sum_j m_j \mathbf{x}_j^i}{\sum_j m_j}$
- the total angular momentum $\mathbf{L}^i = \sum_j \mathbf{x}_j^i \times m_j \mathbf{v}_j^i$
- the total linear momentum $\mathbf{p}^i = \sum_j m_j \mathbf{v}_j^i$

where

$$V(x_i, x_j) = 4\varepsilon \left[\left(\frac{\sigma}{|\mathbf{x_i} - \mathbf{x_j}|} \right)^{12} - \left(\frac{\sigma}{|\mathbf{x_i} - \mathbf{x_j}|} \right)^{6} \right], \tag{11}$$

is the Lennard-Jones potential.

a) Implement a Lennard-Jones N-body solver in C++.

Generate initial conditions by placing particles uniformly on a grid and assigning them velocities by drawing each velocity component from a normal distribution with a standard deviation of σ. In order to fix the total energy, compute the total energy of the generated initial conditions and compute the rescaling factor r:

$$E_{\text{kin}}^{\text{target}} = E_{\text{tot}}^{\text{target}} - E_{\text{pot}}^{\text{gen}} \tag{12}$$

$$r^2 = \frac{E_{\text{kin}}^{\text{target}}}{E_{\text{kin}}^{\text{gen}}} \tag{13}$$

where the subscripts kin, pot and tot refer to kinetic, potential and total respectively and the superscripts target and gen refer to the required energy and generated energy respectively. We rescale all velocities with it:

$$\mathbf{v} := r\mathbf{v}. \tag{14}$$

Plot the *first integrals* (see above) at each timestep and verify that the total energy is conserved. Measure and report the time to solution of the individual kernels. What parts of your code would you optimize first? Why?

The first integrals for the N-body problem with 100 particles are shown in Figure 2. Both the total energy and the linear momentum, as expected, are constant over time. On the other hand, the angular momentum is not constant because of the periodic boundary conditions. The center of the mass is also not constant (and almost never is).

Table 1 shows the average time of kernels K1, K2 and the remaining part of the code ("rest") measured over one time step for 100, 1000 and 10000 particles. The average is computed from 10 samples, all obtained from a single simulation, such that the i-th sample corresponds to the execution time of the timestep at the simulation time $i \times T_{end}/10$. The majority of the simulation time is spent on the kernel K2 and therefore this is the part of the code one should optimise at first.

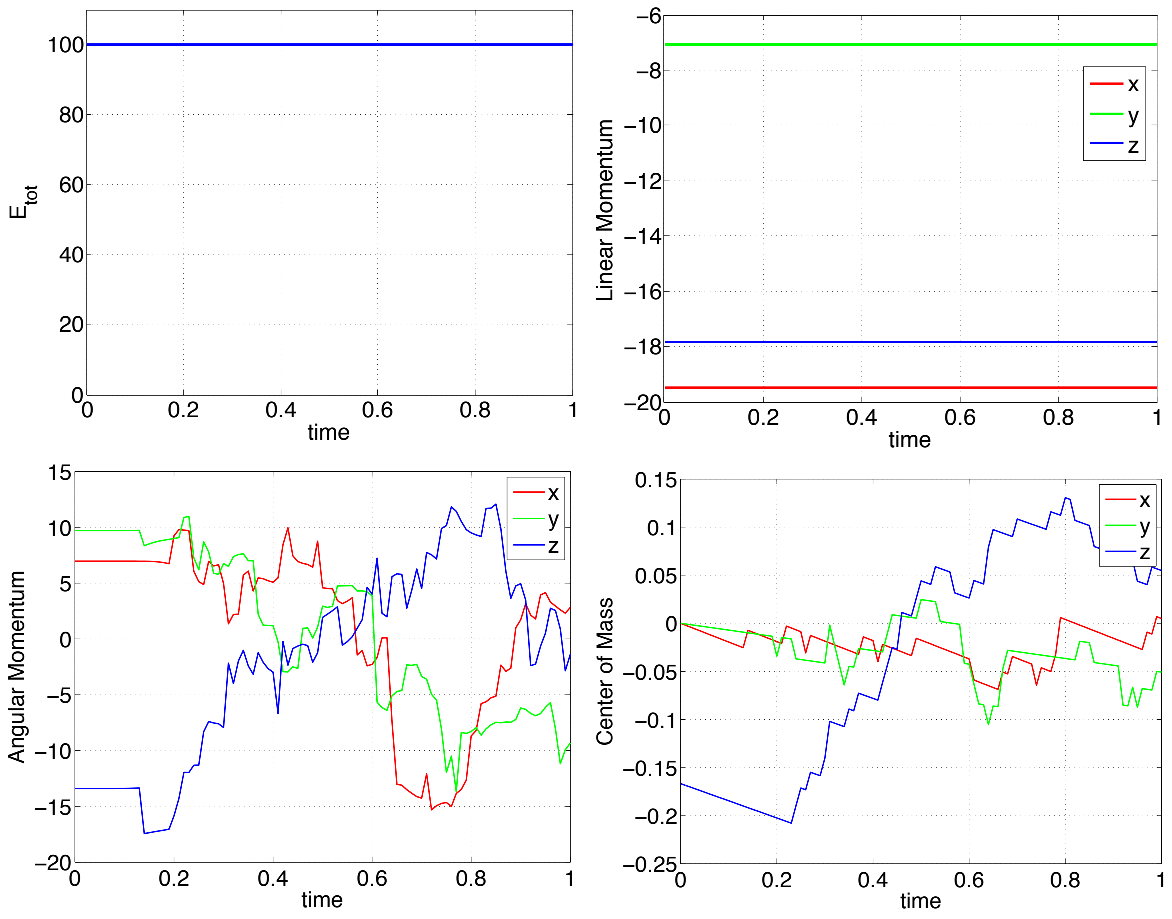

Figure 2: The time evolution of the total energy (top left), the linear momentum (top right), the angular momentum (bottom left) and the center of mass (bottom right). $\Delta t = 10^{-4}$

# Particles	Kernel	CPU time $[ms]$	%
100	K1	0.002	1.7
	K2	0.113	96.4
	rest	0.02	1.9
1000	K1	0.009	0.1
	K2	8.7334	99.8
	rest	0.009	0.1
10000	K1	0.116	0.016
	K2	734.797	99.97
	rest	0.109	0.014

Table 1: Execution time of the kernels K1, K2 and the remaining part of the code for the n-body. The reported CPU time corresponds to the average execution time over one time step.

b) Plot execution time against problem size (amount of bodies). Comment on the scaling behavior.

Figure 3 shows a plot of time versus problem size. The CPU time scales with the square of the problem size, as expected for the n-body problem, which is an N^2 problem.

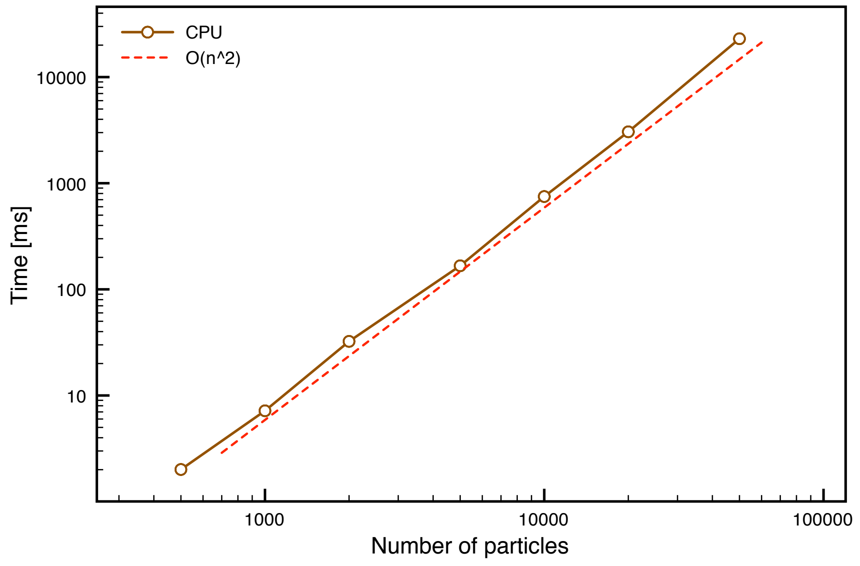

Figure 3: CPU time vs number of particles of the n-body problem on CPU. Run on a single core of AMD Opteron 6174

Question 3: N-Body Problem on GPUs

a) Implement a Lennard-Jones N-body solver with CUDA. Plot the *first integrals* (see above) at each timestep and verify that the total energy is conserved. Measure and report the time to solution of the individual kernels. What parts of your code would you optimize first? Why? Compare and comment on the relative performance of the CPU and GPU versions.

GPU code is much faster than CPU one and we can check the energy and momentum conservation for much larger ensembles. See Figure 4 for the energy evolution of the two different sizes of the particle system. Note that for the larger ensemble energy oscillations are larger, but still less than 0.1%. The linear momentum conservation is achieved due to the third Newton's law and is held constant up to machine precision (see Figure 5).

Figure 6 shows the comparison between K1 and K2 performance versus problem size. It is clear that K2, which contains the computation of forces, account for the majority of the computations and, moreover, has quadratic complexity. That makes K2 a perfect candidate for future improvement.

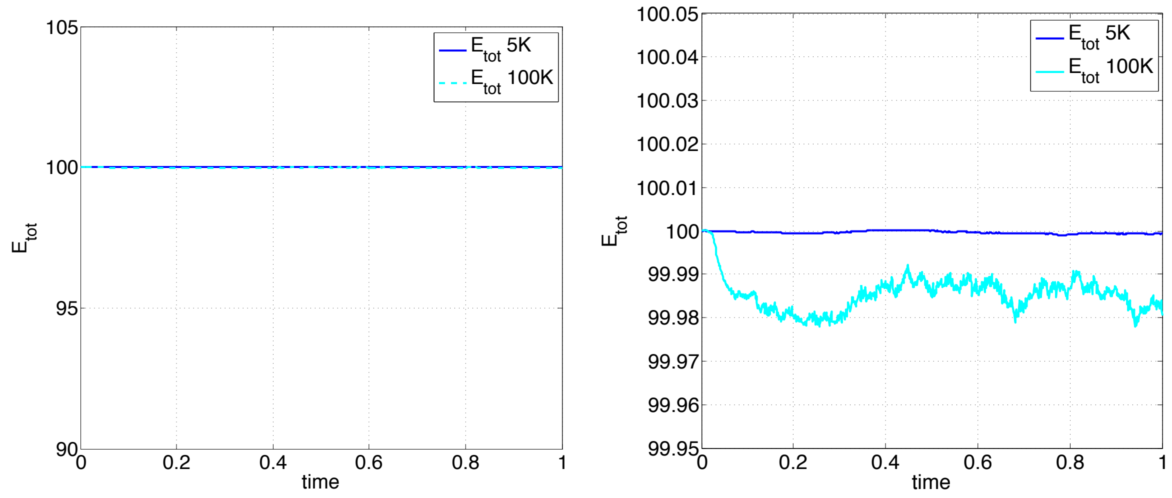

Figure 4: Evolution of the total energy for two particle ensembles of different sizes. Total energy is constant function of time, but if one zoom in the left plot (see plot on the right) small oscillations become visible. $\Delta t = 0.5 \cdot 10^{-5}$

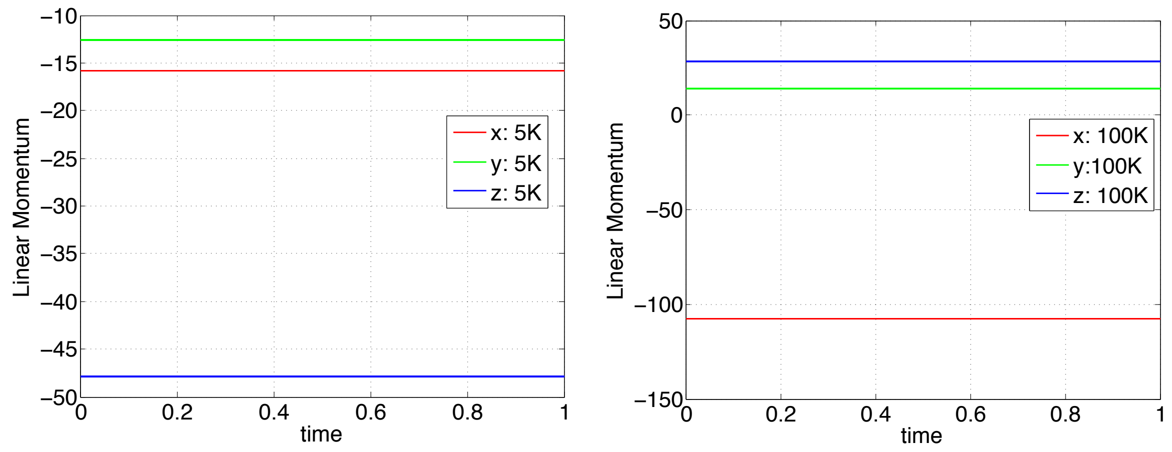

Figure 5: Evolution of the linear momentum for two particle ensembles of different sizes. $\Delta t = 0.5 \cdot 10^{-5}$

See Figure 7 for the performance plots versus problem size. As one can see, a GPU works the better the more parallel work it is given. To hide the memory latency there should be enough threads per streaming multiprocessor. As one thread effectively processes one particle, the more particles are there, the better parallelization is achieved. That is the reason why the GPU code scales better than it should according to $O(n^2)$ complexity of the problem. The saturation point is about $10,000$ particles which corresponds to about 6 blocks with 128 threads each per each of 14 streaming multiprocessors. After this point the GPU code outperforms the CPU code by a factor of 40x.

b) Optimize as much as you can your implementation of the Lennard-Jones forces and report the improvements with respect to the baseline from the previous subquestion.

The pseudocode in Listing 3 can be further improved by making use of the GPUs shared memory

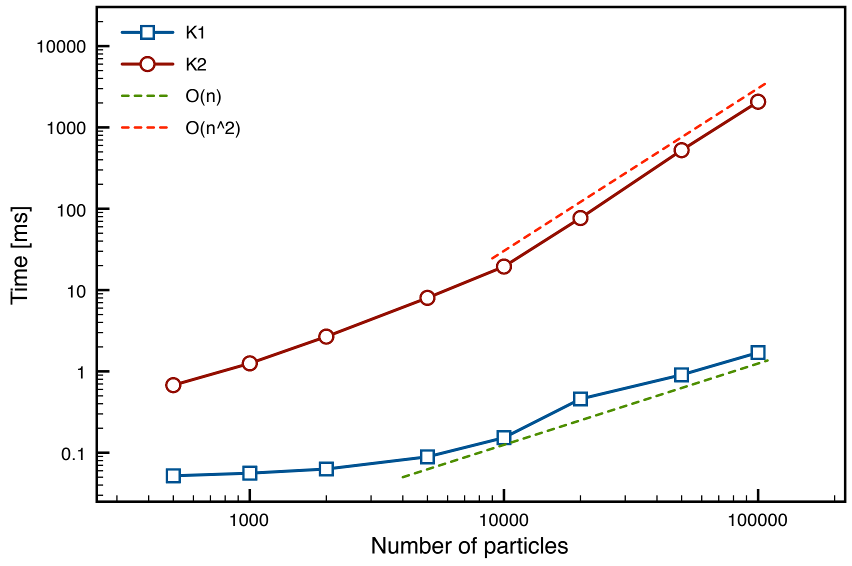

Figure 6: Performance of different computational GPU kernels. Time is averaged for 100 iteration, run on Tesla M2050.

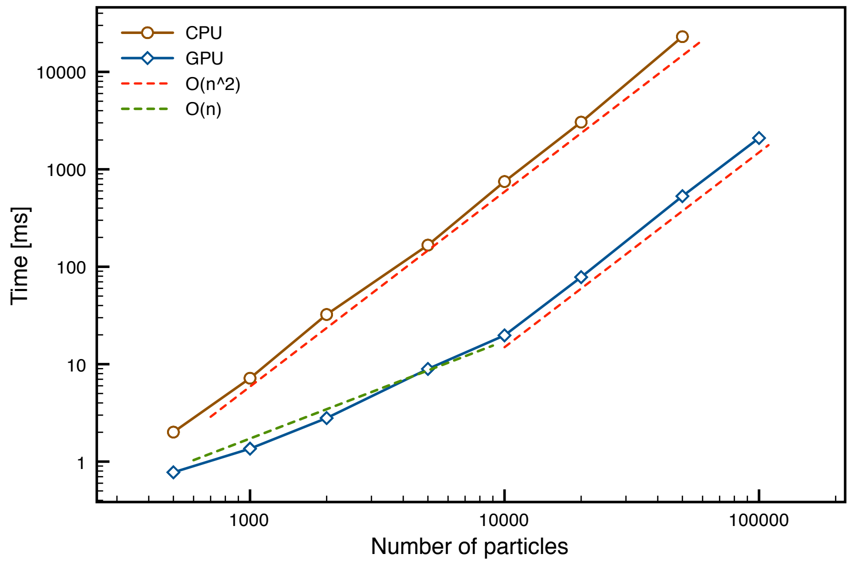

Figure 7: Comparison of CPU and GPU codes. Time is averaged for 100 time-steps, CPU implementation is run on a single core of AMD Opteron 6174, GPU implementation run on a Tesla M2050.

(see Listing 4). Instead of reading values from global memory every time they are used we are prefetching a block of memory (one element per thread) into shared memory. This helps in creating perfectly coalesced memory transactions and reduces the amount of requested global memory transactions. Shared memory in this context can be thought of as a explicitly controlled cache. Since recent GPU architectures (Fermi, Kepler) already have caches, the effect of this optimization is not as pronounced as it would be on GPUs that do not have a cache (such as a Tesla C1060 for example). The effects of the optimization can still be observed when looking at profiling data.

```
// K2 for particle i
c1 =  − 24 ∗ beta ∗ epsilon / m; // 4 FLOP
f = 0;
float3 p_i = p[i]; // remove loading p[i] from the inner loop
__shared__ float3 p_local[];
for (int k = 0; k < N; k+=BLOCK_SIZE)
{
    // preload BLOCK_SIZE values from global array p into the
       local array p_local
    p_local[threadIdx.x] = p[k+threadIdx.x];
    __syncthreads();
    for (int j = 0; j < min(BLOCK_SIZE, N−k); j++)
    {
        if (i != j+k)
        {
            dx = p_local[j].x − p_i.x; // 1 FLOP, 2 reads
            dy = p_local[j].y − p_i.y; // 1 FLOP, 2 reads
            dz = p_local[j].z − p_i.z; // 1 FLOP, 2 reads

            IrI2 = dx ∗ dx + dy ∗ dy + dz ∗ dz; // 5 FLOP

            invIrI2 = 1 / IrI2; // 1 FLOP
            sr2 = sigma ∗ sigma ∗ invIrI2; // 2 FLOP
            sr6 = sr2 ∗ sr2 ∗ sr2; // 2 FLOP

            c2 = (2 ∗ sr6 ∗ sr6 − sr6) ∗ invIrI2; // 4 FLOP

            fx = fx + dx ∗ c2; // 2 FLOP
            fy = fy + dy ∗ c2; // 2 FLOP
            fz = fz + dz ∗ c2; // 2 FLOP
        }
    }
    __syncthreads();
}
v[i].x = v[i].x + c1 ∗ fx; // 2 FLOP, 1 read, 1 write
v[i].y = v[i].y + c1 ∗ fy; // 2 FLOP, 1 read, 1 write
v[i].z = v[i].z + c1 ∗ fz; // 2 FLOP, 1 read, 1 write
```

Listing 4: Pseudocode for the optimized K2 kernel using shared memory.

Detailed profiling data of a specific kernel can be obtained from nvprof or the visual profiler in nsight. Some of the memory related metrics obtained from running the reference solution K2 kernel for 8192 particles are summarized in Table 2. As can be seen from this data the shared memory kernels issue significantly less global memory transactions and consume less bandwidth while the generic kernels heavily rely on available caches. The generic kernel issues a high amount of transactions with and without L1, but since most data is read multiple times this is reflected in high cache hit rates either on L1 or L2 (if L1 is deactivated). The global write transactions on the other hand are identical for all versions since there is actually only a single write happening at the end of the kernel. The non memory metrics such as occupancy, multiprocessor activity and instructions per cycle are relatively even across all versions as well and indicate high compute utilization. Overall the metrics show that the kernel is compute bound which is the reason why the memory optimizations barely show any effect on actual running time. Further factors that can influence performance include the thread block size, unroll pragmas and specifying exact target architectures to nvcc (for example `-arch=sm_20` for the tesla GPUs on brutus). Most of these optimizations are tradoffs in practice and have to be verified with timings and the use of a profiler. Using very agressive unrolling for example could lead to increased register usage which in turn reduces achievable occupancy. Other tradoffs are between speed and precision such as the usage of the rsqrt (reciprocal square root) function, which is very optimized on GPUs since normalization of vectors is a common task in graphics shaders but with relaxed precision requirements. A possible improvement to the algorithm itself is to use the symmetry of the potential calculation which halves the amount of calculations that are required but requires a more involved access scheme to avoid race conditions. This reduces the amount of work in exchange for potentially less parallelism.

Metric	Generic		Shared	
	with L1	without L1	with L1	without L1
Achieved Occupancy	0.65	0.64	0.66	0.66
Multiprocessor Activity	99.0%	99.0%	99.6%	99.6%
Instructions per Cycle (IPC)	1.49	1.39	1.39	1.38
Global Load Transactions	10208128	10208128	13312	13312
Global Store Transactions	768	768	768	768
Global Memory Load Throughput	138.9 GB/s	58.8 GB/s	161.3 MB/s	161.8 MB/s
Shared Memory Load Throughput	0 GB/s	0 GB/s	76.4 GB/s	76.5 GB/s
L1 Global Hit Rate	99.9%	0.0%	46.3%	0.0%
L2 Hit Rate	81.1%	99.9%	92.8%	96.2%

Table 2: A selection of profiling data obtained with nvprof for the referenc solution `K2_kernel` and `K2_kernel_shared` on a GTX 560 TI (single precision, 8192 particles)

Question 4: Cell Lists

Cell lists are data structures to efficiently find nearest neighbors and can be used to accelerate the computation of the N-body problem, if the interactions are local, i.e. short range. By using cell lists, the $\mathcal{O}(N^2)$ complexity of the N-body problem can be reduced to $\mathcal{O}(cN)$, where c is a constant that depends on the interaction cutoff and other parameters. In this question we employ cell lists to evaluate Lennard-Jones forces.

There are several ways to implement cell lists. In the following paragraphs we introduce a technique that allows us to have fine-grained parallel work whose homogeneous computation leverages the increased data-level parallelism of multi/many-core platforms.

For the rest of this exercise we will denote particle positions with (x_i, y_i, z_i) where i is the particle index and ranges from 0 to $N-1$, where N is the total number of particles. The positions are stored in an array $X = \{(x_i, y_i, z_i)\}_i$. We consider a cubic domain of $[-1;1[^3$ with cell size h (per dimension) and periodic boundary conditions. The number of cells in each direction is therefore $M = \frac{2}{h}$. We will only consider cases where h is larger or equal than the cutoff.

Construction of cell data. The first step is to populate the cell data array. Given a particle p_i with coordinates (x_i, y_i, z_i), we compute its cell ID c_i in the following way:

$$c_i = H(p_i) = \left\lfloor \frac{x_i+1}{2} M \right\rfloor + \left\lfloor \frac{y_i+1}{2} M \right\rfloor M + \left\lfloor \frac{z_i+1}{2} M \right\rfloor M^2. \tag{15}$$

The pair (i, c_i) is stored in an array $A = \{(i, c_i)\}_{i=0}^{N-1}$.

Sorting. The data contained in the array A is sorted by using cell IDs as keys. Note that we sort the array in $\mathcal{O}(N)$ as the bit length of the hash codes is constant and known a priori. We denote the sorted array as $B = \{(i, c_i)\}_{i=0}^{N-1}$.

Data reordering. To improve the locality for accessing particle properties of particle $\{p_i\}_i$ such as positions $\{(x_i, y_i, z_i)\}_i$, we reorder the data in X based on the sorted array B so that two particles that are neighbors in B are also located close to each other in memory:

$$(x'_i, y'_i, z'_i) = (x_{id_i}, y_{id_i}, z_{id_i}), \tag{16}$$

where id_i is the particle ID contained in B at position i.

Remarks: the amount of concurrency available in this step is N.

Scan and creation of supporting structures. In order to efficiently traverse cell lists we create the supporting structure $S = \{(s_i, e_i)\}_{i=0}^{M^3-1}$ that, given a cell ID c_i returns the range of particle indices $[s_{c_i}, e_{c_i}[$ in the array B corresponding to that cell. The structure can be build by computing the difference $d_k = c_k - c_{k-1}$ where k is the k^{th} element of the array B. If for any k $d_k \neq 0$, then it means that between positions $k-1$ and k the cell ID c_i changed and we therefore set $s_{c_k} = k$ and $e_{c_{k-1}} = k$.

Remarks: special care must be taken to deal with empty cells and we therefore initialize all elements in S to 0. Also for the last particle p_{N-1} we need to set $e_{c_{N-1}} = N$. One might also try to rewrite S to omit e_i but this requires additional care for empty cells. We note that the amount of concurrency available in this step is N.

Traversal with supporting structures. Given a particle p_i we want to access the cell with cell ID c_i and its neighboring cells $N(c_i)$ in order to find all possible interactions with p_i. We therefore use $[s_{c_i}, e_{c_i}[$ to identify the index range in the array B that contain the particles p_j that could potentially interact with p_i, we compute the interactions and we update the values of particle p_i.

Remarks: the computation of the neighboring cells $N(c_i)$ of c_i should be consistent with the periodic boundary conditions. Furthermore, the amount of concurrency available in this step is N.

See also: `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html`

a) What is the additional memory footprint of the cell lists compared to the N^2 solver? What are the possible implications on performance?

According to the above described implementation technique of cell lists, we use two integers vectors of size N, i.e. equal to the number of particles. The first vector contains the cell ID each particle belongs to, while the second vector is used for storing the mapping between spatially sorted and original particles. In addition, we use an integer vector of size $c = M^3 + 1$, where c is the total number of cells, to store the index of the first particle that is contained in each cell. Therefore, the memory footprint of the simulation increases by $(2N + c) \cdot 4$ Bytes. Despite the additional memory requirements and the overhead associated with maintaining the cell structure, cell lists allow the N-body solver to scales linearly with the number of particles. Therefore, for larger systems this technique results in a significant performance gain and clearly outperforms the N^2 algorithm.

b) Implement the N-body solver for the Lennard-Jones forces in C++ using cell lists. For this question use the truncated potential (see Equation 10). Choose σ such that $N\sigma^3 = 1$ (where N is the amount of particles), set the cutoff at 2.5σ and $\varepsilon = 1$. Generate initial conditions as explained previously.

The implementation of cell lists can be found in the Celllist.h file. The code is based on the thrust library and therefore it targets both CPU and GPU hardware. The migrate() method is called at every simulation step to appropriately update the cell structures. The version of the K2 kernel that takes advantage of the cell lists can be found in the Cpukernels.h file and is also depicted in Listing 5.

```
void _K2(Particles* part, LennardJones* potential, Cells<
  Particles>* cells, double L)
{
  int origIJ[3];
  int ij[3], sh[3];
  double xAdd[3];

  _fill(part->ax, part->n, 0);
  _fill(part->ay, part->n, 0);
  _fill(part->az, part->n, 0);

  double fx, fy, fz;

  // Loop over all the particles
  for (int i=0; i<part->n; i++)
  {
    double x = part->x[i];
    double y = part->y[i];
    double z = part->z[i];

    // Get id of the cell and its coordinates
    int cid = cells->which(x, y, z);
    cells->getCellIJByInd(cid, origIJ);

    // Loop over all the 27 neighboring cells
    for (sh[0]=-1; sh[0]<=1; sh[0]++)
      for (sh[1]=-1; sh[1]<=1; sh[1]++)
        for (sh[2]=-1; sh[2]<=1; sh[2]++)
        {
          // Resolve periodicity
          for (int k=0; k<3; k++) ij[k] = origIJ[k] + sh[k];

          cells->correct(ij, xAdd);

          cid = cells->getCellIndByIJ(ij);
          int begin = cells->pstart[cid];
          int end   = cells->pstart[cid+1];

          for (int j=begin; j<end; j++)
          {
            int neigh = cells->pobjids[j];
            if (i != neigh)
            {
              double dx = part->x[neigh] + xAdd[0] - x;
              double dy = part->y[neigh] + xAdd[1] - y;
              double dz = part->z[neigh] + xAdd[2] - z;

              potential->F(dx, dy, dz, fx, fy, fz);

              part->ax[i] += fx;
              part->ay[i] += fy;
              part->az[i] += fz;
            }
          }
        }

        const double m_1 = 1.0 / part->m[i];
        part->ax[i] *= m_1;
        part->ay[i] *= m_1;
        part->az[i] *= m_1;
  }
}
```

Listing 5: K2 kernel with cell lists.

c) Implement the N-body solver for the Lennard-Jones forces in CUDA using cell lists. For this question use the truncated potential (see Equation 10). Verify the correctness of your code by plotting the total energy and linear momentum at each timestep.

Remarks: you can find implementations of GPU sorting functions in the thrust library (which is part of the CUDA SDK) or CUDPP (`http://code.google.com/p/cudpp/`).

Due to the use of the thrust library, the code that handles the cell lists remains the same. The GPU-based K2 kernel maps a work item (thread) to the computation of the forces for a single particle. In addition, it calculates each interaction twice, without gaining any benefit from the symmetry of the computations. A multipass implementation may be an optimization for the code, but additional care should be taken as it increases complexity of the code and decreases the level of parallelism. The code of the K2 kernel can be found Gpukernels.h file and is also depicted in Listing 6.

```
__global__ void _K2_kernel(Particles part, int n,
  LennardJones potential, CellsInfo cells)
{
  int ij[3], origIJ[3];
  double xAdd[3];

  const int gid = threadIdx.x + blockDim.x*blockIdx.x;
  if (gid >= n) return;

  double x = part.x[gid];
  double y = part.y[gid];
  double z = part.z[gid];

  double fx, fy, fz;
  double ax = 0, ay = 0, az = 0;

  // Get id of the cell where the particle is situated and
    its coordinates
  int cid = cells.which(x, y, z);
  cells.getCellIJByInd(cid, origIJ);

  // Loop over all neigboring cells
  for (int direction_code = 0; direction_code < 27;
    direction_code++)
  {
    // Get the coordinates of the cell we work on
    ij[0] = (direction_code % 3) - 1;
    ij[1] = (direction_code/3 % 3) - 1;
    ij[2] = (direction_code/9 % 3) - 1;

    // Resolve periodicity
    for (int k=0; k<3; k++) ij[k] += origIJ[k];

    cells.correct(ij, xAdd);

    // Get our cell id
    cid = cells.getCellIndByIJ(ij);
    int begin = cells.pstart[cid];
    int end   = cells.pstart[cid+1];

    // Calculate forces for all the particles in our cell
    for (int j=begin; j<end; j++)
    {
      int neigh = cells.pobjids[j];
      if (gid != neigh)
      {
        double dx = part.x[neigh] + xAdd[0] - x;
        double dy = part.y[neigh] + xAdd[1] - y;
        double dz = part.z[neigh] + xAdd[2] - z;

        potential.F(dx, dy, dz, fx, fy, fz);

        ax += fx;
        ay += fy;
        az += fz;
      }
    }
  }

  const double m_1 = 1.0 / part.m[gid];
  part.ax[gid] = m_1 * ax;
  part.ay[gid] = m_1 * ay;
  part.az[gid] = m_1 * az;
}

void _K2(Particles* part, LennardJones* potential, Cells<
  Particles>* cells, double L)
{
  const int threads = 128;
  const int blocks  = (part->n + threads - 1) / threads;
  _K2_kernel<<<blocks, threads>>>(*part, part->n, *potential,
     *cells);
}
```

Listing 6: K2 kernel with cell lists.

The total energy and the linear momentum of the GPU-based simulation that uses cell lists are depicted in Figure 8. We observe that the results are in accordance to those of the same simulation that did not use the cell lists technique.

d) Plot execution time against problem size (amount of bodies). Comment on the scaling behavior. How does it differ from the scaling observed without cell lists? Compare and comment on the relative performance of the CPU and GPU versions.

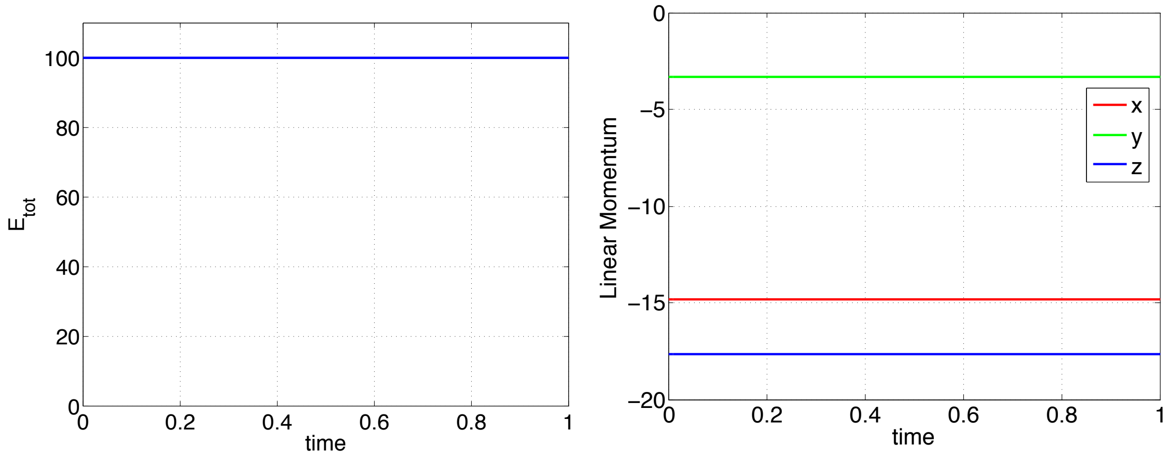

Figure 8: Evolution of the integrals for simulation with 100 particles: total energy on the left and the three components of the linear momentum on the right. $\Delta t = 10^{-4}$

Figure 9 depicts the execution time of the various implementations against problem size. We observe that both (CPU and GPU) implementations of cell lists result in linear complexity and outperform the N^2 solver, especially as the number of particles increases. For very low number of particles, both cell list implementations show the same or even worse performance as their N^2 counterparts, due to the runtime overheads for maintaining the cell structures and underutilization of the GPU architecture. Though when the number of particles is of order of hundreds, cell list code outperforms the generic variant. Moreover, the GPU implementation outperforms the CPU code by the factor of about 20x.

e) Optional: Use VMD or OpenGL to visualize the data (particle positions) and create a movie of them.

The snapshot from the simulation visualized with VMD is shown in Figure 10.

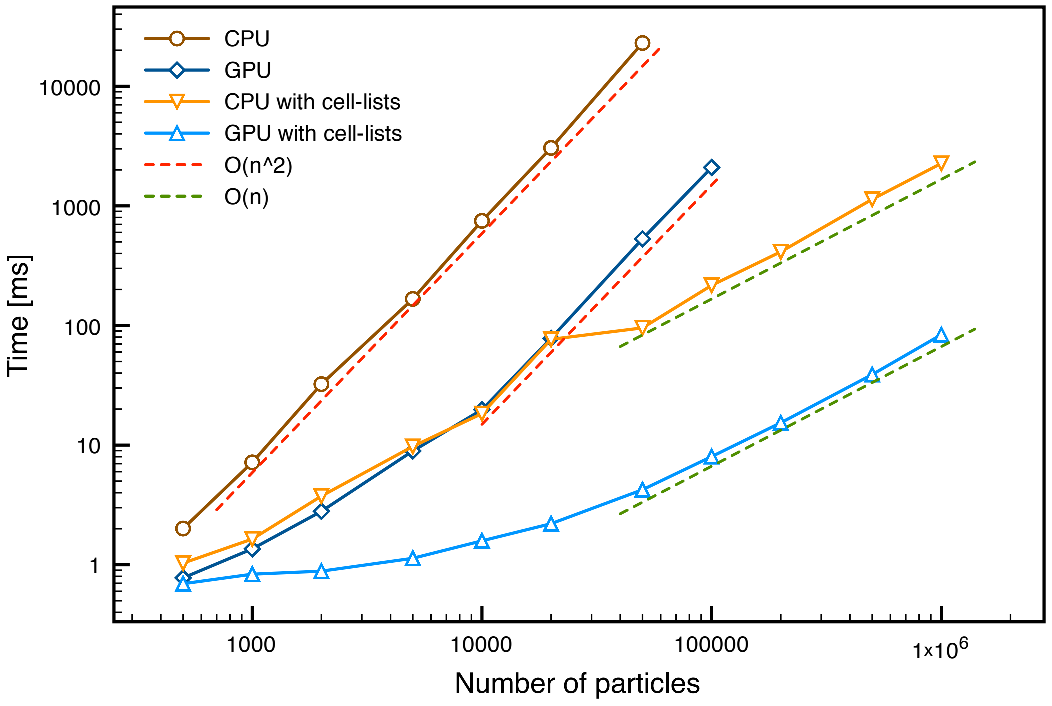

Figure 9: Performance comparison of CPU and GPU codes with and without cell lists. Time is averaged for 100 time-steps, CPU implementation is run on a single core of AMD Opteron 6174, GPU implementation is run on a Tesla M2050.

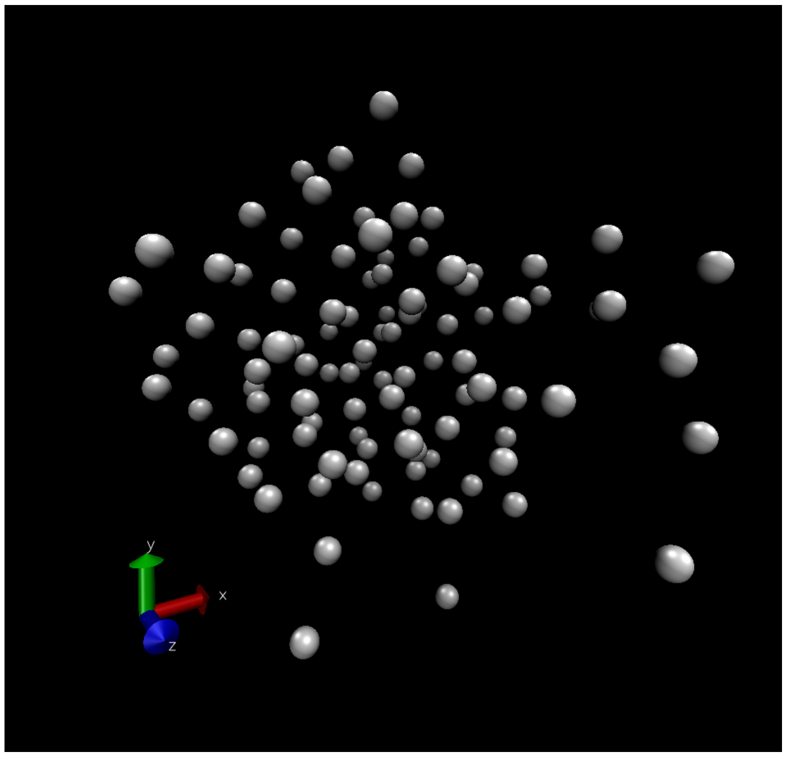

Figure 10: Snapshot from the simulation visualized with VMD.