Background

Electrophoretic experiments on Polyelectrolytes (PEs) are of great importance in the field of molecular dynamics. The studies can be extrapolated and techniques can be applied to protein folding, DNA separation, molecular separation, etc. which are very crucial inputs to more complex technologies. For example, molecular separation is a key technology for future miniature devices.

Algorithm by Netz

The paper by Netz [1] assumes presence of both, PE monomers and counterions, and a strong coupling between the two. It analyzes both the static (zero electric field) and dynamic (non-zero electric field) non-equilibrium behaviors of the PE sequence and develops scaling arguments for both. The PE monomers and counterions condense together in the former case while the PE aligns along the direction of electric field in the latter.

The experimental set-up consists of a single PE with N monomers in a cubical box of length L along with N oppositely charged counterions of the same valency (q) and unity radius. The box volume L^3 corresponds to the inverse PE concentration and plays an important role. The position Langevin equation is discretized over time step Δ to determine the velocity (r) of the i^{th} particle at any discrete time instance t:

$r_i((t/\Delta)+1) = r_i(t/\Delta) - \mu_o \nabla_{ri} U(t/\Delta) + \mu_o q e S_i E_f + \sqrt{6\mu_o \xi_i(t/\Delta)}$, where

μ_o – bare particle mobility
U – dimensionless (i.e. scalar) potential energy
e – single coulomb's charge
S – +/-1 for Monomers/Counterions respectively
E_f – Applied electric field
ξ_i – Vectorial random force with unit variance, acting on the i^{th} particle

The dimension-less potential energy has several contributions: $U = U_{nn} + U_{LJ} + U_C$, where

U_{nn} = Connectivity component of potential energy
U_{LJ} = Lennard Jonnes component of potential energy
U_C = Coulombic component of potential energy

The connectivity of the PE is ensured by $U_{nn} = K\Sigma_{(i,j)}(|r_i-r_j|-2)^2$ where the sum runs over nearest neighbors of the PE chain only. The bond stiffness is K=100 which gives a very narrow distribution of bond lengths.

Collapse of counterions and charged monomers is prevented by a truncated Lennard-Jones term acting between all particles in the simulation: $U_{LJ} = \varepsilon\Sigma_{(i<j)}((2^{12}/(r_i-r_j)^{12})-(2^7/(r_i-r_j)^6)+1)$ used for separation $|r_i-r_j|<2$ only with an energy parameter ε=1.

The coulombic part is $Uc = \Xi\Sigma_{(i<j)}(S_iS_j/|r_i-r_j|)$, where $\Xi=q^2l_B$ is the coupling strength and measures the ratio of the coulombic interaction and the thermal energy at a typical distance l_B which is the Bjerrum length in water.

Equilibration takes roughly 10^6 time steps and therefore simulations are run for at least 10^7 time steps.

In the absence of electric field and presence of a very small coupling between the monomers and counterions, the PE sequence resembles a neutral polymer since the electrostatic repulsion between monomers is very small. As the coupling increases, the monomer-monomer repulsion leads to a more swollen configuration (the standard PE effect). However, as the coupling further increases, counterions condense on the PE, decrease the repulsion between monomers and the PE starts to shrink. Finally, at very large electrostatic coupling, the PE is collapsed to a close-packed, almost charge-neutral condensate which contains most of its counterions.

When the PE is subjected to external electric field, the electric field induces motion in monomers and counterions, thus dissipating energy and causing a nonequilibrium situation. The simulations in the paper show that linear-response theory describes the induced dipole moment of a condensed PE globule in an electric field quantitatively up to a critical field strength at which the PE unfolds and orients in the direction of the field. This nonequilibrium unfolding transition occurs at a polarization energy equivalent to approximately thermal energy.

The Langevin Equation is decomposed (∇ operator) into three components (viz. X, Y and Z) in order to calculate the potential energies of each particle in three directions. These energies are then used to update the three co-ordinates of each particle at the end of every time-step. Eg., for the x xo-ordinate:

$U_{nn\text{-}x} = K\Sigma_{(i,j)}[-2 *(|x_i\text{-}x_j|)*(|r_i\text{-}r_j|\text{-}2)/ (|r_i\text{-}r_j|)]$
$U_{LJ\text{-}x} = \Sigma_{(i<j)}[3* (|x_i\text{-}x_j|)*((2^{14}/|r_i\text{-}r_j|^{14})\text{-}(2^8/|r_i\text{-}r_j|^8))]$
$U_{C\text{-}x} = \Xi\Sigma_{(i<j)} [S_i*S_j*|x_i\text{-}x_j|/|r_i\text{-}r_j|^3]$
$X_{i\text{-}updated} = \mu_o*(U_{nn\text{-}x} + U_{LJ\text{-}x} + U_{C\text{-}x} + \text{Gaussian-Noise} + E_f*q_i)$ (Assuming, E_f is applied in the x-direction)

The integrated Langevin equation itself is used to compute the net potential energy of individual particles. Energies of all particles add up to give the total energy of the system at the end of every time-step.

$TE_i = \Sigma_n(U_{nn} + U_{LJ} + U_C)$.

Fig.1 summarizes the algorithm in [1] for every time-step. The same is repeated for 10^6- 10^7 time steps. The per time-step procedure can be divided into 4 major parts with time complexities of the order of O(N), $O(N^2)$, $O(N^2)$ and O(N) respectively. The insignificant 5^{th} part comprises of miscellanious tasks like file reading, memory allocations, writing outputs to the file, etc. Fig.2 shows the computational intensity of the 5 parts for a PE of length 256 over 10^3 time-steps. Though both parts 2 and 3 execute particle-particle simulation, part 2 takes much more time owing to more complex mathematical functions and extra computations coming from branching to check $|r_i\text{-}r_j|<2$. The 4 parts constitute almost 100% of the running time while parts 2 & 3 consume close to 90% of the running time justifying the need for parallelization. The algorithm is a good candidate for parallelization on many-cores because of it's largely SIMD nature (in a single time-step, every particle can be computed upon independently) and typically high input size. GPU becomes an obvious choice because of the availability of large number of processing cores. A parallel implementation (as explained in the sections ahead) was done on Nvidia's GTX280 GPU with frequency =1296MHz of the unified processors.

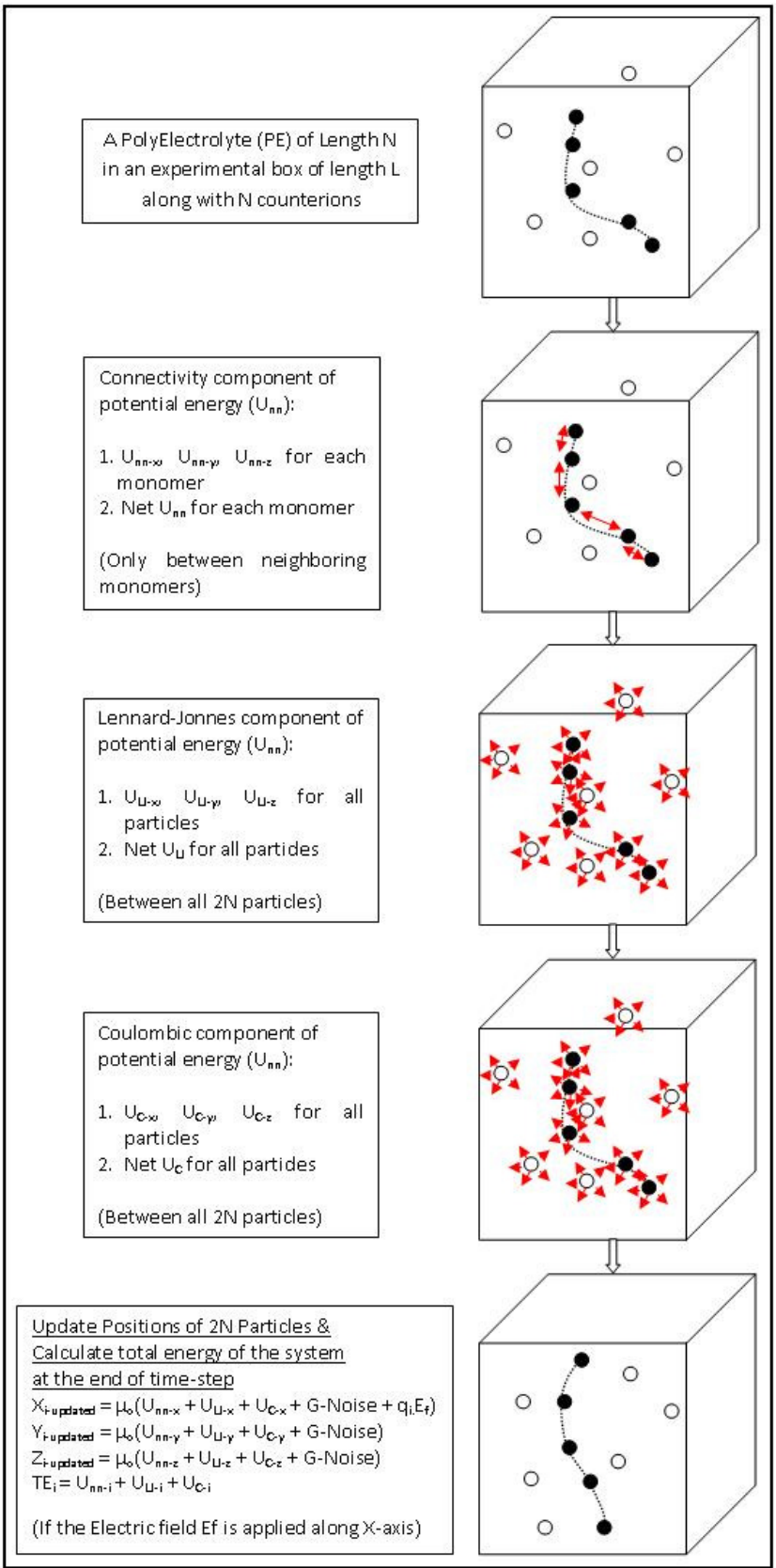


Fig.1 Flow diagram of the algorithm by Netz in [1]

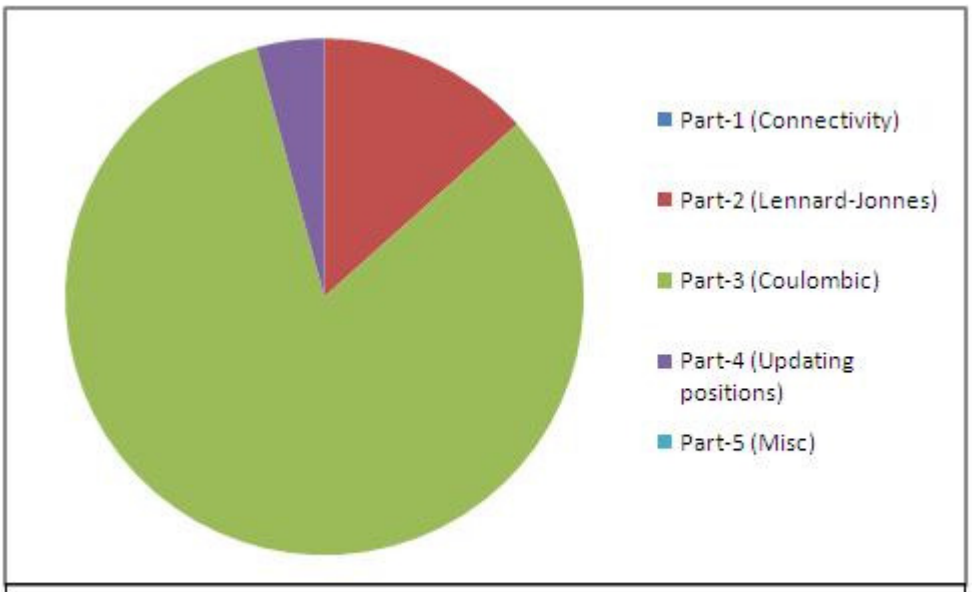


Fig.2 Computational time profiled for algorithm in [1]

Parallelizing on the GPU

The system of PE under study is very data-parallel for 2N particles. The CUDA implementation can be designed to evenly distribute work among threads such that each thread accesses only one particle and thus most computations can be performed inside a single kernel without the need of any global synchronization. However, this cannot happen due to practical considerations.

In order to take advantage of the many-cores on the GPU, the algorithm in [1] needs to be analyzed for parallelizing.

COMPUTATIONS:
The updated co-ordinates at the end of a time-step become the input to the next time-step. Thus strict data-dependency exists between successive time-steps. Within a time-step, the following need to be computed and can be parallelized in the following way:

1. Connectivity component of potential energy of every monomer, in the X, Y and Z directions, needs to be computed using decomposed Langevin equation. Though every monomer interacts with its neighboring particles for this computation, it can be calculated independently for every monomer though input data of the neighboring monomers needs to be available. Thus, as many number of threads as the number of monomers may be launched to perform the computation in parallel.
2. Computation of the net connectivity component of potential energy for every monomer using the Langevin equation. It can be calculated independently for every monomer though input data of the neighboring monomers needs to be available. Thus, as many number of threads as the number of monomers may be launched to perform the computation in parallel.
3. Lennard-Jonnes component of the potential energy, of monomers and counterions, in the X, Y and Z directions, needs to be computed using decomposed Langevin equation. Though every

particle interacts with every other particle, it can be calculated independently for every particle though input data of other particles needs to be available. Thus, as many number of threads as the total number of monomers and counterions may be launched to perform the computation in parallel.

4. Computation of the net Lennard Jonnes component of potential energy for every particle using the Langevin equation. It can be calculated independently for every particle though input data of the neighboring particles needs to be available. Thus, as many number of threads as the total number of monomers and counterions may be launched to perform the computation in parallel.
5. Coulombic component of the potential energy, of monomers and counterions, in the X, Y and Z directions, needs to be computed using decomposed Langevin equation. Though every particle interacts with every other particle, it can be calculated independently for every particle though input data of other particles needs to be available. Thus, as many number of threads as the total number of monomers and counterions may be launched to perform the computation in parallel.
6. Computation of the net Coulombic component of potential energy for every particle using the Langevin equation. It can be calculated independently for every particle though input data of the neighboring particles needs to be available. Thus, as many number of threads as the total number of monomers and counterions may be launched to perform the computation in parallel.
7. Updating co-ordinates of each monomer and counterion in the X, Y and Z directions. It can be done independently for every particle. Thus, as many number of threads as the total number of monomers and counterions may be launched to perform the computation in parallel.
8. Total potential energy of each particle is computed by adding up net Connectivity, Lennard-Jonnes and Coulombic components of potential energies for that particle. It can be done independently for every particle. Thus, as many number of threads as the total number of monomers and counterions may be launched to perform the computation in parallel.
9. Total energy of the system = sum total of net potential energies of all the particles. This can be computed only in series since threads for different particles cannot write simultaneously to a single variable.

MEMORY:
In the brute-force method, for 2N particles (N monomers and N counterions), a lot of memory would be required to store the inputs, intermediate data and the output.

1. 2N input elements need 6N storage to store co-ordinates of each particle in X, Y and Z directions.
2. 2N input elements need 6N storage to store the Connectivity of potential energy of each particle in the X, Y and Z directions.
3. 2N input elements need 6N storage to store the Lennard-Jonnes component of potential energy of each particle in the X, Y and Z directions.
4. 2N input elements need 6N storage to store the Coulombic component of potential energy of each particle in the X, Y and Z directions.
5. 2N input elements need 6N storage to store the net Connectivity, Lennard-Jonnes and Coulombic components of potential energy of each particle.
6. 2N output elements need 6N storage to store co-ordinates of each particle in X, Y and Z directions.
7. 2N output elements need 2N storage to store total energy of each particle.

Thus, it would require 38N storage for a system of 2N particles. A system of 1000 PEs (i.e. 1000 N) will need storage for 38000 elements of 4 bytes each (assuming single precision floating point data) i.e.

149KB of storage space. A GTX 280 with <= 1GB of global memory space will thus allow a computation (between two sets of data-transfers between host and device) on a maximum of 7061 monomers. The scalability and efficiency of this design can be improved by re-using memory for storage.

MEMORY RE-USE:

1. 2N input elements need 6N storage to store co-ordinates in X, Y and Z directions. At the end of every time-step, X, Y and Z co-ordinates of every particle need to be updated and then fed as input to the next time-step. Once the output of one time-step is computed, the input is no more required. Thus, the ouput can easily replace the input itself. Thus only 6N storage is sufficient to stote the input and output co-ordinates in X, Y and Z directions.
2. 2N input elements need 6N storage to store the Coulombic component of potential energy of each particle in the X, Y and Z directions. Lennard-Jones component and Connectivity of potential energy of each particle in the X, Y and Z directions can be added to the same respective location if calculated one after the other. Thus only 6N storage would be required to store the intermediate results of potential energy in each direction.
3. 2N elements need 2N storage to store total energy of each particle. The 3 components of the total potential energy viz. Connectivity, Lennard-Jonnes and Coulombic, can be added to the same location.

Thus, a requirement of 38N storage is reduced to 14N making the system available for over 18900 monomers in one go.

Based on the above parallelizing and memory considerations, the algorithm has been implemented on a GPU using CUDA in the following way:

Computations 1, 3 and 5 above are mapped on the GPU over three kernels (kernels 1, 2 and 3 respectively). Computations 2, 4 and 6 can be clubbed with the computations 1, 3 and 5 above because of similar computational requirement. Computations 1-2 and 3-6 are kept in separate kernels since one set requires N threads (for N monomers) while the other set requires 2N threads (for N monomers and for N counterions). Theoretically, computations 3 to 6 can be clubbed in a single kernel. But in order to re-use the memory to store the outputs of these computations (as suggested in step-2 of memory re-use), synchronized read-after-write is required between computations 3 and 5. Hence, these steps are split into separate kernels. We would naturally think of clubbing computations 3 and 4 in kernel-2. However, since computations 4 and 6 share a lot of operations, they are clubbed with computation 5 in kernel-3. In order to re-use memory as suggested in step-3 of memory re-use, computation 8 is split into three parts, viz. for Connectivity, Lennard-Jonnes and Coulombic components, executed in kernels 1, 2 and 3 respectively. The results of each of these sub-computations are stored in the same location as the net potential energy of each particle. Computation 7 is carried out in a 4^{th} kernel to ensure synchronized read-after-write after all the energy components are calculated in kernels 1, 2 and 3. Computation 9, if computed on GPU, would have required reduction addition in either the same kernel or multiple kernels causing a lot of resource wastage or overhead respectively. Instead, since a lot of host-device bandwidth is available (along with transfering updated co-ordinates back to the host), the array storing total energies of all particles is also sent back to the host. These energies are then added in series on the CPU to give the total energy of the system.

Fig.3 summarizes the GPU implementation of algorithm in [1] for every time-step.

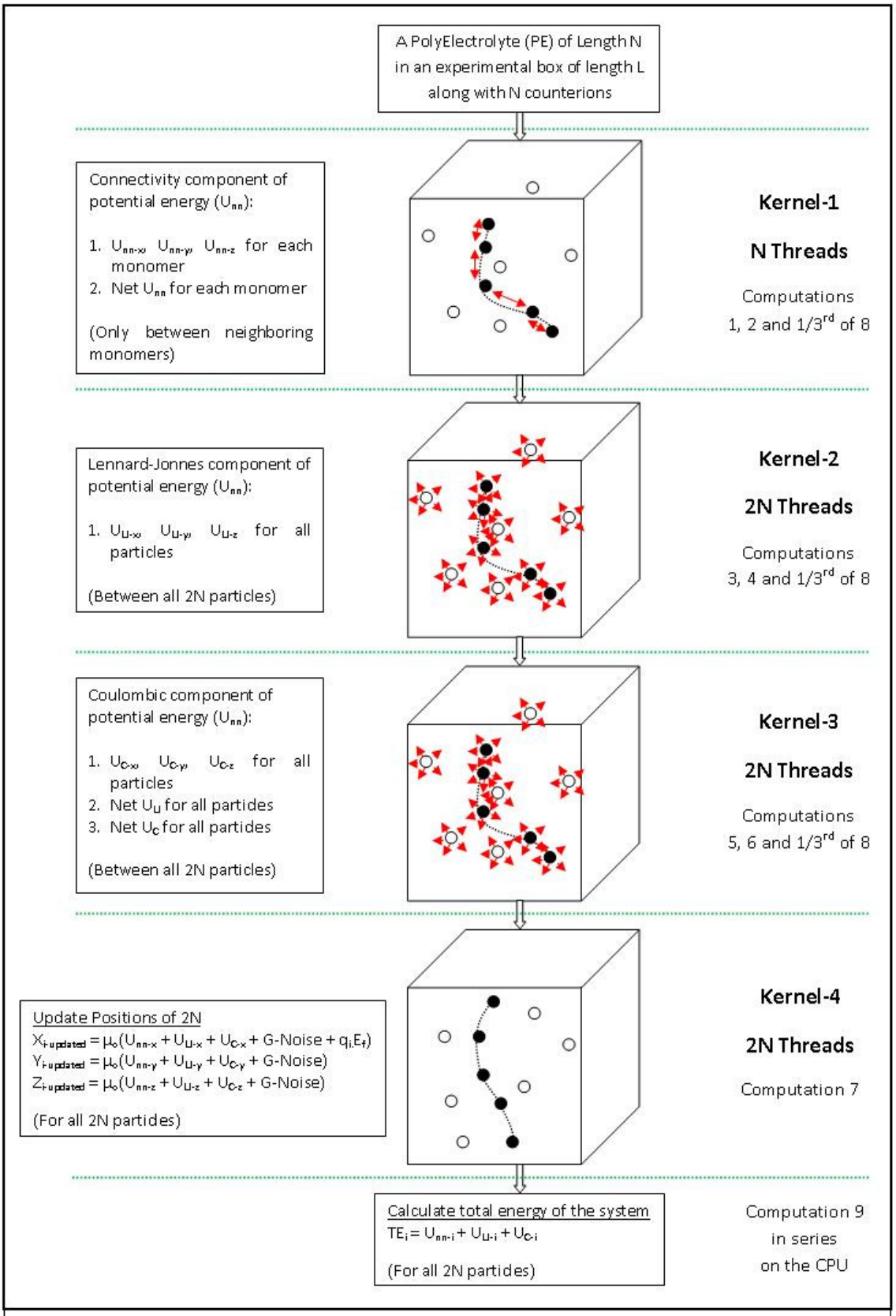
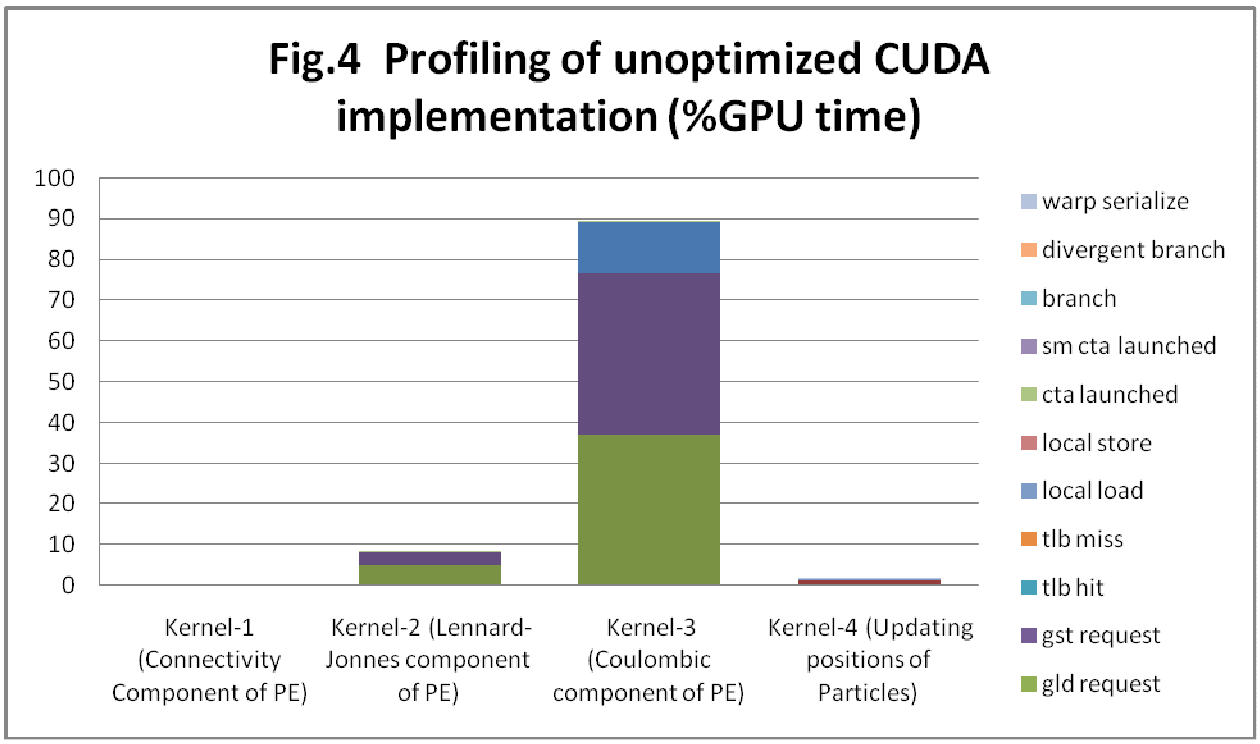


Fig.3 Parallel implementation of algorithm [1] on GPU

Optimizations

Fig.4 shows profiling of the un-optimized CUDA implementation. The counter values have been shown as percentage contributors to the total time on the GPU. As is seen, gld32, gst32 and gld64 constitute 97% of the time with more than 87% of it spent in kernel-3 alone. Memory read (gld) is close to 87% of all memory accesses for obvious reasons. Fig.4 when compared to fig.2, it is seen that kernel-2 and kernel-3 times almost reverse from serial to parallel implementation. This is because of three-fold reasons. One, the net Lennard-Jonnes component' calculation has been moved to kernel-3 in the parallel implementation. The branching in kernel-3 is far more than in kernel-2. Kernel-2 uses many more registers, leading to faster access, and reduced global memory access compared to kernel-3.



The implementation was not required to be a much optimized one. Hence only some basic optimizations have been carried out towards memory accesses. They are explained ahead. Few other optimizations that can be carried out in future have also been discussed ahead. Optimizations not applicable have also been mentioned. The following discussion follows the recommendations in ch.5 of [2].

Maximize Memory Throughput

The obvious optimization is to use a low-access-latency memory like shared memory. However, most of the computations in [1] being massively independent and parallel, most threads need only self data. Hence, use of registers to store repeatedly used thread-specific inputs/intermediate results/outputs, is a faster option than using shared memory. Hence, in all the four kernels, all repeatedly used inputs, intermediate results and outputs are deliberately stored into thread-personal registers. The register usage rises by 31% giving an approximately 17% rise in the speed-up for a range of input sizes.

In the kernels 2 and 3 based on particle-particle (n-body) simulation, limited use of shared memory has been made. Input data of a certain number (multiple of block-size) of particles is pulled into shared memory. Each thread of the block (i.e. each particle of the block) computes interaction with each of these particles in the shared memory. Once done, next set of particles are pulled into shared memory and so on. The percentage of particles (monomers and counterions) that can be pulled into shared memory per block decides the benefit gained. As in the current implementation, because of increased register usage, only 64 particles are being pulled into shared memory per block. Hence, performance benefit for smaller input sizes is good while for larger input sizes is negligible. Also, the register usage rises by an additional 7% giving an approximately 4% additional rise in the speed-up for a range of input sizes.

A comparison was made between two schemes: one with reduced registers and increased number of particles in shared-memory and the other with increased registers and reduced particles (64 per block) in shared memory. The latter scheme out performed the former because registers being faster-access' than shared memory. Hence the latter has been retained. This scheme resulted in 2 to 5% rise in speed-up depending on the input size. Also, profiler results show negligible serializatin suggesting correct use of shared memory in the implementation.

Another basic optimization, that has been implemented, is to limit the input and output data exchanged between the host and the device. Only the initial positions of the monomers and counterions (only at the begining of the first iteration) are sent on to the device as input. After all the required time-steps, only the net potential energies of individual particles are sent back to the host as output.

The effective bandwidth for the unoptimized version for say N=4096 for 10^3 iterations will be = (2*4*2*4096*4*10*1000)/(10^9*9.61) = 0.273 GB/s and for the optimized version will be (2*4*2*4096*4*10*1000)/(10^9*8.08) = 0.324 GB/s. Thus an improvement in the effective bandwidth is achieved.

Yet another obvious optimization (not implemented) is using built-in vector type like float3 or float 4 to store x, y and z co-ordinates and charge (wherever required) of every particle. This would have maximized 128 byte accesses reducing the net number of memory accesses. However, the resultant achievement can't be commented upon till it is implemented and measured.

Constant memory can be used to store certain initially defined constant parameters. However, resultant increase in use of static shared memory (for kernel arguments) needs to be balanced. Texture memory cannot be used since inputs are over-written by outputs, at the end of every time-step, in order to re-use memory. No input needs to be retained for more than a time-step.

Maximization of Utilization

The implementation is divided into strictly serial kernels in order to be able to re-use memory. Hence using asynchronous function calls is not possible. Time-steps being strictly data-dependent, streams of kernels cannot be launched.

A block-size of 32 has been frozen based on experimentation with many block-sizes (all multiples of warp-size). With the block-size of 32, the GPU SMs remain under-utilized till a certain size of input. It was observed that the speed-up stagnated only beyond 8192 threads i.e. atleast 8192 threads were needed to exploit the GPU completely. This would mean atleast 8 blocks or 8 warps (since here block-

size = warp-size) per SM, on GTX280, are required to take care of scheduling for latencies and synchronization barriers. With the amount of global memory accesses and data-dependency between subsequent instructions in the implementation, it is slightly higher than the typical figure of 6 warps per SM.

Because of high usage of registers and small block-size, the occupancy in all the four kernels of the unoptimized implementation is only 25%. This further reduces as we deliberately increase the number of registers. The discussion of occupancy involves an important learning particular to this kind of programs. With a block-size of 32, the occupancy is limited by maximum number of active blocks per SM while the number of active warps per SM stays much below the maximum physical limit of 32 for GTX280. If the block-size was doubled to 64, the occupancy would double to 50% while the number of active warps per SM still stays below the maximum physical limit of 32. But surprisingly a block-size of 32 gives higher speed than 64 for all input sizes. Why so? Especially in an implementation suffering from memory accesses, one would expect increased occupancy to increase performance. Why is the outcome not as expected?

The reason is “low arithmetic intensity of the program”. It is difficult to calculate this ratio from the definition in [2].

The alternate definition could be the number of operations performed per word of memory transferred. This can be roughly approximated from the profiler counters as the ratio of instruction throughput to global memory overall throughput. It should be noted that instruction throughput will be quite higher than floating point operation throughput (FLOPS). Hence this ratio will be slightly higher than even the peak arithmetic intensity. The average ratio for all the four kernels for N=4096 and 10^3 iterationsranged from 0.38 for the unoptimized version to 1.11 for the fully optimized version, which is very low. Suppose the number of warps is increased under the general assumption that scheduling of warps will hide the memory access latency. But because of the low arithmetic intensity, each new warp scheduled will soon stall at memory access. Thus scheduling overhead will take over latency hiding giving increased times of computations. Hence, block-size of 32 works better than 64. Experimenting with 32 active warps per SM (maximum limit) would have been interesting. Unfortunately that is not possible with the high register usage in the program.

Maximize Instruction Throughput

With the minimal optimizations implemented, a good increase in the instruction throughput is achieved. This can be measured from the profiler. Also the arithmetic intensity, by the above definition rises, has risen.

Routine optimizations for arithmetic instructions (viz. using $i>>\log_2 n$ for (i/n), $i<<\log_2 n$ for (i*n), and (i&(n-1)) for (i%n), when n is a power of 2) have been implemented.

For upto 10^7 time-steps, the single precision CUDA implementation results match with single precision CPU results till only upto 10^{-2}. However, since the visualization of the results match we can safely assume that higher accuracy is not needed. Instruction throughput with single precision is certainly higher than that with double precision. Also, fast math options cannot be availed due to already reduced precision.

Limited warp-divergence (resulting from control flow) is unavoidable due to the nature of the algorithm. Thankfully it forms less than even 1% of the GPU time (ref fig.4) else unrolling of loops can be experimented with.

Since the overall occupancy is low, another suggestion is to optimize by exploiting instruction level parallelism by taking inspiration from [5].

Parallelizing on Multi-core X86 processor

For inputs of smaller sizes to the explained CUDA implementation, the GPU remained under-utilized giving poor GFLOPS. Hence the algorithm was also implemented on an Intel i7 2.67GHz CPU (4 cores + hyperthreading) using POSIX.

The implementation remained the same as the CUDA implementation with the following difference:

1. Kernel killing was not required to be replaced with equivalent global synchronization barrier in POSIX. Re-use of memory location is not perceived as read-after-write as in CUDA.
2. Updating positions at the end of every time-step requires ensuring that all energy computations were over. Similarly, beginning a new time-step requires ensuring that all particle-positions are updated at the end of the previous time-step. Hence a global synchronization barrier (pthread_barrier_wait) was used at the beginning and end of the calculation for updating positions.
3. After experimentation, the optimum number of POSIX threads, for the implementation, was confirmed to be 8. It concurs with underlying architecture of 4 cores supporting hyperthreading i.e. 4x2 = 8 virtual cores.

Results

There being strict data-dependency. Only per time-step computations are moved to the GPU at a time

Fig.5 shows the speed-up results with both GPU and multi-core for 1000 iterations:

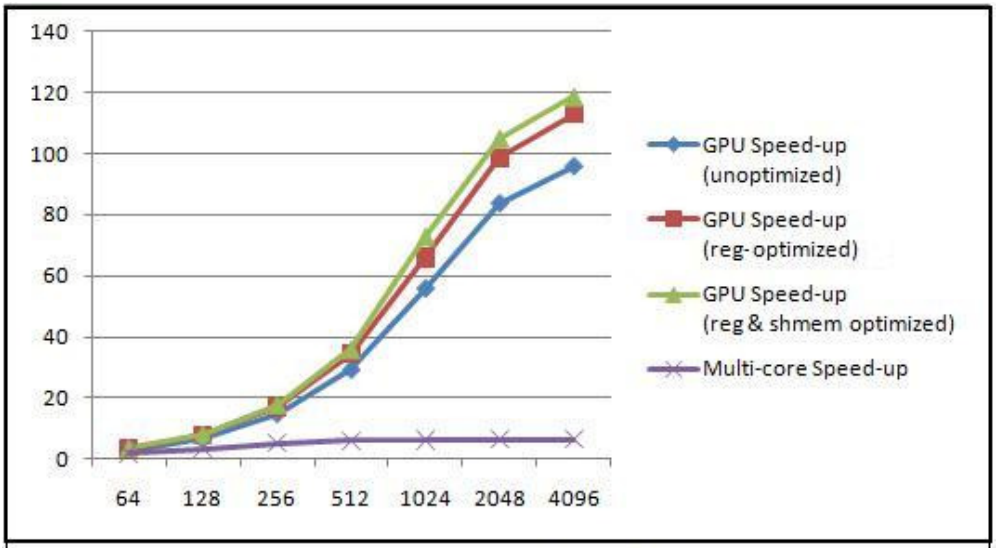


Fig.5 Speed-up on GPU and Multi-core for different input sizes

As is visible from fig.5, the multi-core cannot exploit the data-parallelism as well as the GPU. Since time-steps are data-dependent, the speed-up remains constant for higher number of time-steps than 1000.

Another metric to evaluate performance is GFLOPS. The following assumptions were followed while calculating the number of floating-point operations (FLOPs):

- Addition, Subtraction, Multiplication and Division were all taken as one FLOP each.
- All complex math-functions were taken as one FLOP each (though in practice, they are over one FLOP in a lot of cases).
- Since the implementation involves a lot of comparison operations, they couldn't have been ignored. It was found from CUDA literature that one comparison takes $1/4^{th}$ the number of clock-cycles as required for a FLOP. Hence each comparison operation was considered as 0.25FLOP.
- For if-else statement, FLOPs were computed for the part which would be encountered by maximum number of threads. In case of no clarity about that, the one with maximum FLOPs was considered.
- FLOPs under an 'if' condition were also considered, thus giving peak FLOPs.
- All other operations were ignored.

With the above assumptions, the total GFLOPS for the implementation were calculated to be 23.49 for the un-optimized version and 28.14 for the optimized version. Kernel-2 implements pure n-body simulations with complex math functions. The GFLOPs for kernel-2 were calculated to be 314.23 which can be considered better than in [4] owing to complexity of math involved.

Thus GPU was a good choice for parallelizing [1].

Bibliography

1. "Non-equilibrium unfolding of Polyelectrolyte Condensates in Electric Fields" by R R Netz, Sektion Physik, LMU Munich, Theresienstrasse 37, 80333 Munich, Germany
2. NVIDIA CUDA programming guide 2.3
3. http://www.scribd.com/doc/49661618/23/Throughput-Reported-by-cudaprof
4. GPU Gems 3 – Chapter 31. Fast N-body simulation with CUDA
5. http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf on Better Performance at Lower Occupancy