Accelerated RMSD Calculation for Molecular Metadynamics

Jiří Filipovič, Jana Pazúriková, Aleš Křenek
Institute of Computer Science, Masaryk University
email: fila,pazurikova,ljocha@ics.muni.cz

Vojtěch Spiwok
University of Chemistry and Technology
email: Vojtech.Spiwok@vscht.cz

KEYWORDS
Molecular metadynamics, RMSD, GPU acceleration

ABSTRACT

In this paper, we introduce GPU acceleration of RMSD approximation, which is computationaly demanding task in molecular metadynamics. Comparing to tuned CPU implementation, we have reached 4.4× speedup using mid-end GPU. The scaling of our GPU implementation is sufficient to be usable in real-world application.

MOLECULAR METADYNAMICS

Standard *molecular dynamics* (MD) is a simulation method, established for decades, which computes behaviour of one or more molecules in time. The result of the simulation, a *trajectory* (atom positions in time) of the molecule, can be used for a plethora of purposes (modelling biological processes, predicting molecular properties, evaluating effectivity of a drug, ...).
MD is an N-body simulation, its core is integration of the Newton's equation of motion ($d^2x/dt^2 = F/m$) of all atoms of the molecule in femtosecond steps. It is driven by *force field*, a function which assigns potential energy to a given vector $\mathbf{x}$ of all $3N$ atomic coordinates (N is number of atoms in the system). In order to fit to the integration step, not only the force field (energy) but also its partial derivatives by all components of $\mathbf{x}$ (all atomic coordinates) are usually computed.
Standard molecular dynamics shows expected properties of a thermodynamic system—it tends to oscillate in low energy areas, overcoming higher potential and reaching other low energy areas only occasionally. Therefore, if the purpose of the MD simulation is sampling the whole potential energy surface, the simulation must run rather long to explore all minimas.
Metadynamics (Laio and Parrinello (2002)) addresses the problem by "filling the free energy wells with computational sand"; it biases the potential energy with terms that penalize already visited areas, driving the simulation over energy barriers faster.
A common approach to define such terms is expressing them as *collective variables* which characterize behaviour of the molecule somehow in low-dimensional space. Trivial collective variables can be, for example, torsion angles of freely rotatable bonds.

A promising approach works with *landmark structures*, a set of fixed shapes of the molecule (vectors $\mathbf{a}_i$) which sample the energy surface coarsely. Then, the bias potential is expressed in terms of root mean square distance (RMSD) between the current structure (shape of molecule) and the landmarks:

$$P(\mathbf{x}) = \frac{\sum_{i=1}^{n} P_i e^{-\lambda \mathrm{RMSD}(\mathbf{x},\mathbf{a}_i)}}{\sum_{i=1}^{n} e^{-\lambda \mathrm{RMSD}(\mathbf{x},\mathbf{a}_i)}} \quad (1)$$

where P_i are appropriate constants derived from low-dimensional embedding of the structures $\mathbf{a}_i$ (Branduardi et al. (2007)). In this calculation, the critical part is evaluation of $\mathrm{RMSD}(\mathbf{x}, \mathbf{a}_i)$, which, for all $\mathbf{a}_i$, involves finding the best fitting rotation+shift between the pair of structures. Moreover, partial derivatives w.r.t. all components of $\mathbf{x}$ are required as well (as mentioned before). Profiling reference implementation shows that virtually all the computing time of metadynamics extensions is spent in RMSD calculation.

We focus our work in two directions: reduce the total number of RMSD calculations, and accelerate the remaining ones, the latter being the main topic of this paper.

RMSD APPROXIMATION

First, considering that MD is a contiguous process, we can assume that in each step the change of the current structure $\mathbf{x}$ is tiny w.r.t. the previous one. Therefore we can assign the *close structure* $\mathbf{y} = \mathbf{x}$ in a particular simulation step, compute all $\mathrm{RMSD}(\mathbf{y}, \mathbf{a}_i)$ including the best-fitting rotation $R_{\mathbf{a}_i\mathbf{y}}$, and we retain it fixed for a certain number of subsequent steps. Then, in each step, we compute the best fitting $R_{\mathbf{yx}}$ only (one computation instead of n), and we approximate

$$\mathrm{RMSD}(\mathbf{a}_i, \mathbf{x}) \doteq \frac{1}{N} \| R_{\mathbf{a}_i\mathbf{y}} R_{\mathbf{yx}} \mathbf{x} - \mathbf{a}_i \| \quad (2)$$

Besides reducing the number of fitting operations, partial derivatives w.r.t. $\mathbf{x}$ have to be computed for $R_{\mathbf{yx}}$ and within Eq. 2 only. Consequently, also the code for RMSD calculation is simplified considerably, making it a better candidate for accelerated implementation.

ACCELERATED RMSD CALCULATION

The purpose of this work is to show the potential of accelerated RMSD calculation coupled with the close structure approximation described in the previous section. Therefore we deliberately set up a bit artificial but well-controlled experiment, leaving the full integration with the complex MD software for our future work.
The code of approximated RMSD calculation was extracted from the prototype (implemented on top of Plumed+Gromacs), and wrapped with driver routines which simulate its typical invocation, i.e. appropriate number of calls with fixed $\mathbf{y}$, looping over $\mathbf{a}_i$ etc.

Performance Boundaries

First, we analyse performance boundaries of the RMSD calculation. The input for the computation are set of n landmark structures a, n rotations $R_{\mathbf{a}_i\mathbf{y}}$, displace vector d (weights of particular atoms) and actual simulation state x. Output of the computation is vector of derivatives δ and RMSD distances r. The size of a is $3nN$, size of x and δ is $3N$, size of d is N, size of r is n and size of $R_{\mathbf{a}_i\mathbf{y}}$ is $9n$.
In a metadynamics step, RMSD$(\mathbf{a}_i, \mathbf{x})$ is computed for $i \in < 1, n >$ resulting in complexity $\mathcal{O}(nN)$. We need to perform $34nN$ arithmetic operations and transfer $12nN + 28N + 3n$ bytes to/from memory, so the flop-to-word ratio is about 2.83. This signs the memory-bound problem, so we expect the computation can be roughly as fast as reading atoms of all landmark structures.
Considering GPU as accelerator, there is a good chance to improve the RMSD calculation performance, as GPU memory bandwidth exceeds CPU memory bandwidth significantly. However, most of middle and high-end GPUs are connected via PCI-E bus, thus copying data between CPU and GPU memory may become a new bottleneck. In a metadynamics step, we need to transfer vector x of size $3N$ into GPU memory and copy δ of size $3N$ back. Considering complexity $\mathcal{O}(nN)$, the computation time should dominate when n grows.

Modification of CPU Implementation

In our original implementation within Plumed, the computation for one landmark structure (Eq. 2) has been separated in a method. Our first modification was separation of whole RMSD computation, showed in Alg. 1. The RMSD is computed using single-precision numbers (as it is already approximated by the close structure, high precision arithmetics is not needed), whereas Plumed use double precision, so we retype x, r and δ in each metadynamics iteration.
We have vectorized CPU implementation using autovectorization in Intel C++ compiler and parallelized it using OpenMP. We have performed those fairly well known optimization steps Jeffersi and Reinders (2015): storing

Algorithm 1 CPU RMSD

```
1: for i=0; i < n; i++ do
2:     r_local ← 0
3:     for j = 0; j < N; j++ do
4:         compute δ_local using a_{i,j}, x_j and d_j
5:         r_local ← r_local + (1/4)|δ_local|^2
6:         δ_j ← δ_j + δ_local
7:     r_i ← sqrt(r_local)
```

all atom coordinates as structure of arrays instead of array of structures and informing compiler about memory alignment by pragmas (both improves vectorization efficiency); setting the loop over atoms (line 3) to be vectorized (compiler has not done it automatically); parallelization of loop over landmarks (line 1); creating local copies of array δ (one for each thread), so number of atomic modifications is minimized; setting thread affinity to scatter mode and use only one thread per physical core (improves speed of memory-bound codes). The loop tiling improves cache locality, however, this optimization has been performed by the compiler.

GPU Implementation

To implement GPU version of RMSD computation, we have used CUDA framework NVIDIA (2015). In CUDA, the code runs in high number of lightweight threads grouped into thread blocks. Thread within the block can synchronize and use fast shared memory. To utilize whole GPU, multiple thread blocks need to be executed.
To compute derivatives, we can parallelize the loop running over atoms (line 3 of Alg. 1), such that i-th CUDA thread contains loop running over all landmark structures and computing δ_i. This parallelization pattern yields efficient coalesced memory access (neighbouring threads access neighbouring atoms). Furthermore, each thread can cache its value of x_i, d_i and δ_i in registers. However, computing r is not efficient in such implementation. The array δ is reduced sequentially without synchronization, however, we need to reduce r_{local} individually for each landmark structure (see line 7 of Alg. 1). Thus, reduction of r has to be performed in each iteration running over landmark structures in parallel, which is less efficient comparing to serial reduction. Alternatively, second kernel with parallel loop over landmark may be executed to compute r, however, the array a is read twice. Both parallel reduction and double access to a result in about half performance.
To improve performance, we have modified parallelization pattern used in our previous work Filipovič et al. (2015) (for matrix-vector operations) which allows us to compute r efficiently: the distances needed to compute r are computed and stored in shared memory buffer *buf* and the reduction is performed serially on *buf*. In this implementation, thread block contains $b_x \times b_y$ threads and process t atoms of t landmark structures (forming a

Algorithm 2 GPU Tiled Algorithm

1: $g_x \leftarrow$ global thread index
2: $l_x, l_y \leftarrow$ local thread indices
3: **for** i=0; i < n; i += t **do**
4: **for** ii = 0; ii < t; ii++ **do**
5: $l \leftarrow i + ii$
6: compute δ_{local} using $a_{g_x,l}$, x_{g_x} and d_{g_x}
7: $buf_{ii,l_x} \leftarrow buf_{ii,l_x} \frac{1}{4}|\delta_{local}|^2$
8: $\delta_{g_x} \leftarrow \delta_{g_x} + \delta_{local}$
9: barrier sync.
10: **if** $l_y == 0$ **then**
11: $r_{local} \leftarrow 0$
12: **for** j = 0; j < t; j++ **do**
13: $r_{local} \leftarrow r_{local} + buf_{l_x,j}$
14: atomically $r_{i+l_y} \leftarrow r_{i+l_y} + r_{local}$

Table 1: Performance in ideal conditions.

	CPU orig	CPU opt.	GPU
complete	3.36 GFlops	61.24 GFlops	270.0 GFlops
δ only	3.36 GFlops	61.45 GFlops	298.6 GFlops

$t \times t$ tile in 2D array a). For implementation simplicity, we set $b_x = t$ and $b_y \mid t$. The pseudocode is in Alg. 2. There are two nested loops performed by each thread. The outer loop (line 3) iterates over all landmarks with step of size t. The first inner loop (line 4) iterates within tile, computes δ and store distances in *buf*. After the first inner loop finishes, the block-local reduction of data within *buf* is performed: b_x threads iterates over buffer columns, summing the local distances serially (lines 10-13). The global reduction is performed at line 14 in reasonable number of atomic operations. Square roots of components of r are performed in CPU code in $\mathcal{O}(n)$. Note that this implementation requires padding of landmark structures to number divisible by t (in our implementation, we set $t = 32$). However, it is not in principle problem to constraint number of landmark structures in metadynamics. Also the number of atoms have to be divisible by x-dimension of thread block, which can be easily satisfied by adding dummy atoms with zero displacement which add zero to r.

EVALUATION

In this section, we compare our GPU implementation including all data conversions and transfers with original and tuned CPU implementation. All measurements have been performed on computer equipped by Intel Core i7-3820 (4 cores at 3.6 GHz and GeForce GTX 480). First, we measure the performance in ideal conditions, i.e. for sufficiently large input. We have used molecules of 8192 atoms and 4096 landmark structures. Results are depicted in Table 1. The speedup of GPU implementation is 4.4× over tuned and 80.4× over original CPU implementation. When only derivations are computed and thus simple parallelism pattern is employed,

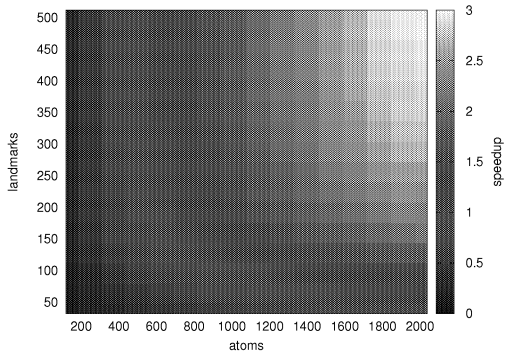

Figure 1: GPU speedup scaling.

the performance is similar to our buffered algorithm. Second, we measure how the performance scales according to number of atoms and number of landmark structures. Fig. 1 shows speedup of GPU over tuned CPU implementation. As we can see, the GPU implementation is preferable for realistic simulation configurations (using hundreds of landmark structures and thousands of atoms).

CONCLUSION AND FUTURE WORK

The GPU implementation significantly outperforms both original and tuned CPU implementation. It is also usable for realistic problem sizes. In our future work, we plan to integrate our implementations into Plumed+Gromacs. Moreover, we plan to add code which automatically select target for computation (GPU or CPU) according to input data and hardware configuration, or even splits workload between GPU and CPU.

ACKNOWLEDGEMENT

This work was supported by the Grant Agency of the Czech Republic, project no. 15-17269S.

REFERENCES

Branduardi D.; Gervasio F.L.; and Parrinello M., 2007. *From A to B in free energy space. Journal of Chemical Physics*, 126, no. 5, 054103.

Filipovič J.; Madzin M.; Fousek J.; and Matyska L., 2015. *Optimizing CUDA code by kernel fusion: application on BLAS. The Journal of Supercomputing*, 71, no. 10. doi: 10.1007/s11227-015-1483-z.

Jeffersi J. and Reinders J., 2015. *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches.* Morgan Kaufmann.

Laio A. and Parrinello M., 2002. *Escaping Free-Energy Minima. Proceedings of the National Academy of Sciences of the United States of America*, 99, no. 20, 12562—12566.

NVIDIA, 2015. *CUDA C Programming Guide, version 7.5.*