A Survey of Computational Molecular Science using Graphics Processing Units

M J Harvey[1] and Gianni De Fabritiis[2]

[1]Information and Communications Technologies, Imperial College London, South Kensington, London, SW7 2AZ, UK
[2]Computational Biochemistry and Biophysics Lab (GRIB-IMIM), Universitat Pompeu Fabra, Barcelona Biomedical Research Park (PRBB), Carrer Doctor Aiguader 88, 08003 Barcelona, Spain

Email: M J Harvey - m.j.harvey@imperial.ac.uk; G De Fabritiis - gianni.defabritiis@upf.edu;

*Corresponding author

Abstract

Computational molecular science is a very computationally-intense discipline, and the use of parallel programming and high performance computers well-established as being necessary to support research activities. Recently, graphical processing units (GPUs) have garnered substantial interest as alternative sources of high performance computing capability. These devices, though capable of very high rates of floating point arithmetic, are also intrinsically highly-parallel processors and their effective exploitation typically requires extensive software re-factoring and development. Here we review the current landscape of GPU hardware and programming models, and provide a snapshot survey of the current state of computational molecular science codes ported to GPUs in order to help domain scientists and software developers understand the potential benefits and drawbacks of this new computing architecture.

1 Introduction

The demand for increasing fidelity and complexity in computer graphics has led to the evolution of graphical output systems from dumb frame-buffers, through dedicated, fixed-function hardware to fully-fledged processors. These graphics processing units (GPUs) are now capable of running complex scientific algorithms with very high rates of floating-point arithmetic, making them an attractive target for scientific computation.

Early attempts to exploit these processors for non-graphics tasks were encumbered by the need to map algorithms onto the graphics-processing pipeline model, to compensate for programming model limitations. [1] In 2007 Nvidia, a major vendor of discrete GPU devices, released the CUDA (Compute Unified Device Architecture) programming toolkit, designed to make their GPU products amenable to general purpose computation. As this was, and remains, supported on their whole range of graphics hardware, the barrier to entry for the prospective GPU programmer was very low. Coupled with the rapid product refresh cycle of the commodity consumer computing market, these devices have remained attractive targets for computation and not been rendered obsolete by Moore's Law increases in CPU performance.

Nevertheless, despite being accessible high-performance processors, the architecture of GPUs is currently radically different from that of CPUs. This makes the porting of existing applications a non-trivial task, often requiring extensive code-refactoring and algorithmic changes. For scientific applications in particular, which may have a very long useful lifetime that far outlasts any one generation of hardware, it is understandable that developers take a conservative approach to adapting to novel platforms.

In this review, we describe the current state of GPU hardware, programming tools and survey the scientific applications that have emerged to exploit them.

2 GPU Hardware

Conventional CPUs are designed to maximise the performance of a single thread of execution and modern devices employ a variety of strategies to minimise instruction latency, for example speculative and out-of-order execution of instructions to exploit instruction-level parallelism (ILP) and large caches to hide the latency of main memory access. This has lead to CPU architectures that, though able to perform well on a wide variety of codes, are very complex.

Floating point arithmetic, an attribute important for numerically intensive computing but of less relevance in many other application spaces, is performed by dedicated hardware within the CPU. In modern CPUs, this is most often accomplished with floating-point units that are able to operate on short vectors of values. Intel and AMD x86 CPUs, for example, can operate on 256bit vector words (4 double-precision or 8 single-precision values, using the AVX instruction set).

GPU processors, in contrast, have been designed for the specific task of render a polygonal 3D scene representation to a rectilinear array of pixels. Because the value of each pixel is often able to be computed with reference to no more than its immediate neighbours, there is a high degree of intrinsic parallelism in

the processing. Furthermore, the algorithms used in rendering are frequently arithmetic intensive and perform limited branching or conditional operation, unlike typical CPU code. This has led to an architecture for GPUs that is designed to execute a large number of independent paths of execution and optimised for overall instruction throughput at the expense of single-instruction performance.

The majority of contemporary high performance GPUs are manufactured by Nvidia and AMD. Although there are architectural differences between the two competing product lines, there are sufficient high-level correspondences for a comparison to be drawn and the essential characteristics described. The GPU processor is a single chip that contains several independent *compute units*. Each compute unit is composed of a an array of *processing elements* the share access to a small directly-addressable *scratchpad memory*. The compute units share a common, high bandwidth link to an external memory. Caching of the main memory is typically limited to read caching, with limited or no cache coherency between compute units. GPUs are discrete peripherals attached to a host computer via a PCI express connection, and have their own memory, physically separate from that connected to the CPU (Figure 1).

3 GPU Programming

At a low level, the execution model for the GPU is one of a serial program that is run in many simultaneous instances which are in turn divided into groups. Each group executes within a single compute unit, and each program instance maps to a particular processing element. The instances within a group are able to communicate and share data with their siblings via the scratchpad memory. A group may contain many more instances than there are physical processing elements, in which case the hardware time slices the execution of the groups.

In current GPUs, each processing element is a simple device that is incapable of an independent flow of execution. Instead, the program execution is managed at the compute unit level, with each individual processing element constrained to operate in lock-step with its peers. A consequence is that if the path of program execution diverges amongst instances, their execution will be serialised and overall performance will be reduced.

Program execution is always initiated by a control program executing on the host CPU system. Once queued to run, the GPU program executes asynchronously, allowing the CPU to perform simultaneous computation. There are two prevailing programming models that expose this view of program execution to the programmer. The first, CUDA [2], is a programming environment proprietary to NVIDIA, while the second, OpenCL [3], is an industry standard promoted by the Khronos Group and provides a model that is

slightly more abstract, allowing it to be mapped not only to NVIDIA and AMD GPUs but also to CPUs (see Table 1). Both CUDA and OpenCL define C-like languages for writing GPU programs, with CUDA including some C++ features (*eg* templating). The CUDA compiler, furthermore, is able to compile host CPU C code and provides language extensions that simplify the task of launching programs. A Fortran compiler, that extends much the host-side benefits of CUDA to that language, is available commercially from the Portland Group [4].

A second, implicit, programming model for GPU programming can be found in OpenACC [5]. In this model, reminiscent of OpenMP [6], existing source code is annotated with directives that the programmer uses to indicate the properties of the associated block of code and whether it should be a candidate for GPU acceleration. The compiler then performs manipulations to emit code for the GPU program and also transparently manages the data movement and GPU execution. In this manner, an existing code base may be retargeted for GPU execution without extensive rewriting, although the success of this depends on the program already expressing a high degree of data parallelism. OpenACC is supported by the CAPS HMPP and Portland Group Accelerator compilers, at this time only for Nvidia GPUs.

4 Scientific Applications for GPUs

In this section we review the current landscape of GPU-accelerated software available in the computational molecular science field. Unless otherwise stated, software packages mentioned should be assumed to require a CUDA-capable NVIDIA GPU. We do not generally quote performance figues as these date rapidly with hardware and software improvements; instead we refer the interested reader to the webpages and publications for the packages of interest. In general, from the differences in GPU and CPU performance characteristics (Table 1 an approximate order-of-magnitude improvement can be expected in comparison to an optimised CPU code.

4.1 Numerical Libraries

Although not specifically within the domain of computational molecular science, numerical libraries are a building block used by many software packages. Perhaps the most popular numerical libraries are the BLAS[1] and LAPACK[2] linear algebra packages. A variety of GPU implementations that offer partial coverage of these libraries have emerged: CUBLAS, included within the NVIDIA CUDA toolkit, provides

[1] http://www.netlib.org/blas
[2] http://www.netlib.org/lapack

BLAS level 1-3 routines. Since CUBLAS has a custom API, BLAS-using applications must be modified before they can be linked to it.

The MAGMA library [7] of Dongarra & co-workers, is a linear algebra library designed for heterogeneous GPU/CPU systems and is able to use the combined resources of all available processors. As well as providing BLAS functions, MAGMA contains one- and two-sided factorisations and linear and eigen/singular value solvers.

The CULA package[3] [8] is a commercial library offering BLAS and a large subset of LAPACK. The CULA library conforms to the conventional Netlib APIs, meaning that existing application require no code-modification to exploit it.

Associated with CUBLAS in the CUDA toolkit is the CUSPARSE library, which provides levels 1-3 BLAS for sparse matrices and a triangular solver along with associated support functions for format conversion. CUSP[4], also provided by Nvidia, is C++ library of generic parallel algorithms for sparse linear algebra and graph computations that has a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems.

The PETSc package [9], a suite of data structures and routines for the scalable (parallel) solution of scientific applications modelled by partial differential equations, is extensively used across a wide variety of disciplines. Recent enhancements to this package have introduced GPU support [10] allowing the Krylov methods, nonlinear solver and integrator components to run unmodified - and transparently - on GPU hardware. This work makes use of the CUSP package, as well as the Thrust CUDA templating library[5].

Turning to Fourier methods, the CUDA CUFFT library provides an FFT library capable of 1,2 and 3 dimensional transforms of complex and real data. Its interface is patterned after that of FFTW [11], although, as with CUBLAS, the API is custom and source-code modification is required to use it.

4.2 Bioinformatics

Perhaps the most computationally-demanding task in the field of bioinformatics is the reconstruction of long nucleotide sequences from the short fragments of sequence read by modern sequencing equipment. This field is experiencing significant growth stemming from rapid innovation in sequencing technology [12]. The computational cost of assembly increases with each generation of sequencer as fragments become both more numerous.

[3] urlhttp://www.culatools.com
[4] http://developer.nvidia.com/cusp
[5] http://research.nvidia.com/news/thrust-cuda-library

There has therefore been substantial activity in accelerating sequence alignment and assembly on GPUs, with the one of the first reported codes, GPU ClustalW [13] by Schmidt and co-workers, an example of an application that pre-dates CUDA and relies on the GLSL graphics shading language [14]. A subsequent re-implementation of ClustalW in CUDA [15] has been demonstrated to match the performance of 32 processor parallel runs of the reference CPU code. Other GPU-accelerated assembly programs include MummerGPU [16].

Error correction methods for large-scale short read assembly have also received attention, with algorithmic development [17] leading to the stand-alone tool CUDA-EC [18] and the DecGPU library [19] that has been interfaced with the popular *de novo* assembly tools Velvet [20] and ABySS [21].

GPUs have also been employed to accelerate genome-wide association studies, for example the BOOST [22] and MDR [23] packages for detecting epistatic (gene-gene) interactions have CUDA re-implementations in GBOOST [24] and MDR-GPU [25] with their respective authors reporting significant speed-ups over their CPU equivalents.

A CUDA implementation of sequence database searching using the Smith-Waterman algorithm is to be found in CUDASW++ [26]. The MEME iterative algorithm for motif discovery in sequences has a CUDA implementation in CUDA-MEME [27]. The popular HMMER sequence search tools have a GPU implementation [28], as does the protein database search tool BLASTP [29].

4.3 Quantum Chemistry

Computational modelling of the electronic structure of many-body systems using approximate solutions of the Schrödinger equation is an established and computationally demanding simulation method. Modern density functional theory (DFT) [30] represents an electronic system as a set of non-interacting electrons moving in an effective potential which accounts for the exchange and correlation interactions. By making the Bohr-Oppenheimer approximation, separation of the motions of nuclei and electrons is made possible, simplifying the wave equation.

Many mature quantum chemistry packages exist, of which Gaussian, MOLCAS, GAMESS(US) and Orca are amongst the most comprehensive and versatile. Parallel programming methods are widely adopted in this field, with many codes able to run efficiently on large cluster machines.

These packages have been developed over many years and have grown complex and feature-rich. The challenges of modifying these large, established code bases to exploit GPUs are significant. Not only may these code have been developed incrementally over decades and use programming models and

parallel-computing abstractions that make interfacing with modern CUDA code difficult, but also the diversity of algorithms and methods in the field means that porting all commonly-used code pathways and functions to use a GPU is a major endeavour.

Consequently, GPU-acceleration has been slow to appear in the major existing codes. The majority of work in the development of GPU implementations of QC algorithms has been reported by Ufimtsev and co-workers [31, 32, 33]. Much of this work has been made commercially available in the Terachem package, which is able to perform restricted and unrestricted Hartree-Fock, DFT, Car-Parrinello MD, coupled quantum mechanics/classical mechanics (QM/MM) and natural bond analaysis (NBO). It is designed for simulation of biomolecules and is able to use multiple GPUs.

A long-standing weakness of GPUs has been that single-precision floating-point rates have far exceeded their peak double-precision performance (by as much as a factor of 8 in consumer models, with high-end cards now operating at half of their single-precision rate). Indeed, the earliest CUDA-capable GPUs has no double-precision capability at all. Because QC calculations rely extensively on double-precision arithmetic to achieve the necessary accuracy, much early work was focused on the use of hybrid and mixed-precision methods

Of the established QC codes, preliminary results have been reported for experimental GPU-acceleration of Quantum Espresso and MOLCAS7. Work on Quantum Espresso,a plane-wave DFT package for nanoscale materials, has been undertaken by a worldwide consortium of groups. Both CUDA and OpenACC (HMPP) approaches have been evaluated. A different approach has been taken with MOLCAS7 [34], with code modification limited to using GPU-accelerated linear algebra libraries [35] for DGEMM matrix operations. For large molecular systems, a modest speed-up has been reported, demonstrating that an "easy-win" may be gained with codes that make heavy use of library functions. GPU implementations of coupled-cluster methods has been reported [36, 37] which are expected[6] to be incorporated into version 4 of the PSI code [38].

The field of GPU quantum chemistry is further reviewed in [39]

4.4 Classical molecular dynamics

Of all the fields of computational molecular science, it is that of classical molecular dynamics (MD) that has arguably benefited most from GPU computing to date. MD, in which a particle system is allowed to evolve according to classical mechanics, is a commonly used methodology for modelling large molecular

[6] Personal communication with J. Hammond, Argonne National Laboratory

systems. A force-field, parametrised to capture the chemical properties of the environment of each type of particle, governs the evolution of the system which proceeds according to classical dynamics in a stepwise manner. At each step of the simulation, the net force on each particle is calculated and the particles' positions updated. To correctly capture the dynamics of the system the timestep of each iteration must lie close to the timescale of the fastest modes of vibration present. For atomic-resolution simulations this timestep is usually less than 5 fs. Because the timescales of biological interest lie in the microsecond to millisecond range, some 9 orders of magnitude higher, it is a significant engineering challenge to produce computer systems able to access this range in an acceptable amount of time. Consequently, several highly-optimised MD applications designed to exploit parallel cluster computers already exist. Developers in this field have often been early-adopters of novel processing hardware (for example CELLMD [40] on the Cell processor [41]) and GPU adoption has been swift.

As with quantum chemistry, the most mature GPU MD packages are those that have been developed specifically for GPUs. Three of these, HOOMD-Blue, ACEMD and OpenMM are quite different in character, having been optimised for different specialisations. HOOMD-Blue [42] is a versatile, scriptable molecular dynamics engine optimised for coarse-grained polymer and liquid systems. It has an extensive, and extensible Python interface and supports a variety of pair potentials and integration schemes including dissipative particle dynamics [43]. ACEMD [44] is a very high-performance package designed for production simulation of biomolecular simulations specifically using Amber and Charmm force-fields. It is also able to exploit multiple GPUs within a single host. OpenMM [45], uniquely, uses OpenCL in preference to CUDA, allowing it to be used on any OpenCL-supported system, including AMD GPUs. It exploits the just-in-time compilation feature of OpenCL to perform run-time generation of optimised kernels. Unlike either HOOMD-Blue or ACEMD, it is not a stand-alone application, but a library, and requires a supporting program framework to operate. A version of Gromacs that uses OpenMM is available.

Many of the existing parallel CPU MD applications have also been modified to exploit GPUs to various extents: LAMMPS [46], a code widely used for nanoscale simulations, is a highly modular code that is easily extensible. Two independently-developed GPU acceleration packages exist (the "GPU" [47] and "USER-CUDA" [48] packages). Gromacs, a package noted for its particularly highly-optimised CPU implementation is able to exploit GPUs using the OpenMM library, as of version 4.5. The DLPOLY code, most popularly used for materials simulations has been successfully ported to GPUs as of version 4. It is parallelised with a mix of MPI, OpenMP and CUDA, allowing an efficient usage of both the CPUs and the GPUs of a HPC cluster. Speedups of $\approx 4\times$ have been reported [49].

The highly-parallel biomolecular simulation code NAMD has also been adapted to use GPUs [50], with a focus on maintaining scalability for very large molecular systems on GPU-equipped clusters. In contrast, and similar to ACEMD, the GPU-adaptation of Amber has focused on achieving high performance on few GPUs, sacrificing scalability for high peak performance. ACEMD and Amber 11 currently provide the highest levels of performance for all-atom MD simulations, achieved by conducting every aspect of the simulation on the GPU, including the particle-mesh Ewald summation for long-range electrostatics [51, 52]. As indicated, GPU use promises to have a profound impact on this field. This is a consequence of the poor strong scaling properties of CPU MD codes: ACEMD, for example, despite not being able to exceed the peak performance attainable with parallel CPU codes is able to achieve a simulation rate on a single GPU that would otherwise require many nodes of an HPC cluster system together with a high-performance interconnecting network. Since adequately long simulations may now be routinely performed on hardware that is cheap in comparison to supercomputers/clusters, the cost of MD simulation dramatically reduced, allowing a qualitative change from a focus on high-performance to high-throughput methods of analysis [53, 54].

The field of molecular mechanics on GPUs is further reviewed in [55, 56]

5 Discussion and Future Prospects

From the preceding survey, it is apparent that GPUs provide a clear performance benefit in several important fields of computational molecular science. For a scientific programmer considering GPU development, however, it is useful to consider the properties of an application that render it particularly amenable to such a conversion.

Current GPUs are characterised by high (single-precision) floating point rates, highly parallel execution and large memory bandwidth to a comparatively small memory. To exploit this hardware, software must generally be explicitly re-written; profound architectural differences mean that a simple recompilation of existing code is not (yet) feasible. Thus, an application well-suited to a GPU is one that performs large amounts of arithmetic on a relatively small working set that can be kept entirely in GPU memory (typically a few gigabytes in size). Ideally, the code will have a steep performance profile *ie* one in which a small footprint in the source code accounts for large fraction of the program runtime. In this way, the amount of code refactoring/redevelopment required to yield a ported code that exhibits good performance is minimised. The inertial effect of having to adapt a large established code-base that has many modes of functionality is clearly seen in the quantum chemistry field where packages have been slow to be adapted,

despite the clear benefits demonstrated by the *de novo* GPU codes. It is also in this case that the benefits of using GPU-accelerated math libraries to minimise code-modification are to be seen.

When considering the merit of porting already parallel applications, the calculus should include the scaling properties of the existing CPU code. It is important to distinguish between strong and weak scaling cases: In the former, parallel efficiency is measured for a fixed size of problem, while in the latter problem size increases with the number of processors , maintaining a constant amount of computation per core. For many real-world scientific problems, an application's strong scaling performance is most important, as the problem size is fixed by the physical properties of the system under study (for example, the number of atoms). If this scaling is poor, the diminishing return of adding more processors to a poorly-scaling job may make the desired level of performance prohibitively expensive - or even impossible - to achieve.

This is the situation for classical molecular dynamics, where poor CPU scaling and small system sizes mean that running sufficiently long simulations within a reasonable amount of time can become computationally expensive, not to say wasteful of resources. GPU codes, by being able to perform the entire calculation at an acceptable level of performance on hardware that is cheap in comparison to a high performance cluster, yields not just a quantitative change, but a qualitative one as molecular dynamics can now in many cases be considered a throughput, rather than performance-limited activity.

For applications that already operate in a throughput regime, where large ensembles of tasks are performed independently, and the important performance metric is the time to overall completion, rather than the performance of any one single task, the argument for investing time in converting to GPU use is more difficult to make, particularly as very large CPU-based systems are readily accessible, while large scale deployments of GPUs remain scarce. This may account for the low uptake of GPUs in the field of molecular docking, for example.

Having established that there is a tangible performance return to be gained from investing the effort to develop a GPU code, a reasonable question is then to ask what software engineering approach is best taken to produce code that is both high-performance and portable to future GPUs and – in the interests of not having to maintain multiple code paths for different architectures – also CPUs.

From recent developments in the microprocessor market, we take the view that something of a convergence between GPU and CPU systems is occurring. Early indications are present in current consumer CPUs from both Intel and AMD that include GPUs on-die. Although in both cases the GPU is very low performance when compared to a stand-alone device, the AMD processor's GPU is fully capable of running OpenCL code. Furthermore, the CPU and GPU components are tightly-integrated, sharing a unified

memory subsystem. It is highly likely that the distinction will continue to blur and future processors will be heterogeneous devices containing a mix of few complex, low-latency cores for high-performance and arrays of simpler cores optimised for high-throughput, with the exact configuration balanced for different market segments.

Indeed, the prevailing opinion of the scientific programming community, as succinctly described in [57] is one in which processor clock speeds and so serial program performance continues to stagnate, but with the number of independent processing cores within a CPU increasing with each generation.

Considering the trends in CPU core design, floating-point vector word sizes are slowly increasing (from 128bit SSE to 256bit AVX, with an instruction set designed to accommodate future extension of word size). This change goes some way to eroding the floating-point advantage of GPUs, especially in conjunction with rising core counts further increasing the aggregate performance of a CPU package. However, as vector word size increases, the chance of compiler being able to automatically extract sufficient instruction-level parallelism from scalar source code to fill the vector word diminishes and programming models that gives the programmer sufficient control to express data parallelism explicitly will be required. At the same time, the higher level parallelism of multiple cores must be addressed.

The OpenCL programming model provides such an abstraction to allow programs expressed within it to map naturally to this style of hardware with multiple levels of parallelism. The Intel OpenCL compiler for example, is able to perform "implicit" vectorisation, in which individual program instances are mapped to the vector lanes of the SSE or AVX word. This blurring of the line between CPU and GPU at a high-level programming level serves to emphasise that increasingly, the architectural differences between the two classes of processor will be become one of degree and not essence.

Although NVIDIA hardware, and the CUDA programming model are close to ubiquitous in the current scientific GPU-computing landscape, we contend that the continued dominance of this combination depends to a large extent on the the unavailability of directly and effectively competing hardware. Future generations of CPU and GPU architecture may provide an effective performance challenge, while – being non-Nvidia devices – unable to exploit CUDA codes. Since the correspondences between OpenCL and the C subset of CUDA are so striking, we contend that a practical conservative approach to engineering a durable GPU code is to develop either directly in OpenCL or use code generation or abstraction tools [58] that allow CUDA and OpenCL to be inter-converted.

Device	NVIDIA Geforce GTX580	AMD Radeon HD6890	Intel Intel E5 Xeon
Compute Units	15	20	8
Scratchpad Memory/CU (kB)	48	32	384 (L1$)
Processing Elements/CU	32	16	1(≤8)
PE Clock (GHz)	1.4	0.85	2.93
PE architecture	Scalar in-order	4-way VLIW in-order	Scalar+vector out-of-order
Peak GFLOPS/device	FMA	FMA	MUL+ADD
sp	1344	2720	375
dp	672	544	188
Peak Memory BW (GB/s)	177	155	50
Scatter/Gather	Yes	Yes	No

Table 1: Comparison of NVIDIA , AMD GPUs and Intel CPUs as OpenCL devices. Xeon processors have no directly-addressable scratchpad memory, but do have L1 cache. Intel's OpenCL implementation is able to treat each Xeon core as a compute unit containing a single vector-capable processing element, or as containing up to 8 scalar PEs, by mapping the OpenCL program instances onto individual vector lanes in the AVX vector word.

6 Conflicts of Interest

The authors declare a financial interest in Acellera Ltd.

Tables

References

1. Elsen E, Vishal V, Houston M, Pande V, Hanrahan P, Darve E: **N-Body Simulations on GPUs**. In *SC2006: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* 2006:188.

2. NVIDIA Corporation: **NVIDIA CUDA Compute Unified Device Architecture Programming Guide**. Tech. rep., NVIDIA Corporation 2011.

3. Khronos Group: **OpenCL 1.2 Specifcation, revision 1.5**. Tech. rep., Khronos Group 2011.

4. Portland Group: **CUDA Fortran Programming Guide and Reference, Release 2011**. Tech. rep., Portland Group 2011.

5. NVIDIA Corporation: **The OpenACC Application Programming Interface, Version 1.0**. Tech. rep., NVIDIA Corporation 2011.

6. OpenMP Architecture Review Board: **OpenMP Application Program Interface version 3.0**. Tech. rep., OpenMP Architecture Review Board 2008.

Figure 1: The Nvidia Tesla M2090 GPU, capable of 665GFLOPs of double-precision floating-point arithmetic in a power envelope of 250W. It is designed to be attached to a 16-line PCI Express version 2.0 interface within a high-performance compute cluster node.

7. Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, Tomov S: **Numerical Linear Algebra on Emerging Architecture: the PLASMA and MAGMA libraries**. *J. of Phys.: Conf. Series* 2009, (180):012037.

8. Humphrey JR, Price DK, Spagnoli KE, Paolini AL, Kelmelis EJ: **CULA: Hybrid GPU Accelerated Linear Algebra Routines**. In *SPIE Defense and Security Symposium (DSS)* 2010.

9. Balay S, Brown J, Buschelman K, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H: **PETSc Web page** 2011. [Http://www.mcs.anl.gov/petsc].

10. Minden V, Smith B, Knepley M: **Preliminary implementation of PETSc using GPUs**. *Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering* 2010.

11. Frigo M, Johnson SG: **The Design and Implementation of FFTW3**. In *Proceedings of the IEEE, Volume 93* 2005:216–231.

12. Schuster SC: **Next-generation sequencing transforms today's biology**. *Nature Methods* 2008, **5**.

13. Liu W, Schmidt B, Voss G, Müller-Wittig W: **GPU-ClustalW: Using Grpahics Hardware to Accelerate Multiple Sequence Alignment**. *Lecture Notes in Computer Science* 2006, **4297**:363–374.

14. Kessenich J, Baldwin D, Rost R: **GLSL: The OpenGL shading Language, rev 59**. Tech. rep., Khronos Group 2005.

15. Liu Y, Schmidt B, Maskell DL: **MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA**. In *20th IEEE International Conference on Application-specific Systems, Architectures and Processors* 2009.

16. Schatz MC, Trapnell C, Delcher AL, Vershney A: **High-throughput sequence alignment using graphics processing units**. *Bioinformatics* 2007, **8**(474).

17. Shi H, Schmidt B, Liu W, Müller-Wittig W: **A parallel algorithm for error correction in high-throughput short-read data on CUDA-eneabled graphics hardware**. *J. Comput. Biol.* 2010, **17**(4):603–15.

18. Shi H, Schmidt B, Liu W, Müller-Wittig W: **Accelerating Error Correction in High-Throughput Short-Read DNA Sequencing Data with CUDA**. In *International Workshop on High Performance Computational Biology (HiCOMB 2009)* 2009.

19. Liu Y, Schmidt B, Maskell D: **DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI**. *Bioinformatics* 2011, **12**(85).

20. Zerbino D, Birney E: **Velvet: algorithms for de novo short read assembly using de Bruijn graphs**. *Genome Res* 2008, **18**(5):821–829.

21. Simpson J, Wong K, Jackman S, Schein J, Jones S, Birol I: **ABySS: a parallel assembler for short read sequence data**. *Genome Res* 2009, **19**(6):1117–1123.

22. Wan X, Yang C, Yang Q, Xue H, Fan X, Tang NLS, Yu W: **BOOST: A fast approach to detecting gene-gene interaction in genome-wide case-control studies**. *Am. J. Hum. Genet.* 2010, **87**(3):325–340.

23. Ritchie MD: **Multifactor dimensionality reduction reveals high-order interactions among estrogen metabolism genes in sporadic breast cancer**. *Am. J. Hum. Genet.* 2001, **69**:138–147.

24. Yung LS, Yang C, Wan X, Yu W: **GBOOST : A GPU-based tool for detecting gene-gene interactions in genome-wide case control studies**. *Bioinformatics* 2011, **27**(21):2936–2943.

25. Greene CS, Sinnott-Armstrong NA, Himmelstein DS, Park PJ, Moore JH, HArris T Brent: **Multifactor dimensionality reduction for gprahics processing units enables genome-wide testing of epistasis in sporadic ALS**. *Bioinformatics* 2010, **26**(5).

26. Liu Y, Maskell DL, Schmidt B: **CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units**. *BMC Research Notes* 2009, **2**(73).

27. Liu B Y amd Schmidt, Liu W, Maskell D: **CUDA-MEME: Accelerating Motif Discovery in Biological Sequences Using CUDA-enabled Graphics Processing Units**. *Pattern Recognition Letters* 2009, **31**(14).

28. Walter JP, Balu V, Kompalli S, Chaudhary V: **Evaluating the use of GPUs in Liver Image Segmentation and HMMER Database Searches**. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)* 2009.

29. Liu W, Schmidt B, Müller-Wittig W: **CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware**. *IEEE/ACM Trans Comput. Biol. Bionform.* 2011, **in press**.

30. **Self-consistent equations including exchange and correlation effects**. *Phys. Rev.* 1965, **140**(4):A1133–A1138.

31. Ufimtsev I, Martinez T: **Graphical Processing Units for Quantum Chemistry**. *Computing in Science Engineering* 2008, **10**(6):26 –34.

32. Luehr N, Ufimsev IS, Martinez TJ: **Dynamic Precision for Electron Repulsion Integral Evaluation on Graphical Processing Units (GPUs)**. *J. Chem. Theor. and Comp.* 2011, **7**:949–954.

33. Isborn CM, Luehr N, Ufimsev IS, Martinez TJ: **Excited-State Electronic Stucture with Configuration Interaction Singles and Tamm-Dancoff Time-Dependent Density Functional Theory on Graphical Processing Units**. *J. Chem. Theor. and Comp.* 2011, **7**:1814–1823.

34. Aqullante F, De Vico L, Ferré N, Ghigo G, Malmqvist PA, Neogrády P, Pedersen TB, Pitonak M, Relher M, Roos BO, Serano-Andrés L, Urban M, Veryazov V, Lindh R: **MOLCAS 7: The Next Generation**. *J. Comput. Chem.* 2010, **31**:224–247.

35. Delcey M, Malmqvist PA, Vancoillie S, Veryazov V: **How does hardware development influence computational chemistry?** In *Poster at 8th European Conference on Computational Chemistry, Satellite Meeting to 3rd EuCheMS Chemistry Congress in Nürnberg* 2010.

36. Hammond J, DePrince E, Hammond JR: **Coupled Cluster Theory on Graphics Processing Units I. The Coupled Cluster Doubles Method**. *J. Chem. Theory. Comput.* 2011, **7**(5):1287–1295.

37. Ma W, Krishnamoorthy S, Villa O, Kowalski K: **GPU-based implementations of the noniterative regularized-CCSD(T) corrections: application to strongs correlated systems**. *J. Chem. Theory. Comput.* 2011, **7**(5):1316–1327.

38. Turney JM, Simmonett AC, Parrish RM, Hohenstein EG, Evangelista FA, Fermann JT, Mintz BJ, Burns LA, Wilke JJ, Abrams ML, Russ NJ, Leininger ML, Janssen CL, Seidl ET, Allen WD, Schaefer HF, King RA, Valeev EF, Sherrill CD, Crawford TD: **Psi4: an open-source ab initio electronic structure program**. *Wiley Interdisciplinary Reviews: Computational Molecular Science* 2012, :n/a–n/a, [http://dx.doi.org/10.1002/wcms.93].

39. Götz AW, Wölfle T, Walker RC: **Quantum Chemistry on Graphics Processing Units**. In *Annual Reports in Computational Chemistry, Volume 6*, Elsevier 2010.

40. De Fabritiis G: **Performance of the Cell Processor for Biomolecular Simulations**. *Comp. Phys. Commun.* 2007, **176**:670.

41. Kahle JA, Day MN, Hofstee HP, Johns CR, Maeurer TR, Shippy D: **Introduction to the Cell Multiprocessor**. Tech. Rep. 4, IBM 2005.

42. Anderson JA, Lorenz CD, Travesset A: **General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units**. *J. Comp. Phys.* 2008, **227**:5432.

43. Phillips CL, Anderson JA, Glotzer SC: **Dynamics and Dissipative Particle Dynamics simulations on GPU devices**. *J. Comp. Phys.* 2011, **230**(19):7191–7201.

44. Harvey MJ, Giupponi G, De Fabritiis G: **ACEMD: Accelerating biomolecular dynamics in the microsecond timescale**. *J. Chem. Theory Comput.* 2009, **5**:1632–1639.

45. Friedrichs MS, Eastman P, Vaidyanathan V, Houston M, Grand SL, Beberg AL, Ensign DL, Bruns CM, Pande VS: **Accelerating Molecular Dynamic Simulation on Graphics Processing Units.** *J. Comp. Chem.* 2009, **30**(6):864–872.

46. Plimpton S: **Fast Parallel Algorithms for Short-Range Molecular dynamics**. *J. Comp. Phys.* 1995, **117**:1–19.

47. Brown WM, Wang P, Plimpton SJ, Tharrington AN: **Implementing Molecular Dynamics on Hybrid High Performance Computers - Short Range Forces**. *Comp. Phys. Commun.* 2011, **182**:989–911.

48. Trott CR, Winterfield L, Crozier PS: **General-purpose molecular dynamics simulations on GPU based clusters**. *arXiv eprint* 2011, **1009**.

49. Kartsaklis C: **DL_POLY 3: Hybrid CUDA/OpenMP porting of the non-bonded force-field for two-body systems**. In *Proceedings of the 240th American Chemical Society National Meeting, Boston*, American Chemical Society 2010.

50. Phillips JC, Stone JE, Schulten K: **Adapting a message-driven parallel application to GPU-accelerated clusters**. *J. Parallel Comp.* 2009, **35**:164–177.

51. Harvey MJ, De Fabritiis G: **An Implementation of the smooth particle mesh Ewald summation on GPUs**. *J. Chem. Theroy Comput.* 2009, **5**(9):2371–2377.

52. Goetz AW, Williamson MJ, Xu D, Poole D, Grand SL, Walker RC: **Routine microsecond molecular dynamics simulations with AMBER - Part II: Particle Mesh Ewald"**. *in preparation.*

53. Buch I, Harvey MJ, Giorgino T, Anderson DP, De Fabritiis G: **High-throughput all-atom molecular dynamics simulations using distributed computing**. *J. Chem. Inf. Model.* 2010, **50**(3):397–403.

54. Buch I, Giorgino T, De Fabritiis G: **Complete reconstruction of an enzyme-inhibitor binding process by molecular dynamics simulations**. *Proc. Nat. Acad. Sci.* 2011, **108**(25):10184–10189.

55. Baker JA, Hirst JD: **Molecular Dynamics Simulations Using Graphics Processing Units**. *J. Mol. Graph.* 2011, **30**(6-7).

56. Xu D, Williamson MJ, Walker RC: **Advancements in Molecular Dynamics Simulations of Biomolecules on Graphical Processing Units**. In *Annual Reports in Computational Chemistry*, *Volume 6*, Elsevier 2010.

57. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA: **The Landscape of Parallel Computing Research: A View from Berkeley**. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley 2006, [http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html].

58. Harvey MJ, De Fabritiis G: **SWAN: A tool for porting CUDA programs to OpenCL**. *Comp. Phys. Commun* 2010, **182**:1093.