Journal of Molecular Graphics and Modelling 29 (2010) 116–125

Contents lists available at ScienceDirect

Journal of Molecular Graphics and Modelling

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

Topical perspectives

GPU-accelerated molecular modeling coming of age

John E. Stone [a], David J. Hardy [a], Ivan S. Ufimtsev [b], Klaus Schulten [c,*]

[a] Beckman Institute, University of Illinois at Urbana-Champaign, 405 N. Mathews Ave., Urbana, IL 61801, United States
[b] Department of Chemistry, Stanford University, 333 Campus Drive, Stanford, CA 94305, United States
[c] Department of Physics, University of Illinois at Urbana-Champaign, 1110 W. Green, Urbana, IL 61801, United States

ARTICLE INFO

Article history:
Received 9 February 2010
Received in revised form 24 June 2010
Accepted 30 June 2010
Available online 8 July 2010

Keywords:
GPU computing
Molecular modeling
Molecular dynamics
Quantum chemistry
Molecular graphics

ABSTRACT

Graphics processing units (GPUs) have traditionally been used in molecular modeling solely for visualization of molecular structures and animation of trajectories resulting from molecular dynamics simulations. Modern GPUs have evolved into fully programmable, massively parallel co-processors that can now be exploited to accelerate many scientific computations, typically providing about one order of magnitude speedup over CPU code and in special cases providing speedups of two orders of magnitude. This paper surveys the development of molecular modeling algorithms that leverage GPU computing, the advances already made and remaining issues to be resolved, and the continuing evolution of GPU technology that promises to become even more useful to molecular modeling. Hardware acceleration with commodity GPUs is expected to benefit the overall computational biology community by bringing teraflops performance to desktop workstations and in some cases potentially changing what were formerly batch-mode computational jobs into interactive tasks.

1. Introduction

Graphics processing units (GPUs) have recently evolved into sophisticated co-processors for throughput-oriented, data-parallel computational workloads [1–3]. As molecular modelers study progressively larger biomolecular complexes and the dynamics of cellular processes occurring on longer timescales, the computational demands continue to grow. Seeking to satisfy demand for computing, the molecular modeling community was among the earliest adopters of GPU computing [4–10]. Early attempts to devise molecular modeling algorithms that harness the power of GPUs have been largely successful, in some cases providing performance levels two orders of magnitude faster than that achieved with a CPU alone. Performance increases of this magnitude have the potential to revolutionize computational biology today by changing what were formerly batch-mode computational jobs into interactive tasks. While a survey of these successes is of interest to general practitioners looking for present and future computing solutions, the rapid adoption of GPUs in molecular modeling offers, in particular, guidance for other areas of computational biology.

Prior to the turn of the millennium, continuing advances in microprocessor design and semiconductor manufacturing provided an exponential performance growth curve fueling the capabilities of molecular modeling applications. During this period, parallel computing techniques were typically employed only for solving the most computationally demanding tasks, as most application users were satisfied to wait for the expected steady performance increases using existing sequential molecular modeling codes. Beginning around the year 2002, practical limitations on the power consumption and heat dissipation resulting from increasing microprocessor clock rates caused the performance growth curve for single-core microprocessors to fall flat. This left the molecular modeling community with no alternative but to begin development of parallel and multi-core versions of applications that require performance levels exceeding the capabilities of single-core CPUs. This situation has been further exacerbated by continued advances in experimental structure determination techniques that have yielded an overabundance of ever more challenging molecular structure data to be studied.

There has been a great deal of interest in the use of accelerator devices to augment multi-core CPUs for computationally demanding molecular modeling applications as a result of the relentless drive for increased performance, improved price/performance, and greater performance/watt efficiency. Previous efforts with accelerators, including field-programmable gate arrays (FPGAs) [11,12], several generations of MDGRAPE [13–15], the Cell processor [9,16–20], and GPUs, have demonstrated the potential performance benefits available to molecular modeling, while bringing to light many of the software engineering challenges involved in adapting existing applications for heterogeneous computing. Although the

* Corresponding author. Tel.: +1 217 244 2212.
E-mail addresses: johns@ks.uiuc.edu (J.E. Stone), dhardy@ks.uiuc.edu (D.J. Hardy), ufimtsev@stanford.edu (I.S. Ufimtsev), kschulte@ks.uiuc.edu (K. Schulten).

1093-3263/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jmgm.2010.06.010

specific details of programming GPUs differ somewhat from other accelerators, all heterogeneous computing approaches involve porting or adapting existing algorithms for the target accelerator device, managing multiple independent memory spaces, balancing work among host CPUs and accelerator devices, and coping with host-device communication latency and bandwidth limitations. In this sense, developers of GPU-accelerated molecular modeling applications have benefited from the techniques and experiences gained on the other accelerator platforms.

Owing to architectural features resulting from their graphics lineage, GPUs are well suited to arithmetic-intensive computational workloads that are highly data-parallel. State-of-the-art GPUs achieve aggregate floating point performance levels that exceed that of contemporary CPUs by up to a factor of ten, reaching over one trillion single-precision floating point operations per second. GPUs contain high-bandwidth memory systems as a result of the needs of graphics, capable of intra-GPU transfer rates of over one hundred gigabytes per second. GPUs implement hardware multithreading and machine instructions for many of the mathematical functions used by molecular modeling applications. In combination, these attributes enable GPUs to outperform traditional CPU cores on highly data-parallel workloads by factors ranging from ten to twenty times faster in the majority of cases, up to as much as one hundred times faster in a few ideal cases [3,21–23].

Previously, the applications of parallel processing techniques to molecular modeling have remained largely confined to batch-mode simulation workloads such as molecular dynamics and quantum chemistry simulation. The use of GPUs as data-parallel co-processors has provided a unique opportunity to tremendously increase the effective computational capability of desktop workstations and laptop computers and the applications that run on them, without requiring end users to develop expertise in managing clusters or using remote supercomputers. GPUs can also be used to provide significant performance increases for latency-sensitive interactive visualization and analysis tasks that are poorly suited to traditional HPC clusters, in some cases giving rise to visualization and analysis capabilities not possible with conventional techniques.

2. GPU overview

Early efforts in the use of GPUs for non-graphical computations were based on graphics-specific programmable shading languages, requiring scientific calculations to be expressed in terms of graphics drawing operations. Adding to the difficulty of using such a convoluted approach, early programmable GPU devices did not fully support standard IEEE floating point data types, had limited support for data-dependent branching, and placed limits on program size and complexity. Despite these challenges, the implementation of a variety of proof-of-concept algorithms demonstrated the feasibility of using GPUs for non-graphical computations and encouraged GPU architects to target general purpose computation in subsequent hardware generations, ultimately leading to better programming models for GPU computing [1,24–27].

2.1. GPU hardware architecture

In their primary role as engines for high performance computer graphics, GPUs rasterize complex images from streams of graphics commands and geometric data. The most computationally demanding parts of the rasterization and shading process are inherently data-parallel, leading hardware architects to design GPUs as massively parallel throughput-oriented devices. As graphics algorithms have evolved and applications have begun to employ increasingly sophisticated shading techniques, GPU designs have shifted away from task-specific fixed-function graphics logic toward large arrays of programmable processor cores [1].

Continued architectural refinement has increased both the performance and generality of GPUs, enabling them to operate on all standard integer and floating point data types supported by contemporary CPUs. These improvements have made GPUs ideal devices for arithmetic-intensive scientific and engineering applications containing significant parallelism [2]. Current GPUs contain hundreds of independent execution units, organized into groups forming tightly-coupled processor complexes that share fast on-chip memory systems, specialized graphics “texture” hardware for fast interpolation of multi-dimensional arrays, and high bandwidth channels to large off-chip global memory. Recent GPUs have improved peak host-GPU memory transfer performance and latency to the point that they are limited mainly by the PCI-Express bus itself and by driver software overhead. Current GPUs also support asynchronous host-GPU memory transfers and allow complete overlapping of communication with computations.

The individual processing elements that compose a GPU are grouped into clusters, each following a single-instruction multiple-data (SIMD) organization, whereby all of the processing elements execute the same instruction in lock-step, but on different data. The group of threads executing in lock-step on the SIMD processing elements are collectively known as a “warp” or a “wavefront.” The mapping of computational work items to threads in the same warp can have a significant impact on performance. The evaluation of a conditional branch among threads in the same warp can lead to a bifurcation of the code path, known as “branch divergence,” which is handled by executing both code paths in sequence, masking the execution of the individual threads not participating in one of the two paths of execution until they both are again unified among all threads in the warp. Branch divergence is avoided and performance improved whenever all threads in the warp evaluate to the same condition, in which case just one of the two code paths is executed. While early GPUs were limited to stream-oriented computations, contemporary GPUs support arbitrary memory access patterns, albeit with performance characteristics that vary according to the details of the access pattern. Peak performance for accesses to the large off-chip global memory is attained only when the access pattern yields so-called “coalesced” memory transfers, where all memory banks are utilized in parallel for a single transfer operation with no conflicts among threads in the same warp.

An excellent overview of NVIDIA’s “Tesla” compute-oriented GPU architecture has been provided by Lindholm et al. [28]. Fig. 1 illustrates the hardware organization of the state-of-the-art NVIDIA “Fermi” GPU architecture [29]. Some of the main features added in the Fermi architecture are greatly increased double-precision floating point arithmetic performance, an enlarged 64 kB on-chip shared memory and L1 cache capacity, a large 768 kB L2 cache for main memory, and support for complete overlapping of asynchronous bidirectional host-GPU memory transfers and GPU kernel execution. The Fermi GPU architecture also enables reliable execution of long-running computations through the use of register files, caches, and memory systems protected by error correcting codes (ECC) that can correct single-bit errors and detect two-bit errors.

The particular combination of hardware features provided by GPUs is aptly suited to the needs of many molecular modeling applications. Molecular modeling algorithms typically contain fine-grained data parallelism resulting from the atomic interactions they compute. Existing CPU algorithms must often be reformulated in order to expose this data parallelism and to effectively exploit the throughput-oriented architecture of GPUs. Due to the heavy use of square roots, exponentials, and other transcendental functions in computer graphics algorithms, GPUs provide machine instructions for these functions that greatly outperform their CPU equivalents. Molecular modeling applications that manage to take advantage of

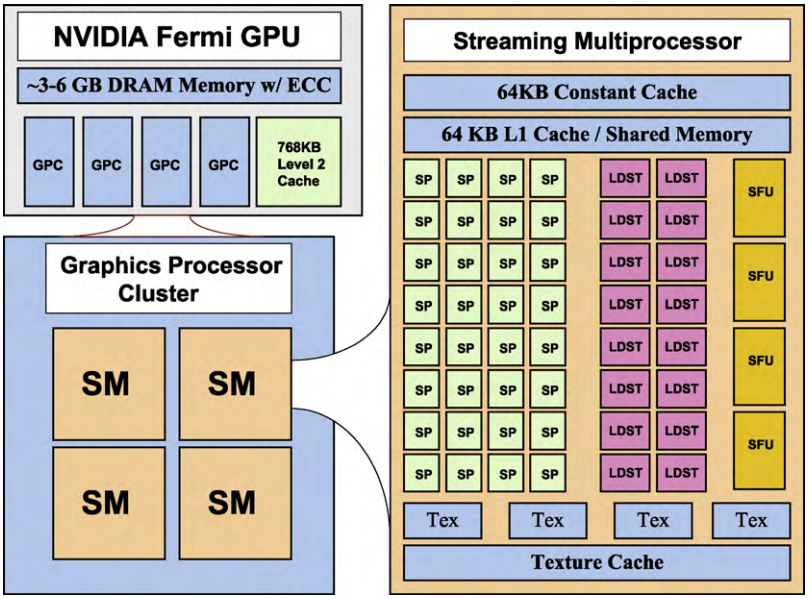

Fig. 1. A simplified hardware block diagram for the NVIDIA "Fermi" GPU architecture. Fermi contains up to 512 general purpose arithmetic units known as "streaming processors" (SP) and 64 "special function units" (SFU) for computing special transcendental and algebraic functions not provided by the SPs. Memory load/store units (LDST), texture units (TEX), fast on-chip data caches, and a high-bandwidth main memory system provide the GPU with sufficient operand bandwidth to keep the arithmetic units productive. Groups of 32 SPs, 16 LDSTs, 4 SFUs, and 4 TEXs compose a "streaming multiprocessor" (SM). One or more CUDA "thread blocks" execute concurrently on each SM, with each block containing 64–512 threads.

several of the GPU hardware features at once have been shown to achieve performance levels of up to one hundred times that of a single CPU core [21,22]. It is not surprising that applications that closely match the characteristics of graphics workloads achieve such speedups, since GPUs provide even higher effective speedups for the graphics applications that they are primarily designed to accelerate.

2.2. GPU programming models and abstractions

Early data-parallel GPU computing languages, such as Brook [30] and Sh [31], wrapped existing graphics-specific shading languages with more broadly applicable languages and programming abstractions, leading to early GPU computing successes. Though a big step in the right direction, these efforts were still hindered by limitations of the GPU hardware available at the time. The introduction of hardware support for standard data types, branching instructions, and greatly increased memory capacity has allowed development of GPU-specific dialects of popular computer languages such as C and Fortran. In 2007, NVIDIA released the "CUDA" GPU programming toolkit, providing a dialect of C with extensions for data-parallel programming targeting NVIDIA GPUs [32]. The CUDA toolkit enables execution of a C function, or "kernel," on a target GPU device. CUDA was used as the basis for several early successes in GPU-accelerated molecular modeling [5,33]. Since then, CUDA has gained a tremendous following as the first GPU programming toolkit to achieve significant traction among developers of science and engineering applications. An excellent introduction to CUDA and data-parallel GPU programming concepts has been provided by Nickolls et al. [34].

In 2008, an industry consortium released a specification for OpenCL, the first broadly supported multi-platform data-parallel programming interface for heterogeneous computing on GPUs and similar devices [35]. Like CUDA, OpenCL is also based on a dialect of C. One of the distinguishing features of OpenCL is that it also specifically targets multi-core CPUs. With careful design, it is possible to construct OpenCL kernels that perform at moderate efficiency on CPUs, GPUs, and other accelerators. Although OpenCL provides portability and correctness guarantees across architectures, different architectures have varying warp or wavefront sizes and memory coalescing requirements, necessitating the use of platform or accelerator-dependent optimizations for peak performance [36]. OpenCL has the potential to displace vendor-specific CPU vector arithmetic intrinsics, reducing the number of unique code paths that must be maintained in scientific applications while broadening the range of supported hardware. Optimized OpenCL kernels are generally much more readable than routines containing, for example, heavy use of Intel x86 SSE vector intrinsics. As the specification matures and receives better support from vendor implementations, OpenCL is expected to become more popular in the future.

Although GPUs are nearly complete computers in their own right, they must be managed by application software running on a host computer. An application program on the host uses attached GPUs through language extensions such as "Jacket" for Matlab[1] or PyCUDA[2] for Python, by calling libraries containing GPU-accelerated BLAS or FFT subroutines [3,37], or by invoking custom-written CUDA or OpenCL kernels. Language extensions and subroutine libraries are the easiest way for an application to take advantage of GPUs. Custom-written GPU kernels typically provide the greatest performance by minimizing sources of overhead and matching GPU data structures and kernel parameters as closely as possible to the needs of the application. The key programming interfaces provided by GPU programming toolkits, such as CUDA and OpenCL, include mechanisms for enumerating and managing available GPUs, managing GPU memory allocations, transferring data between the host machine and attached GPUs, launching calculations, querying execution progress, and checking for errors.

2.3. GPU algorithm design

Custom-written GPU kernels can often outperform standardized or generically written GPU subroutine libraries, but achieving high

[1] http://www.accelereyes.com/.
[2] http://pypi.python.org/pypi/pycuda.

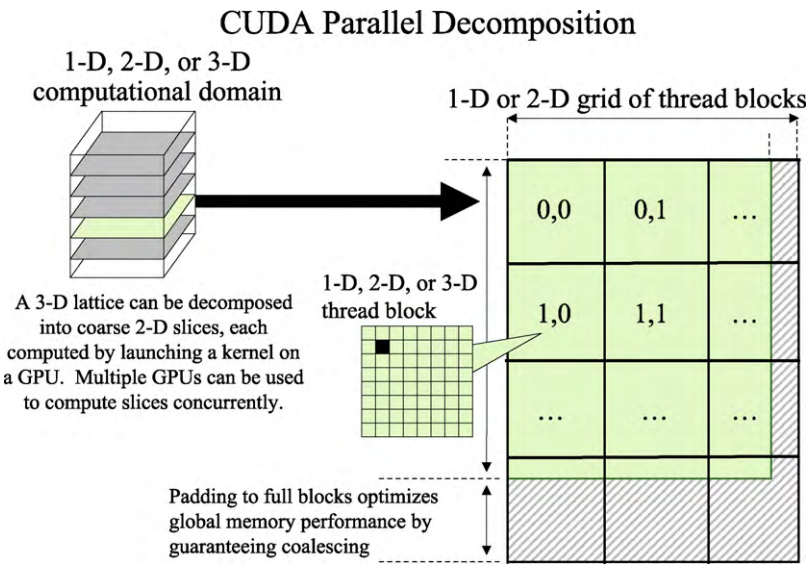

Fig. 2. In CUDA, a data-parallel problem is decomposed into independent work items called "threads." Threads are executed together in cooperative work groups called "thread blocks," which are grouped into "grids." The computation for a CUDA grid corresponds to exactly one GPU kernel invocation. OpenCL uses a very similar scheme with essentially the same semantics, but in OpenCL these groupings are named "work items," "work groups," and "index ranges."

performance requires careful attention to the use of GPU hardware features and appropriate use of software interfaces.

Due to their massively parallel and multithreaded hardware architecture, GPUs perform best with fine-grained parallel decompositions. Contemporary GPUs require a data parallel workload on the order of 10,000–30,000 independent threads in order to fully occupy the device and provide enough work to hide various sources of latency. Fortunately, the fine-grained parallel decomposition required by GPUs is often complementary to the coarse-grained decompositions typically employed for multi-core CPUs and clustering. It is often relatively easy to compose GPU algorithms within existing applications targeting multi-core CPUs or HPC clusters by adding an additional level of parallel decomposition for the GPU. The work that would have previously been performed by one or more CPU cores on a single cluster node can often be further decomposed into thousands of even smaller individual computations suitable for the GPU.

Fig. 2 illustrates a typical parallel decomposition strategy shown for a three-dimensional computational domain such as an electrostatic potential map. The computational domain is first decomposed into two-dimensional slices, which are then broken up into thousands of independent threads and organized into a grid of thread blocks. It is usually beneficial to pad data structures and throw away unneeded results, rather than attempting to avoid doing unnecessary computations. Padding the data also helps to maintain memory alignment requirements for high-bandwidth coalesced memory transfers [34].

A modern GPU can achieve over 1000 GFLOPS peak performance for single-precision arithmetic, but has peak global memory bandwidth on the order of 100 gigabytes per second. This leaves roughly a factor of 40 gap between the rate at which arithmetic units consume single-precision operands and the peak bandwidth to GPU global memory. In order to achieve peak arithmetic performance, GPU algorithms must reduce or eliminate this performance gap through the use of registers and fast on-chip GPU memory subsystems, effectively amplifying overall operand bandwidth. Since GPU registers run at the same rate as the arithmetic units, they are the first choice for storage of data and temporary values that are private to individual GPU threads. For read-only data, a small on-chip constant cache can provide near register-speed performance when all threads in a warp read the same constant data location. For data that are too large to be held in thread-private registers or that must be shared or communicated among multiple threads, a small on-chip shared memory or parallel data cache provides roughly 50% of the throughput of registers. Good GPU performance necessitates coalesced global memory access, achieved through use of the proper memory block size, alignment, and thread access pattern over each memory block. The effective bandwidth can be amplified by using the shared memory as a program-managed cache, with the required bookkeeping minimized by accessing data blocks consecutively. Recent GPUs such as the NVIDIA Fermi also provide hardware-managed caching of global memory accesses in the on-chip shared memory or parallel data cache.

A problem that frequently arises when adapting serial or coarse-grained parallel algorithms to GPUs is the occurrence of output conflicts from multiple threads, where uncoordinated concurrent memory writes from multiple threads will likely produce unpredictable results. In serial or coarse-grained parallel algorithms for CPUs, this problem is often avoided by having independent threads write to independent intermediate output buffers, followed by a reduction that combines the independent buffers into a final result. It is impractical to use this approach on GPUs since efficient execution requires over 10,000 threads to be running concurrently, far too many for completely duplicated output buffers to be practical. Although modern GPUs support atomic updates to global memory, this access is available with comparatively low performance, slower than a standard global write and slower, still, if there is contention between threads.

Performance and correctness issues arising from conflicting memory writes are most efficiently handled by reformulating algorithms to avoid them. Output conflicts are eliminated by recasting *scatter* memory access patterns, in which a single thread writes with conflicts to a number of global memory locations, into *gather* memory access patterns, in which a single thread reads without conflicts from a number of global memory locations. The cutoff summation algorithm discussed in the following section provides a nice example of how a reorganization of the data combined with interchanging the nesting of loops converts the typical scatter-based serial approach into a gather-based approach that is well suited to GPU computation. A GPU algorithm might also benefit from trading floating point operations for memory accesses. An example is to double the number of non-bonded force evaluations,

rather than taking advantage of Newton's Third Law, in order to convert a gather-scatter memory access pattern of a typical serial implementation into a simple gather.

3. GPU-accelerated applications

With the release of CUDA [32] and OpenCL [35], GPU programming has become significantly more accessible, and computational scientists no longer need to have an extensive background in computer graphics to harness the computational capabilities of GPUs. In the past 3 years, tremendous progress has been made in the development of GPU implementations of fundamental algorithms and widely used subroutine libraries, providing the foundation for broader adoption within computationally demanding scientific applications. In this section, brief summaries are presented of some recently published work on the utilization of GPU acceleration techniques for several areas of molecular modeling.

3.1. Electrostatics

One of the early applications of GPU computing was the calculation of three-dimensional electrostatic potential maps [5]. Since the electrostatic environment surrounding biomolecules has a strong impact on their function, rapid calculation of these maps is desirable for molecular structure visualization and analysis, and can serve as part of a systematic method for placing ions when preparing a system for simulation [38–40]. More generally, GPU particle-grid algorithms that evaluate spatially related quantities on grids surrounding a molecule arise in many other applications, such as the calculation of time-averaged spatial occupancy maps and the visualization of molecular orbitals [22].

3.1.1. Direct summation

The electrostatic potential contribution from each atom is summed to every point in a three-dimensional lattice. The computation has quadratic time complexity $O(MN)$ for N atoms and M lattice points. Although direct summation is impractical for large problem sizes, its implementation on GPUs has provided valuable insight into the design of similar algorithms and was an early test case for distributing computation across multiple GPUs within the same host machine [2,5]. Kernel development for direct summation showed that contemporary GPUs perform best with workloads dominated by arithmetic operations, in which the most effective optimizations increased arithmetic intensity and decreased memory references. Use of the fast on-chip GPU constant memory system for broadcasting atomic coordinates and charges yields operand access at near-register-speed and, in combination with data reuse and manual loop unrolling techniques, results in a kernel that is entirely arithmetic bound. GPU-accelerated direct summation achieves speedup factors of 44 for G80 and 88 for GT200 architectures compared to a single CPU core. The optimal kernels achieve a large fraction of the theoretical peak GPU arithmetic performance, providing a rough measure of the peak floating point performance possible for related molecular modeling algorithms.

3.1.2. Cutoff summation

Distance-based potentials are typically truncated to zero beyond a cutoff radius, reducing overall algorithm time complexity from quadratic to linear $O(M+N)$. GPU kernels for efficient summation of a cutoff electrostatic potential require changes to the data structures and memory utilization from those employed by the kernels for direct summation. The CPU performs a spatial hashing of the atom data into an array of atom bins, which is then copied to the GPU. Each work group cooperatively streams the sphere of surrounding atom bins into the GPU on-chip shared memory to calculate contributions to its cubic region of grid points, effectively using the shared memory as a managed cache. The atom bins are chosen to have a fixed data size that is compatible with coalesced memory reads, and the spatial dimensions assigned to the bins are chosen to maintain a high average occupancy of atoms to optimize GPU performance. The CPU concurrently calculates contributions from any atoms that overflow the bin size for regions of the system with an unusually high atom density, making productive use of the combined computational resources [41]. Even with algorithmic parameters that effectively hide memory access latency, the GPU kernels for cutoff summation require enough extra bookkeeping to limit speedup factors to no more than about 32 for GT200 compared to a single CPU core [42].

3.1.3. Multilevel summation

Rather than neglecting the longer-range contributions to the electrostatic potential, it is desirable to approximate them using an efficient method. The multilevel summation method calculates a smooth approximation to the full electrostatic potential through the nested interpolation of successive smoothings of $1/r$ from a hierarchy of multi-resolution lattices [43,44]. The GPU-accelerated implementation of multilevel summation for calculating a map of the electrostatic potential makes use of the cutoff summation GPU kernel for the short-range part and introduces another GPU kernel for calculating the three-dimensional convolutions performed over the multi-resolution lattices, with the CPU calculating the less computationally intensive work [42]. The overall computational complexity is linear $O(M+N)$, with the GPU implementation achieving a speedup factor of 26 for GT200.

3.1.4. Fast multipole method

Older and better known than multilevel summation, the fast multipole method also approximates the full electrostatic potential with computational work that scales linearly in the number of atoms [45]. Although capable of providing high accuracy, the fast multipole method is best suited for applications that do not depend on continuity, such as the calculation of electrostatic potential maps, since it produces discontinuous potentials. Work by Gumerov et al. that performs the entire computation on the GPU reports for 1 million particles speedup factors, depending on the accuracy settings, in the range of 30–60, although the benchmarking has been performed against a less extensively optimized CPU implementation [46].

3.2. Molecular dynamics

One of the most compelling applications for GPU computing has been the acceleration of molecular dynamics simulations based on classical mechanics. Despite years of investment in efficient algorithms and large scale parallel processing techniques, the demands of biomedical research require simulations on biomolecular complexes of increasing size and sophistication, on longer timescales, with better sampling, and with improved force fields. Each of these dimensions creates demand for more computation, something that GPUs can help address.

3.2.1. Folding@Home

An early success in the application of GPUs to molecular dynamics is the Folding@Home distributed computing project,[3] originally based on the BrookGPU programming toolkit [4]. Within a short time of introducing GPU-accelerated clients, a large fraction of the project's overall computing power was being supplied by GPUs. The Folding@Home team has continued to develop new GPU kernels

[3] http://www.folding.stanford.edu/.

optimized for fast protein folding simulations. They have devised GPU kernels for Onufriev–Bashford–Case Generalized Born implicit solvent simulations with non-periodic boundary conditions using an all pairs $O(N^2)$ non-bonded force evaluation technique, which works best on small systems of a few thousand atoms. Both the ATI and NVIDIA GPU implementations yield GPU speedup factors well over 100 compared to AMBER 9 running on a single-core CPU [3,23,33]. Today, the overwhelming majority of Folding@Home computational power is supplied by GPUs.[4] These GPU kernels are also being deployed in the OpenMM[5] software library for molecular simulation.

3.2.2. NAMD

Prior to the introduction of CUDA [32] in 2007, none of the general purpose molecular dynamics simulation packages had yet been modified to take advantage of GPU acceleration. With the availability of CUDA, NAMD[6][47] became the first such software package to incorporate GPU acceleration [5]. The CUDA programming environment's support for more sophisticated data structures and memory access patterns, combined with the general purpose GPU hardware support for fast on-chip shared memory, made it practical for the first time to design full-featured kernels for molecular dynamics.

Since NAMD is a complex program supporting a diversity of desktop and supercomputing platforms, an incremental approach has been undertaken to incorporate GPU acceleration. The initially developed GPU kernels calculate just the $O(N)$ short-range non-bonded interactions, which is the single most demanding part of the force evaluation, leaving the host CPUs to perform the remaining computations and message passing operations. NAMD is typically used for simulations of large biomolecular complexes in explicit solvent with periodic boundary conditions, and the GPU calculation has to support the short-range part of the particle-mesh Ewald (PME) [48] method for approximating full electrostatics in a periodic domain. The various challenges overcome include maintaining sufficient numerical accuracy despite the use of single-precision floating point arithmetic, handling exclusions, and using CPU-fallback methods to provide compatibility with advanced simulation features [2,5].

Subsequent NAMD GPU development efforts have focused on making effective use of GPU clusters, which present several challenges when compared with single-machine scenarios [8]. Contemporary GPUs require a large amount of work to operate efficiently, so GPU algorithms typically aggregate work until enough is available to effectively use the GPU. In a cluster or other large scale parallel environment, the simulation workload on a single node can be partitioned into two sets, the "remote" portion resulting from communications with peers, and the "local" portion that depends only on node-local data. To optimize GPU execution, one would ideally first submit all of the already available local work, followed by submission of the remote work once it arrives. Although this strategy gives the GPU the best opportunity to achieve peak performance, it adds latency to the execution of remote work which can delay the critical path for overall parallel execution. NAMD benchmarks on up to 60 GPUs on the "AC" cluster at the National Center for Supercomputing Applications (NCSA) have demonstrated overall application speedups of 3.4–7 depending on system size. A 1 million atom virus simulation running on 60 GPUs achieved performance equivalent to 330 CPU cores on the same cluster [8]. Performance-per-watt measurements have shown an overall GPU-accelerated NAMD energy efficiency improvement factor of 2.7, compared to CPU-only runs [49]. Recent NAMD benchmarks of large simulations on the NCSA "Lincoln" cluster have yielded per-GPU performance benefits roughly equivalent to adding 12 CPU cores. GPUs have not only improved the price/performance of NAMD simulations, but also the space, power, and cooling requirements for high performance NAMD clusters. Recent NAMD GPU work has focused on broadening the range of simulation types that benefit from GPU acceleration, improved parallel scalability for GPU-accelerated clusters, better support for asymmetric numbers of CPU cores and GPUs within cluster nodes, and tuning for newer GPU hardware such as NVIDIA's Fermi.

3.2.3. HOOMD

Specializing in molecular dynamics simulations of polymer systems, HOOMD[7] is freely available software explicitly designed for GPU execution [3,7]. Achieving speedups of over a factor of 30 compared to the same simulations run with the LAMMPS [50] package on CPUs, HOOMD has enabled a variety of coarse-grained particle simulations that would have been impractical before. Results from a recent study to determine the stability of points of the double gyroid phase for a polymer required more than 600 different simulation runs of 48 GPU-hours apiece [51]; the same set of simulations run with LAMMPS would have required almost one million CPU-hours. Rather than using the GPU to accelerate only part of the force calculation, HOOMD keeps the entire simulation data within the GPU memory to overcome the CPU to GPU memory transfer bottleneck. A number of GPU-specific algorithms and approaches are used, including the sorting of atoms to reduce branch divergence, the effective use of pair lists, and optimizations that take advantage of atomic operations, along with other features found only in state-of-the-art GPUs.

3.2.4. ACEMD

Released as a commercially licensed biomolecular dynamics software package, ACEMD[8] is explicitly designed for execution by a single workstation with multiple GPUs [52]. Almost all of the computational work is performed on the GPUs in an effort to maximize performance. ACEMD provides many features typically required for production simulations, including compatibility with CHARMM and AMBER force fields, long-range electrostatics with PME [53], temperature control, and hydrogen bond constraints. The design uses a task-parallel decomposition strategy, rather than the spatial data-parallel decomposition used in NAMD, which limits its scalability to not more than three or four GPUs. ACEMD appears to be most effective for system sizes of 10K to 100K atoms.

3.2.5. Other work

Some of the other research into accelerating molecular dynamics with GPUs includes work by Liu et al. [54] and Davis et al. [55]. Liu et al. reports on accelerating the time integration of particles acting under a simple Lennard–Jones potential with a cutoff distance, showing speedup factors of about 7–11 over optimized routines from LAMMPS on systems ranging from 8K to 131K in size. Davis et al. simulates flexible water entirely on the GPU, showing a speedup factor of about 7 over CHARMM on a single CPU for a system of 20K atoms.

3.3. Quantum chemistry

Quantum chemistry simulations constitute the most computationally demanding applications within molecular modeling. As the size of simulated molecules grows, or more accurate techniques are used, computational complexity often grows non-linearly, such

[4] http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats.
[5] https://simtk.org/home/openmm.
[6] http://www.ks.uiuc.edu/Research/namd/.
[7] http://codeblue.umich.edu/hoomd-blue.
[8] http://multiscalelab.org/acemd.

as quadratically $O(N^2)$, cubically $O(N^3)$, quartically $O(N^4)$, or more. However, increased computational complexity provides greater opportunity for data reuse, which can favor the GPU over the CPU by increasing the ratio of arithmetic operations to memory accesses. Many first principles quantum chemistry algorithms are also embarrassingly parallel and thereby particularly well suited to GPU architectures.

3.3.1. Quantum chemistry simulation

In 2007, Anderson et al. implemented for the first time the Quantum Monte-Carlo (QMC) algorithm on a GPU [56]. Remarkably, the code was written using the Cg language, a popular graphics-specific programmable shading language available prior to the introduction of CUDA and OpenCL. After the introduction of CUDA in 2007, the effectiveness of GPU-accelerated quantum chemistry calculation was demonstrated for QMC [57], evaluation of two-electron ($2e$) integrals [21,58,59], Hartree-Fock (HF) [60–62], density functional theory (DFT) [63–65], and correlated methods [66,67]. HF and DFT algorithms require calculation of millions and sometimes billions of the $2e$-integrals that describe electrostatic repulsion between uncorrelated pairs of electrons. In addition, most popular DFT methods also involve numerical integration of various functionals on a unified set of spherical atom-centered grids. The $2e$ integral and numerical integration problems are embarrassingly parallel and perfectly map onto the GPU architecture demonstrating from one- to two-order of magnitude performance gains with respect to traditional CPU-based solutions.

On the other hand, porting existing serial and even parallel CPU-based quantum chemistry applications to the GPU architecture is not straightforward, because in most cases the programs are designed in such a way as to extensively reuse intermediate data. This strategy generally reduces the total number of arithmetic operations required to complete the same amount of work but necessitates higher utilization of the memory bandwidth and causes random (non-contiguous) memory access patterns, hampering GPU performance significantly. In many cases, to obtain the maximum computation speed on GPUs one needs to redesign existing quantum chemistry algorithms, sometimes increasing the number of floating point operations in exchange for a reduced number of memory operations and contiguous memory accesses [21,60,63,64]. In contrast, applications designed from the outset for massively parallel processors (in this particular case, GRAPE-DR) can be ported to the GPU quite efficiently due to conceptual similarities between various massively parallel architectures [58].

Another way to accelerate existing CPU-based applications is to replace standard CPU library (BLAS, LAPACK) function calls by their counterparts from GPU-accelerated libraries available from several vendors. This porting strategy is especially appealing because it requires essentially no code reorganization and has been demonstrated to be relatively efficient for applications formulated in terms of BLAS-3 operations [66]. Limitations in the amount of on-board GPU memory for large-sized matrix problems and the slower performance of double precision versus single precision operations on the GPU have led to the development of customized matrix libraries [67]. Such libraries address memory capacity limitations by decomposing a large problem into smaller problems that fit into the GPU memory. Performance is improved through the use of mixed precision calculations that take advantage of the speed and memory bandwidth benefits of single precision arithmetic while preserving accuracy with limited use of double precision.

3.3.2. Quantum chemistry visualization

As the size of systems routinely treated at the quantum chemistry level approaches 1000 atoms and the typical length of first

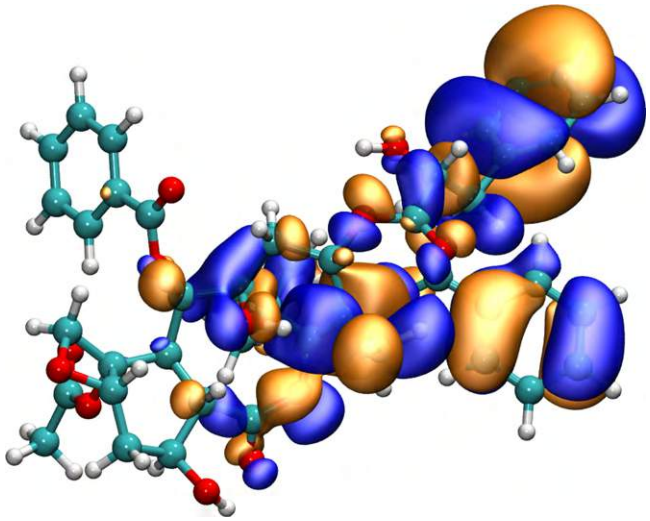

Fig. 3. This molecular orbital representation of a 102 atom Taxol molecule is an example of the kind of quantum chemistry data that can be interactively displayed and animated in VMD, with on-the-fly GPU computation of molecular orbital grids [22].

principles molecular dynamics simulations reaches 1 ns, visualization of the results of such calculations (molecular orbitals, electron and spin density, etc.) becomes a challenge for traditional processors. A large computational cost is associated with the evaluation of various functions on the three-dimensional grids needed for rendering of the corresponding isosurfaces. Because the evaluation of functions on grids is intrinsically parallel, it creates an ideal opportunity for offloading this part of the visualization process to the GPU. In fact, the recent development of orbital visualization tools within the popular molecular visualization program VMD [68] has clearly demonstrated great superiority of GPU over CPU for this type of problem [22]. Fig. 3 depicts a typical molecule whose molecular orbitals can be rendered interactively using VMD. Molecular orbital computations using four GPUs have yielded speedups as high as 412 compared to a single CPU core. The use of multiple GPUs to accelerate latency-sensitive visualization calculations requires special techniques for dynamic load balancing and handling of errors. VMD contains a framework for efficient execution of latency-sensitive multi-GPU computations.

3.4. Additional applications

Besides the application areas already discussed, GPUs have been successfully employed for several other important tasks in molecular modeling.

3.4.1. Protein–ligand docking

Molecular docking problems often involve computationally demanding calculations of protein–ligand or protein–protein interaction energies, for many possible translations, orientations, or poses of docked structures. Korb has developed GPU-accelerated transformations and scoring functions for protein–ligand docking, yielding an overall speedup factor of 5 or more [69]. Although this work was done using the OpenGL graphics interface, the algorithms could be directly expressed in the compute-specific CUDA or OpenCL toolkits, thereby alleviating concerns about floating point accuracy and portability that exist for implementations based on OpenGL programmable shading techniques. Sukhwani et al. present a GPU-accelerated version of the docking program PIPER [70], achieving an overall speedup factor of 17 or more [71]. Sukhwani et al. also compare performance with FPGA accelera-

tors, and evaluate both FFT-based correlation and direct correlation approaches for a variety of ligand sizes.

3.4.2. Molecular surface area

The computation of solvent-accessible, solvent-excluded, and molecular surface areas are important for calculation of solvation energies and are of interest in the study of protein folding. Juba et al. have created a parallel molecular surface area algorithm based on the use of a surface defined by summed Gaussian radial basis functions [72]. The surface is then sampled stochastically, estimating surface area by counting line-surface intersections. They report runtime and difference in estimated surface area compared to several other popular programs, achieving performance levels of up to 24 times faster than other tools running on the CPU, and with surface area differences typically less than 5% over a wide range of problem sizes.

Dynerman et al. describe a GPU algorithm for computing solvent-accessible surface areas, as used in computing desolvation energy [73]. The surface area algorithm uses an $O(N^2)$ pairwise geometric area formulation that yields a twice differentiable approximation practical for optimization problems. The strong linear relationship between desolvation energy and solvent-accessible surface area is exploited to compute the desolvation energy. The GPU algorithm for computing solvent-accessible surface area yielded speedups ranging from 30 to over 200 for a range of problem sizes, when comparing runtime for 2 CPUs against 2 GPUs, although the authors note that the CPU implementation was not extensively optimized. Since the runtime of the desolvation algorithm is composed primarily of the surface area computation time, it yields similar speedups.

3.4.3. Implicit ligand sampling

The computation of the occupancy map reveals within a protein the most likely channels for a gas, such as methane [74,75]. The concepts behind implicit ligand sampling and parts of its calculation bear resemblance to protein–ligand docking. The occupancy map is averaged over a series of RMSD-aligned trajectory frames of the molecule, for which each of the map contributions is determined by calculating a dense lattice of the non-bonded potential between the fixed molecular structure and the rotations of the small ligand. The application of a single GPU to this compute intensive procedure has shown speedup factors of 10–30 over highly optimized CPU code, depending on the size and shape of the ligand. Although the details of the GPU algorithm are as yet unpublished, the GPU implementation is already available for use in VMD. The GPU algorithm shares similarities with the electrostatics map calculation of a cutoff potential [41], but must look up Lennard–Jones parameters and stream through a list of rotations of the ligand. Performance is enhanced by an initial culling of the lattice points for which occupancy is near zero.

3.4.4. Other work

Haque et al. have recently described a GPU algorithm for Gaussian molecular shape overlay [76]. Their algorithm is an open source implementation of the Gaussian volume overlap optimization approach used by the ROCS (Rapid Overlay of Chemical Structures) package produced by OpenEye Scientific Software. These algorithms are used to identify spatial features common among a set of molecules, and can be used to measure similarity for ligand-based compound discovery approaches. Haque et al. report speedups of 20 compared to highly optimized CPU implementations and also include results for low-cost platforms of interest to distributed computing efforts such as Folding@Home.

Histogramming algorithms tend to perform well on GPUs due to their intrinsic parallelism, particularly in the case of recent devices that support arbitrary scatter and gather operations to global memory. State-of-the-art GPUs also support *atomic-add* operations, further simplifying the construction of efficient histogramming techniques, reducing the need for multi-pass algorithms. Recent development has begun toward the implementation of radial distribution functions on GPUs, taking advantage of the GPUs' arithmetic capabilities for calculating particle pair distances and their ability to perform fast histogramming. Preliminary testing has achieved GPU speedups as high as 70 compared to a single CPU core when computing radial distribution histograms. The completed implementation is planned for release in VMD [68], augmenting the existing CPU implementation.

Many other molecular modeling tools are currently in the process of incorporating GPU acceleration. GPU-accelerated versions of AMBER,[9] CHARMM, DL_POLY,[10] GROMACS,[11] LAMMPS,[12] AutoDock,[13] BigDFT,[14] and QMCPACK[15] are in development at the time of this writing. It seems likely that within a short time, many of the mainstream molecular simulation packages will support GPU acceleration to some degree.

4. Future outlook

The broad range of existing work demonstrates performance benefits from the use of GPUs as massively parallel co-processors for arithmetic-intensive molecular modeling applications. State-of-the-art GPU hardware designs, like the AMD Cypress and NVIDIA Fermi, offer greatly increased computational capabilities and solve some of the past limitations of GPUs. Double precision is now supported at just twice the cost of single precision operations, and error correcting memory is supported. CUDA driver and compiler technology is mature enough for production-level software development, and OpenCL is anticipated to follow this lead as vendor support improves.

Parallelization across GPU-accelerated clusters brings additional issues. Several problems from molecular modeling have already demonstrated good scaling for large problem sizes on GPU clusters, with improved performance and reduced power consumption as compared with traditional HPC clusters. Strong scaling on GPU clusters is still difficult to attain for small problem sizes due to the additional sources of latency introduced by GPU devices. In particular, a key challenge here is to improve the performance of molecular dynamics simulations of small systems over longer timescales. The recent GPU support for the execution of workloads provided in smaller batches will help to improve latency issues with GPU clusters. Future improvements are expected that will reduce host-GPU memory transfer overhead and that will provide new interfaces allowing efficient communication directly between GPUs within the same host.

The most exciting applications of GPUs to molecular modeling are perhaps those that have resulted in a "computational phase transition," transforming what had previously been batch-mode computations on clusters into interactive computations that can now be performed on laptop or desktop computers. Recent and continuing improvements to the state-of-the-art GPUs, such as new special-purposes caches and hardware reduction operations, will enable development of GPU algorithms that were previously difficult to map to GPU hardware with high efficiency, while reducing the number of development hurdles encountered by computational

[9] http://ambermd.org/gpus/.
[10] http://www.ichec.ie/research/gpgpu_projects.
[11] https://simtk.org/home/openmm.
[12] http://code.google.com/p/gpulammps/.
[13] http://sourceforge.net/projects/gpuautodock/.
[14] http://inac.cea.fr/L_Sim/BigDFT/.
[15] http://qmcpack.cmscc.org/.

scientists that are just beginning to learn GPU programming techniques.

Acknowledgements

This work was supported by the National Institutes of Health under grant P41-RR05969. Performance experiments were made possible by a generous hardware donation by NVIDIA. Ivan Ufimtsev would like to acknowledge an NVIDIA fellowship and National Science Foundation grant (CHE-06-26354).

References

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell, A survey of general-purpose computation on graphics hardware, Comput. Graph. Forum 26 (2007) 80–113.
[2] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips, GPU computing, Proc. IEEE 96 (2008) 879–899.
[3] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov, Parallel computing experiences with CUDA, IEEE Micro 28 (2008) 13–27.
[4] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, V. Pande, N-body simulation on GPUs, in: SC06 Proceedings, IEEE Computer Society, 2006.
[5] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, J. Comp. Chem. 28 (2007) 2618–2640.
[6] J. Yang, Y. Wang, Y. Chen, GPU accelerated molecular dynamics simulation of thermal conductivities, J. Chem. Phys. 221 (2007) 799–804.
[7] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Chem. Phys. 227 (2008) 5342–5359.
[8] J.C. Phillips, J.E. Stone, K. Schulten, Adapting a message-driven parallel application to GPU-accelerated clusters, in: SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008.
[9] G. Giupponi, M. Harvey, G.D. Fabritiis, The impact of accelerator processors for high-throughput molecular modeling and simulation, Drug Discov. Today 13 (2008) 1052–1058.
[10] J.C. Phillips, J.E. Stone, Probing biomolecular machines with graphics processors, Commun. ACM 52 (2009) 34–41.
[11] N. Azizi, I. Kuon, A. Egier, A. Darabiha, P. Chow, Reconfigurable molecular dynamics simulator, in: Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp. 197–206.
[12] Y. Gu, T. VanCourt, M. Herbordt, Accelerating molecular dynamics simulations with configurable circuits, in: International Conference on Field Programmable Logic and Applications, 2005, pp. 475–480.
[13] R. Susukita, T. Ebisuzaki, B.G. Elmegreen, H. Furusawa, K. Kato, A. Kawai, Y. Kobayashi, T. Koishi, G.D. McNiven, T. Narumi, K. Yasuoka, Hardware accelerator for molecular dynamics: MDGRAPE-2, Comput. Phys. Commun. 155 (2003) 115–131.
[14] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, A. Konagaya, Protein Explorer: a petaflops special-purpose computer system for molecular dynamics simulations, in: Supercomputing, 2003 ACM/IEEE Conference, 2003, pp. 15–115.
[15] T. Narumi, Y. Ohno, N. Okimoto, T. Koishi, A. Suenaga, N. Futatsugi, R. Yanai, R. Himeno, S. Fujikawa, M. Taiji, M. Ikei, A 55 TFLOPS simulation of amyloid-forming peptides from yeast prion sup35 with the special-purpose computer system MDGRAPE-3, in: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM, New York, NY, USA, 2006.
[16] H.P. Hofstee, Power efficient processor architecture and the Cell processor, in: HPCA'05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, IEEE Computer Society, Washington, DC, USA, 2005, pp. 258–262.
[17] D. Kunzman, G. Zheng, E. Bohm, L.V. Kalé, Charm++, Offload API, and the Cell Processor, in: Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism, ACM, Seattle, WA, USA, 2006.
[18] S. Swaminarayan, K. Kadau, T.C. Germann, G.C. Fossum, 369 Tflop/s molecular dynamics simulations on the Roadrunner general-purpose heterogeneous supercomputer, in: SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–10.
[19] G. Shi, V. Kindratenko, Implementation of NAMD molecular dynamics non-bonded force-field on the Cell Broadband Engine processor, in: IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, 2008, pp. 1–8.
[20] D.M. Kunzman, L.V. Kalé, Towards a framework for abstracting accelerators in parallel applications: experience with cell, in: SC'09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing, ACM, New York, NY, USA, 2009, pp. 1–12.
[21] I.S. Ufimtsev, T.J. Martinez, Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation, J. Chem. Theor. Comp. 4 (2008) 222–231.
[22] J.E. Stone, J. Saam, D.J. Hardy, K.L. Vandivort, W.W. Hwu, K. Schulten, High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs, in: Proceedings of the 2nd Workshop on General-Purpose Processing on Graphics Processing Units, ACM International Conference Proceeding Series, vol. 383, ACM, New York, NY, USA, 2009, pp. 9–18.
[23] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, V.S. Pande, Accelerating molecular dynamic simulation on graphics processing units, J. Comp. Chem. 30 (2009) 864–872.
[24] F. Chinchilla, T. Gamblin, M. Sommervoll, J.F. Prins, Parallel N-Body Simulation Using GPUs, Technical Report TR04-032, Department of Computer Science, University of North Carolina at Chapel Hill, 2004.
[25] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU cluster for high performance computing, in: SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2004, p. 47.
[26] H. Takizawa, H. Kobayashi, Hierarchical parallel processing of large scale data clustering on a pc cluster with GPU co-processing, J. Supercomput. 36 (2006) 219–234.
[27] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S.H.M. Buijssen, M. Grajewski, S. Turek, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, Parallel Comput. 33 (2007) 685–699.
[28] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: a unified graphics and computing architecture, IEEE Micro 28 (2008) 39–55.
[29] NVIDIA, NVIDIA's next generation CUDA compute architecture: Fermi, White Paper, NVIDIA, 2009. Available online, Version 1.1, 22 pp.
[30] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, in: SIGGRAPH'04: ACM SIGGRAPH 2004 Papers, ACM Press, New York, NY, USA, 2004, pp. 777–786.
[31] M. McCool, S. Du Toit, T. Popa, B. Chan, K. Moule, Shader algebra, ACM Trans. Graph. 23 (2004) 787–795.
[32] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA, Santa Clara, CA, USA, 2007.
[33] E. Elsen, V. Vishal, M. Houston, V. Pande, P. Hanrahan, E. Darve, N-Body Simulations on GPUs, Technical Report, Stanford University, Stanford, CA, 2007. http://arxiv.org/abs/0706.3060.
[34] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, ACM Queue 6 (2008) 40–53.
[35] A. Munshi, OpenCL Specification Version 1.0, 2008. http://www.khronos.org/registry/cl/.
[36] J.E. Stone, D. Gohara, G. Shi, OpenCL: a parallel programming standard for heterogeneous computing systems, Comput. Sci. Eng. 12 (2010) 66–73.
[37] V. Volkov, J.W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–11.
[38] J. Gumbart, L.G. Trabuco, E. Schreiner, E. Villa, K. Schulten, Regulation of the protein-conducting channel by a bound ribosome, Structure 17 (2009) 1453–1464.
[39] L.G. Trabuco, E. Villa, E. Schreiner, C.B. Harrison, K. Schulten, Molecular Dynamics Flexible Fitting: A practical guide to combine cryo-electron microscopy and X-ray crystallography, Methods 49 (2009) 174–180.
[40] L.G. Trabuco, C.B. Harrison, E. Schreiner, K. Schulten, Recognition of the regulatory nascent chain TnaC by the ribosome, Structure 18 (2010) 627–637.
[41] C.I. Rodrigues, D.J. Hardy, J.E. Stone, K. Schulten, W.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, NY, USA, 2008, pp. 273–282.
[42] D.J. Hardy, J.E. Stone, K. Schulten, Multilevel summation of electrostatic potentials using graphics processing units, J. Parallel Comput. 35 (2009) 164–177.
[43] R.D. Skeel, I. Tezcan, D.J. Hardy, Multiple grid methods for classical molecular dynamics, J. Comp. Chem. 23 (2002) 673–684.
[44] D.J. Hardy, Multilevel Summation for the Fast Evaluation of Forces for the Simulation of Biomolecules, Ph.D. thesis, Univ. of Illinois at Urbana-Champaign, 2006. Also Department of Computer Science Report No. UIUCDCS-R-2006-2546, May 2006.
[45] L. Greengard, V. Rokhlin, A fast algorithm for particle simulation, J. Comp. Phys. 73 (1987) 325–348.
[46] N.A. Gumerov, R. Duraiswami, Fast multipole methods on graphics processors, J. Comp. Phys. 227 (2008) 8290–8313.
[47] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, K. Schulten, Scalable molecular dynamics with NAMD, J. Comp. Chem. 26 (2005) 1781–1802.
[48] 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.
[49] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, W. Hwu, QP: a heterogeneous multi-accelerator cluster, in: 10th LCI International Conference on High-Performance Clustered Computing, Boulder, CO, USA, 2009.
[50] S.J. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comp. Phys. 117 (1995) 1–19.
[51] C.L. Phillips, C.R. Iacovella, S.C. Glotzer, Stability of the double gyroid phase to nanoparticle polydispersity in polymer-tethered nanosphere system, Soft. Mat. 6 (2010) 1693–1703.
[52] M.J. Harvey, G. Giupponi, G.D. Fabritiis, ACEMD: accelerating biomolecular dynamics in the microsecond time scale, J. Chem. Theor. Comp. 5 (2009) 1632–1639.
[53] M.J. Harvey, G.D. Fabritiis, An implementation of the smooth particle mesh Ewald method on GPU hardware, J. Chem. Theor. Comp. 5 (2009) 2371–2377.

[54] W. Liu, B. Schmidt, G. Voss, W. Müller-Wittig, Accelerating molecular dynamics simulations using graphics processing units with CUDA, Comput. Phys. Commun. 179 (2008) 634–641.
[55] J.E. Davis, A. Ozsoy, S. Patel, M. Taufer, Towards large-scale molecular dynamics simulations on graphics processors, in: BICoB'09: Proceedings of the 1st International Conference on Bioinformatics and Computational Biology, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 176–186.
[56] A.G. Anderson, W.A. Goddard III, P. Schröder, Quantum Monte Carlo on graphical processing units, Comput. Phys. Commun. 177 (2007) 298–306.
[57] J.S. Meredith, G. Alvarez, T.A. Maier, T.C. Schulthess, J.S. Vetter, Accuracy and performance of graphics processors: A quantum Monte Carlo application case study, J. Parallel Comput. 35 (2009) 151–163.
[58] K. Yasuda, Two-electron integral evaluation on the graphics processor unit, J. Comp. Chem. 29 (2008) 334–342.
[59] A. Asadchev, V. Allada, J. Felder, B.M. Bode, M.S. Gordon, T.L. Windus, Uncontracted Rys quadrature implementation of up to g functions on graphical processing units, J. Chem. Theor. Comp. 6 (2010) 696–704.
[60] I.S. Ufimtsev, T.J. Martinez, Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation, J. Chem. Theor. Comp. 5 (2009) 1004–1015.
[61] N. Sanna, I. Baccarelli, G. Morelli, SCELib3.0: The new revision of SCELib, the parallel computational library of molecular properties in the Single Center Approach, Comput. Phys. Commun. 180 (2009) 2544–2549.
[62] I.S. Ufimtsev, T.J. Martinez, Graphical processing units for quantum chemistry, Comput. Sci. Eng. 10 (2008) 26–34.
[63] I.S. Ufimtsev, T.J. Martinez, Quantum chemistry on graphical processing units. 3. Analytical energy gradients, geometry optimization, and first principles molecular dynamics, J. Chem. Theor. Comp. 5 (2009) 2619–2628.
[64] K. Yasuda, Accelerating density functional calculations with graphics processing unit, J. Chem. Theor. Comp. 4 (2008) 1230–1236.
[65] L. Genovese, M. Ospici, T. Deutsch, J.-F. Méhaut, A. Neelov, S. Goedecker, Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures, J. Chem. Phys. 131 (2009) 034103.
[66] L. Vogt, R. Olivares-Amaya, S. Kermes, Y. Shao, C. Amador-Bedolla, A. Aspuru-Guzki, Accelerating resolution-of-the-identity second-order Møller–Plesset quantum chemistry calculations with graphical processing units, J. Phys. Chem. A 112 (2008) 2049–2057.
[67] R. Olivares-Amaya, M.A. Watson, R.G. Edgar, L. Vogt, Y. Shao, A. Aspuru-Guzik, Accelerating correlated quantum chemistry calculations using graphical processing units and a mixed precision matrix multiplication library, J. Chem. Theor. Comp. 6 (2010) 135–144.
[68] W. Humphrey, A. Dalke, K. Schulten, VMD – Visual Molecular Dynamics, J. Mol. Graphics 14 (1996) 33–38.
[69] O. Korb, Efficient Ant Colony Optimization Algorithms for Structure- and Ligand-Based Drug Design, Ph.D. thesis, Universität Konstanz, Universitätsstr. 10, 78457 Konstanz, 2008.
[70] D. Kozakov, R. Brenke, S.R. Comeau, S. Vajda, PIPER: an FFT-based protein docking program with pairwise potentials, Proteins: Struct., Func., Bioinf. 65 (2006) 392–406.
[71] B. Sukhwani, M.C. Herbordt, GPU acceleration of a production molecular docking code, in: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, New York, NY, USA, 2009, pp. 19–27.
[72] D. Juba, A. Varshney, Parallel, stochastic measurement of molecular surface area, J. Mol. Graph. Model. 27 (2008) 82–87.
[73] D. Dynerman, E. Butzlaff, J.C. Mitchell, CUSA and CUDE: GPU-accelerated methods for estimating solvent accessible surface area and desolvation, J. Comp. Biol. 16 (2009) 523–537.
[74] J. Cohen, A. Arkhipov, R. Braun, K. Schulten, Imaging the migration pathways for O_2, CO, NO, and Xe inside myoglobin, Biophys. J. 91 (2006) 1844–1857.
[75] J. Cohen, K.W. Olsen, K. Schulten, Finding gas migration pathways in proteins using implicit ligand sampling, in: R.K. Poole (Ed.), Globins and other NO-reactive Proteins in Microbes, Plants and Invertebrates, vol. 437 of Methods in Enzymology, Elsevier, 2008, pp. 437–455.
[76] I.S. Haque, V.S. Pande, PAPER - accelerating parallel evaluations of ROCS, J. Comp. Chem. 31 (2009) 117–132.