Accelerating Energy Minimization using Graphics Processors*

Bharat Sukhwani Martin C. Herbordt
Department of Electrical and Computer Engineering, Boston University

Abstract: Energy minimization is an important step in molecular modeling, with applications in molecular docking and in mapping binding sites. Minimization involves repeated evaluation of various bonded and non-bonded energies of a protein complex. It is a computationally expensive process, with runtimes typically being many hours on a desktop system. In the current article, we present the acceleration of the energy evaluation phase of minimization on a graphics processor, resulting in an 11x total speedup compared with a single core of a desktop machine.

1. Introduction

Molecular docking refers to the computational prediction of the least energy pose between two interacting proteins. This is a computationally demanding process, requiring many hours to days of CPU runtime. Due to the computational complexity of the problem, most docking systems adopt a two step process. The first step docks the two proteins to find the best fit, often assuming the proteins to be rigid. It scores billions of poses between the two proteins and a few thousand top scoring poses are preserved for further evaluation.

The second step performs minimization of potential energy of the docked complexes generated by the first step. During minimization, the larger molecule is generally held fixed whereas the side chain atoms of the smaller molecule (the ligand) are free to move. This accounts for the flexibility in the molecule. At each minimization iteration, the potential energy of the complex is computed and a move to a neighboring point is made using an optimization technique such as Newton-Raphson or quasi-Newtonian (L-BFGS). This process is repeated until the energy converges within a given threshold. Often, up to a thousand iterations can be performed per conformation, resulting in runtimes of about 30 seconds per conformation [1].

With many thousand conformations to be minimized per protein-ligand pair, the total time for minimization phase runs can be many hours. Moreover, drug discovery involves docking-based screening of millions of candidate ligands for a given protein. Acceleration of energy minimization is thus highly desirable, as it will lead to faster molecular docking and quicker turn-around times in drug discovery.

In addition to docking, energy minimization is also used in other applications such as mapping, which aims at determining the likely binding sites on a given protein. The process involves docking a set of small molecule probes using rigid docking and performing the minimization for each protein-probe complex. FTMap [1], one of the mapping algorithms, docks 16 different probes to the given protein. It retains 2000 top scoring conformations from rigid docking step for each probe, which are further minimized using the CHARMM potential.

While energy minimization can potentially benefit from acceleration, and while this computation is superficially similar to the well-studied molecular dynamics, little or no work has been done in this area. The difficulty is perhaps because there is comparatively little computation to be performed per iteration and that a substantial fraction of it is apparently serial: a direct mapping onto a GPU is therefore likely to be dominated by data transfer overhead. We address this problem by revising the underlying data structures and applying a new work allocation scheme.

The focus of the current paper is accelerating the energy evaluation step of the FTMap algorithm using graphics processors. FTMap employs Analytic Continuum Electrostatics (ACE) and solvation energies [3] in addition to other CHARMM energy terms such as van der Waals and bonded energies. Of these, ACE electrostatics and solvation and the van der Waals terms represent the non-bonded (external) interactions and constitute most of the computation. In the current paper, we accelerate the evaluation of ACE electrostatics and solvation terms on GPU, achieving 15x speedup on computations that constitute more than 90% of total runtime.

2. Energy minimization

The total potential energy of a complex is given as a sum of various bonded and non-bonded terms:

$$E^{total} = \underbrace{E^{vdw} + E^{elec}}_{non-bonded} + \underbrace{E^{bond} + E^{angle} + E^{torsion} + E^{improper}}_{bonded} \tag{1}$$

This expression is evaluated during each minimization iteration and is the core computation.

The electrostatic energy of a solute can be decomposed into two components: a self-energy term, which depends only on the charge of the atom under consideration, and a pairwise interaction term, which depends on the charges of the two atoms. The total electrostatic energy of the complex is given as a sum of all the self energies, E_i^{self}, and pair-wise interaction energies, E_{ij}^{int} [3].

For ACE, the self energy of an atom is represented as a sum of its Born self-energy plus the sum of the effective interactions due to other solute atoms within a certain volume associated with the atom [3].

$$E_i^{self} = \frac{q_i^2}{2\varepsilon_s R_i} + \sum_{k \neq i} E_{ik}^{self} \tag{2}$$

$$E_{ik}^{self} = \frac{\tau q_i^2}{\omega_{ik}} e^{-\left(\frac{r_{ik}^2}{\sigma_{ik}^2}\right)} + \frac{\tau q_i^2 \tilde{V}_k}{8\pi} \left(\frac{r_{ik}^3}{r_{ik}^4 + \mu_{ik}^4} \right)^4 \tag{3}$$

where q_i represents the charge on atom 'i', r_{ik} is the distance between atoms 'i' and 'k', $\tilde{V}_k$ is the size of the

* This work was supported in part by the NIH through award #R01-RR023168-01A1. Web: www.bu.edu/caadlab. Email: {herbordt|bharats} @bu.edu

solute volume associated with atom 'k', and ω_{ik}, σ_{ik}, and μ_{ik} are atom-atom parameters.

The pair-wise interaction energy is given by the generalized Born (GB) equation, which is a sum of Coulomb's law in dielectric and Born equation [4]

$$E_{ij}^{\text{int}} = 332\sum_{j\neq i}\frac{q_i q_j}{r_{ij}} - 166\tau\sum_{j\neq i}\frac{q_i q_j}{\sqrt{r_{ij}^2 + \alpha_i \alpha_j e^{-\left(r_{ij}^2 / 4\alpha_i\alpha_j\right)}}} \quad (4)$$

where α_i and α_j represent the Born radii for atoms 'i' and 'j', respectively. These in turn depend on the self energies of the atoms.

Equations (3) and (4) represent the main computations that need to be performed for all atom-atom pairs to evaluate the total electrostatic energy of a given conformation. In addition, the energy gradients need to be computed to determine the new position and to update the forces acting on the atoms.

3. Energy computations: Mapping to GPUs

In the original ACE computation, atoms are arranged in neighbor lists. Each "first atom" 'i' in the array of neighbor lists has a list of "second atoms" 'k' that contribute to its self energy and the interaction energy (Figure 1).

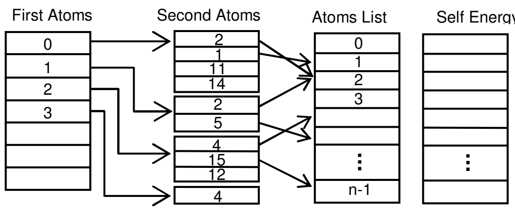


Figure 1. Array of neighbor lists

As the atom positions change, the neighbor lists are updated. To compute the self energies of the atoms, the original FTMap program cycles through the list of first atoms. For each first atom, it traverses the list of second atoms, updating the self energies of both the first atom as well as the second atom. After the self energies of all the atoms have been computed, they are accumulated to obtain the total self energy of the system. Similarly, the neighbor list array is again traversed to compute the pair-wise interaction energies using the generalized Born equation. The final step involves computing the gradients for each atom and updating the forces acting on the atoms.

We divide the above computations into three GPU kernels: (i) computing the self energies and gradients, (ii) computing the pair-wise interactions and gradients, and (iii) updating the forces on the atoms. There are different ways of mapping these computations on the GPU using the NVIDIA CUDA architecture. The main complication comes from the need to accumulate energies computed by different threads. Moreover, a particular atom might be present in the neighbor lists of different first atoms. This requires some serialization during accumulation.

Straightforward distribution of this work across GPU threads can lead to the second atoms of a given first atom being distributed across multiple GPU multiprocessors, making it difficult to accumulate the total energy of the atom. On the other hand, distributing one first atom on each multiprocessor enables fast and easy accumulation. This approach, however, results in very few threads per block, and many threads sitting idle, leading to poor performance.

Though there are various ways to solve these problems, most of them require much data transfer between the host and the GPU. We tried various techniques to address this issue. Here we discuss two of those. We explain these with respect to computing the self energies; computing pair-wise interactions, gradients, and force updates follow the same approach.

For efficient distribution of work among different GPU threads, we drop the structure of neighbor list. Rather, we distribute the two dimensional first atom-second atoms set as a one dimensional list of atom-atom pairs (Figure 2). In our first approach, we distribute the pairs equally among the GPU threads, with each thread independently updating the self energies of the two atoms in the GPU global memory. This is followed by accumulation of the self energies of different atoms from the structure of Figure 2. This needs to be performed in a serial order, mainly due to the random occurrences of atoms in the second atom list.

We tried accumulation on both the host and on the GPU. Accumulation on the host is very fast, but requires transferring two arrays of floats from the GPU to the host during every iteration. Due to the relatively small amount of computation performed by each thread, the time for data transfer is actually larger than the computation time on the GPU. Performing accumulation on GPU, on the other hand, requires accessing GPU global memory and results in significant slowdown. With accumulation the on host, our first approach resulted in a modest speedup of 2x-3x.

	Atom index		Self energy	
Pair id	Atom 1	Atom 2	Atom 1	Atom 2
0	0	2		
1	0	1		
2	0	11		
3	0	14		
4	1	2		
5	1	5		
6	2	4		
7	2	15		
8	2	12		
9	3	4		

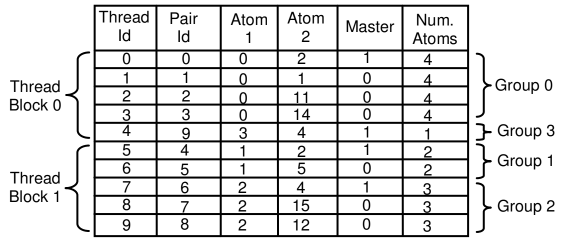
Figure 2. Atom-atom pair list with two energy fields

In our second approach to work distribution we still use the atom-atom pair list, albeit with two changes. The first change is to split the list into two parts – the first part is based on the original neighbor lists while the second part is based on something we call the reverse neighbor list. We generate the reverse neighbor lists by treating every second atom in the original neighbor list (Figure 1) as the first atom of the new list. With this, while evaluating the self energies, we only compute the energies of the first atoms. For the second atoms, we repeat the process with the reversed list.

The second change involves statically mapping the computations onto different threads (explained next). These two changes add determinism in the order in which the atoms appear in the list, helping us to better distribute the work on GPU threads and allowing us to use GPU shared memory for accumulation. This avoids the problems associated with accumulation from global memory or accumulation on host, as discussed above.

Once we have the atom-atom pair list, we statically distribute it among threads running on different multiprocessors. Note that the pairs in the pair-list (see Figure 2) are ordered by the first atom. We can thus group them into different groups, where all the pairs in a group have the same first atom. The static mapping scheme takes each group and maps it to threads such that the entire group lies on the threads in the same thread block. If the current thread block does not have enough threads left to accommodate the entire group, it is allocated on the next available thread block. This allows us to store the results generated by each thread on the shared memory and perform faster accumulation.

We generate a new list, called the assignment table (Figure 3), which is ordered by the thread ids and assigns pairs to threads. The list contains 5 fields: pair id, indices of the two atoms, a master field, indicating if the current thread is the first thread for this group and the number of atoms present in this group. The last two fields (master and num. atoms) are used for accumulation of self energy of the atoms, as explained below.


Thread Block 0
Thread Block 1
Group 0
Group 3
Group 1
Group 2

Thread Id	Pair Id	Atom 1	Atom 2	Master	Num. Atoms
0	0	0	2	1	4
1	1	0	1	0	4
2	2	0	11	0	4
3	3	0	14	0	4
4	9	3	4	1	1
5	4	1	2	1	2
6	5	1	5	0	2
7	6	2	4	1	3
8	7	2	15	0	3
9	8	2	12	0	3

Figure 3. Assignment table

The table in Figure 3 is stored in GPU global memory. One table is generated from each of the original and the reverse neighbor lists and is transferred to the GPU only once at the beginning. There is no further data transfer per iteration, unless the neighbor list is updated, in which case we regenerate the assignment tables and transfer them to GPU. This happens only a few times in 1000 minimization iterations; thus the transfer time is negligible.

Each thread works on the pair assigned to it in the assignment table. In case the number of pairs is larger than the number of threads, each thread would be responsible for multiple rows. Energies computed by different threads are stored in an array in the GPU shared memory. The length of this array is equal to the number of threads in the thread block, with each thread storing at the index equal to its local thread id (id within the block).

Once all the threads have finished their computations, the threads with master fields equal to 1 perform the accumulation round. Each such thread reads the number of atoms for the group associated with it and accumulates that many values from the shared memory, starting from its local thread id. This way, many threads perform accumulation in parallel and from the shared memory, resulting in significant speedup compared to previous schemes. The master threads then store the accumulated values in the GPU global memory. Note that we can use this scheme only because we are computing and updating the energies of only the first atom. For the second atom, we repeat the above process with the assignment table corresponding to the reverse neighbor list.

Calling the kernel twice leads to repeating some of the computations. We tried to avoid this by storing those values in GPU global memory during the first kernel call and reusing during the second call. This, however, resulted in slowdown compared to repeating the computation due to slower global memory access. Similar approaches have also been applied in molecular dynamics simulations [2].

4. Results

We present results of accelerating the ACE energy evaluation on GPU and the overall speed-up. We performed minimization on five different protein-probe complexes. Each complex contains around 2260 atoms and requires evaluations of 9780 atom-atom pairs per iteration. 1000 minimization iterations were run on each complex to obtain the total runtime. The GPU accelerated code runs on a NVIDIA TESLA C1060 GPU card and the original un-accelerated code runs on a quad core Intel Xeon processor at 2.00 GHz (with only one core utilized).

Table 1. Speedups for various computations

Computation	*Serial Time*	*GPU Time*	*Speedup*
Self energies	6.15 ms	0.22 ms	27.9x
Pairwise	2.75 ms	0.23 ms	11.9x
Force updates	0.95 ms	0.14 ms	6.7x

Table 2. Overall speedups for five different complexes

Complex	*Serial Time*	*GPU Time*	*Speedup*
Complex 1	11.9 sec	1.098 sec	10.8x
Complex 2	11.87 sec	1.078 sec	11x
Complex 3	11.8 sec	1.078 sec	10.9x
Complex 4	10.74 sec	0.906 sec	11.8x
Complex 5	11.87 sec	1.094 sec	10.8x

As stated earlier, we divided the entire ACE computation into three GPU kernels: (i) computing self energies and its gradients, (ii) computing pairwise interaction energies and gradients and (iii) updating forces acting on the atoms. Table 1 reports the speedups achieved on these steps (for one minimization iteration). GPU times include time for kernel execution for both forward and reverse neighbor-lists. The first of these computations is most computationally rich and hence affords maximum speedup. Updating forces involves very simple computation, resulting in much modest speedup.

The overall speedup for one iteration of ACE energy evaluation is 15.2x. For 1000 minimization iterations, we achieve an overall end-to-end speedup of 11x for the FTMap energy minimization (Table 2), which includes evaluation of other energy terms (van der Waals and bonded terms) on the host. Accelerating these computations on the GPU will further improve the overall performance and is future work.

References

[1] Brenke, R. *et al.* (2009) Fragment-based identification of druggable 'hot spots' of proteins using Fourier domain correlation techniques. *Bioinformatics*, 25(5), 621-627.
[2] Phillips, J.C., Stone, J.E., K. Schulten (2008): Adapting a message-driven parallel application to GPU-accelerated clusters. Supercomputing.
[3] Schaefer, M. and Karplus, M. (1996) A Comprehensive Analytical Treatment of Continuum Electrostatics. J. Phys. Chem., 100 (5), 1578-1599
[4] Still, W. C., Tempczyk, A., Hawley, R. C., Hendrickson, T. (1990) Semianalytical treatment of solvation for molecular mechanics and dynamics. *J. Am. Chem. Soc.*, 112 (16), 6127-6129.