CHAPTER 1

Advancements in Molecular Dynamics Simulations of Biomolecules on Graphical Processing Units

Dong Xu[1,2], **Mark J. Williamson**[1], and **Ross C. Walker**[1]

Contents

Abstract

Over the past few years competition within the computer game market coupled with the emergence of application programming interfaces to support general purpose computation on graphics processing units (GPUs) has led to an explosion in the use of GPUs for acceleration of scientific applications. Here we explore the use of GPUs within the context of condensed phase molecular dynamics (MD) simulations. We discuss the algorithmic differences that the GPU architecture imposes on MD codes, an overview of the challenges involved in using GPUs for MD, followed by a

[1] San Diego Supercomputer Center, University of California San Diego, La Jolla, CA, USA

[2] National Biomedical Computation Resource, University of California San Diego, La Jolla, CA, USA

Annual Reports in Computational Chemistry, Volume 6
ISSN: 1574-1400, DOI 10.1016/S1574-1400(10)06001-9

critical survey of contemporary MD simulation packages that are attempting to utilize GPUs. Finally we discuss the possible outlook for this field.

Keywords: GPU; CUDA; stream; NVIDIA; ATI; molecular dynamics; accelerator; OpenMM; ACEMD; NAMD; AMBER

1. INTRODUCTION

Since the first molecular dynamics (MD) simulation of an enzyme was described by McCammon et al. [1]. MD simulations have evolved to become an important tool in understanding the behavior of biomolecules. Since that first 10 ps long simulation of merely 500 atoms in 1977, the field has grown to where small enzymes can be routinely simulated on the microsecond timescale [2–4]. Simulations containing millions of atoms are now also considered routine [5,6]. Such simulations are numerically intensive requiring access to large-scale supercomputers or well-designed clusters with expensive interconnects that are beyond the reach of many research groups.

Many attempts have been made over the years to accelerate classical MD simulation of condensed-phase biological systems by exploiting alternative hardware technologies. Some notable examples include ATOMS by AT&T Bell Laboratories [7], FASTRUN designed by Columbia University in 1984 and constructed by Brookhaven National Laboratory in 1989 [8], the MDGRAPE system by RIKEN [9] which used custom hardware–accelerated lookup tables to accelerate the direct space nonbond calculations, Clearspeed Inc. which developed an implicit solvent version of the AMBER PMEMD engine [10,11] that ran on their custom designed Advance X620 and e620 acceleration cards [12], and most recently DE Shaw Research LLC who developed their own specialized architecture for classical MD simulations code-named Anton [13].

All of these approaches have, however, failed to make an impact on mainstream research because of their excessive cost. Table 1 provides estimates of the original acquisition or development costs of several accelerator technologies. These costs have posed a significant barrier to widespread development within the academic research community. Additionally these technologies do not form

Table 1 Example cost estimates for a range of hardware MD acceleration projects

Accelerator technology	Manufacturer	Estimated cost per node
CX600	ClearSpeed	~$10,000
MDGRAPE-3	Riken	~$9,000,000[a]
ATOMS	AT&T Bell Laboratories	~$186,000 (1990)
FASTRUN	Columbia University and Brookhaven National Laboratory	~$17,000 (1989)
GPU	NVIDIA/ATI	$200–800

[a] Total development cost: $15 million [14].

part of what would be considered a standard workstation specification. This makes it difficult to experiment with such technologies leading to a lack of sustained development or innovation and ultimately their failure to mature into ubiquitous community-maintained research tools.

Graphics processing units (GPUs), on the other hand, have been an integral part of personal computers for decades. Ever since 3DFX first introduced the Voodoo graphics chip in 1996, their development has been strongly influenced by the entertainment industry in order to meet the demands for ever increasing realism in computer games. This has resulted in significant industrial investment in the stable, long-term development of GPU technology. Additionally the strong demand from the consumer electronics industry has resulted in GPUs becoming cheap and ubiquitous. This, combined with substantial year over year increases in the computing power of GPUs, means they have the potential, when utilized efficiently, to significantly outperform CPUs (Figure 1). This makes them attractive hardware targets for acceleration of many scientific applications including MD simulations. The fact that high-end GPUs can be considered standard equipment in scientific workstations means that they already exist in many research labs and can be purchased easily with new equipment. This makes them readily available to researchers and thus tempting instruments for computational experimentations.

The nature of GPU hardware, however, has made their use in general purpose computing challenging to all but those with extensive three-dimensional (3D) graphics programming experience. However, as discussed in Section 2 the development of application programming interfaces (APIs) targeted at general purpose scientific computing has reduced this complexity to the point where GPUs are beginning to be accepted as serious tools for the economically efficient acceleration of an extensive range of scientific problems.

In this chapter, we provide a brief overview of GPU hardware and programming techniques and then review the progress that has been made in using GPU hardware to accelerate classical MD simulations of condensed-phase biological systems; we review some of the challenges and limitations that have faced those trying to

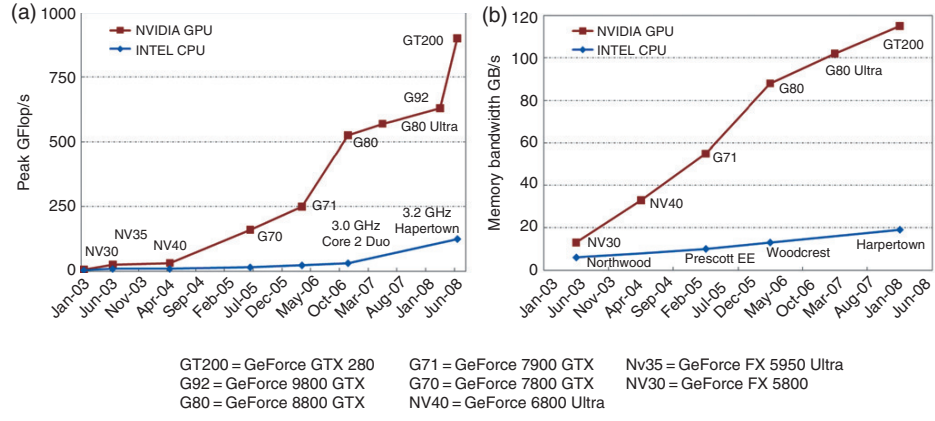

Figure 1 Peak floating-point operations per second (a) and memory bandwidth (b) for Intel CPUs and NVIDIA GPUs. Reproduced from [15].

implement MD algorithms on GPUs, consider performance numbers and validation techniques, and then highlight some recent applications of GPU-accelerated MD. Finally, we comment on the limitations of current GPU MD implementations and what the future may hold for acceleration of MD simulations on GPU hardware.

2. AN OVERVIEW OF GPU PROGRAMMING

2.1 GPU/CPU hardware differences

In order to comprehend where the performance benefits lie and understand the complexity facing programmers wishing to utilize GPUs, it is necessary to compare the underlying nature, and design philosophies, of the GPU with that of the CPU.

Conventional CPUs found in the majority of modern computers, such as those manufactured by Intel and advanced micro devices (AMD), are designed for sequential code execution as per the Von Neumann architecture [16]. While running a program, the CPU fetches instructions and associated data from the computer's random access memory (RAM), decodes it, executes it, and then writes the result back to the RAM. Within the realm of Flynn's taxonomy [17], this would be classified as single instruction, single data (SISD).

Physically, a CPU generally comprises of the following units (Figure 2). The control unit receives the instruction/data pair from RAM during the decoding phase and disseminates out the instruction to give to the arithmetic logic unit (ALU) which is the circuitry that carries out the logical operations on the data. Finally, there are cache units which provide local and fast temporary data storage for the CPU. Historically, performance improvements in sequential execution have been obtained by increasing CPU clock speeds and the introduction of more complex ALUs that perform increasingly composite operations in fewer clock cycles. Additionally, pipelining, which is executing instructions out of order or in parallel while maintaining the overall appearance of sequential execution, has also improved performance (but not calculation speed) by increasing the number of instructions a CPU can execute in a unit amount of time; and larger on chip cache memory is often used to hide latency.

In contrast to the CPU's generality, GPUs (Figure 2) have been designed to facilitate the display of 3D graphics by performing large numbers of floating-

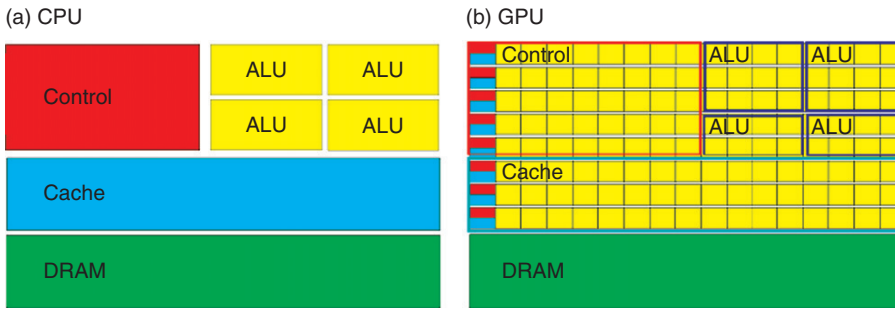

Figure 2 Abstraction contrasting CPU and GPU design. Adapted from [18].

point operations per video frame: they are essentially specialized numeric computing engines. The dominant strategy adopted by the graphics industry to meet this requirement has been to maximize the throughput of a massive number of parallel threads which can all access the RAM on the GPU board. Herein lies the key difference with CPUs: the same operation can be carried out on different parts of the input data within the GPU's memory by an army of individual threads concurrently. Within Flynn's taxonomy, this falls into the single instruction, multiple data (SIMD) category.

A GPU has a hierarchical structure composed of multiple streaming multiprocessors (SMs) which in turn consist of sub units of streaming processors. Memory is also hierarchical, maintaining an approximately constant size to speed ratio; all SMs share the same device global memory which is large, but relatively slow. Smaller, lower latency, on-chip memory which is local to each SM and available to all streaming processors within that SM is provided and even faster register-like memory is present on each streaming processor. A read-only cache of the device global memory is available to each SM in the form of a texture cache. Physically, GPUs have a much larger number of ALUs than a CPU, but the ALUs are not as complex as the ones found in a CPU. The GPU's clock speed is normally about half that of a contemporary CPU's; however, GPUs typically have an order of magnitude larger memory bandwidth to their onboard device global memory.

2.2 The emergence of GPU programming languages

The spectrum of GPU accessibility for scientific use has two extremes. Prior to the development of current general purpose GPU programming models by the major GPU hardware manufacturers, heroic attempts [19] had been made by pioneers in the field in hijacking graphic specific APIs, such as OpenGL, and using them as vehicles for carrying out general purpose calculations. However, development was time consuming and essentially hardware specific. At the other extreme, a compiler should exist which can compile existing scientific code for execution on GPUs without the scientist having to consider the underlying nature of the hardware one is calculating on.

At present, we are somewhere in-between these points; the barrier to utilizing GPU hardware for general purpose computation has been reduced by the introduction of general purpose GPU programming models such as NVIDIA's Compute Unified Device Architecture (CUDA) [15] and AMD's Stream [20]. However, algorithmic paradigm shifts are often required in existing codes to maximize such performance offered by the massively parallel GPU hardware.

The CUDA programming model from NVIDIA appears to be the most mature and widespread in scientific applications at this moment in time, hence the discussion here will focus on specifics pertaining to it. CUDA, a C-like programming language, enables code to run concurrently on the CPU and GPU, with the assumption that the numerically intensive parts of a program will be executed on the GPU and remaining sections, which are perhaps not suited to the GPU, remain executing on the CPU. A mechanism is provided for the two parts of the running code to communicate with each other.

CUDA abstracts the hierarchical GPU hardware structure outlined, into a programming framework, requiring the coder to write in an intrinsically parallel fashion. The small numerically intensive subroutines of code that run specifically on the GPU are termed kernels. These are executed in blocks where each block contains multiple instances of the kernel, termed threads.

This partitioning enables the following (CUDA runtime mediated) physical mapping onto the GPU hardware: each block is run on an individual MP with the number of threads determined by the number of physical SPs within the MP. As a result, only threads within the same block can synchronize with each other. This block-based parallelism and the need to keep all SM units busy in order to achieve efficient performance lead to a number of nontrivial programming considerations.

2.3 GPU programming considerations

A key strategy in improving wall clock time to scientific problem solution is recasting an algorithm in a way that makes it computationally palatable for the nature of the hardware that it is being executed on; an algorithm that performs poorly on a CPU may perform many orders of magnitude better on a GPU and vice versa. However, when dealing with scientific problems, it is essential that alternative approaches to solving the underlying physics yield the same solution, albeit via different paths. It is very tempting given the architectural differences of GPU hardware to change the nature of the problem being solved without a thorough understanding of the implications this has on the scientific results.

General strategies when developing efficient algorithms on GPUs include the following:

1. Ensure that host-to-device communication during a calculation is kept to a minimum. For example, one should ensure that as much of the calculation remains on the GPU as possible. Ferrying data back and forth between the GPU and the host machine is costly due to the latency of the PCIe bus, hence if one is storing atomic coordinates on the host's memory, then the GPU is going to be idle while it is waiting for an updated set to arrive. The above holds within the GPU as well. A corollary to this is that very often it is more efficient to recalculate an existing result on the GPU again, rather than fetch it from a nonlocal location.
2. Accuracy issues that arise from hardware single precision (SP) limitations need to be controlled in a way that is acceptable to the scientific algorithm being simulated. Approaches to this include sorting floats by size prior to addition and making careful use of double precision (DP) where needed [15].
3. Recasting the problem in a vector fashion that groups data that will be operated on in the same way allows for maximizing the efficiency of the SPs.

It should be clear from the above discussion that while GPUs offer an attractive price performance ratio, there are significant hurdles to utilizing them efficiently. Indeed, in some cases, the development costs of GPU-specific code may negate the cost/performance benefits.

3. GPU-BASED IMPLEMENTATIONS OF CLASSICAL MOLECULAR DYNAMICS

As illustrated in the previous section, GPUs have come a long way in terms of their ease of use for general purpose computing. In the last four years, beginning in 2006, NVIDIA's CUDA and ATI's Stream APIs have made programming GPUs significantly easier and the addition of DP hardware in NVIDIA's GT200 line and ATI's FireStream series has facilitated effective implementation of MD algorithms. Due to the reasons discussed above, GPUs are still significantly more complex to program than traditional CPUs. However, the potential cost/performance benefit makes them enticing development platforms. It is only very recently, however, that the use of GPUs for MD simulations has begun to mature to the point where fully featured production MD codes have appeared. The lure of very high performance improvements for minimal cost has influenced early attempts at accelerating MD on GPUs. As we see below, the race to develop MD codes on this "new" hardware has led many to take inappropriate or untested approximations rather than taking the time to address the shortcomings of GPUs. It is also very difficult to compare successes and performance between implementations since a number of manuscripts show only speedups of small parts of the code or comparison against very different types of simulations. A detailed look at what appears, at first sight, to be a very crowded and successful field uncovers only a few select codes that could be considered production ready. In this section, we provide an overview of the peer-reviewed literature on GPU-based MD along with a discussion of these production ready codes.

3.1 Early GPU-based MD code development

In what was arguably the first published implementation of GPU-accelerated MD, Yang et al. [19] reported an algorithm designed for MD simulation of thermal conductivity. This work was prior to the release of the CUDA and Stream APIs and hence the authors were forced to implement their algorithm directly in OpenGL [21]. Using an NVIDIA GeForce 7800 GTX, they observed performance improvements of between 10 and 11 times that of a single Intel Pentium 3.0 GHz processor. While an impressive proof of concept, the Yang et al. implementation was very simplistic containing just Lennard–Jones interactions and a neighbor list that was constructed to remain static over the course of the simulation. It thus lacked many of the important features, such as covalent terms, short- and long-range electrostatics, thermostats, barostats, neighbor list updates, and restraints needed for MD of biological systems. Nevertheless, this pioneering study demonstrated that implementing an MD code on GPUs was feasible.

The advent of the CUDA and Stream programming APIs made programming GPUs significantly easier and brought with them an explosion of GPU MD implementations. Most early implementations of MD on GPUs are characterized by an exploration of the field with the development of codes and GPU-specific algorithms focused on simplistic, artificial, or very specific model problems rather than the application of GPUs to "real-world" production MD simulations.

The first apparent MD implementation to use CUDA was by Liu et al. [22]. Like Yang et al., they too chose to implement just a simplistic van der Waals potential allowing them to avoid all of the complexities inherent in production MD simulations of condensed-phase systems. Unlike Yang, Liu et al. recomputed their neighbor list periodically providing the first example of a neighbor list update for MD on GPUs.

Stone et al. [23] published a lengthy discussion on the implementation of a series of target algorithms for molecular modeling computations, including techniques for direct Coulomb summation for calculating charge–charge interactions within a cutoff. They also discussed possible techniques for evaluation of forces in MD, providing the first mention of a combined treatment of direct space van der Waals and electrostatics in a GPU implementation. Their implementation, however, did not include any actual MD but instead focused on the more simplistic applications of ion placement and the calculation of time-averaged Coulomb potentials in the vicinity of a simulated system. While providing an example of how Coulomb interactions can be accelerated with GPUs and laying the groundwork for developing an experimental GPU-accelerated version of NAMD [24], the example applications are of limited interest for conducting production MD simulations.

Following on the heels of Yang et al., a number of groups begun implementing their own MD codes on GPUs although most were still simply proof-of-concept prototypes with limited applicability for production MD calculations. For example, van Meel et al. [25] implemented a cell-based list algorithm for neighbor list updates but still only applied this to simple van der Waals fluids while Rapaport [26] provided a more detailed look at neighbor list approaches for simple van der Waals potentials. Anderson et al. [27] were the first to include the calculation of covalent terms, adding GPU computation of van der Waals and harmonic bond potentials to their HOOMD code in order to study nonionic liquids. They also included integrators and neighbor lists in their implementation; however, while the HOOMD GPU implementation went a step closer to a full MD implementation, it still neglected most of the complexities including both short- and long-range electrostatics, angle terms, torsion terms, and constraints required for simulating condensed-phase systems.

Davis et al. [28] used a simple truncated electrostatic model to carry out simulations of liquid water. Their approach was similar to Anderson but also included angle and short-range electrostatic terms. While a demonstration of a condensed-phase simulation, the approach used was still extremely restrictive and of limited use in real-world applications.

These early GPU-based MD implementations are characterized by significantly oversimplifying the mathematics in order to make implementation on a GPU easier, neglecting, for example, electrostatics, covalent terms, and heterogenous solutions. This has resulted in a large number of GPU implementations being published but none with any applicability to "real-world" production MD simulations. It is only within the last year (2009/2010) that useful GPU implementations of MD have started to appear.

3.2 Production GPU-based MD codes

The features typically necessary for a condensed-phase production MD code for biological simulations are explicit and implicit solvent implementations, correct treatment of long-range electrostatics, support for different statistical ensembles (NVT, NVE and NPT), thermostats, restraints, constraints, and integration algorithms. At the time of writing, there are only three published MD GPU implementations that could be considered production quality codes. These are the ACEMD code of Harvey et al. [29], the OpenMM library of Friedrichs et al. [30], and NAMD of Phillips et al. [24], although other independent implementations such as support for generalized Born implicit solvation in AMBER 10 [10] (http://ambermd.org/gpus) and support for explicit solvent PME calculations in AMBER 11 [31] are available but have not yet been published.

The ACEMD package by Harvey et al. could be considered the first GPU-accelerated fully featured condensed-phase MD engine [29]. This program includes support for periodic boundaries and more importantly both short- and long-range electrostatics using a smooth particle mesh Ewald (PME) approach [32–34]. The OpenMM library initially only implemented the implicit solvent generalized Born model on small- and medium-sized systems using direct summation of nonbonded terms [30]; Eastman and Pande further improved the OpenMM library and adapted it to explicit solvent simulation [35] although initially using reaction field methods instead of a full treatment of long-range electrostatics. Additionally, a GPU-accelerated version of GROMACS has been developed which works via links to the OpenMM library. GPU acceleration of explicit solvent calculations are also available in NAMD v2.7b2, although acceleration is limited since only the direct space nonbond interactions are calculated on the GPU at present, necessitating a synchronization between GPU and CPU memory on every time step [24]. A comparison of the key features of production MD codes, at the time of writing, is listed in Table 2. From a functionality perspective, at the time of writing, AMBER 11 includes the broadest set of features, capable of running implicit and explicit solvent simulations in all three ensembles with flexible restraints on any atoms as well as allowing the use of multiple precision models although it only supports a single GPU per MD simulation at present. Some of the other codes do not include all of the key features for MD simulation such as pressure coupling and implicit solvent models although this will almost certainly change in the future. The NAMD implementation is CPU centric, focusing on running MD in a multiple node, multiple GPU environment, whereas others implement all MD features on the GPU and strive to optimize MD performance on a single GPU or multiple GPUs on a single node. We note that of all the production MD codes available OpenMM is the only one to support both NVIDIA and ATI GPUs; the others are developed just for NVIDIA GPUs. We also note that ACEMD and AMBER are commercial products, whereas the others are available under various open-source licensing models.

Table 2 Key feature comparison between the GPU-accelerated MD codes

Code	Simulation implementation	GPU acceleration	Multiple GPU support	GPU type	Licensing model
ACEMD	Explicit solvent, PME, NVE, NVT, SHAKE	All features	Three GPUs at present	NVIDIA	Commercial
OpenMM[a]	Explicit solvent, implicit solvent (GB), PME, NVE, NVT, SHAKE	All features	Single GPU at present	ATI/NVIDIA	Free, open source
NAMD	Explicit solvent, PME, NVE, NVT, NPT, SHAKE, Restraint	Direct space on nonbonded interactions only	Multiple GPUs on multiple nodes, but scalability bottlenecked by internode communication	NVIDIA	Free, open source
AMBER11 (PMEMD)	Explicit solvent, implicit solvent (3 GB variants), PME, NVE, NVT, NPT, SHAKE, Restraint	All features	Single GPU at present	NVIDIA	Commercial (source available)

[a] GROMACS has been implemented with OpenMM.

4. PERFORMANCE AND ACCURACY

4.1 Performance and scaling

The performance of MD simulations on modern clusters and supercomputers is currently limited by the communication bottlenecks that occur due to the significant imbalances that exist between CPU speeds and hardware interconnects. The use of GPUs does nothing to alleviate this and indeed actually exacerbates it by making an individual node faster and thus increasing the amount of communication per unit of time that is required between nodes. For this reason, GPU-accelerated MD does not offer the ability to run substantially longer MD simulations than are currently feasible on the best supercomputer hardware, nor does it provide a convincing case for the construction of large clusters of GPUs; however, what it does offer is the ability to run substantially more sampling on a workstation or single node for minimal cost. The huge performance gap that exists between cluster interconnects and GPUs has meant that the majority of implementations have focused on utilizing just a single GPU (OpenMM, AMBER) or multiple GPUs within a single node (ACEMD). Only NAMD has attempted to utilize multiple nodes but with success that is largely due to simulating very large systems and not attempting to optimize single-node performance, thus requiring large numbers of GPUs to achieve only modest speedups and negating many of the cost/performance benefit arguments. Thus the benefit of GPUs to condensed-phase MD should be seen in the concept of condensing small (2–8 node) clusters into single workstations for a fraction of the cost rather than providing a way to run hundreds of microseconds of MD per day on large clusters of GPUs.

A fair comparison of performance across current implementations is very difficult since it is almost impossible to run identical simulations in different programs, and indeed even within the same program it is not always possible to make a fair comparison since additional approximations are often made to the GPU implementation in the desire to achieve larger speedups without considering such approaches on the CPU. There are also numerous situations where people compare the performance of individual kernels, such as the Coulomb sum, rather than the complete implementation. Indeed a careful look at the current literature finds speedups ranging from 7 to 700+. To understand why such numbers might be misleading, consider, for example, the performance reported by Davis et al. [28] in which they compare simulations of various boxes of water with their GPU implementation against that of the CHARMM [36] code. They claim on average to be 7 $\times$ faster than CHARMM on a single CPU but at no point in their paper mention the version of CHARMM used, the compilers used, or even the settings used in the CHARMM code. It should be noted that, largely for historical reasons, the use of default settings in CHARMM tends to give very poor performance. There are then of course multiple optimizations that can be made on the GPU due to the simplicity of the water model. The first is the use of cubic boxes which can benefit vectorization on the GPU, for codes supporting PME it also provides more optimum fast fourier transform (FFT) performance. The second is the use of the SPC/Fw water

model [37] which avoids the complexities of doing SHAKE-based constraints on the GPU. Finally, the use of a pure water box means that all molecules are essentially identical. This allows one to hard code all of the various parameters, since all bonds are identical, all oxygen charges are identical, etc., and thus avoid the additional costs associated with doing such lookups on the GPU. For these reasons, the performance and speedups quoted for various GPU implementations should typically be considered an upper bound on the performance achievable.

Additionally, many factors determine the performance of GPU-accelerated MD codes. Implicit solvent simulations in general show much greater performance boosts over explicit solvent simulation due to the reduced complexities of the underlying algorithm. Specifics include avoiding the need for FFTs and the use of infinite cutoffs which in turn remove the complexity of maintaining a neighbor list. Friedrichs et al. [30] reported more than 60-fold speedup between their single-precision OpenMM code and presumably AMBER 9's DP Sander implementation for systems of 600 atoms and more than two orders of magnitude speedup for systems of 1200 atoms in OpenMM implicit solvent simulations [30]. Similar speedup has been observed in direct comparisons between AMBER's PMEMD code running on 2.8 GHz Intel E5462 CPUs and NVIDIA C1060 Tesla cards [38,39]. Phillips et al. reported up to 7-fold speedup for explicit solvent simulation with GPU-accelerated NAMD, relative to CPU-based NAMD [40], while OpenMM also showed impressive linear performance scaling over system size in its non-PME explicit solvent simulations and at least 19-fold speedup compared to single-CPU MD on simulations of the lambda repressor [30]. However, it is unclear from the OpenMM manuscript if the comparisons are like for like since the AMBER and NAMD numbers appear to be for full PME-based explicit solvent simulations. ACEMD showed that its 3-CPU/3-GPU performance was roughly equivalent to 256-CPU NAMD on the DHFR system and 16-CPU/16-GPU accelerated NAMD on the apoA1 system [29].

4.2 Validation

While the majority of articles describing new GPU MD implementations have focused considerable attention on performance comparison to CPU simulations, there has been very little effort to comprehensively test and validate the implementations, both in terms of actual bugs and in the use of various approximations such as single precision or alternative electrostatic treatments. Since DP has only recently become available on GPUs and because SP still offers a more than 10-fold performance enhancement, all of the GPU-based MD implementations use either single precision or a combination of hybrid single and DP math. Several authors have attempted to provide validation of this and other approximations but often only in a limited fashion while instead preferring to focus on performance. For example, van Meel et al. [25] and Phillips et al. [24] made no mention of validation. Davis et al. [28] simply ran their water box simulations on the CPU and GPU and then provided plots of energy and temperature profiles for the two simulations without any form of statistical analysis.

Liu et al. [22] simply stated that their CUDA version of the code gives output values that are within 0.5% of their C++ version, while Anderson et al. [27] just compare the deviation in atom positions between two runs on different CPU counts and on the GPU.

Harvey et al. [29] attempted more in-depth validation of their code; however, this was still far from comprehensive. For example, they stated in their manuscript that "Potential energies were checked against NAMD values for the initial configuration of a set of systems, ..., in order to verify the correctness of the force calculations by assuring that energies were identical within 6 significant figures." Since scalar potential energies do not convey information about the vector forces, it is unclear how the authors considered this a validation of their force calculations. They provide a table with energy changes in the NVE ensemble per nanosecond per degree of freedom but do not provide any independent simulations for comparison. The authors also state that "... we validate in this section the conservation properties of energy in a NVT simulation ..." which is of little use in validation since energy is not a conserved quantity in the NVT (canonical) ensemble. Additionally, they carried out calculations of Na–Na pair distribution functions using their ACEMD GPU code and also GROMACS on a CPU; however, the lack of consistency in the simulation parameters between GPU and CPU and the clear lack of convergence in the results mean that the validation is qualitative at best.

Friedrichs et al. [30] attempted to validate their OpenMM implementation by simply examining energy conservation for simulations of the lambda repressor and stating, although as with Harvey et al. not providing the numbers in the table to ease comparison, that this compares favorably with other DP CPU implementations.

The push to highlight performance on GPUs has meant that not one of the currently published papers on GPU implementations of MD actually provide any validation of the approximations made in terms of statistical mechanical properties. For example, one could include showing that converged simulations run on a GPU and CPU give identical radial distribution functions, order parameters, and residue dipolar couples to name but a few possible tests.

5. APPLICATIONS

While a significant number of papers published describe GPU implementations of MD, a review of the literature reveals very few cited uses of these codes in "real-world" simulations. Indeed only Pande et al. have such papers published at the time of writing. This serves to underscore the nascent nature of this field.

5.1 Protein folding

In the only published examples of the use of GPU-accelerated bio-MD simulations, Pande et al. have used the OpenMM library to study protein folding in

implicit solvent [41]. This work studied the folding pathways of a three-stranded beta-sheet fragment derived from the Hpin1 WW domain (Fip35) [41] and the 39 residue protein NTL9 [42]. The estimated folding timescale of Fip35 experimentally is ~13 ms. With an average performance of 80–200 ns/day on a single GPU, for this 544-atom protein fragment and utilizing the Folding@Home distributed computing network [43], they were able to generate thousands of independent trajectories totaling over 2.73 ms of ensemble-averaged results, with an average length of 207 ns per trajectory and with some trajectories of greater than 3 ms in length allowing a direct exploration of the folding landscape. Similar trajectory lengths were calculated for the NTL9 (922 atom) case. Additionally, Harvey and De Fabritiis performed a 1 ms explicit solvent MD simulation of the villin headpiece to probe its folding kinetics as part of their ACEMD benchmark results and achieved 66 ns/day on a three-GPU-equipped workstation [29]. These studies have demonstrated the significance of GPU-accelerated MD implementations in helping researchers use personal workstations to reach simulation timescales that would typically only be possible using large clusters and obtain ensemble-averaged results that provide sampling timeframes comparable to experiment. This potentially opens the door to studying a whole range of relevant biological events without requiring access to large-scale supercomputer facilities.

6. CONCLUSIONS AND FUTURE DIRECTIONS

It should be clear from this chapter that the field of GPU acceleration of condensed-phase biological MD simulations is still in its infancy. Initial work in the field concentrated on artificially simplistic models and it is only recently that production quality MD codes have been developed that can make effective use of this technology. The pressure to achieve maximum performance has led to a number of shortcuts and approximations being made, many without any real validation or rigorous study. What initially appears to be an established and extremely active field actually, upon scraping the surface, consists of only a few select codes which could be considered to be production ready and even less examples of "real-world" use. However, the current cost benefits of GPUs are enticing and this is driving both code and hardware development.

In a few short years, GPU-based MD codes have evolved from proof-of-concept prototypes to production-level software packages. Despite the substantial progress made in the code development, the difficulty in programming GPU devices still persists, forcing approximations to be made to circumvent some of the limitations of GPU hardware. However, NVIDIA's recently released Fermi [44] architecture and the accompanying CUDA 3.0 library [15] for the first time provides features such as full support for DP and error-correcting memory along with a more versatile FFT implementation that many consider vital to effective use of GPUs for MD simulations. Given this, a number of established groups in the biological MD field are in the process of developing GPU-accelerated versions of

their software. This will bring more competition to the field and hopefully with it a better focus on extensive validation of the approximations made.

It is anticipated that with the release of GPU versions of widely used MD codes the use of GPUs in research involving MD will likely increase exponentially over the coming years assuming that developers can demonstrate the credibility of these implementations to the same degree to which CPU implementations have been subjected over the years.

ACKNOWLEDGMENTS

This work was supported in part by grant 09-LR-06-117792-WALR from the University of California Lab Fees program and grant XFT-8-88509-01/DE-AC36-99GO10337 from the Department of Energy to RCW.

REFERENCES

1. McCammon, J.A., Gelin, B.R., Karplus, M. Dynamics of folded proteins. Nature 1977, 267, 585–90.
2. Duan, Y., Kollman, P.A. Pathways to a protein folding intermediate observed in a 1-microsecond simulation in aqueous solution. Science 1998, 282, 740–4.
3. Yeh, I., Hummer, G. Peptide loop-closure kinetics from microsecond molecular dynamics simulations in explicit solvent. J. Am. Chem. Soc. 2002, 124, 6563–8.
4. Klepeis, J.L., Lindorff-Larsen, K., Dror, R.O., Shaw, D.E. Long-timescale molecular dynamics simulations of protein structure and function. Curr. Opin. Struct. Biol. 2009, 19, 120–7.
5. Sanbonmatsu, K.Y., Joseph, S., Tung, C. Simulating movement of tRNA into the ribosome during decoding. Proc. Natl. Acad. Sci. USA 2005, 102, 15854–9.
6. Freddolino, P.L., Arkhipov, A.S., Larson, S.B., Mcpherson, A., Schul-ten, K. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. Structure 2006, 14, 437–49.
7. Bakker, A.F., Gilmer, G.H., Grabow, M.H., Thompson, K. A special purpose computer for molecular dynamics calculations. J. Comput. Phys. 1990, 90, 313–35.
8. Fine, R., Dimmler, G., Levinthal, C. FASTRUN: A special purpose, hardwired computer for molecular simulation. Protein Struct. Funct. Genet. 1991, 11, 242–53.
9. Susukita, R., Ebisuzaki, T., Elmegreen, B.G., Furusawa, H., Kato, K., Kawai, A., Kobayashi, Y., Koishi, T., McNiven, G.D., Narumi, T., Yasuoka, K. Hardware accelerator for molecular dynamics: MDGRAPE-2. Comput. Phys. Commun. 2003, 155, 115–31.
10. Case, D.A., Darden, T.A., Cheatham, T.E., Simmerling, C.L., Wang, J., Duke, R.E., Luo, R., Crowley, M., Walker, R.C., Zhang, W., Merz, K.M., Wang, B., Hayik, S., Roitberg, A., Seabra, G., Kolossvary, I., Wong, K.F., Paesani, F., Vanicek, J., Wu, X., Brozell, S.R., Steinbrecher, T., Gohlke, H., Yang, L., Tan, C., Mongan, J., Hornak, V., Cui, G., Mathews, D.H., Seetin, M.G., Sagui, C., Babin, V., Kollman, P.A., AMBER 10, University of California, San Francisco, 2008.
11. Case, D.A., Cheatham, T.E., Darden, T., Gohlke, H., Luo, R., Merz, K.M., Onufriev, A., Simmerling, C., Wang, B., Woods, R.J. The amber biomolecular simulation programs. J. Comput. Chem. 2005, 26, 1668–88.
12. Yuri, N. Performance analysis of clearspeed's CSX600 interconnects, in Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium, pp. 203–10.
13. Shaw, D.E., Deneroff, M.M., Dror, R.O., Kuskin, J.S., Larson, R.H., Salmon, J.K., Young, C., Batson, B., Bowers, K.J., Chao, J.C., Eastwood, M.P., Gagliardo, J., Grossman, J.P., Ho, R.C., Ierardi, D.J., Kolossváry, I., Klepeis, J.L., Layman, T., Mcleavey, C., Moraes, M.A., Mueller, R., Priest, E.C., Shan, Y., Spengler, J., Theobald, M., Towles, B., Wang, S.C. Anton, a special-purpose machine for molecular dynamics simulation. SIGARCH Comput. Archit. News 2007, 35, 1–12.
14. Narumi, T., Ohno, Y., Noriyuk, F., Okimoto, N., Suenaga, A., Yanai, R., Taiji, M. In From Computational Biophysics to Systems Biology: A High-Speed Special-Purpose Computer for Molecular

Dynamics Simulations: MDGRAPE-3 (eds J. Meinke, O. Zimmermann, S. Mohanty and U.H.E. Hansmann) J. von Neumann Institute for Computing, Jülich, 2006, pp. 29–36.
15. NVIDIA: Santa Clara, CA, CUDA Programming Guide, http://developer.download.nvidia.com/compute/cuda/30/toolkit/docs/NVIDIACUDAProgrammingGuide3.0.pdf (Accessed March 6, 2010)
16. von Neumann, J. First draft of a report on the EDVAC. IEEE Ann. Hist. Comput. 1993, 15, 27–75.
17. Flynn, M.J., Some computer organizations and their effectiveness. IEEE Trans. Comput. 1972, C-21, 948–60.
18. Kirk, D.B., Hwu, W.W. Programming Massively Parallel Processors, Morgan Kaufmann Publishers, Burlington, 2010.
19. Yang, J., Wang, Y., Chen, Y. GPU accelerated molecular dynamics simulation of thermal conductivities. J. Comput. Phys. 2007, 221, 799–804.
20. AMD: Sunnyvale, CA, ATI, www.amd.com/stream (Accessed March 14, 2010)
21. Woo, M., Neider, J., Davis, T., Shreiner, D. OpenGL Programming Guide: The Official Guide to Learning OpenGL, version 1.2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999.
22. Liu, W., Schmidt, B., Voss, G., Müller-Wittig, W. In High Performance Computing—HiPC 2007: Lecture Notes in Computer Science (eds S. Aluru, M. Parashar, R. Badrinath and V.K. Prasanna), Vol. 4873, Springer, Berlin/Heidelberg, 2007, pp. 185–96.
23. Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., Schulten, K. Accelerating molecular modeling applications with graphics processors. J. Comput. Chem. 2007, 28, 2618–40.
24. Phillips, J.C., Stone, J.E., Schulten, K. Adapting a message-driven parallel application to gpu-accelerated clusters, In SC '08: Proceedings of the 2008 ACM/IEEE conference on Super computing, 1–9, IEEE Press, Piscataway, NJ, USA, 2008.
25. van Meel, J.A., Arnold, A., Frenkel, D., Portegies Zwart, S.F., Belleman, R.G. Harvesting graphics power for MD simulations. Mol. Simulat. 2008, 34, 259–66.
26. Rapaport, D.C. Enhanced molecular dynamics performance with a programmable graphics processor, arXiv Physics, 2009, arXiv:0911.5631v1
27. Anderson, J.A., Lorenz, C.D., Travesset, A. General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Comput. Phys. 2008, 227, 5342–59.
28. Davis, J., Ozsoy, A., Patel, S., Taufer, M. Towards Large-Scale Molecular Dynamics Simulations on Graphics Processors, Springer, Berlin/Heidelberg, 2009.
29. Harvey, M.J., Giupponi, G., De Fabritiis, G. ACEMD: Accelerating biomolecular dynamics in the microsecond time scale. J. Chem. Theory Comput. 2009, 5, 1632–9.
30. Friedrichs, M.S., Eastman, P., Vaidyanathan, V., Houston, M., Le Grand, S., Beberg, A.L., Ensign, D. L., Bruns, C.M., Pande, V.S. Accelerating molecular dynamic simulation on graphics processing units. J. Comput. Chem. 2009, 30, 864–72.
31. Case, D.A., Darden, T.A., Cheatham, T.E.III, Simmerling, C.L., Wang, J., Duke, R.E., Luo, R., Crowley, M., Walker, R.C., Williamson, M.J., Zhang, W., Merz, K.M., Wang, B., Hayik, S., Roitberg, A., Seabra, G., Kolossváry, I., Wong, K.F., Paesani, F., Vanicek, J., Wu, X., Brozell, S.R., Steinbrecher, T., Gohlke, H., Yang, L., Tan, C., Mongan, J., Hornak, V., Cui, G., Mathews, D.H., Seetin, M.G., Sagui, C., Babin, V., Kollman, P.A. Amber 11, Technical report, University of California, San Francisco, 2010.
32. Darden, T., York, D., Pedersen, L. Particle mesh ewald: An Nlog(N) method for ewald sums in large systems. J. Chem. Phys. 1993, 98, 10089–92.
33. Essmann, U., Perera, L., Berkowitz, M.L., Darden, T., Lee, H., Pedersen, L.G. A smooth particle mesh Ewald method. J. Chem. Phys. 1995, 103, 8577–93.
34. Harvey, M.J., De Fabritiis, G. An implementation of the smooth particle mesh Ewald method on GPU hardware. J. Chem. Theory Comput. 2009, 5, 2371–7.
35. Eastman, P., Pande, V.S. Efficient nonbonded interactions for molecular dynamics on a graphics processing unit. J. Comput. Chem. 2010, 31, 1268–72.
36. Brooks, B.R., Bruccoleri, R.E., Olafson, B.D., States, D.J., Swaminathan, S., Karplus, M. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. J. Comput. Chem. 1983, 4, 187–217.
37. Wu, Y., Tepper, H.L., Voth, G.A. Flexible simple point-charge water model with improved liquid-state properties. J. Chem. Phys. 2006, 124, 24503.

38. Grand, S.L., Goetz, A.W., Xu, D., Poole, D., Walker, R.C. Accelerating of amber generalized born calculations using nvidia graphics processing units. 2010 (in preparation).
39. Grand, S.L., Goetz, A.W., Xu, D., Poole, D., Walker, R.C. Achieving high performance in amber PME simulations using graphics processing units without compromising accuracy. 2010 (in preparation).
40. Phillips, J.C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R.D., Kale, L., Schulten, K. Scalable molecular dynamics with NAMD. J. Comput. Chem. 2005, 26, 1781–802.
41. Ensign, D.L., Pande, V.S. The Fip35 WW domain folds with structural and mechanistic heterogeneity in molecular dynamics simulations. Biophys. J. 2009, 96, L53–5.
42. Voelz, V.A., Bowman, G.R., Beauchamp, K., Pande, V.S. Molecular simulation of ab initio protein folding for a millisecond folder NTL9(1-39). J. Am. Chem. Soc. 2010, 132, 1526–8.
43. Shirts, M., Pande, V.S. Computing: Screen savers of the world unite! Science 2000, 290, 1903–4.
44. NVIDIA Corporation Next generation CUDA compute architecture: Fermi, 2009.