40th Fluid Dynamics Conference and Exhibit
28 June - 1 July 2010, Chicago, Illinois

AIAA 2010-5099

A GPU-accelerated Boundary Element Method and Vortex Particle Method

Mark J. Stock* and Adrin Gharakhani†

Applied Scientific Research, Santa Ana, California

Vortex particle methods, when combined with multipole-accelerated boundary element methods (BEM), become a complete tool for direct numerical simulation (DNS) of internal or external vortex-dominated flows. In previous work,[1] we presented a method to accelerate the vorticity-velocity inversion at the heart of vortex particle methods by performing a multipole treecode N-body method on parallel graphics hardware. The resulting method achieved a 17-fold speedup over a dual-core CPU implementation. In the present work, we will demonstrate both an improved algorithm for the GPU vortex particle method that outperforms an 8-core CPU by a factor of 43, but also a GPU-accelerated multipole treecode method for the boundary element solution. The new BEM solves for the unknown source, dipole, or combined strengths over a triangulated surface using all available CPU cores and GPUs. Problems with up to 1.4 million unknowns can be solved on a single commodity desktop computer in one minute, and at that size the hybrid CPU/GPU outperforms a quad-core CPU alone by 22.5 times. The method is exercised on DNS of impulsively-started flow over spheres at Re=500, 1000, 2000, and 4000.

I. Introduction

Lagrangian vortex methods are a novel alternative to the more common Eulerian methods for direct numerical simulation of fluid flows for several reasons: absence of numerical diffusion, no firm CFL advection limit, and substantially simpler mesh preprocessing. Their disadvantages include more narrow applicability, somewhat more complex code structure, and greater calculation cost per time step. There is ongoing research to address vortex methods' applicability, but in the present work we aim to improve its performance.

A vortex particle method solves the incompressible vorticity transport equations in Lagrangian form by discretizing vorticity as particles. The kinematic velocity-vorticity relationship is solved in a grid-free fashion by convolving the Biot-Savart integral with a Gaussian core function. Fast algorithms for the Biot-Savart integration are available, the two most common being the multipole treecode[2] and Fast Multipole Method (FMM).[3] These methods use hierarchical subdivision of the problem space to reduce computation and are also applicable to gravitational and electrodynamic computations. The present method uses an improvement of a treecode developed previously.[1] The treecode method is theoretically of order $\mathcal{O}(N \log N)$, and results presented below indicate $\mathcal{O}(N^{1.1})$ or better performance.

With the current proliferation of hardware and software for data-parallel GPU (graphics processing unit) 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 operations per second (GFLOP/s) is 2 USD for CPUs and 0.10 to 0.20 for GPUs. The disparity is even higher for single-precision calculations. This reduction in cost is greatest in cases with algorithms that exhibits high arithmetic intensity, or have a very large ratio of instruction count to required memory access. Vortex methods (and other N-body methods) are well-suited to this new programming paradigm, and have demonstrated substantial performance improvements over more traditional multi-core CPUs.[1,4–6]

Previous fast N-body GPU work includes both treecode[1] and FMM[5,6] implementations. Treecode- and FMM-accelerated boundary element methods are common in the literature, but only CPU implementations. Buatois *et al.*[7] implemented a GPU sparse linear solver, which is one component of a dense BEM. More

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

Copyright © 2010 by Applied Scientific Research, Inc. Published by the American Institute of Aeronautics and Astronautics, Inc., with permission.

recently, Takahashi and Hamada[8] presented a GPU-accelerated BEM for Helmholtz' equations, though it used a direct $\mathcal{O}\left(N^2\right)$ method, and did not store coefficients in the influence matrix. No GPU implementations of a treecode or FMM-accelerated boundary element method for flow simulation were found in the literature at the time of writing.

In the present work, we will describe the GPU-accelerated BEM that has been added to Ω-*Flow*, a Fast Multipole treecode solver created at Applied Scientific Research (ASR).[1,9] The following sections will lay out the underlying equations, algorithm and implementation details, and results from applying the method to the unsteady flow over an impulsively-started sphere.

II. Method

ASR's Ω-*Flow* solver is a Lagrangian vortex particle method that uses a hierarchical spatial decomposition scheme and high-order multipole treecode solver[2] 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 work[1,9] and the present method are sufficient that the important parts of the formulation, algorithms, and implementation will be described below.

II.A. Vortex particle method

The fluid velocity $\vec{u}(\vec{x})$ in a compressible wall-bounded flow is prescribed by the following generalized Helmholtz integral formula in terms of the fluid vorticity $\vec{\omega}$, vortex sheet strength $\vec{\gamma}$, dilatation θ, and the boundary velocity $\vec{u}(\vec{x}')$:

$$\begin{aligned}
\vec{u}(\vec{x}) = \vec{U}_\infty &+ \nabla \times \int_V \vec{\omega}(\vec{x}')\, G(\vec{x},\vec{x}')\, dV(\vec{x}') \\
&- \nabla \int_V \theta(\vec{x}')\, G(\vec{x},\vec{x}')\, dV(\vec{x}') \\
&+ \nabla \times \int_S \big(\vec{\gamma}(\vec{x}') + \hat{n}(\vec{x}') \times \vec{u}(\vec{x}')\big) G(\vec{x},\vec{x}')\, dS(\vec{x}') \\
&- \nabla \int_S \big(\hat{n}(\vec{x}') \cdot \vec{u}(\vec{x}')\big) G(\vec{x},\vec{x}')\, dS(\vec{x}')
\end{aligned} \tag{1}$$

where $G(\vec{x},\vec{x}') = 1/(4\pi|\vec{x}-\vec{x}'|)$ is the Green's function in 3D, $\hat{n}$ is the unit normal into the fluid domain, and $\vec{U}_\infty$ is the freestream velocity. Currently, the method supports only incompressible flows, thus $\theta = \nabla \cdot \vec{u} = 0$, which simplifies the above equation.

The volume and surface integrals in Eqn. (1) are discretized using N_v Gaussian-cored particles having vectorial circulation $\vec{\Gamma}$:

$$\vec{\omega}_\sigma(\vec{x}) = \sum_{j=1}^{N_v} \vec{\Gamma}_j\, g_\sigma(\vec{x} - \vec{x}_j) \tag{2}$$

(where g_σ is a smoothed Dirac delta function) and N_p triangular panels with piecewise constant vortex sheet strength $\vec{\gamma}_p$. The vortex particle velocities $\vec{u}_\sigma$ and their gradients are smooth and are evaluated by convolving the Biot-Savart integral for velocities with a smoothing or core function g with radius σ. For example, the first line in Eqn. (1) becomes:

$$\vec{u}_{\sigma,i}\left(\vec{x}_i\right) = \vec{U}_\infty + \sum_{j=1}^{N_v} K_\sigma\left(\vec{x}_j - \vec{x}_i\right) \times \vec{\Gamma}_j \tag{3}$$

$$K_\sigma(\vec{x}) = K(\vec{x}) \int_0^{|\vec{x}|/\sigma} g(r)\, r^2 dr \tag{4}$$

$$K(\vec{x}) = -\frac{\vec{x}}{4\pi|\vec{x}^3|} \tag{5}$$

$$g(r) = (3/4\pi)\exp\left(-r^3\right) \tag{6}$$

Core radius σ is set to a multiple of the nominal particle spacing δ_v, normally 1.5, to ensure sufficient overlap, and thus, accuracy. The smooth velocity gradient is obtained by differentiating equation (3) directly. The vorticity transport equations are solved in the Lagrangian frame using a second-order forward integrator via the following equations:

$$\frac{d\vec{x}_p}{dt} = \vec{u}_p \tag{7}$$

$$\frac{d\vec{\Gamma}_p}{dt} = \Gamma_p \cdot \nabla \vec{u}_p \tag{8}$$

$$\frac{d\vec{\omega}_p}{dt} = \frac{1}{\mathrm{Re}} \nabla^2 \vec{\omega}_p \tag{9}$$

Diffusion is accomplished using the Vorticity Redistribution Method (VRM),[10, 11] which, in this case, conserves moments up to second order. VRM works by solving for the fraction of the diffusing particle's circulation to redistribute to its neighbors. The implementation results in an underdetermined system of equations that are solved in the L_∞-norm using linear programming. VRM has some advantages over the more common Particle Strength Exchange (PSE) methods.[12] One is that the computational stencil for VRM is much smaller. Performing VRM for one particle typically only requires consideration of the nearest 30 particles, while PSE for viscous diffusion requires about 200. VRM also performs automatic detection and filling of holes in the vorticity support, while PSE requires either frequent regridding or an extra step to add particles where resolution is low or support not continuous. On the other hand, the PSE algorithm is simpler, requires less cached memory, and is more data-parallel. Both methods for diffusion are supported in the software.

II.B. Boundary element method

No-slip/no-flux boundary conditions are imposed by placing vortex sheets at the wall, with their surface-tangent strengths $\vec{\gamma}$ (two per element in 3D) obtained by the solution of a Fredholm boundary integral equation of the second kind.

$$\frac{1}{2}\vec{\gamma}(\vec{x}) \times \hat{n}(\vec{x}) + \int_{\partial\Omega} \vec{\gamma}(\vec{x}') \times K(\vec{x} - \vec{x}')\, d\vec{x}' = \vec{U}_{slip}(\vec{x}) = \vec{U}_\infty(\vec{x}) - \int_\Omega \vec{\omega}(\vec{x}') \times K(\vec{x} - \vec{x}')\, d\vec{x}' \tag{10}$$

The solution of this equation is obtained in the collocation formulation by discretizing the surface into N_p contiguous triangular panels with piecewise constant vortex sheet (vector) strength. A similar equation exists for the unknown scalar surface source strengths, though it is of less use in a dynamic vortex particle method because it makes diffusion of vorticity from the surface more difficult. The surface integrals are evaluated analytically,[13] which provides more accurate solutions than quadrature techniques when the source and target panels are nearby, but has a greater computing cost per evaluation. Note that in Eqn. (10) the influence of the vortex particles is evaluated using the singular, and not the smooth, velocity kernel K to allow analytic integration.

II.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 elements are organized into a binary tree structure (a VAM-Split 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 (Biot-Savart summation) or long-range (spherical multipoles cast to Cartesian coordinates) methods.

For the vortex particle treecode, the trees, multipole coefficients, and interaction lists are built using multithreaded CPU routines. In a pure-CPU 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 present. The algorithm then runs two 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. 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

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 velocity and velocity gradients are moved off the GPU and stored in main memory.

Similarly, the dense influence matrix 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 supports solving, instead, 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 a $K(\vec{x})$ influence over a triangle and involve 334 FLOPs for the CPU version and 340 FLOPs for the GPU version. 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 in order to guarantee sufficient overlap and to keep the particle distribution uniform. This is in addition to the hole-filling capability of VRM. These three routines have not been ported to the GPU, but of them, only VRM takes more than a small amount of time. These overlap-maintaining routines collectively obviate the need for any regridding of the particle distribution while still keeping the particle density capped. To repeat, this method requires no periodic regridding or smoothing of the vorticity to maintain long-time accuracy.

The sequence of substeps in one time step using the 2nd-order forward advection algorithm is as follows:

1. Solve the BEM equations for the unknown surface element vortex strengths. This consists of the following steps:
   (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 ($0.5\,\delta_v$, typically) of any surface element;
   (d) Compute the RHS of the BEM equations via GPU-accelerated treecode summation of the analytic influence of all particles on all elements;
   (e) Iterate using GMRES until changes in the solution vector between steps is 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 element tree structure;
      ii. For each group of elements, traverse the tree and sum the far-field component of the influence matrix using multipole multiplication (GPU);
      iii. For each group of elements, 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 Eqn. (1) and a fast multipole treecode solver;
3. Advect each particle according to its local velocity, update each particle circulation according to its velocity gradient;
4. Repeat the preceding steps to complete the 2nd 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 local vorticity curvature;

6. Merge any particles that approach to within a small threshold distance;
7. Create new particles just above the surface elements with circulations drawn from the solution to the BEM;
8. Use VRM to diffuse vorticity among particles;
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 effort made to optimize the algorithms and the code itself, but with no explicit user-optimized machine language or SSE instructions. Gnu's *gfortran* and *gcc* compiled the code using the following performance options: `-O2 -ffast-math -funroll-loops`. CPU code has been multithreaded using OpenMP extensions, and the parallel efficiency of said multithreading is excellent (>95% for 8 cores) for all routines which will be compared to 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 two four-core Intel Xeon E5345 CPUs at 2.33GHz (both 64-bit chips running 64-bit Linux operating systems). It is unfair to compare to a single-threaded or single-core CPU, and these are considered to be representative hardware.

Data structures used by both CPU and GPU codes were designed to prevent cache misses while retaining enough flexibility to be used for multiple purposes. Specifically, the particle locations, radii, and strengths each have their own arrays, as did the panel node locations, node pointers, and panel strengths. Within each array, though, the elements are frequently re-ordered 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 was the panel node locations, which for the GPU routines were copied into one 9-by-N_p array to facilitate loading by the GPU kernels, while the CPU routines did 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, though, 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,[14] 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 that 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). 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.*

III.A. Vortex particle method

To determine the peak expected efficiency of the vortex particle method on GPU hardware, we tested the portion of the calculation with highest arithmetic intensity: the direct particle-particle Biot-Savart interactions. The velocity and velocity gradient were calculated for particles distributed randomly in a unit cube with overlapping core radii. Each particle-particle interaction computes the vortex velocity and 9-component velocity gradient influence on a target point, and does so using 66 floating-point operations (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** as fast as the multithreaded quad-core CPU version, which achieves a peak rate of 4.7 GFLOP/s, and 89 times as fast as the 8-core Xeon system (10.6 GFLOP/s). The theoretical peak performance of the GPU hardware is 2177 GFLOP/s (240 processing elements times 1.512 GHz shader clock times three floating point operations per cycle[14] times two GPUs), which makes the computational efficiency 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[4,15,16] when instructions are counted similarly. *Note that the above 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[1] demonstrated 218 GFLOP/s on a 8800 GTX GPU (128 PEs at 1.35 GHz), or 63% efficiency. While that may seem better than our current effort, note that the 8800 GTX can only perform one FMAD operation per PE per cycle (2 FLOPs), while 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 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, while the new version achieves 4.7 GFLOP/s on its 4-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 above, both GPU and CPU versions required the same number of FLOPs, thus comparisons are easy. But in order 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.[1] 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 9-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. While many authors report exemplary speedups when porting code to the GPU, most of those algorithms are trivially parallelizable (like the direct summation). These results show that an algorithm with better big-$\mathcal{O}$ performance can also benefit from GPU hardware, in this case finding a **109-fold speedup** vs. the direct method at 5M particles and with a break-even point for the better algorithm at 50k particles and 0.2s runtime.

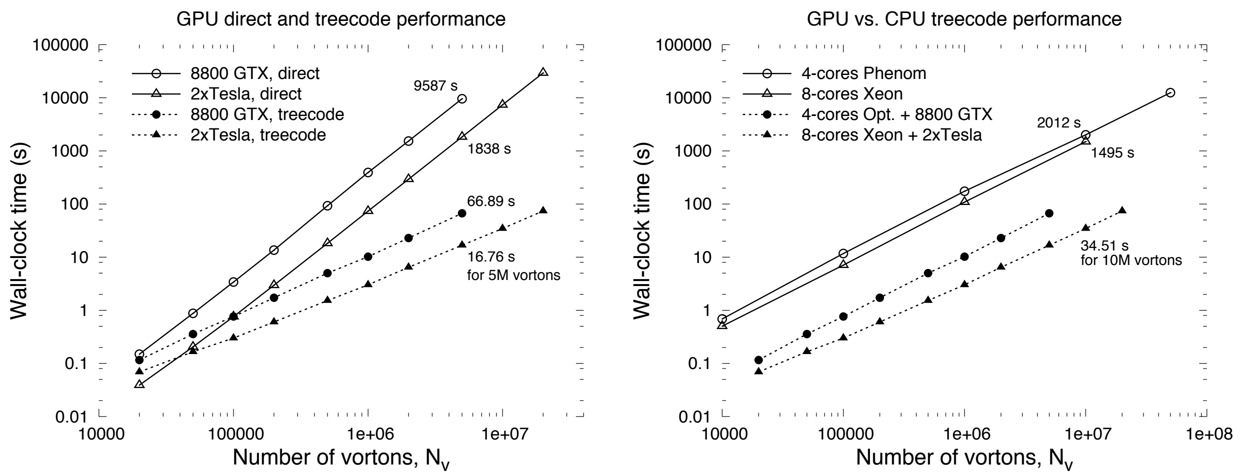

Figure 1. Left: Performance boost obtained moving from a naive direct summation algorithm to a multipole treecode for the N-vortex problem for various GPU hardware. "8800 GTX" is a system with two dual-core Opteron 2216HE CPUs and a single 8800 GTX GPU with 128 PEs, "2x Tesla" is the 8-core Xeon system described in the text. Right: Performance of multipole treecode on pure-CPU and hybrid CPU/GPU systems.

Also in Fig. 1 are comparisons between the fastest pure-CPU 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, 7th order multipoles, and a unique tree traversal for each leaf node of particles. Despite doing more work, the GPU version on the 4-core Phenom system completes the calculation **60.7 times faster** than the CPU version, and the GPU version on the 8-core Xeon system finishes **43.3 times faster** than the pure CPU version. Various algorithmic and implementation changes from the previous hybrid CPU/GPU particle treecode[1] 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 Opteron CPUs, and with the new treecode solves for the velocities and gradients on 500,000 particles in 5.01s instead of 14.9s. It is worth mentioning that both the CPU and GPU versions exhibit approximately $\mathcal{O}\left(N_v^{1.05}\right)$ scaling at large N_v.

Gumerov and Duraiswami[5] compare a single-threaded FMM on a 2.67GHz Intel Core 2 Duo to their GPU FMM on a 8800 GTX GPU and show 72-fold speedup for 1M particles with multipole order 8 and 29-fold speedup for multipole order 4 (the accuracy of which more closely corresponds to the present method). Their GPU FMM used a singular particle kernel and did not tackle problems large enough to confirm the $\mathcal{O}(N)$ theoretical scaling of FMM. Yokota *et al.*[6] ported an FMM algorithm to CUDA and found a 60-fold performance gain between a single 3 GHz core of an Intel Core 2 Duo and a single NVIDIA 8800 GT (112 PEs). Scaling was found to be $\mathcal{O}\left(N^{1.15}\right)$ for both versions.

While 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 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,[17] 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 gradient of each of its own particles due to the influence of its local particles and the LET.

Performance of the MPI-parallel treecode is measured with parallel efficiency, $e = t_{serial}/(N_{proc}\, t_{parallel})$, 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 NCSA, on 1 to 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 equation above represents a single process, but with 8 threads operating in parallel, and not a serial, single-threaded application. Thus, the CPU version on 64 nodes, which took 2.144s and 26.77s for 1M and

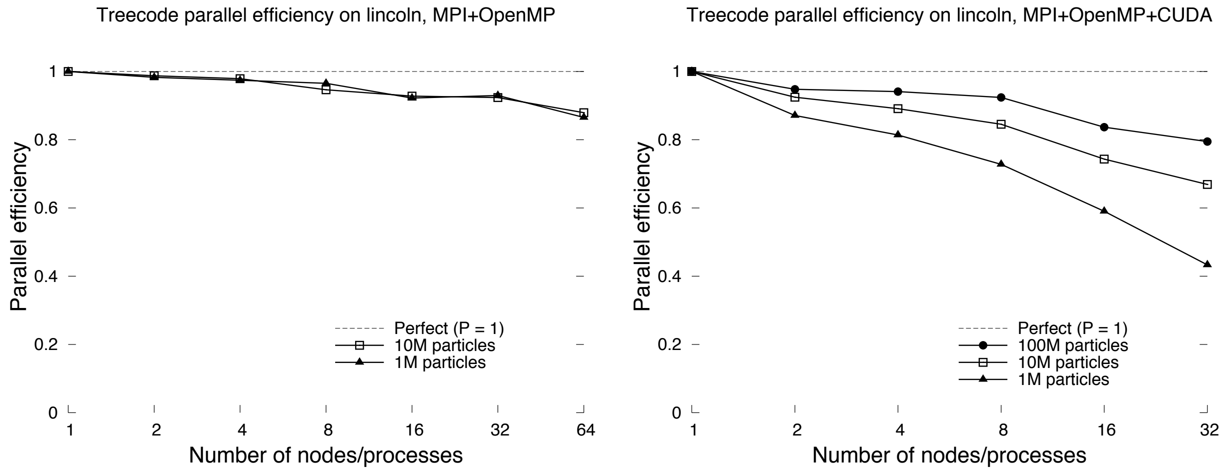

Figure 2. Parallel efficiency of all stages of hardware-optimized treecode without (left) and with (right) GPUs enabled. Each node of NCSA's *Lincoln* contains two quad-core Xeon CPUs and connections to two of the four GPUs in a Tesla S1070.

10M particles respectively, was running on 512 CPU cores, and achieved 88% efficiency. The GPU version running on 32 nodes took 0.263s and 1.928s 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 (100M particles), the load can be split more evenly across all devices, and the calculation completes in 18.36s 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 parallel efficiency of 68% on 32 processes for 1M 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 1M particles and 60% for 10M.

III.B. Aorta and heart valve

To test the performance of the hybrid CPU/GPU boundary element method, we used Eqn. (10) and the algorithm in §II.C to solve for the unknown source strengths on a triangulated mesh of an artificial heart valve and aorta.[18] The surface was discretized into a number of flat, triangular elements, and runs were

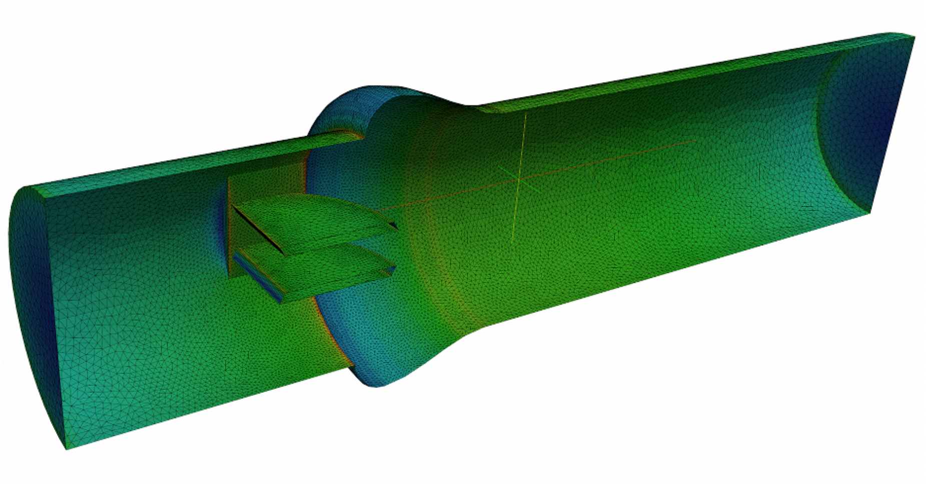

Figure 3. Triangulated geometry of aorta and heart valve, cut along center line to show interior, with 89,040 panels.

made for a variety of resolutions from 5000 to 1.425 million triangles. The flat discs 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 re-uses the coefficients of the influence matrix, only the smallest models (with 5000, 8000, 22k, and 32k panels) could be solved using the direct method due to the memory required. The performance of the method on these models appears in Fig. 4. The speedup peaks at 125 for the 8k-panel model, and drops to 83 for the largest case (which involved 11 GPU kernel calls per GPU and 4GB of memory transfer). The 4-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 that require large numbers of solver iterations, but limits the problem size, especially when using direct solution methods. Each entry in their influence matrix is the result of integration over 7 Gaussian quadrature points, and each quadrature point calculation requires 36 FLOPs (counting transcendental functions as 1 FLOP for consistency with the present results). Using an SSE-optimized, single-precision, 4-core CPU version, their method performs one GMRES iteration for their 32k-panel geometry in about 47s; and on one 8800 GTS GPU the same iteration takes about 5s. While 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 peaked at 11.5 vs. their optimized CPU code and 22.6 vs. a non-optimized version.

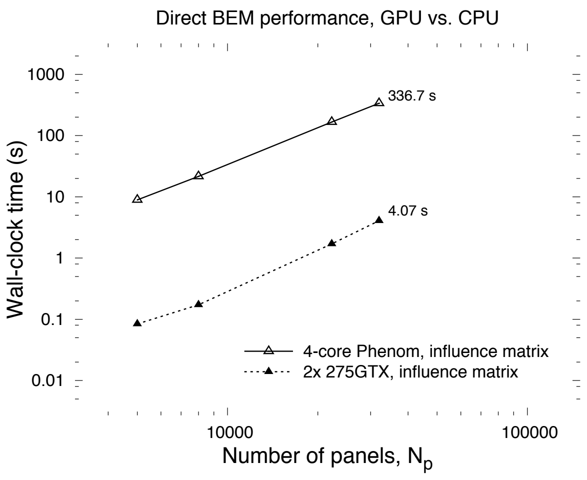

Figure 4. Time required by CPU and GPU methods to build and save the dense influence matrix from direct BEM of identical geometry with differing panel counts. CPU version used quad-core Phenom at 2.5 GHz, GPU version used two GTX 275 at 1.512 GHz.

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 above geometry were tested with these parameters, with the GMRES solver requiring 21 to 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** as fast as the four-core CPU version, comparable to the direct method, above. The GPU version exhibits $\mathcal{O}\left(N^{1.05}\right)$ scaling for the entire solution, and the CPU version $\mathcal{O}\left(N^{1.1}\right)$.

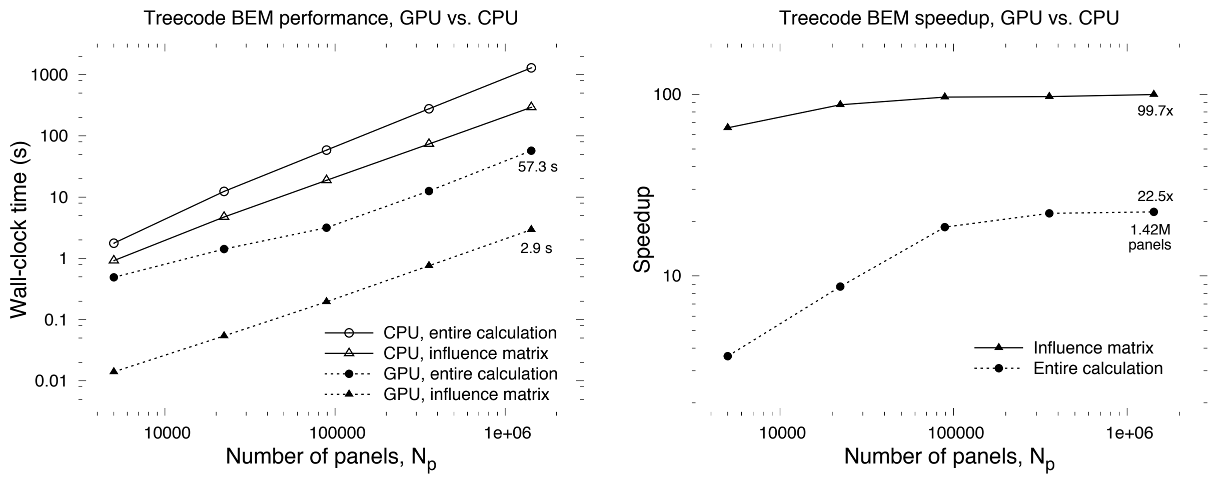

Figure 5. Left: Performance of CPU and hybrid CPU-GPU solvers on boundary element method (BEM) problem, broken down by calculation and active hardware. Right: speedup between CPU-only and hybrid methods. CPU version used quad-core Phenom, GPU version used two GTX 275.

For the case with 1.4M unknowns, the direct part of the influence matrix is computed in 2.938s, and the entire calculation was complete in 57.3s. Each iteration of the GMRES solver is composed of three steps: the direct portion 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, 0.735s, 0.463s, and 0.507s. 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 5.20s and finished the entire problem in 62.5s, still a 20.6-fold speed improvement over the 4-core CPU version. For comparison's sake, solving the 1.4M-unknown problem using the direct method would require 9 hours using the GPU for the influence matrix, and 8 days using the 4-core CPU. Those numbers increase to 2 and 185 days if the influence matrix needed to be recomputed every iteration.

III.C. Impulsively-started spheres

The GPU-accelerated BEM is integrated into ASR's Ω-*Flow*, and allows multithreaded and multi-GPU simulations of vortex-dominated flow. The complete method is tested by running DNS of impulsively-started spheres with Reynolds numbers 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, 20480, 81920, and 327,680 elements are used for the Re=500, 1000, 2000, and 4000 cases, respectively. Time steps are $\Delta t = 0.04$, 0.02, 0.01, and 0.005, which resulted in $||\omega||^2_{max}/\Delta t \approx 0.7$. Particle core sizes are $\sigma = 0.0379$, 0.0190, 9.49e-3, and 4.74e-3.

Figure 6 shows the center line vorticity for each of the four Reynolds number studies at $t = 3$. Illustrated

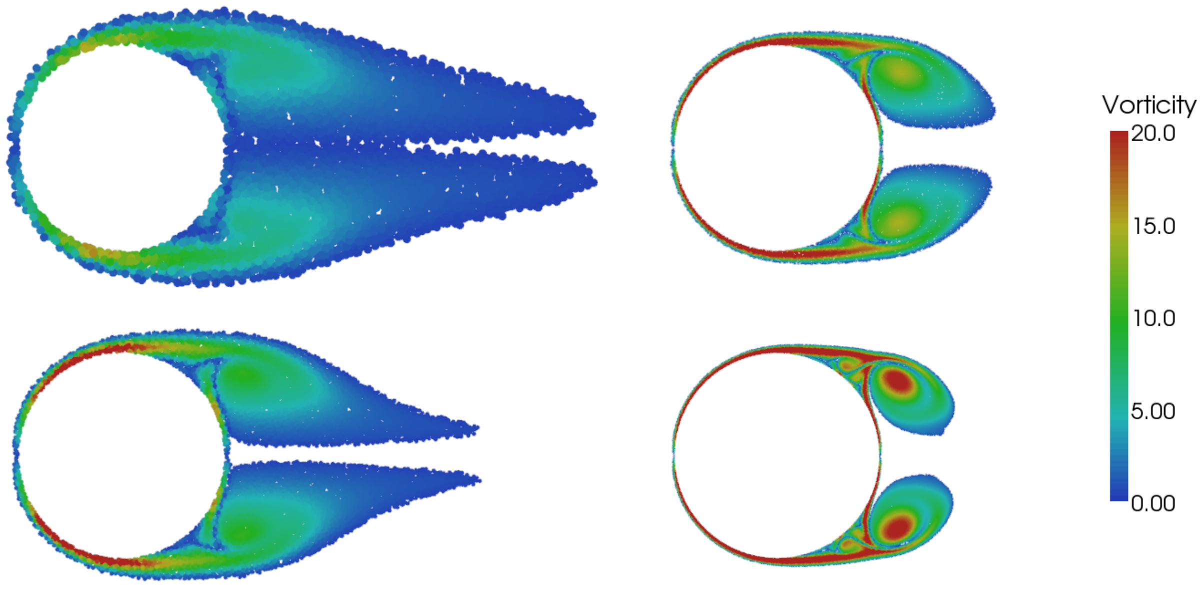

Figure 6. Center line vorticity around impulsively started spheres at t = 3; Re=500 (top left), 1000 (bottom left), 2000 (top right), 4000 (bottom right).

are all of the particles in a band $2\sigma/3$ wide. At this stage, each case shows formation of the primary wake vortex ring, though the rings are significantly stronger for the higher Re cases. In addition, the higher the Re, the more numerous the secondary recirculation bubbles in the near-separation region. The Re=4000 case contains 4 stagnation rings on its downwind hemisphere. The full three-dimensional simulations, at $t = 3$ contain 183k, 908k, 4.75M, and 26.46M particles, respectively. Despite aggressive particle removal, these simulations are still uniform resolution, and thus significant computation savings could be found using adaptive resolution methods.

Figure 7 depicts the onset of asymmetry for the cases with Re=500, 1000, and 2000. The 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, 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 naturally longer as Re decreases.

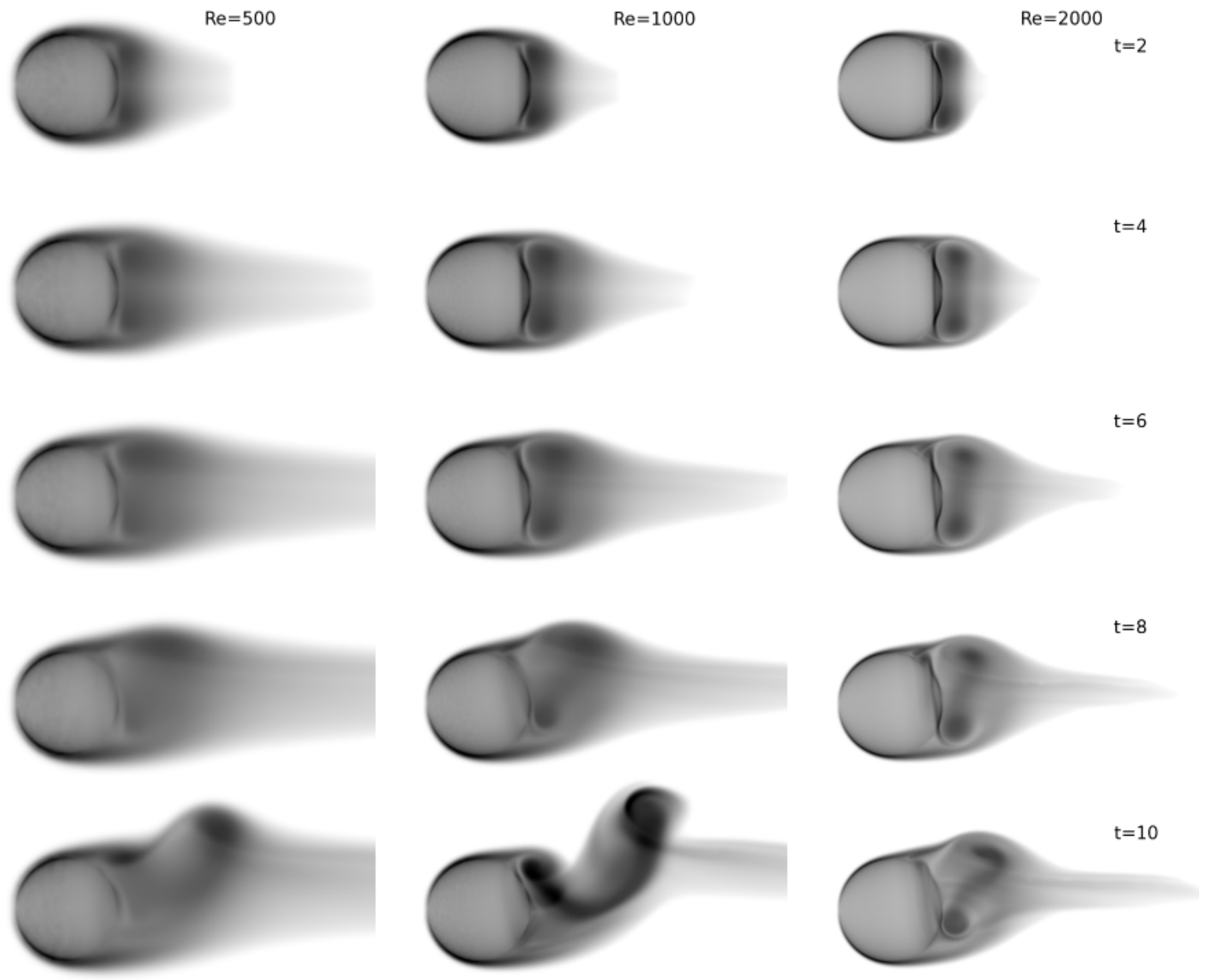

Figure 7. Vortex shedding from impulsively-started spheres at various times; images show "x-rays" of circulation.

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 1.4M-unknown BEM in just about one minute, much faster than the 8 hours required by a GPU-accelerated direct method (on 1.06M unknowns).[8] In addition, we have demonstrated a vortex particle method that solves for the velocities and velocity gradients of 100M particles in 18.4s on a cluster of 32 multi-core, multi-GPU 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

Development of the parallel LES method in this work was supported by Phase I SBIR Contract W911W6-10-C-0018 from the U.S. Army Research, Development, and Engineering Command (AMRDEC). The treecode parallelization effort and 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]Stock, M. J. and Gharakhani, A., "Toward efficient GPU-accelerated N-body simulations," *46th AIAA Aerospace Sciences Meeting*, 2008, AIAA-2008-608-552.

[2]Barnes, J. E. and Hut, P., "A hierarchical $\mathcal{O}$ (N log N) force calculation algorithm," *Nature*, Vol. 324, 1986, pp. 446–449.

[3]Greengard, L. and Rokhlin, V., "A fast algorithm for particle simulations," *J. Comput. Phys.*, Vol. 73, 1987, pp. 325–348.

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

[5]Gumerov, N. A. and Duraiswami, R., "Fast multipole methods on graphics processors," *J. Comput. Phys.*, 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 Lvy, 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," *Int. J. Numer. Methods Eng.*, 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," *17th AIAA Computational Fluid Dynamics Conference*, No. AIAA-2005-4624, Toronto, Ontario, Canada, June 6-9 2005.

[10]Shankar, S. and van Dommelen, L., "A new diffusion procedure for vortex methods," *J. Comput. Phys.*, Vol. 127, 1996, pp. 88–109.

[11]Gharakhani, A., "Grid-free simulation of 3-D vorticity diffusion by a high-order vorticity redistribution method," *15th AIAA Computational Fluid Dynamics Conference*, No. AIAA-2001-2640, Anaheim, CA, 2005.

[12]Degond, P. and Mas-Gallic, S., "The Weighted Particle Method for Convection-Diffusion Equations, Part 1: The Case of an Isotropic Viscosity," *Math. Comput.*, Vol. 53, No. 188, 1989, pp. 485–507.

[13]Gharakhani, A. and Ghoniem, A. F., "BEM solution of the 3D internal Neumann problem and a regularized formulation for the potential velocity gradients," *Intl. J. Num. Meth. Fluids*, Vol. 24, No. 1, 1997, pp. 81–100.

[14]NVIDIA, "NVIDIA CUDA Programming Guide," Tech. rep., NVIDIA Corporation, Santa Clara, CA, April 2009, Version 2.2.

[15]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.

[16]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.

[17]Salmon, J. K., *Parallel Hierarchical N-Body Methods*, Ph.D. thesis, California Institute of Technology, 1991.

[18]Gharakhani, A., "Grid-free LES of bileaflet mechanical heart valve motion," *Proceedings of BIO2006*, Amelia Island, FL, June 21-25 2006.