Towards Accelerating Molecular Modeling via Multi-Scale Approximation on a GPU

Mayank Daga, Wu-chun Feng, and Thomas Scogland
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
{mdaga, feng, njustn}@cs.vt.edu

***Abstract*—Research efforts to analyze biomolecular properties contribute towards our understanding of biomolecular function. Calculating non-bonded forces (or in our case, electrostatic surface potential) is often a large portion of the computational complexity in analyzing biomolecular properties. Therefore, reducing the computational complexity of these force calculations, either by improving the computational algorithm or by improving the underlying hardware on which the computational algorithm runs, can help to accelerate the discovery process. Traditional approaches seek to parallelize the electrostatic calculations to run on large-scale supercomputers, which are expensive and highly contended resources.**

Leveraging our multi-scale approximation algorithm for calculating electrostatic surface potential, we present a novel mapping and optimization of this algorithm on the graphics processing unit (GPU) of a desktop personal computer (PC). Our mapping and optimization of the algorithm results in a speed-up as high as *four orders of magnitude*, when compared to running serially on the same desktop PC, without deteriorating the accuracy of our results.

Keywords: biomolecular function, molecular modeling, electrostatic surface potential, multi-scale approximation, graphics processing unit (GPU).

I. Introduction

Electrostatic interactions are of utmost importance in the analysis of the structure of biomolecules [1]–[3] as well as their functional activities like ligand binding [4], complex formation [5] and proton transport [6]. The study of the electrostatics phenomenon of a macromolecule has used the linear Poisson-Boltzmann (PB) [7] method for computing the electrostatic surface potential produced by a molecule. Though PB computes the potential on the atomic scale, it is highly computationally expensive [8]. Closed-form analytical approximations reduce the computational complexity while maintaining accuracy, but still continue to be a computational bottleneck due primarily to their long-range nature [9]. Consequently, more efficient algorithms, such as the spherical cut-off method [10], fast multipole approximation [11], and particle mesh Ewald (PME) method [12], have been proposed to reduce the computational complexity. These algorithms, in turn, have also been parallelized to run on large-scale supercomputers in order to reduce execution times further.

We propose a hybrid approach that combines the algorithmic efficiency of a multi-scale approximation algorithm called hierarchical charge partitioning (HCP) [13] *with* the massively parallel architecture of a graphics processing unit (GPU) in a desktop personal computer (PC) in order to deliver up to *four orders of magnitude of speed-up*, when compared to running on a traditional serial processor, thus rivaling the performance of traditional supercomputers.

HCP is an approximation algorithm based on the *natural* partitioning of biomolecules as explained in Section III-A. It has certain benefits over both the PME and spherical cut-off methods. PME is presently not suitable for implicit solvent simulations and requires an artificial periodicity to be imposed on the system [14]. While the spherical cut-off method is inherently the simplest algorithm, its accuracy is not as good as HCP.

II. Related Work

Recently, several molecular modeling applications have used the GPU to speed-up electrostatic computations. Rodrigues et al. [15] and Stone et al. [16] demonstrate that the estimation of electrostatic interactions can be accelerated by the use of spherical cut-off method and the GPU. In [17], Hardy et al. used a multi-level summation method on the GPU. Each of the aforementioned implementations artificially maps the n atoms of a molecule onto a m-point lattice grid and then applies their respective approximation algorithm. By doing so, they reduce the time complexity of the computation from $O(nn)$ to $O(nm)$. In contrast, we use HCP, which performs approximations based on the natural partitioning of biomolecules. The advantage of using the natural partitioning is that even with the movement of atoms during molecular dynamics simulations, the hierarchical nature is preserved, whereas with the lattice, atoms may move in and out of the lattice during the simulation.

Anandakrishnan et al. present a GPU implementation that accelerates the computation of electrostatic surface potential using only a single level of approximation of HCP on an AMD GPU [18]. Unfortunately, due to architectural and system software limitations of the AMD GPU, they could only incorporate one level of HCP approximation. In the present work, we use NVIDIA GPUs so as to incorporate three levels of HCP and extensive optimizations in order to deliver a four order-of-magnitude speed-up.

GPUs have been found to be useful not only for molecular modeling but also for molecular dynamics. Pande et al. [19], developed a GPU-based protein folding client called openMM. More recently, GPU-accelerated implementations of the NAMD molecular dynamics simulation package have been reported [20]. These algorithms determine cut-off pair interactions between atoms, ultimately resulting in forces that determine the dynamics of simulated molecules.

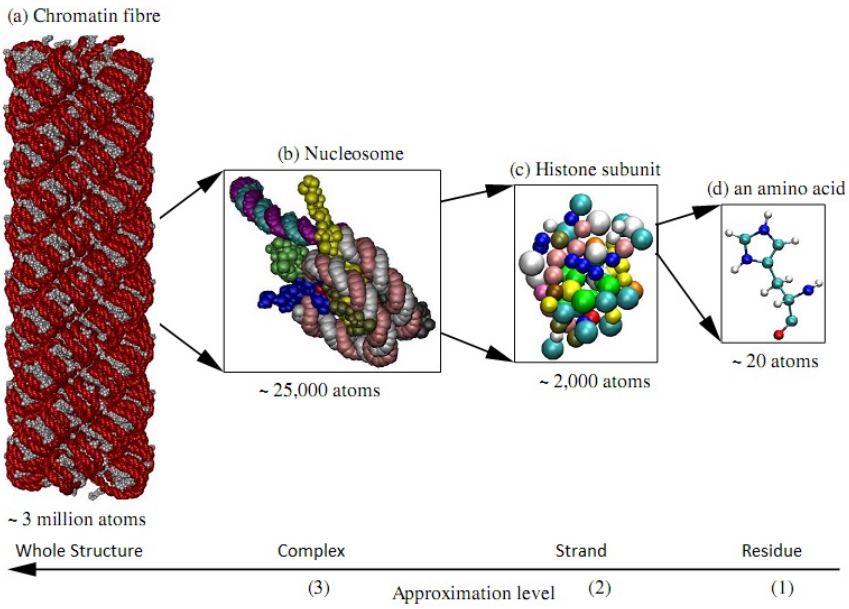


Fig. 1. Example of Natural Partitioning in Biomolecules

III. Background

Here we discuss the methods that we used to compute the electrostatic surface potential of a molecule as well as the architecture of the GPU used in our experiments.

A. Electrostatics and Hierarchical Charge Partitioning

We use an Analytic Linearized Poisson-Boltzmann (ALPB) [21] model to perform the electrostatic computations. Equation (1) computes the electrostatic potential at a point on the surface of the molecule due to a single point charge, q. The potential at each vertex on the surface can be computed as the summation of the potentials generated by all charges in the system. If there are P vertices, the total surface potential can be found as the summation of the potential at each vertex. Gordon et al. analyze the effect of salts in the solvent and assess the accuracy of said model [9].

$$\phi_i^{outside} = \frac{q_i}{\epsilon_{in}} \frac{1}{(1+\alpha\frac{\epsilon_{in}}{\epsilon_{out}})} \left[\frac{1+\alpha}{d_i} - \frac{\alpha(1-\frac{\epsilon_{in}}{\epsilon_{out}})}{r} \right] \quad (1)$$

Computing the potential at P vertex points on the surface results in a time complexity of $O(NP)$ where N is the number of atoms in the molecule. To reduce the time complexity, we apply an approximation algorithm called hierarchical charge partitioning (HCP), which reduces the upper bound of computation to $O(P \log N)$.

HCP strives to accelerate electrostatic computations via the natural partitioning of biomolecules into their constituent components. As shown in Figure 1, a biomolecule can be viewed as being made up of a number of molecular complexes, which consist of a number of strands, which then consist of nucleotide residues. Atoms belong to the lowest level in this hierarchy. Each level in the hierarchy corresponds to an incremental level of approximation that can be applied.

Each constituent component, i.e., complex, strand and residue, is approximated by fewer point charges and the electrostatic effect due to distant components is calculated using these point charges instead of all the atoms present in that component, thereby reducing the total number of computations to be performed.

Algorithm: HCP

```
for v = 0 to #Vertices do
  for c = 0 to #Complexes do
    if (dist(v, c) >= 3rd threshold) then
      potential += calcPotential(v, c)
    else
      for s = 0 to #Strands in c do
        if (dist(v, s) >= 2nd threshold) then
          potential += calcPotential(v, s)
        else
          for r = 0 to #Residues in s do
            if (dist(v, r) >= 1st threshold) then
              potential += calcPotential(v, r)
            else
              for a = 0 to #Atoms in r do
                potential += calcPotential(v, a)
```

The decision to use the exact charges or the approximate charge of the component is made by comparing the distance between the vertex in consideration and the component as shown in the HCP algorithm. If the distance is greater than the threshold for that level, then the approximate charge is used.

B. GPU Architecture and Programming Interface

Originally, the GPU was dedicated hardware to perform graphics routines extraordinarily fast. In recent years however, the GPU architecture has evolved into a general-purpose

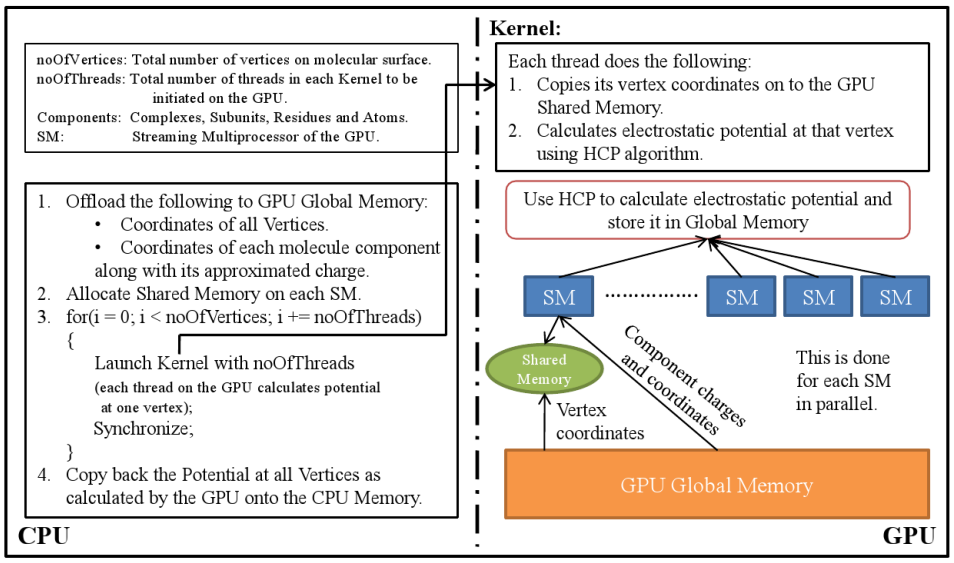


Fig. 2. Mapping of HCP onto a GPU

processor enabling programming models such as the Compute Unified Device Architecture (CUDA) for NVIDIA GPUs.

Data-parallel computations such as the electrostatic surface potential are ideally suited for the GPU. However, given that the GPU has more transistors devoted to such computations than for caching and managing control flow, memory accesses and divergent branches can be particularly expensive on the GPU. Thus, a key aspect of GPU programming is to hide the latency of memory accesses with computation via massive multi-threading. The GPU allows thousands of threads to be initiated such that when one thread is waiting on a memory access, other threads can perform meaningful computations.

A key architectural feature of NVIDIA GPUs is its memory hierarchy. For example, in a GT200 GPU, each streaming multiprocessor (SM) possesses a set of 32-bit registers and shared memory, both located on-chip. In addition, the GPU has a read-only constant cache. Finally, the GPU device memory consists of thread local[1] and global memory, both of which reside off-chip.

CUDA provides a C-like language with an application programming interface (API) for the NVIDIA GPU. A CUDA program is executed by a calling a function known as 'kernel' from the host, i.e., a CPU. CUDA logically arranges threads into blocks, which are in turn grouped into a grid. Each thread has its own ID, which provides for one-to-one mapping. Each block of threads is executed on a SM and the threads within a block may safely share data via shared memory.

Additional details on the NVIDIA GPU architecture and CUDA programming environment can be found at [22].

IV. Approach

Calculating the electrostatic surface potential is inherently data parallel in nature. That is, the potential at one point on the surface can be computed independently from the computation of potential at some other point on the surface.

Figure 2 shows the execution path of our algorithm, both on CPU and GPU. The algorithm first offloads the vertex coordinates at which the potentials are to be calculated along with the coordinates of all molecular components and their approximated point charges to the global memory of the GPU. Each thread in the GPU kernel code is then assigned the task of computing electrostatic potential at one vertex using Equation (1). Specifically, each thread copies its vertex coordinate from global memory to shared memory, resulting in a significant reduction in the number of global memory loads, as explained in Section IV-E. Next, the HCP algorithm is applied at the vertex in question and the result stored back to global memory. This is done by all threads in parallel. After the threads finish, the computed potential at each vertex is then copied back to CPU memory, where a reduce (sum) operation is performed to calculate the total molecular surface potential. Our analysis shows that the computation time on the GPU outperforms the time required to copy data to and fro between the host and the device. Hence, one needs to optimize the computation in order to reduce execution time.

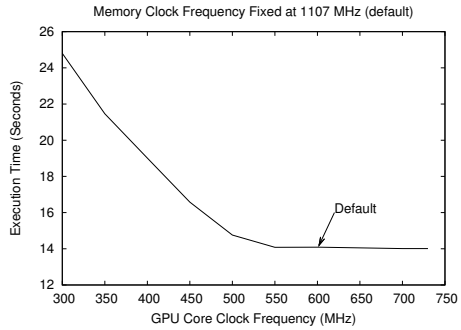
In [23], Ryoo et al. illustrate that writing a program optimized for performance on the GPU takes non-trivial effort as the optimization space is very large. In subsequent sections, we describe our approach towards massively accelerating the calculation of electrostatic surface potential.

A. Boundedness

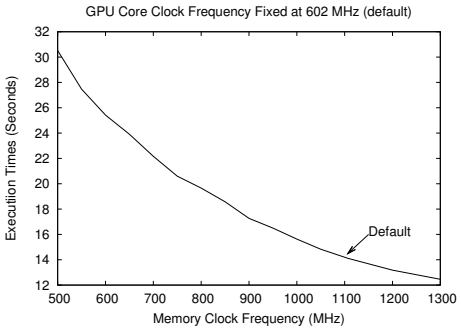
We evaluated the boundedness of our application on the GPU in order to reduce the optimization search space. To do so, we studied the change in execution time of the application with varying GPU core and memory frequencies via *Coolbits*, a utility that allows tweaking of features via the NVIDIA driver control panel.

[1]The name "local" is actually misleading as it is not actually located on-chip.

In Figure 3a, we freeze the GPU memory frequency at the default value of 1107 MHz and vary the GPU core frequency. As expected, the execution time decreases steadily with the increase in the clock frequency. However, this decrease asymptotes around 550 MHz. Increasing clock frequency further, even over-clocking has no effect on the execution time. Thus, the application is *compute bound* when the clock frequency is 550 MHz or less but *memory bound* thereafter.



(a) Execution Time vs GPU Core Frequency



(b) Execution Time vs GPU Memory Frequency

Fig. 3. Memory Boundedness on the GPU

To corroborate our claim, we plot Figure 3b, where the GPU core frequency is kept constant at the default value of 602 MHz and the memory frequency is varied. The execution time decreases steadily as the GPU memory frequency increases. In this case, even overclocking the GPU memory (i.e., greater than 1107 MHz) reduces the execution time. Thus, the application is *memory bound* at the default GPU core frequency and the various memory frequencies of the GPU.

Because the application must wait on memory requests to complete and not on the computations to finish, optimizations that reduce the number of global memory accesses would likely improve performance. Optimizations like the use of fast_math library functions and loop unrolling would not help to improve performance as they merely enable faster computations. Presented below are the optimizations that we performed.

B. Reducing Parameter Size

A limitation of the CUDA GPU kernel is that it restricts the parameter size to 256 bytes. To use HCP, it is required to pass many data structures to the GPU kernel. However, the combined size of pointers to these structures exceeded 256 bytes and hence, it was not possible to pass all the data-structures to the kernel. To overcome this problem, we packed all the necessary data structures in a structure and then passed the address of that structure to the kernel. This not only allowed us to execute the kernel with all the required parameters, but it also reduced the register utilization per thread on the GPU, which in turn, enabled us to launch more threads, resulting in better occupancy of the GPU.

C. Reducing Global Load and Store Transactions

If all threads in a half warp [2] access data from contiguous memory locations, though not necessarily consecutive, then the number of memory transactions required is reduced by a factor of 16, hence increasing the bandwidth of global memory [18]. To take advantage of this GPU architectural feature, we transformed all the data structures from arrays of structures to arrays of primitives so that we could have a linear arrangement in memory. Moreover, if all the threads in a half warp access data from only one half of the data segment in memory, then the transaction size is reduced by half.

D. Using Constant Memory

A constant variable should be stored in constant[3] memory rather than global memory. This results in relatively fewer accesses to global memory, hence improving performance. We took advantage of the presence of cache-like constant memory and stored all the constants in it. This optimization technique delivered a *1.2x* speedup over the non-optimized implementation.

E. Using Shared Memory

HCP reuses vertex coordinates for the computation of distance with the molecule component at each approximation level. Therefore, in worst case when no approximation could be applied, same data is accessed four times from global memory. To reduce these memory accesses, we have used shared memory available per SM on the GPU to store the vertex coordinates at which the potential would be calculated by that SM. The percentage reduction in the number of global memory loads due to the use of shared memory with and without HCP approximation is shown in Table I. The number of memory accesses were taken from the CUDA Visual Profiler provided by NVIDIA.

From the table, we note that there is a *50%* reduction in global memory loads for all structures when the potential is calculated without HCP. In this case, coordinates of vertices and that of atoms, are required from the global memory. In order to access the atom coordinates, one has to cycle through the residue groups. Therefore, when we do not use shared memory, the vertex coordinate is loaded twice, once for residue

[2]A "warp" is a group of 32 threads and it is the smallest unit of execution on a NVIDIA GPU.

[3]To be clear, this is the cached constant memory which is chip local, not texture memory.

and once for the atom, though when shared memory is used, the vertex is loaded only once.

However, with HCP, amount by which the number of loads are reduced varies across structures which can be reasoned as follows. For each structure the effective number of computations to be performed are different. For example, if for a structure, 1st level approximation could be applied use of shared memory restricts the number of global memory loads to one, i.e., copying vertex coordinates from global memory to shared memory. While if shared memory would not have been used, three global memory loads would have been required to copy vertex coordinates at each of the three component levels. From the table, we can guess that least number of components could be approximated in case of virus capsid, thus, maximum percentage reduction. Use of shared memory resulted in providing about *2.7x* speedup over the non-optimized implementation.

TABLE I
% REDUCTION IN THE NUMBER OF GLOBAL MEMORY LOADS

Structure	Without HCP	With HCP
H Helix myoglobin	50%	32%
nucleosome core paricle	50%	62%
chaperonin GroEL	50%	84%
virus capsid	50%	96%

V. EXPERIMENTAL SETUP

To illustrate the scalability of our implementation, we have used four different input structures with varied sizes. The characteristics of these structures are presented in Table II. The host machine consists of an E8200 Intel Quad core running at 2.33 GHz with 4 GB DDR2 SDRAM. Programming and access to the GPU was provided by CUDA 3.2 toolkit and SDK with the NVIDIA driver version 256.40. We ran our tests on a NVIDIA GTX280 graphics card with GT200 GPU which belongs to compute capability 1.3.

TABLE II
CHARACTERISTICS OF STRUCTURES

Structure	# Atoms	# Vertices
H helix myoglobin, 1MBO	382	5,884
nuclesome core particle, 1KX5	25,086	258,797
chaperonin GroEL, 2EU1	109,802	898,584
virus capsid, 1A6C	476,040	593,615

VI. RESULT

In this section, we present an analysis of performance as well as accuracy of our computational kernel on both CPU and GPU platforms. Section VI-A presents the execution times of three different CPU versions and one GPU version; (i) basic serial CPU version, (ii) CPU serial version optimized with `-O3` flag of the gnu/gcc compiler, (iii) CPU serial version optimized with *hand-tuned* SSE instructions and (iv) CUDA optimized GPU version. Execution times with and without the use of HCP approximation algorithm are presented. All the numbers presented are an average of 10 runs performed on each platform. For HCP, the 1st level threshold was set to 10Å and the 2nd level threshold was fixed at 70Å.

In Section VI-B, we demonstrate how accurate our results are when compared to the CPU implementation.

A. Performance

In Table III, we present the execution times on both CPU and GPU. From the table, we note that for non_HCP, speedup of around 2x over basic CPU version is achieved by using the `-O3` compiler flag while almost another 2x speedup is obtained when optimized with *hand-tuned SSE*. SSE uses a 128-byte vectorized float data structure which consists of four 32-byte floats, thus, enabling it to perform the same computation on these four floats in parallel. However, there is a hidden cost for vectorizing the code which is why our reported speedup is not close to 4x. When HCP is used, the speedup is only 1.4x because HCP reduces the number of computations to be performed by a large amount, not sufficient to overcome the cost of creation of the vector.

TABLE III
EXECUTION TIMES (SECONDS)

	CPU (Basic)	CPU (-O3)	CPU (-O3+SSE)	GPU
Mb.Hhelix_non_HCP	0.43	0.21	0.09	0.06
Mb.Hhelix_HCP	0.20	0.09	0.03	0.05
nuclesome_non_HCP	1376.00	657.00	371.00	3.30
nuclesome_HCP	45.00	21.00	13.00	0.24
2eu1_non_HCP	21395.00	11048.00	5752.00	53.00
2eu1_HCP	223.00	110.00	73.00	1.24
capsid_non_HCP	62499.00	29673.00	16921.00	150.00
capsid_HCP	115.00	50.00	44.00	0.89

From the table, we note that the speedup due to GPU alone when compared against non_HCP CPU version, is almost constant for all three structures barring Mb.Hhelix. This is due to the fact that Mb.Hhelix is a very small structure and not enough threads are required to be executed on the GPU, resulting in few SMs to remain idle and hence, under-utilizing the GPU. For other structures the threshold of the number of threads to be executed is met and almost similar speedup is achieved for both `-O3` and `-O3+SSE`. The observed speedup is around *200x* and around *110x* respectively.

Total application speedup due to the combined power of GPU and HCP increases with the increase in the size of the structure. Virus capsid has the largest number of atoms among the four, therefore, it reports largest speedup also; *33,000x* in case of `-O3` and *19,000x* in case of `-O3+SSE`. HCP introduces a large number of divergent branches because of a check that has to be performed to apply the approximation. This adversely affects performance on the GPU, recollect that a GPU has more transistors devoted to computation and fewer devoted to managing control-flow as mentioned in Section III-B. As the number of molecular components in the structure increases, the number of computations to be performed also increases and hence, the cost of introduction of divergent branches is more effectively overcome.

For the non_HCP version of the algorithm, divergent branches do not come into play as all computations are exact

atom-atom, hence, similar speedup for all structures. Divergent branches are also the reason that the speedup achieved due to HCP is much more on the CPU than on the GPU. If we look at the execution times of virus capsid in Table III, HCP achieves a speedup of 380x for `-O3+SSE` version on CPU, while on the GPU, only 170x is achieved.

TABLE IV
RELATIVE RMS ERROR

Structure	Version	Relative RMSE
H helix myoglobin	CPU with HCP	0.215821
	GPU	0.000030
	GPU with HCP	0.236093
nuclesome core particle	CPU with HCP	0.022950
	GPU	0.000062
	GPU with HCP	0.022853
chaperonin GroEL, 2eu1	CPU with HCP	0.008799
	GPU	0.000042
	GPU with HCP	0.008816
virus capsid	CPU with HCP	0.015376
	GPU	0.000173
	GPU with HCP	0.015273

B. Accuracy

Because we used single-precision arithmetic in our GPU experiments, we present an estimate of how much it affects the accuracy of the results. We compute the relative *root mean squared (RMS) error* against double precision on the CPU and present the results in Table IV. From the table, we note that the error introduced by the GPU itself is fairly negligible when compared to the error introduced by HCP on the CPU. Thus, the total error due to HCP and GPU is almost equivalent to the error on the CPU and hence, we can safely conclude that single precision on the GPU does not jeopardize the accuracy of our results. The errors presented can be deemed acceptable for the computation of molecular surface potential but may be unsatisfactory for molecular dynamics as in that case, the error would accumulate after each time step of simulation.

VII. CONCLUSION

Due to the need to understand biomolecular function, analyzing biomolecular properties, such as non-bonded forces, is of critical importance. In this paper, we focus specifically on calculating the electrostatic surface potential of non-bonded forces as it is a large portion of the computational complexity. Approaches to reduce this complexity include improving the algorithm or improving the underlying hardware upon which the algorithm is run. In this paper, we do both, and hence, present the effect of using a multi-scale approximation algorithm for calculating electrostatic surface potential and mapping it onto a GPU. The end result is a *19,000*-fold speed-up in calculating the electrostatic surface potential on a GPU over a traditional hand-tuned CPU implementation.

ACKNOWLEDGMENT

This work was supported in part by NSF grants CNS-0915861 and CNS-0916719, a NVIDIA Professor Partnership Award, and the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program. We are grateful to Alexey Onufriev and his group for making us familiar with HCP approximation.

REFERENCES

[1] M. Perutz, *Electrostatic Effect in Proteins*, Science **201**, 1187, 1978.
[2] N.A. Baker and J.A. McCammon, *Electrostatic Interactions*, Structural Bioinformatics - Weissig and Bourne, Wiley, New York, 2002.
[3] B. Honig and A. Nichols, *Classical Electrostatics in Biology and Chemistry*, Science **268**, 1144, 1995.
[4] G. Szabo, G. Eisenman, S. McLaughlin and S. Krasne, *Ionic Probes of Membrane Structures*, Annual N.Y. Academy of Sciences **195**, 273, 1972.
[5] F.B. Sheinerman, R. Norel and B. Honig, *Electrostatic Aspects of Protein-Protein Interactions*, Current Opinion in Structural Biology **10**, 153, 2000.
[6] A. Onufriev, A. Smondyrev and D. Bashford, *Protein Affinity Changes Driving Unidirectional Proton Transport in the Bacteriorhodopsin Photocycle*, Journal of Molecular Biology **332**, 1183, 2003.
[7] B. Roux and T. Simonson, *Implicit Solvent Models*, Biophysical Chemistry **78**, 1, 1999.
[8] J.G. Kirkwood, *Theory of Solutions of Molecules Containing Widely Seperated Charges with Special Application to Zwitterions*, Journal of Chemical Physics **2**, 351, 1934.
[9] J.C. Gordon, A.T. Fenley and A.O. Onufriev, *An Analytical Approach to Computing Biomolecular Electrostatic Potential II* Journal of Chemical Physics **129**, 075102, 2008.
[10] A.M. Ruvinsky and I.A. Vakser, *Interaction Cutoff Effect on Ruggedness of Protein-Protein Energy Landscape*, Proteins: Structure, Function and Bioinformatics **70**, 1498, 2008.
[11] W. Cai, S. Deng and D. Jacobs, *Extending the Fast Multipole Method to Charges Inside or Outside a Dielectric Sphere*, Journal of Computational Physics **223**, 846, 2007
[12] T. Darden, D. York, L. Pedersen, *Particle mesh Ewald: An N-log(N) Method for Ewald Sums in Large Systems*, Journal of Chemical Physics **98**, 10089, 1993.
[13] R. Anandakrishnan and A.O. Onufriev, *An N-log(N) Approximation Based on the Natural Organization of Biomolecules for Speeding Up the Computation of Long Range Interactions*, Journal of Computational Chemistry, in press, 2009.
[14] A.O. Onufriev, *Implicit Solvent Models in Molecular Dynamics Simulations: A Brief Overview*, Annual Reports in Computational Chemistry **4**, 125, 2008.
[15] C.I. Rodrigues, D.J. Hardy, J.E. Stone, K. Schulten and W.W. Hwu, *GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications*, Computing Frontiers, 2008.
[16] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco and K. Schulten, *Accelerating Molecular Modeling Applications with Graphics Processors*, Journal of Computational Chemistry **28**, 2618, 2007.
[17] D.J. Hardy, J.E. Stone and K. Schulten, *Multilevel Summation of Electrostatic Potentials using Graphics Processing Units*, Parallel Computing **35**, 164, 2009.
[18] R. Anandakrishnan, T. Scogland, A.T. Fenley, J. Gordon, W. Feng and A. Onufriev, *Accelerating Electrostatic Surface Potential Calculation with Multiscale Approximation on Graphics Processing Units.*, Biopolymers CS Tech Reports, http://eprints.cs.vt.edu/archive/00001081/
[19] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, B.M. Bruns and V.S. Pande, *Accelerating Molecular Dynamic Simulation on Graphics Processing Units*, Journal of Computational Chemistry **30**, 864, 2009.
[20] L. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Vardarajan and K. Schulten, *NAMD2: Greater Scalability for Parallel Molecular Dynamics*, Journal of Computational Physics **151**, 238, 1999.
[21] A.T. Fenley, J.C. Gordon and A.O. Onufriev, *An Analytical Approach to Computing Biomolecular Electrostatic Potential I* Journal of Chemical Physics **129**, 075102, 2008.
[22] *CUDA Programming Guide*, http://developer.nvidia.com/object/cuda_3_0_downloads.html
[23] S. Ryoo, C.I. Rodrigues, S.S. Stone, S.S. Baghsorkhi, S. Ueng, J.A. Stratton and W.W. Hwu, *Program Optimization Space Pruning for a Multithreaded GPU*, International Symposium on Code Generation and Optimization, 2008.