GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications

Christopher I. Rodrigues[*]
cirodrig@crhc.uiuc.edu

David J. Hardy[†]
dhardy@ks.uiuc.edu

John E. Stone[†]
johns@ks.uiuc.edu

Klaus Schulten[†]
kschulte@ks.uiuc.edu

Wen-Mei W. Hwu[*]
hwu@crhc.uiuc.edu

[†]Beckman Institute
University of Illinois at
Urbana-Champaign

[*]Department of Electrical and
Computer Engineering
University of Illinois at
Urbana-Champaign

ABSTRACT

The advent of systems biology requires the simulation of ever-larger biomolecular systems, demanding a commensurate growth in computational power. This paper examines the use of the NVIDIA Tesla C870 graphics card programmed through the CUDA toolkit to accelerate the calculation of cutoff pair potentials, one of the most prevalent computations required by many different molecular modeling applications. We present algorithms to calculate electrostatic potential maps for cutoff pair potentials. Whereas a straightforward approach for decomposing atom data leads to low compute efficiency, a newer strategy enables fine-grained spatial decomposition of atom data that maps efficiently to the C870's memory system while increasing work-efficiency of atom data traversal by a factor of 5. The memory addressing flexibility exposed through CUDA's SPMD programming model is crucial in enabling this new strategy. An implementation of the new algorithm provides a greater than threefold performance improvement over our previously published implementation and runs 12 to 20 times faster than optimized CPU-only code. The lessons learned are generally applicable to algorithms accelerated by uniform grid spatial decomposition.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiprocessors*; I.6.3 [**Computing Methodologies**]: Simulation and Modeling—*Applications*; J.3 [**Computer Applications**]: Life and Medical Sciences

General Terms

Algorithms, Performance

CF'08, May 5–7, 2008, Ischia, Italy.
Copyright 2008 ACM 978-1-60558-077-7/08/05 ...$5.00.

Keywords

Molecular dynamics, graphics processors, GPGPU, CUDA

1. INTRODUCTION

Molecular modeling is an indispensable approach to understanding chemical and biomolecular systems. The modeling and visualization of atomic level details provides insight into the function and dynamics of biomolecular structures that can ultimately benefit human health through improved understanding of the molecular basis of disease, designing of pharmaceuticals, and engineering of bionanodevices. With continuing advances in computing technology, the size of feasible simulations has grown to encompass larger systems and longer time scales. The desire for more computational power has stimulated the development of highly parallel algorithms [13], heterogeneous multicore implementations [25], and specialized hardware such as MD-GRAPE [29] and Anton [24].

Due to their computational power, widespread availability, and ease of integration into contemporary computer systems, graphics processing units (GPUs) are a useful platform on which to accelerate molecular modeling applications. Since graphics rendering is inherently data-parallel, GPUs have gradually evolved into highly parallel processors to achieve tremendous computational throughput. State-of-the-art GPUs can perform up to a half trillion floating point operations per second, a peak arithmetic performance level once associated only with supercomputers. Massive multithreading, fast context switching, and high memory bandwidth enable GPUs to tolerate ever-increasing latencies, run many small cores in parallel, and sustain high computational throughput. Fixed-function graphics pipelines are being replaced by fully programmable cores supporting features important for general-purpose computing such as IEEE floating point arithmetic, a multiprocessor shared memory, and arbitrary memory addressing [1, 18]. These recent advances in GPU architecture have enabled them to be used as high-performance, massively parallel processing engines for scientific computing.

Until recently, GPU programming required the use of graphics application programming interfaces (APIs) such as OpenGL, or a high-level language layered on top of graphics APIs [5, 30]. Routines were limited to computation that could be expressed in terms of graphics drawing operations, forcing a cumbersome or inefficient expression of algorithms. For example, inter-thread communication and writing to arbitrary memory addresses could not be used [20]. New GPU software interfaces allow computations to be expressed much more naturally as communicating groups of threads in an SPMD programming model and expose more of the functionality available in the hardware [18].

The most prevalent and costly computation required by molecular modeling applications is the calculation of nonbonded pair potentials which describe the electrostatic and van der Waals interactions between pairs of atoms. The quadratic computational complexity of considering all pairs of atoms is generally reduced to a problem of linear complexity by introducing a cutoff distance, beyond which the pair potential is set to zero. The use of a cutoff potential is the method of choice for calculating van der Waals interactions, and the cutoff calculation comprises the short-range interaction contribution for many methods that approximate the full electrostatic potential.

In this paper, we investigate the GPU-accelerated calculation of electrostatic potential maps, which consist of a regularly spaced lattice of points in space containing the summed potential contributions from the atoms. The computation of electrostatic potential maps is directly applicable to methods for ion placement, mean field molecular dynamics simulation, molecular visualization, and other various tasks used for analysis. The algorithmic approaches examined here can also be applied to solve the more general N-body problem of computing the potential and forces between pairs of atoms. This work is based on previously published results with the NVIDIA G80 hardware used to accelerate calculations of both the direct (infinite distance) and cutoff (range-limited) potential maps [28]. We specifically discuss a new implementation of the cutoff potential with a performance speedup of more than a factor of 20 compared to optimized CPU-only code and greater than a threefold improvement over a previous GPU implementation. This exploration not only reveals techniques particular to the NVIDIA G80 and C870, but also provides more general insight into adapting algorithms to intensively data parallel architectures.

2. BACKGROUND

Molecular modeling can provide scientists with a detailed view of a system of atoms and its evolution over time. The demand for simulating larger system sizes for longer timescales has grown over the years together with available computer power. Improvements in computer technology already allow the simulation of large biomolecular systems, with the largest simulations comprising 100,000 to 1,000,000 atoms. Simulating for long timescales, of 100 ns (100 million time steps) or longer, is important for studying the function of biomolecules. The ability to harness the computational power of GPUs to achieve an order of magnitude greater performance will offer new opportunities to biomedical researchers for simulating larger system sizes and longer timescales.

2.1 Cutoff Pair Potentials in Molecular Modeling

Biomolecular systems are generally modeled in a classical framework. The atomic motions are governed by Newton's equations, with distance-dependent atomic forces described by a semi-empirical forcefield that accounts for interaction energies among chains of covalently bonded atoms and between pairs of nonbonded atoms. Determining the atomic forces is by far the most computationally costly part of molecular simulation. The work due to bonded forces grows linearly with the number of atoms, whereas the number of nonbonded interactions increases quadratically as the number of atomic pairs in the system. The computational complexity has traditionally been reduced by using a cutoff pair potential in which the pairwise interaction is zero beyond a cutoff distance. The computational effort for simulation is compounded by the need to compute the atomic forces at every timestep, where the step size is restricted to the order of femtoseconds. The cost of force computation over the duration of a full simulation can add up to anywhere from hours to years of CPU time.

The nonbonded electrostatic interactions between atoms are of particular importance in molecular simulation. Atoms are modeled as point charges by assigning to each atom i at position $\vec{r}_i$ a fixed partial charge q_i. Closely related to the electrostatic force and interaction energy between atoms is the electrostatic potential V at a position $\vec{r}$, expressed here as a sum over all N atoms,

$$V(\vec{r};\vec{r}_1,\vec{r}_2,\ldots,\vec{r}_N) = \sum_{i=1}^{N} \frac{q_i}{4\pi\varepsilon_0|\vec{r}-\vec{r}_i|} s(|\vec{r}-\vec{r}_i|), \tag{1}$$

with dielectric constant ε_0, where the $(1/r)$-factor is due to the physical model, and the unitless scaling factor $0 \leq s(r) \leq 1$ is included to improve computational efficiency. Equation (1) is often sampled at regularly spaced points over a volume to generate a map of the electrostatic potential in that volume. In this case, $\vec{r}$ ranges over a set of M regularly spaced lattice points. Setting the function $s(r) \equiv 1$ calculates the full (infinite distance) electrostatic potential contributed by all atoms to position $\vec{r}$, requiring quadratic computational work. Algorithmic efficiency is improved by choosing $s(r)$ to yield a cutoff potential truncated beyond a fixed cutoff distance r_c. A common choice (e.g., as used by molecular dynamics program NAMD [13]) is given by

$$s(r) = \begin{cases} (1-r^2/r_c^2)^2, & \text{if } r < r_c, \\ 0, & \text{otherwise}, \end{cases} \tag{2}$$

which smoothly shifts the potential to zero beyond r_c and regains the full electrostatic potential in the limit as r_c approaches infinity.

Efficient computation of cutoff pair potentials is of particular importance due to their prominence in molecular modeling applications. Molecular simulations were, in their earliest days, made tractable by using an electrostatic potential of the form (1) with $s(r)$ chosen as (2) or similarly to truncate the nonbonded interactions. The form of equation (1) also includes the short-range parts of modern methods that approximate the full electrostatic potential, such as particle-mesh Ewald [6], particle-particle particle-mesh [11], and multilevel summation [26], made available by choosing different scaling functions $s(r)$. The nonbonded van der Waals interactions, accounting for both the hardcore repulsion between nearby atoms and the weak dipole attraction between more distant atoms, have a different potential form than (1) but are similarly evaluated using a cutoff distance.

In this paper, we consider two algorithmic approaches to computing electrostatic potential maps described by (1):

1. the *direct summation* approach which uses $s(r) \equiv 1$ for $O(MN)$ computational work, summing to each of M grid points the pair potential contributions from all N atoms, and

2. the *cutoff summation* approach which defines $s(r)$ by equation (2) for $O(M+N)$ computational work, summing to each of M grid points the pair potential contributions from all atoms within cutoff distance r_c.

We focus on innovative techniques to accelerate cutoff summation using the GPU. The basic algorithm for evaluating a cutoff pair potential involves performing spatial hashing of the atoms into bins of a uniform grid [8], then considering for each bin its interactions with nearby bins that lie within or along the cutoff distance. In collision detection parlance, the spatial hashing step is called the *broad phase* and subsequent distance tests are called the *narrow* *phase*. Since biomolecules exhibit a generally uniform density of about 1 atom per $10\,\text{Å}^3$, the cutoff summation approach for a fixed bin size yields a constant amount of work per bin, with the number of bins scaling linearly with the problem size, hence the linear scaling for total computational work.

The computation of electrostatic potential maps is directly applicable to ion placement methods used to initially setup a biomolecular system [28], where a lattice spacing of $0.5\,\text{Å}$ is often used in practice, providing a reasonable compromise between resolution of the electrostatic potential map and the cost of computing it. A cutoff distance of $8\,\text{Å} \leq r_c \leq 12\,\text{Å}$ is typical for molecular simulation. For $0.5\,\text{Å}$ lattice spacing and $12\,\text{Å}$ cutoff distance, with our best performing implementation storing bins of size $4\,\text{Å}$ with 8 atom slots per bin plus 3 bins worth of padding surrounding the cubic domain, the 1536 MB of memory on the C870 will admit in a single computation pass problem sizes up to a cubic domain of length $360\,\text{Å}$ containing approximately 4.67 million atoms. The majority of biomolecular complexes that are studied computationally tend to be on the order of $100,000$ atoms in a $10^6\,\text{Å}^3$ volume. Since water molecules can comprise as much as 90% of a biomolecular complex, we benchmark our implementations using water boxes created by the solvate plugin included with VMD [12], with an atom density representative of typical molecular dynamics simulations of biomolecular complexes.

In addition to molecular simulation setup, electrostatic potential maps are also useful for analysis and visualization of the completed simulation, for instance, when visualizing the electrostatic contour lines around a molecular surface. Advanced simulation techniques exist that use time-averaged potential maps to realistically model reduced parts of a much larger system too costly to simulate using a full atom representation [28]. The techniques presented here for accelerating particle-grid interactions can also be generalized to solve the N-body problem involving particle-particle interactions.

2.2 GPU Acceleration of Related Workloads

Although molecular simulations are run on systems with a wide range of computing power, ranging from desktops to supercomputers, there are few GPU-accelerated molecular modeling algorithms in the literature. The Folding@Home project has developed a GPU-based folding client derived from the GROMACS molecular dynamics engine, implemented using the Brook stream programming system [5, 4, 7, 27, 20]. Another early experiment with molecular dynamics on GPUs and other emerging architectures was reported [16], using the Cg graphics shading language. More recently, initial GPU-accelerated implementations of the NAMD molecular dynamics simulation package have been reported [28, 20], based on the CUDA GPU computing interface to the current generation GPU hardware. These algorithms compute cutoff pair interactions between atoms, ultimately resulting in forces that determine the dynamics of simulated molecules. The present work presents algorithms for computation of electrostatic potentials on a uniformly spaced lattice. The spatial regularity of this problem results in a different algorithmic approach that benefits much more from precomputation and parallel data reuse. This work significantly improves on the short-range component of the electrostatic potential map calculation algorithms in VMD [12, 28], as described in Section 4.2.

Short-range pairwise interactions are a component of several computational tasks that have been implemented or accelerated using GPUs. Probably the most similar problem domain is Smoothed Particle Hydrodynamics (SPH), where fluids are simulated as particles interacting through various short-range forces. Several authors [2, 9, 15] have accelerated SPH using GPU shaders, of which [2, 9] spatially partition the particles. These algorithms were developed within the constraints of shader programming and do not necessarily take advantage of the full hardware capabilities available in modern GPUs. The spatial partitioning used in this paper takes advantage of a near-uniform density of particles, which also holds in fluid-filled volumes in SPH simulations. Other GPU-accelerated computations utilizing spatial partitioning include radiance calculations for image rendering [21], N-body force calculation [19], and game physics [20].

3. THE TESLA C870 GRAPHICS CARD

We use the Tesla C870 as a data-parallel coprocessor for scientific computing. The graphics card is, for our purposes, a large set of processor cores that are able to directly address a global memory. It supports an SPMD programming model, which is more general and flexible than the streaming programming models supported by previous generations of GPUs. In this section we discuss the programming model of NVIDIA's Compute Unified Device Architecture (CUDA) and the microarchitectural features of the C870 that are most relevant to accelerating cutoff pair potential computation. A more complete description is found in [17, 18]. We also touch briefly on general optimization principles for GPU programming.

A CUDA program consists of host code that executes on a system's CPU and device code that executes on a compute device such as a GPU. Code that exhibits ample parallelism suitable for GPU execution is written as device code, which is compiled by NVIDIA's compiler and loaded onto the GPU by a runtime library. Device code is written using ANSI C extended with keywords for labeling parallel functions, called kernels, and their associated data structures. A kernel, when run, generates many lightweight threads on the device to exploit parallelism. Code that exhibits little parallelism or is otherwise unsuitable for device execution is written as host code, which is compiled with the host's standard C compiler and runs as an ordinary process. The host transfers data to and from the GPU's global memory via API calls, and initiates kernel code by performing a function call.

The host and device execute asynchronously with respect to each other. After initiating a kernel, the host does not wait for it to complete, and may execute in parallel with the device. Data transfers between host and device may be performed either synchronously or asynchronously. It is possible to explicitly synchronize.

3.1 Microarchitecture of the Tesla C870

The C870 is equipped with a G80 GPU, whose microarchitecture is depicted in Figure 1. It consists of 16 *streaming multiprocessors* (SMs), each containing eight *streaming processors* (SPs), or processor cores, running at 1.35 GHz. Each SM has 8,192 registers that are shared among all threads assigned to the SM. Each core has a lone arithmetic unit that performs single-precision floating point arithmetic and 32-bit integer operations. Additionally, each SM has two *special functional units* (SFUs), which perform more complex FP operations such as inverse square root and trigonometric functions with high throughput. Both the arithmetic units and the SFUs are fully pipelined. Thus, each of the 16 SMs can perform 18 FLOPS per clock cycle (1 multiply-add operation per SP and one complex operation per SFU), yielding 388.8 GFLOPS of peak theoretical performance for our purposes.

Threads executing on the G80 are organized into a three-level hierarchy. At the highest level, each kernel creates a single *grid*, which consists of many *thread blocks*. Thread blocks are co-scheduled groups of threads that can share data through a fast, writable shared memory and synchronize with one another using barrier instructions. A thread block consists of up to 512 threads;

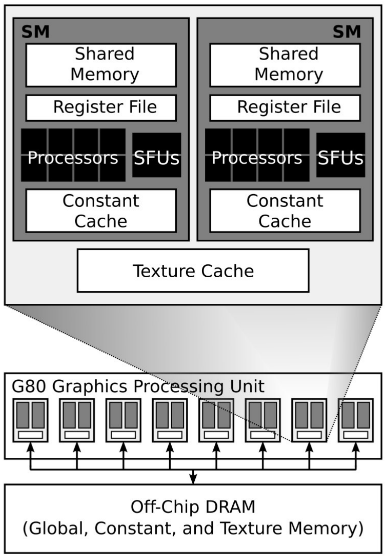

Figure 1: Basic organization of the Tesla C870.

the number is usually chosen statically by the developer for each kernel. The hardware initiates thread block execution dynamically as resources become available. Each thread block is assigned to a single SM for the duration of its execution.

Threads within a block are organized into *warps* of 32 threads. Each warp executes in SIMD fashion, with the SM's instruction unit broadcasting the same instruction to the eight cores on four consecutive clock cycles. If threads within a warp take different control paths (referred to as *branch divergence*), it is assumed that multiple passes with suppression of threads on divergent paths are required to complete execution.

SMs can perform zero-overhead scheduling to interleave warps on an instruction-by-instruction basis to hide the latency of global memory accesses and long-latency arithmetic operations. When one warp stalls, the SM can quickly switch to a ready warp in the same thread block or a ready warp in some other thread block assigned to the SM. The SM stalls only if there are no warps with ready operands available. Scheduling freedom is high in many applications because threads in different warps are independent with the exception of explicit synchronization among threads in the same thread block.

The C870 has 76.8 GB/s of bandwidth to its 1536MB off-chip global memory. With computational resources supporting nearly 400 GFLOPS of performance and each FP instruction operating on up to 12 bytes of data, applications can easily saturate that bandwidth. To fully use the available global bandwidth, the 16 threads in the first or last half of a warp must each concurrently access contiguous, aligned blocks of memory so that the 16 accesses will be coalesced into a single large-word access; otherwise, more bandwidth is consumed as the hardware services each access separately.

With judicious use of several on-chip memories, depicted in Figure 1, an application can exploit data locality and data sharing to reduce its demands for off-chip memory bandwidth. For example, the G80 GPU has a 64KB, off-chip *constant memory*, and each SM has an 8KB constant memory cache. Because the cache is single-ported, simultaneous accesses of different addresses yield stalls. However, when multiple threads access the same address during the same cycle, the cache broadcasts the address's value to those threads with the same latency as a register access. Each SM also has a 16KB writable *shared memory* that is useful for software-managed data reuse or communication among threads in a thread block. There is also a cacheable *texture memory*, but we do not use it since its latency is much longer than that of shared or constant memory.

3.2 Software Optimization Principles

The increased programmability afforded by CUDA and additional hardware features on the G80 remove many of the constraints imposed by shader programming. For example, the availability of shared memory on an SM and barrier synchronization makes inter-thread coordination and communication possible, along with consequent software optimizations and algorithm implementations that are not possible under a shader programming model. Nevertheless, a developer aiming to maximize GPU performance must still adapt his program to the graphics-oriented capabilities of the GPU hardware. The first-order performance concerns imposed by the GPU's hardware are to hide the latency of slow operations (SFU arithmetic and global loads and stores) through parallelism, manage the global memory bandwidth, and minimize the number of dynamically executed instructions [22]. Latency hiding is achieved by ensuring that many threads are simultaneously active, so that there are always unstalled threads available for execution. Global bandwidth is used efficiently by using cacheable memory, such as constant memory, or by explicitly using shared memory as a software-managed cache.

Tuning the performance of a CUDA kernel often involves a fundamental trade-off between the efficiency of individual threads and the thread-level parallelism among all threads. This trade-off exists because many optimizations that improve the performance of an individual thread tend to increase the thread's use of limited resources that are shared among all threads assigned to an SM. For example, as each thread's register usage increases, the total number of threads that can simultaneously occupy the SM decreases. Because threads are assigned to an SM at a granularity of thread blocks, a small increase in register usage can cause a much larger relative decrease in SM occupancy. Currently, only experience and manual experimentation can produce a set of optimizations whose resource use is cost-effective for a given kernel [23].

4. IMPLEMENTATION

This section discusses the implementation of the direct and cutoff summation algorithms described in Section 2 for computing electrostatic potential maps. The potential value at each point in a lattice can be computed independently, offering a natural decomposition of the work. The considerations dealt with in crafting an implementation are how to allocate the work to computing resources, use memory efficiently, and avoid redundant work. In the remainder of this section, we discuss the implementation and performance-enhancing optimizations of three GPU algorithm variants of increasing sophistication and performance and an optimized CPU algorithm variant. The algorithm variants in sections 4.1 and 4.2 are adapted from the direct and cutoff kernels, respectively, in [28]. Section 4.3 presents our new, faster algorithm. Section 4.4 presents our CPU algorithm.

4.1 Direct Summation

For direct summation, the computation for each potential map point is a straightforward translation of equation (1). Each parallel thread loops over all atoms in the system and maintains a running total of the potential for one or more lattice points. Although the quadratic complexity of direct summation makes it undesirable for large problem sizes, its simplicity and uniformity makes it much easier to implement. Aspects of its programming and optimization

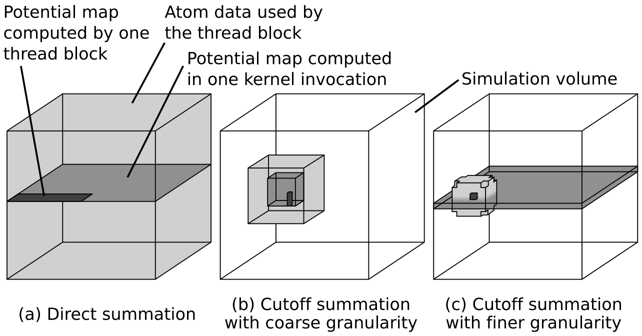
Potential map computed by one thread block
Atom data used by the thread block
Potential map computed in one kernel invocation
Simulation volume
(a) Direct summation
(b) Cutoff summation with coarse granularity
(c) Cutoff summation with finer granularity

Figure 2: Data access patterns of a CUDA grid and thread block for different potential summation implementations.

remain relevant to the more sophisticated cutoff summation variants.

Computation is assigned to threads pursuant to a spatial decomposition of work. The potential map volume is first decomposed into individual planes which may be computed independently. Each plane is computed using a separate CUDA kernel call. Each plane is further decomposed into 2-D tiles, each of which is computed in a thread block. Each thread within a thread block computes the potential at eight lattice points within the 2-D tile. The decomposition is illustrated in Figure 2(a). The dimensions of a 2-D tile are constrained by performance optimizations as described below; to compute potentials on arbitrary size volumes, the potential map size is rounded up to an integral number of tiles.

Taking full advantage of the parallelism in the hardware requires tailoring the implementation to efficiently utilize the hardware's data transfer mechanisms. The algorithm demands high memory throughput for reading atom data, since every thread traverses the entire set of atoms in the system. We place atom data in the GPU's constant memory, which satisfies the demand through the hardware's constant-memory caches and its ability to service multiple concurrent reads of the same datum with a single cache access. The major drawback of constant memory is its limited size. It can hold just over 4000 atom coordinates and charges, necessitating a multi-step computation that interleaves GPU processing with CPU-mediated data transfers to reload constant memory between steps. Nonetheless, the overhead is dwarfed by the time spent in kernel computation. With this data layout, the kernel is not limited by memory bandwidth.

In the now compute-bound kernel, some components of the per-atom distance calculation are constant for individual planes and rows within the map. Data reuse and precomputation optimizations reduce the amount of redundant arithmetic. We enhance performance by using a thread to evaluate an atom's potential energy contribution to eight points in a row, aggregating several threads' work in a transformation analogous to loop unroll-and-jam [14]. This enables value reuse across the eight points and amortizes memory references over a larger number of arithmetic operations. A less obvious benefit is that the unrolled threads make more efficient use of the shared register file on an SM by economizing on register values that are not replicated by the unroll-and-jam transformation. The register file can thus accommodate more lattice points' worth of simultaneously active threads.

Further performance gain comes from reducing the data transfer time for updating the potential map values, which are stored in global memory. We assign each thread a group of potential map points with a stride of 16 so that lattice points simultaneously computed by a half-warp are contiguous in memory. This interleaving enables the hardware to coalesce reads and writes of potential map data and reduces the time spent in global memory accesses. The combination of unroll-and-jam and strides forces a thread block to compute a tile a multiple of $16 \times 8 = 128$ potential map points wide.

Data	Direct	Cutoff (coarse)	Cutoff (fine)
Potential map	Global	Global	Global
Atoms	Host	Host	Global
Atom working set	Constant	Constant	Shared

Table 1: CUDA memory storage used in different potential summation implementations.

4.2 Cutoff Summation with Large Bin Sizes

The chief speed gain in cutoff calculation comes through skipping atoms that are known to be outside the cutoff radius for a localized region of the potential map. Whereas direct summation required the entire set of atom data to compute a part of the potential map (Figure 2(a)), cutoff summation requires only the data for atoms lying within the cutoff radius of some point in the region. To minimize the volume whose atom data needs to be processed, we reorganize the computation so that each kernel computes a cube-shaped volume of the potential map (Figure 2(b)) instead of a plane, in order to minimize the ratio of the surrounding volume of atoms to the size of the potential map region. The first cutoff summation variant reads the atoms from the GPU constant memory, as done for direct summation. For each kernel call, only the atoms in that volume are transferred to the GPU, minimizing the amount of atom data repeatedly copied to the GPU.

The implementations of this variant are all restricted to operate exactly for lattice spacing $h = 0.5$Å and cutoff distance $r_c = 12$Å. Each kernel invocation operates on a block of 48^3 lattice points corresponding to a cubic volume 24Å on a side, to provide sufficient work for each invocation. Atoms are sorted using uniform grid spatial hashing on the CPU prior to potential summation. Computing the selected 24Å block of the potential map requires a cubic volume of atom data at least 48Å on a side, surrounding the block and extending the length of the 12Å cutoff distance in each direction. To satisfy this requirement, the bins for spatial hashing of atoms are cubes of size 24Å, offset from the lattice point blocks by 12Å in each dimension. Each block of lattice points is covered by a cube of 8 bins. The driver routine that calls the kernel first performs the spatial hashing of atoms and then loops over each block of lattice points, where it fills the constant memory on the GPU with atoms from as many of the surrounding bins as will fit before invoking the kernel. Depending on the number of atoms in each bin, it might require multiple kernel calls before all of the lattice point potentials have been fully summed.

A distance test is needed within the inner loop to check whether the current atom is within the cutoff distance to a thread's lattice point. The square of the distance is used, as it avoids a costly square-root operation in the common case where the atom is excluded from further computation. Within a warp, threads simultaneously checking distances to their respective lattice points may not all pass or fail the test, leading to branch divergence. Divergence is more likely for widely separated lattice points. We assign the threads of a warp a compact cluster of lattice points to minimize divergence. Each warp computes a cluster of $4 \times 4 \times 2$ lattice points at a time. The enhanced kernel unrolls the lattice points in the z-direction, allowing components of distance calculation to be reused within a thread in the same manner as direct summation. Limited register availability permitted unroll-and-jam by only three times.

4.3 Cutoff Summation with Small Bin Sizes

The previous cutoff summation algorithm suffers from two deficiencies. First, the distance test success rate is only about 6.5% measured as the ratio of the volume of the r_c-sphere to the enclosed cover of atoms. Failed distance tests represent computation that could have been avoided with better spatial partitioning. Second, the decomposition does not easily generalize to other cutoff distances and lattice spacings.

The small-bin cutoff summation variant uses a finer-grained spatial hashing and traversal scheme. In large-bin cutoff summation, all threads in a kernel traverse the same atom data; an atom needed by *any* thread will be inspected by *all* threads. This restriction, observed by both the direct and large-bin kernels, ensures that constant memory can fulfill the high memory access throughput demanded by the kernel. In small-bin cutoff summation, each thread block selects which bins it will traverse, allowing a thread block to traverse only the volume of atoms needed by its own threads (Figure 2(c)). Different GPU hardware features are exploited to provide high memory throughput. Additionally, the smaller volume is now sufficiently close to spherical to justify the extra complication involved in traversing a non-cubic volume.

Prior to GPU execution, the CPU performs spatial sorting and copies the entire set of atom data into global memory. This avoids the bottleneck imposed by constant memory's limited capacity on how much processing can occur in a single kernel call. Tolerating global memory's long ($>$ 200 cycles) latency is essential to kernel performance and is achieved through careful data layout and explicit sharing between threads. We use shared memory as a software-managed cache into which 128-byte, aligned blocks of global memory are fetched as needed. Using one warp to cooperatively load this size block ensures that the hardware retrieves it in two large-word reads from off-chip DRAM. The bin data structure is sized so that each bin is one block. Since an atom is stored as 4 single-precision floating point numbers, each 128-byte bin can hold 8 atoms. Leftover space within a bin is filled with dummy atoms having zero charge. Dummy atoms must be loaded to achieve coalescing, but can be skipped once loaded. When a bin reaches maximum capacity, any additional atoms intended for that bin are put into an overflow list that is processed on the CPU. As with the previous algorithm variant, each bin is associated with a fixed, cubic volume of space. This variant uses bins with edge length 4Å. At a typical density of 1 atom per 10Å^3, a bin's volume is expected to enclose 6.4 atoms. The average number of atoms stored in a bin is less because some atoms are put into the overflow list. For a 10^6 Å^3 cubic water box containing 96,603 atoms, bins contain an average of 5.35 atoms, and the overflow list holds 2.64% of all atoms. This distribution of atoms between the overflow list and the main data structure is favorable because overflow list summation on the CPU takes roughly the same time as GPU cutoff summation (Table 3), and its latency can be hidden by running the two steps in parallel.

Careful design of the bin data structure enables efficient use of global memory bandwidth, but the global memory latency is still exposed. To minimize time spent waiting for global loads, threads in a thread block cooperatively load and share atom data. Execution within a block proceeds in phases, alternating between loading bins from global memory into shared memory and summing electrostatic potentials. Bins needed by the thread block are loaded into shared memory once, and shared among threads. Multiple warps load bins in parallel, exploiting memory-level parallelism.

A thread block computes the potentials in a cubic *region* of 8^3 potential map points. For a given region, there is a *neighborhood* of bins that lie within or straddle the cutoff radius of some point in the region and thus need to be scanned to compute that region of the potential map. All threads in the thread block traverse all bins in the neighborhood, to maintain SIMD execution and avoid bank conflicts when accessing shared memory. A region's neighborhood is illustrated in Figure 3. The shape of the neighborhood is precomputed with respect to the binning lattice in the form of a list of neighbor offsets and stored in the GPU's constant memory. To determine its own neighborhood, a thread block computes the index of the bin closest to the center of its region, then translates the neighbor list offsets to that site.

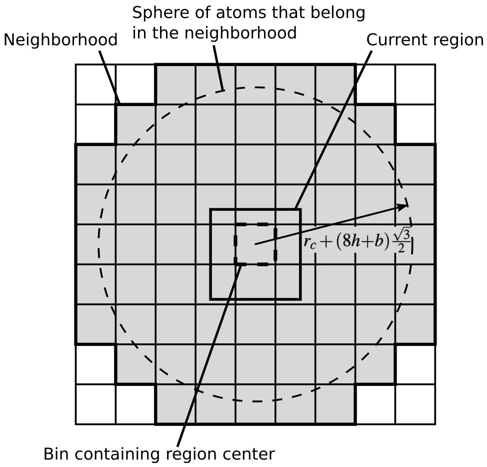
Sphere of atoms that belong in the neighborhood
Neighborhood
Current region
$r_c + (8h+b)\frac{\sqrt{3}}{2}$
Bin containing region center

Figure 3: Identification of the neighborhood of a region to be used in cutoff summation.

For a lattice spacing of h, a bin spacing of b, and a cutoff radius r_c, the neighborhood should contain all points within a radius of $r_c + 8h\sqrt{3}/2$, where the $8h\sqrt{3}/2$ term is the length from the center of a region to its corner and accounts for differing distances to different potential map points within a region. If the bin spacing is not an exact multiple of the lattice spacing, the distance is extended by $b\sqrt{3}/2$ to account for the worst-case misalignment between the region center and the closest bin. This distance is shown in Figure 3. The neighbor list contains all bins whose centers lie within a radius $R = r_c + (8h+b)\sqrt{3}/2$ if the bin spacing is an exact multiple of the lattice spacing, or $R = r_c + (8h+2b)\sqrt{3}/2$ otherwise. The extra distance included in R accounts for bins that straddle the cutoff radius. For the optimized case of lattice spacing $h = 0.5$Å and cutoff distance $r_c = 12$Å, the neighborhood list contains 335 bins. The success rate for cutoff distance testing increases to over 33%. Relative to the large-bin algorithm variant, five times as many of the atoms traversed contribute to the potential at a lattice point.

In the small-bin cutoff summation kernel variant, each SM performs the role that the entire GPU performed in the large-bin variant. Each thread block resembles a scaled-down version of the large-bin kernel. Specifically, atom data for the large-bin kernel was copied into constant memory, then scanned by all threads. Similarly, a thread block in the small-bin kernel loads atoms into shared memory, whereupon the data are scanned by all threads in the block. This algorithm variant's caching and sharing of data utilizes the GPU more like a multiprocessor than a stream processor, reflecting a degree of convergence between modern GPUs and multiprocessors.

4.4 Cutoff Summation on the CPU

To properly quantify the speedup of cutoff summation using GPU hardware, it is important to also develop the fastest possible sequential implementation to run on current CPU technology. We use SSE vector instructions generated by the Intel C Compiler. The CPU algorithm variant is also used with the small-bin kernel for overflow list summation.

Our fastest CPU implementation reverses the nested loop structure used in the GPU kernels. The kernel loops over atoms in the outer loop, and for each atom it loops over the cube of lattice points surrounding that atom. This loop nesting allows the CPU kernel to select precisely the potential map points to update for each atom, in contrast to the GPU kernels which must conservatively select a larger group of potential map points because they operate at the granularity of kernel execution or regions and bins. Performance is improved by initially sorting the atoms into bins (of size 4 Å) and traversing the ordered set of atoms to increase memory locality when updating the potential lattice points. The inner loop over the lattice points is a triply-nested loop over z-, y-, and x-coordinates to correspond to the row-major alignment of the potential lattice in memory. The separate terms of the distance-squared computation are factored out of the innermost loops (e.g., the $(\Delta z)^2$ term is factored out of the inner loops over y and x). We note that, unlike CUDA, the SSE vector instructions are much more restrictive; in particular, branching is not supported, so in order to achieve automatic vectorization by the Intel compiler, the inner loop must always compute the potential function and then perform a conditional selection at the end of the loop to accumulate to the lattice point either the computed value or zero. This is the CPU analogue of the cutoff distance test in the GPU kernels.

The SSE-enabled version of this basic algorithm was found to perform almost twice as fast as an SSE-enabled CPU port of the best GPU kernel. We clip the second-level loop to a cylinder (i.e., early termination if $(\Delta z)^2 + (\Delta y)^2 \geq {r_c}^2$), which improves performance by almost 20% over simply traversing a cube of lattice points. We found that over 67% of the clipped lattice points pass the cutoff distance test. We also tested a sphere clipping enhancement, which adjusts the bounds of the innermost loop to avoid any unnecessary computation and eliminates the cutoff test, but this was found to reduce performance. The cylinder clipping enhancement is also applicable to the GPU kernels that implement the unroll-and-jam optimization, resulting in about a 6.7% performance increase. This enhancement is included in the GPU kernels except for the large-bin kernel, to keep its performance comparable with previously published results that do not use the enhancement.

5. EXPERIMENTAL EVALUATION

We have evaluated the performance of several cutoff pair potential kernels formulated for the computation of electrostatic potential maps. The CPU-SSE3 reference kernel computes potentials entirely on the CPU and is representative of the single-threaded performance achievable on a workstation without GPU acceleration. The CPU kernel is vectorized and uses SSE SIMD instructions to achieve the best performance possible short of coding in assembly language. The key CPU code optimizations involve expressing the inner loop without any loop carried dependencies and using arithmetic operations and control logic that can be mapped directly to x86 SSE SIMD instructions.

The LargeBin kernel follows the algorithm and implementation described in Sec. 4.2 and is also the closest in design and performance to a short-range multilevel Coulomb summation kernel described previously [28]. The SmallBin kernel implements the binned short-range cutoff algorithm described in Sec. 4.3, with operations on the CPU and GPU occurring serially, one after the other in separate phases of computation. In this kernel, the CPU simply busy-waits while the GPU executes and vice-versa. The SmallBin-Overlap kernel implements the binned short-range cutoff algorithm described in Sec. 4.3, but uses the streaming API provided by CUDA 1.1 to overlap CPU and GPU computations. Once launched, the GPU kernel proceeds executing asynchronously, allowing the CPU to perform bin overflow handling concurrently. When the CPU completes its bin overflow handling, it waits, synchronizing with the GPU kernel. Once the stream synchronization completes, the CPU performs the remaining I/O operations to bring results back from the GPU, and sums them with the bin overflow handling results, yielding the final potential values for the lattice region being computed. The SmallBin-CompSum kernel is a minor variation of the SmallBin-Overlap that uses single-level compensated summation for accumulation of potentials at lattice points, rather than the native floating point addition operation.

All tests were run on a quiescent system 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 4. The CPU benchmarks were performed using a single core, a best case scenario in terms of memory bandwidth. The CPU code was compiled using the Intel C/C++ Compiler (ICC) version 9.0. All GPU benchmarks were performed on a NVIDIA Tesla C870 GPU, and were compiled and executed using the CUDA development toolkit version 1.1 with driver version 169.09.

5.1 Floating-Point Arithmetic

One limitation of the most commonly available GPU hardware is that floating point operations can only be performed in single precision. While this poses a problem for some algorithms, many produce results with acceptable precision or can be adapted to do so through the use of compensated summation, native pair arithmetic, or other precision enhancement techniques [3, 10].

We compared the potential lattice values calculated by the CPU and GPU kernels with those produced by a double precision CPU implementation to determine the largest absolute and relative error for each, for lattice points with potential magnitudes larger than 10^{-4}, for a 10^6 Å^3 cubic water box containing 96,603 atoms. Since the shifted potential function smoothly drops to zero at the cutoff distance, the effects of rounding on selection of atoms at the cutoff radius were minimized. The relative and absolute error test results summarized in Table 2 show that the worst case error encountered among the selected lattice points for the GPU potential kernels is within a factor of two of the single-precision CPU-based kernel.

Due to its limited precision representation, floating point arithmetic is non-associative, yielding different results when values are summed in different orders. The CPU kernels were based on the same algorithm, and performed summation in identical order, an ideal scenario for error comparison. The sources of GPU floating point error relative to the reference CPU implementation are the result of entirely different summation order, minor differences in the precision of basic arithmetic operations, and arithmetic expression transformations resulting from compiler optimizations.

The SmallBin-CompSum GPU kernel in Table 2 uses single-level compensated summation for accumulation of the electrostatic potential at each lattice point. Its maximum relative error is 34% lower than that of the SmallBin kernel. Compensated summation eliminates much of the error attributable to differences in summation ordering in the CPU and GPU kernels; in some test cases, the maximum relative error of the SmallBin-CompSum kernel actually falls below that of the CPU-SSE3 kernel. The performance cost associated with performing compensated summation in place of native floating point summation operations is shown in Table 4, and is always minor, at under 10% for the range of problem sizes presented here. This makes compensated summation a very attractive technique in cases where greater precision may be desired.

Kernel	Max % relative error	Max % absolute error
CPU-SSE3	0.4793	0.0000939
LargeBin	0.7457	0.0001561
SmallBin	0.8715	0.0001985
SmallBin-CompSum	0.5710	0.0001579

Table 2: Maximum error values for single precision CPU and GPU kernels vs. a double-precision version of the CPU reference kernel.

Component	CPU-SSE3	SmallBin
Spatial hashing	0.008 s	0.008 s
Cutoff summation	19.444 s	0.842 s
I/O between host and GPU		0.052 s
Overflow list summation		0.586 s
Potential map transpose		0.030 s
Total execution time	19.452 s	1.518 s

Table 3: Execution time of components of cutoff potential calculation.

5.2 Performance and Scalability

Table 3 breaks down the components of execution time in the SmallBin GPU kernel, which does not overlap GPU and CPU computations. Only cutoff summation in the SmallBin kernel runs on the GPU. Overflow list summation uses the CPU-SSE3 kernel. The transpose step converts from the data layout used for the potential map on the GPU to the layout used on the CPU.

For this test, a 10^6 Å^3 cubic water box containing 96,603 atoms was used. The water box was created using the solvate plugin included with VMD [12], with a volume and atom density representative of typical molecular dynamics simulations of biomolecular complexes. One of the key observations from the execution time profile in Table 3, is that the CPU time spent handling overflowed bins is similar in magnitude to the GPU time spent calculating the bulk of the cutoff summation. This observation implies the potential for increasing performance by performing the CPU and GPU computations concurrently. The performance results in Table 4 more clearly demonstrate the benefit of overlapping the CPU and GPU computations.

The performance scaling behavior of each of the kernels was evaluated for water-filled cubic volumes ranging in size from 10^3 Å^3 to over 10^7 Å^3. The water boxes were generated with the same methodology described previously. The computational complexity of the cutoff potential algorithms implemented by all of the kernels scales linearly with the the number of atoms.

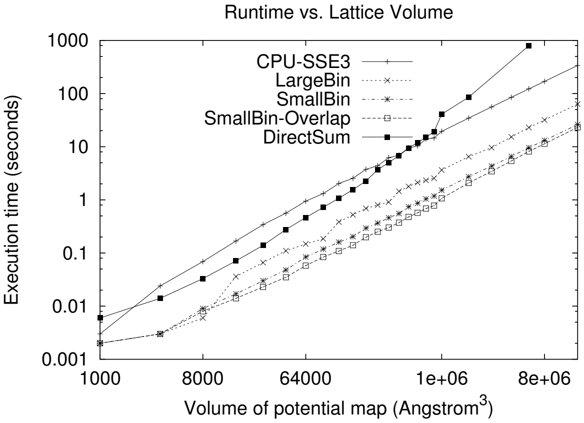

Figure 4: Cutoff kernel execution time.

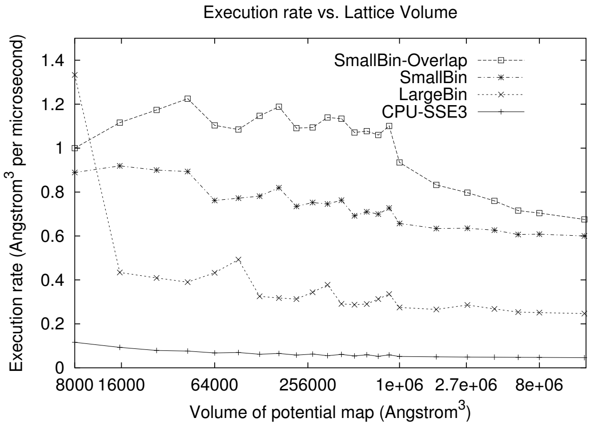

Figure 5: Cutoff kernel execution rate.

Kernel	Runtime (s)	Speedup
CPU-SSE3	19.452	1.0
LargeBin	3.64	5.3
SmallBin	1.51	12.81
SmallBin-Overlap	1.07	18.17
SmallBin-CompSum	1.15	16.91

Table 4: Comparison of performance for short-range cutoff potential kernels tested with a 10^6 Å^3 water box containing 96,603 atoms.

The DirectSum kernel is a GPU implementation of the direct Coulomb summation algorithm reported previously [28], which provides important context for the performance of the cutoff kernels. The direct summation kernel has quadratic time complexity, which is clearly visible in the slope of its execution time plotted in Figure 4 relative to the short-range cutoff kernels. While the results of the cutoff kernels do not include long-range potential contributions, the execution time of algorithms for computing these contributions would merely offset the cutoff kernels, shifting the crossover point where the CPU-SSE3 kernel begins outperforming the DirectSum kernel a little farther to the right.

The performance scaling results in Figure 6 show that the GPU cutoff kernels maintain a significant performance advantage over the CPU for all but the smallest test cases. Once the problem size is large enough to fully utilize the GPU and to amortize the GPU kernel invocation overhead, the GPU kernels begin greatly outperforming the CPU kernel. The CPU execution time exhibits its steepest ascending slope in the smallest problem size ranges and improves to linear performance with the larger problem sizes. As the problem size increases by orders of magnitude, the relative performance advantage held by the GPU small-bin kernels declines slightly due to global memory references being serviced less efficiently. This effect is most likely a result of the increasing rate of DRAM page boundary crossings with problem size. The performance plots for the two GPU small-bin kernels exhibit a gradual decrease in the effectiveness of overlapping of CPU and GPU calculations as the GPU runtime component becomes dominant for problem size increases. At small bin sizes, much of the volume is

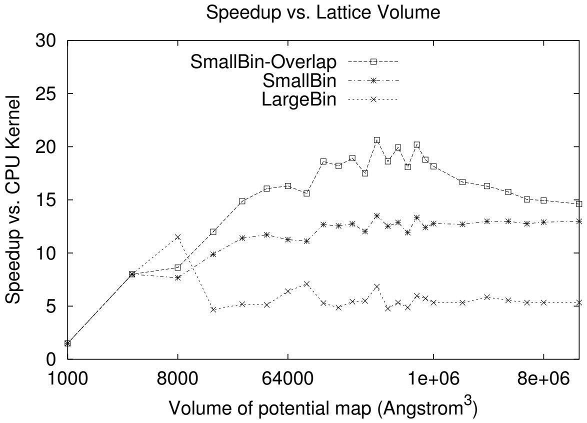

Figure 6: GPU cutoff kernel speedup (relative to CPU implementation).

located in the vicinity of the empty space outside the water box. The empty space is processed quickly and contributes to faster execution. This contribution decreases for larger volumes.

Although none of the tests described in this paper were performed using multiple CPU cores or multiple GPUs, our past experience with multi-core and multi-GPU implementations of direct summation kernels suggests that a multi-core CPU-SSE kernel would likely saturate available main memory and shared cache bandwidth with just a few cores, yielding sub-linear performance scaling beyond that point. Conversely, we expect that a multi-GPU implementation would parallelize very effectively, achieving near-linear speedups.

5.3 Memory System Performance

We measured the memory behavior of the SmallBin GPU kernel to check whether it uses the C870's memory system efficiently. Since NVIDIA's driver does not currently provide a way to directly measure time spent in memory accesses, we measure the performance indirectly using a modified kernel and artificially distributed data sets. In the modified kernel, the cutoff radius is set to a large value so that the kernel does not exclude any atoms in the neighborhood from potential calculation. This makes the kernel's runtime independent of the atoms' positions. While its output is not numerically valid, the modified kernel has the same memory behavior as the SmallBin kernel. The neighborhood is computed using a cutoff radius of 12Å, as for the normal kernel. These tests used a 10^6 Å^3 volume of atom data in which each bin contains the same number of atoms; bin occupancy was varied from 0 to 8. The amount of data loaded from global memory depends only on the number of bins in the neighborhood, while the amount of computation to process a bin is linearly proportional to the number of atoms per bin.

The kernel execution time varies linearly with the number of atoms per bin as shown in Figure 7. The residual execution time when bins are empty is the time spent doing work other than electrostatic potential calculation. Average global memory bandwidth consumption can be computed from the execution time and the quantity of loaded data (26^3 regions $\times$ 335 bins $\times$ 128 B/bin). Bandwidth consumption varies from 0.44 GB/s when bins are full of atoms to 4.1 GB/s when bins are empty. Even for empty bins, when the memory bandwidth demand is highest, execution time follows the linear relationship. This is where one would expect to see slowdown from any memory bandwidth limits.

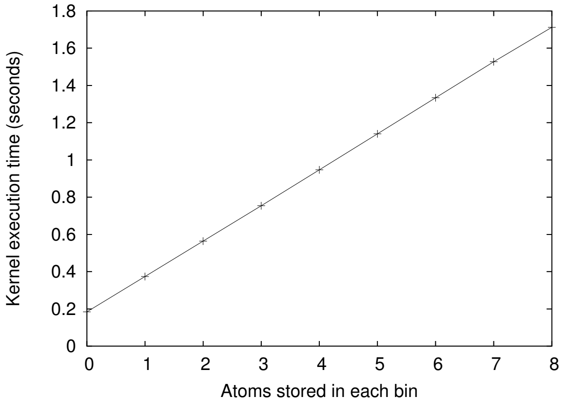

Figure 7: Execution time of a modified cutoff potential kernel in memory performance tests.

We also eliminated all global memory loads from the kernel to estimate the time consumed by long-latency load instructions. Each load was replaced by arithmetic instructions that fabricate a small positive floating-point number out of the address. Since the generated values are nonzero, the kernel computes as if it were operating on full bins. The load-free kernel's execution time is 1.709 seconds, very close to the 1.712 seconds spent to load and process full bins. We conclude that the time spent waiting for long-latency memory accesses is negligible. Any further performance gains would have to come from reducing either the number of bins traversed or the computation time to process a bin once loaded.

6. CONCLUSIONS

We have presented a new GPU-accelerated algorithm for calculating cutoff pair potentials. The new algorithm demonstrates significant speedup relative to a highly efficient SSE-vectorized CPU reference implementation and provides a factor of three performance improvement relative to our best prior results on the same GPU hardware. The spatial decomposition of work inherent in the algorithm maps naturally to CUDA thread blocks while the spatial decomposition of data achieves fast and efficent use of the memory system. The numerical accuracy of the GPU's single-precision floating point hardware can be increased to levels on par with single-precision CPU code at low computational cost through the use of compensated summation.

The new GPU-accelerated cutoff pair potential algorithm can be applied to accelerate fast methods, such as multilevel summation, for approximating a full electrostatic potential map. We plan to incorporate these advances into a future version of VMD [12], bringing a significant performance increase to scientific tasks that require calculation of electrostatic potential maps such as ion placement and calculation of mean field potential maps for molecular dynamics simulations.

Acknowledgments

This work was supported by the National Institutes of Health, under grant P41-RR05969. This work was also supported by the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. Performance experiments were made possible by a generous hardware donation by NVIDIA, NSF CNS grant 05-51665, and the National Center for Supercomputing Applications.

7. REFERENCES

[1] Advanced Micro Devices, Inc. *ATI CTM Guide, version 1.01*, 2006.

[2] T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, and K. Chihara. Particle-based fluid simulation on GPU. In *ACM Workshop on General-Purpose Computing on Graphics Processors*, 2004.

[3] D. H. Bailey. High-precision floating-point arithmetic in scientific computation. *Computing in Science and Engineering*, 07(3):54–61, 2005.

[4] I. Buck. Case studies: Ray tracing and molecular dynamics. In *IEEE Visualization 2004 GPGPU Tutorial*. IEEE Computer Society, 2004.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[6] T. A. Darden, D. M. York, and L. G. Pedersen. Particle mesh Ewald. An N·log(N) method for Ewald sums in large systems. *Journal of Chemical Physics*, 98:10089–10092, 1993.

[7] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V. Pande. N-body simulation on GPUs. In *SC06 Proceedings*. IEEE Computer Society, 2006.

[8] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann series in interactive 3D technology. Morgan Kaufman, San Francisco, CA, 2005.

[9] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on GPUs. In *Proceedings of the 25th Computer Graphics International Conference*, May 2007.

[10] Y. He and C. H. Q. Ding. Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 225–234, New York, NY, USA, 2000. ACM Press.

[11] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. McGraw-Hill, New York, 1981.

[12] W. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics*, 14:33–38, 1996.

[13] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[14] K. Kennedy and R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based approach*. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[15] A. Kolb and N. Cuntz. Dynamic particle coupling for GPU-based fluid simulation. In *Proceedings of the 18th Symposium on Simulation Technique*, pages 722–727, September 2005.

[16] J. S. Meredith, S. R. Alam, and J. S. Vetter. Analysis of a computational biology simulation technique on emerging processing architectures. In *Sixth IEEE International Workshop on High Performance Computational Biology*, 2007.

[17] J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. In *Microprocessor Forum*, May 2007.

[18] NVIDIA Corporation. *NVIDIA CUDA Programming Guide, version 1.1*, 2007.

[19] L. Nyland, M. Harris, and J. Prins. Fast n-body simulation with CUDA. *GPU Gems 3*, pages 677–695, 2007.

[20] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5), May 2008.

[21] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50, 2003.

[22] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008.

[23] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Optimization space pruning for a multithreaded GPU. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, April 2008.

[24] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. In *Proceedings of the 34th International Symposium on Computer Architecture*, June 2007.

[25] G. Shi and V. Kindratenko. Implementation of NAMD molecular dynamics non-bonded force-field on the Cell Broadband Engine processor. In *Proceedings of the 9th International Workshop on Parallel and Distributed Scientific and Engineering Computing*, April 2008.

[26] R. D. Skeel, I. Tezcan, and D. J. Hardy. Multiple grid methods for classical molecular dynamics. *Journal of Computational Chemistry*, 23:673–684, 2002.

[27] C. D. Snow, H. Nguyen, V. S. Pande, and M. Gruebele. Absolute comparison of simulated and experimental protein-folding dynamics. *Nature*, 420:102–106, 2002.

[28] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, December 2007.

[29] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya. Protein explorer: A petaflops special-purpose computer system for molecular dynamics simulations. In *Proceedings of the ACM/IEEE SC2003 Conference*, November 2003.

[30] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 325–335, 2006.