High performance implementation of hydrodynamic interactions and applications with the sub-cellular element method

High performance implementation of hydrodynamic interactions and applications with the sub-cellular element method

Andrei Frunza
frunzaa@student.ethz.ch

Master Thesis
Computational Science and Engineering Laboratory

ETH Zürich

Advisor:
Gerardo Tauriello

Professor:
Prof. Petros Koumoutsakos

May 28, 2012

1 Abstract

An $\mathcal{O}(N^2)$ algorithm for computing hydrodynamic interaction (HI) in Brownian dynamics (BD) simulation has been implemented. A CPU and a GPU versions have been build, with the GPU one being tuned for performance, up to 40% of the maximum peak performance being obtained. The implementation was validated through simulations of diffusion polymers and comparisons of the yielded parameters with theoretical values. The hydrodynamic interactions were added to cell stretching experiments, in which the cells are modelled with the sub-cellular element method and a delayed failure behaviour was observed.

Contents

2 Introduction

This paper aims to present the findings following simulations of stretching of biological cells. The core of the work done consisted in building a LAMMPS module for adding hydrodynamic interactions (HI) to Brownian dynamics (BD) simulations. This was used alongside the sub-cellular element method to study the creep response of a single cell and compare to power-laws observed in experiments.

Succinctly, BD describes the motion of particles immersed in a fluid which are assumed to have a larger mass and volume then the underlying medium. Usually the particles of the fluid are not considered for the simulation as this can be very computationally expensive for the simulation of large systems. When the solvent is modelled implicitly, the interaction of the Brownian particles through the solvent has to be added separately. When a Brownian particle moves through the liquid, it displaces some liquid around it which in turn influences the displacement of other Brownian particles immersed in the same fluid.

In their seminal work, Ermak and McCammon [2] present a coarse grained method for computing HI through a position-dependent interparticle friction tensor. In their algorithm the computation of HI scales in run time as $\mathcal{O}(N^3)$, while the pairwise interactions can be computed in $\mathcal{O}(N^2)$ or less. For this reason Geyer and Winter [3] proposed an approximation to HI that is less computationally involved and only approximates the hydrodynamic coupling for the random forces in the simulation. Another way to model a fluid, with explicit water molecules ad within a cutoff distance, is Dissipative particle dynamics (DPD), presented by Groot in [4]. For the implementation of HI we chose Geyer's method.

For the simulation of the cells we used the sub-cellular elements method [6] in a similar setup as was used by Sandersius in [8]: the cell was stretched between one fixed and one mobile walls with constant strain. The final goal of this work was to check how adding HI to the cell influences the creep response and how it compares to living cell experiments done by Desprat [1].

The paper is structured in three main parts. The first one presents in more detail the methods used for computing HI, the second describes how the implementation was done and what was the performance obtained and in the third one we talk about how the implementation was validated and how the cell stretching experiment was conducted and what were the results. Finally the last two section of the paper present a few possible further developments and draw the final conclusions of this work.

3 Background Information

Brownian dynamics (BD) describes the dynamic behaviour of a system of particles whose mass and size are much larger than the ones of the medium they are dissolved in, for example water. For computational cost reasons, one usually wants to eliminate the simulation of all the solvent particles and compute their influence on the dissolved particles in some other way.

For the simulation of the diffusive movement of a system of N Brownian particles in solvent different methods are used. In their work, Ermak and McCammon [2] present a method that uses a set of $3N$ coupled Langevin equations and compare it with a derivation from the Fokker-Plank differential equations for the system.

The Langevin description derives from Newton's law of motion

$$m\ddot{\mathbf{r}} = \mathbf{F} + \mathbf{F}^D + \mathbf{F}^R. \tag{1}$$

where $\mathbf{F}$ are the forces exerted on the particle, $\mathbf{F}^D$ are dissipative forces due to hydrodynamics and $\mathbf{F}^R$ are random forces following the fluctuation-dissipation theorem.

The dynamics of the system can be encoded with a $3N \times 3N$ friction tensor $\boldsymbol{\zeta}$ and we solve Eq. (1) with

$$F_i^D = -\zeta_{ij} v_j, \tag{2}$$

where v_j are the velocities and the random forces are sampled to have the following mean and correlations:

$$\begin{aligned} \left\langle F_i^R(t) \right\rangle &= 0, \\ \left\langle F_i^R(t) F_j^R(t') \right\rangle &= 2k_B T \zeta_{ij} \delta(t - t'). \end{aligned} \tag{3}$$

In practice one would set the force for a timestep of Δt as

$$f_i = m\ddot{r}_i = -\sum_j \zeta_{ij} v_j + F_i + \sqrt{2k_B T/\Delta t} \sum_j \alpha_{ij} X_j, \tag{4}$$

where α_{ij} are computed from the Cholesky factorization $\boldsymbol{\zeta} = \alpha\alpha^T$ and $X_j \sim \mathcal{N}(0,1)$ are sampled from uncorrelated standard normal distributions. Note that this requires a $\mathcal{O}(N^2)$ computation to compute $\mathbf{F}^D$ and a $\mathcal{O}(N^3)$ computation for the Cholesky factorization to get $\mathbf{F}^R$.

In BD the effects of inertia are assumed negligible ($f_i = 0$) and a diffusion tensor $\mathbf{D} = k_B T \boldsymbol{\zeta}^{-1}$ is introduced. We further assume that F_i do not change considerably during Δt (just like with any time integration) and that $\Delta t \gg m_i/\zeta_{ij}$ or $\Delta t \gg m_i D_{ii}/k_B T$.

The displacement equation for each component in the Ermak-McCammon algorithm is given by:

$$\Delta r_i(\Delta t) = \sum_j \frac{\partial D_{ij}}{\partial r_j}\Delta t + \sum_j \frac{D_{ij}F_j}{k_B T}\Delta t + R_i(\Delta t), \tag{5}$$

where D_{ij} is the diffusion tensor, F_j is the external force acting on component j, k_B is Boltzmann's constant, T the temperature of the system and $R_i(\Delta t)$ is a random displacement chosen from a multivariate normal distribution with:

$$\langle R_i(\Delta t)\rangle = 0 \text{ and } \langle R_i(\Delta t), R_j(\Delta t)\rangle = 2D_{ij}\Delta t. \tag{6}$$

As pointed out in [2] the random displacements can be determined as $R_i(\Delta t) = \sqrt{2\Delta t}\sum_j B_{ij}X_j$ with $X_j \sim \mathcal{N}(0,1)$. The $\mathbf{B}$ matrix is computed so that $\mathbf{D} = \mathbf{B}\mathbf{B}^T$, in the Ermak-McCammon algorithm a Cholesky factorisation being used to solve this.

Based on this implicit solvent method, Geyer and Winter proposed in [3] an approximation that circumvents the expensive Cholesky factorization resulting in an $\mathcal{O}(N^2)$ algorithm for computing the hydrodynamically correlated displacements of the Brownian particles. We will present this method in more detail in the next subsection.

Another method to model a fluid, but with explicit water particles this time, was described by Groot in [4]. In Dissipative Particle Dynamics (DPD) particle interactions are limited to a cutoff distance r_c, thus allowing the forces to be computed in $\mathcal{O}(N)$. Next, we solve Eq. (1) with pairwise forces and pairwise random forces ($i, j \in [1, N]$):

$$\begin{aligned}
m\ddot{\mathbf{r}}_i &= \sum_{j\neq i} \mathbf{F}_{ij} + \mathbf{F}^D_{ij} + \mathbf{F}^R_{ij},\\
\mathbf{F}_{ij} &= \begin{cases} a_{ij}(1 - r_{ij}/r_c)\hat{\mathbf{r}}_{ij} & \text{if } r_{ij} < r_c \\ 0 & \text{else} \end{cases},\\
\mathbf{F}^D_{ij} &= -\gamma w^D(r_{ij})(\hat{\mathbf{r}}_{ij}\cdot\mathbf{v}_{ij})\hat{\mathbf{r}}_{ij},\\
\mathbf{F}^R_{ij} &= \sqrt{2\gamma k_B T w^D(r_{ij})}\theta_{ij}\hat{\mathbf{r}}_{ij},\\
\langle\theta_{ij}(t)\rangle &= 0,\\
\langle\theta_{ij}(t)\theta_{kl}(t')\rangle &= (\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})\delta(t - t'),\\
w^D(r_{ij}) &= \begin{cases} (1 - r_{ij}/r_c)^2 & \text{if } r_{ij} < r_c \\ 0 & \text{else} \end{cases},
\end{aligned} \tag{7}$$

where k_B is the Boltzmann constant, T is temperature, $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$ is the relative velocity, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the distance vector between particles i and j, $r_{ij} = \|\mathbf{r}_{ij}\|$ is the distance and $\hat{\mathbf{r}}_{ij} = \mathbf{r}_{ij}/r_{ij}$ the direction.

3.1 Mathematical model - TEA-HI

As in the Ermak-McCammon algorithm the particle positions along each coordinate are updated without inertia in a time step of length Δt as $r_i = r_i + \Delta r_i(\Delta t)$ where Δr_i is given by:

$$\Delta r_i(\Delta t) = \sum_j \frac{D_{ij}F_j}{k_BT}\Delta t + \sqrt{2\Delta t}C_i\left(\sqrt{D_{ii}}X_i + \beta' \sum_{j\neq i}\frac{D_{ij}X_j}{\sqrt{D_{jj}}}\right), \quad (8)$$

where $X_i \sim \mathcal{N}(0,1)$ are sampled from uncorrelated normal distributions, the other notations are not changed and the values for C_i and β' as derived by Geyer and Winter in their proposed approximation [3], can be computed in the following way:

$$\left(\frac{1}{C_i}\right)^2 = 1 + \beta'^2 \sum_{k\neq i}\frac{D_{ik}^2}{D_{ii}D_{kk}}, \quad (9)$$

$$\beta' = \frac{1-\sqrt{1-[(3N-3)\epsilon^2-(3N-6)\epsilon]}}{(3N-3)\epsilon^2-(3N-6)\epsilon}, \quad (10)$$

$$\epsilon = \langle D_{ij}/D_{ii}\rangle = \frac{1}{3N(3N-3)}\sum_i\sum_{j\neq i}\frac{D_{ij}}{D_{ii}}. \quad (11)$$

These values are derived under the assumption that the hydrodynamic coupling is weak, i.e. the diagonal values of the diffusion tensor $\mathbf{D}$ are much larger than the off-diagonal values. For determining the scaling constants C_i the covariance constraint for the random forces (Eq. (6)) is used, and due to symmetry reasons a single constant β' is needed.

In their proposal, Geyer and Winter treat the forces and random forces on equal footing, introducing the effective force so that the displacement in one time step is computed from this force as

$$\Delta r_i(\Delta t) = \frac{D_{ii}\Delta t}{k_BT}(F_i^{eff} + f_i^{eff}). \quad (12)$$

with the hydrodynamically corrected effective force F_i^{eff}:

$$F_i^{eff} = \sum_j \frac{D_{ij}}{D_{ii}}F_j, \quad (13)$$

and the hydrodynamically corrected random force f_i^{eff}:

$$f_i^{eff} = C_i\left(f_i + \beta' \sum_{j\neq i}\frac{D_{ij}}{D_{ii}}f_j\right), \quad (14)$$

with $f_j = \sqrt{\frac{2(k_BT)^2}{D_{ii}\Delta t}} X_j$, $X_j \sim \mathcal{N}(0,1)$.

3.2 Diffusion tensors

Similarly to Geyer we used the Rotne-Prager-Yamakawa tensor [7] for equal-sized spherical particles. Here the $\mathbf{D}$ tensor consists of $N \times N$ submatrices $\mathbf{D}_{ij}$ of size 3×3 ($i, j \in [1, N]$):

$$\mathbf{D}_{ii} = \frac{k_BT}{6\pi\eta a}\mathbf{I}, \tag{15}$$

$$\mathbf{D}_{ij} = \frac{k_BT}{8\pi\eta r_{ij}}\left[(\mathbf{I} + \hat{\mathbf{r}}_{ij} \otimes \hat{\mathbf{r}}_{ij}) + \frac{2a^2}{3r_{ij}^2}(\mathbf{I} - 3\hat{\mathbf{r}}_{ij} \otimes \hat{\mathbf{r}}_{ij})\right] \quad \text{for } i \neq j \text{ and } r_{ij} \geq 2a, \tag{16}$$

$$\mathbf{D}_{ij} = \frac{k_BT}{6\pi\eta a}\left[\left(1 - \frac{9r_{ij}}{32a}\right)\mathbf{I} + \frac{3r_{ij}}{32a}\hat{\mathbf{r}}_{ij} \otimes \hat{\mathbf{r}}_{ij}\right] \quad \text{for } i \neq j \text{ and } r_{ij} < 2a. \tag{17}$$

Another commonly used hydrodynamic tensor is the Oseen tensor [5]. We again write it as a $N \times N$ matrix of 3×3 sub-matrices $\mathbf{D}_{ij}$:

$$\mathbf{D}_{ii} = \frac{k_BT}{6\pi\eta a}\mathbf{I} \tag{18}$$

$$\mathbf{D}_{ij} = \frac{k_BT}{8\pi\eta r_{ij}}\left(\mathbf{I} - \hat{\mathbf{r}}_{ij} \otimes \hat{\mathbf{r}}_{ij}\right), i \neq j. \tag{19}$$

In both tensors k_B is Boltzmann's constant, T is the temperature of the system, η denotes the viscosity of the solvent, a is the radius of the spherical particles, $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ is the distance vector between particles i and j, $r_{ij} = ||\mathbf{r}_{ij}||$ is the distance and $\hat{\mathbf{r}}_{ij} = \mathbf{r}_{ij}/r_{ij}$ is the direction; $\mathbf{I}$ denotes the unit tensor and $\otimes$ the outer product.

Both these tensors have the property:

$$\sum_j \frac{\partial D_{ij}}{\partial r_j} \equiv 0, \tag{20}$$

which is very useful by simplifying the computation of the displacements of the Brownian particles as described by eq. Eq. (5).

3.3 Pairwise interactions

With this method of computing HI, two types of systems have been simulated: a diffusing polymer and a cell stretching experiment.

The polymer was modelled as a bead-spring system where the beads of radius a were connected to their direct neighbours by harmonic springs of length L with the potential:

$$V_h(x) = a_h(x - L)^2, \tag{21}$$

and for the non-neighbouring beads a repulsive potential was applied to prevent overlaps:

$$V_r(x) = a_h(x - 2a)^2. \tag{22}$$

For the in-silico cell stretch experiments the sub-cellular elements method (SEM) [6] was used to model the cell structure and computation of particle-particle pairwise forces. For the pairwise intracellular interactions a setup similar to the one used in [8] was used. The potential used has the form:

$$V_{sem} = U_0 e^{2\rho\left(1-\frac{r^2}{r_{eq}^2}\right)} - 2U_0 e^{\rho\left(1-\frac{r^2}{r_{eq}^2}\right)}, \tag{23}$$

where U_0 is the depth of the potential well, r_{eq} denotes the equilibrium distance and ρ is a scaling factor.

4 Implementation in LAMMPS

In order to use different inter-particle potentials together with the hydrodynamic interaction approximation a LAMMPS-integrated implementation was preferred. LAMMPS ("Large-scale Atomic/Molecular Massively Parallel Simulator") is a molecular dynamics code that has a large number of different interaction potentials, integrators, takes care of setup, time stepping, saving and loading of simulation snapshots, computes different output parameters and has an easy-to-use script as input interface.

Besides these general features, another reason for choosing LAMMPS is performance and relative ease of development. Being written modularly with MPI it can take advantage of parallel machines for large simulations, it computes and keeps track of neighbour lists and it also has proven ability of using GPU code for the acceleration of compute-intensive parts of a simulation. All these reasons determined us to integrate the implementation of hydrodynamic interactions in LAMMPS writing CPU code for debugging and testing and GPU code for improved performance.

4.1 Implementation details

In order to have an implementation of HI that scales as $\mathcal{O}(N^2)$ with the number particles we opted for the implementation of the approximation of HI as briefly described in section 3.1.

As we can see in equation Eq. (8), the trajectory update for each component during one time step depends on the external forces F_j and random forces f_j on all other components, the diffusion tensor D_{ij} which as described in section 3.2 are functions of the particle positions and the scaling factors C_i and β'. Furthermore equations Eq. (9), Eq. (10) and Eq. (11) show that these scaling factors only depend on the diffusion tensor.

In the end, the input for this problem consists of the positions, external and random forces, which scales linearly with the size of the problem, while the interaction pattern and thus the computational cost scale with N^2. In order to have a chance at obtaining good performance we tried to push the implementation to being as close to compute bound as possible. We did this by limiting the use of memory and not storing the full diffusion tensor, which has N^2 memory footprint. Instead, we computed the values of the diffusion tensor one 3×3 sub-matrix at a time, which corresponds to a particle-particle interaction. These values were accumulated in sums to compute the values of C_i, β' and partial results for the effective forces. A pseudo-code can be found in algorithm 1 where the partial sums are represented by the values $\mathbf{c}$ for C_i, $\mathbf{r}_f$ for the displacement due to force, $\mathbf{r}_{rand}$ for the displacement due

to the random force and ϵ for the value of β'.

Algorithm 1 TEA-HI

Input: N particles: x - positions (3N), f - forces (3N), $rand$ - $\mathcal{N}(0,1)$ (3N)
Output: f^{eff}: effective forces (3N), x: updated positions (3N)

```
Reset ϵ, c, r_f, r_rand
for i = 1 → N do
  for j = 1 → N do
    Compute D_ij as 3 × 3 matrix
    ϵ+ = Σ D_ij
    c(i)+ = Σ_j D_ij^2 {Element wise square}
    r_f(i)+ = D_ij f_j {f_j has the 3 components of particle j}
    r_rand(i)+ = D_ij rand_j {rand_j has the 3 components of particle j}
  end for
  ϵ/ = D_ii
end for
ϵ/ = 3N(3N − 3)
β' = (1−√(1−[(3N−3)ϵ^2−(3N−6)ϵ])) / ((3N−3)ϵ^2−(3N−6)ϵ)
for i = 1 → N do
  c(i) = 1 / √(1−β'^2 c(i)/D_ii^2)
  Compute effective forces described in Eq. (13) and Eq. (14)
  f^eff(i) = r_f(i)/D_ii + c(i) ∗ (rand_i + β r_rand(i)/D_ii)
  Compute displacements as described in Eq. (12)
  x(i)+ = (D_ii Δt / k_B T) f^eff(i)
end for
return f^eff, x
```

This algorithm is based on three kernels: kernel one (K1) that computes the diffusion tensor and accumulates partial results, kernel two (K2) which uses these partial results to compute the effective force and kernel 3 (K3) that updates the positions of the particles. The first kernel consists of the N^2 loop and this is the main place where performance was obtained.

Based on the algorithm's inputs and outputs and on how HI will be used in the experiments, the implementation was integrated in LAMMPS in the following way:

- a LAMMPS **Fix** was developed as it exposed the best interface for what we needed to compute during a time step, for example build a custom integration method that updates positions based only on the force applied to the particle;

- The `post_force(int)` method was implemented which incorporated the first kernel and the computation of the "effective" that is a sum of the effective forces as defined by Geyer in [3]. This approach of splitting the positions update from the computation of the force is useful because it allows us to alter these forces by other LAMMPS commands before the actual position updates. This method implements the first kernel and the second kernel of the algorithm;

- The `final_integrate()` method was implemented for the update of the positions based on the hydrodynamically corrected forces computed in the `post_force()` and possibly later altered by other LAMMPS commands.

Based on the above considerations two implementations were done. The first one uses only the CPU with the ability of using multiple MPI processes to compute the HI but without any optimizations - this implementation was used for testing correctness and then as a comparison base for the second one. The second implementation uses GPU for accelerating the compute-intensive part — namely the first kernel.

For the generation of the random numbers two different methods were used. For the CPU implementation, the C++ STD Technical Report 1 extension library was used. On the GPU, we used `curand` library functionality for generating normally distributed random numbers.

4.2 Performance considerations

For testing the correctness of the implementation and for performance measurements two machines have been used. For correctness tests and timings used for determining the bottlenecks/compute-intensive parts of the simulations I used the laptop machine on which the code was implemented. This machine has an Intel Core i7-2620M CPU running at 2.7 GHz, with 8 GB RAM, 64 KB L1 cache, 256 KB L2 cache and 4 MB L3 cache. The GPU code was developed in CUDA and the GPU on the laptop is a NVS 4200M with CUDA capability 2.1, 1023MB global memory, 1 multiprocessor with 48 cores, a clock speed of 1.64 GHz, the memory clock rate of 800 MHz and bus width of 64 bit and L2 cache of 64 KB.

A more powerful machine (called Cyrus) was used for performance measurements of the GPU implementation. This machine has a GeForce GTX 580 with CUDA capability 2.0, 1536 MB global memory, 16 multiprocessors with 32 cores each, a clock speed of 1.62 GHz, the memory clock rate of 2016 MHz, bus width of 384 bit and L2 cache size of 786432 bytes.

Time measurement was done with the `gettimeofday()` method for CPU the implementation and by using the events mechanism.

For micro-benchmarks of the HI implementation in CUDA a small module outside of LAMMPS was build that generates 3 vectors of random numbers that correspond to the positions of the particles, forces and random kicks and feed them together with the other needed parameters in the 2 kernels that implement the hydrodynamic interaction. With this system we were able to also use nVidia's `computeprof` tool to get more detailed information about how the kernels behave and some hints on how to improve performance.

CPU vs. GPU

The baseline implementation on both CPU and GPU closely follow the algorithm 1 with the parallelism used for the outer loop of kernel 1. Each thread is taking care of the computation of the hydrodynamically corrected effective force of one particle; it loops through all other particles, computes the hydrodynamic tensor which consists of 6 values (being a 3×3 symmetric matrix), and accumulates the partial sums for C_i, β, effective force and random effective force.

A first measurement that we did was to compare how the baseline implementations on both the CPU and GPU behave. For this I used chains of beads of different lengths where beads were connected to their neighbours by harmonic springs and ran the simulations for a large number of time steps and averaged the time. This measurement was run on the "laptop" computer and the timing results measured were the total time it took the simulation to advance one time step and the time spent to compute HI. The results can be seen in Table 1.

N	CPU		GPU	
	time step (ms)	HI (ms)	time step (ms)	HI (ms)
128	0.98	0.95	1.66	0.53
256	3.93	3.87	1.87	0.73
512	15.63	15.55	2.43	1.24
1024	62.26	62.26	4.60	3.35
2048	244.82	244.74	10.93	9.50
2560	380.69	380.30	16.20	14.69
5120	1521.82	1521.53	56.41	54.42
10240	6062.80	6058.90	213.57	210.36

Table 1: CPU vs GPU baselines

As expected almost all the time is spent in HI, the forces exerted from the

springs being computed very fast. We can also see that the straightforward GPU implementation beats the CPU one by an order of magnitude for large sizes. As the two computing devices have very different architectures and peak performance, from now on performance of an implementation will only be compared to the peak performance of the device it runs on.

N	Init	Gen	Copy	K1	Reduce	K2&K3	Results
256	31.60	2.61	1.46	53.86	5.07	2.21	3.19
512	18.59	1.52	0.93	60.17	15.11	1.52	2.16
1024	12.07	0.95	0.69	74.26	9.50	0.95	1.58
2048	7.57	0.59	0.53	84.14	5.42	0.56	1.20
2560	6.41	0.51	0.49	86.60	4.44	0.46	1.10
5120	2.15	0.19	0.26	95.07	1.56	0.17	0.59
10240	1.05	0.08	0.15	97.74	0.55	0.08	0.36
20480	0.56	0.03	0.09	98.92	0.17	0.03	0.20

Table 2: Distribution of GPU time for different parts of the algorithm.

Next we analysed how the time spend in HI is distributed among the different parts of the implementation. We timed how much allocating device memory and copying the input data to the GPU (Init), generating the random numbers on the device (Gen), copying the random numbers if they are generated on the CPU (Copy), computing the $\mathcal{O}(N^2)$ kernel (K1), the reduction (Reduce), second kernel (K2) and the copying the results back to the host memory (Results) take. All these timings are in table 2 expressed as percentages of the total.

We can see that the most expensive operation is K1 consuming 75% of total time for sizes in the order of thousand and exceeding 95% for sizes larger than 5000. We also observe that the copying of the random numbers if they are computed on the CPU is almost as expensive, and even more expensive for large problem sizes, than the generation on the device. Besides optimizing K1 other improvements can be done by overlapping the generation of the random numbers with the copying of the other inputs, and the allocation of the device memory just once per simulation and not every time step, as number of particles doesn't change.

Optimizations and performance obtained

By profiling the baseline implementation with nVidia's `computeprof` tool we obtained similar results to the ones presented in table 2, the $\mathcal{O}(N^2)$ kernel taking 75% of the total time of execution for $N \approx 1000$ and over 95% for larger N. As we didn't use shared memory, the traffic to global memory

was a concern, but it appeared that the reading access pattern - all threads accessing the same position and all positions in sequential order - and the writing of the results make good use of the caching mechanism, the hit rate of the L1 global memory cache being reported over 99%.

One of the possible difficulties in trying to get more performance was the number of registers per thread used. The `computeprof` reported a usage of 39 registers per thread which translates into less threads being able to run in parallel and thus a lower possibility that the time expensive memory accesses are overlapped by computation on the parallel threads. Even with this high use of registers the kernel is quite heavy, each thread doing many floating point operations compared to the memory accesses, so the lack of possible parallelism might be covered by the amount of work each thread does and the very high speed of the registers.

Another possible place where performance can be lost is warp divergence in the computation of the diffusion tensor. The Rotne-Prager-Yamakawa tensor Eq. (17) has different forms for overlapping and non-overlapping particles, which is dependent on the particle id, therefore the thread id in a block.

Performance is defined as the number of operations a computing device performs per second. By using the Rotne-Prager-Yamakawa diffusion tensor, the baseline implementation does 104 floating point operations per particle-particle interaction, out of which 60 are multiplications and divisions and 40 are additions. Also a sqrt is needed for computing the distance between the particles. This operation was implemented with the `rsqrtf()` as it is cheaper, and only counted as one operation even if in fact it is more expensive. This being said we must keep in mind that this analysis gives us an approximate for the actual performance.

A good way for analysing an implementation is by creating the roof-line model of the machine that it is ran on and comparing the measured performance to the maximum the machine can achieve. In order to build the roof-line model two measures are needed: the theoretical peak performance and the theoretical peak bandwidth of the global memory.

The theoretical peak performance is computed as

$$\text{Peak performance } = \#cores \times frequency \times 2, \tag{24}$$

as one multiplication and one addition can be processed in each clock cycle. For computing the peak memory bandwidth the following formula is used

$$\text{Peak bandwidth } = memory\ clock\ rate \times bus\ width/8 \times 2 \tag{25}$$

where the division by 8 converts from bits to Bytes and the multiplication by 2 is due to the double data rate of the GDDR5 type of RAM memory.

For Cyrus the theoretical peak performance is 1581.056 Gflop/s and theoretical peak bandwidth is 193.536 GB/s. In practice these values are very hard to reach because usually memory accesses and computations are not uniformly distributed in time, there is imbalance between additions and multiplications, memory access patterns are always optimal and many other factors that are neglected in the computation of the theoretical peaks. For having an idea what is the maximum achievable peak bandwidth we ran the `bandwidthtest` included in NVIDIA GPU Computing SDK that reported a bandwidth of 166.33 GB/s.

To compare the performance of our implementation to the roof-line we computed the operational intensity as

$$\text{Operational intensity } = \frac{\#operations}{\#off-chip\ memory\ accesses}. \tag{26}$$

For the computation of the Rotne-Prager-Yamakawa the number of operations varies with the problem size in the following way: #operations $= N[(N-1)*104+6]$ and the number of memory accesses in Bytes is $4N[15+9(N-1)]$. This way of computing the number of memory accesses (not taking into consideration the cache) gives an operational intensity of around 3.

In Fig. 1 we plotted the roof-line model for the Cyrus machine and the performance obtained with the baseline implementation for a size of 20480 and by running the kernel with 256 threads per block.

It is thus obvious that the yielded performance is larger then the peak for the computed operational intensity. This means that the caching actually increases the operational intensity, but not knowing how the cache works makes it hard to correctly compute the number of off-chip memory accesses. The ridge point is obtained at operational intensity 10, so if the caching brings the operational intensity above 10, then we could theoretically take advantage of the full computational power of Cyrus.

In order to improve the performance a series of methods have been tried. First we tried to minimize the use of registers by reusing the temporary variables needed for computing the hydrodynamic tensor as much as possible. Also the constant parameters were moved to constant memory. Another improvement was to move the instructions around in order to improve the instruction level parallelism. These changes improved the performance by up to 17%.

Another optimization we tried was to use the shared memory for the inputs. All threads in a thread-block were copying the inputs from global memory into shared memory and then were using these values from shared

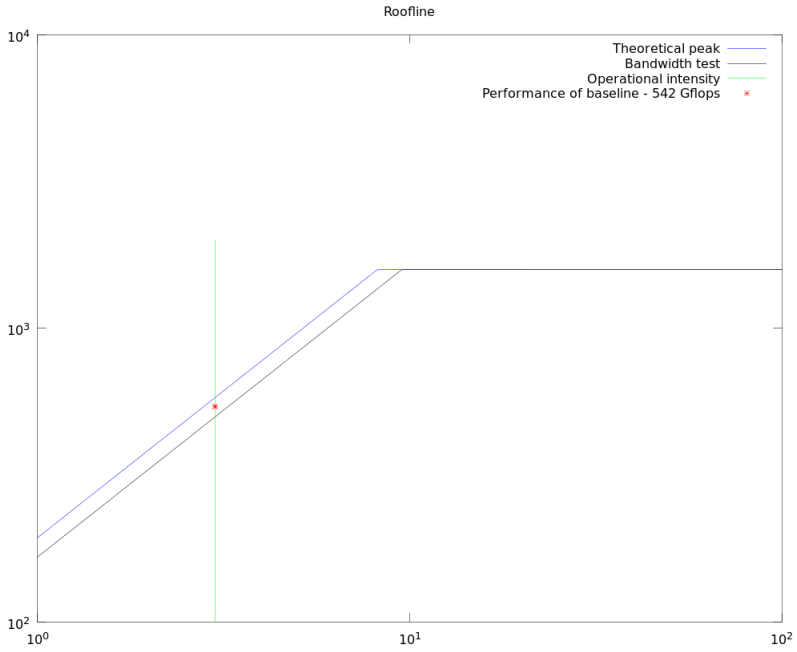

Figure 1: Roofline model of Cyrus with approximated operational intensity and performance obtained by baseline implementation

memory. The use of shared memory actually increased the number of used registers to 41 and decreased the performance by 10 to 15%.

We also changed the way the threads are used, by having one thread per component and not one thread per particle as before. Using this approach, the diffusion tensor had to be computed in shared memory and every three threads had to communicate the values computed for it. Besides duplicating a small part of the work, a couple of synchronisations were needed for the computation of the diffusion tensor. All this led to a dramatic decrease in performance.

Another important parameter to consider for performance measurements is the size of the thread blocks and the size of the problem. In table 3 we can observe how the performance is influenced by the size of the problem and the number of threads per block. For problem sizes up to 1000 the yielded performance is less then 10% of maximum peak performance, so a powerful GPU will get will not be well used for small simulations. For $N \leq 10000$ we obtain up to 40% of the maximum peak performance of Cyrus. We can also see that sizes of blocks that are a power of 2 give better results then other sizes, and that the performance increases with the block size up tu a point.

	Threads per block						
N	32	64	96	128	192	256	512
256	33.10	32.97	33.06	32.62	32.25	30.25	30.21
512	68.3	67.88	68	66.96	65.69	62.13	38.58
1024	138.14	137.95	138.10	135.57	132.212	126.04	77.96
2048	273.09	273.41	268.23	273.03	265.961	253.65	156.74
2560	339.72	334.29	335.75	312.94	332.865	317.49	196.14
5120	324.59	540.22	468.34	489.01	498.362	390.21	392.68
10240	430.82	507.54	572.44	615.91	525.64	528.97	392.49
20480	489.82	593.97	571.07	617.86	602.858	635.62	522.70

Table 3: Performance obtained on Cyrus by varying problem size and thread block size in Gflops.

By increasing the block size from 128 to 256 in the case of $N = 10240$ we see a performance drop by approx 15%. For larger N this drop is observed at larger block size.

By further increasing the problem size we observed that the obtained performance did not exceed 640 Gflops.

In conclusion, for large enough problems and well chosen thread block configurations a maximum of 40% of theoretical peak performance was obtained.

As the cell stretching experiment doesn't use such a large number of particles, we also tested if having multiple cells with HI only inside each cell would benefit from concurrent kernel execution. We launched K1 10 times in different streams and timed the execution of all calls. By running one kernel with $N = 1024$ and 256 treads per block we obtained 125.5 Gflops and by calling in 10 times with the same settings in concurrent mode we obtained 523.52 Gflops. So, by having multiple smaller size simulations we can still take advantage of the computational power of the GPU almost as with a single large one.

5 Results

5.1 Simulation setup

For validating the implementation we used the same simulation setups as it was used in [3]:

- First we used a simple system of two particles of the same radius connected with a harmonic spring – a dimer, the equilibrium distance of the spring being varied;
- Then a generalization of the simple system was used: longer chains of identical beads connected by harmonic springs. Also harmonic repulsive potentials were added for overlapping particles.

The harmonic potentials are described in Eq. (21) and Eq. (22). In both cases the spring constant was set to $a_h = 50k_BT/a^2$ and the time step was set to $\Delta t = 0.001$.

The parameters a, k_B and T were all set to 1 and η was chosen in such a way that $D_{ii} = 1$ for simplicity, as the results for the diffusion coefficient and the relaxation time are functions of the radius of a bead and the diffusion coefficient of a single bead, which equals D_{ii}. The parameters of k_B, T and η are irrelevant for this setup.

The dimer simulation was started with the two beads positioned on the x axis with the center of mass in the origin of the system and the spring in equilibrium.

The polymer measurements were started with the coiled polymers. In order to reach the coiled state for the polymers I used the same technique as in [3] and saved the coiled configuration for the rest of the simulations. The coiling simulations were started with the beads arranged in linear chains along the x axis with separation equal to the equilibrium distance of the springs and advanced until the radius of gyration reached equilibrium. The equilibrium was approximated by fitting the gyration radius by $\langle R_g^2 \rangle(t) = R_0 e^{-\alpha t} + c$ and taking c as the equilibrium value.

In figure Fig. 2 you can see the variation of the radius of gyration of a 50 beads polymer with time, the exponential fit and the approximated radius of gyration obtained with this method. In figure Fig. 3 you can see the snapshots of a stretched 50 beads polymer and of the coiled up version of it.

As Geyer suggested in [3], for the longer polymers (above 70 beads) we also tried to run the coiling simulations without HI included, but kept the time step to the value of $\Delta t = 0.001$. With this setup the equilibrium state of the radius of gyration for the 70-polymer was reached in 3 times more

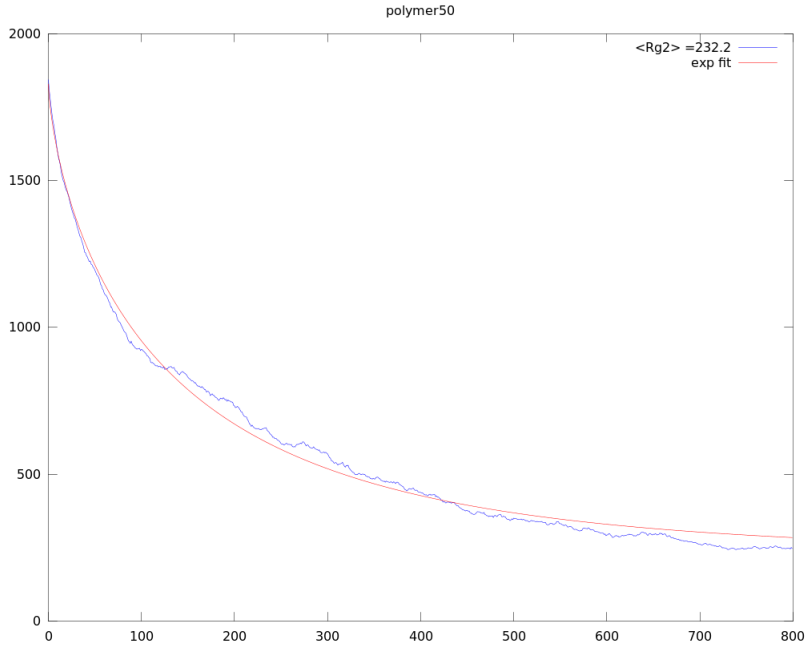

Figure 2: Evolution of the radius of gyration of a 50 beads polymer with time. The exponential fit decays to a constant that was chosen as the equilibrium radius of gyration of the molecule.

Figure 3: Snapshots of stretched and coiled up 50 beads polymers.

time steps than with HI included, so we can draw the conclusion that the hydrodynamic coupling, even if more computationally expensive, can help in applications like protein folding.

5.2 Validation measurements

In both cases the translational diffusion coefficient D_{CM} of the molecule was measured:

$$D_{CM} = \frac{\langle \Delta r_c^2 \rangle}{6\Delta t}, \tag{27}$$

where $\langle \Delta r_c^2 \rangle$ denotes the mean squared displacement of the center of mass and Δt is the simulation time.

Dimer dynamics

In the dimer case also the inverse of the rescaled relaxation time of the orientation of the dimer α_D was compared to analytical results. This value was obtained by studying the average value of

$$\cos\theta(t) = \hat{r}_0(t) \cdot \hat{r}_0(0), \tag{28}$$

where $\hat{r}_0$ is a unit vector from the center of one bead to the center of the other. The average value of $\cos\theta(t)$ decays exponentially with time $\langle \cos\theta(t) \rangle = e^{-\alpha_D t}$.

For averaging purposes a 10 million time steps long simulation war run, then split into windows of 10000 steps and in each window the squared displacement and $\cos\theta$ were compared to the position and orientation at the beginning of the window. The displacements and coses for each time step were averaged across all windows. For the squared displacements a polynomial of grade 1 was fit to the average values and for the coses an exponential.

The analytical values with this form of HI, as expressed by Geyer [3] are:

$$D_{CM} = \frac{D_0}{2}\left(1 + \frac{a}{d}\right), \tag{29}$$

$$\alpha_D = 4\left(1 - \frac{3a}{4d} - \frac{1}{2}\left(\frac{a}{d}\right)^3\right), \tag{30}$$

where d is the separation between the two beads. In the dimer simulations the average separation distance between the two beads was slightly larger then the equilibrium distance of the spring, so we used the computed average distance to compute the analytical values for D_{CM} and α_D.

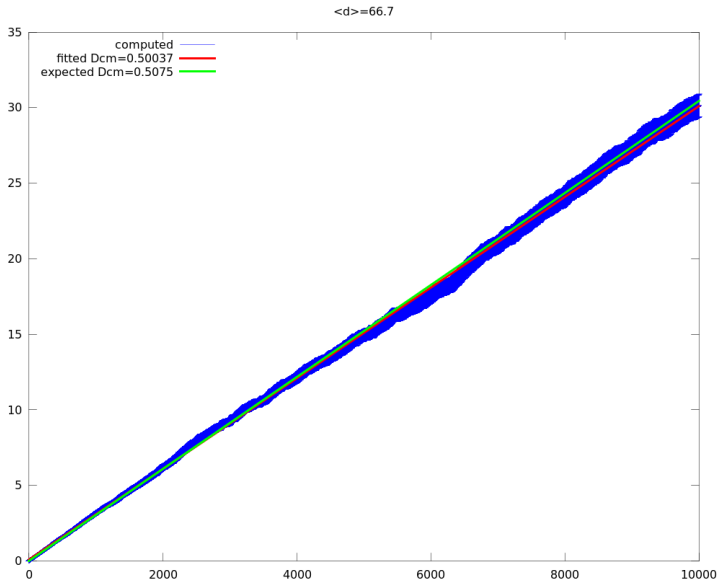

Figure 4: Variation of the mean square displacement of the dimer with time with errorbars.

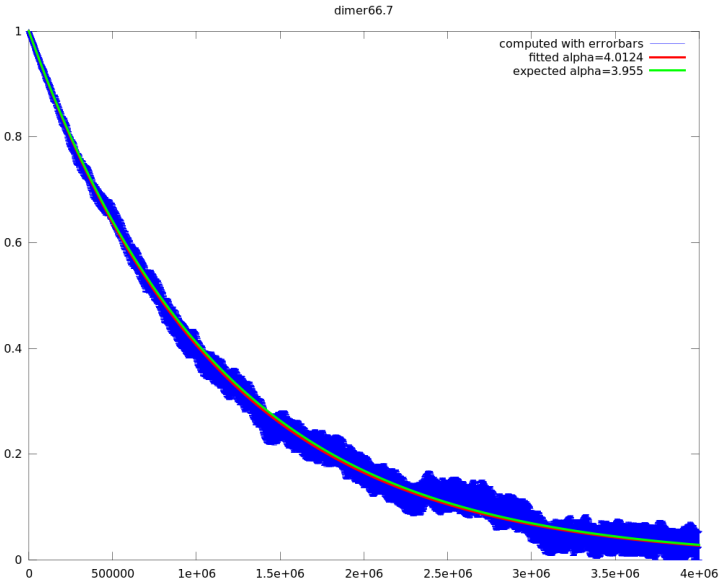

Figure 5: Variation of $\langle \cos\theta(t) \rangle$ with errorbars. The fitted exponential decay (red) and the expected value for α_D (green) are very close and within the measurement errors.

		D_{CM}		α_D	
L	$\langle d \rangle$	TEA-HI	Theoretical	TEA-HI	Theoretical
2	2.099	0.7239	0.7488	2.32	2.26
3	3.007	0.6504	0.6663	3.01	2.93
4	4.005	0.6107	0.6148	3.38	3.22
8	8.002	0.5510	0.5624	3.42	3.62
20	20.001	0.5137	0.525	4.01	3.85
66.7	66.7	0.4969	0.5075	4.01	3.96

Table 4: Comparison of the center of mass diffusion coefficient D_{CM} and the rescaled correlation time α_D of a dimer obtained with simulations with HI and their theoretical values.

As seen in table 4, the simulation results reproduce the theoretical values quite well. We must mention that these results are very sensitive to how the fitting is done and to how many windows of 10000 time steps are averaged. Also as it can be seen in Fig. 5 and Fig. 4 the linear and exponential fits are very close to the expected values and also within the sampling errors for the number of simulation windows used. To further improve the fittings, much longer simulations should be run, especially for the relaxation time of the orientation for large separation.

Bead-spring polymers

For the polymer case the variation of the following parameters versus the chain length was recorded: the diffusion coefficient D_{CM}, the average radius of gyration $\langle R_g^2 \rangle$ and end-to-end distance $\langle R_{ee}^2 \rangle$ and the relaxation time τ_{corr} of the autocorrelation function of the end-to-end vector.

The values of $\langle R_g^2 \rangle$ and $\langle R_{ee}^2 \rangle$ are given by

$$\langle R_g^2 \rangle = \frac{1}{2N^2} \sum_{ij} \langle r_{ij} \rangle, \tag{31}$$

and

$$\langle R_{ee}^2 \rangle = \langle (\vec{r}_N - \vec{r}_1)^2 \rangle. \tag{32}$$

and as pointed out by Geyer and Winter, the theoretically predicted scaling behaviour is

$$\langle R_g^2 \rangle \sim \langle R_{ee}^2 \rangle \sim (N-1)^{2\nu}, \tag{33}$$

with $\nu \approx 0.588$ in a good solvent.

The diffusion coefficient of the center of mass of the polymers and the relaxation time τ_{corr} were computed in a similar way as in the dimer case:

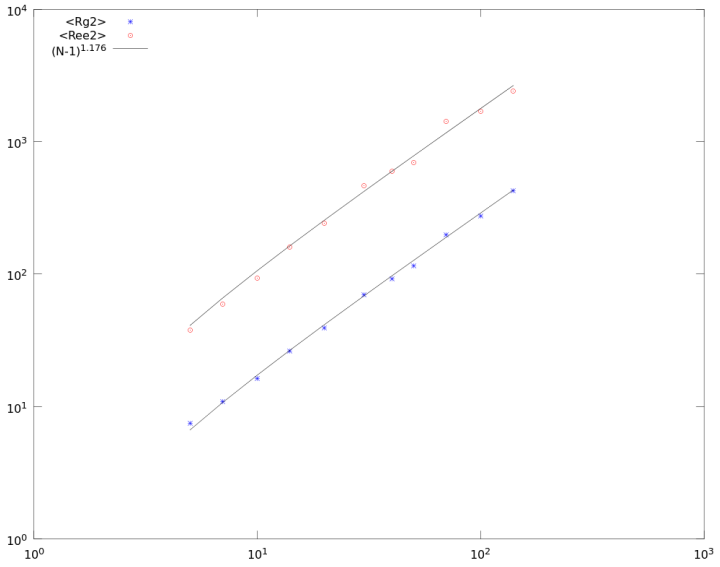

Figure 6: $\langle R_g^2 \rangle$ and $\langle R_{ee}^2 \rangle$ for various polymer chain sizes against the expected $(N-1)^{1.176}$ scaling behaviour.

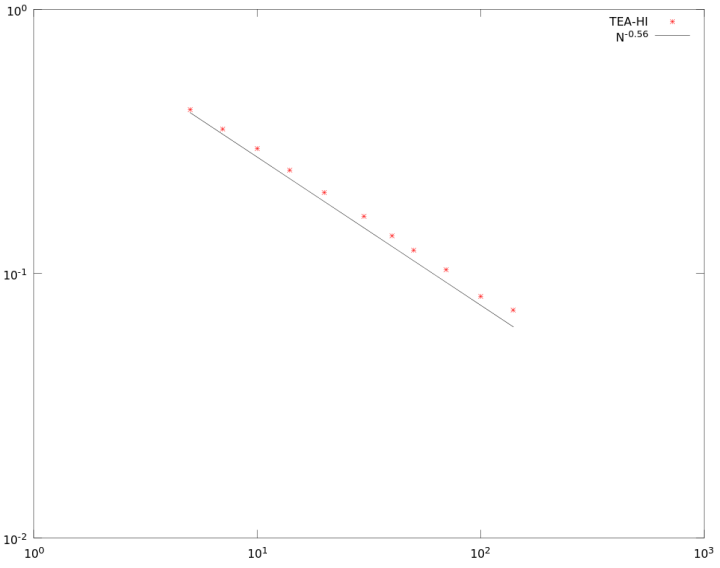

Figure 7: Diffusion coefficient of the center of mass for various chain length polymers.

a long simulation was run and then split in 10000 time steps windows for the diffusion coefficient and longer windows depending on the length of the chain for the relaxation time. The squared displacement and orientation of the end-to-end vector were compared to the position and orientation at the beginning of the window and averaged through all the windows. After averaging, a line was fitted to the mean squared displacement retrieve D_{CM} and a stretched exponential was fitted to the autocorrelation function of the end-to-end vector $\langle \vec{R}_{ee}(t) \cdot \vec{R}_{ee}(0) \rangle = e^{-(t/\tau_{corr})^{\beta}}$

In Fig. 6 we can see that the obtained values for $\langle R_g^2 \rangle$ and $\langle R_{ee}^2 \rangle$ follow the expected scaling. In Fig. 7 we can see that the diffusion coefficient variation with N can be fitted well with $N^{0.56}$ just like Geyer and Winter observed.

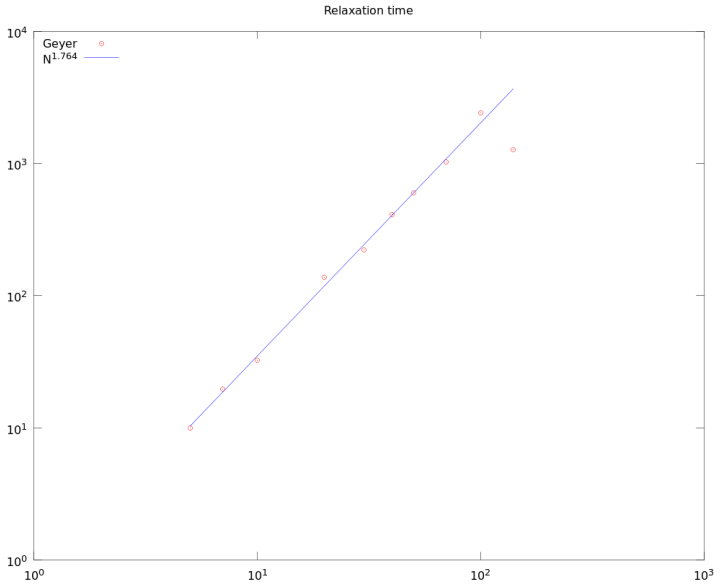

Figure 8: Relaxation times τ_{corr} of the autocorrelation function of the end-to-end vector $\vec{R}_{ee}$ of polymers of various chain lengths N. The line indicate the scaling $N^{3\nu}$ with $\nu = 0.588$.

In Fig. 8 we plotted the relaxation time τ_{corr} for various chain lengths. As we can observe τ_{corr} fits well the expected scaling. For large chain sizes, the fit is worse because the simulation was not long enough for a good exponential fit.

Considering that the results for the simulation of both dimer and polymers closely follow the results presented by Geyer and Winter in their work, we can conclude that the implementation of the HI algorithm is correct.

5.3 In-silico cell stretching

For the cell stretching experiment we modelled a single cell between 2 rigid walls. The cell was modelled with 1000 sub-cellular elements with pairwise potentials as described in Eq. (23). Together with these interactions also hydrodynamic interactions were added to the simulation. One of the walls was fixed and the other was used to apply a constant stretching force to the cell, and the cell strain evolution with time was recorded. A snapshot of the cell before the stress is applied can be seen in Fig. 9.

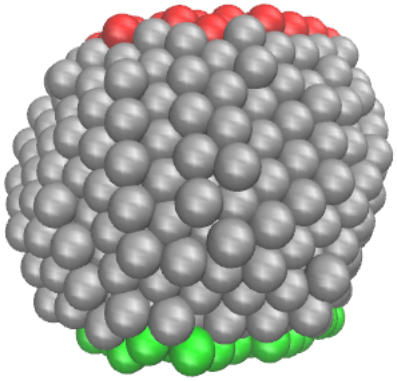

Figure 9: Snapshot of a relaxed cell. The bottom layer of elements are connected to a fixed wall and the top layer are connected to a mobile wall to which a constant force is applied.

In all the simulations we stretched the cells for 7 seconds and then released the stress.

Without HI the cell strain has the behaviour depicted by the plot in Fig. 10. By applying higher stresses, the top layer of sub-cellular elements breaks off from the cell.

When including HI, there are 2 parameters to consider: the radius of the elements a and the viscosity η of the solvent. We chose a as $1/3$ of the equilibrium distance of the pairwise potential and varied η in such way that we are close to replicating the damping of the BD simulation in the absence of HI.

In Fig. 11 we compared the strain evolution in time for different values of η. By comparing to the simulation without HI and the ones with HI we can see that the equilibrium elongation is higher with HI included independently

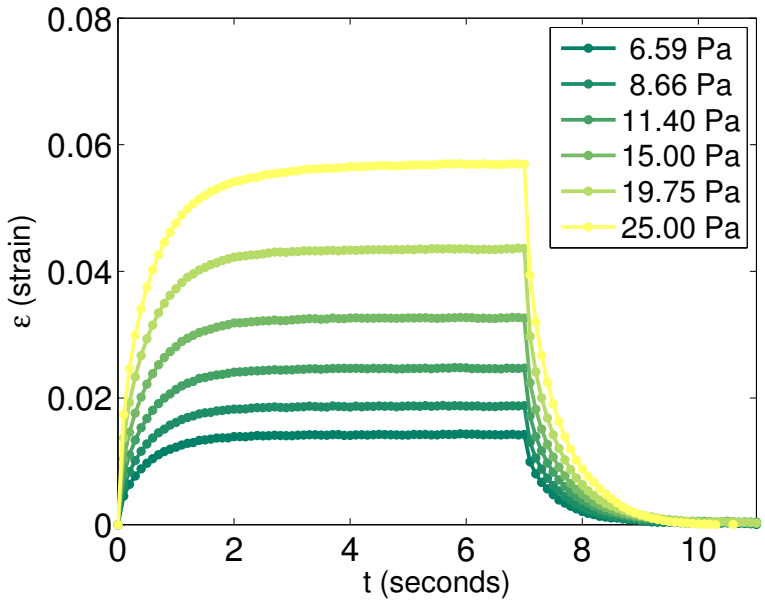

Figure 10: Cell strain versus time for different stresses without HI included.

of η and is reached faster for lower viscosity. Also we can observe that after the stress release, the cell doesn't fully recover and the deformation is larger for larger values of η.

If we further increase the stress the cell starts behaving differently. Up to some point in time after applying tension it resists deformation and after this point it cannot resist the strain and it stretches until it breaks. This delayed failure behaviour resembles the one of colloidal gels [9].

In Fig. 12 we plotted a comparison between the cell strain stretched without HI at 25 Pa and stretched with HI at 35 Pa. After 1.5 seconds we can see the second type of response to the strain. In Fig. 13 and Fig. 14 we took snapshots of the cell showing the second type of behaviour.

We introduced HI in the simulation in the hope of reproducing the results of the living cell stretching experiments of Desprat *et all* [1]. The analysis revealed that just adding HI is not enough for reproducing the short time regime they proposed. However we observed an underlying slow evolving response that becomes major after the stress is applied long enough.

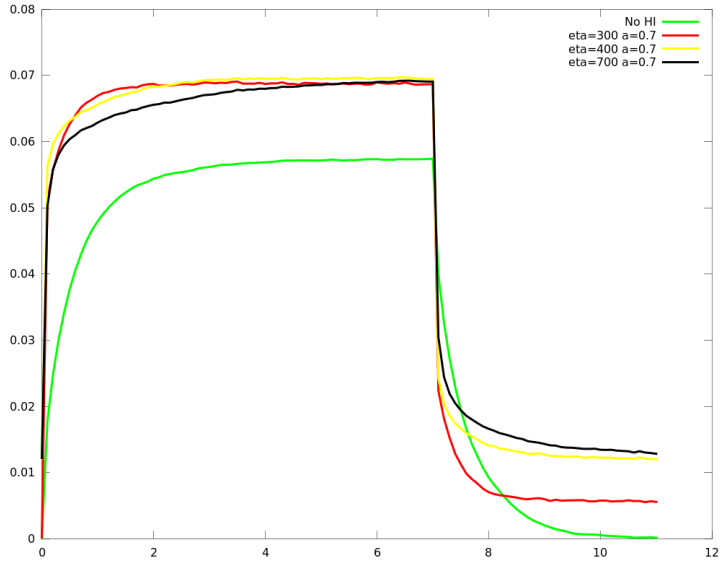

Figure 11: Cell strain versus time without HI and with HI for different values of η and the same stress of 25 Pa.

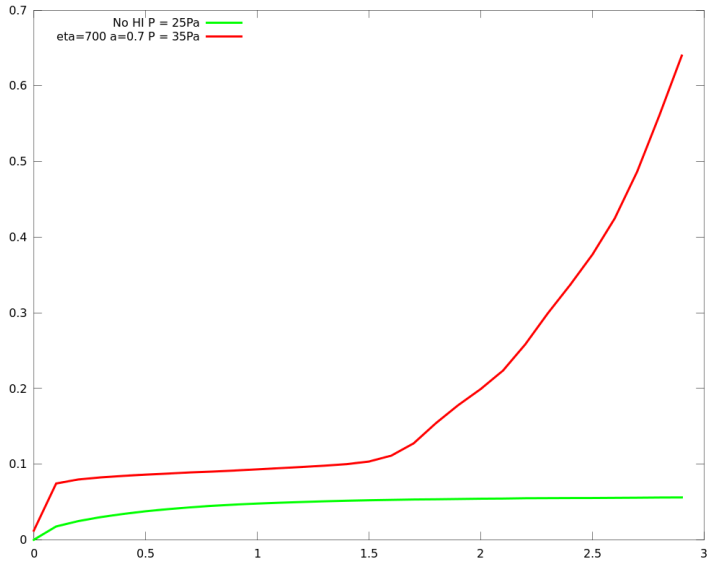

Figure 12: Cell strain versus time without HI and with HI for $\eta = 700$ and the stress of 35 Pa.

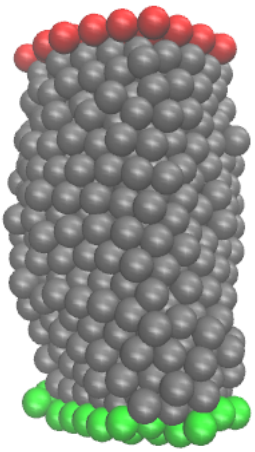

Figure 13: Snapshot of a stretched cell at 35 Pa at 3 seconds.

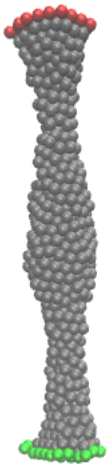

Figure 14: Snapshot of a stretched cell at 35 Pa at 4.5 seconds.

6 Future work

Further work in the field could be directed towards:

- adding other types of interactions between the sub-cellular elements, or other methods for modelling cells could be developed;
- analysing more in depth how the slow response depends on the parameters of the simulation (a, η, stiffness of springs;
- increasing performance;
- modelling tissue;
- using HI for other systems;

For the performance development for the GPU another method can be tried: computing the hydrodynamic tensor in one kernel and storing it, and then in different kernels using the hydrodynamic tensor to compute the needed parameters for the effective force and displacement of the particles. Also the CPU implementation can be improved for computing machines that don't have a GPU. These improvements could also take into consideration the size of the problems for which the code will be used and have a smart way of computing the HI for different cells simultaneously - for example making use of the simultaneous kernel execution ability of new GPUs.

For limiting the simulation time, an implementation that computes HI only for inter-particle distances smaller then a cutoff can be devised.

Another point of interest is the modelling of multiple cells. HI can be added inside a cell and between cells and interactions in such systems can be studied.

HI can be added to other systems. For instance blood is a fluid that contains large cells (red cell, white cells...) suspended in water. The interaction between such particles and with the blood vessel walls can be studied.

As observed in the polymer simulations, HI can be used for folding protein simulations with water playing an important role when creating the structure of the molecule.

7 Conclusions

We have build a system through which HI can be added to molecular dynamics simulations. It was integrated in LAMMPS which is a broadly used code for these types of simulations.

A CPU and a high performance GPU version have been developed, with the GPU one being able to use up to 40% of the maximum peak performance of the machine used for testing. Even if this value is reached for big problem sizes, there are possibilities of running more smaller sized problems concurrently on the same GPU, reaching up to 30% of the peak.

Two different types of systems have been simulated: polymer diffusion and cell stretching. For the polymer diffusion experiments the scaling behaviour of the radius of gyration and end-to-end distance with the number of beads was compared with theoretical values and properly recovered. The scaling of the diffusion coefficient of the center of mass of the polymers with the chain length is 95% of the theoretical value, but it is consistent with the results reported by Geyer and Winter in their work [3].

In the cell stretch experiments, the addition of HI slightly changes the behaviour of the cell if the force is not big enough to break the cell. In this case the observed equilibrium elongation is larger then the one obtained without HI and after the stress is removed the cell does not regain its initial size, remaining deformed. Moreover, if the applied stress is larger then a certain threshold, the cell resists for some time and then deforms to strain sizes similar to the ones observed in living cell stretch experiments [1].

Our final conclusions lead us to believe that the HI within cells change the cell strain response to stress significantly. Nevertheless, adding HI does not fully replicate the real cells so there is still a need to build other models of the cell to better simulate the real world.

References

[1] Nicolas Desprat, Alain Richert, Jacqueline Simeon, and Atef Asnacios. Creep function of a single living cell. *Biophysical Journal*, 88(3):2224 – 2233, 2005.

[2] Donald L. Ermak and J. A. McCammon. Brownian dynamics with hydrodynamic interactions. *The Journal of Chemical Physics*, 69(4):1352–1360, 1978.

[3] Tihamer Geyer and Uwe Winter. An $\mathcal{O}(n^2)$ approximation for hydrodynamic interactions in brownian dynamics simulations. *The Journal of chemical physics*, 130(11):8, 2008.

[4] Robert Groot and Patrick Warren. Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation. *The Journal of Chemical Physics*, 107(11):4423–4435, 1997.

[5] JG Kirkwood and J Riesman. The intrinsic viscosities and diffusion constants of flexible macromolecules in solution. *Journal of chemical physics*, 16(6):565–573, 1948.

[6] T. J. Newman. Modeling multicellular systems using subcellular elements. *Mathematical biosciences and engineering : MBE*, 2(3):613–624, July 2005.

[7] Jens Rotne and Stephen Prager. Variational treatment of hydrodynamic interaction in polymers. *The Journal of Chemical Physics*, 50(11):4831–4837, 1969.

[8] Sebastian A Sandersius and Timothy J Newman. Modeling cell rheology with the subcellular element model. *Physical Biology*, 5(1):015002 (13pp), 2008.

[9] Joris Sprakel, Stefan B. Lindström, Thomas E. Kodger, and David A. Weitz. Stress enhancement in the delayed yielding of colloidal gels. *Phys. Rev. Lett.*, 106:248303, Jun 2011.