JCTC Journal of Chemical Theory and Computation

An Implementation of the Smooth Particle Mesh Ewald Method on GPU Hardware

M. J. Harvey*,† and G. De Fabritiis*,‡

High Performance Computing Service, Information and Communications Technologies, Imperial College London, South Kensington, London, SW7 2AZ, United Kingdom and Computational Biochemistry and Biophysics Lab (GRIB-IMIM), Universitat Pompeu Fabra, Barcelona Biomedical Research Park (PRBB), C/ Doctor Aiguader 88, 08003 Barcelona, Spain

Received May 28, 2009

Abstract: The smooth particle mesh Ewald summation method is widely used to efficiently compute long-range electrostatic force terms in molecular dynamics simulations, and there has been considerable work in developing optimized implementations for a variety of parallel computer architectures. We describe an implementation for Nvidia graphical processing units (GPUs) which are general purpose computing devices with a high degree of intrinsic parallelism and arithmetic performance. We find that, for typical biomolecular simulations (e.g., DHFR, 26K atoms), a single GPU equipped workstation is able to provide sufficient performance to permit simulation rates of ≈50 ns/day when used in conjunction with the ACEMD molecular dynamics package[1] and exhibits an accuracy comparable to that of a reference double-precision CPU implementation.

I. Introduction

Atomistic molecular mechanics simulations are widely used in the study of a range of biomolecular and inorganic molecular systems. However, the $O(N^2)$ scaling of the electrostatic interactions, combined with the explicit time step scheme, make it challenging to access the microsecond time scale. To reduce the computational cost, the contribution of longer-range interactions is typically approximated by using a scheme with more favorable scaling properties, such a derivative of the Ewald summation method.[2] The smoothed particle mesh Ewald summation (SPME),[3] which scales with $O(N\log N)$ is perhaps the most widely used of these methods in popular MD packages.

Production molecular dynamics (MD) simulations are typically performed on highly parallel multiprocessor or cluster supercomputers. Parallelization of the directly calculated nonbonded electrostatic interactions may be efficiently achieved by using spatial decomposition methods, such as the neutral territory schemes of Bowers et al.[4] as only local neighbor communication is required. Parallelization of the PME method is more challenging because of the long-range communications pattern required by the 3D Fourier transform step, and consequently, the development of efficient implementations remains an active topic of research.[5,6]

Due to the computing resources required to perform MD simulations, there has been considerable research into accelerating the computation by using, for example dedicated, specialized hardware[7] or commodity, high-performance accelerator processors.[8] In recent years, commodity graphics processing units (GPUs) have acquired nongraphical, general purpose programmability and have undergone a doubling of computational power every 12 months. Of the devices currently available on the market, those produced by Nvidia offer the most mature programming environment, the so-called compute unified device architecture (CUDA),[9] and have been the focus of the majority of investigation in the computational science field.

Several groups have lately shown results for MD codes which are accelerated by use of CUDA capable GPUs, for example,[10−14] and it has been recognized that high-performance, low-cost GPU accelerated molecular dynamics simulations could have a significant practical impact in the

* Corresponding authors. E-mail: m.j.harvey@imperial.ac.uk and gianni.defabritiis@upf.edu.

† Imperial College London.

‡ Universitat Pompeu Fabra.

10.1021/ct900275y CCC: $40.75
Published on Web 08/14/2009

field of in silico drug discovery.[15] In recent work, we presented ACEMD, a MD package designed to run on high-performance Nvidia GPUs.[1,16] ACEMD exhibits performance comparable to clusters of ≈100−200 CPUs and enables microsecond scale simulations on a single GPU equipped workstation, representing a significant improvement in the accessibility and cost-efficiency of MD simulation. A principal feature of ACEMD is its SPME treatment of long-range electrostatic terms as this is generally considered a requirement for production quality biomolecular simulations.

In this paper, we focus on the GPU implementation of the SPME scheme used in ACEMD and discuss the resulting performance.[1] The SPME calculation is performed on a single GPU but utilizes the device's massive intrinsic parallelism, necessitating significant modification of the implementation in comparison to a serial or parallel block decomposed version for conventional CPUs.

II. Algorithm and Implementation

A. The SPME Method. The SPME method of Essman and co-workers[17] is an efficient scheme for computing long-range electrostatic forces which exhibits improved $O(N\log N)$ scaling in comparison to the $O(N^2)$ scaling of standard Ewald summation, differing from the latter in the use of interpolation and the 3D FFT in computing the reciprocal space summation.

The electrostatic interaction energy of an N particle system is given by the sum of the pairwise Coulombic interactions:

$$E = \sum_{i=1}^{N-1}\sum_{j>i}^{N}\frac{q_i q_j}{r_{ij}} \tag{1}$$

where q_i is the charge of the ith particle and r_{ij} the distance between the ith and jth particles. The traditional Ewald summation method re-expresses E as the sum of three potentials $E = E_{\mathrm{dir}} + E_{\mathrm{rec}} + E_{\mathrm{self}}$, where E_{dir} is summed over all pairs in real space, E_{rec} is summed in reciprocal space, and E_{self} is a corrective term representing the self-energy of the system. These three potentials are defined as

$$E_{\mathrm{dir}} = \sum_{i=1}^{N-1}\sum_{j>i}^{N}\frac{q_i q_j \mathrm{erfc}(\beta r_{ij})}{r_{ij}} \tag{2}$$

$$E_{\mathrm{rec}} = \frac{1}{2\pi V}\sum_{\vec{m}\neq 0}\frac{\exp(-\pi^2\vec{m}^2/\beta^2)}{\vec{m}^2}S(\vec{m})S(-\vec{m}) \tag{3}$$

$$E_{\mathrm{self}} = -\frac{\beta}{\sqrt{\pi}}\sum_{i}^{N} q_i^2 \tag{4}$$

where $\mathrm{erfc}(x)$ is the complementary error function $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$, β is the Ewald parameter, $\vec{m} = (m_1, m_2, m_3)$ are the reciprocal space lattice vectors, and V is volume of the unit cell in reciprocal space. $S(\vec{m})$ is the lattice structure factor, given by

$$S(\vec{m}) = \sum_{j} q_j \exp(\vec{m}\vec{r}_j) \tag{5}$$

The SPME method approximates this with

$$S(\vec{m}) \approx \tilde{S}(\vec{m}) = b_1(m_1)b_2(m_2)b_3(m_3)F(Q)(m_1, m_2, m_3) \tag{6}$$

where $b_i(m_i)$ are Euler exponential splines given by

$$b_\alpha(m_\alpha) = \exp(2\pi i(n-1)m_\alpha/K_\alpha)\times \left[\sum_{k=0}^{n-2} M_n(m+1)\exp(2\pi i m_\alpha k/K_\alpha)\right]^{-1} \tag{7}$$

for $\alpha = 1, 2, 3$. $\mathrm{F}(Q)$ is the discrete Fourier transform of the 3D matrix Q of dimension $K_1 \times K_2 \times K_3$ which is given by

$$\begin{aligned} Q(k_1,k_2,k_3) &= \sum_{i=0}^{N}\sum_{p_\alpha=0}^{K_\alpha-1} q_i M_n(u_i^1 - k_1 - p_1K_1) \\ &\times M_n(u_i^2 - k_2 - p_2K_2) \\ &\times M_n(u_i^3 - k_3 - p_3K_3) \end{aligned} \tag{8}$$

where $\vec{u}_i$ is the scaled fractional coordinate of particle i within the bounds $0 \le u_i^\alpha \le K_\alpha$. M_n is the cardinal B-spline of order n. The summations are over all integers p_α in the range $0 \le p_\alpha < K_\alpha$. Cardinal B-splines have the following recursive form:

$$M_n(u) = \frac{u}{n-1}M_{n-1}(u) + \frac{n-u}{n-1}M_{n-1}(u-1) \tag{9}$$

for $n \ge 2$. For $n = 2$, $M_2(u) = 1 - |u - 1|$ for $0 \le u \le 2$ else $M_2(u) = 0$. B-splines have compact support, and the effect of applying Equation 8 is to 'spread' each charge q_i out over a cubic volume of n^3 elements of Q.

The approximate reciprocal space energy sum $\tilde{E}_{\mathrm{rec}}$ can be re-expressed as the convolution:

$$E_{\mathrm{rec}} \approx \tilde{E}_{\mathrm{rec}} = \frac{1}{2}\sum_{m_\alpha=0}^{K_\alpha-1} Q(m_1,m_2,m_3)\times(\theta_{\mathrm{rec}}{*}Q)(m_1,m_2,m_3), \tag{10}$$

where

$$\theta_{\mathrm{rec}} = F(B\times C) \tag{11}$$

and B and C are the arrays given by

$$B(m_1,m_2,m_3) = \prod_{i=1}^{3}|b_i(m_i)|^2 \tag{12}$$

$$C(m_1,m_2,m_3) = \frac{1}{\pi V}\frac{\exp(-\pi^2\vec{m}^2/\beta^2)}{\vec{m}^2} \tag{13}$$

From which the per particle forces may be derived by differentiation with respect to $\vec{r}_i$. As Q is continuously differentiable $n-2$ times with respect to the particle positions, and θ_{rec} is independent of the particle positions, the atomic force terms are given by

$$\frac{\delta\tilde{E}_{\mathrm{rec}}}{\delta r_i^\alpha} = \sum_{m_\alpha=0}^{K_\alpha-1}\frac{\delta Q}{\delta r_i^\alpha}(m_1,m_2,m_3)\times(\theta_{\mathrm{rec}}{*}Q)(m_1,m_2,m_3) \tag{14}$$

B. Nvidia GPU Architecture. Contemporary GPUs produced by Nvidia Corp. are general purpose, single program

multiple data (SPMD) processing units with a high degree of intrinsic parallelism and arithmetic performance, for example, over 30K execution threads may run concurrently on the Tesla C1060 device, with a peak arithmetic rate of ≤933 single precision GFLOPS. These devices are composed of a set of independent *multiprocessors*, each of which contains eight scalar cores. A scalar program fragment, known as a *kernel* may be run concurrently in a *block* of *threads* in parallel *warps* of 32 threads. Each block is restricted to executing on a single multiprocessor. Multiple independent blocks may be executed concurrently across the device in a *grid*.

Threads within a common block may intercommunicate via a small 16 kB region of in-core shared memory. Each multiprocessor has a large register file of which each thread receives a private, static allocation. The total number of threads, which may execute on a multiprocessor, and thus, the degree of parallelism, is dependent on the register resources required by each individual thread and also on the shared memory required by the block. Current devices support up to 1 024 threads and up to 8 blocks per multiprocessor.

Multiprocessors are grouped together on a single package and share a high-bandwidth link to external DRAM, known as *global memory*. Access to this memory is uncached and so is particularly costly, although the interleaved execution of warps acts to mitigate the cost of this latency.

The current programming model for these GPUs is CUDA, a C-like language with type qualifier extensions for indicating data locality and a special function call syntax for specifying the parallelism of a kernel invocation. An associated API provides functions for managing GPU host memory allocation and transfer. CUDA code is compiled to an intermediate byte code which is interpreted at runtime, providing forward compatibility with future device architectures.

For further details on device capabilities and programming model, the reader is referred to the CUDA SDK documentation.[9]

C. SPME Implementation. The implementation of SPME has two distinct components: the real space evaluation of the pairwise E_{dir} term and the reciprocal space evaluation $\tilde{E}_{rec}$. The evaluation of the former is trivially implemented into an existing nonbonded electrostatic force computation program with a change of potential function,[1] and we do not discuss this further. $\tilde{E}_{rec}$, however, involves several different computational steps, and its calculation frequently becomes the limiting step in parallel molecular dynamics simulations.[6,18]

In this section, we describe in detail the implementation of the computation of $\tilde{E}_{rec}$ using CUDA. Neither part of the computation is performed by the host CPU nor is it necessary to transfer any state between the GPU and host memory. In this way the performance of the code is entirely dependent upon the specification of the GPU.

The calculation of E_{rec} may be divided into five steps, which we summarize below before describing their implementation:

(1) Charge spreading: Spreading of charges on to the array Q (eq 8). The scaled fractional coordinates of each particle are calculated, and the Q array is populated with n^3 terms computed from the B-spline coefficients $M_n(u_i^\alpha - j)$, where $i = 1, ..., N$, $\alpha = 1, 2, 3$ and $j = 0, ..., n$.

(2) 3D fast Fourier transform (FFT): Real-to-complex 3D FFT in place transformation of Q into reciprocal space.

(3) Energy computation: Application of eq 10 in reciprocal space to compute terms for $\tilde{E}_{rec}$. Q is replaced by the product of itself with array B and C, as defined in eqs 12 and 13.

(4) 3D FFT: Computation of the convolution θ_{rec} (eq 11) through a complex-to-real 3D FFT of the array resulting from step 3.

(5) Force computation: real space computation of per atom force terms $\delta\tilde{E}_{rec}/\delta r_i^\alpha$ by multiplication with $\delta Q/\delta r_i^\alpha$ (eq 14).

Charge Spreading. In this step, each charge q_i is mapped to a site on the real space PME grid at a location determined by particle scaled fractional coordinates $\vec{r}_i$. The charge is then distributed over points in a surrounding volume according to eq 8. The volume of the spreading region is dependent on the order n of the cardinal B-spline. For $n = 4$, typically used in production simulations, each charge is spread over $n^3 = 64$ grid points.

This spreading is straightforward to do in a serial implementation but poses a significant challenge when performed with fine-grained parallelism. A naïve implementation, which used one thread to map the spread of each individual charge onto the grid, would encounter synchronization problems when different threads attempt to accumulate charge on the same grid location, thus, necessitating the use of thread safe atomic memory operations.[23] However, as only integer atomic operations are supported on current hardware, either the floating point atomic operations must be emulated using an atomic compare-and-set loop construct or the charge spreading must be altered to use fixed precision arithmetic. Furthermore, in order to achieve acceptable performance from the uncached GPU memory subsystem, it is necessary for the threads within a block to perform memory accesses to contiguous address ranges. The naïve approach has an essentially unordered memory access pattern, leading to poor performance.

Rather than performing a charge spreading, a per grid point gather is used instead. This is conducted in three steps which we term *placement*, *accumulation*, and *overflow*. First, each particle is mapped to a grid location, and its charge and position are recorded in a three-dimensional array. At the PME grid sizes and system densities typical of biomolecular simulation, this array is 90% sparse. Nevertheless, because the placement is performed in parallel, one particle per thread, the setting of elements within this array must still be performed atomically using the global memory atomic operation primitives in order to prevent access conflicts. Each grid site is permitted to hold a single charge; additional charges are placed in a simple list referred to as the overflow list.

Second, a separate kernel is used to sum the charge at each grid point by finding the contributions from all charges within the surrounding n^3 points (Figure 1). This kernel is executed in a block of K_1 threads which operates on a single full grid width stencil along the m_x vector. Each thread i accumulates the charge for the ith grid point in the row. Each

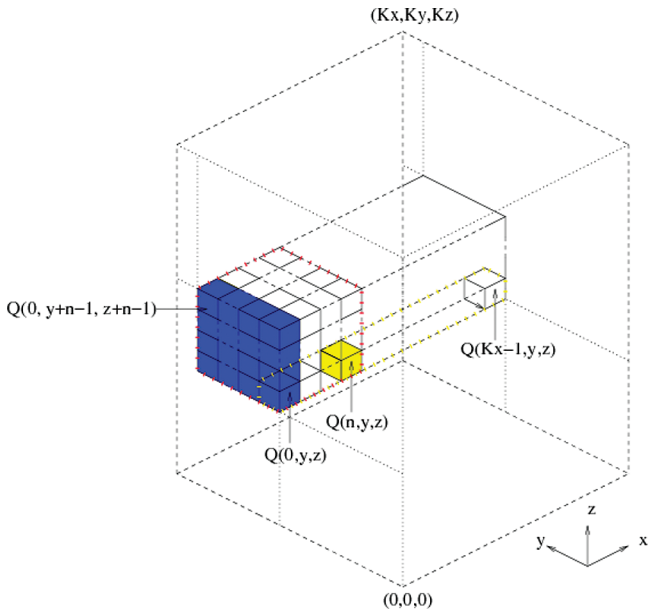

Figure 1. Charge-spreading phase in the real space charge grid. For efficient implementation, the charges are gathered to each grid point (shown as cells for clarity) for a spline order of n, each grid point receives charge terms from n^3 neighboring points (red dashed region indicates the cubic charge import region for the yellow (light-gray) point $Q(n, y, z)$. The charge gathering computational kernel operates in (K_y, K_z) blocks of K_x threads, each block calculating charge terms for a whole x row of the grid. Each thread loops over the n^2 neighbor cells in the y, z plane (blue/dark-gray region), calculating in turn the $3n$ B-spline terms M_n, corresponding to any charge located in the cell and accumulating the charge contribution for its cell. The spline terms are held in shared memory, allowing each thread to reuse the values computed by its neighbors for computing charge contributions from the adjacent $(n - 1)$ y, z planes.

thread loads in turn the n^2 surrounding points in the y, z plane, and, if containing a particle, the associated $3n$ spline terms $M_n(u_i^\alpha - j)$ (for $\alpha = 1, 2, 3$, $j = 0, ..., n$) are computed and placed into shared memory where they may be accessed by other threads.

This use of shared memory reduces the number of global memory accesses by a factor of n and has a memory access pattern favorable for the hardware. Although this method requires each set of spline terms to be computed n^2, the arithmetic cost of the calculation is sufficiently low to make their recomputation preferable to loading them from a precomputed table in global memory.

Lastly, the charges in the overflow list are spread onto the grid using both the naïve method described initially and the floating point atomic memory operations synthesized with an atomic integer compare-and-set loop construct. Although this processing is slow, it is much faster than a second iteration of the gathering kernel.

One consequence of the use of the per-grid-point gather is that the scaling characteristic of the charge griding phase changes from $O(N)$ to $O(K^3)$.

3D FFT. 3D FFTs are performed using the CUDA cuFFT library.[19] Although this library provides 3D FFT routines, the current version (version 2.2) shows poor performance.

Table 1. Performance of the PME Code in the Parallel MD Program ACEMD (Running on Three GTX280 GPUS) for a Range of Model Sizes Using Production Parameters of B-spline Order $n = 4$ and Timestep = 4 fs[a]

model	atoms	PME grid size	10^6 timesteps/day	ns/day
DHFR	26K	65	12	49
ApoA1	92K	108	2.7	11
STMV	1M	220	0.17	0.69

[a] STMV performance estimated from timings given in Figure 3.

Consequently, we implement the multidimensional FFT explicitly using three sets of orthogonal 1D transforms. The transforms are interleaved with in-place transpositions of the charge array Q, which is necessary to satisfy the contiguous memory layout of the input data required by the FFT library.

Energy Computation. The $b(m_i)$ and exponential terms for the Euler exponential splines (eq 7) are precomputed once and stored in the GPU's *constant* memory. This is a small (64 kB), read-only region of global memory that has cached read access for reduced access latency. Calculation of the product of the transformed Q with $B \times C$ (eq 11) is then performed in place using a kernel that computes one column of values along m_z per thread. When required for diagnostic output, energy terms can be accumulated in a separate buffer, and the sum transferred back to the host.

Force Computation. The computation of $\delta\tilde{E}_{rec}/\delta r_i^\alpha$ is performed using a kernel which operates on a single particle per thread. The $3n$ spline terms M_n are again computed, along with the first derivatives M'_n, once the per particle and n^3 force terms are computed as per eq 14. Because the particle distribution is essentially unordered, memory accesses are uncoalesced, but the kernel is sufficiently simple to permit a large number of threads per block, effectively hiding the memory access latency.

III. Discussion

A. Performance. The performance was also measured on production simulations of models of dihydrofolate reductase (DHFR) ($62 \times 62 \times 62$ Å^3, 23 558 atoms) and apoA1 ($108 \times 108 \times 78$ Å^3, 92 224 atoms), two protein systems commonly used for benchmarking exercises.[1] The DHFR and apoA1 models are typical of the scale of system routinely simulated using all-atom biomolecular mechanics with 1 million atom simulations of complete virions (e.g., the satellite tobacco mosaic virus, STMV)[20] at the far extreme, representing a range of simulation cell sizes from $\approx 60^3$−220^3 Å^3. We determine the practical performance in timesteps/day achievable with production simulation parameters and ACEMD, as shown in Table 1.

ACEMD is a parallel code able to distribute all other aspects of an MD simulation (bonded, nonbonded force terms, etc.) over multiple GPUs and, in practice, the PME computation represents the performance-limiting critical path in simulations over this range of sizes, making the optimization of the PME implementation essential for best performance.

To more closely assess the performance characteristics of the code, it was tested over a range of synthetic cubic input systems constructed with linear dimension in the range of

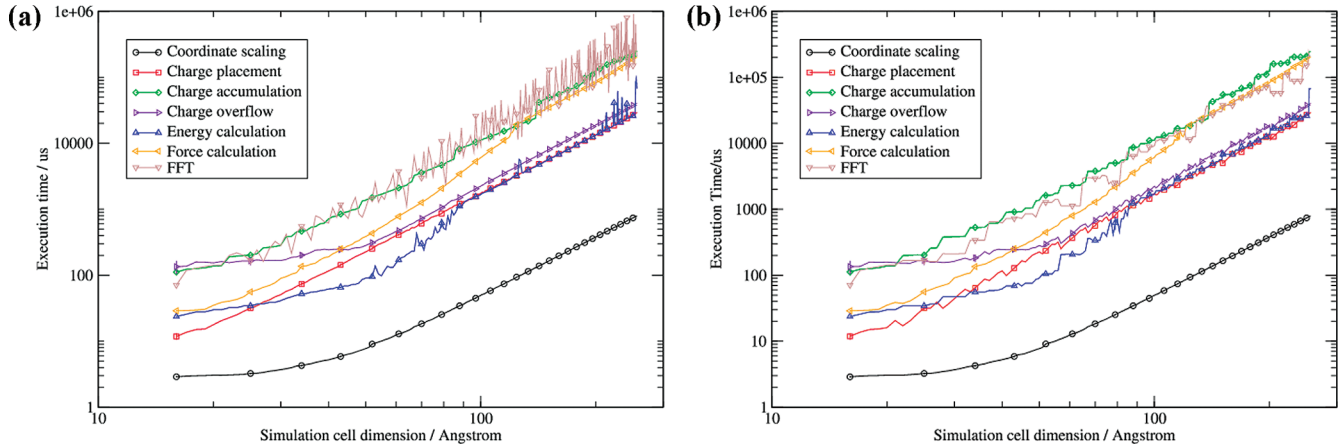

Figure 2. Execution times for the individual GPU kernels given as a function of the linear dimension of a cubic simulation cell for a constant system number density of 0.1Å^{-3}. FFT timing includes both forward and reverse 3D FFT steps. (a) Shows timings for the case of the PME grid having linear dimension $K = \text{int}(l_{\text{sim}})$. (b) Timings are given for optimized K, where $K \geq \text{int}(l_{\text{sim}})$, as determined by the method given in the main text. Error bars indicate σ^2.

$16 \leq l_{\text{sim}} \leq 256$ Å and with a random uniform distribution of particles with number density ρ_N of 0.1 atoms Å^{-3}, representative of the sizes and density of biomolecular systems of interest.

Execution times, T_{kernels}, for each individual kernel were recorded using the standard CUDA runtime profiling tool. The total execution wall time, T_{PME}, was measured from start to finish and so includes all additional host(CPU)-side overhead.

K_α were initially chosen to be equal to $\text{int}(l_{\text{sim}})$, as is commonly used in production MD simulations. For each synthetic system size, 100 iterations were performed, each with a different random particle distribution in order to characterize the sensitivity of the implementation to fluctuations in the distribution (Figure 2a).[24]

For large K, T_{PME} exhibits cubic scaling with respect to the PME grid dimension (dashed lines in Figure 3), corresponding to a broadly linear scaling with respect to particle number. However, for small grid sizes and particle counts, $K < 50$, there is insufficient available parallelism to fully occupy the GPU resources. This can be seen more clearly in the component kernel timings in Figure 2 (a and b), most prominently in the coordinate scaling, and in the energy calculation and overflow list kernels. In the case of the first two, the kernels are computationally simple, and a high thread occupancy may be achieved, e.g., the coordinate scaling kernel may run 15 360 threads (512 threads/multiprocessor) simultaneously, corresponding to a test system size of $K = 53$. For smaller systems, there is insufficient parallelism to fully occupy the GPU, and the execution time of the kernel is weakly dependent on system size; all blocks may be spread out across multiprocessors and executed completely in parallel. At $K = 53$, the GPU is fully subscribed, and a step in the execution time can be seen, indicating that some multiprocessors have now to process a second block. After this point, the execution time scales approximately linearly with particle count as subsequent step changes are much less significant owing to divergence between the work executing on the separate multiprocessors.

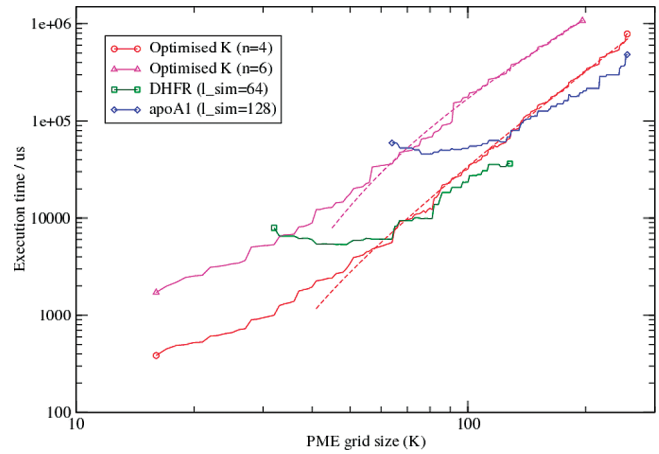

Figure 3. T_{PME} timing showing the effect of varying the PME grid size for a fixed system size K and the B-spline order n. Magenta (circles) and red (triangles) lines show timings for optimized PME grid dimensions with a spline order of 4 (red) and 6 (magenta) for the DHFR model. Dashed lines show cubic fits for $l_{\text{sim}} > 40$. For the green (squares) and blue (diamonds) lines, the simulated systems are of constant size (26K atoms DHFR and 92K apoA1 benchmarks, respectively), and the ordinal indicates the PME grid dimension K.

In the case of the overflow and charge placement kernels, a significant variation in runtime is observable; this arises from collisions between atomic operations of different threads. The synthesized floating point atomic operations used in the overflow kernel are much more sensitive to collisions as they use a looping construction that may make several accesses to global memory. Because a warp of threads is executed synchronously, it requires only one thread to suffer an unfavorable memory access for the execution time of the entire block to increase.

For the charge accumulation and energy calculation kernels, the number of threads per block is directly related to K_1. Initially, several blocks may run concurrently per multiprocessor, but as K_1 increases, the occupancy falls.

We use the Nvidia-supplied CUDA FFT library. As is characteristic of the FFT algorithm, the computational cost is highly sensitive to the prime factorization of the input

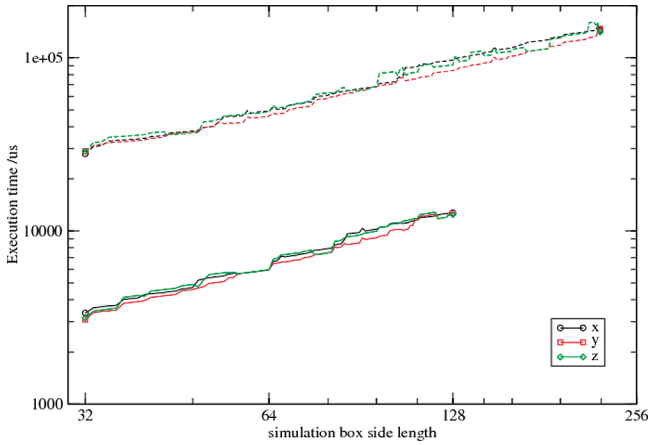

Figure 4. T_{PME} timing showing the minor sensitivity of the implementation to the orientation of the input for cuboid systems. The length of each simulation box dimension l_α is varied in turn within the range $l_{sim}/2 \leq l_\alpha \leq 2\ l_{sim}$ with the remaining sides held at a length of l_{sim}, where $l_{sim} = 64$ (solid lines) and 128 (dashed lines). A constant system density of $\rho_N = 0.1$ Å^3 is maintained.

dimension (Figure 2a). For this reason, some PME implementations, such as that of Desmond[21] are designed to use only specific grid sizes (2^n in that case). However, for the relatively small systems (O(100K) atoms) that ACEMD is intended to simulate, this would be quite restrictive. Therefore, for each l_{sim}, we select a modified K in the range $l_{sim} \leq K \leq 1.5\ l_{sim}$ such that T_{PME} is minimized. By selecting for the minimum T_{PME}, rather than just minimum FFT cost, we compensate for the increased runtime of components of the implementation which are sensitive to the ρ_N and K of the input. $T_{kernels}$ for these optimized dimensions are shown in Figure 2b, and T_{PME} is shown in Figure 3. The sawtooth pattern of the charge placement and overflow kernels are indicative of atomic operation collisions, the probability of them occurring dropping as K becomes larger than int(l_{sim}).

Because the accumulation and energy kernels operate preferentially along one dimension of the PME grid, we test the sensitivity of the implementation to the orientation of the input for cuboid systems. Figure 4 shows the execution time for synthetic input systems of constant ρ_N, varying each dimension l_α (for $\alpha = 1, 2, 3$) in turn in the range $0.5\ l_{sim} \leq l_\alpha \leq 2\ l_{sim}$ for $l_{sim} = 64\ 128$ and using the predetermined 'optimized' grid dimensions. While there are differences between the three variations, reflecting differing grid sizes and multiprocessor occupancies, we conclude that these are insufficiently large to require care over the orientation of the input system.

Finally, we consider the effect of the spline interpolation order on performance. When optimizing the SPME parameters for production MD simulations, it is not unusual to reduce grid size and increase the spline interpolation order to optimize performance or control accuracy. We tested the performance impact of these changes using two commonly used biomolecular benchmarks − DHFR (26K atoms) and apoA1 (92K atoms). In each case, the PME grid size was varied between $0.5\ l_{sim} < K_\alpha < 2\ l_{sim}$, and the timings were taken (Figure 3, green (squares) and blue (diamonds) lines). It can be seen that T_{PME} is minimal at approximately $K_\alpha = 0.6 l_{sim}$, increasing for smaller grids as the overflow list processing becomes significant. However, increasing the interpolation order to six significantly increases the execution time of the charge accumulation (Figure 3, magenta line) and no performance increase is yielded.

B. Accuracy. The GPU code exclusively uses single precision floating point arithmetic and, thus, operates at a reduced precision in comparison to a typical double precision CPU code. To quantify any reduction in accuracy, we compared the force terms produced by the GPU code (using ACEMD)[1] with a reference double precision CPU code (NAMD)[22] when performing the joint Amber−Charmm DHFR benchmark. The relative error of the reciprocal force terms, $|F_{NAMD}| - |F_{ACEMD}|/|F_{NAMD}|$, was found to be 10^{-5}, significantly below the 10^{-3} generally considered the acceptable maximum for relative error in the force terms for biomolecular simulation.[21] As such, we consider single precision arithmetic to be adequate for production simulation.

For long simulations in the NVE ensemble in which energy conservation is important, the code could be converted to use the double precision arithmetic capabilities of the latest GPUs. As this would incur a significant performance penalty (up to ≈ 8 times slower) and is not presently necessary for our work. This remains an area for future investigation. Furthermore, a double precision version of the cuFFT library is not yet available (as of CUDA 2.2).

Further data on the accuracy of benchmark MD simulations performed with PME in conjunction with ACEMD are given in Section IV of ref 1.

IV. Conclusions

We have implemented the smooth particle mesh Ewald method on Nvidia GPU hardware using the CUDA programming language. The implementation has been integrated into ACEMD,[1] permitting all-atom biomolecular simulations with long-range electrostatics to be fully accelerated on GPU devices with an accuracy comparable to a reference double precision CPU implementation. On contemporary hardware, a high level of performance is observed, commensurate with molecular dynamics simulation rates between 100 ns/day for systems of a few thousand atoms and 1 ns/day for one million atom systems, when used within ACEMD.

Acknowledgment. This work was partially funded by the HPC-EUROPA project (R113-CT-2003-506079). G.D.F. acknowledges support from the Ramon y Cajal scheme and the EU Virtual Physiological Human Network of Excellence. We gratefully acknowledge Nvidia Corporation (http://www.nvidia.com) for their hardware donations.

References

(1) Harvey, M. J.; De Fabritiis, G.; Giupponi, G. *J. Chem. Theor. Comp.* **2009**, *5* (*6*), pp 1632−1639.

(2) Ewald, P. *Ann. Phys.* **1921**, *369*, 253–287.

(3) Essman, U.; Perera, L.; Berkowitz, M. L.; Darden, T.; Lee, H.; Pedersen, L. G. *J. Chem. Phys.* **1995**, *19*, 8577–8593.

(4) Bowers, K. J.; Dror, R. O.; Shaw, D. E. *J. Phys.: Conf. Series* **2005**, *16*, 300–304.

(5) Oh, K. J.; Deng, Y. *Comp. Phys. Commun.* **2007**, *177*, 426–431.

(6) Fang, B.; Martyna, G.; Deng, Y. *Comp. Phys. Commun.* **2007**, *177*, 362–377.

(7) Shaw, D. E. Anton, a Special-Purpose Machine for Molecular Dynamics Simulation. *Proc. 34th Int. Symp. Comput. Architecture* **2007**, 1−12.

(8) De Fabritiis, G. *Comput. Phys. Commun.* **2007**, *176*, 600.

(9) Technical Report, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide 2.0*; Nvidia Corp **2008**, http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf; (Accessed August 4, 2008).

(10) Anderson, J. A.; Lorenz, C. D.; Travesset, A. *J. Comp. Phys.* **2008**, *22*, 5342–5359.

(11) Meel, J. A. V.; Arnold, A.; Frenkel, D.; Portegies, S. F.; Belleman, R. G. *Mol. Simul.* **2008**, *34*, 259–266.

(12) Phillips, J. C.; Stone, J. E.; Schulten, K. Adapting a message-driven parallel application to GPU-accelerated clusters *Proc. 2008 ACM/IEEE Conf. Supercomputing* **2008**; Article no. 8.

(13) Friedrichs, M. S.; Eastman, P.; Vaidyanathan, V.; Houston, M.; LeGrand, S.; Beberg, A. L.; Ensign, D. L.; Bruns, C. M.; Pande, V. S. *J. Comput. Chem.* **2009**, *30*, 864–872.

(14) Liu, W.; Schmidt, B.; Voss, G.; Müller-Wittig, W. *Comput. Phys. Commun.* **2008**, *179*, 634–641.

(15) Giupponi, G.; Harvey, M. J.; de Fabritiis, G. *Drug Discov. Today* **2008**, *13*, 1052.

(16) ACEMD home page: http://multiscalelab.org/acemd. Accessed August 4, 2008.

(17) Essman, U.; Perera, L.; Berkowitz, M. L.; Darden, T.; Lee, H.; Pedersen, L. G. *J. Chem. Phys.* **1995**, *103*, 8577.

(18) Toukmaji, A.; Paul, D.; Board, J., Jr. Technical Report: Distributed Particle Mesh Ewald: A Parallel Ewald Summation Method. *Proc. Int. Conf. Parallel Distributed Processing Techniques Applications*; Arabnia, H. R., Ed.; CSREA Press, **1996**; pp 33−43.

(19) Technical Report, *CUDA CUFFT Library, Document PG-00000−003 V2.0*; Nvidia Corp., **2008**, http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUFFT_Library_2.0.pdf; (Accessed August 4, 2008).

(20) Freddolino, P. L.; Arkhipov, A. S.; McPherson, S. B.; Schulten, K. *Structure* **2006**, *14*, 437–449.

(21) Bowers, K. J.; Chow, E.; Xu, H.; Dror, R. O.; Eastwood, M. P.; Gregersen, B. A.; Klepeis, J. L.; Kolossvary, I.; Moraes, M. A.; Sacerdoti, F. D.; Salmon, J. K.; Shan, Y.; Shaw, D. E. Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters; *Proc. Supercomputing 2006*; Tampa, FL, November 11−17, 2006.

(22) Phillips, J. C.; Braun, R.; Wang, W.; Gumbart, J.; Tajkhorshid, E.; Villa, E.; Chipot, C.; Skeel, R. D.; Kale, L.; Schulten, K. *J. Comput. Chem.* **2005**, *26*, 1781–1802.

(23) Atomic functions are memory operations that are designed to ensure that concurrent accesses to the same memory location by competing threads always leave the contents of that location in a consistent state by preventing race conditions. These functions require dedicated hardware support and have limited set of operations, as detailed in Appendix C of ref 9.

(24) The test hardware was a single Nvidia Tesla C1060 an HP xw660 workstation (dual Xeon 5430, 4GB RAM, Red Hat Enterprise Linux 5.3, CUDA 2.1, Nvidia driver 180.22).

CT900275Y