JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 8, July 2011

Graphics Processing Unit-Accelerated Boundary Element Method and Vortex Particle Method

Mark J. Stock* and Adrin Gharakhani†
Applied Scientific Research, Santa Ana, CA 92705-5803

DOI: 10.2514/1.52938

Vortex particle methods, when combined with boundary element methods, become a powerful tool for the simulation of internal or external vortex-dominated flows. To speed up the simulations substantially, algorithms are presented herein for the implementation of a parallel adaptive multipole-accelerated treecode for evaluating the velocity field induced by vortex particles on a cluster of graphics processing units, as well as the implementation of a graphics processing unit-accelerated treecode for evaluating the velocity field induced by vortex and source boundary panels. The graphics processing unit-accelerated treecode for vortex particles outperforms its eight-core central processing unit counterpart by a factor of 43. Furthermore, the parallel efficiency of the treecode on a 32-node cluster of dual-graphics processing units is 80% for 100 million particles. The boundary element method treecode can solve for unknown source, vortex, or combined strengths over triangular surface meshes using all available central processing unit cores and graphics processing units. Problems with up to 1.4 million unknowns can now be solved on a single commodity desktop computer in 1 min, and at that size the hybrid central processing unit/graphics processing unit outperforms a quad-core central processing unit alone by 22.5 times. The method is applied to simulations of the early stages of impulsively started flow over a sphere at Reynolds numbers 500, 1000, 2000, and 4000 based on the diameter and freestream velocity.

Nomenclature

G	Green's function
g	vortex particle smoothing/core function
$i,\ j$	particle number indices
K	velocity kernel
N	number of computational elements
N_p	number of triangular panels
N_v	number of vortex particles
$\vec{n}$	unit surface normal vector into the fluid domain
Re	Reynolds number
S	boundary surface
t	time
$\vec{U}_\infty$	freestream velocity
$\vec{u}$	fluid velocity

Received 25 October 2010; accepted for publication 22 February 2011. Copyright © 2011 by Applied Scientific Research. Published by the American Institute of Aeronautics and Astronautics, Inc., with permission. Copies of this paper may be made for personal or internal use, on condition that the copier pay the $10.00 per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923; include the code 1542-9423/11 $10.00 in correspondence with the CCC.

* Research Scientist, mstock@Applied-Scientific.com, AIAA Senior Member.
† President, Corresponding Author, adrin@Applied-Scientific.com, AIAA Senior Member.

V	fluid volume
$\vec{x},\ \vec{x}'$	position vector
$\vec{\Gamma}$	vectorial circulation
$\vec{\gamma}$	wall-generated vortex sheet strength
$\vec{\nabla}$	the gradient operator
Δt	time step
D/Dt	material derivative, $(\partial/\partial t) + \vec{u} \cdot \vec{\nabla}$
δ_v	nominal interparticle spacing
$\partial/\partial t$	time derivative
θ	dilatation
σ	smoothing/core radius
$\vec{\omega}$	fluid vorticity
$\cdot$	dot product
$\times$	cross product

I. Introduction

LAGRANGIAN vortex methods are a novel alternative to the more common Eulerian methods for the simulation of fluid flows for several reasons: absence of numerical diffusion, no firm Courant–Friedrichs–Lewy advection limit, and substantially simpler mesh preprocessing. Their disadvantages include more narrow applicability and greater calculation cost per computational element. To this end, there is ongoing research to expand the applicability of vortex methods, whereas the aim of the present work is to improve its performance.

A vortex particle method solves the incompressible vorticity transport equations in Lagrangian form by discretizing the vorticity field onto a set of particles and keeping track of vorticity transport along the trajectory of the particles. The vorticity-induced velocity field is obtained via the solution of the Biot–Savart integral equation. The wall velocity boundary condition is imposed via the solution of a Fredholm equation of the second kind using boundary element method (BEM). Fast algorithms for evaluating both the Biot–Savart and BEM integrals based on hierarchical subdivision of the problem space are available; the two most common being the multipole treecode, which is $\mathcal{O}(N \log N)$ [1], and the fast multipole method (FMM), which is $\mathcal{O}(N)$ [2]. The present method uses an improved treecode developed previously [3], with computational complexity $\mathcal{O}(N^{1.1})$ or better.

With the current proliferation of hardware and software for data-parallel graphics processing unit (GPU) computing comes the opportunity to drastically reduce the cost of equivalent computing resources. For example, as of mid-2010, the best price per theoretical billion double-precision floating operations per second (GFLOP/s) is 2 USD for central processing units (CPUs) and 0.10–0.20 USD for GPUs. The disparity is even higher for single-precision calculations. This reduction in cost is greatest in cases with algorithms that exhibit high arithmetic intensity or those that have a very large ratio of instruction count to required memory access. Vortex (and other N-body) methods are well suited to this new programing paradigm and have demonstrated substantial performance improvements over more traditional multicore CPUs [3–6].

Previous fast N-body GPU work includes both treecode [3] and FMM [5,6] implementations. Treecode- and FMM-accelerated BEMs are common in the literature, but only for CPU implementations. Buatois et al. [7] implemented a GPU sparse linear solver, which is one component of a dense BEM. More recently, Takahashi and Hamada [8] presented a GPU-accelerated BEM for Helmholtz equations, although they used a direct $\mathcal{O}(N^2)$ method and did not store the coefficients in the influence matrix. No GPU implementations of a treecode or FMM-accelerated BEM for flow simulation were found in the literature at the time of writing this paper.

In the present work, a GPU-accelerated fast adaptive treecode for BEM and a parallel fast adaptive treecode for vortex particles on a cluster of GPUs are implemented in Ω-*Flow*, a grid-free incompressible flow simulation technology developed at Applied Scientific Research [3,9]. To this end, in what follows, a brief description of the underlying equations, the details of the algorithms and implementation, as well as partial results from simulations using the GPU-accelerated Ω-*Flow* will be presented. The cases presented are 1) the velocity and velocity gradients induced by randomly distributed particles in a unit cube, 2) the BEM component of the simulation of flow about a simplified model of a mechanical heart valve in the aortic root, and 3) the simulation of the early stages of impulsively started unsteady flow over a sphere at various Reynolds numbers.

II. Method

Ω-*Flow* is a Lagrangian vortex particle method that uses an adaptive hierarchical spatial decomposition scheme and high-order multipole treecode solver [1,3,9] to calculate the velocities and velocity gradients at any point in space. The software distributes much of the computational load to all local CPUs and compute-capable GPUs, and among distributed-memory computers in a cluster. The differences between previous works [3,9] and the present method are sufficient that the important parts of the formulation, algorithms, and implementation will be described in subsequent sections.

A. Vortex Particle Method

The governing equations of incompressible fluid flow in terms of the transport of vorticity are

$$\frac{\partial \vec{\omega}}{\partial t} + \vec{u} \cdot \vec{\nabla}\vec{\omega} = \vec{\omega} \cdot \vec{\nabla}\vec{u} + \frac{1}{\mathrm{Re}}\nabla^2\vec{\omega} \tag{1a}$$

$$\vec{\nabla} \cdot \vec{u} = 0 \tag{1b}$$

supplemented with the appropriate velocity boundary conditions.

The fluid velocity induced by the vorticity field and wall effects is prescribed by the following generalized Helmholtz integral formula:

$$\begin{aligned}\vec{u}(\vec{x}) = \vec{U}_\infty &+ \vec{\nabla} \times \int_V \vec{\omega}(\vec{x}')G(\vec{x},\vec{x}')dV(\vec{x}') \\ &- \vec{\nabla} \int_V \theta(\vec{x}')G(\vec{x},\vec{x}')dV(\vec{x}') \\ &+ \vec{\nabla} \times \int_S \big(\vec{\gamma}(\vec{x}') + \vec{n}(\vec{x}') \times \vec{u}(\vec{x}')\big)G(\vec{x},\vec{x}')dS(\vec{x}') \\ &- \vec{\nabla} \int_S \big(\vec{n}(\vec{x}') \cdot \vec{u}(\vec{x}')\big)G(\vec{x},\vec{x}')dS(\vec{x}')\end{aligned} \tag{2}$$

where $G(\vec{x},\vec{x}') = 1/(4\pi|\vec{x}-\vec{x}'|)$, and $\theta = \vec{\nabla}\cdot\vec{u} = 0$ for incompressible flow. The velocities in the first and second surface integrals in Eq. (2) are the prescribed tangential and normal velocity boundary conditions, respectively. Note that $\vec{U}_\infty = 0$ for internal flow problems.

The volume and surface integrals in Eq. (2) are discretized using N_v smooth vortex particles and N_p triangular panels with piecewise constant vortex and source strengths, respectively. The fluid vorticity is discretized using the following smooth particle description:

$$\vec{\omega}(\vec{x}) = \sum_{j=1}^{N_v} \vec{\Gamma}_j g_{\sigma_j}(|\vec{x}-\vec{x}_j|) \tag{3a}$$

$$g_\sigma(|\vec{x}|) = \frac{3}{4\pi\sigma^3}\exp(-(|\vec{x}|/\sigma)^3) \tag{3b}$$

The corresponding vortical velocity due to the Biot–Savart volume integral in Eq. (2) is

$$\vec{u}_\omega(\vec{x}) = \vec{\nabla} \times \int_V \vec{\omega}(\vec{x}')G(\vec{x},\vec{x}')dV(\vec{x}') = \sum_{j=1}^{N_v} K_{\sigma_j}(\vec{x}-\vec{x}_j) \times \vec{\Gamma}_j \tag{4a}$$

$$K(\vec{x}) = -\frac{\vec{x}}{4\pi|\vec{x}|^3} \tag{4b}$$

$$K_\sigma(\vec{x}) = K(\vec{x})[1-\exp(-(|\vec{x}|/\sigma)^3)] \tag{4c}$$

The particle velocity gradient is obtained by differentiating Eq. (4) directly.

Given the particle discretization of the vorticity field via Eq. (3), the vorticity transport equations (1) are solved for each vortex particle in a viscous splitting strategy, whereby the convection and stretch of vorticity are evaluated along the trajectory of the particles in the Lagrangian frame, followed by the solution of the diffusion equation in the Eulerian frame:

$$\frac{D\vec{x}_i}{Dt} = \vec{u}(\vec{x}_i) \tag{5a}$$

$$\frac{D\vec{\Gamma}_i}{Dt} = \vec{\Gamma}_i \cdot \vec{\nabla}\vec{u}(\vec{x}_i), \quad i = 1, 2, \ldots, N_v \tag{5b}$$

$$\frac{\partial\vec{\omega}(\vec{x}_i)}{\partial t} = \frac{1}{\mathrm{Re}}\nabla^2\vec{\omega}(\vec{x}_i) \tag{5c}$$

Equations (5a) and (5b) are integrated in this work using a second-order Runge–Kutta. Diffusion, Eq. (5c), is evaluated using the vorticity redistribution method (VRM) [10,11]. Vorticity redistribution method works by redistributing the vectorial circulation, $\vec{\Gamma}_i$, of each diffusing particle to its neighboring particles such that the moments of the diffusion equation are conserved up to an arbitrary order; in this work, up to and including the second-order moment. The latter results in an underdetermined linear system of equations for each particle, which is solved in the L_∞-norm using linear programing. The details of VRM are beyond the scope of this paper but can be found, for example, in [9–11].

B. Boundary Element Method

The velocity boundary conditions are imposed in this work by discretizing the boundary into N_p contiguous triangular panels and obtaining the unknown surface-tangent vortex sheet strengths (two per element in 3D) via the solution of the following Fredholm boundary integral equation of the second kind in a collocation formulation:

$$\vec{n}(\vec{x}_i) \times \left\{ \frac{1}{2}\vec{\gamma}(\vec{x}_i) \times \vec{n}(\vec{x}_i) + \sum_{j=1}^{N_p} \vec{\gamma}(\vec{x}_j) \times \vec{I}_j(\vec{x}_i) = \vec{R}(\vec{x}_i) \right\}, \quad i = 1, 2, \ldots, N_p \tag{6a}$$

$$\vec{I}_j(\vec{x}_i) = \int_{S_j} K(\vec{x}_i - \vec{x}')\, dS(\vec{x}') \tag{6b}$$

$$\begin{aligned} \vec{R}(\vec{x}_i) = {} & \vec{u}(\vec{x}_i) - \vec{U}_\infty(\vec{x}_i) \\ & - \sum_{j=1}^{N_v} K(\vec{x}_i - \vec{x}_j) \times \vec{\Gamma}_j \\ & + \sum_{j=1}^{N_p} (\vec{n}(\vec{x}_j) \cdot \vec{u}(\vec{x}_j))\, \vec{I}_j(\vec{x}_i) \\ & + \sum_{j=1}^{N_p} (\vec{n}(\vec{x}_j) \times \vec{u}(\vec{x}_j)) \times \vec{I}_j(\vec{x}_i) \end{aligned} \tag{6c}$$

Equation (6) is the discrete version of Eq. (2) written in terms of the unknown vortex sheet strengths, $\vec{\gamma}$. Here, both $\vec{\gamma}$ and the wall velocities, $\vec{u}$, are assumed to be piecewise constant along each triangular panel. This simplifies the surface integrals considerably, requiring only the evaluation of Eq. (6b), which is performed analytically in this work [12]. Note that $\vec{I}_j(\vec{x}_i)$ is the velocity at the centroid of panel i induced by a unit source distributed uniformly on panel j. Note also that, for accuracy reasons, the influence of the vortex particles in Eq. (6c) is evaluated using the singular, and not the smooth, velocity kernel K.

C. Algorithm and Implementation

The essence of most fast N-body algorithms is the division of effort into short- and long-range summations. In the present method, all particles and panels are organized into a binary tree structure—a (Variance Approximate Median

Split) *VAMSplit k-d* tree—and a single tree traversal is performed for each leaf node in the tree of targets. That traversal determines which nodes in the source tree are considered near and far, and thus whether their influences will be calculated using short-range (direct summation) or long-range (spherical multipoles cast into Cartesian coordinates [5]) methods.

For the vortex particle treecode, the trees, multipole coefficients, and interaction lists are built using multithreaded CPU routines. In a CPU-only implementation, these routines account for about 5% of the total CPU time. Then lists of the particles' locations (and index pointers used to delimit the particles into target tree nodes) are sent to any GPUs that may be present. The algorithm then runs two Compute Unified Device Architecture (CUDA) kernels serially: one to compute the near-field (particle) influences on each target point, and the other to compute the far-field (multipole) influences (CUDA is NVIDIA's higher-level programing language [13]). For each of these two kernels, all necessary source data and the respective interaction lists are sent to the GPU. The load is split among as many GPUs as are present in the system using Open MultiProcessing (OpenMP) multithreading, and within each GPU/thread the load is distributed into several-second chunks to avoid problems that non-Tesla devices have with long kernel run-times. The resulting velocities and velocity gradients are moved off the GPU and stored in main memory.

Similarly, the dense influence matrix Eq. (6) for the BEM can be considered to have short- and long-range components. The short-range entries are computed and stored explicitly using standard sparse-block-matrix storage formats, while the far-field components are approximated using multipole multiplication. The direct portion of the influence matrix is sparse and is calculated once at the beginning of the simulation using the GPUs. As with the treecode, the work is divided evenly among all available GPUs and potentially broken up into even smaller blocks to fit into GPU memory. Each 2-by-2 block in the influence matrix corresponds to the effect of a unit-strength circulation along each of the two axes of the source panel's tangent vectors on each of the two axes of the target panel. The code also supports solving for the unknown scalar source strength, in which case each entry in the influence matrix contains the influence of a unit-strength potential source distribution on the normal vector of the target panel. These coefficients, whether for source or vortex panels, are the result of an analytic integration of Eq. (6b) and involve 334 floating operations (FLOPs) for the CPU version and 340 FLOPs for the GPU version. The solution of the BEM matrix is achieved iteratively using the generalized minimal residual (GMRES) method. During the GMRES solver iterations, the sparse (direct) matrix–vector multiplication is performed in multiple threads on the CPU. The far-field portion of the matrix multiplication is then performed using multipole multiplication on the GPU, the coefficients of which are first recalculated on the CPU before every solver iteration. Particle splitting and merging operations are performed to keep the particle distribution uniform. This is in addition to the particle generation capability of VRM to account for diffusion of vorticity accurately. These three routines have not been ported to the GPU, but of them, only VRM takes more than a small amount of time.

The sequence of substeps in one time step using a second-order Runge–Kutta for advection is as follows:

1) solve the BEM equations for the unknown surface element vortex strengths. This consists of the following substeps:
   a) transform the surface elements to their proper positions for the given simulation time;
   b) create and save the sparse component of the influence matrix if that has not yet been done;
   c) remove any particles that are within a small threshold distance (typically $0.5\delta_v$) of any surface element;
   d) compute the right-hand side of the BEM equations via GPU-accelerated treecode summation of the analytic influence of all particles on all panels;
   e) iterate using GMRES until changes in the solution vector between steps are less than a threshold (usually 10^{-5}); each iteration involves the following steps:
      i) using the previous step's solution vector, compute the multipole moments for each node in the panel tree structure;
      ii) for each group of panels, traverse the tree and sum the far-field component of the influence matrix using multipole multiplication (GPU);
      iii) for each group of panels, march through and sum the near-field (sparse) component of the influence matrix, as computed and stored earlier (CPU);
      iv) divide by the diagonal to determine the new solution vector.
2) compute the velocity and velocity gradient on all particles using the discretized form of Eq. (2) and a fast multipole treecode solver;

3) advect each particle according to its local velocity, Eq. (5a), and update each particle circulation according to the local stretch, Eq. (5b);
4) repeat the preceding steps to complete the second-order advection;
5) split in two any particles that have been elongated beyond a threshold; use the local vorticity gradient to place and orient the new particles to maintain the local vorticity curvature;
6) merge any particles that approach each other to within a small threshold distance;
7) create new particles just above the surface elements with circulations drawn from the solution of the BEM;
8) use VRM to diffuse vorticity among particles, Eq. (5c);
9) optionally remove any particles with circulation lower than a threshold.

III. Results

Before any performance results are reported, and to avoid any misunderstandings, it is important to state the nature of the CPU–GPU comparisons. All CPU code was created in Fortran 90 or ANSI C, with an effort made to optimize the algorithms and the code itself, but with no explicit user-optimized machine language or streaming single instruction multiple data extensions (SSE) instructions. The code was compiled with Gnu *gfortran* and *gcc* using the following performance options: `-O2 -ffast-math -funroll-loops`. The CPU code has been multithreaded using OpenMP extensions, yielding excellent parallel efficiency (>95% for eight cores) for all routines, which will be compared with their GPU equivalents. All problems fit in memory, so no swapping was necessary. Two machines were used for these performance results: one contained a single four-core AMD Phenom 9850 CPU clocked at 2.5 GHz, and the other contained two four-core Intel Xeon E5345 CPUs at 2.33 GHz (both 64-bit chips running 64-bit Linux operating systems). To create a fair comparison, the GPU-accelerated results are compared to those generated by efficient multi-threaded applications running on similar-era multicore CPU systems such as these.

The data structures used by both CPU and GPU codes were designed to prevent cache misses, at the same time retaining enough flexibility to be used for multiple purposes. Specifically, the particle locations, radii, and strengths each have their own arrays, as do the panel node locations, node pointers, and panel strengths. Within each array, though, the elements are frequently reordered to reflect the organization imposed by the (frequently remade) binary tree. Thus, no "pointer-chasing" occurs when traversing the tree and looping over blocks of nearby particles or panels. The only exception to this is the panel node locations, which for the GPU routines are copied into one 9-by-N_p array to facilitate loading by the GPU kernels, whereas the CPU routines do no such expansion and thus the accesses are to random positions in memory and are slower.

All particle and panel data are stored in single-precision, and all GPU and most CPU routines perform arithmetic in single-precision (though hardware can internally increase the precision during some operations). Again, the exception to this should be noted: the analytic panel influence routine runs in double-precision on the CPU. As a general rule, however, we have found that switching all CPU arithmetic to double-precision reduces FLOP/s by no more than 10%.

All GPU code was created using version 2.2 of NVIDIA's CUDA language [13], which uses *gcc* under the hood to compile binary versions of the host (CPU) code. Optimization options passed to the compilation stage include `-O2 -use_fast_math`, but those only affect device (GPU) code. The algorithms and logic used in the GPU code are very similar to those of the CPU code: no recursion or excess subroutine calls, frequent manual loop unrolling, and mostly the same operation order (although the compiler can and will change much of this). The primary test machines were connected to two GPUs: the Phenom to two NVIDIA GeForce GTX 275 GPUs with shader clock speeds of 1.512 GHz, and the Xeons to two of the four GPUs in a Tesla S1070 at 1.44 GHz. Each of these GPUs contains 240 processing elements (PEs). *All GPU performance results include the time required for all threads to call the C host code from Fortran, move all data to the GPU, call the GPU kernel, and clean up the GPU memory before returning.*

A. Vortex Particle Method

To determine the peak expected efficiency of the vortex particle method on GPU hardware, we tested the direct particle–particle Biot–Savart interactions, which is the portion of the calculation with the highest arithmetic intensity. The velocity, Eq. (4), and velocity gradients were calculated for particles distributed randomly in a unit cube with overlapping core radii of size $\sigma = 1.5\delta_v$. Each particle–particle interaction computes the influence of the vortex

velocity and its nine-component velocity gradients on a target point using 66 FLOPs (counting `sqrt` and `exp` instructions once). At its peak, the dual-GPU version conducted *14.4 billion* interactions per second, or *949 GFLOP/s*, counting the time to transfer data to and from the GPU (which is serialized in the case of multiple GPUs). This is *201 times faster* than the multithreaded quad-core CPU version, which achieves a peak rate of 4.7 GFLOP/s, and 89 times faster than the eight-core Xeon system (10.6 GFLOP/s). The theoretical peak performance of the GPU hardware is 2177 GFLOP/s (240 PEs times 1.512 GHz shader clock times three floating point operations per cycle [13] times two GPUs), which makes the computational efficiency of the particle velocity and gradient evaluations 43.6%. This is about as well as can be expected, considering the presence of `sqrt` and `exp` operations and the fact that much of the computation does not use the combined one-cycle FMAD + FMUL operations. Most GPU direct N-body implementations exhibit equivalent efficiencies [4,14,15] when instructions are counted similarly. *Note that the earlier-mentioned performance of nearly 1 TeraFLOP/s of real computation was achieved on a computer that cost no more than 1200 USD in 2009.*

Our previous effort [3] demonstrated 218 GFLOP/s on a NVIDIA 8800 GTX GPU (128 PEs at 1.35 GHz), or 63% efficiency. Although that may seem better than our current effort, note that the 8800 GTX can only perform one FMAD operation per PE per cycle (two FLOPs), whereas the new hardware can perform three FLOPs per cycle. If we calculate the efficiency of the old method assuming three FLOPs per cycle, it drops to 42.1%, or slightly less than that of the present work. This indicates that the new code does not yet take advantage of the combined one-cycle FMAD + FMUL operations available on new hardware. In contrast, the previous CPU performance (on a dual-core Opteron 2216HE at 2.4 GHz) peaked at 1.6 GFLOP/s, whereas the new version achieves 4.7 GFLOP/s on the quad-core Phenom at 2.5 GHz. This represents a *41% improvement* in FLOPs/Hz efficiency for the CPU code, making the overall GPU speedup of 201 even more impressive.

Comparing treecode performance is much trickier than direct summation performance, as the parameters governing the tree structure and tree traversal have a substantial effect on the accuracy and computational effort. In the direct summation tests mentioned earlier, both GPU and CPU versions required the same number of FLOPs, making comparisons easy. However, to obtain the best treecode performance on each type of hardware for a given level of accuracy, variations in these parameters often result in the GPU performing many more arithmetic operations [3]. Therefore, comparisons will be made using the parameters that produce the fastest wall-clock times for each type of hardware for the same level of accuracy, regardless of the number of FLOPs required.

Setting the tree and tree traversal parameters to achieve a mean velocity error of 2×10^{-4}, and solving for the velocity and nine-component velocity gradient tensor on N_v vortex particles distributed randomly in a unit cube, with smoothing radius $\sigma = 1.5 N_v^{-1/3}$, results in the numbers appearing in Fig. 1. Although many authors report exemplary speedups when porting code to the GPU, most of those algorithms, such as the direct summation, are trivially parallelizable. The present results show that an algorithm with better big-$\mathcal{O}$ performance can also benefit from GPU hardware; in this case, showing a *109-fold speedup* vs. the direct method at 5 M particles and with a break-even point for the better algorithm at 50 k particles and 0.2 s runtime.

Also in Fig. 1 are comparisons between the fastest CPU-only runs and the fastest CPU/GPU runs for several systems. The CPU treecode used bucket sizes of 64 for the trees, multipoles up to order 9, and a unique tree traversal was performed for each particle. The GPU treecode used bucket sizes of 128 or 192, seventh order multipoles, and a unique tree traversal for each leaf node of particles. Despite doing more work, the GPU version on the four-core Phenom system completes the calculation *60.7 times faster* than the CPU version, and the GPU version on the eight-core Xeon system finishes *43.3 times faster* than the pure CPU version. These speedups are the result of various algorithmic and implementation changes from the previous hybrid CPU/GPU particle treecode [3], which have improved upon what was once a 16.9-fold speedup on a machine with one 8800 GTX and a dual-core Opteron. The machine used in the previous work now has two dual-core Opteron CPUs, and with the new treecode solves for the velocities and gradients on 500,000 particles in 5.01 s instead of 14.9 s. It is worth mentioning that both the CPU and GPU versions exhibit approximately $\mathcal{O}(N_v^{1.05})$ scaling at large N_v.

Gumerov and Duraiswami [5] compare their nonoptimized single-threaded FMM on a 2.67 GHz Intel Core 2 Duo with their GPU FMM on a 8800 GTX GPU, and they show 72-fold speedup for 1 M particles with multipole order 8 and 29-fold speedup for multipole order 4 (the accuracy of which corresponds to the present method more closely). Their GPU FMM used a singular particle kernel, which obviates the cost associated with using the Gaussian smoothing function used in this work, neither did their computations involve the evaluation of the nine-component

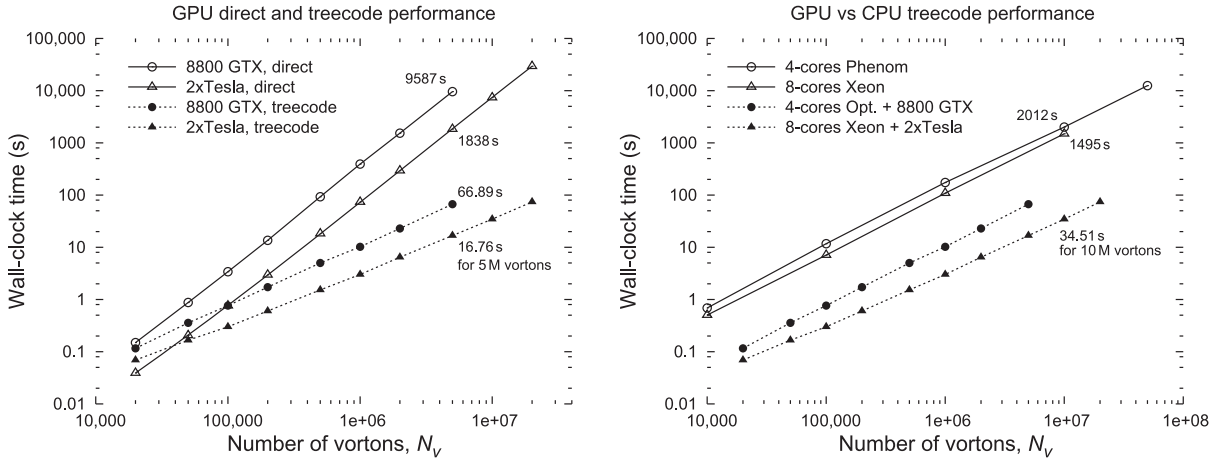

Fig. 1 Left: Performance boost obtained moving from a naive direct summation algorithm to a multipole treecode for the N-vortex problem for various GPU hardware. "8800 GTX" is a system with two dual-core Opteron 2216HE CPUs and a single 8800 GTX GPU with 128 PEs, "2x Tesla" is the eight-core Xeon system described in the text. Right: Performance of multipole treecode on CPU-only and hybrid CPU/GPU systems.

velocity gradient tensor, which increases the computational overhead by at least a factor of 2. Yokota et al. [6] ported the FMM algorithm of [5] to CUDA for smooth vortex particles and found a 60-fold performance gain between a single NVIDIA 8800 GT (112 PEs) and a *single* 3 GHz core of an Intel Core 2 Duo. Scaling was found to be $\mathcal{O}(N^{1.15})$ for both CPU and GPU versions. It is noted here that the clock rate on the NVIDIA 8800 GT (112 PE) used in [6] is 1.56 GHz, whereas the clock rate on the NVIDIA 8800 GTX (128 PE) used in this work is 1.35 GHz. Therefore, the two cards are equal in their performance: $(112 \times 1.56)/(128 \times 1.35) = 1.0$. On the other hand, the CPU in [6] is 35% faster than the Opteron 2216HE reported here. Nevertheless, the two GPU solutions are nearly identically fast at 500 k particles.

Although future improvements in single-system performance can be expected, true leaps in capability can only be achieved using clusters of GPU-enabled computers. To support vortex particle simulations in a distributed-memory environment, the present method uses message passing interface (MPI) commands to allow one large problem to be split evenly across many computers, each with one or more CPU cores and one or more GPUs. The method used is very similar to that of Salmon [16], with minor modifications. Particles are split among processors using a recursive binary tree algorithm, where dynamic load balancing is achieved by weighting each particle based on the actual performance of the process within which it is stored. Once distributed, each processor recursively creates a locally essential tree (LET) containing the particles and multipole moments from other processors that are determined to be necessary to complete its own treecode calculation. Once the LET is filled, the CPU or GPU treecode algorithm computes the velocity and velocity gradients of each of its own particles due to the influence of its local particles and the LET.

The performance of the MPI-parallel treecode is measured with parallel efficiency, $e = t_{\text{serial}}/(N_{\text{proc}}\, t_{\text{parallel}})$, where $e = 1$ means perfect parallelization with no effort lost. Network communication and any extra work involved in splitting up a large problem both push efficiency down. The code was exercised on *Lincoln*, a 192-node hybrid CPU/GPU cluster at National Center for Supercomputing Applications (NCSA), on 1–64 nodes using OpenMP to spread the load on each node to all available CPU cores and GPUs. The results appear in Fig. 2. Note that in this context, t_{serial} from the earlier-mentioned equation represents a single process, but with eight threads operating in parallel, and not a serial single-threaded application. Thus, the CPU version on 64 nodes, which took 2.144 and 26.77 s for 1 M and 10 M particles, respectively, was running on 512 CPU cores and achieved 88% efficiency. The GPU version running on 32 nodes took 0.263 and 1.928 s to solve the same problems, but was effectively splitting the work over 256 CPU cores and 15,360 GPU PEs, explaining the 43 and 67% efficiencies. For the large problem size (100 M particles), the load can be split more evenly across all devices, and the calculation completes in 18.36 s

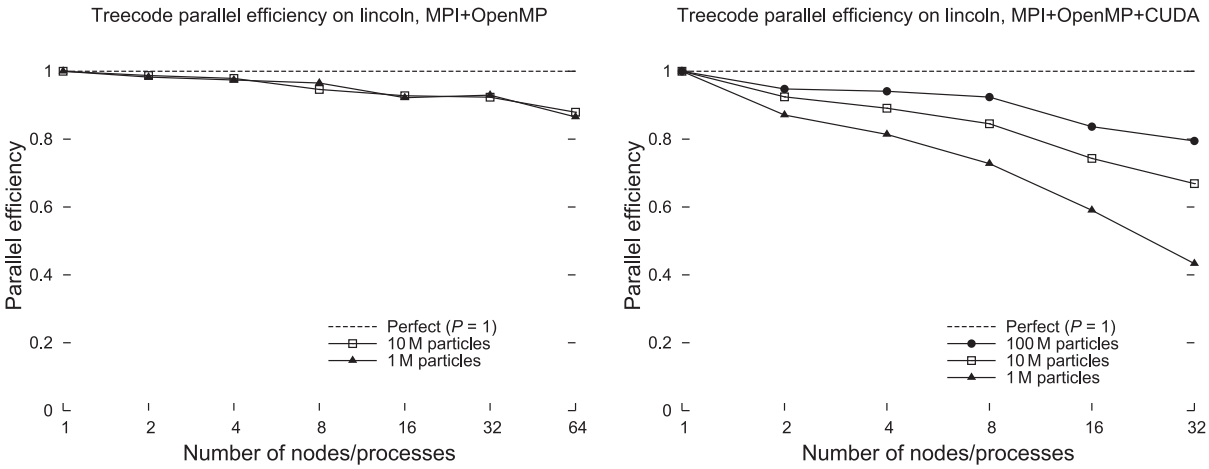

Fig. 2 Parallel efficiency of all stages of hardware-optimized treecode without (left) and with (right) GPUs enabled. Each node of NCSA's *Lincoln* contains two quad-core Xeon CPUs and connections to two of the four GPUs in a Tesla S1070.

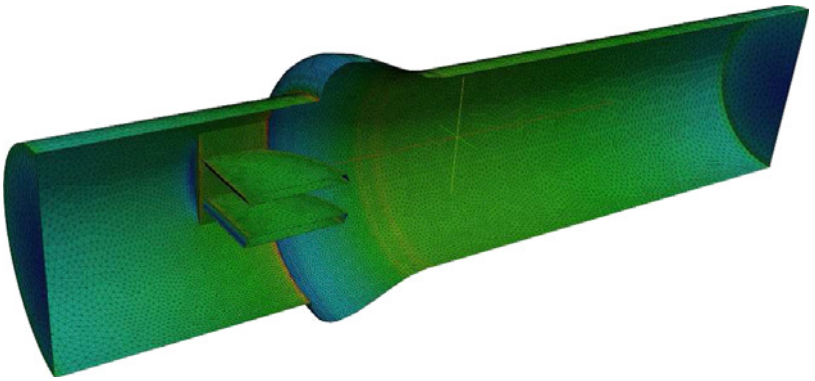

Fig. 3 Triangulated geometry of aorta and St. Jude Medical mechanical heart valve, cut along the center plane to show the interior, with 89,040 panels.

with 80% efficiency. For the CPU version to match that performance, it would require approximately 1024 nodes, or 8192 CPU cores. Note that no effort was made to overlap communication and computation, which leaves room in the future to improve the software's parallel efficiency.

For comparison, the CPU-only FMM of Yokota et al. [6] achieves a parallel efficiency of 68% on 32 processes for 1 M particles (on 32 nodes). Their parallel GPU–FMM, run on 32 nodes each with two GPUs and using 64 processes, returned efficiencies of 24% for 1 M particles and 60% for 10 M.

B. Boundary Element Method

To test the performance of the hybrid CPU/GPU BEM, we used Eq. (6) and the algorithm in Sec. II.C to solve for the unknown source strengths on a triangulated mesh of a model St. Jude Medical mechanical heart valve and aorta [17]. The surface was discretized into a number of flat triangular panels, and runs were made for a variety of resolutions from 5000 to 1.425 million triangles. The flat disks on either end represent inlet and outlet, and have a prescribed unit normal velocity boundary condition. Figure 3 illustrates the geometry and solution for the case with 89,040 triangles.

Because the present method stores and reuses the coefficients of the influence matrix, only the smallest models (with 5 k, 8 k, 22 k, and 32 k panels) could be solved using the direct method due to the memory required. The

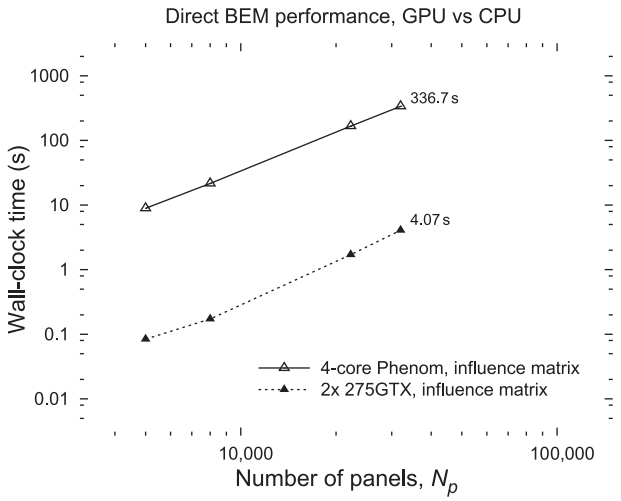

Fig. 4 Time required by CPU and GPU methods to build and save the dense influence matrix from direct BEM of identical geometry with differing panel counts. Central Processing Unit version used quad-core Phenom at 2.5 GHz, GPU version used two GTX 275 at 1.512 GHz.

performance of the method on these models appears in Fig. 4. The speedup peaks at 125 for the 8 k-panel model, and drops to 83 for the largest case (which involved 11 GPU kernel calls per GPU and 4 GB of memory transfer). The four-thread CPU version peaked at 0.915 GFLOP/s and the dual-GPU version peaked at 112.7 GFLOP/s; the performance was brought down by the large number of divisions and trigonometric operations in the 340-FLOP analytic kernel, and by the fact that every result was written to main memory. Takahashi and Hamada's [8] GPU-accelerated direct BEM solver differs somewhat in that it does not store the full influence matrix, but instead recreates it during each solver iteration. Storing the influence coefficients significantly benefits problems with computationally intensive coefficients or problems that require large numbers of solver iterations, but it limits the problem size, especially when using direct solution methods. Each entry in their influence matrix is the result of integration over seven Gaussian quadrature points, and each quadrature point calculation requires 36 FLOPs (counting transcendental functions as one FLOP for consistency with the present results). Using a SSE-optimized, single-precision, four-core CPU version, their method performs one GMRES iteration for their 32 k-panel geometry in about 47 s; and on one 8800 GTS GPU the same iteration takes about 5 s. Although these times compare well with the present method, the computations did not involve nearly the amount of data transfer and memory-writing required by our method, and also must be repeated at every iteration. Speedups in [8] peaked at 11.5 vs their optimized CPU code and at 22.6 vs the nonoptimized version. Recall that results in this work peak at 125.

Just as the vortex particle method benefits from using a treecode instead of direct summations, so too should the BEM. It is the goal of this work to merge the performance benefits of GPU computing with the algorithmic advantages of a multipole-accelerated treecode. In this case, only a fraction of the dense influence matrix is precomputed and saved, with the remainder (the far-field influences) being approximated with multipole expansions once per solver iteration. For both CPU and GPU versions, it was found that a tree built with a bucket size of 32 performed best; the multipole order was 6, and interaction lists were created for each leaf node of panels. A variety of resolutions of the earlier-mentioned geometry were tested with these parameters, with the GMRES solver requiring 21–27 iterations to converge to 10^{-5} (for comparison, the spheres in the following section need 3–6 iterations). The performance of the entire GPU–BEM solution relative to the CPU-only version appears in Fig. 5 and reaches a *22.5-fold speedup*. The direct portion of the influence matrix is created *99.7 times faster* than the four-core CPU version, comparable to the direct method mentioned earlier. The GPU version exhibits $\mathcal{O}(N^{1.05})$ scaling for the entire solution and the CPU version $\mathcal{O}(N^{1.1})$.

For the case with 1.4 M unknowns, the direct part of the influence matrix was computed in 2.938 s, and the entire calculation was complete in 57.3 s. Each iteration of the GMRES solver is composed of three steps: the direct portion

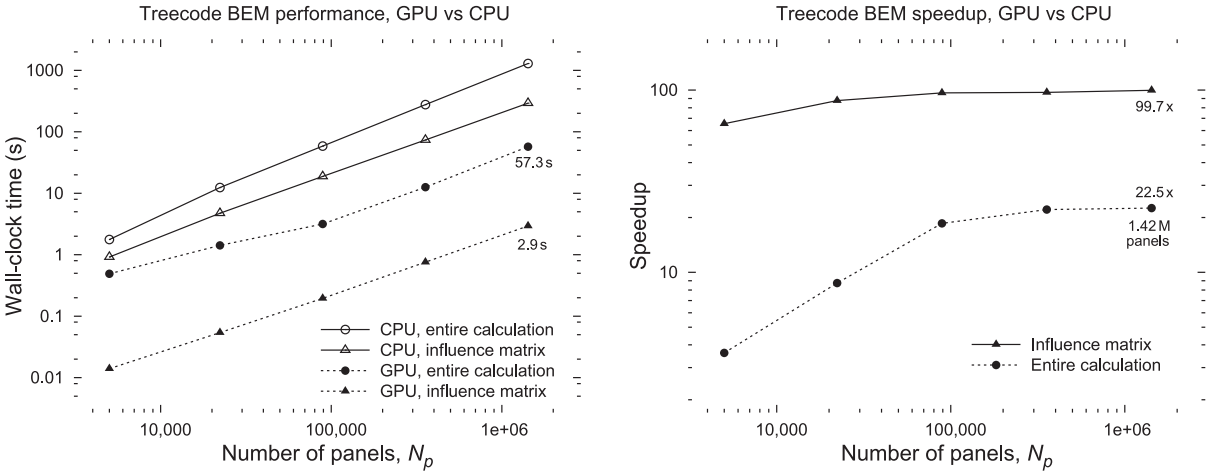

Fig. 5 Left: Performance of CPU and hybrid CPU–GPU solvers on BEM problem, broken down by calculation and active hardware. Right: speedup between CPU-only and hybrid methods. Central processing unit version used quad-core Phenom, GPU version used two GTX 275.

is performed on the CPU, the multipole coefficients are made on the CPU, and the multipole multiplication is done on the GPU. For this large problem, those steps took, on average, 0.735, 0.463, and 0.507 s, respectively. Because the greatest portion of the work in the GMRES iterations is done on the CPU, the second GPU was of little help. The same problem performed with just one GPU active created the influence matrix in 5.20 s and finished the entire problem in 62.5 s—still a 20.6-fold speed improvement over the four-core CPU version. For the sake of comparison, solving the 1.4 M unknown problem using the direct method would require 9 h using the GPU for the influence matrix, and eight days using the four-core CPU. Those numbers increase to 2 and 185 days if the influence matrix is recomputed at every iteration.

C. Impulsively Started Flow Over a Sphere

The GPU-accelerated BEM and the parallel GPU-accelerated vortex particle solver are integrated into Ω-*Flow*, which allow multithreaded and multiGPU simulations of vortex-dominated flow. The complete method is tested by running simulations of impulsively started flow over a sphere with Reynolds numbers, based on the diameter and freestream velocity, ranging from 500 to 4000 on a single node containing one or two four-core CPUs and two 240-PE GPUs. Triangulated spheres with 5120, 20,480, 81,920, and 327,680 elements are used for the $\mathrm{Re} = 500$, 1000, 2000, and 4000 cases, respectively. Time steps are $\Delta t = 0.04$, 0.02, 0.01, and 0.005. The nominal interparticle spacing and core size are given by $\delta_v = \sqrt{8\Delta t/\mathrm{Re}}$ and $\sigma = 1.5\delta_v$, respectively, yielding $\sigma = 0.0379$, 0.0190, 9.49e-3, and 4.74e-3.

Figure 6 shows the azimuthal vorticity at the center plane for each of the four Reynolds number studies at $t = 3$. Illustrated are all of the particles in a band $2\sigma/3$ wide. At this stage, each case shows the formation of the primary wake vortex ring, although the rings are significantly stronger for the higher Re cases, as expected. In addition, as the Reynolds number increases, so does the number of smaller scale recirculation bubbles in the near-separation region. The $\mathrm{Re} = 4000$ case contains four stagnation rings on its downwind hemisphere. The full three-dimensional simulations at $t = 3$ contain 183 k, 908 k, 4.75 M, and 26.46 M particles, respectively. Despite aggressive particle removal, these simulations are still uniform resolution, and thus significant computational savings could be achieved by using adaptive resolution methods.

Figure 7 depicts the onset of asymmetry for the cases with $\mathrm{Re} = 500$, 1000, and 2000. The $\mathrm{Re} = 4000$ sphere remained axisymmetric until $t = 3$, when the run was ended. The models received no initial "nudge" to force the primary ring to shed in a predictable direction, and so the data were rotated for easier visualization. All runs were visibly axisymmetric at $t = 2$ and 4, and asymmetric afterward. The first shed horseshoe vortex is much larger at $t = 10$ for the lower-Re cases, and the tails are longer as Re decreases.

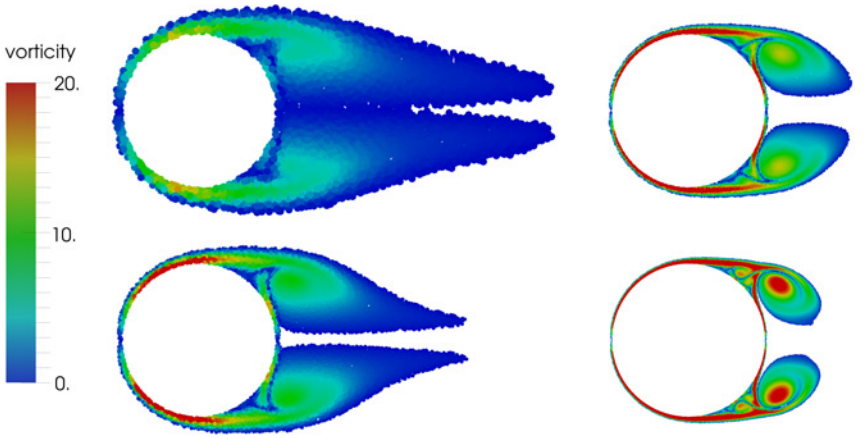

Fig. 6 Azimuthal vorticity at the center plane around an impulsively started sphere at $t = 3$; Re = 500 (top left), 1000 (bottom left), 2000 (top right), 4000 (bottom right).

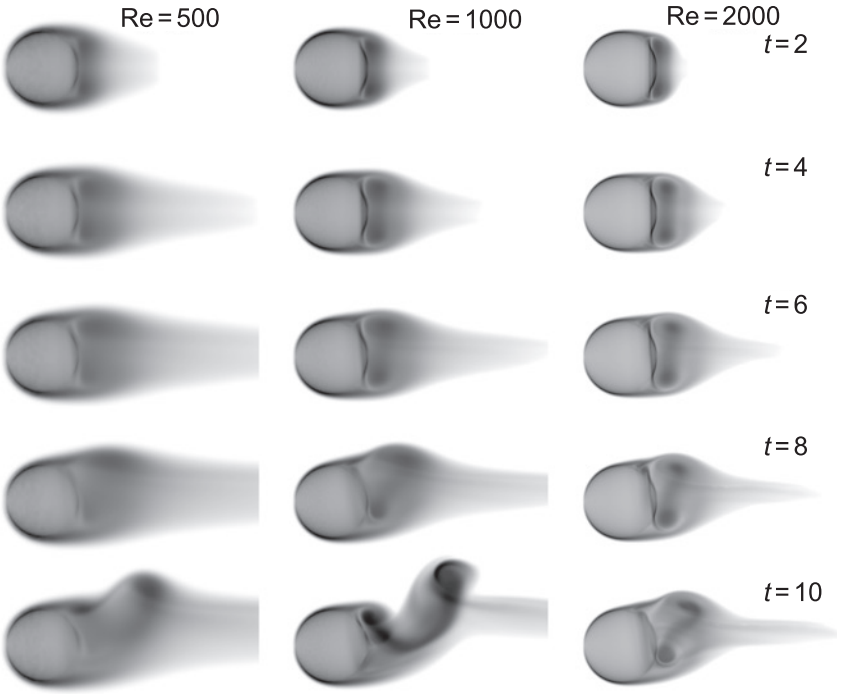

Fig. 7 Vortex shedding from impulsively started sphere at Re = 500, 1000, and 2000, and at $t = 2, 4, 6, 8, 10$; "x-rays" of vorticity strength $|\vec{\Gamma}|$.

IV. Conclusion

We have implemented and tested what we believe to be the first GPU-accelerated treecode BEM, and have shown almost 100-fold speedup for the creation of the influence matrix, and over 20-fold speedup for the entire solution. Using this method, a single desktop computer costing less than 1000 USD can solve a BEM with 1.4 M unknowns in just about 1 min, much faster than the 8 h required by a GPU-accelerated direct method (on 1.06 M unknowns) [8]. In addition, we have demonstrated a vortex particle method that solves for the velocities and velocity gradients of 100 M

particles in 18.4 s on a cluster of 32 multicore, multiGPU systems. The effort put into developing data-parallel and distributed-memory computational methods now should reward the simulation scientist in the future, as the largest performance gains are expected to come from the combination of cluster computing and more powerful GPUs.

Acknowledgments

The development of parallelization on a cluster of GPUs was supported by Phase I SBIR Contract W911W6-10-C-0018 from the US Army Research, Development, and Engineering Command (AMRDEC). The implementation and testing of the parallelized treecode as well as the simulations in this paper were supported in part by the National Science Foundation through TeraGrid resources provided by NCSA under grant number TG-ASC090070. The remainder of this work was supported by Award Number R44RR024300 from the National Center For Research Resources. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Center For Research Resources or the National Institutes of Health.

References

[1] Barnes, J. E., and Hut, P., "A Hierarchical $\mathcal{O}$ (N log N) Force Calculation Algorithm," *Nature*, Vol. 324, 1986, pp. 446–449.

[2] Greengard, L., and Rokhlin, V., "A Fast Algorithm for Particle Simulations," *Journal of Computational Physics*, Vol. 73, 1987, pp. 325–348.

[3] Stock, M. J., and Gharakhani, A., "Toward Efficient GPU-Accelerated N-Body Simulations," *Proceedings of the 46th AIAA Aerospace Sciences Meeting*, Reno, NV, Jan. 2008, AIAA-2008-608.

[4] Harris, M., "Mapping Computational Concepts to GPUs," *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, ACM Press, New York, NY, 2005, p. 50.

[5] Gumerov, N. A., and Duraiswami, R., "Fast Multipole Methods on Graphics Processors," *Journal of Computational Physics*, Vol. 227, 2008, pp. 8290–8313.

[6] Yokota, R., Narumi, T., Sakamaki, R., Kameoka, S., Obi, S., and Yasuoka, K., "Fast Multipole Methods on a Cluster of GPUs for the Meshless Simulation of Turbulence," *Computer Physics Communications*, Vol. 180, No. 11, 2009, pp. 2066–2078.

[7] Buatois, L., Caumon, G., and Lévy, B., "Concurrent Number Cruncher—A GPU Implementation of a General Sparse Linear Solver," *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 24, No. 3, 2008, pp. 205–223.

[8] Takahashi, T., and Hamada, T., "GPU-Accelerated Boundary Element Method for Helmholtz' Equation in Three Dimensions," *International Journal for Numerical Methods in Engineering*, Vol. 80, No. 10, 2009, pp. 1295–1321.

[9] Gharakhani, A., and Stock, M. J., "3-D Vortex Simulation of Flow Over a Circular Disk at an Angle of Attack," *Proceedings of the 17th AIAA Computational Fluid Dynamics Conference*, Toronto, Ontario, Canada, 6–9 June 2005, AIAA-2005-4624.

[10] Shankar, S., and van Dommelen, L., "A New Diffusion Procedure for Vortex Methods," *Journal of Computational Physics*, Vol. 127, 1996, pp. 88–109.

[11] Gharakhani, A., "Grid-Free Simulation of 3-D Vorticity Diffusion by a High-order Vorticity Redistribution Method," *Proceedings of the 15th AIAA Computational Fluid Dynamics Conference*, Anaheim, CA, 2005, AIAA-2001-2640.

[12] Gharakhani, A., and Ghoniem, A. F., "BEM Solution of the 3D Internal Neumann Problem and a Regularized Formulation for the Potential Velocity Gradients," *International Journal for Numerical Methods in Fluids*, Vol. 24, No. 1, 1997, pp. 81–100.

[13] NVIDIA, "NVIDIA CUDA Programming Guide," Technical Report, NVIDIA Corporation, Santa Clara, CA, April 2009, Version 2.2.

[14] Hamada, T., and Iitaka, T., "The Chamomile Scheme: An Optimized Algorithm for N-Body Simulations on Programmable Graphics Processing Units," *ArXiv Astrophysics e-prints*, Vol. astro-ph/0703100, 2007.

[15] Nyland, L., Harris, M., and Prins, J., "Fast N-Body Simulation with CUDA," *GPU Gems 3*, edited by H. Nguyen, Chap. 31, Addison-Wesley, 2007, pp. 677–695.

[16] Salmon, J. K., "Parallel Hierarchical N-Body Methods," Ph.D. Thesis, California Institute of Technology, 1991.

[17] Gharakhani, A., "Grid-Free LES of Bileaflet Mechanical Heart Valve Motion," *Proceedings of the BIO2006, ASME 2006 Summer Bioengineering Conference*, Amelia Island, FL, 21–25 June 2006, BIO2006-157424.

James Hargrave
Associate Editor