Parallel Computing 35 (2009) 164–177

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Multilevel summation of electrostatic potentials using graphics processing units ☆

David J. Hardy [a], John E. Stone [a], Klaus Schulten [a,b,*]

[a] Beckman Institute, University of Illinois at Urbana-Champaign, 405 N. Mathews Ave., Urbana, IL 61801, USA
[b] Department of Physics, University of Illinois at Urbana-Champaign, 1110 W. Green, Urbana, IL 61801, USA

ARTICLE INFO

Article history:
Received 16 June 2008
Received in revised form 2 November 2008
Accepted 16 December 2008
Available online 25 December 2008

Keywords:
Multilevel summation
Electrostatics
Molecular dynamics
Molecular modeling
GPU computing

ABSTRACT

Physical and engineering practicalities involved in microprocessor design have resulted in flat performance growth for traditional single-core microprocessors. The urgent need for continuing increases in the performance of scientific applications requires the use of many-core processors and accelerators such as graphics processing units (GPUs). This paper discusses GPU acceleration of the multilevel summation method for computing electrostatic potentials and forces for a system of charged atoms, which is a problem of paramount importance in biomolecular modeling applications. We present and test a new GPU algorithm for the long-range part of the potentials that computes a cutoff pair potential between lattice points, essentially convolving a fixed 3D lattice of "weights" over all sub-cubes of a much larger lattice. The implementation exploits the different memory subsystems provided on the GPU to stream optimally sized data sets through the multiprocessors. We demonstrate for the full multilevel summation calculation speedups of up to 26 using a single GPU and 46 using multiple GPUs, enabling the computation of a high-resolution map of the electrostatic potential for a system of 1.5 million atoms in under 12 s.

1. Introduction

The electrostatic fields surrounding biomolecules govern many aspects of their dynamics and interactions. Three-dimensional "maps" of these electrostatic fields are often used in the process of constructing and visualizing molecular models, in analyzing their structure and function, and as a means of accelerating molecular dynamics simulations of very large systems. The computational requirements for evaluating electrostatic fields using the most efficient methods scale linearly with problem size. It is becoming increasingly common for biomedical researchers to construct and simulate biomolecular systems containing over 1 million atoms, and recently, a 100-million-atom molecular dynamics simulation was specified as a model problem and acceptance test for the upcoming NSF "Blue Waters" petascale supercomputer. At these problem scales, even linear time algorithms become a computational constraint for molecular modeling applications, and the use of novel accelerator devices and hardware architectures becomes attractive.

As power and heat dissipation constraints have prevented microprocessor clock rates from increasing substantially in the last several years, the high performance computing community has begun to look to accelerator technologies as a means of achieving continued performance growth for compute-bound applications. Graphics processing units (GPUs) have recently

☆ Supported by the National Institutes of Health under Grant P41-RR05969 and the University of Illinois through a grant from the Institute for Advanced Computing Applications and Technologies.

* Corresponding author. Address: Beckman Institute, University of Illinois at Urbana-Champaign, 405 N. Mathews Ave., Urbana, IL 61801, USA. Tel.: +1 217 244 1604; fax: +1 217 244 6078.

E-mail addresses: dhardy@ks.uiuc.edu (D.J. Hardy), johns@ks.uiuc.edu (J.E. Stone), kschulte@ks.uiuc.edu (K. Schulten).

0167-8191/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2008.12.005

emerged as an exciting new acceleration option and have gained particular interest due to high performance, low cost, widespread availability, and the existence of the high-performance graphics market which helps sustain continued research and development investments in GPU technology.

The high performance of GPUs is primarily attributable to their design as throughput-oriented, massively parallel, highly multithreaded processor arrays. GPUs are designed as parallel processors out of the necessity to perform rasterization and complex shading arithmetic on large geometric models resulting in multimillion pixel images, typically rendered 30–120 times per second. State-of-the-art GPUs consist of hundreds of arithmetic processing units, achieving peak floating point arithmetic performance levels approaching 1 trillion floating point operations per second.

Several groups have recently reported GPU-accelerated algorithms that achieve 10- to 100-fold speedups compared to current generation CPUs, providing commensurate application performance increases up to the limits dictated by Amdahl's law. Several of the early successes with GPU acceleration have been algorithms and applications related to molecular modeling [1–5].

In this paper we review the use of the multilevel summation method (MSM) for calculating electrostatic potentials and compare the performance of an optimized CPU implementation of the method with a newly-developed GPU-accelerated implementation, building upon our previous work.

2. Background

Molecular modeling applications require the computation of potential energies and forces between interacting atoms. The empirical potential functions employed for a classical mechanics treatment of the physical behavior of biomolecules introduce bonded terms, describing spring and angle oscillations of the atoms in the molecular structure, and nonbonded terms, describing the gradual attraction and hardcore repulsion between pairs of atoms (van der Waals interactions) and the pairwise interaction due to the assignment of fixed partial charges to the atoms (electrostatic interactions). In the advent of systems biology, we have the means to use the computer as a computational microscope that permits investigation of atomic level descriptions of increasing size and complexity, with a detailed view of the mechanical behavior of molecules working together. As we progress to larger systems, the computation of electrostatics is of paramount importance, providing the longer range forces that drive the overall conformational changes.

2.1. Electrostatics and multilevel summation

The multilevel summation method (MSM) is a fast algorithm for computing electrostatics [6,7], employing hierarchical interpolation of softened pair potentials to calculate an approximation to the $O(N^2)$ pairwise interactions with just $O(N)$ computational work. This approach offers comparable accuracy and performance to the more commonly used particle-mesh Ewald (PME) [8,9] calculation for atomic systems with periodic boundary conditions. However, MSM is more flexible than PME since MSM is also applicable to non-periodic boundaries. MSM also permits improved parallel scalability over PME, with a communication structure for a spatial domain decomposition involving nearest neighbor communication with a reduction operation followed by a broadcast. PME, on the other hand, requires performing two many-to-many communication stages to compute two 3D FFTs over the reciprocal space lattice points. Furthermore, MSM offers improved multiscale resolution due to calculation of more slowly varying potentials on lattices of coarser spacing, making MSM better suited to multiple time stepping than PME [10]. For non-periodic boundaries, the fast multipole method (FMM) is better known and more widely used [11,12]. The spherical harmonics used for FMM converge faster than the known interpolation schemes for MSM which allows very high accuracy from FMM to be accomplished with less cost. However, FMM suffers from computing discontinuous potentials and forces which are bad for dynamics [13], as they result in heating that gradually destabilizes a simulation. If used for dynamics, FMM requires high enough accuracy that it turns out to be much less efficient than MSM [6].

The use of hierarchical interpolation of softened pairwise potentials was initially introduced for solving integral equations [14] and later applied to long-range charge interactions [15]. We use the splitting and interpolation strategies introduced by Skeel et al. [6], which compute continuous forces for dynamics. Our application here is to compute electrostatic potential maps,

$$V_i = \sum_j \frac{q_j}{4\pi\epsilon_0 |\mathbf{r}_j - \mathbf{r}_i|}, \tag{1}$$

where each electrostatic potential V_i is computed at lattice point position $\mathbf{r}_i$, with the sum taken over the atomic coordinates $\mathbf{r}_j$ that have associated fixed partial charge q_j. The factor ϵ_0 is the dielectric constant.

We have previously introduced [1] the use of MSM for computing electrostatic potentials but provide here a brief summary of the mathematical description for completeness. MSM approximates the sums in Eq. (1) by first exactly splitting the $1/r$ factor into a series that contains a leading short-range part g_* followed by ℓ smoothed potentials,

$$\frac{1}{|\mathbf{r}' - \mathbf{r}|} = g_*(\mathbf{r}, \mathbf{r}') + \sum_{k=0,\ldots,\ell-2} g_k(\mathbf{r}, \mathbf{r}') + g_{\ell-1}(\mathbf{r}, \mathbf{r}'), \tag{2}$$

where each successive potential is more slowly varying. We do this by defining the potentials in terms of an inter-atom cutoff distance a and an unparameterized smoothing γ of the function $1/\rho$,

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

We choose γ using the second order Taylor expansion of $s^{-1/2}$ about $s = 1$,

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

so that each of the potentials $g_*, g_0, \ldots, g_{\ell-2}$ smoothly vanish for pairwise distances beyond cutoffs $a, 2a, \ldots, 2^{\ell-1}a$, respectively.

We also define interpolation operators,

$$\mathcal{I}_k g(\mathbf{r},\mathbf{r}') = \sum_\mu \sum_\nu \phi^k_\mu(\mathbf{r}) g(\mathbf{r}^k_\mu, \mathbf{r}^k_\nu) \phi^k_\nu(\mathbf{r}'), \qquad k = 0,1,\ldots,\ell-1, \tag{5}$$

where the $\mathbf{r}^k_\mu$ positions are defined on a uniform lattice of spacing $2^k h$ with corresponding nodal basis functions ϕ^k_μ defined by

$$\phi^k_\mu(\mathbf{r}) = \Phi\left(\frac{x - x^k_\mu}{2^k h}\right)\Phi\left(\frac{y - y^k_\mu}{2^k h}\right)\Phi\left(\frac{z - z^k_\mu}{2^k h}\right), \tag{6}$$

in terms of a finest lattice spacing h and a dimensionless basis function Φ of unit spacing and having local support. We choose Φ to be the linear blending of quadratic interpolating polynomials,

$$\Phi(\xi) = \begin{cases} (1 - |\xi|)\left(1 + |\xi| - \frac{3}{2}\xi^2\right), & \text{for} \quad |\xi| \leqslant 1, \\ -\frac{1}{2}(|\xi| - 1)(2 - |\xi|)^2, & \text{for} \quad 1 \leqslant |\xi| \leqslant 2, \\ 0, & \text{otherwise}, \end{cases} \tag{7}$$

providing a continuously differentiable approximation. The nested interpolation of Eq. (2),

$$\begin{aligned} \tfrac{1}{|\mathbf{r}'-\mathbf{r}|} &= (g_* + g_0 + g_1 + \ldots + g_{\ell-2} + g_{\ell-1})(\mathbf{r},\mathbf{r}') \\ &\approx (g_* + \mathcal{I}_0(g_0 + \mathcal{I}_1(g_1 + \cdots \mathcal{I}_{\ell-2}(g_{\ell-2} + \mathcal{I}_{\ell-1} g_{\ell-1})\cdots)))(\mathbf{r},\mathbf{r}'), \end{aligned} \tag{8}$$

yields a multilevel approximation on ℓ progressively coarser lattices containing just $O(N)$ terms, where the amount of work at each successive lattice level is reduced by about a factor of 1/8. A detailed theoretical analysis of Eq. (8) has been done [7] to determine constants for an asymptotic error bound of the form

$$\text{potential energy error} < O\left(\frac{h^p}{a^{p+1}}\right), \tag{9}$$

where p is the degree of the interpolant (in this case $p = 3$). The experiments in this paper use cutoff $a = 12$ Å and finest lattice spacing $h = 2$ Å, parameter choices demonstrated to calculate the electrostatic potentials to about 2.5 digits of accuracy.

2.2. MSM algorithm and parallelization

Replacing the $1/r$-term in Eq. (1) with the multilevel approximation from Eq. (8) produces the following algorithm [1]. The electrostatic potential is computed as the sum of scaled short- and long-range parts,

$$V_i \approx \frac{1}{4\pi\epsilon_0}\left(e^{\text{short}}_i + e_i^{\text{long}}\right), \tag{10}$$

with an exact short-range part computed using a cutoff pair potential,

$$e^{\text{short}}_i = \sum_j g_*(\mathbf{r}_i, \mathbf{r}_j) q_j, \tag{11}$$

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, compute cutoff potentials on each lattice level, sum together contributions from previous levels, and finally interpolate the long-range contributions from the finest level lattice. The ordered steps of the long-range algorithm can be defined as follows:

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

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

$$\text{lattice cutoff: } e_\mu^{k,\text{cutoff}} = \sum_\nu g_k(\mathbf{r}_\mu^k, \mathbf{r}_\nu^k) q_\nu^k, \quad k = 0, 1, \ldots, \ell - 2, \tag{14}$$

$$\text{top level: } e_\mu^{\ell-1} = \sum_\nu g_{\ell-1}(\mathbf{r}_\mu^{\ell-1}, \mathbf{r}_\nu^{\ell-1}) q_\nu^{\ell-1}, \tag{15}$$

$$\text{prolongation: } e_\mu^k = e_\mu^{k,\text{cutoff}} + \sum_\nu \phi_\nu^{k+1}(\mathbf{r}_\mu^k) e_\nu^{k+1}, \quad k = \ell - 2, \ldots, 1, 0, \tag{16}$$

$$\text{interpolation: } e_i^{\text{long}} = \sum_\mu \phi_\mu^0(\mathbf{r}_i) e_\mu^0. \tag{17}$$

We note that the sums shown for anterpolation, restriction, prolongation, and interpolation are quite small due to the local support of the nodal basis functions, with the number of terms for each on the order of $(p+1)^3$, for pth degree polynomial interpolation. The lattice cutoff summations contain a larger number of terms, on the order of $(4/3)\pi(2a/h)^3$, i.e., the volume of the sphere of radius $(2a/h)$ measured in lattice cells. The top level (for non-periodic boundaries) is an all pairs computation, but has been reduced to a constant number of lattice points.

Fig. 1 depicts some of the concurrency available to the MSM algorithm: the short-range part can be computed concurrently with the long-range part, and, moreover, a similar splitting of the problem is available between the restriction and lattice cutoff at each level, with the two parts joined together in the corresponding prolongation part. There is also a sequence of dependent steps, e.g., anterpolation must precede lattice cutoff on level 0, restriction to level 1 must precede lattice cutoff on level 1, interpolation must be proceeded by prolongation to level 0, and so on, with the critical path requiring $O(\log N)$ steps. A scalable distributed memory parallelization strategy would employ spatial decomposition of the domain, as done by the molecular dynamics program NAMD [16,17]. The summations in Eqs. (11)–(17) involve nearby atoms and lattice points, so the parallel communication pattern resembles a reduction operation followed by an accumulating broadcast operation. A typical shared-memory multithreading approach is able to take advantage of the fine degree of parallelism available at every step. Achieving good parallelization through the application of a massively multithreaded architecture offered by

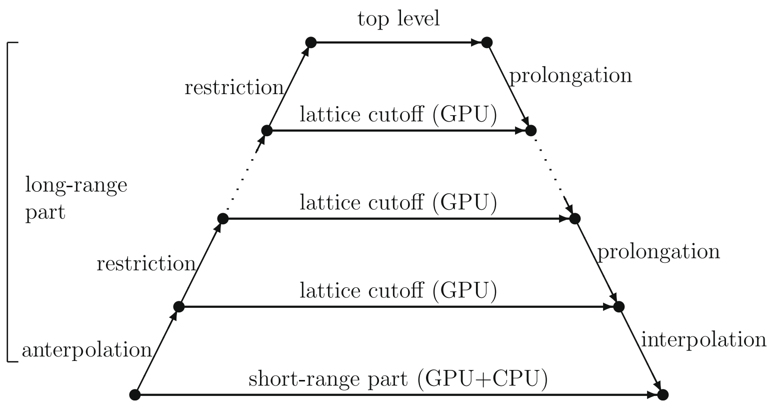

Fig. 1. Algorithmic steps for MSM. The short-range part is computed together with the GPU and CPU. The lattice cutoff parts are computed collectively by the GPU. The remaining parts are computed by the CPU.

Table 1
CPU execution time profile of sequential MSM used to calculate electrostatic potential for 1,534,539 atom system. The long-range calculation is decomposed into its constituent parts.

	Time in seconds	Percentage of total
Short-range part	480.07	89.98
Anterpolation	0.18	0.03
Restriction, levels $0, 1, \ldots, 7$	0.16	0.03
Lattice cutoff, level 0	43.14	8.09
Lattice cutoff, level 1	5.55	1.04
Lattice cutoff, level 2	0.68	0.13
Lattice cutoff, levels $3, 4, \ldots, 8$	0.10	0.02
Prolongation, levels $7, 6, \ldots, 1$	0.17	0.03
Interpolation	3.47	0.65
Long-range part	53.45	10.02

GPUs is much more challenging. The successful acceleration of MSM on the GPU requires that the calculation be formulated as a data-parallel problem, such that thousands of threads can concurrently execute the same instruction streams on independent data.

Table 1 shows the sequential performance profile for the MSM calculation of the electrostatic potential of a 1,534,539 atom system to a map resolution of 0.5 Å. The percentages of total runtime justify our initial parallelization of just the short-range part [1,5]. However, without also treating the lattice cutoff computation that collectively requires over 10% of the runtime, we would by Amdahl's Law be restricted to a maximum speedup of less than 10. Our present work shows speedups of up to 36 on the lattice cutoff computation, with speedups of up to 26 for the full MSM calculation.

2.3. GPU hardware background

Commodity graphics hardware has evolved tremendously over the past two decades, incorporating more computational capability and programmability in order to address the growing demands of multimedia, gaming, engineering, and scientific visualization applications. Early graphics processors were essentially fixed-function devices that implemented a dedicated graphics rasterization pipeline. Modern GPUs are highly programmable devices that execute software in much the same way a CPU does, with each generation increasingly relying on large arrays of fully programmable processing units in place of the fixed-function hardware used in previous generation devices.

In the past two years, significant advances have been made in adapting GPU hardware architecture to better support general purpose computation in addition to the existing graphics workloads. The increasing requirement for software programmability for geometric processing and other operations beyond shading led modern GPUs to transition away from using special-purpose numeric formats toward standard machine representations for integers and floating point numbers, along with single- and double-precision IEEE-754 floating point arithmetic capabilities comparable to that provided by CPU instruction set extensions such as SSE and AltiVec.

State-of-the-art GPUs contain hundreds of processing units, and are capable of performing single-precision floating point arithmetic at rates approaching one trillion floating point operations per second (TFLOPS), with on-board memory systems that provide bandwidths of over 140 GB/s. Unlike CPUs, which have historically been designed to optimize the performance of applications with largely sequential code paths, GPUs are designed as throughput-oriented processing units that use sophisticated hardware to schedule the execution of tens of thousands of threads concurrently on a large pool of processing units. CPUs are designed with large cache memories to reduce the latency for frequently accessed memory locations in main memory. Caches consume a large portion of the die area of CPU designs, area that could be used for additional arithmetic processing units. GPUs take a very different approach and employ hardware multithreading to hide memory access latency by context switching to another runnable thread when the active thread encounters a dependency on a pending global memory operation. With a sufficient number of threads multiplexed onto each of the GPU processing units, latency is effectively hidden, and the device achieves high performance without the need for a cache. GPUs use the die area that would have been consumed for cache memory for additional arithmetic units, yielding a throughput-oriented architecture with extremely high peak arithmetic performance. GPUs further optimize their use of die area with variations of single-instruction multiple-data (SIMD) control logic feeding an identical instruction stream to a group of processing units, thereby reducing the die area and power consumed by instruction decoding logic.

2.4. Architecture of the NVIDIA GT200 GPU

The GT200 GPU is the newest generation of NVIDIA's Tesla GPU architecture [18], incorporated in the GeForce GTX 2x0 series graphics cards and the compute-specific Tesla C1060 and S1070 products. The GT200 GPU architecture is similar in many respects to its predecessors G80 and G92, which were the first with support for NVIDIA's Compute Unified Device Architecture (CUDA) for GPU computing [19,20]. As with the previous generation designs, GT200 is composed of a large pool of programmable processing units that are clustered into groups that share resources. The top level of this hierarchy, referred to as the *streaming processor array*, is composed of up to ten identical *texture processor clusters*, each containing three *streaming multiprocessors*, and with each of these consisting of a group of eight *streaming processors*, for a grand total of up to 240 streaming processors. The streaming multiprocessors within a texture processor cluster execute completely independently of one another, though they share some resources. The streaming multiprocessors of the GT200 and its predecessors implement a single-instruction multiple-thread (SIMT) instruction unit to achieve efficient use of die area and power while hiding global memory latency [18]. The instruction unit in the streaming multiprocessor drives the eight scalar streaming processors with a single instruction stream that is 32 threads wide, known as a *warp*. When executing data-dependent branch instructions, the hardware serializes execution of both sides of the branch until all of the threads within the warp have converged. Branch divergence only penalizes the performance of warps in which it occurs.

Fig. 2 illustrates the groupings of GT200 processing units and the class of functions they perform. The replicated nature of this type of hardware design often makes it possible to recover partially functional GPU chips by disabling non-functional hardware units, allowing such devices to be sold as part of a lower cost product with a minor reduction in performance while effectively increasing the overall GPU semiconductor fabrication yield.

A key architectural trait of modern GPUs is the use of multiple high-bandwidth memory systems to keep the large array of processing units supplied with data. The GT200 architecture supports a high bandwidth (140 GB/s), high latency main or

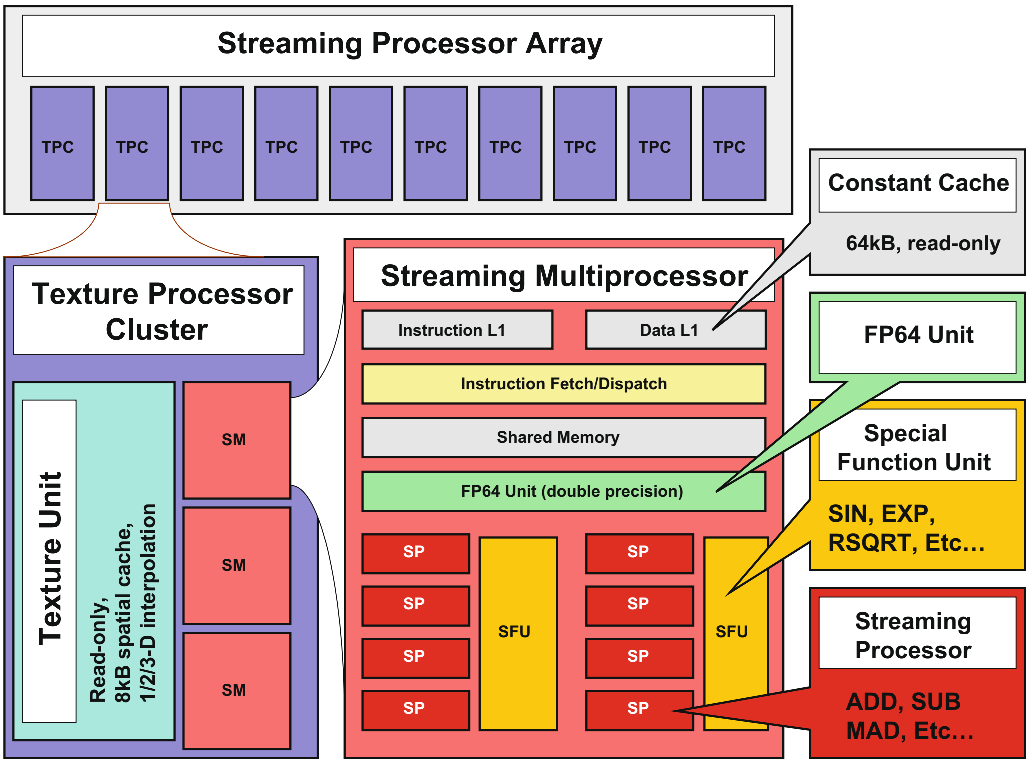

Fig. 2. NVIDIA GT200 GPU block diagram.

“global” memory. The main memory system is supplemented with dedicated hardware texturing units which provide read-only caching with hardware support for multidimensional spatial locality of reference, multiple filtering and interpolation modes, texture coordinate wrapping and clamping, and support for multiple texture data formats. A 64-kB constant cache provides an efficient means of broadcasting identical read-only data elements to all threads within a streaming multiprocessor at register speed. The constant memory can be an effective tool in achieving high performance for algorithms requiring all threads to loop over identical read-only data. As with its predecessors, the GT200 architecture incorporates a 16-kB shared memory area in each streaming multiprocessor. Threads running on the same multiprocessor can cooperatively load and manipulate blocks of data into this fast register-speed shared memory, avoiding costly accesses to the larger global memory. Accesses to the shared memory area are coordinated through the use of a thread barrier synchronization primitive, guaranteeing all threads have completed their shared memory updates before other threads begin accessing results. The shared memory is often effectively utilized as a software-managed cache, reducing bandwidth utilization and latency that repetitive loads and stores to the global memory system would otherwise incur.

Some of the key differences between the GT200 architecture and its predecessors include native support for double-precision floating point arithmetic, a 1.88$\times$ increase in the number of streaming processors (240), a doubling in the register file size for each streaming multiprocessor, improved hardware for coalescing global memory accesses, support for atomic operations to shared memory, “warp vote” boolean reduction operations for threads within the same warp, support for directed rounding modes for double-precision floating point operations, and bidirectional overlapping of asynchronous I/O and GPU kernel execution. The present work benefits primarily from the aggregate increase in the number of streaming processors, the doubled register file size, and improved asynchronous I/O and kernel execution.

2.5. GPU programming models

GPU software development tools have closely tracked advances in hardware architecture and the general evolution of GPU technology toward increased programmability and acceleration of general purpose computation. Early compute-specific GPU programming tools such as BrookGPU [21] and Sh [22] were layered on top of graphics-specific programming interfaces and provided a stream programming abstraction, laying the foundation for early GPU acceleration successes in computational biology [23–25]. With hardware level support for standard data types, arithmetic operations, and flow control instructions, the remaining barriers to the development of GPU-targeted compilers for dialects of popular programming languages have been removed. Recently, commercially supported compilers and runtime systems for the development of GPU-accelerated computational applications have become available, eliminating the necessity for application developers to work within the constraints imposed by graphics-specific programming languages and interfaces. At the present time, CUDA [19], RapidMind [26], and Brook+ [27] are already available to developers, and OpenCL [28] has recently been announced.

The work described in this paper is based on the CUDA GPU programming toolkit developed by NVIDIA [19]. CUDA is a dialect of the C and C++ programming languages with extended syntax supporting multiple memory address spaces and GPU kernel launch, while eliminating language features, such as recursion, that do not map well to the many-core GPU hardware. A full overview of the CUDA programming model is beyond the scope of this paper, but it is worth mentioning a few of its key characteristics to improve the clarity of subsequent discussions. An excellent overview of the CUDA programming model is provided in [20]. Discussions of the CUDA programming model as it relates to molecular modeling applications are provided in [2,1,3,5].

The CUDA programming model decomposes work into a *grid* of *thread blocks* that are concurrently executed by a pool of SIMT *multiprocessors*. Each thread block normally contains 64–512 threads, which are executed by the processing units within a single multiprocessor. Each SIMT multiprocessor executes a group of threads, known as a *warp*, in lockstep. CUDA kernels are written as serial code sequences, with barrier synchronizations to enforce ordering of operations among peer threads within a thread block. The expansion of a kernel into a grid of thread blocks is determined by the kernel launch parameters specified at runtime, and may be varied dynamically according to problem size or other attributes. Data-dependent branch divergence is handled in the hardware by serializing execution of both sides of the branch until all of the processing units have converged on the same instruction. The virtualization of processing resources provided by the CUDA programming model allows applications written with existing GPUs to scale up with future hardware designs.

The key issues that must be considered when designing CUDA kernels involve the decomposition of work into tiles that can be mapped to thread blocks, further decomposed into warps and individual threads. The selection of work decomposition strategy is a particularly important decision for CUDA, as current GPU hardware favors specific access patterns for each of the memory systems which must be explicitly taken into consideration in order to achieve peak performance. The process of designing kernels to meet these requirements can be somewhat complex but is comparable to the difficulty of writing high performance algorithms for multi-core processors with instruction set extensions. The data alignment requirements that must be met for high performance global memory operations in CUDA are no more onerous than one would encounter in developing software to take advantage of the Intel x86 SSE SIMD instruction set extensions, for example.

The CUDA programming interface allows *streams* of kernels and I/O operations to be queued and processed asynchronously from the host CPU. This facility enables applications to completely overlap GPU operations with operations on the host CPU, allowing both resources to be utilized concurrently with high efficiency. An additional event notification system provides a mechanism for the host CPU to monitor ongoing progress of asynchronously executing GPU kernels and I/O operations through occasional polling. Beyond its use for overlapping operations on the host and GPU, the asynchronous interface can also be used to overlap GPU I/O operations for one stream with GPU kernel computations in a different stream, on devices with appropriate hardware support. By submitting independent work into multiple stream queues, an application can take maximum advantage of the host-GPU I/O bandwidth and computational capabilities of the GPU at the same time.

Although it was designed for programming GPUs, many of the abstractions provided by CUDA are well suited to multi-core CPUs. The tiled work decomposition and explicit requirement for high data locality that programmers satisfy when creating a high performance CUDA kernel are often an excellent starting point for a cache-based multi-core processor as well. Stratton et al. [29] recently demonstrated a compiler and runtime framework for retargeting CUDA kernels to multi-core CPUs, achieving performance within 30% of hand-coded multi-core versions of the same kernels.

3. Algorithmic details

We have developed CUDA kernels for accelerating the most demanding parts of the MSM calculation of electrostatic potential maps: the short-range part given by Eq. (11), which for finely spaced maps dominates the entire computation, and the lattice cutoff part given by Eq. (14), which is the lattice-based analog of the short-range calculation that dominates the long-range part of MSM. Recall the MSM sequential time profile in Table 1 and the algorithm decomposition diagrammed in Fig. 1 showing the computational elements of MSM along with the workload division between GPU and CPU.

3.1. Parallelizing the short-range part

Efficient algorithms for computing short-range atomic interactions generally perform geometric hashing of atoms to bins so that interactions need only be considered between atoms in nearby bins [30]. This approach guarantees for molecular modeling applications an algorithm that scales linearly with the number of atoms since the density of atoms is bounded.

We have investigated two approaches for GPU acceleration of the MSM short-range part, both using geometric hashing. The earlier approach [1] features the CPU performing a geometric hashing of atoms into large bins. Multiple GPU kernel calls are made to calculate large cubic blocks of the potential lattice, with the atom data read from the constant memory cache. Bin sizes of twice the cutoff distance are necessary to provide the GPU with enough computation. However, experiments on a system of water exhibited only a 6% success rate for interactions to be within the cutoff. The wasted pairwise distance evaluations combined with the repeated number of small (<64 kB) memory transfers from CPU to GPU limits the large-bin approach to a speedup of 11 for the problem sizes studied in this paper. A substantial improvement is obtained by using small bins, where the neighborhood of surrounding atom bins are streamed from the GPU global memory into shared memory. Our initial investigation using small bins tackled just the isolated problem of computing a cutoff pair potential on a lattice and presented details of a CUDA implementation [5]. We summarize below the main algorithmic ideas and show in Section 4 a speedup of 32 using the small-bin approach for computing the MSM short-range part given by Eq. (11).

For the small-bin approach, the CPU performs geometric hashing of atoms into bins which are then copied to the GPU global memory. The thread blocks are assigned sub-cubes of lattice potential points to calculate, and the thread blocks loop over the surrounding bins of atoms, cooperatively loading each into shared memory for processing. The threads determine from their block index which atom bins overlap their region and then determine the nearby bins by reading from a table of offsets stored in constant memory. The surrounding bins are streamed through shared memory, effectively using shared memory as a self-managed cache. Tests for this approach show an improved 33% success rate for interactions to be within the cutoff distance. After the atom data has been initially loaded into the device memory, the kernel executions read all memory over the high speed bus in the device. Coalesced global memory reads for the atom bins are accomplished by choosing a fixed depth of 8 atoms, stored in $x/y/z/q$ form, with each thread in a warp collectively reading one of the 32 floats.

The CPU regularizes the problem in the small-bin approach for the GPU by padding the surrounding domain with empty bins to avoid exceptional boundary cases and also by processing the contributions of any extra atoms remaining from completely full bins. This provides the CPU with useful work to overlap with the asynchronous GPU kernel. For most efficient computation, we can keep the CPU and GPU almost equally occupied by controlling the spatial size of the bins, where the optimal bin size would give the CPU enough extra atoms so that it takes almost as much time as the GPU to complete. We find that setting the bin size to 4 Å is near-optimal for our test cases.

3.2. Parallelizing the lattice cutoff part

The lattice cutoff part computes a cutoff pair potential between points on a lattice, where the potential at a given lattice point is the distance-weighted sum of the surrounding charges, as illustrated by Fig. 3. The computation involves the evaluation, between point charges on a lattice of spacing $2^k h$, of the smoothed potential function g_k having cutoff distance $2^{k+1}a$, as defined in Eq. (3) for $k = 0, 1, \ldots, \ell - 2$. At each level k, the spheres of charge contributions enclosed by the cutoff distance contains the same number of lattice points. Defining $\varepsilon = \lceil 2a/h \rceil - 1$ to be the radius (measured in lattice points) of these spheres, the summation of the lattice point potentials in Eq. (14) can be expressed as

$$e^k[v_x, v_y, v_z] = \frac{1}{2^k} \sum_{\delta_x=-\varepsilon}^{\varepsilon} \sum_{\delta_y=-\varepsilon}^{\varepsilon} \sum_{\delta_z=-\varepsilon}^{\varepsilon} w[\delta_x, \delta_y, \delta_z] q^k[v_x + \delta_x, v_y + \delta_y, v_z + \delta_z], \tag{18}$$

where we use a three-dimensional indexing of the charge and potential lattices. The cube of weights bounding the sphere is computed a priori,

$$w[\delta_x, \delta_y, \delta_z] = g_0(\mathbf{0}, \mathbf{r}^0[\delta_x, \delta_y, \delta_z]) = 2^k g_k(\mathbf{0}, \mathbf{r}^k[\delta_x, \delta_y, \delta_z]) = 2^k g_k(\mathbf{r}^k[v_x, v_y, v_z], \mathbf{r}^k[v_x + \delta_x, v_y + \delta_y, v_z + \delta_z]), \tag{19}$$

since the g_k are defined in terms of the unparameterized smoothing function γ from Eq. (4) and since the relative distances are the same between points on a uniform lattice.

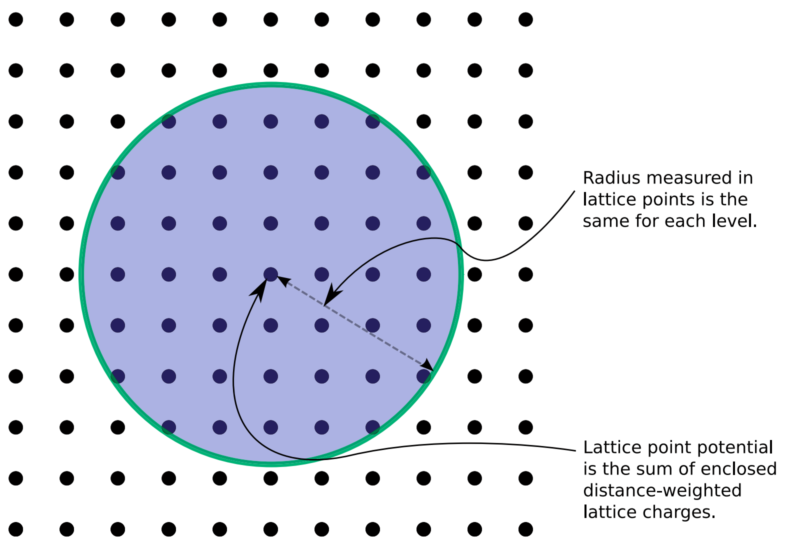
Radius measured in lattice points is the same for each level.
Lattice point potential is the sum of enclosed distance-weighted lattice charges.

Fig. 3. Schematic of the lattice cutoff calculation, showing the regularity of calculating potentials on each lattice level.

The strategy for the GPU parallelization of Eq. (18) is similar to the small-bin approach, with each thread block assigned to calculate potentials for a sub-cube of lattice points. We choose a sub-cube size of $4^3 = 64$ and assign one thread per lattice point, which is the minimum possible thread block size. The implementation described here successfully uses the GPU shared memory area as a kernel-managed cache for streaming the surrounding sub-cubes of charge and the constant memory cache as fast access storage for the cube of weights.

The size of the constant memory cache is constrained to just under 64 kB available to user kernels. Practical application of MSM for molecular modeling uses a cutoff distance a of up to 12 Å with a lattice spacing h no smaller than 2 Å. The upper bound on the ratio $a/h = 6$ gives 23^3 weights which, when padded to 24^3 to be a multiple of the sub-cube size, is still small enough to fit the entire cube into constant memory.

The constant memory cache is read at register speed when all of the threads in the warp collectively read the same value, which is accomplished if every thread in the thread block simultaneously applies the same weight. We achieve this by using the "sliding window" diagrammed in Fig. 4. The working set of charges read into shared memory is a cubic region of 2^3 sub-cubes, containing 8^3 charges. The window is of size 4^3 charges, corresponding to one sub-cube. Within a triply-nested loop, the window is shifted four steps in the x-, y-, and z-dimensions (for a total of 64 shifts), and at each shift every thread applies the single weight value to its respectively located charge to accumulate to its potential.

The sliding window loops are embedded within another triply-nested loop that iterates over the sub-cube neighborhood. We define the "radius" of the neighborhood to be $\varepsilon_s = \lceil \varepsilon/4 \rceil \leqslant 3$ (assuming the bound $a/h \leqslant 6$), so that the neighborhood consists of $(2\varepsilon_s + 1)^3$ sub-cubes. We avoid wasted computation by clipping the edges of this neighborhood so as to be contained within the domain of sub-cubes. Correctness of the sliding window technique at the edges of the domain of sub-cubes requires that we pad the entire domain with a single layer of sub-cubes containing zero charge.

The streaming of the sub-cubes of charge through shared memory is made optimal by performing coalesced reads from global memory. We achieve memory coalescing by transposing the lattice points within each sub-cube. Instead of storing each lattice level in row-column-depth ordering, as would be most natural for a sequential implementation, we store each sub-cube contiguously in memory in its x–y–z ordering, and arrange the block of sub-cubes in its row-column-depth ordering. Doing so produces memory-aligned reads when the (64) threads in a thread block collectively read a sub-cube.

The amount of work available to the lattice cutoff part decreases by almost a factor of 1/8 for each successive level, so we run out of sufficient work for the GPU if we try to compute each level with a separate kernel call. Instead, we note from Eq. (18) that the only distinction between the levels is the scaling factor $1/2^k$ applied after the summation is calculated. This observation means that, not only can the work depicted in Fig. 1 by the horizontal arrows all be done in parallel, but we can expressly schedule all of the intermediate levels (the lattice cutoff parts) together on the GPU. The top level calculation depicted in Fig. 1 uses a different set of weights than those used for the lattice cutoff calculation, so cannot be as conveniently scheduled on the GPU. Instead, the CPU can calculate this part concurrently with the GPU. Although the optimal balance of work between CPU and GPU might involve assigning just the first two or three lattice levels to the GPU and the rest to the CPU, it is much better to under-load the CPU than overload it. The upper levels account for a vanishingly small amount of work on the GPU so will have little impact on the speedup, whereas overloading the CPU and forcing the GPU to wait will severely diminish the speedup.

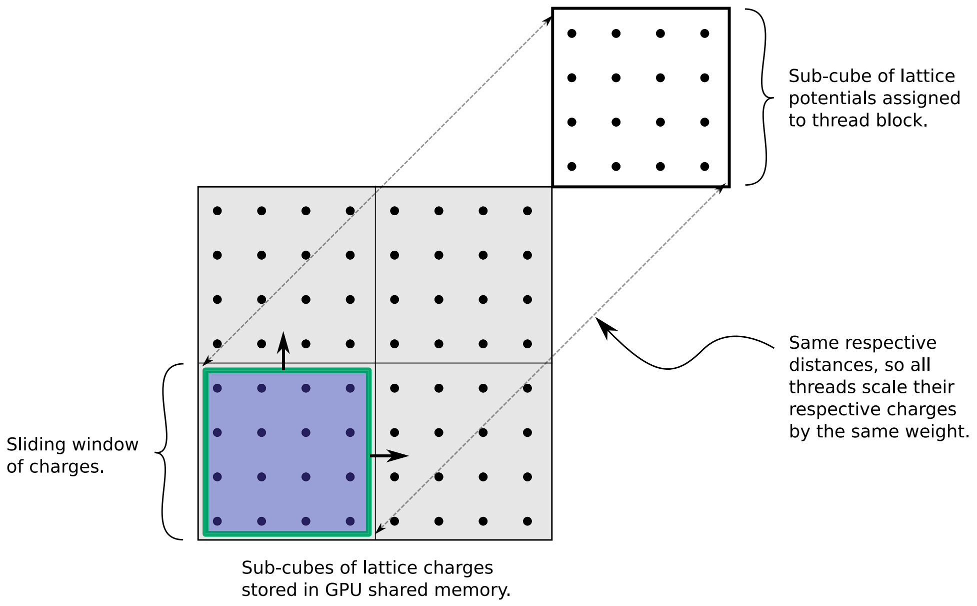
Sub-cube of lattice potentials assigned to thread block.
Same respective distances, so all threads scale their respective charges by the same weight.
Sliding window of charges.
Sub-cubes of lattice charges stored in GPU shared memory.

Fig. 4. Schematic of the sliding window technique that produces optimal reading of distance-based weights from GPU constant memory.

The CPU first computes the anterpolation and restrictions to attain the charges at all lattice levels. The charge lattices are padded to a multiple of the sub-cube size plus a surrounding layer of zero-charge sub-cubes, with the lattices transposed so that the charges in each sub-cube are contiguous in memory. These charge sub-cubes for all intermediate levels are arranged consecutively in a single 1D array that is transferred to the GPU main memory. The lattice cutoff kernel is then invoked on the GPU to execute a 1D grid of thread blocks that correspond to the inner sub-cubes of charge (i.e., not the padded sub-cubes). Each thread first determines which lattice potential within the hierarchy that it is responsible for calculating by using its thread block ID and thread index within the block, along with a small "map" stored in the GPU constant memory indicating the sub-cube dimensions of each lattice level. Then the threads in each thread block cooperatively iterate over the sub-cube neighborhoods as described above, using the sliding window approach to optimally read the weights stored in constant memory. The CPU concurrently performs the top level calculation. After the GPU kernel completes, the CPU transfers a similarly structured 1D array of sub-cubes of potential back from the GPU main memory and then reverses the earlier transformations to obtain the potential lattices for each level. The CPU then computes the prolongation and interpolation parts to finish the MSM long-range calculation.

The basic version of the lattice cutoff algorithm just described is shown in Section 4 below to produce good speedup to the CPU implementation over a wide range of problem sizes. We discovered some optimizations to the basic algorithm that produce better speedup. The innermost loop that moves the sliding window (in our implementation, four shifts in the x-direction) can be unrolled. Also note that half of the working set of charge (four of eight sub-cubes) is used again as we iterate over the sub-cube neighborhood in the innermost loop. The basic algorithm implements the loading of charge sub-cubes from global memory in a triply-nested loop of $2 \times 2 \times 2$ iterations. If within the inner loop over the sub-cube neighborhood we copy the second half of the shared memory ($4 \times 8 \times 8$ charges) to the first half, and combine this with unrolling the resulting loops that move the four sub-cubes within shared memory and copy four new sub-cubes from global memory, we exhibit additional performance improvement. We observed that just reducing the global memory reads without also unrolling the loops that perform the reading did not improve performance, but that using both optimizations together does. When the unrolling optimization is used by itself, the deleterious effects of register pressure can hamper the performance of a kernel by limiting the SIMT hardware's ability to hide global memory latency. By halving the number of global memory accesses through the use of the shared memory copy, we have reduced the kernel's dependence on global memory latency hiding, and the pair of optimizations applied together give a substantial performance boost. Both the basic and optimized kernels use a lot of registers (40 for the basic and 58 for the optimized), but this high register count appears to be balanced by using the smallest number of threads possible (64) per thread block.

Imposing the bound $a/h \leqslant 6$ is necessary for being able to fit the entire padded cube of weights into constant memory. It is also possible to implement versions of the kernel that can handle larger bounds on a/h by storing as little as 1/8th of the cube of weights and exploiting the symmetry of this cube, since these weights are a function of the lattice spacings in the three dimensions. However, exploiting symmetry will complicate the index calculations for reading the weight values, which we would expect to be more costly and would likely consume more registers.

3.3. *Extension to periodic boundary conditions*

The multilevel summation method is applied to periodic boundaries by wrapping around the edges of the lattice. The GPU algorithms discussed here can be extended to periodic systems. For the short-range algorithm, in particular the small-bin approach, the bin sizes in x-, y-, and z-directions can be determined as integer divisions of the respective periodic cell lengths. Instead of padding the edges with empty atom bins, the edges are padded with periodic image atoms. For the lattice cutoff algorithms, instead of padding with one layer of empty sub-cubes, the lattice levels must all be padded with ε_s layers of image lattice points (where ε_s is the radius of the sub-cube neighborhood). Note that depending on the relative periodic cell lengths and lattice spacings in the three dimensions, the radius values ε_s and ε discussed above might actually differ in the three dimensions (the semi-principal axes of an ellipsoid), but these modifications are straightforward. Also note that these implementations can also be extended to handle non-orthogonal periodic cells; the necessary modifications to MSM have been previously discussed [7] and are essentially already handled by the CUDA kernels here extended to orthogonal periodic cells.

4. Experimental evaluation

The performance of the multilevel summation method for computing electrostatic potentials was evaluated for a range of problem sizes. Each of the performance tests shown in Tables 2–4 were conducted on a 1.5×10^7 Å^3 water box containing 1,534,539 atoms. Potential maps were computed with a 0.5 Å lattice spacing, using MSM cutoff distance $a = 12$ Å and finest lattice spacing $h = 2$ Å. The water box was created using the solvate plugin included with VMD [31], with a volume and atom density representative of the biomolecular complexes studied by large scale molecular dynamics simulations [32].

The specification of a 100-million-atom molecular dynamics simulation as a model problem for the NSF Blue Waters Track 1 petascale supercomputer provides a strong motivation for the development of software tools capable of operating in this regime. The 1.5 million atom test case is small enough to run in one pass on a single GPU, and large enough to yield accurate timings and to provide performance predictions for much larger problems.

Table 2
Comparison of performance for the short-range cutoff small-bin kernel tested with a 1.5×10^7 Å^3 water box containing 1,534,539 atoms.

Kernel	Runtime (s)	Speedup
CPU Intel QX6700	480.07	1.00
CUDA C870 (G80)	20.02	23.98
CUDA GTX 280 (GT200)	14.86	32.30

Table 3
Comparison of performance for long-range lattice cutoff kernels tested with a 1.5×10^7 Å^3 water box containing 1,534,539 atoms.

Kernel	Runtime (s)	Speedup
CPU Intel QX6700	49.47	1.0
CUDA C870 (G80) Latcut-basic	2.96	16.7
CUDA C870 (G80) Latcut-optimized	2.19	22.5
CUDA GTX 280 (GT200) Latcut-basic	1.45	34.1
CUDA GTX 280 (GT200) Latcut-optimized	1.36	36.4

Table 4
Comparison of performance for the full multilevel summation calculation tested with a 1.5×10^7 Å^3 water box containing 1,534,539 atoms.

Test platform	Runtime (s)	Speedup
CPU Intel QX6700 CPU	533.67	1.0
CUDA C870 (G80)	26.52	20.1
CUDA D870 ($2 \times$ G80) task-parallel	20.81	25.6
CUDA GTX 280 (GT200)	20.50	26.0
CUDA GeForce 8800 GTX (G80) and D870 ($2 \times$ G80) task- and data-parallel	11.53	46.3

The experiments were run on a quiescent test platform with no windowing system running, using a 2.6 GHz Intel Core 2 Extreme QX6700 quad core CPU running 64-bit Red Hat Enterprise Linux version 4 update 5. (The final test platform in Table 4 features the same software running on a 2.4 GHz Intel Core 2 Q6600 quad core CPU.) The CPU code was compiled using the Intel C/C++ Compiler (ICC) version 9.0. Experiments were performed using the NVIDIA CUDA programming toolkit version 1.1, except for the data-parallel experiment which used version 2.0. The G80-series GPU benchmarks were performed on a deskside NVIDIA Tesla D870 GPU using either one or both internally contained Tesla C870 GPUs. The GT200-series GPU benchmarks were performed using an NVIDIA GeForce GTX 280 GPU.

The performance results listed in Table 2 compare the performance of the the short-range cutoff portion of MSM on the CPU and GPU. The benchmarked implementation is an improved variant of the small-bin short-range cutoff kernel [5] summarized in Section 3.1. The version presented here has been modified to avoid filling the CUDA asynchronous stream queue with too many outstanding requests, which could otherwise cause blocking behavior when computing large potential maps. The short-range cutoff computation consumes the vast majority of the runtime for problems with fine lattice spacing, approximately 90% of the runtime for the CPU version of the code, and approximately 75% of the runtime for the GPU version code running on the C870 and GTX 280.

Table 3 presents benchmark results for the long-range lattice cutoff portion of MSM on the CPU and GPU. Two GPU algorithm variations are reported. The "Latcut-basic" algorithm is a direct implementation of the lattice cutoff algorithm described earlier in Section 3.2. The "Latcut-optimized" algorithm eliminates almost half of the global memory reads by shifting and copying previously-loaded lattice data within GPU shared memory, and uses loop unrolling and code hoisting techniques to improve the performance of the innermost loop. The resulting runtimes for the lattice cutoff kernels place them at 9% (CPU) and 8% (GPU) of the runtime of the complete MSM computation.

The results shown in Table 4 summarize the performance of the full MSM calculation on the host CPU, with GPU acceleration on a first-generation G80-based Tesla C870 GPU and a second-generation GT200-based GTX 280 GPU. The runtimes listed are for the full MSM electrostatics calculation, including all algorithm components on both the CPU and GPU, and all host-GPU I/O. The results for the D870 task-parallel approach utilize two CPU cores and two GPUs to simultaneously compute the short-range and long-range kernels on separate devices and CPU cores. The results for the three G80 devices combine task- and data-parallel approaches: the long-range part is computed using one CPU core with the GeForce 8800 GTX GPU, while the short-range part is partitioned into two equal pieces and computed simultaneously using two additional CPU cores with the two GPUs of the D870. The testing platform for this final configuration uses a slower 2.4 GHz Intel Core 2 Q6600 quad core CPU, while the speedups are calculated relative to a faster CPU.

A plot of speedup versus problem size for the full MSM calculation is shown in Fig. 5. The GPU-accelerated MSM implementation performs well for a wide range of problem sizes, achieving 50% of its peak speedup for a test case with only 26,247 atoms, and 77% of its peak speedup for a test case containing 167,877 atoms.

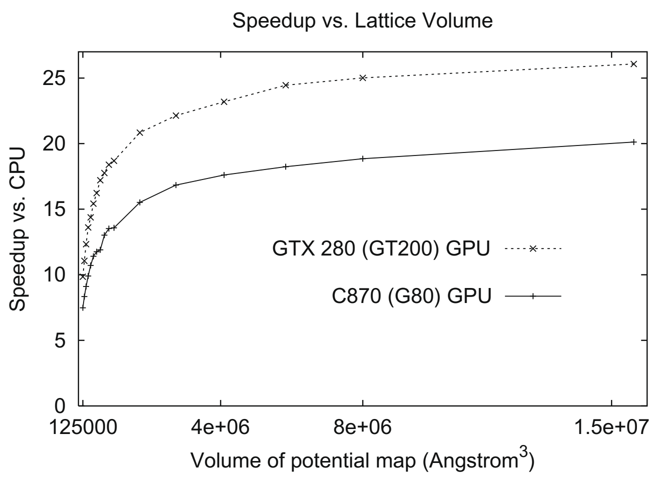

Fig. 5. Speedup over varying problem sizes for the full GPU-accelerated MSM potential map calculation relative to an optimized CPU implementation.

Table 5
Time profile of GPU-accelerated MSM potential map calculation for 1,534,539 atom system. The overall time with the GT200-based GTX 280 GPU was 20.51 s. The long-range part is decomposed into its sequential dependencies.

	Time in seconds	Percentage of total
Device setup	0.15	0.73
Short-range part (GPU + CPU)	14.87	72.50
Anterpolation	0.18	0.88
Restriction, levels $0, 1, \ldots, 7$	0.16	0.78
Lattice cutoff (GPU + CPU)	1.36	6.63
Prolongation, levels $7, 6, \ldots, 1$	0.17	0.83
Interpolation	3.47	16.92
Long-range part (GPU + CPU)	5.34	26.04
Summing short- and long-range maps	0.15	0.73

Table 5 shows a detailed performance profile for the full GPU-accelerated MSM calculation running on the GeForce GTX 280, summarized in Table 4. Comparing these results to the sequential CPU performance profile in Table 1, we see that the largest sequential bottleneck now remaining, the interpolation of the finest level MSM lattice to the electrostatic potential map, is still less than one quarter of the accelerated short-range part, and its acceleration would permit a speedup of no more than 31. The "device setup" measures the time required to attach a host thread to a GPU device, a fixed cost that becomes inconsequential for longer calculations. The "summing short- and long-range maps" measures the time for the host CPU to add together the two electrostatic potential maps of size 541^3 (604 MB) for the final result. We note that the performance of this summation step is limited by the memory bandwidth of the host CPU.

5. Discussion

MSM is an efficient and flexible algorithm that can be applied to different boundary conditions and can even be generalized to other (i.e., non-Coulombic) pairwise potentials, although molecular modeling benefits the most from approximating the long-range contribution from a $1/r$ dependence. While Eq. (9) shows that the accuracy of MSM can be improved by increasing the cutoff distance a or decreasing the finest lattice spacing h, it turns out that increasing the ratio a/h quickly degrades performance, increasing the computational complexity of the short-range part as $O(a^3)$ and of the long-range part as $O(a^3/h^6)$. A better choice is to increase the order of accuracy p by using a higher degree interpolating polynomial to construct basis function Φ than given by Eq. (7). A higher degree interpolating polynomial increases the cost of the anterpolation, restriction, prolongation, and interpolation parts, although these do not dominate the overall cost. The use of higher order interpolation also warrants increasing the continuity of the smoothing function γ, an issue that has already been examined in detail [7]. With respect to our GPU implementation of MSM, we note that the small-bin kernel depends on γ, but is easily modified to accommodate a higher continuity smoothing function. The lattice cutoff kernel does not depend explicitly on γ, rather on the CPU to compute the weights that depend on γ. Neither kernel depends on Φ.

The experimental test results presented in Section 4 demonstrate the performance of the GPU implementation of MSM. Several observations about these results and our experiences in developing and testing the implementation are generally applicable when designing other GPU-accelerated codes. In the initial state of algorithm design, we began with a profile of the runtime of the various algorithmic components for an existing CPU implementation. From this point, we observed Amdahl's Law and selected the most time consuming portions of the full MSM calculation for detailed study. The effective use of heterogeneous accelerator devices such as GPUs hinges upon balancing the tasks of transferring data to and from the accelerator, launching and managing kernel execution, and integrating results back into the main host program, along with any other computations that can only be run on the host CPU.

Many of the earliest successes with GPU acceleration were achieved for algorithms with quadratic complexity in arithmetic operations relative to the input problem size. Algorithms with quadratic arithmetic complexity easily cover the costs of I/O transfers and other associated sources of overhead, and often perform well even for very small problem sizes. Algorithms with linear arithmetic complexity can be more challenging to accelerate, as sufficient work must be assigned to the accelerator to amortize the I/O overhead. The ratio of arithmetic complexity relative to the I/O is of fundamental concern when designing GPU algorithms. If it is not immediately clear that arithmetic will dominate I/O overhead, then one must take steps to reduce I/O or eliminate it altogether by performing consecutive algorithmic steps solely on the GPU. The optimized MSM lattice cutoff kernel described in this paper uses shared memory to reduce redundant global memory operations and takes advantage of constant memory to broadcast the weight values to all threads in the same thread block. Both of these strategies amplify the effective bandwidth of operands to the GPU arithmetic units and help make the kernel arithmetic bound. Another strategy for achieving high performance with GPU acceleration is to target the GPU-accelerated computations on a regularized problem, using the host CPU to "clean up" or handle exceptional cases that might otherwise reduce the performance of the GPU. This strategy is employed very effectively in the small-bin kernel for the MSM short-range part.

Our initial MSM implementation decomposed the entire problem into a single CPU–GPU pair for each stage of the algorithm. We next attempted to improve performance by executing the independent short- and long-range parts of MSM on multiple CPU–GPU pairs. While effective, the benefit from such a task-parallel approach is limited by the imbalance between the two parts, offering just a 27% speed increase on our largest test case. The more computationally demanding short-range part readily lends itself to a data-parallel decomposition, accomplished by partitioning the electrostatic potential map into equally sized slabs to be computed concurrently. Our test case combining task- and data-parallel approaches, using two CPU–GPU pairs on the short-range part concurrently with another CPU–GPU pair on the long-range part, provides a 133% speed improvement over the single C870 GPU. The performance profile in Table 5 reveals a ratio between the short- and long-range timings of almost 3:1, suggesting that a platform with four GPUs could use our present implementation to achieve 80–90$\times$ speedups for the largest test case. More importantly, the data-parallel decomposition of the short-range part removes the primary memory constraint on our initial GPU-accelerated MSM implementation, permitting the calculation of electrostatic potential maps for system sizes approaching 100 million atoms.

6. Conclusions

We have described algorithms and techniques for efficiently implementing the multilevel summation method for computing electrostatic potential maps on heterogeneous processing resources with GPU-accelerated computation. Our experimental results demonstrate the feasibility of algorithmic approaches that use the CPU to regularize the work assigned to GPU accelerator devices to achieve high performance, and show the ability of the GPU runtime system to perform operations asynchronously from the host to be a key enabling technology for this purpose.

The ability to calculate maps of the electrostatic potential for 1-million-atom systems within seconds enables interactive analysis previously unavailable. The implementation described in this paper is included in the `cionize` ion placement tool distributed with VMD [31]. We also plan to incorporate the algorithm directly into a future version of VMD itself, bringing a significant performance increase to other visualization and analysis tasks, such as the calculation of mean field potential maps for molecular dynamics simulations. Work is in progress to develop GPU-accelerated kernels for the multilevel summation short-range force calculation necessary for dynamics, with the implementation being done initially in the NAMD-Lite framework and ultimately for NAMD [16,17].

Acknowledgements

We are grateful to Purdue University Prof. Robert D. Skeel for introducing us to the multilevel summation method and providing valuable insight into its use for molecular dynamics. We are also grateful to University of Illinois Prof. Wen-mei Hwu and to NVIDIA Chief Scientist David Kirk for introducing us to the CUDA technology and for many valuable discussions since. We would also like to thank Chris Rodrigues and John Stratton of the IMPACT group at University of Illinois for their insights. Performance experiments on the Tesla D870, an engineering sample prototype T10P GPU, and the GeForce GTX 280 were made possible by a generous hardware donation by NVIDIA. This work was supported by the National Institutes of Health under Grant P41-RR05969 and the University of Illinois through a grant from the Institute for Advanced Computing Applications and Technologies.

References

[1] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, Klaus Schulten, Accelerating molecular modeling applications with graphics processors, J. Comp. Chem. 28 (2007) 2618–2640.

[2] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, James C. Phillips, GPU computing, Proc. IEEE 96 (2008) 879–899.
[3] Joshua A. Anderson, Chris D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Chem. Phys. 227 (10) (2008) 5342–5359.
[4] I.S. Ufimtsev, T.J. Martinez, Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation, J. Chem. Theory Comput. 4 (2) (2008) 222–231.
[5] Christopher I. Rodrigues, David J. Hardy, John E. Stone, Klaus Schulten, Wen-Mei W. Hwu, GPU acceleration of cutoff pair potentials for molecular modeling applications, in: CF'08: Proceedings of the 2008 Conference on Computing Frontiers, ACM, New York, USA, 2008, pp. 273–282.
[6] Robert D. Skeel, Ismail Tezcan, David J. Hardy, Multiple grid methods for classical molecular dynamics, J. Comp. Chem. 23 (2002) 673–684.
[7] David Joseph Hardy, Multilevel Summation for the Fast Evaluation of Forces for the Simulation of Biomolecules, PhD Thesis, University of Illinois at Urbana-Champaign, 2006, Also Department of Computer Science Report No. UIUCDCS-R-2006-2546, May 2006. Available from <http://www.cs.uiuc.edu/research/techreports.php>.
[8] T.A. Darden, D.M. York, L.G. Pedersen, Particle mesh Ewald. An $N \cdot \log(N)$ method for Ewald sums in large systems, J. Chem. Phys. 98 (1993) 10089–10092.
[9] U. Essmann, L. Perera, M.L. Berkowitz, T. Darden, H. Lee, L.G. Pedersen, A smooth particle mesh Ewald method, J. Chem. Phys. 103 (1995) 8577–8593.
[10] D. Barash, L. Yang, X. Qian, T. Schlick, Inherent speedup limitations in multiple time step/particle mesh Ewald algorithms, J. Comput. Chem. 24 (2003) 77–88.
[11] L. Greengard, V. Rokhlin, A fast algorithm for particle simulation, J. Comp. Phys. 73 (1987) 325–348.
[12] John A. Board Jr., J.W. Causey, James F. Leathrum Jr., Andreas Windemuth, Klaus Schulten, Accelerated molecular dynamics simulation with the parallel fast multipole algorithm, Chem. Phys. Lett. 198 (1992) 89–94.
[13] Thomas C. Bishop, Robert D. Skeel, Klaus Schulten, Difficulties with multiple time stepping and the fast multipole algorithm in molecular dynamics, J. Comp. Chem. 18 (1997) 1785–1791.
[14] A. Brandt, A.A. Lubrecht, Multilevel matrix multiplication and fast solution of integral equations, J. Comput. Phys. 90 (1990) 348–370.
[15] Bilha Sandak, Multiscale fast summation of long range charge and dipolar interactions, J. Comp. Chem. 22 (7) (2001) 717–731.
[16] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, Klaus Schulten, NAMD2: greater scalability for parallel molecular dynamics, J. Comp. Phys. 151 (1999) 283–312.
[17] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kale, Klaus Schulten, Scalable molecular dynamics with NAMD, J. Comp. Chem. 26 (2005) 1781–1802.
[18] Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym, NVIDIA Tesla: a unified graphics and computing architecture, IEEE Micro. 28 (2) (2008) 39–55.
[19] NVIDIA, CUDA Compute Unified Device Architecture Programming Guide, NVIDIA, Santa Clara, CA, USA, 2007.
[20] John Nickolls, Ian Buck, Michael Garland, Kevin Skadron, Scalable parallel programming with CUDA, ACM Queue 6 (2) (2008) 40–53.
[21] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan, Brook for GPUs: stream computing on graphics hardware, in: SIGGRAPH'04: ACM SIGGRAPH 2004 Papers, ACM Press, New York, USA, 2004, pp. 777–786.
[22] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, Kevin Moule, Shader algebra, ACM Transactions on Graphics 23 (3) (2004) 787–795. August.
[23] Maria Charalambous, Pedro Trancoso, Alexandros Stamatakis, Initial experiences porting a bioinformatics application to a graphics processor, in: Panhellenic Conference on Informatics, 2005, pp. 415–425.
[24] Daniel Reiter Horn, Mike Houston, Pat Hanrahan, ClawHMMER: a streaming HMMer-search implementation, in: SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2005, p. 11.
[25] Erich Elsen, V. Vishal, Mike Houston, Vijay Pande, Pat Hanrahan, Eric Darve, N-body simulations on GPUs, Technical Report, Stanford University, Stanford, CA, June 2007. <http://arxiv.org/abs/0706.3060>.
[26] Michael McCool, Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform, in: GSPx Multicore Applications Conference, October/November 2006.
[27] Advanced Micro Devices Inc., Brook+ SC07 BOF session, in: Supercomputing 2007 Conference, November 2007.
[28] Apple Computer Inc., OpenCL, in: Apple Worldwide Developer's Conference, June 2008.
[29] John A. Stratton, Sam S. Stone, Wen mei W. Hwu, MCUDA: an effective implementation of CUDA kernels for multi-core cpus, in: Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing, LNCS 5335, 2008, pp. 16–30.
[30] G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon, D.W. Walker, Solving Problems on Concurrent Processors, vol. 1, Prentice Hall, Englewood Cliffs, NJ, 1988.
[31] William Humphrey, Andrew Dalke, Klaus Schulten, VMD – visual molecular dynamics, J. Mol. Graphics 14 (1996) 33–38.
[32] Peter L. Freddolino, Anton S. Arkhipov, Steven B. Larson, Alexander McPherson, Klaus Schulten, Molecular dynamics simulations of the complete satellite tobacco mosaic virus, Structure 14 (2006) 437–449.