WILEY InterScience® DISCOVER SOMETHING GREAT

Accelerating Molecular Modeling Applications with Graphics Processors

JOHN E. STONE,[1*] JAMES C. PHILLIPS,[1*] PETER L. FREDDOLINO,[1,2*] DAVID J. HARDY,[1*]
LEONARDO G. TRABUCO,[1,2] KLAUS SCHULTEN[1,2,3]

[1]*Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, Illinois, 61801*
[2]*Center for Biophysics and Computational Biology, University of Illinois at Urbana-Champaign, Urbana, Illinois, 61801*
[3]*Department of Physics, University of Illinois at Urbana-Champaign, Urbana, Illinois, 61801*

Received 5 April 2007; Revised 27 June 2007; Accepted 30 July 2007
DOI 10.1002/jcc.20829
Published online 25 September 2007 in Wiley InterScience (www.interscience.wiley.com).

Abstract: Molecular mechanics simulations offer a computational approach to study the behavior of biomolecules at atomic detail, but such simulations are limited in size and timescale by the available computing resources. State-of-the-art graphics processing units (GPUs) can perform over 500 billion arithmetic operations per second, a tremendous computational resource that can now be utilized for general purpose computing as a result of recent advances in GPU hardware and software architecture. In this article, an overview of recent advances in programmable GPUs is presented, with an emphasis on their application to molecular mechanics simulations and the programming techniques required to obtain optimal performance in these cases. We demonstrate the use of GPUs for the calculation of long-range electrostatics and nonbonded forces for molecular dynamics simulations, where GPU-based calculations are typically 10–100 times faster than heavily optimized CPU-based implementations. The application of GPU acceleration to biomolecular simulation is also demonstrated through the use of GPU-accelerated Coulomb-based ion placement and calculation of time-averaged potentials from molecular dynamics trajectories. A novel approximation to Coulomb potential calculation, the multilevel summation method, is introduced and compared with direct Coulomb summation. In light of the performance obtained for this set of calculations, future applications of graphics processors to molecular dynamics simulations are discussed.

© 2007 Wiley Periodicals, Inc. J Comput Chem 28: 2618–2640, 2007

Key words: GPU computing; CUDA; parallel computing; molecular modeling; electrostatic potential; multilevel summation; molecular dynamics; ion placement; multithreading; graphics processing unit

Introduction

Molecular mechanics simulations of biomolecules, from their humble beginnings simulating 500-atom systems for less than 10 ps,[1] have grown to the point of simulating systems containing millions of atoms[2,3] and up to microsecond timescales.[4,5] Even so, obtaining sufficient temporal sampling to simulate significant motions remains a major problem,[6] and the simulation of ever larger systems requires continuing increases in the amount of computational power that can be brought to bear on a single simulation. The increasing size and timescale of such simulations also require ever-increasing computational resources for simulation setup and for the visualization and analysis of simulation results.

Continuing advances in the hardware architecture of graphics processing units (GPUs) have yielded tremendous computational power, required for the interactive rendering of complex imagery for entertainment, visual simulation, computer-aided design, and scientific visualization applications. State-of-the-art GPUs employ IEEE floating point arithmetic, have on-board memory capacities as large as the main memory systems of some personal computers, and at their peak can perform over 500 billion floating point operations per second. The very term "graphics processing unit" has replaced the use of terms such as graphics accelerator and video board in common usage, indicating the increased capabilities, performance, and autonomy of current generation devices. Until recently, the computational power of GPUs was very difficult to harness for any but graphics-oriented algorithms due to limitations in hardware architecture and, to a lesser degree, due to a lack of general purpose application pro-

*The authors contributed equally

Correspondence to: K. Schulten; e-mail: kschulte@ks.uiuc.edu; web: http://www.ks.uiuc.edu/

Contract grant sponsor: National Institutes of Health; contract grant number: P41-RR05969

© 2007 Wiley Periodicals, Inc.

gramming interfaces. Despite these difficulties, GPUs were successfully applied to some problems in molecular modeling.[7–9] Recent advances in GPU hardware and software have eliminated many of the barriers faced by these early efforts, allowing GPUs to now be used as performance accelerators for a wide variety of scientific applications.

In this article, we present a review of recent developments in GPU technology and initial implementations of three specific applications of GPUs to molecular modeling: ion placement, calculation of trajectory-averaged electrostatic potentials, and nonbonded force evaluation for molecular dynamics simulations. In addition, both CPU- and GPU-based implementations of a new approximation to Coulomb energy calculations are described. In the sample applications discussed here, GPU-accelerated implementations are observed to run 10–100 times faster than equivalent CPU implementations, although this may be further improved by future tuning. The ion placement and potential averaging tools described here have been implemented in the molecular visualization program VMD,[10] and GPU acceleration of molecular dynamics has been implemented in NAMD.[11] The additional computing power provided by GPUs brings the possibility of a modest number of desktop computers performing calculations that were previously only possible with the use of supercomputing clusters; aside from molecular dynamics, a number of other common and computationally intensive tasks in biomolecular modeling, such as Monte Carlo sampling and multiple sequence alignment, could also be implemented on GPUs in the near future.

Graphics Processing Unit Overview

Until recently, GPUs were used solely for their ability to process images and geometric information at blistering speeds. The data parallel nature of many computer graphics algorithms naturally led the design of GPUs toward hardware and software architectures that could exploit this parallelism to achieve high performance. The steadily increasing demand for advanced graphics within common software applications has allowed high-performance graphics systems to flourish in the commodity market, making GPUs ubiquitous and affordable. Now that GPUs have evolved into fully programmable devices and key architectural limitations have been eliminated, they have become an ideal resource for acceleration of many arithmetic and memory bandwidth intensive scientific applications.

GPU Hardware Architecture

GPU hardware is fundamentally suited to the types of computations required by present-day interactive computer graphics algorithms. Modern GPU hardware designs are far more flexible than previous generations, lending themselves to use by a much broader pool of applications. Where previous generation hardware designs employed fixed function pipelines composed of task-specific logic intended to perform graphics computations, modern GPUs are composed of a large number of programmable processing units capable of integer operations, conditional execution, arbitrary memory access, and other fundamental capabilities required by most application software. The recent evolution of GPU hardware to fully programmable processing units has removed the largest barrier to their increased use for scientific computing.

While far more general purpose than previous generations, current GPU hardware designs still have limitations. GPU hardware is designed to exploit the tremendous amount of data parallelism inherent in most computer graphics algorithms as the primary means of achieving high performance. This has resulted in GPU designs that contain 128 or more processing units, fed by multiple independent parallel memory systems. While the large number of parallel processing units and memory systems accounts for the tremendous computational power of GPUs, it is also responsible for constraining the practical size of memories, caches, and register sets available to each processor.

GPUs are typically composed of groups of single-instruction multiple-data (SIMD) processing units. SIMD processors are composed of multiple smaller processing units that execute the same sequence of instructions in lockstep. The use of SIMD processing units reduces power consumption and increases the number of floating point arithmetic units per unit area relative to conventional multiple-instruction multiple-data (MIMD) architectures. These issues are of paramount importance for GPUs, since the space, power, and cooling available to a GPU are severely constrained relative to what is afforded for CPUs in the host computer. SIMD architectures have had a long history within scientific computing and were used in parallel computers made by IBM,[12] MasPar,[13] Thinking Machines,[14] and others. In the past several years, general purpose CPUs have also incorporated small-scale SIMD processing units for improved performance on computer graphics and scientific applications. These add-on SIMD units are accompanied by CPU instruction set extensions that allow applications to detect and utilize SIMD arithmetic hardware at runtime.

Some of the large SIMD machines made in the past failed to achieve their full performance potential due to bursty patterns of main memory access, resulting in inefficient use of main memory bandwidth. Another weakness of very large SIMD arrays involves handling conditional execution operations that cause the execution paths of individual processors to diverge. Since individual SIMD processors must remain in lockstep, all of the processors in an array must execute both halves of conditional branches and use predication to prevent writing of results on the side of the branch not taken. Modern SIMD processors, and especially GPU designs, make use of hardware multithreading, clusters of smaller SIMD units, virtualized processors, and other techniques to eliminate or at least ameliorate these problems.[15–17]

To illustrate key architectural features of a state-of-the-art GPU, we briefly summarize the NVIDIA GeForce 8800GTX GPU. The GeForce 8800GTX was also the GPU used for the algorithm design, benchmarking, and application results presented in this article. For the sake of brevity, we refer to this architecture as “G80,” which is the name of the actual GPU chip on the GeForce 8800GTX. The G80 GPU chip is the first member of a family of GPUs supporting NVIDIA’s compute unified device architecture (CUDA) for general purpose GPU computation.[18]

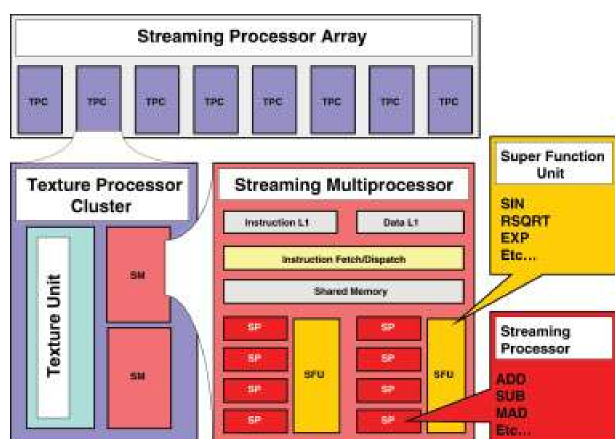

Figure 1. NVIDIA G80 hardware block diagram. 241 × 172 mm (600 × 600 DPI). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

Continuing the recent trend toward larger numbers of processing units, the G80 contains a total of 128 programmable processing units. The individual processors are contained within several hierarchical functional units. The top level of this hierarchy, referred to as the *streaming processor array*, is subdivided into eight identical *texture processor clusters*, each containing two *streaming multiprocessors*. Figure 1 illustrates the groupings of G80 processing units and the functions they perform. Table 1 summarizes the relative numbers of these groupings along with the amount of shared memory, cache resources, and the number of registers apportioned to each. The multiprocessors operate independently of each other, though they share the same texture processing unit. Each multiprocessor contains an eight-way SIMD unit which executes instructions in lockstep on each of its constituent *streaming processors*.

The full G80 streaming processor array can perform up to 330 GFLOPS peak single precision floating point arithmetic. When combined with the processing capabilities of the special graphics functional units, the entire system can perform an aggregate of 550 GFLOPS single precision floating point arithmetic. In addition to this staggering floating point computation rate, the G80 hardware can provide up to 80 GBytes per second of memory bandwidth to keep the multiprocessors fed with data.

Like other GPUs, G80 includes dedicated hardware texturing units, which provide caching with hardware support for multidimensional locality of reference, multiple filtering modes, texture coordinate wrapping and clamping, and many texture formats. A fast globally accessible 64-kB constant memory provides read-only data to all threads at register speed, if all threads read the same data elements concurrently. The constant memory can be an effective tool in achieving high performance for algorithms requiring all threads to loop over identical read-only data.

Some of the unique strengths of GPUs versus CPUs include support for fast hardware instructions implementing square roots, exponentiation, and trigonometric functions, often augmented with extremely fast reduced precision versions. These types of operations are particularly important for the high performance shading computations that GPUs are normally used for. The G80 architecture includes *super function units* to perform these operations that take longer to compute. The super function units allow the streaming processors to continue on, concurrently performing simpler arithmetic operations. Scientific codes that perform many square roots can perform square roots almost for "free," assuming there is enough other work to fully occupy the streaming processors. The overlapping of complex function evaluation with simpler arithmetic operations can provide a tremendous performance boost relative to a CPU implementation.

The G80 architecture implements single precision IEEE-754 floating point arithmetic, but with minor variations from the standard in areas such as floating point exception handling, underflow handling, lack of configurable rounding modes, and the use of multiply-add instructions and reciprocals in the implementation of other operations. Although worth noting, these limitations are in line with the behavior of many optimizing compilers used for CPU-based codes, particularly those targeting CPU SIMD extensions such as SSE. The single precision hardware implementation limits the G80 to applications for which it is adequate, or to those for which techniques such as Kahan summation or single–single arithmetic[19,20] can be used to improve numerical precision at the cost of performing more operations. The CUDA software architecture is designed to support native IEEE double precision floating point, and the first devices providing this functionality are expected to become available in 2007.

The G80 architecture differs in several important ways from prior GPU designs. G80 is capable of performing both *scatter* and *gather* memory operations, allowing programs to access arbitrary memory locations. Previous generation GPU designs were limited to gather operations supporting a very restricted range of memory reference patterns. Accesses to global memory incur hundreds of clock cycles of latency, which the G80 can hide from the application by dynamically scheduling other runnable threads onto the same processor while the original thread waits for its memory operations to complete. The G80 has a multithreaded architecture, which allows well-written computational kernels to take advantage of both the tremendous arithmetic capability and the high memory bandwidth of the GPU by

Table 1. NVIDIA GeForce 8800GTX Hardware Specifications.

Texture processor clusters (TPC)	8
Streaming multiprocessors (SM) per TPC	2
Super function units (SFU) per SM	2
Streaming processors (SP) per SM	8
Total SPs in entire processor array	128
Peak floating point performance GFLOPS	330, 550 w/Tex units
Floating point model	Single precision IEEE-754
Integer word size	32-bit
Global memory size	768 MB
Constant memory	64 kB
Shared memory per SM	16 kB
Texture cache per SM	8 kB, 2-D locality
Constant cache per SM	8 kB
Registers per SM	8192 registers
Aggregate memory bandwidth	80 GB/s

overlapping arithmetic and memory operations from multiple hardware threads. The G80 architecture virtualizes processing resources, mapping the set of independently computed thread blocks onto the available pool of multiprocessors and provides the means for multiple threads within a thread block to be run on a single processing unit.[21] The clustering of streaming processors into relatively small SIMD groups limits the negative impact of branch divergence to just the threads concurrently running on a single streaming multiprocessor.

The G80 incorporates a fast 16-kB shared memory area for each separate multiprocessor. Threads can cooperatively load and manipulate blocks of data into fast register-speed shared memory, amortizing costlier reads from the large global memory. Accesses to the shared memory area are coordinated through the use of a thread barrier synchronization primitive, guaranteeing that all threads within a multiprocessor have completed their shared memory updates before other threads begin accessing results. The shared memory area is a key architectural feature of the G80 that allows programmers to deviate from the strictly streaming model of GPU computation required with previous generation GPU architectures.

GPU Application Programming Interfaces

As GPU hardware became progressively more programmable, new software interfaces were required to expose their programmability to application software. The earliest GPU programming interfaces were based on hardware-independent pseudo-assembly language instructions, making it possible to write powerful shading and texturing routines that far exceeded the capabilities provided by fixed-function hardware. Graphics applications refer to these programs executing on the GPU as *shaders*, a name suggestive of the functions they perform in that context. These pseudo-assembly programs were just-in-time compiled into native machine instructions at runtime by the graphics driver and were then executed by the GPU, replacing the use of some of the stages of the traditional fixed-function graphics pipeline. Because of the tedious nature of writing complex shaders at the assembly language level, high-level shading languages were introduced, allowing shaders to be written in C-like languages with extensions for vector and matrix types, and including libraries of highly optimized graphics functions. Up to this stage of development, using a GPU for nongraphical purposes required calculations to be expressed in terms of an image rendering process. Applications using this methodology "draw" the computation, with the results contained in a grid of pixels which are read back to the host CPU and interpreted.

The hurdles posed by the limitations of previous generation GPU hardware and software made early forays into GPU acceleration of general purpose algorithms an arduous process for anyone lacking extensive graphics programming experience. Subsequently, Buck et al. introduced a GPU-targeted version of Brook, a machine independent stream programming language based on extensions to C.[22,23] The Brook stream programming abstraction eliminated the need to view computations in terms of graphics drawing operations and was the basis for several early successes with GPU acceleration of molecular modeling and bioinformatics applications.[7,24]

Over the past few years, data parallel or streaming implementations of many fundamental algorithms have been designed or adapted to run on GPUs.[25] State-of-the-art GPU hardware and software developments have finally eliminated the need to express general purpose computations in terms of rendering and are now able to represent the underlying hardware with abstractions that are better suited to general purpose computation. Several high-level programming toolkits are currently available, which allow GPU-based computations to be described in more general terms as streaming, array-oriented, or thread-based computations, all of which are more convenient abstractions to work with for non-graphical computations on GPUs.

The work described in this article is based on the CUDA[18] GPU programming toolkit developed by NVIDIA for their GPUs. CUDA was selected for the implementations described in the article due to its relatively thin and lightweight design, its ability to expose all of the key hardware capabilities (e.g., scatter/gather, thread synchronizations, complex data structures consisting of multiple data types), and its ability to extract extremely high performance from the target NVIDIA GeForce 8800GTX GPUs.

The CUDA programming model is based on the decomposition of work into *grids* and *thread blocks*. Grids decompose a large problem into thread blocks which are concurrently executed by the pool of available multiprocessors. Each thread block contains from 64 to 512 threads, which are concurrently executed by the processors within a single multiprocessor. Since each multiprocessor executes instructions in a SIMD fashion, each thread block is computed by running a group of threads, known as a *warp*, in lockstep on the multiprocessor.

The abstractions and virtualization of processing resources provided by the CUDA thread block programming model allow programs to be written with GPUs that exist today but to scale to future hardware designs. Future CUDA-compatible GPUs may contain a large multiple of the number of streaming multiprocessors in current generation hardware. Well-written CUDA programs should be able to run unmodified on future hardware, automatically making use of the increased processing resources.

Target Algorithms

As a means of exploring the applicability of GPUs for the acceleration of molecular modeling computations, we present GPU implementations of three computational kernels that are representative of a range of similar kernels employed by molecular modeling applications. While these kernels are interesting test cases in their own right, they are only a first glimpse at what can be accomplished with GPU acceleration.

Direct Coulomb Summation

What we here refer to as direct Coulomb summation is simply the brute-force calculation of the Coulomb potential on a lattice, given a set of atomic coordinates and corresponding partial charges. Direct Coulomb summation is an ideal test case for GPU computation due to its extreme data parallelism, high arithmetic intensity, simple data structure requirements, and the ease

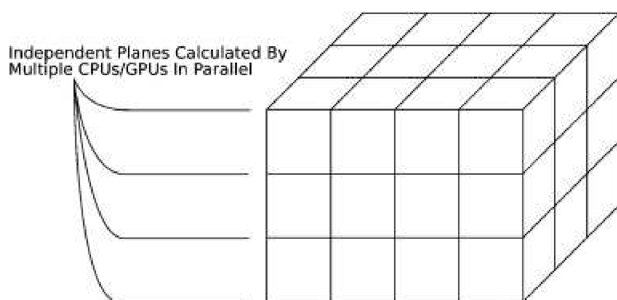

Figure 2. Decomposition of potential map into slices for parallel computation on multiple GPUs. 190 × 91 mm (600 × 600 DPI).

with which its performance and numerical precision can be compared with optimized CPU implementations. The algorithm is also a good test case for a GPU implementation, as many other grid-based function summation algorithms map to very similar CUDA implementations with only a few changes.

No distance-based cutoffs or other approximations are employed in the direct summation algorithm, so it will be used as the point of reference for numerical precision comparisons with other algorithms. A rectangular lattice is defined around the atoms with a specified boundary padding, and a fixed lattice spacing is used in all three dimensions. For each lattice point i located at position $\mathbf{r}_i$, the Coulomb potential V_i is given by

$$V_i = \sum_j \frac{q_j}{4\pi\varepsilon_0\ \varepsilon(r_{ij})r_{ij}}, \qquad (1)$$

with the sum taken over the atoms, where atom j is located at $\mathbf{r}_j$ and has partial charge q_j, and the pairwise distance is $r_{ij} = |\mathbf{r}_j - \mathbf{r}_i|$. The function $\varepsilon(r)$ is a distance-dependent dielectric coefficient, and in the present work will always be defined as either $\varepsilon(r) = \kappa$ or $\varepsilon(r) = \kappa r$, with κ constant. For a system of N atoms and a lattice consisting of M points, the time complexity of the direct summation is $O(MN)$.

The potential map is easily decomposed into planes or slices, which translate conveniently to a CUDA grid and can be independently computed in parallel on multiple GPUs, as shown in Figure 2. Each of the 2-D slices is further decomposed into CUDA thread blocks, which are scheduled onto the available array of streaming multiprocessors on the GPU. Each thread block is composed of 64–256 threads depending on the implementation of the CUDA kernel and the resulting give-and-take between the number of concurrently running threads and the amount of shared memory and register resources each thread consumes. Figure 3 illustrates the decomposition of the potential map into a CUDA grid, thread blocks, individual threads, and the potential values calculated by each thread.

Since the direct summation algorithm requires rapid traversal of either the voxels in the potential map or the atoms in the structure, the decision of which should become the inner loop was determined by the architectural strengths of the CUDA hardware and software. CUDA provides a small per-multiprocessor constant memory that can provide operands at the same speed as reading from a register when all threads read the same operands at the same time. Since atoms are read-only data for the purposes of the direct Coulomb summation algorithm, they are an ideal candidate for storage in the constant cache. GPU constant memory is small and can only be updated by the host CPU, and so multiple computational kernel invocations are required to completely sum the potential contributions from all of the atoms in the system. In practice, just over 4000 atom coordinates and charges fit into the constant memory, but this is more than adequate to amortize the cost of executing a kernel on the GPU. By traversing atoms in the innermost loop, the algorithm affords tremendous opportunity to hide global memory access latencies that occur when reading and updating the summed potential map values by overlapping them with the inner atom loop arithmetic computations.

Performance of the direct Coulomb summation algorithm can be greatly improved through the observation that components of the per-atom distance calculations are constant for individual planes and rows within the map. By evaluating the potential energy contribution of an atom to several points in a row, these values are reused, and memory references are amortized over a larger number of arithmetic operations. By far the costliest arithmetic operation in the algorithm is the reciprocal square root, which unfortunately cannot be eliminated. Several variations of the basic algorithm implement optimization strategies based on these observations, improving both CPU and GPU implementations. The main optimization approach taken by CPU implementations is the reduction or elimination of floating point arithmetic operations through precalculation and maximized performance of the CPU cache through sequential memory accesses with unit stride. Since the G80 has tremendous arithmetic capabilities, the strategy for achieving peak performance revolves around keeping the arithmetic units fully utilized, overlapping arithmetic and memory operations, and making concurrent use of the independent constant, shared, and global memory subsystems to provide data to the arithmetic units at the required rate. Although loading atom data from constant memory can be as fast as reading

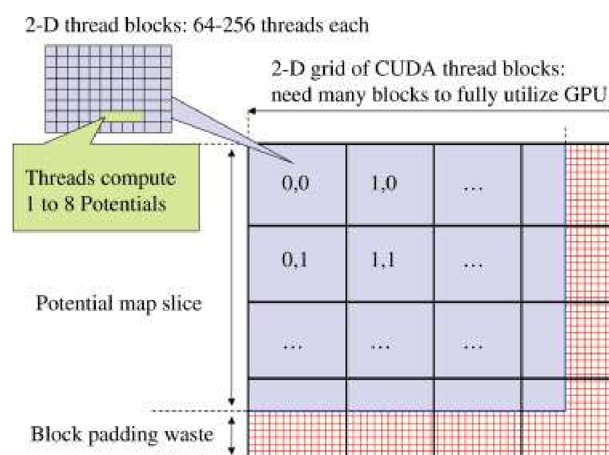

Figure 3. Decomposition of potential map slice into CUDA grids and thread blocks. 235 × 172mm (600 × 600 DPI). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

Table 2. Direct Coulomb Summation Kernel Performance Results.

Kernel	Normalized performance vs GNU GCC	Normalized performance vs Intel C	Atom evals per second	GFLOPS
CPU-GCC-SSE	1.0	0.052	0.046 billion	0.28
CPU-ICC-SSE	19.3	1.0	0.89 billion	5.3
CUDA-Simple	321	16.6	14.8 billion	178
CUDA-Precalc	360	18.6	16.6 billion	166
CUDA-Unroll4x	726	37.5	33.4 billion	259
CUDA-Unroll8x	752	38.9	34.6 billion	268
CUDA-Unroll8y	791	40.9	36.4 billion	191

from a register (described above), it costs instruction slots and decreases the overall arithmetic rate.

The performance results in Table 2 are indicative of the performance levels achievable by a skilled programmer using a high-level language (C in this case) without resorting to the use of assembly language. All benchmarks were run on a system containing a 2.6-GHz Intel Core 2 Extreme QX6700 quad core CPU running 32-bit Red Hat Enterprise Linux version 4 update 4. Tests were performed on a quiescent system with no windowing system running. The CPU benchmarks were performed using a single core, which is a best case scenario in terms of the amount of system memory bandwidth available to that core, since multicore CPU cores share a single front-side memory bus. CUDA benchmarks were likewise performed on a single GeForce 8800GTX GPU.

The CPU results included in Table 2 are the result of highly-tuned C versions of the algorithm, with arrays and loop strides explicitly arranged to allow the use of SSE instructions for peak performance. The results show an enormous difference between the performance of code compiled with GNU GCC 3.4.6 and with the Intel C/C++ Compiler (ICC) version 9.0. Both of these tests were performed enabling all SSE SIMD acceleration optimizations for each of the compilers. The Intel compilers make much more efficient use of the CPU through software pipelining and loop vectorization. The best performing CPU kernels perform only six floating point operations per iteration of the atom potential evaluation loop. Since the innermost loop of the calculation is so simple, even a small difference in the efficiency of the resulting machine code can be expected to account for a large difference in performance. This was also observed to a lesser degree with CUDA implementations of the algorithm. The CPU kernels benefit from manual unrolling of the inner loop to process eight potential values at a time, making significant reuse of atom coordinates and precomputed distance vector components, improving performance by a factor of two over the best non-unrolled implementation. It should be noted that although the results achieved by the Intel C compiler show a significant performance advantage versus GNU GCC, the resulting executables are not necessarily able to achieve this level of performance on non-Intel processors. As a result of this, GNU GCC is frequently used to compile executables of scientific software that is also expected to run on CPUs from other manufacturers.

The CUDA-Simple implementation loops over atoms stored in constant memory, without any loop unrolling or data reuse optimizations. It can compute 14.8 billion atom Coulomb potential values per second on the G80, exceeding the fastest single-threaded CPU version (CPU-ICC-SSE) by a factor of 16, as shown in Table 2. Given the inherent data parallelism of the direct summation algorithm and the large number of arithmetic units and significant memory bandwidth advantages held by the GPU, a performance ratio of this magnitude is not surprising. Although this level of performance improvement is impressive for an initial implementation, the CUDA-Simple kernel achieves less than half of the performance that the G80 is capable of.

As with the CPU implementations, the CUDA implementation benefits from several obvious optimizations that eliminate redundant arithmetic and, more importantly, those that eliminate redundant memory references. The CUDA-Precalc kernel is similar to CUDA-Simple except that it precalculates the Z component of the squared distance vector for all atoms for an entire plane at a time, eliminating two floating point operations from the inner loop.

The CUDA-Unroll4x kernel evaluates four values in the X lattice direction for each atom it references as well as reusing the summed Y and Z components of the squared distance vector for each group of four lattice points, greatly improving the ratio of arithmetic operations to memory references. The CUDA-Unroll4x kernel achieves a performance level just shy of the best result we were able to obtain, for very little time invested. The CUDA-Unroll4x kernel uses registers to store all intermediate sums and potential values, allowing thousands of floating point operations to be performed for each slow global memory reference. Since atom data is used for four lattice points at a time, even loads from constant memory are reduced. This type of optimization works well for kernels that otherwise use a small number of registers, but will not help (and may actually degrade) the performance of kernels that are already using a large number of registers per thread. The practical limit for unrolling the CUDA-Unroll4x kernel was four lattice points, as larger unrolling factors greatly increased the register count and prevented the G80 from concurrently scheduling multiple thread blocks and effectively hiding global memory access latency.

The remaining two CUDA implementations attempt to reduce per-thread register usage by storing intermediate values in the G80 shared memory area. By storing potential values in shared memory rather than in registers, the degree of unrolling can be increased up to eight lattice points at a time rather than four, using approximately the same number of registers per thread. One drawback that occurs as a result of unrolling is that the size of the computational "tiles" operated on by each thread block increases linearly with the degree of inner loop unrolling, summarized in Table 3. If the size of the tiles computed by the thread blocks becomes too large, a correspondingly larger amount of performance is lost when computing potential maps that are not evenly divisible by the tile size, since threads at the edge operate on padding values which do not contribute to the final result. Similarly, if the total number of CUDA thread blocks does not divide evenly into the total number of streaming multiprocessors on the GPU, some of the multiprocessors will become idle as the last group of thread blocks is processed,

Table 3. Data Decomposition and GPU Resource Usage of Direct Coulomb Summation Kernels.

Kernel	Block dimensions	Tile dimensions	Registers per thread	Shared memory
CUDA-Simple	16 × 16 × 1	16 × 16 × 1	12	0
CUDA-Precalc	16 × 16 × 1	16 × 16 × 1	10	0
CUDA-Unroll4x	4 × 16 × 1	16 × 16 × 1	20	0
CUDA-Unroll8x	4 × 32 × 1	32 × 32 × 1	21	4100 bytes
CUDA-Unroll8y	32 × 4 × 1	32 × 32 × 1	19	4252 bytes

All kernels use the entire 64-kB constant memory to store atom data.

resulting in lower performance. This effect is responsible for the fluctuations in performance observed on the smaller potential map side lengths in Figure 4.

For the CUDA-Unroll8x kernel, shared memory storage is used only to store thread-specific potential values, loaded when the threads begin and stored back to global memory at thread completion. For small potential maps, with an insufficient number of threads and blocks to hide global memory latency, performance suffers. The kernels that do not use shared memory are able to continue performing computations while global memory reads are serviced, whereas the shared memory kernels immediately block since they must read the potential value from global memory into shared memory before beginning their computations.

The CUDA-Unroll8y kernel uses a thread block that is the size of one full thread warp (32 threads) in the X dimension, allowing the global memory reads that fetch initial potential values to be coalesced into the most efficient memory transaction. Additionally, since the innermost potential evaluation loop is unrolled in the Y direction, this kernel is able to precompute both the Y and Z components of the atom distance vector, reusing them for all threads in the same thread block. To share the precomputed values among all the threads, they must be written to shared memory by the first thread warp in the thread block. An additional complication arising in loading values into the shared memory area involves coordinating reads and writes to the shared memory by all of the threads. Fortunately, the direct Coulomb summation algorithm is simple and the access to the shared memory area occurs only at the very beginning and the very end of processing for each thread block, eliminating the need to use barrier synchronizations within the performance-critical loops of the algorithm.

Multilevel Coulomb Summation

For problems in molecular modeling involving more than a few hundred atoms, the direct summation method presented in the previous section is impractical due to its quadratic time complexity. To address this difficulty, fast approximation algorithms for solving the N-body problem were developed, most notably Barnes-Hut clustering[26] and the fast multipole method (FMM)[27–29] for nonperiodic systems, as well as particle-particle particle-mesh (P3M)[30,31] and particle-mesh Ewald (PME)[32,33] for periodic systems. Multilevel summation[34,35] provides an alternative fast method for approximating the electrostatic potential that can be used to compute continuous forces for both nonperiodic and periodic systems. Moreover, multilevel summation is more easily described and implemented than the aforementioned methods.

The multilevel summation method is based on the hierarchical interpolation of softened pairwise potentials, an approach first used for solving integral equations,[36] then applied to long-range charge interactions,[37] and, finally, made suitable for use in molecular dynamics.[34] Here, we apply it directly to eq. (1) for computing the Coulomb potential on a lattice, reducing the algorithmic time complexity to $O(M + N)$. Although the overall algorithm is more complex than the brute force approach, the multilevel summation method turns out to be well-suited to the G80 hardware due to its decomposition into spatially localized computational kernels that permit massive multithreading. We first present the basic algorithm, then benchmark an efficient sequential implementation, and finally present benchmarked results of GPU kernels for computing the most demanding part.

The algorithm incorporates two key ideas: smoothly splitting the potential and approximating the resulting softened potentials on lattices. Taking the dielectric to be constant, the normalized electrostatic potential can be expressed as the sum of a short-range potential (the leading term) plus a sequence of softened potentials (the ℓ following terms),

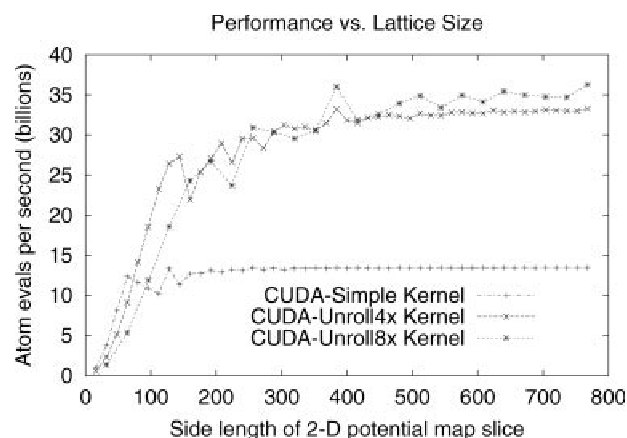

Figure 4. Direct Coulomb potential kernel performance versus lattice size.

$$\frac{1}{r} = \left(\frac{1}{r} - \frac{1}{a}\gamma\left(\frac{r}{a}\right)\right) + \left(\frac{1}{a}\gamma\left(\frac{r}{a}\right) - \frac{1}{2a}\gamma\left(\frac{1}{2a}\right)\right) + \cdots + \left(\frac{1}{2^{\ell-2}a}\gamma\left(\frac{r}{2^{\ell-2}a}\right) - \frac{1}{2^{\ell-1}a}\gamma\left(\frac{r}{2^{\ell-1}a}\right)\right) + \frac{1}{2^{\ell-1}a}\gamma\left(\frac{r}{2^{\ell-1}a}\right), \quad (2)$$

defined in terms of an unparameterized smoothing function γ, chosen so that $\gamma(\rho) = \rho^{-1}$ for $\rho \geq 1$, which means that each of the groupings in the first ℓ terms of eq. (2) vanishes beyond a cutoff distance of $a, 2a, \ldots, 2^{\ell-1}a$, respectively. The softened region $\rho \leq 1$ of γ is chosen so that $\gamma(\sqrt{x^2+y^2+z^2})$ and its partial derivatives are everywhere slowly varying and smooth; this can be accomplished by using even polynomials of ρ, for example, by taking a truncated Taylor expansion of $s^{-1/2}$ about $s = 1$ and then setting $s = \rho^2$. The idea of splitting the $1/r$ kernel into a local short-range part and a softened long-range part has been previously discussed in the multilevel/multigrid literature (e.g., ref. 38). We note that eq. (2) is a generalization of the two-level splitting presented in ref. 34, and its formulation is similar to the interaction partitioning used for multiple time step integration of the electrostatic potential.[39]

Multilevel approximation is applied to eq. (2) by nested interpolation of the telescoping sum starting at each subsequent splitting from lattices of spacings $h, 2h, \ldots, 2^{\ell-1}h$, respectively. This can be expressed concisely by defining a sequence of functions,

$$\begin{aligned} g_*(\mathbf{r},\mathbf{r}') &= \frac{1}{|\mathbf{r}'-\mathbf{r}|} - \frac{1}{a}\gamma\left(\frac{|\mathbf{r}'-\mathbf{r}|}{a}\right), \\ g_k(\mathbf{r},\mathbf{r}') &= \frac{1}{2^k a}\gamma\left(\frac{|\mathbf{r}'-\mathbf{r}|}{2^k a}\right) - \frac{1}{2^{k+1}a}\gamma\left(\frac{|\mathbf{r}'-\mathbf{r}|}{2^{k+1}a}\right), \\ & \qquad k = 0, 1, \ldots, \ell-2, \\ g_{\ell-1}(\mathbf{r},\mathbf{r}') &= \frac{1}{2^{\ell-1}a}\gamma\left(\frac{|\mathbf{r}'-\mathbf{r}|}{2^{\ell-1}a}\right), \end{aligned} \quad (3)$$

and operators I_k that interpolate a function g from a lattice of spacing $2^k h$,

$$I_k g(\mathbf{r},\mathbf{r}') = \sum_{\mu}\sum_{\nu} \phi_\mu^k(\mathbf{r})\; g(\mathbf{r}_\mu^k, \mathbf{r}_\nu^k)\; \phi_\nu^k(\mathbf{r}'), \quad k = 0, 1, \ldots \ell-1, \quad (4)$$

where ϕ_μ^k and ϕ_ν^k are nodal basis functions of points $\mathbf{r}_\mu^k$ and $\mathbf{r}_\nu^k$ defined by

$$\phi_\mu^k(\mathbf{r}) = \Phi\left(\frac{x - x_\mu^k}{2^k h}\right)\Phi\left(\frac{y - y_\mu^k}{2^k h}\right)\Phi\left(\frac{z - z_\mu^k}{2^k h}\right), \quad (5)$$

in terms of a master basis function $\Phi(\xi)$ of unit spacing. The multilevel approximation of eq. (2) can be written as

$$\begin{aligned} \frac{1}{|\mathbf{r}'-\mathbf{r}|} &= (g_* + g_0 + g_1 + \cdots + g_{\ell-2} + g_{\ell-1})(\mathbf{r},\mathbf{r}') \\ &\approx (g_* + I_0(g_0 + I_1(g_1 + \cdots I_{\ell-2}(g_{\ell-2} + I_{\ell-1}g_{\ell-1})\cdots)))(\mathbf{r},\mathbf{r}'). \end{aligned} \quad (6)$$

Using interpolation with local support, the computational work is constant for each lattice point, with the number of points reduced by almost a factor of 8 at each successive level. Summing eq. (6) over all pairwise interactions yields an approximation to the potential energy requiring just $O(M + N)$ operations for N atoms and M points.

This method can be easily extended to use a distance-dependent dielectric coefficient $\varepsilon(r) = \kappa r$, in which case the splitting in eq. (2) (written to two levels) can be expressed as

$$\frac{1}{r^2} = \left(\frac{1}{r^2} - \frac{1}{a^2}\gamma\left(\frac{r^2}{a^2}\right)\right) + \frac{1}{a^2}\gamma\left(\frac{r^2}{a^2}\right), \quad (7)$$

where the short-range part again vanishes by choosing $\gamma(\rho) = \rho^{-1}$ for $\rho \geq 1$. The region $\rho \leq 1$ of $\gamma(x^2 + y^2 + z^2)$ and its partial derivatives are smoothly bounded for all polynomials in ρ, which permits the softening to be a truncated Taylor expansion of ρ^{-1} about $\rho = 1$.

Use of multilevel summation for molecular dynamics involves the approximation of the electrostatic potential function,

$$U(\mathbf{r}_1, \ldots, \mathbf{r}_N) = \frac{1}{2}\sum_i \sum_{j \neq i} \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}}, \quad (8)$$

by substituting eq. (6) for the $1/r$ potential. The atomic forces are computed as the negative gradient of the approximate potential function, for which stable dynamics with continuous forces require that Φ be continuously differentiable. Ref. 35 provides a detailed theoretical analysis of eq. (6), showing asymptotic error bounds of the form

$$\begin{aligned} \text{potential energy error} &< 12 c_p M_p \frac{h^p}{a^{p+1}} + O\left(\frac{h^{p+1}}{a^{p+2}}\right), \\ \text{force component error} &< \frac{4}{3} c'_{p-1} M_p \frac{h^{p-1}}{a^{p+1}} + O\left(\frac{h^p}{a^{p+2}}\right), \end{aligned} \quad (9)$$

where M_p is a bound on the pth order derivatives of γ, the interpolant Φ is assumed exact for polynomials of degree $< p$, and constants c_p and c'_{p-1} depend only on Φ. An analysis of the cost versus accuracy shows that the spacing h of the finest lattice needs to be close to the inter-atomic spacing, generally 2 Å $\leq h \leq 3$ Å, and, for a particular choice of Φ and γ, the short-range cutoff distance a provides control over accuracy, with values in the range 8 Å $\leq a \leq 12$ Å appropriate for dynamics, producing less than 1% relative error in the average force. For these choices of h and a, empirical results suggest improved accuracy by choosing γ to have no more than the C^{p-1} continuity suggested by theoretical analysis, where the lowest errors have been demonstrated with γ instead having $C^{\lceil p/2 \rceil}$ continuity. Ref. 35 also shows that comparable accuracy to FMM and PME is available for multilevel summation through the use of higher order interpolation, while demonstrating stable dynamics using cheaper, lower accuracy approximation.

Our presentation of the multilevel summation algorithm focuses on approximating the Coulomb potential lattice in eq. (1) to offer a fast alternative to direct summation for ion place-

ment and related applications, discussed later. The algorithmic decomposition follows eq. (6) by first dividing the computation, from the initial splitting,

$$V_i \approx e_i^{\text{short}} + e_i^{\text{long}}, \tag{10}$$

into an exact short-range part,

$$e_i^{\text{short}} = \sum_j \frac{q_j}{4\pi\varepsilon_0}\left(\frac{1}{r_{ij}} - \frac{1}{a}\gamma\left(\frac{r_{ij}}{a}\right)\right), \tag{11}$$

and a long-range part approximated on the lattices. The nested interpolation performed in the long-range part is further subdivided into steps that assign charges to the lattice points, from which potentials can be computed at the lattice points, and finally the long-range contributions to the potentials are interpolated from the finest-spaced lattice. These steps are designated as follows:

$$\text{anterpolation:} \quad q_\mu^0 = \sum_j \phi_\mu^0(\mathbf{r}_j) q_j, \tag{12}$$

$$\text{restriction:} \quad q_\mu^{k+1} = \sum_\nu \phi_\mu^{k+1}(\mathbf{r}_\nu^k) q_\nu^k, \quad k = 0, 1, \ldots, \ell-2, \tag{13}$$

$$\text{lattice cutoff:} \quad e_\mu^{k,\text{cutoff}} = \sum_\nu g_k(\mathbf{r}_\mu^k, \mathbf{r}_\nu^k) q_\nu^k, \quad k = 0, 1, \ldots, \ell-1, \tag{14}$$

$$\text{prolongation:} \quad e_\mu^{\ell-1} = e_\mu^{\ell-1,\text{cutoff}}, \quad e_\mu^k = e_\mu^{k,\text{cutoff}} + \sum_\nu \phi_\nu^{k+1}(\mathbf{r}_\mu^k) e_\nu^{k+1}, \quad k = \ell-2, \ldots, 1, 0, \tag{15}$$

$$\text{interpolation:} \quad e_i^{\text{long}} = \sum_\mu \phi_\mu^0(\mathbf{r}_i) e_\mu^0. \tag{16}$$

There is constant work computed at each lattice point (i.e., left-hand side) for eqs. (12)–(16), where the sizes of these constants depend on the choice of parameters Φ, h, and a. To estimate the total work done for each step, we assume that the number of points on the h-spaced lattice is approximately the number of atoms N and that the interpolation is by piecewise polynomials of degree p, giving Φ a stencil width of $p + 1$.

The anterpolation step in eq. (12) uses the nodal basis functions to spread the atomic charges to their surrounding lattice points, with total work proportional to p^3N. The interpolation step in eq. (16) sums from the N lattice points of spacing h to the M Coulombic lattice points of eq. (1), which in practice has a much finer lattice spacing. This means that the total work for interpolation might take as much time as p^3M and require evaluation of the nodal basis functions; however, with an alignment of the two lattices, the ϕ function evaluations have fixed values repeated across the potential lattice points, so a small subset of values can be precomputed and reapplied across the points. A further optimization can be made, because of the regularity of interpolating one aligned lattice to another, in which the partial sums are stored while marching across the lattice in each separate dimension; this lowers the work to pM but requires additional temporary storage proportional to $M^{2/3}$.

The restriction step in eq. (13) is anterpolation performed on a lattice. Since the relative spacings are identical at any lattice point and between any consecutive levels, the nodal basis function evaluations can all be precomputed. A marching technique similar to that for the interpolation step can be employed with total work for the charges at level $k + 1$ proportional to $2^{-3(k+1)}pN$. The prolongation step in eq. (15) is similarly identical to the interpolation step computed between lattice levels, with the same total work requirements as restriction.

The lattice cutoff summation in eq. (14) can be viewed as a discrete version of the short-range computation in eq. (11), with a spherical cutoff radius of $\lceil 2a/h \rceil - 1$ lattice points at every level. The total work required at level k is approximately 2^{-3k} $(2a/h)^3N$. Unlike the short-range computation, the pairwise g_k evaluation is between lattice points; this sphere of "weights" that multiplies the lattice charges is unchanging for a given level and, therefore, can be precomputed. An efficient implementation expands the sphere of weights to a cube padded with zeros at the corners, allowing the computation to be expressed as a convolution at each point of the centered sublattice of charge with the fixed lattice of weights. All of the long-range steps in eqs. (12)–(16) permit concurrency for summations to the individual lattice points that require no synchronization, i.e., are appropriate for the G80 architecture.

The summation for the short-range part in eq. (11) is computed in fewest operations by looping over the atoms and summing the potential contribution from each atom to the points contained within its sphere of radius a. The computational work is proportional to a^3N times the density of the Coulombic lattice, with each pairwise interaction evaluating a square root and the smoothing function polynomial. This turns out to be the most demanding part of the entire computation due to the use of a much finer spacing for the Coulombic lattice. Best performance is obtained by first "sorting" the atoms through geometric hashing[40] so that the order in which the atoms are visited aligns with the Coulombic lattice memory storage. The timings listed in Table 4 show for a representative test problem that the short-range part takes more than twice the entire long-range part. To improve overall performance, we developed CUDA implementations for the short-range computational kernel.

Table 4. Multilevel Coulomb Summation, Sequential Time Profile.

	Time (s)	Percentage of total
Short-range part	50.30	69.27
Long-range part	22.31	30.73
anterpolation	0.05	0.07
restriction, levels 0,1,2,3,4	0.06	0.08
lattice cutoff, level 0	13.89	19.13
lattice cutoff, level 1	1.83	2.52
lattice cutoff, level 2	0.23	0.32
lattice cutoff, levels 3,4,5	0.03	0.04
prolongation, levels 5,4,3,2,1	0.06	0.08
interpolation	6.16	8.49

Unlike the long-range steps, the concurrency available to the sequential short-range algorithm discussed in the preceding paragraph does require synchronization, since lattice points will expect to receive contributions from multiple atoms. Even though the G80 supports scatter operations, with contributions from a single atom written to the many surrounding lattice points and also supports synchronization within a grid block of threads, there is no synchronization support across multiple grid blocks. Thus, a CUDA implementation of the short-range algorithm that loops first over the atoms would need to either separately buffer the contributions from the atoms processed in parallel, with the buffers later summed into the Coulombic lattice, or geometrically arrange the atoms so as to process in parallel only subsets of atoms that have no mutually overlapping contributions, in which the pairwise distance between any two atoms is greater than $2a$.

An alternative approach to the short-range summation that supports concurrency is to invert the loops, first looping over the lattice points and then summing the contributions from the nearby atoms to each point. For this, we seek support from the CPU for clustering nearby atoms by performing geometric hashing of the atoms into grid cells. The basic implementation hashes the atoms and then loops over subcubes of the Coulombic lattice points; the CUDA kernel is then invoked one or more times on each subcube to sum its nearby atomic contributions. The CUDA implementations developed here are similar to those developed for direct summation, where the atom and charge data are copied into the constant memory space, and the threads are assigned to particular lattice points. One major difference is that the use of a cutoff potential requires a branch within the inner loop.

For the implementations tested here, the multilevel summation parameters are fixed as $h = 2$ Å, $a = 12$ Å,

$$\Phi(\xi) = \begin{cases} (1-|\xi|)\left(1+|\xi|-\frac{3}{2}\xi^2\right), & \text{for } |\xi| \le 1, \\ -\frac{1}{2}(|\xi|-1)(2-|\xi|)^2, & \text{for } 1 \le |\xi| \le 2, \\ 0, & \text{otherwise,} \end{cases} \tag{17}$$

$$\gamma(\rho) = \begin{cases} \frac{15}{8} - \frac{5}{4}\rho^2 + \frac{3}{8}\rho^4, & \rho \le 1, \\ 1/\rho, & \rho \ge 1, \end{cases} \tag{18}$$

where Φ is the linear blending of quadratic interpolating polynomials, providing C^1 continuity for the interpolant and where the softened part of γ is the first three terms in the truncated Taylor expansion of $s^{-1/2}$ about $s = 1$, providing C^2 continuity for the functions that are interpolated. We note that compared with other parameters investigated for multilevel summation,[35] these choices for Φ and γ exhibit higher performance but lower accuracy. The accuracy generally provides about 2.5 digits of precision for calculation of the Coulomb potentials, which appears to be sufficient for use in ion placement, since the lattice minima determined by the approximation generally agree with those from direct summation. As noted earlier, improved accuracy can be achieved by higher order interpolation. Also, since continuous forces are not computed, it is unnecessary to maintain a continuously differentiable Φ. For this particular case, Φ could be obtained directly from a cubic interpolating polynomial, which would increase the order of accuracy by one degree, i.e., increasing p in eq. (9) by one, without affecting performance.

Performance testing has been conducted for $N = 200{,}000$ atoms, assigned random charges and positions within a box of length 192 Å. The Coulombic lattice spacing is 0.5 Å, giving $M = 384^3$ points. These dimensions are comparable to the size of the ion placement problem solved for the STMV genome, discussed later in the article. Table 4 gives the timings of this system for an efficient and highly-tuned sequential version of multilevel summation, built using the Intel compiler with SSE options enabled. Special care was taken to compile vectorized inner loops for the short-range kernel.

Four CUDA implementations have been developed for computing the short-range part. The operational workload for each CUDA kernel invocation is chosen as subcubes of 48^3 points from the Coulombic lattice, intended to provide sufficient computational work for a domain that is a multiple of the $a = 12$ Å cutoff distance. The grid cell size for the geometric hashing is kept coarse at 24 Å, matching the dimensions of the subcube blocks, with the grid cell tiling offset by 12 Å in each dimension, so that the subcube block plus a 12-Å cutoff margin is exactly contained by a cube of eight grid cells. The thread block dimension has been carefully chosen to be $4 \times 4 \times 12$ (modified to $4 \times 4 \times 8$ for the MShort-Unroll3z kernel), with the intention to reduce the effects of branch divergence by mapping a 32-thread warp onto a lattice subblock of smallest surface area. An illustration of the domain decomposition and thread block mapping is shown in Figure 5.

Table 5 compares the performance benchmarking of the CUDA implementations, with speedups normalized to the atom evaluation rate of the CPU implementation. The benchmarks were run with the same hardware configuration as used for the previous section. The MShort-Basic kernel has a well-optimized floating point operation count and attempts to minimize register

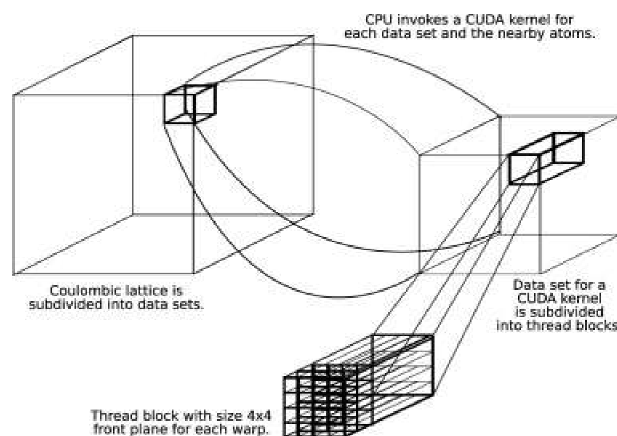

Figure 5. Domain decomposition for short-range computation of multilevel Coulomb summation on the GPU.

Table 5. Multilevel Coulomb Summation, Short-Range Kernel Performance Results.

Kernel	Time (s)	Normalized performance vs. CPU	Atom evals per second	GFLOPS
MShort-CPU-icc	50.30	1.0	0.214 billion	5.62
MShort-Basic	12.31	4.1	0.876 billion	103
MShort-Cluster	11.69	4.3	0.922 billion	108
MShort-Unroll2z	8.44	6.0	1.28 billion	107
MShort-Unroll3z	7.40	6.8	1.46 billion	105

use by defining constants for method parameters and thread block dimensions. However, this implementation is limited by invoking the kernel to process only one grid cell of atoms at a time. Given an atomic density of 1 atom per 10 Å^3 for systems of biomolecules, we would expect up to $24^3/10 = 1382.4$ atoms per grid cell, which is a little over one-third of the available constant memory storage. The MShort-Cluster implementation uses the same kernel as MShort-Basic but has the CPU buffer as many grid cells as possible before each kernel invocation. The difference in performance demonstrates the overhead involved with kernel invocation and copying memory to the GPU.

Much better performance is achieved by combining the MShort-Cluster kernel invocation strategy with unrolling the loops along the *Z*-direction. Each thread accumulates two Coulombic lattice points in MShort-Unroll2z and three points in MShort-Unroll3z. Like the previous kernels, the thread assignment for the unrolling maintains the warp assignments together in adjacent $4 \times 4 \times 2$ blocks of lattice points. These kernels have improved performance but lower GFLOPS due to computing fewer operations. Unfortunately, attempting to unroll a fourth time decreases performance due to the increased register usage.

Although the six-fold speedup for the short-range part improves the runtime considerably, Amdahl's Law limits the speedup for the entire multilevel summation to just 2.4 due to the remaining sequential long-range part. Performance is improved by running the short-range GPU-accelerated part concurrently with the sequential long-range part using two threads on a multicore CPU, which effectively hides the short-range computation and makes the long-range computation the rate-limiting part. Looking back at Table 4, the lattice cutoff step could be parallelized next, followed by the interpolation step, and so on. Even though the algorithmic steps for the long-range part permit concurrency, applying the GPU to each algorithmic step offers diminished performance improvement to the upper lattice levels due to the geometrically decreasing computational work available. The alternative is to devise a kernel that computes the entire long-range part.

Figure 6 compares the performance of multilevel Coulomb summation implementations (using the MShort-Unroll3z kernel) with direct Coulomb summation enhanced by GPU acceleration (using the CUDA-Unroll4x kernel). The number of atoms N is varied, using positions assigned from a random uniform distribution of points from a cubic $10N$ Å^3 volume of space to provide the expected atomic density for a biomolecular system. The size M of the lattice is chosen so as to provide a 0.5-Å spacing with a 10-Å border around the cube of atoms, which is typical for use with ion placement, i.e.,

$$M = \left\lceil 2((10N)^{1/3} + 20) \right\rceil^3. \tag{19}$$

The logarithmic plots illustrate the linear algorithmic behavior of multilevel summation as lines with slope ≈ 1, as compared with the quadratic behavior with slope ≈ 2 lines of direct summation. The flat plots for $N < 1{,}000$ of the GPU accelerated implementations reveal the overhead incurred by the use of GPUs. The multithreaded GPU-accelerated implementation of multilevel summation performs better than GPU-accelerated direct summation starting around $N = 10{,}000$. The sequential CPU multilevel summation begins outperforming the CPU direct summation by $N = 800$ and the 3-GPU direct summation by $N = 250{,}000$.

Molecular Dynamics Force Evaluation

The preceding methods for evaluating the Coulombic potential surrounding a molecule are used in the preparation of biomolecular simulations. GPU acceleration allows more accurate methods, leading to better initial conditions for the simulation. It is, however, the calculation of the molecular dynamics trajectory that consumes nearly all of the computational resources for such work. For this reason we now turn to the most expensive part of a molecular dynamics (MD) simulation: the evaluation of interatomic forces.

Detailed Description

Classical molecular dynamics simulations of biomolecular systems require millions of timesteps to simulate even a few nanoseconds. Each timestep requires the calculation of the net force $\mathbf{F}_i$ on every atom, followed by an explicit integration step to update velocities $\mathbf{v}_i$ and positions $\mathbf{r}_i$. The force on each atom is

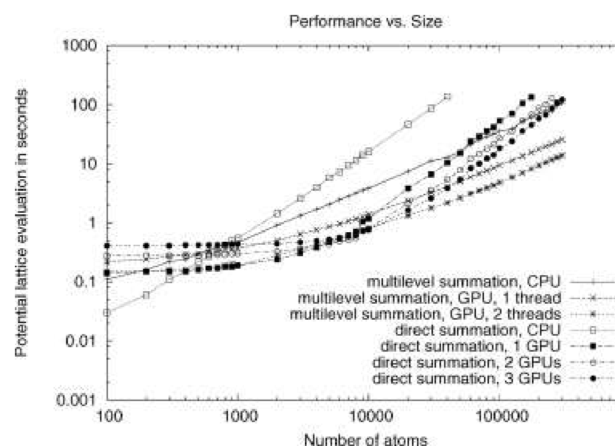

Figure 6. Comparing performance of multilevel and direct Coulomb summation.

the negative gradient of a potential function $U(\mathbf{r}_1, \ldots, \mathbf{r}_N)$. A typical simulation of proteins, lipids, and/or nucleic acids in a box of water molecules comprises 10,000–1,000,000 atoms. To avoid edge effects from a water-vacuum interface, periodic boundary conditions are used to simulate an infinite sea of identical neighbors.

The common potential functions for biomolecular simulation are the sum of bonded terms, describing energies due to covalent bonds within molecules, and nonbonded terms, describing Coulombic and van der Waals interactions both between and within molecules. Bonded terms involve two to four nearby atoms in the same molecule and are rapidly calculated. Nonbonded terms involve every pair of atoms in the simulation and form the bulk of the calculation. In practice, nonbonded terms are restricted to pairs of atoms within a cutoff distance r_c, reducing $N^2/2$ pairs to $\frac{4}{3}\pi r_c^3 \rho N/2$ where ρ is the number of atoms per unit volume (about 0.1 Å^{-3}). The long-range portion of the periodic Coulomb potential can be evaluated in $O(N \log N)$ time by the particle-mesh Ewald (PME) method,[32,33] allowing practical cutoff distances of 8–12 Å for typical simulations.

The functional form of the nonbonded potential is

$$U_{\text{nb}} = \sum_i \sum_{j<i} \left(4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] + \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}} \right). \qquad (20)$$

The parameters ε_{ij} and σ_{ij} are calculated from the combining rules $\varepsilon_{ij} = \sqrt{\varepsilon_i \varepsilon_j}$ and $\sigma_{ij} = \frac{1}{2}(\sigma_i + \sigma_j)$ (some force fields use $\sqrt{\sigma_i \sigma_j}$ instead); ε_i and σ_i are the same for all atoms of the same type, i.e., of the same element in a similar chemical environment. The partial charges q_i are assigned independently of atom type and capture the effects of both ionization (formal charge) and electronegativity (preferential attraction of the shared electrons of a bond to some nuclei over others).

In practical simulations the above nonbonded potential is modified in several ways for accuracy and efficiency. The most significant change is neglect of all pairs beyond the cutoff distance, as mentioned earlier. Atoms crossing the cutoff boundary will cause sharp jumps in the reported energy of the simulation. Since conservation of energy is one measure of simulation accuracy, the potential is modified by a "switching function" sw(r) that brings the potential smoothly to zero at the cutoff distance.

The bonded terms of the potential are parameterized to capture the complete interaction between atoms covalently bonded to each other or to a common atom (i.e., separated by two bonds); such pairs of atoms are "excluded" from the normal nonbonded interaction. The number of excluded pairs is small, about one per bonded term, leading one to consider evaluating the nonbonded forces for all pairs and then subtracting the forces due to excluded pairs in a quick second pass. However, the small separations (close enough to share electrons) of excluded pairs would lead to enormous nonbonded forces, overwhelming any significant digits of the true interaction and leaving only random noise.

When the particle-mesh Ewald method is used to evaluate the Coulomb potential over all pairs and periodic images, the result is scaled by erf (βr_{ij}) (where β is chosen such that erf(βr_c) $\approx$1). Hence, the Coulomb term in the nonbonded calculation must be scaled by $1 - \text{erf}(\beta r_{ij}) \equiv \text{erfc}(\beta r_{ij})$, while the Ewald forces due to excluded pairs must be subtracted off. Note that, unlike the common graphics operation `rsqrt()` implemented in hardware on all GPUs and many CPUs, `erfc()` is implemented in software and is therefore much more expensive. For this reason many molecular dynamics codes use interpolation tables to quickly approximate either `erfc()` or the entire Coulomb potential.

CUDA Implementation

Having presented the full complexity of the nonbonded potential, we now refactor the potential into distance-dependent functions and atom-dependent parameters as

$$U_{\text{nb}} = \sum_i \sum_{j<i} \begin{cases} 0 & r_{ij} > r_c \\ \varepsilon_{ij}\left[\sigma_{ij}^{12}\text{a}(r_{ij}) + \sigma_{ij}^{6}\text{b}(r_{ij})\right] & \\ \quad + q_i q_j \text{c}(r_{ij}) & \text{normal} \\ q_i q_j \text{d}(r_{ij}) & \text{excluded} \end{cases} \qquad (21)$$

where $\text{a}(r) = 4\ \text{sw}(r)/r^{12}$, $\text{b}(r) = -4\ \text{sw}(r)/r^{6}$, $\text{c}(r) = \text{erfc}(\beta r)/4\pi\varepsilon_0 r$, $\text{d}(r) = -\text{erf}(\beta r)/4\pi\varepsilon_0 r$, $\varepsilon_{ij} = \sqrt{\varepsilon_i}\sqrt{\varepsilon_j}$, and $\sigma_{ij} = \frac{1}{2}\sigma_i + \frac{1}{2}\sigma_j$. Thus, to calculate the potential between two atoms requires only $\mathbf{r}$, ε, σ, and q for each atom, four common functions of r, and a means of checking if the pair is excluded.

In a biomolecular simulation, the order of atoms in the input file follows the order of residues in the peptide or nucleic acid chain. Thus, with the exception of rare cross-links such as disulphide bonds, all of the exclusions for a given atom i may be confined to an exclusion range $[i - e_i, i + e_i]$ where $e_i \ll N$. This allows the exclusion status of any pair of atoms to be checked in $O(1)$ time by testing first $|j - i| \leq e_i$ and, if true, element $j - i + e_i$ of an exclusion array $f_i[0\ldots 2e_i]$ of boolean values. We further note that due to the small number of building blocks of biomolecular systems (20 amino acids, four nucleic acid bases, water, ions, a few types of lipids), the number of unique exclusion arrays will be much less than N and largely independent of system size. For example, a 1,000,000-atom virus simulation and a 100,000-atom lipoprotein simulation both have only 700–800 unique exclusion arrays. In addition, the exclusion arrays for hydrogens bonded to a common atom differ only by an index shift and hence may be combined. When stored as single bits, the unique exclusion arrays for most atoms in a simulation will fit in the 8-kB dedicated constant cache of the G80 GPU, with the rest automatically loaded on demand. Thus, exclusion checking for a pair of atoms requires only the two atom indices and the exclusion range and array index for either atom. While four bytes are required for the atom index, the exclusion range and array index can be stored as two-byte values. Counting also $\mathbf{r}$, ε, σ, and q, we see that a total of 32 bytes of data per atom is required to evaluate the nonbonded potential.

In a molecular dynamics simulation, energies are used only as diagnostic information. The equations of motion require only

the calculation of forces. Taking $-\nabla_i U_{\rm nb}$ yields a force on atom i of

$$\mathbf{F}_i^{\rm nb} = \sum_{j \neq i} \frac{\mathbf{r}_j - \mathbf{r}_i}{r_{ij}} \begin{cases} 0 & r_{ij} > r_c \\ \varepsilon_{ij}[\sigma_{ij}^{12} \mathrm{a}'(r_{ij}) + \sigma_{ij}^{6} \mathrm{b}'(r_{ij})] & \\ + q_i q_j \mathrm{c}'(r_{ij}) & \text{normal} \\ q_i q_j \mathrm{d}'(r_{ij}) & \text{excluded} \end{cases} \tag{22}$$

and we note that each atom pair requires only a single force evaluation since $\mathbf{F}_{ij}^{\rm nb} = -\mathbf{F}_{ji}^{\rm nb}$.

We now seek further simplifications to increase performance. Since the `sqrt()` required to calculate r_{ij} is actually implemented on the G80 GPU as `rsqrt()` followed by `recip()`, we can save one special function call by using only r_{ij}^{-1}. To eliminate r_{ij}, define $\mathrm{A}(r^{-1}) = \mathrm{a}'(r)/r$, $\mathrm{B}(r^{-1}) = \mathrm{b}'(r)/r$, $\mathrm{C}(r^{-1}) = \mathrm{c}'(r)/r$, and $\mathrm{D}(r^{-1}) = \mathrm{d}'(r)/r$. Our final formula for efficient nonbonded force evaluation is therefore

$$\mathbf{F}_i^{\rm nb} = \sum_{j \neq i} (\mathbf{r}_j - \mathbf{r}_i) \begin{cases} 0 & r_{ij}^2 > r_c^2 \\ \varepsilon_{ij}\left[\sigma_{ij}^{12} \mathrm{A}(r_{ij}^{-1}) + \sigma_{ij}^{6} \mathrm{B}(r_{ij}^{-1})\right] & \\ + q_i q_j \mathrm{C}\left(r_{ij}^{-1}\right) & \text{normal} \\ q_i q_j \mathrm{D}\left(r_{ij}^{-1}\right) & \text{excluded.} \end{cases} \tag{23}$$

Let $\mathrm{A}(s)$ be interpolated linearly in s from its values A_k along evenly spaced points $s_k \in [r_c^{-1}, \delta^{-1}]$ (and similarly for B, C, and D). For $s > \delta^{-1}$ (corresponding to $r < \delta$), let $\mathrm{A}(s) = \mathrm{A}(\delta^{-1})$, etc., yielding forces proportional to r over $r \in [0, \delta]$. Forces for $r \in [s_k^{-1}, s_{k-1}^{-1}]$ will take the form $r(ar^{-1} + \mathrm{b})$, which simplifies to the linear form $a + rb$. Thus, linear interpolation in r^{-1} of $\mathrm{A}(r^{-1})$ etc. is equivalent to linear interpolation in r of the entire force. The greatest density of interpolation points, and hence the greatest accuracy, is found for $\delta \leq r \ll r_c$. If δ is chosen as the closest physically possible distance between two atoms, then the region of greatest accuracy corresponds to the region of greatest force magnitude and gradient. We also observe that the accuracy of the simulation may be improved by adjusting A_k etc. such that at each interpolation point the integral of the interpolated force exactly matches the energy.

The G80 implements linear interpolation of textures in hardware, which is a perfect fit for efficiently evaluating the distance-dependent functions A, B, C, and D. To use this feature, the texture coordinate r^{-1} must be normalized to [0, 1], which can be accomplished for our purposes by scaling r^{-1} by δ. We observe that choosing $\delta = 1$ Å saves a multiply and a valuable register. The texture hardware can also clamp out-of-range coordinates to [0, 1], resulting in the desired behavior for $r < \delta$. The dedicated 8-kB texture cache can hold 512 four-component elements (2048 single-precision floating-point values), which is sufficient for force interpolation.

A strategy for mapping the nonbonded force calculation now takes shape. The texture unit and cache will be dedicated to force interpolation and the constant cache to exclusion arrays. At this point 16 kB of shared memory and 32 kB of registers remain unused. (Registers hold all scalar automatic variables for a thread and are the largest low-latency memory available on the GPU.)

One of the standard strategies of CUDA programming is minimizing registers-per-thread and shared-memory-per-block to allow more threads and blocks to run simultaneously, thus hiding the latency of global memory accesses. In the present case, however, the number of force evaluations that can be performed between global memory transfers is proportional to the square of the number of atoms that fit in registers and/or shared memory. We must ensure, of course, that we simultaneously load only atoms that are likely to be within the cutoff distance of each other. This can be accomplished by presorting the atoms into bins based on their coordinates, with each bin corresponding to a cubic domain of dimensions equal to the cutoff distance. All atoms in nonneighboring bins will be outside the cutoff distance, and thus only atoms in neighboring bins need to be loaded simultaneously. Given a cutoff of 12 Å and an atom density of 0.1 Å^{-3}, each bin will hold an average of 173 atoms. Loading two such bins of atoms would require minimally 11 kB of memory, plus some allowance for bins with larger numbers of atoms and the accumulated forces on each atom.

In our implementation of the nonbonded force calculation, each block has enough threads that a bin of atoms has at most one atom per thread. A bin of atoms for which forces are to be accumulated is read into registers. The atoms of a second, neighboring bin are read and copied to shared memory. Each thread then simultaneously iterates over the atoms in shared memory, calculating the force on the atom in its registers. The code for this inner loop is shown in Figure 7. This process may be repeated by the same block for all neighboring bins before storing the total accumulated forces, or a separate block and output

```
texture<float4> force_table;
__constant__ unsigned int exclusions[];
__shared__ atom jatom[];
atom iatom;       // per-thread atom, stored in registers
float4 iforce;    // per-thread force, stored in registers

for ( int j = 0; j < jatom_count; ++j ) {
  float dx = jatom[j].x - iatom.x;
  float dy = jatom[j].y - iatom.y;
  float dz = jatom[j].z - iatom.z;
  float r2 = dx*dx + dy*dy + dz*dz;
  if ( r2 < cutoff2 ) {
    float4 ft = tex1D(force_table, 1.f/sqrt(r2));
    bool excluded = false;
    int indexdiff = iatom.index - jatom[j].index;
    if ( abs(indexdiff) <= (int) jatom[j].excl_maxdiff ) {
      indexdiff += jatom[j].excl_index;
      excluded = ((exclusions[indexdiff>>5] & (1<<(indexdiff&31))) != 0);
    }
    float f = iatom.half_sigma + jatom[j].half_sigma;  // sigma
    f *= f*f;  // sigma^3
    f *= f;  // sigma^6
    f *= ( f * ft.x + ft.y );  // sigma^12 * fi.x - sigma^6 * fi.y
    f *= iatom.sqrt_epsilon * jatom[j].sqrt_epsilon;
    float qq = iatom.charge * jatom[j].charge;
    if ( excluded ) { f = qq * ft.w; }  // PME correction
    else { f += qq * ft.z; }  // Coulomb
    iforce.x += dx * f;
    iforce.y += dy * f;
    iforce.z += dz * f;
    iforce.w += 1.f;  // interaction count or energy
  }
}
```

Figure 7. CUDA source code for nonbonded force evaluation inner loop.

Table 6. GPU Performance for Nonbonded Force Calculation in Test Harness.

Bin size		Time per 100,000 atoms		
		8 Å (214)[a]	10 Å (419)[a]	12 Å (724)[a]
8 Å	51 atoms	47 ms	X	X
9 Å	72 atoms	64 ms	X	X
10 Å	100 atoms	66 ms	76 ms	X
11 Å	133 atoms	93 ms	107 ms	X
12 Å	172 atoms	91 ms	104 ms	121 ms
13 Å	219 atoms	95 ms	107 ms	123 ms

Kernel execution time was measured on a single NVIDIA GeForce 8800 GTX and normalized to 100,000 atoms. For comparison, NAMD[11] running serially requires about 1.9 s to execute a nonbonded force evaluation for 100,000 atoms with a 12-Å cutoff and 16-Å bins.
[a]Cutoff distance (forces per atom).

array may be assigned to each pair of neighboring bins and the forces summed in a separate kernel. Accuracy for large simulations is enhanced by storing the atomic coordinates relative to the centers of the bins and shifting one set of atoms by the bin-bin offset vector when they are loaded.

Performance

For performance evaluation, we have constructed a test harness to generate input sets with each bin containing the average number of atoms, randomly located within the cube. While calculating forces, the implementation also counts the number of atoms within the cutoff distance for each atom, ensuring that the proper amount of work is being done. Performance in this harness is summarized in Table 6. There is a 30% performance drop for the 8-Å cutoff case at a bin size of 11 Å, which corresponds to 160 threads per block. At 160 threads per block, the 26 registers per thread needed by the nonbonded force kernel consume more than half of the 8192 registers available. Thus, the observed performance drop corresponds to the point beyond which the hardware cannot hide latency by coscheduling multiple blocks. For larger bin sizes, performance does not degrade despite the additional number of distance comparisons, suggesting that the reduced number of blocks compensates for the increased thread and iteration counts.

We have incorporated the current CUDA implementation of nonbonded forces into NAMD,[11] a popular, highly-tuned parallel MD program for simulations of large biomolecular systems. Since the NAMD parallelization scheme is based on distributing pairs of neighboring bins to processors, the above GPU algorithm was readily adapted. The primary challenge was that the number of atoms per bin, over 640 for the standard apoa1 benchmark, exceeded the number of threads allowed per block. This was addressed first by having each thread calculate forces for two atoms, and then by splitting bins along one dimension, a standard NAMD method for scaling to large processor counts. The beta implementation of the CUDA runtime used not only blocks the calling process on the CPU but also consumes CPU time while the GPU runs, making overlap of CPU and GPU computation impractical. (This deficiency is corrected in the current release of CUDA.) The performance of CUDA-accelerated NAMD is shown in Table 7. Observed parallel scaling is limited by overhead, memory contention within the quad-core processor (on two and three cores) and by gigabit ethernet bandwidth (on six cores). While the performance of a single GPU-accelerated workstation does not approach that of the clusters and supercomputers typically used for NAMD simulations, a GPU-accelerated cluster with an InfiniBand or other high-performance network should provide a highly cost-effective platform for even the largest simulations.

In our implementation the force F_{ij} between a pair of atoms is calculated for both atom i and atom j. This redundancy suggests that performance could be doubled by accumulating forces for the atoms in shared memory as well as those in registers. The simplest method of doing this is for each thread to iterate over the atoms in shared memory starting at a different index. For iteration k, thread i accesses atom $(i + k) \bmod n$ from shared memory. Even after substantial tuning (e.g., replacing the expensive general $j \bmod n$ operation with "if $j \geq n$ then $j = j - n$"), we observed that performance with this reciprocal-forces method was lower than for redundant forces. We attribute this to the `_syncthreads()` call required between iterations, the extra instructions required to accumulate the force into shared memory, and reduced warp coherency during the cutoff test.

Another inefficiency in our implementation is that, even with bin size exactly equal to the cutoff distance, only 15.5% of atoms in the 27 bins that must be checked are actually within the cutoff distance. On a traditional CPU, the number of failed cutoff checks is reduced by building a pairlist of all atoms within the cutoff distance plus a small margin and then reusing the list until an atom has moved more than half the margin dis-

Table 7. Performance of CUDA-Accelerated NAMD.

Cores/GPUs	1	2	3	6
GPU-accelerated performance				
Nonbonded (GPU) time	0.20 s	0.11 s	0.078 s	0.047 s
Other (CPU) time	0.16 s	0.11 s	0.08 s	0.06 s
Total time	0.36 s	0.22 s	0.16 s	0.11 s
Unaccelerated performance				
Nonbonded time	1.78 s	0.95 s	0.70 s	0.50 s
Other time	0.16 s	0.11 s	0.08 s	0.06 s
Total time	1.94 s	1.06 s	0.78 s	0.56 s
Speedup from GPU acceleration				
Nonbonded only	8.9 ×	8.6 ×	9.0 ×	10.6 ×
Total	5.4 ×	4.8 ×	4.9 ×	5.1 ×

Runtime per step was measured on a pair of workstations, each with one Intel Core 2 quad-core 2.66 GHz processor and three NVIDIA GeForce 8800 GTX boards. Nonbonded GPU time is the maximum (across all processes), which on six GPUs is 50% higher than the average. Other CPU time is the difference between nonbonded and total time and is assumed to be the same with or without GPU acceleration. The apoal benchmark used has 92,224 atoms with a 12-Å cutoff, 16-Å bins, and PME full-electrostatics evaluated every step. The three-GPU runtime of 0.16 s/step corresponds to 0.54 ns/day given a 1-fs timestep.

tance. Exclusion checking is required only when building the list, providing a further reduction in work during the force evaluation. Building a pairlist for every pair of bins would result in a large range of list sizes and substantial warp divergence as some threads completed their work much earlier than others. Every atom's complete pairlist should include similar numbers of atoms (around 1030 for a 13.5-Å list), reducing warp divergence if each thread iterates over the complete list for a single atom.

The way to efficiently access this number of atoms in random order is to use the texture unit. The required atomic data can be reduced to 16 bytes by storing the charge q_i in 16-bit fixed-point format and the atom type rather than ε_i and σ_i. Accuracy can be preserved without bin-relative coordinates by storing coordinates in 32-bit fixed-point format. The G80 texture unit supports indexed access to arrays of arbitrary size (limited by device memory) and can load 16 bytes (a complete atom) per fetch. Since nearby atoms will have many pairlist elements in common, the same spatial bins described above are used to maximize the impact of the texture cache. The 8-kB texture cache will be more effective for smaller cutoff distances, for it will hold all of the atoms in a single 10-Å pairlist but only half the elements in a 13.5-Å pairlist.

The lists themselves are read from device memory in a regular pattern (one element per thread per iteration) that can be coalesced by the hardware into a single, efficient data transfer per iteration. If the list read is started immediately after the previous element is consumed then read latency can be hidden by the force computation. At 4 bytes per element, the lists will require 4 MB per atom of device memory, limiting simulations to around 150,000 atoms per GPU. In our experiments, force evaluation with a 13.5-Å list was twice the speed of the bin method with a 12-Å cutoff but this did not include the cost of creating the lists on the CPU or transferring the lists across the PCI Express bus (over 100 ms for 100,000 atoms given a maximum data transfer rate of 4 GB/s). We have not yet explored list creation on the G80, but the ability in CUDA for threads to write to multiple, arbitrary locations in device memory makes this option feasible, which it was not in previous GPUs.

This pairlist-based algorithm does not use the shared memory found on the G80 and is therefore similar to previous GPU implementations of molecular dynamics.[9,41,42] Of these, the implementation in ref. 42 updates the pairlist only every few hundred steps, and that in ref. 9 does not update the pairlist during the entire simulation. The pairlists for these simulations must be small enough to store in the on-board memory of the GPU, limiting the simulation to a few hundred thousand atoms. Folding@Home simulations[43] use small proteins and often do not contain explicit water atoms, resulting in a very small number of atoms with wide variation in the number of neighbors per atom. Therefore, the Folding@Home GPU kernel described in ref. 8 does a complete $O(N^2)$ all-to-all force calculation, dropping the pairlist described in ref. 41.

Our spatial bin method does not require the CPU to build pairlists or check exclusions and uses minimal GPU memory and bandwidth. Spatial bins map well to spatial decomposition for GPU-accelerated clusters. Spatial bins also allow us to dedicate the texture unit to force interpolation, supporting complex or expensive functional forms for nonbonded forces.

Other molecular dynamics calculations that could be ported to the G80 include nonbonded energy evaluation (a trivial extension to the force routine), the particle-mesh Ewald reciprocal sum for long-range electrostatics, bonded force and energy calculations, and finally integration of positions and velocities based on the calculated forces. Since these operations are currently handled efficiently by the CPU in an amount of time comparable to the nonbonded force calculation on the GPU, our next priorities are to modify NAMD to run CPU and GPU calculations simultaneously and to dynamically balance both CPU and GPU load.

Sample Applications

We consider two applications of GPU acceleration to biomolecular simulations. We begin with the usage of GPUs to place ions around a solvated system and continue with the calculation of time-averaged Coulomb potentials in the vicinity of a simulated system. These two applications, although conceptually fairly simple, provide representative examples of the types of calculations in biomolecular modeling that may be greatly accelerated through the use of GPUs.

All benchmarks were run on a system containing a 2.6-GHz Intel Core 2 Extreme QX6700 quad core CPU, running 32-bit Red Hat Enterprise Linux version 4 update 4, unless otherwise stated. Single-threaded CPU-based calculations were performed on one core of this system, whereas multithreaded CPU-based calculations used all four cores. GPU-accelerated calculations were performed on the same system as the CPU calculations, using three cores and three NVIDIA GeForce 8800GTX GPUs for direct summation, or two cores and one GPU for multilevel summation calculations.

General Remarks on Ion Placement

Given the ubiquitous presence of electrostatic interactions in biopolymer dynamics, the presence and placement of ions in molecular modeling is of great practical importance. Approaches for selecting the initial ion positions in MD simulations include random placement of ions corresponding to a specified salt concentration,[44] sequential placement of ions in Coulomb energy minima,[45,46] optimization of ion positions via Brownian dynamics,[47] docking of ions with energetic optimization,[48] placement of ions according to the Debye-Hückel distribution with Monte Carlo refinement,[49] and placement in a Poisson-Boltzmann potential with grand canonical Monte Carlo steps.[50] In all of these cases, the primary goals are to provide an ion distribution for simulations that matches the desired physiological conditions, which provides a realistic enough starting state avoiding simulation artifacts caused by the placement, and that can be performed quickly.

Ion placement is particularly important for systems involving nucleic acids, as the presence of divalent counterions is known to be important to proper folding of ribozymes,[51–54] protein-RNA binding,[55] and the activity of enzymes with nucleotide substrates.[56] Recent computational studies have also demonstrated the importance of divalent cations to RNA structural stability.[57] Treatment of cation binding to nucleic acids is difficult

due to the presence of both site-bound (dehydrated) and diffusively bound ions.[58] It has been shown that diffusively bound ions play the primary role in stabilizing the folded structure of the studied RNA molecules and interact with the overall electrostatic environment of RNA.[53] Proper treatment of site-bound ions requires consideration of the removal of the ion's hydration shell and its direct interaction with the binding site.[59]

The currently used methods for ion placement are computationally too expensive for practical use with large systems. Since MD simulations of very large systems are becoming increasingly common,[2,3] the development of efficient ion-placement methods is highly desirable. Thus, we developed the program `cionize`, which iteratively places ions at Coulomb potential minima and takes full advantage of CUDA. `cionize` calculates the Coulomb potential generated by the structure on a lattice, places an ion at the potential global minimum, updates the potential, and then repeats the procedure for the next ion to be placed. The computation is highly parallelizable, since the calculation of the potential at each lattice point is independent from calculations at all the other points; therefore, this computation is well suited for CUDA acceleration. An important aspect of `cionize` is that it can read in an external potential energy map to begin placements, rather than calculating the Coulomb potential. For example, `cionize` can be used to place ions in the potential obtained from a Poisson-Boltzmann solver by writing a potential map from that solver and using it as input for `cionize`; the potential will then undergo Coulomb-based updates as ions are placed.

To compare the accuracy and performance of a representative set of ion placements, four different ion placement protocols were applied to a set of RNA-containing test systems. The ion placement protocols used are detailed below, followed by a description of the specific software implementations.

For Coulomb placement, the Coulomb potential at each point on a lattice with spacing of 0.5 Å was calculated for the volume occupied by the biomolecule and 10 Å of padding around it and ions placed at the available point of lowest potential energy. The lattice was updated with the effects of each new ion after placement. In this and all subsequent placement protocols, placement of ions closer than 5 Å to either the solute or any other ions was not allowed unless otherwise noted. Partial atomic charges were assigned based on the CHARMM27 force field.[60]

In Coulomb placement with a distance-dependent dielectric (DDD), calculations were performed as described above for Coulomb placement, but with the presence of a distance-dependent dielectric coefficient of $\varepsilon(r) = 3r$, with r the distance between interacting points in Å [see eq. (1)].

For iterative Poisson-Boltzmann placement, the electrostatic potential was calculated by solving the linearized Poisson-Boltzmann equation around the protein/RNA system of interest prior to ion placement. Ions were then placed one at a time at the global potential energy minimum, and a new continuum electrostatic calculation was performed after each ion was placed. CHARMM27[60] charges and atomic radii were used for all atoms, and dielectric constants of 78.54 and 4.0 were used for the solvent and biomolecule, respectively. In addition to any explicitly present ions from previous placements, univalent mobile ions were assumed to be present in the system at a concentration of 0.150 M. The potential was calculated on a cubic lattice, including a region 8 Å away from the maximal extents of the solute in each direction. A lattice spacing of no more than 1.0 Å was used.

Hybrid Poisson-Boltzmann/Coulomb placement is a modification of the previous method where full continuum electrostatics calculations were performed prior to the initial placement and again after every five ion placements, as described in the previous paragraph. Between each of these calculations, a set of five ions was placed sequentially at the global energy minimum, with the lattice updated through Coulomb potential calculation after each individual ion was placed. A dielectric constant of 40 was assumed during the potential update for each ion (using dielectric constants of 20 and 80 yielded similar results). All previously placed ions were included in subsequent continuum electrostatics calculations.

All continuum electrostatics calculations were performed using version 0.5 of the APBS software package,[61] compiled with GCC 3.4.6 using the default build flags distributed with the software. Coulomb potential calculations and ion placements were performed using the `cionize` tool described above. For the Poisson-Boltzmann and hybrid protocols, potential lattices generated using APBS were read into `cionize` and used as a basis for ion placement. `cionize` includes single-threaded, multithreaded, and GPU-accelerated implementations of the direct Coulomb summation and multilevel Coulomb summation. For the benchmarks below, a single-threaded version compiled with GCC 3.4.6 was used for all serial CPU-based calculations (unless otherwise noted), a multithreaded, vectorized version compiled with ICC 9.0 was used for multithreaded CPU-based calculations, and the GPU version (compiled with GCC 3.4.6 and NVCC 0.8) was used where GPU acceleration is noted. `cionize` is distributed free of charge with source code within the molecular visualization program VMD.[10] A single-threaded, optimized version compiled with ICC 9.0 was used where noted for CPU-based direct and multilevel summation benchmarks.

Ion Placement for tRNA and Ribozymes

The ion placement protocols described above were applied to three cases where a number of cation-binding sites were resolved in an RNA or protein-RNA crystal structure: Ile-tRNA^{Ile} synthetase complexed with tRNA^{Ile} (PDB code 1FFY[62]), the P4–P6 ribozyme domain (PDB code 1GID[63]), and the hepatitis delta virus ribozyme (PDB code 1CX0[64]). In each case, the protein and nucleic acid portions of the structure were isolated, charges assigned based on the CHARMM27 force field,[60,65] and then ions placed using the methods described previously. In all cases, a number of Mg^{2+} ions equal to the number of cations in the initial crystal structure were placed (in the case of 1FFY the crystal structure contains 10 Mg^{2+} ions, 2 Zn^{2+} ions, and one K^+ ion, but 13 Mg^{2+} ions were placed for the sake of consistency with the other test cases).

Comparisons of the ion placements obtained by different methods are shown in Figure 8. As can be seen, while none of the methods tested precisely matched the locations of crystal ions, qualitatively similar distributions are obtained in all cases. The lack of precise matches indicates that these methods are not

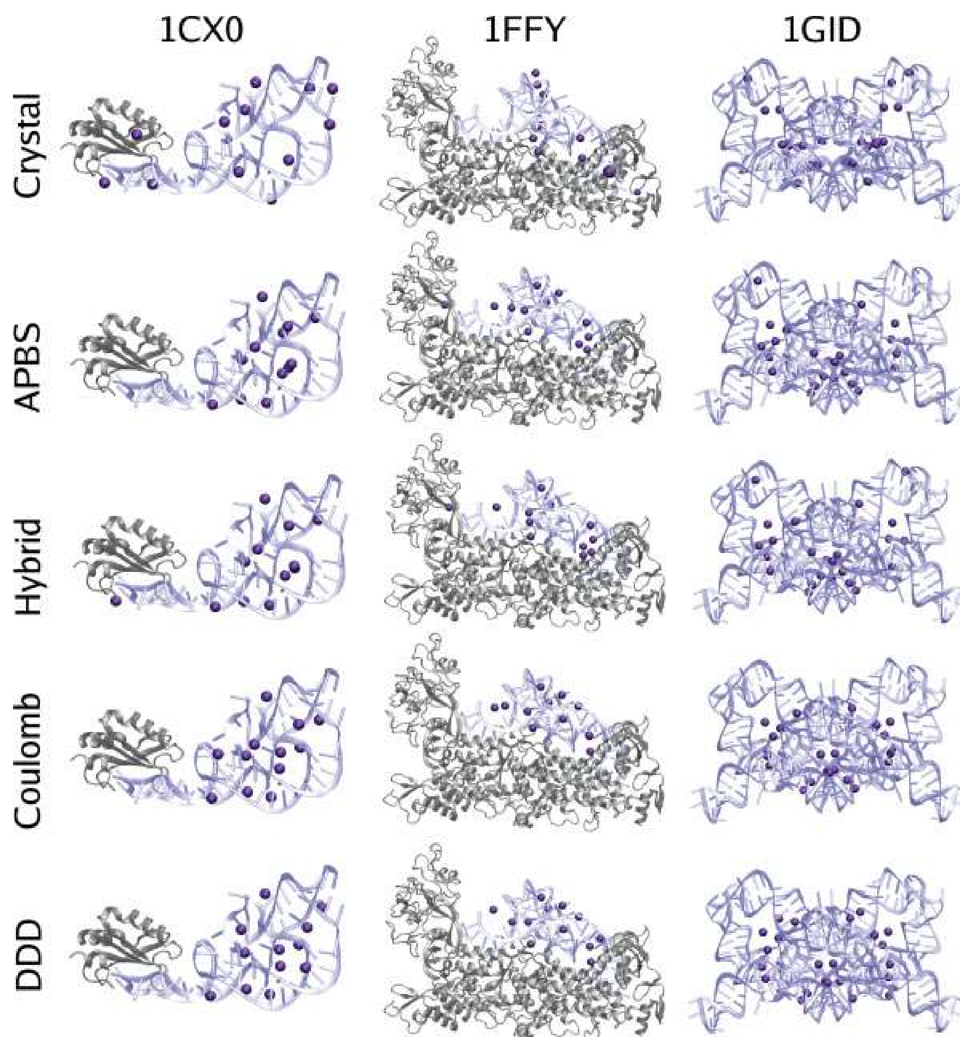

Figure 8. Comparison of ion placement test cases used in this study with crystal ions. Structures are arranged in columns, from left to right, hepatitis delta virus ribozyme (1CX0), tRNA-Ile/synthetase (1FFY), and P4-P6 ribozyme domain (1GID). The crystal structure is shown at the top of each column, followed by placement with the APBS, Hybrid, Coulomb, and DDD methods (see Table 8 for abbreviations). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

suitable for identifying the locations of site-bound ions (where desolvation effects play an important role), but the methods should be sufficient for assigning the locations of diffusively-bound ions, particularly for use in simulations.

Timing information for the different ion placement methods is given in Table 8. Even for the modestly-sized structures used in this study, the Poisson-Boltzmann and Coulomb ion placement methods require significant amounts of time. Unlike the other methods, GPU-accelerated `cionize` calculations showed very little increase in time requirement with system size, as the potential calculation is so fast that it is no longer the rate-limiting step. For all four of these test cases, GPU-accelerated direct summation is faster than GPU-accelerated multilevel summation; the same does not hold for the larger systems used in later sections.

Serial, parallel, multilevel summation, and GPU-accelerated Coulomb placement methods yield identical ion placement results for the four small test cases, although as discussed below, the calculated potential lattice is not identical in the multilevel summation case.

Ion Placement for the Sarcin-Ricin Loop

Hermann and Westhof developed a Brownian-dynamics method for predicting ion-binding sites near RNA and applied this method both to RNA crystal structures with ions resolved and to RNA NMR structures with unknown binding sites.[47] One such structure, that of the sarcin-ricin loop of the eukaryotic 28S rRNA (PDB code 1SCL[66]), was chosen for the application of the ion placement methods used in this study, and the results

Table 8. Ion Placement in Test Structures.

Structure	Atoms	Ions	APBS	Hybrid APBS	Coulomb			ML		DDD GCC
					GCC	ICC[a]	GPU[a]	ICC	GPU[a]	
1CX0	3887	12	359	89	232	5	2	4	3	135
1GID	10166	24	2411	500	1746	39	6	12	10	907
1FFY	17006	13	1457	336	3292	59	7	15	10	1662
1SCL	939	7	186	54	16	1	1	1	1	14

System size, number of ions placed, and timing information for the three crystal structures considered in this study and the sarcin-ricin loop. All times are for complete calculation of ion placements and are given in seconds of wall-clock time. Iterative Poisson-Boltzmann ion placement is abbreviated APBS, hybrid Poisson-Boltzmann/Coulomb placement as Hybrid, direct Coulomb summation ion placement as Coulomb, Coulomb placement using the multilevel summation approximation as ML, and Coulomb-based ion placement with a distance-dependent dielectric as DDD.
[a]A multithreaded calculation; all others were single threaded.

compared with theoretical predictions. While the target data in this case is theoretical and not experimental, it should be noted that the authors of the study[47] predicted the locations of crystal ions in many other structures to 0.3–2.7-Å accuracy.

A comparison of the placements obtained using our protocols with those predicted by Hermann and Westhof is shown in Figure 9. While none of the ion placements used here precisely match the predicted cation-binding sites, ions 1, 2, 5, 6, and 7 appear to be well placed in all cases. Ion 4 is also placed properly in all but the distance-dependent dielectric case, whereas ion 3 is missed in each case in favor of a placement near G10 (see Fig. 9). As in the other test cases, the approaches used in this study do not exactly match the target placements but provide a good approximation using an approach that is both general and quite fast. Timing information for the methods applied to this system is included in Table 8.

Ion Placement for Large Structures

As noted previously, many ion placement protocols become unfeasible when applied to large structures. Here we present two examples where MD simulations were performed on RNA-containing systems larger than one million atoms and compare CPU- and GPU-based Coulomb ion placement for these cases.

STMV Capsid and Genome

Satellite tobacco mosaic virus (STMV) is a small icosahedral plant virus that has previously been studied using all-atom MD.[3] The virus includes a 1063-base single-stranded RNA genome, of which 949 bases are included in the modeled structure. Given the importance of divalent cations in stabilizing the structure of viruses,[67,68] placing some set of counterions near the RNA is an important step in generating a suitable structure of the virus for MD simulations. Initial MD studies of STMV used Mg^{2+} ions randomly placed within 7 Å of the viral RNA[3] but for newer simulations (to be described elsewhere) the Coulomb ion placement strategy was used.

The combined structure of the complete protein and RNA consists of 179,606 atoms; the net charge of the RNA plus the N-terminal tails which partially neutralize it[69] is -348 *e*. To provide a proper ionic environment immediately surrounding the viral RNA, 174 Mg^{2+} ions were placed with `cionize` using the Coulomb ion placement method described previously, but with a lattice spacing of 0.2 Å and minimum ion-solute distance of 2.5 Å. The performance of CPU and GPU-based calculation of the potential lattice required for ion placement is shown in Table 9. Initial calculation of the Coulomb potential lattice was performed on 18 1.5 GHz Intel Itanium 2 processors of an SGI Altix and required 21927.0 s, or 109.6 CPU-hours (note that the Itanium version of `cionize` was compiled with ICC 9.0). An equivalent calculation using three GPUs took 1593.6 s, or 79.68 GPU-minutes (in this case the lattice was calculated as two separate halves because it was too large for the physical memory of the test machine). Calculation of the same potential lattice using multilevel summation required 670.6 s on one CPU or 145.6 s on one GPU; however, the potential lattice differed slightly from the exact solution, as discussed below.

Bacterial Ribosome

Here we present a final example of ion placement that greatly benefited from GPU acceleration. The ribosome is a very complex macromolecular machine responsible for protein synthesis in every cell. The bacterial ribosome has an approximate mass of 2.6–2.8 MDa and a diameter of 200–250 Å.[70] Two-thirds of its mass is composed of RNA, the rest corresponding to around 50 ribosomal proteins. The largest MD simulation reported to date was performed on this system.[2] We developed an all-atom model based on a crystal structure of the *Thermus thermophilus* ribosome at 2.8-Å resolution (PDB codes 2J00/2J01[71]) and performed a 2.8-million-atom MD simulation using NAMD 2.6.[11] Details on the modeling and simulation will be presented elsewhere.

An important step of system preparation was ion placement. This crystal structure has many (741) Mg^{2+} ions resolved. After model building, the system had 260,790 atoms and a net charge of -2616 *e*. Therefore, we placed 1308 diffusively-bound Mg^{2+} ions with `cionize` to neutralize the system, using a lattice spacing of 0.5 Å, padding the system by 10 Å in each direction, and enforcing a minimum ion-ion and ion-solute distance of 5 Å. Atomic partial charges were assigned based on the

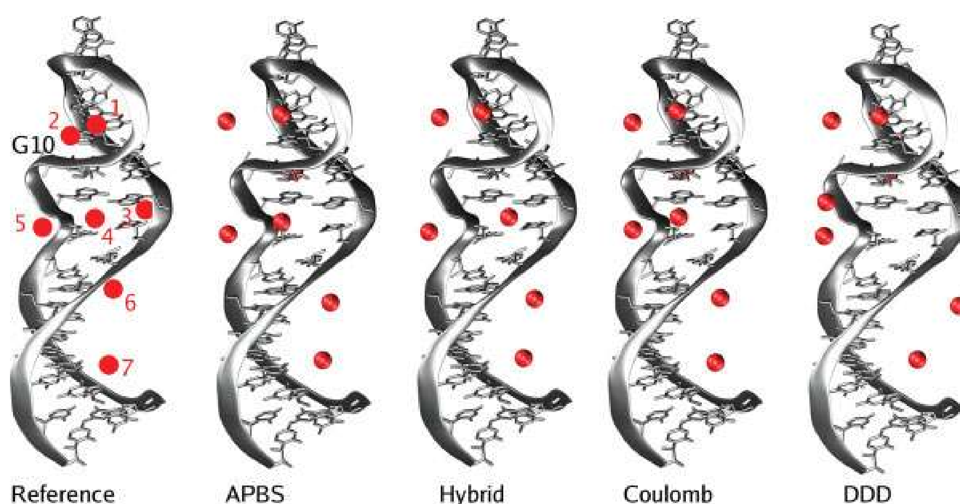

Figure 9. Comparisons of ion placements for the sarcin-ricin loop with the predictions in Fig. 7 of ref. 47. Hermann and Westhof's predictions are shown at left, with arbitrary ion numbering as in their figure; the other methods, from left to right, are APBS, Hybrid, Coulomb, and DDD placement. Abbreviations for ion placement methods are given in the caption of Table 8.

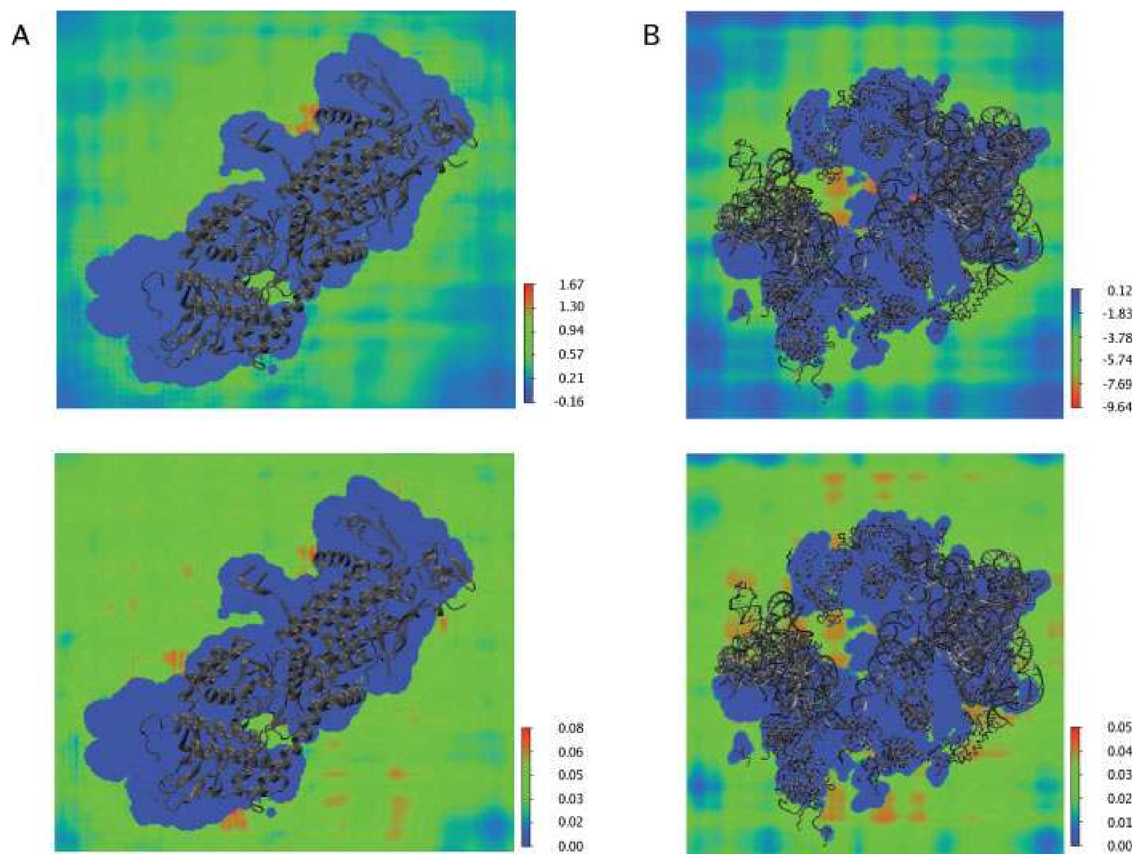

Figure 10. Differences between potentials calculated through direct Coulomb summation and multilevel summation; in each case, the difference is shown above in units of kT/e, and below as a percentage of each particular lattice point. (A) Potential differences for the tRNA-Ile synthetase system (PDB code 1FFY). (B) Potential differences for the ribosome system.

Table 9. Potential Calculation Performance.

		GPU		ML		ML-GPU	
Calculation	ICC	Time	Speedup	Time	Speedup	Time	Speedup
STMV	109.6	1.33	82.5	0.186	588	0.040	2710
Ribosome	41.7	0.38	110	0.043	970	0.018	2085

Comparison of CPU-based and GPU-based calculation of Coulomb potential lattices for different tasks. Time required for each calculation (in CPU-hours or GPU-hours) and the relative speedup of the GPU over the ICC calculation are shown. Abbreviations are defined in Table 8.

CHARMM27 force field.[60] The ion placement was performed using 141 1.5 GHz Intel Itanium 2 processors on an SGI Altix, where the initial lattice calculation took 1065.8 s, or 41.7 CPU-hours (note that the Itanium version of `cionize` was compiled with ICC 9.0). The same calculation took only 456.9 s on three GPUs, or 22.8 GPU-minutes. For comparative purposes, the same initial lattice calculation using multilevel summation required 154.7 s on 1 CPU, but yielded slightly different results, as discussed in the following section. A GPU-accelerated multilevel summation calculation required 64.3 s on 1 GPU, indicating that a structure of this size is significantly past the break-even points for the usage of both GPU acceleration and multilevel summation. These results are summarized in Table 9.

Performance and Accuracy of Multilevel Summation

As seen in Tables 8 and 9, the multilevel summation approximation provides a level of performance on a single CPU similar to that obtained from GPU acceleration of the direct summation algorithm for smaller systems and for large systems (the virus and ribosome) shows better performance than the GPU-accelerated direct summation due to the favorable scaling of the algorithm. GPU-accelerated versions of multilevel summation showed higher performance than the equivalent CPU calculations, but the effects of GPU acceleration are not as significant as in the direct sum cases, as only the short-range part of the calculation currently runs on the GPU.

In the four test cases shown in Table 8, the multilevel summation method yielded identical results to direct summation for ion placement, although small differences were observed in the calculated potential. Figure 10A shows the potential differences along a cross-section of the 1FFY structure. For this structure, the average difference in potential between the multilevel and direct sum calculations was 0.383 $\frac{kT}{e}$, an average difference of 0.037% of the potential from the direct sum, with a maximum difference of 0.086%. The multilevel summation method was also applied to ion placement in the ribosome; in this case, 1,082 ions were placed identically, and of the remainder, 153 of the ions placed by multilevel summation were within 5 Å of a matching ion from the direct summation placement. The deviation between the potential maps in this case is similar to the smaller 1FFY test case: an average deviation of 1.07 $\frac{kT}{e}$, or 0.025% of the direct summation potential, with a maximum difference of 0.053%. A map of the difference in calculated potential between the direct and multilevel summations for the ribosome is shown in Figure 10B. The fact that differing ion placements result from relatively small differences in calculated potential appears to arise from the presence of several closely-spaced local minima at some stages of the placement process. Although the number of placements that are appreciably different between the two methods is quite small, the discrepancies (at least for ion placement) could be further masked by identifying a cluster of low lying potential points at each ion placement step and evaluating the direct Coulomb sum in some neighborhood around each of those points to make the final placement.

Time-Averaged Electrostatic Potential

Ensemble-averaged electrostatic potentials of biomolecules provide a more useful description than potentials calculated from static structures by avoiding overemphasis of certain conformations.[72,73] For example, a potential averaged over relevant timescales is much more realistic for regions where ions bind transiently, compared to a static picture that would either overestimate or underestimate the ionic contribution.[59] Time-averaged electrostatic potentials have been calculated from MD trajectories before, providing good agreement with experimental estimates.[73] A promising application of time-averaged electrostatic potentials is to provide a mean-field description of part of the system that can be incorporated in an MD simulation, reducing the number of degrees of freedom and thus allowing longer timescales to be reached.

As mentioned previously, a time-averaged electrostatic potential can be calculated from an MD trajectory. One possible approach to perform this calculation, available in VMD,[10] is to use the particle-mesh Ewald method[33] to calculate a smoothed electrostatic potential for each frame of a trajectory, and then average them.[73] This approach weakens short-range interactions, which, in principle, could be added to the calculation. But the most important shortcoming is the requirement of periodic boundary conditions, which can introduce artifacts if the center of mass of the system moves considerably throughout the simulation. Even in the absence of large center of mass motion, one may want to calculate the electrostatic potential experienced by a subsystem in the simulation, which requires that every frame is fitted to a reference structure with respect to the subsystem. However, such fitting is not possible if one is to employ periodic boundary conditions. The use of direct Coulomb summation eliminates this shortcoming, but it is prohibitively expensive to calculate time-averaged electrostatic potentials using CPUs; calculation on GPUs makes the use of direct Coulomb summation feasible.

GPU-accelerated direct Coulomb summation was applied to calculate a time-averaged electrostatic potential from a large MD simulation—the aforementioned 2.8-million-atom MD simulation of a bacterial ribosome. All frames were least-square fitted to the initial structure with respect to backbone atoms. For each frame, all atoms within 5 Å of the ribosome were selected (around 580,000 atoms) and the Coulomb potential calculated using partial charges assigned from the CHARMM27 force field.[60] A lattice encompassing the selected structure with a padding of 10 Å was created using a lattice spacing of 1.0 Å. A time-averaged potential from a 1-ns portion of the trajectory,

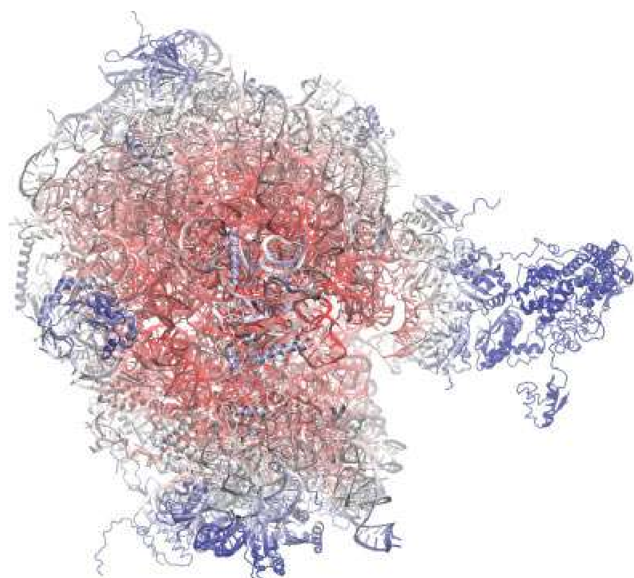

Figure 11. Time-averaged Coulomb potential calculated from 1 ns of an MD simulation of the *T. Thermophilus* ribosome. The potential was calculated every picosecond using three GPUs (see text for details). In the color scale used, red corresponds to potential values smaller than −2500 kT/e, and blue corresponds to values larger than −1800 kT/e, with $T = 298.15$ K. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

with a frame selected every picosecond, was calculated; the results of the calculation are shown in Figure 11.

For one frame, the vectorized potential calculation code compiled with ICC 9.0 within the volumetric data framework of VMD[10] took 18,852 s running on one core, or 5.24 CPU-hours. The same calculation running on one GPU was performed in 529 s, or 0.147 GPU-hours, corresponding to a 36-fold speedup. The time-averaged Coulomb potential calculation for the 1-ns portion of the MD simulation used 3 GPUs and took 1000×178 seconds, or 49 hours. The same calculation would have taken 0.60 years running on 1 CPU or 0.23 years running on three CPUs (see Table 10). The scaling efficiency results in Table 10 illustrate the detrimental effects of multiple CPU cores competing for a fixed amount of cache and memory bandwidth. Since each GPU contains its own independent on-board memory system, bandwidth scales with the number of GPUs and scaling efficiency remains close to 100%. Although the GPUs share the host PCI-Express bus, host bus bandwidth was not a performance-limiting factor for the benchmark system. A multilevel summation calculation (using `cionize`, since this code is not yet implemented in VMD) on a single frame required 74.3 s on one CPU or 67 s (0.0187 GPU-hours) on one GPU and shows an average deviation of 0.025% from the direct summation.

Discussion

We have tested GPU implementations of both Coulomb potential calculations and nonbonded force evaluation for molecular dynamics, both of which are common and time-consuming steps in molecular modeling calculations. While the current generation of GPUs provides an architecture much more suitable for general purpose computing than its predecessors, some room for improvement still exists, as discussed below. The results of our calculations also illustrate the types of calculations in molecular modeling that may be suited for GPU acceleration. Finally, we consider a number of similar applications that may also benefit from this approach.

Current GPU Hardware Restrictions

With CUDA, NVIDIA has enabled the scatter memory-access pattern on GPUs. The utility of this pattern is restricted, however, by the lack of synchronization primitives (such as mutual exclusion locks) between threads in different blocks. In molecular dynamics force calculation, for example, multiple blocks in a grid calculate forces for the same atoms. These forces must currently be written to 27 separate output buffers in device memory that are summed by a separate kernel. This separate invocation could be avoided if each block held a lock while updating the output buffer for a bin of atoms. An alternative to locks would be the implementation of scatter-add operations,[74] allowing `a[i] += b` to occur atomically rather than as separate load, add, and store instructions. Scatter-add to shared memory would be particularly useful, allowing efficient histogram construction without the numerous `_syncthreads()` calls currently needed to detect and resolve conflicts. Either locks or scatter-add to device memory would then allow the partial histograms generated by each block to be combined into a single output buffer, rather than requiring memory for a separate output buffer per block.

Applications to Molecular Modeling

Although the placement of ions in a Coulomb potential is a simple and specific task, our results illustrate the acceleration that is possible for other common molecular modeling tasks through the use of GPUs. GPU-accelerated calculations were typically 10–100 times faster than equivalent CPU-based calculations for ion placement tasks and yielded identical results. The multilevel summation method to calculate the initial Coulomb potential allowed ion placement with high speed, albeit with a slight loss of accuracy; multilevel summation also proved to be amenable to GPU acceleration. Refinement of ion placement through the use of Monte Carlo or Brownian dynamics steps and more sophisticated ion–ion interactions may improve placement accu-

Table 10. Scaling Efficiency of the Coulomb Potential Calculation Code Implemented in VMD.

	Number of CPUs/GPUs (efficiency)			
	1	2	3	4
ICC	18852 s (100 %)	9619 s (98 %)	7099 s (88 %)	6178 s (76 %)
GPU	529 s (100 %)	265 s (100 %)	178 s (99 %)	n/a

Time required for the calculation of the Coulomb potential from one frame of the ribosome MD trajectory (see text for details) and scaling efficiency are presented.

racy,[47,50] and should also be suitable for GPU acceleration.[75] The Coulomb potential calculation used for ion placement can also be applied to the calculation of average electrostatic potentials from MD trajectories. GPU acceleration of this calculation provided a hundredfold speedup over CPU implementations for the examples presented.

Nonbonded force evaluation for molecular dynamics, with cutoffs, exclusion checking, and long-range PME force correction, has been implemented efficiently on the G80 GPU without the use of CPU-generated pairlists. As future generations of GPUs outpace CPU performance growth, the overall performance of the simulation will be limited by the fraction of computation remaining on the CPU and, for parallel runs, by the speed of internode communication.

Aside from the applications to electrostatic potentials and nonbonded force evaluation for MD introduced here, a wide variety of other tasks in molecular modeling are well suited to GPU acceleration. The efficiency and ease of such implementations depends largely on the level of data parallelism present in the target task; GPU acceleration is best suited to algorithms where the same calculation may be carried out independently on many separate inputs. Tasks where some calculation is performed independently at every point on a lattice are a clear example; the Coulomb potential calculations presented above fall into this category, along with, for example, scoring grid calculation for docking[76,77] and calculation of volumetric accessibility maps.[78] In addition, GPU acceleration should be applicable to tasks such as molecular dynamics trajectory analysis, where statistics such as pairwise root mean square deviations and alignments between trajectory frames, contact maps on individual frames, and volumetric time-averaged occupancy maps can be calculated parallelizing by frame or by pairs of frames. Some simulation methodologies may also be well suited to GPU acceleration, including ligand docking, where multiple conformations may be generated and evaluated in parallel, and biomolecular Monte Carlo simulations, where both trial step generation and scoring for acceptance may be performed in parallel. Bioinformatics queries on broad databases are also likely candidates for GPU acceleration, and indeed, an initial GPU implementation of the Hidden Markov Model-based *hmmsearch* algorithm has already been developed.[7] A wide variety of computationally intensive tasks in biomolecular modeling and analysis are similarly suited to GPU acceleration, and adoption of this technology should enable previously unfeasible calculations in a variety of related areas on commodity GPU hardware.

While GPU acceleration provides a method for accelerating many common and time-consuming calculations that are required in molecular modeling, it must be stressed that GPUs should be viewed as an additional resource that may be useful for some types of calculations, and not as a universally applicable solution. As noted above, GPU acceleration is best suited to data-parallel tasks. Obtaining peak performance from the GPU typically requires a different optimization strategy than for CPU-based algorithms. The current generation of GPU hardware is limited to single-precision operations, which excludes some applications in molecular modeling; GPUs that include native double-precision calculations are expected by the end of 2007, but even with this additional capability single-precision calculations will likely be faster and should be favored when sufficient. While it is attractive to consider clusters of nodes performing GPU-accelerated calculations, the speed of floating-point operations on GPUs is so high that bottlenecks in internode communication will become even more significant for such a cluster, likely limiting the total number of nodes that can be usefully applied to a single task (although the high per-node speed will still allow small GPU-accelerated clusters to outperform much larger traditional clusters on suitable applications). Within these limitations, GPU acceleration promises to provide a powerful new technique for accelerating many molecular modeling applications.

Conclusion

We have presented a review of recent advances in GPU hardware and software development and demonstrated the feasibility of accelerating molecular modeling applications with GPUs. Three target algorithms along with details on their implementation using NVIDIA's CUDA software architecture were discussed, namely direct Coulomb summation, multilevel Coulomb summation, and molecular dynamics nonbonded force calculation. The use of GPU-accelerated electrostatic potential calculation was illustrated by two concrete cases: ion placement for MD simulation and average electrostatic potential calculation from an MD trajectory. These examples provide a flavor of the impact that the use of GPUs may have on molecular modeling applications in the near future. Widespread usage of GPU acceleration may eventually make it possible to perform many tasks that currently require specialized computational resources using comparatively inexpensive desktop computers running commodity GPUs.

Acknowledgments

The authors thank Professors Wen-mei Hwu and David Kirk for organizing the CUDA programming class and explicitly inviting the direct participation of many research groups at the University of Illinois at Urbana-Champaign. The authors also acknowledge the direct support provided by NVIDIA, in particular, David Kirk and Mark Harris for their consultation and advice during the development of our GPU algorithms. Professor Nathan A. Baker provided valuable input and suggestions for the implementation of ion placement techniques. Professor Robert D. Skeel has imparted to the authors much insight concerning the multilevel summation method. Computer time at the National Center for Supercomputing Applications was provided through LRAC grant MCA93S028.

References

1. McCammon, J. A.; Gelin, B. R.; Karplus, M. Nature 1977, 267, 585.
2. Sanbonmatsu, K. Y.; Joseph, S.; Tung, C. S. Proc Natl Acad Sci USA 2005, 102, 15854.
3. Freddolino, P. L.; Arkhipov, A. S.; Larson, S. B.; McPherson, A.; Schulten, K. Structure 2006, 14, 437.
4. Yeh, I.; Hummer, G. J Am Chem Soc 2002, 124, 6563.
5. Duan, Y.; Kollman, P. Science 1998, 282, 740.
6. Adcock, S. A.; McCammon, J. A. Chem Rev 2006, 106, 1589.

7. Horn, D. R.; Houston, M.; Hanrahan, P. In Proceedings of the 2005 ACM/IEEE Conference on Supercomputing. Washington, DC, 2005; IEEE Computer Society; p. 11.
8. Elsen, E.; Houston, M.; Vishal, V.; Darve, E.; Hanrahan, P.; Pande, V. In SC06 Proceedings; IEEE Computer Society, 2006.
9. Yang, J.; Wang, Y.; Chen, Y. J Comput Phys 2007, 221, 799.
10. Humphrey, W.; Dalke, A.; Schulten, K. J Mol Graphics 1996, 14, 33.
11. Phillips, J. C.; Braun, R.; Wang, W.; Gumbart, J.; Tajkhorshid, E.; Villa, E.; Chipot, C.; Skeel, R. D.; Kale, L.; Schulten, K. J Comp Chem 2005, 26, 1781.
12. Beetem, J.; Denneau, M.; Weingarten, D. SIGARCH Comput Archit News 1985, 13(3), 108.
13. Blank, T. In Compcon Spring '90. 'Intellectual Leverage'. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference. San Francisco, CA, 1990; IEEE Computer Society; pp. 20–24.
14. Tucker, L. W.; Robertson, G. G. Computer 1988, 21, 26.
15. Ungerer, T.; Robic, B.; Silc, J. ACM Comput Surv 2003, 35, 29.
16. Allen, J. D.; Schimmel, D. E. IEEE Trans Parallel Distrib Syst 1996, 7, 818.
17. Weems, C. In Proceedings of the 1997 Computer Architectures for Machine Perception (CAMP, '97), Washington, DC, 1997; IEEE Computer Society; p. 235.
18. NVIDIA CUDA Compute Unified Device Architecture Programming Guide; NVIDIA: Santa Clara, CA, 2007.
19. He, Y.; Ding, C. H. Q. In ICS '00: Proceedings of the 14th International Conference on Supercomputing; ACM Press: New York, 2000; pp. 225–234.
20. Bailey, D. H. Computing in Science and Engineering 2005, 07, 54.
21. Kirk, D.; Hwu, W. University of Illinois at Urbana-champaign, ECE 498 AL Lecture Notes, 2007.
22. Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M.; Hanrahan, P. In SIGGRAPH '04: ACM SIGGRAPH 2004 Papers; ACM Press: New York, 2004; pp. 777–786.
23. Buck, I. Stream Computing on Graphics Hardware, PhD Thesis, Stanford University, Stanford, CA, 2005.
24. Charalambous, M.; Trancoso, P.; Stamatakis, A. In Panhellenic Conference on Informatics, 2005, pages 415–425.
25. Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krger, J.; Lefohn, A. E.; Purcell, T. J. Computer Graphics Forum 2007, 26, 80.
26. Barnes, J.; Hut, P. Nature 1986, 324, 446.
27. Greengard, L.; Rokhlin, V. J Comp Phys 1987, 73, 325.
28. Board, J. A., Jr.; Causey, J. W.; Leathrum, J. F., Jr.; Windemuth, A.; Schulten, K. Chem Phys Lett 1992, 198, 89.
29. Cheng, H.; Greengard, L.; Rokhlin, V. J Chem Phys 1999, 155, 468.
30. Hockney, R. W.; Eastwood, J. W. Computer Simulation Using Particles; McGraw-Hill: New York, 1981.
31. Pollock, E. L.; Glosli, J. Comput Phys Commun 1996, 95, 93.
32. Darden, T.; York, D.; Pedersen, L. J Chem Phys 1993, 98, 10089.
33. Essmann, U.; Perera, L.; Berkowitz, M. L.; Darden, T.; Lee, H.; Pedersen, L. G. J Chem Phys 1995, 103, 8577.
34. Skeel, R. D.; Tezcan, I.; Hardy, D. J. J Comp Chem 2002, 23, 673.
35. Hardy, D. J. Multilevel Summation for the Fast Evaluation of Forces for the Simulation of Biomolecules, PhD Thesis, University of Illinois at Urbana-Champaign, 2006.
36. Brandt, A.; Lubrecht, A. A. J Comput Phys 1990, 90, 348.
37. Sandak, B. J Comp Chem 2001, 22, 717.
38. Brandt, A. In Proc. IMACS 1st Int. Conf. on Comp. Phys. Boulder, CO, 1991.
39. Skeel, R. D.; Biesiadecki, J. J. Ann Numer Math 1994, 1, 191.
40. Fox, G. C.; Johnson, M. A.; Lyzenga, G. A.; Otto, S. W.; Salmon, J. K.; Walker, D. W. Solving Problems on Concurrent Processors, vol. 1; Prentice Hall: Englewood Cliffs, NJ, 1988.
41. Buck, I. In IEEE Visualization 2004 GPGPU Tutorial. IEEE Computer Society, 2004.
42. Kupka, S. In Central European Seminar on Computer Graphics 2006, 2006.
43. Snow, C. D.; Nguyen, H.; Pande, V. S.; Gruebele, M. Nature 2002, 420, 102.
44. Sotomayor, M.; Schulten, K. Biophys J 2004, 87, 3050.
45. Auffinger, P.; Westhof, E. J Mol Biol 1999, 269, 326.
46. Walser, R.; Hünenberger, P. H.; van Gunsteren, W. F. Proteins: Struct Func Gen 2001, 43, 509.
47. Hermann, T.; Westhof, E. Structure 1998, 6, 1303.
48. Soares, C. M.; Teixeira, V. H.; Baptista, A. M. Biophys J 2003, 84, 1628.
49. Grubmüller, H. SOLVATE 1.0 manual, 1996.
50. Vitalis, A.; Baker, N. A.; McCammon, J. Mol Sim 2004, 30, 45.
51. Wang, N.; Butler, J. P.; Ingber, D. E. Science 1993, 260, 1124.
52. Tinoco, I., Jr.; Kieft, J. S. Nat Struct Biol 1997, 4, 509.
53. Misra, V. K.; Draper, D. E. Biopolymers 1999, 48, 113.
54. Klein, D. J.; Moore, P. M.; Steitz, T. A. RNA 2004, 10, 1366.
55. Cusack, S. Curr Opin Struct Biol 1999, 9, 66.
56. Vianna, A. L. Biochim Biophys Acta 1975, 410, 389.
57. Réblová, K.; Špačková, N.; Koča, J.; Leontis, N. B.; Šponer, J. Biophys J 2004, 87, 3397.
58. Misra, V. K.; Draper, D. E. J Mol Biol 2000, 299, 813.
59. Misra, V. K.; Draper, D. E. Proc Natl Acad Sci USA 2001, 98, 12456.
60. Foloppe, N.; MacKerrell, A. D., Jr.; J Comp Chem 2000, 21, 86.
61. Baker, N. A.; Sept, D.; Joseph, S.; Holst, M. J.; McCammon, J. A. Proc Natl Acad Sci USA 2001, 98, 10037.
62. Silvian, L. F.; Wang, J.; Seitz, T. A. Science 1999, 285, 1074.
63. Cate, J. H.; Gooding, A. R.; Podell, E.; Zhou, K.; Golden, B. L.; Kundrot, C. E.; Cech, T. R.; Doudna, J. A. Science 1996, 273, 1678.
64. Ferré-D'Amaré, A. R.; Zhou, K.; Doudna, J. A. Nature 1998, 395, 567.
65. MacKerell, Jr., A. D.; Brooks, B.; Brooks, III, C. L.; Nilsson, L.; Roux, B.; Won, Y.; Karplus, M. In The Encyclopedia of Computational Chemistry, Schleyer, P. et al., Eds.; John Wiley: Chichester, 1998; pp. 271–277.
66. Fitzpatrick, P. A.; Steinmetz, A. C. U.; Ringe, D.; Klibanov, A. M. Proc Natl Acad Sci USA 1993, 90, 8653.
67. Lucas, R. W.; Larson, S. B.; McPherson, A. J Mol Biol 2002, 317, 95.
68. Eecen, H. G.; Dierendonck, J. H. V.; Pleij, C. W. A.; Mandel, M.; Bosch, L. Biochemistry 1985, 24, 3610.
69. Larson, S. B.; McPherson, A. Curr Opin Struct Biol 2001, 11, 59.
70. Blaha, G. In Protein Synthesis and Ribosome Structure; Nierhaus, K. H.; Wilson, D. H., Eds.; Wiley-VCH: Weinheim, Germany, 2004; pp. 53–84.
71. Selmer, M.; Dunham, C. M.; Murphy, F. V., IV; Weixlbaumer, A.; Petry, S.; Kelley, A. C.; Weir, J. R.; Ramakrishnan, V. Science 2006, 313, 1935.
72. You, T. J.; Bashford, D. Biophys J 1995, 69, 1721.
73. Aksimentiev, A.; Schulten, K. Biophys J 2005, 88, 3745.
74. Ahn, J. H.; Erez, M.; Dally, W. In HPCA-11. 11th International Symposium on High-Performance Computer Architecture, IEEE Computer Society, 2005. pp. 132–142.
75. Tomov, S.; McGuigan, M.; Bennett, R.; Smith, G.; Spiletic, J. Comp and Graph 2005, 29, 71.
76. Morris, G. M.; Goodsell, D. S.; Halliday, R. S.; Huey, R.; Hart, W. E.; Belew, R. K.; Olson, A. J. J Comp Chem 1998, 19, 1639.
77. Zou, X.; Sun, Y.; Kuntz, I. D. J Am Chem Soc 1999, 121, 8033.
78. Cohen, J.; Kim, K.; King, P.; Seibert, M.; Schulten, K. Structure 2005, 13, 1321.