Toward efficient GPU-accelerated *N*-body simulations

Mark J. Stock[*] and Adrin Gharakhani[†]

Applied Scientific Research, Santa Ana, California

***N*-body algorithms are applicable to a number of common problems in computational physics including gravitation, electrostatics, and fluid dynamics. Fast algorithms (those with better than $\mathcal{O}(N^2)$ performance) exist, but have not been successfully implemented on GPU hardware for practical problems. In the present work, we introduce not only best-in-class performance for a multipole-accelerated treecode method, but a series of improvements that support implementation of this solver on highly-data-parallel graphics processing units (GPUs). The greatly reduced computation times suggest that this problem is ideally suited for the current and next generations of single and cluster CPU-GPU architectures. We believe that this is an ideal method for practical computation of large-scale turbulent flows on future supercomputing hardware using parallel vortex particle methods.**

I. Introduction

The N-body problem describes the interaction of a system of N particles each of which affects all others according to a function of their separation distance. N-body algorithms have improved considerably since their inception, and are naturally applicable to problems in gravitation, electrostatics, and vortex fluid dynamics.

The straightforward solution to this problem is to sum, for each particle, the individually-computed influences arising from a loop over all of the other particles in the system. This is called the *direct method* and requires $\mathcal{O}(N^2)$ computational effort, which is clearly not fast enough for problems of engineering interest that typically require millions of data points. Three ways to speed this calculation are to: apply a method that requires less computational effort, parallelize the problem across a cluster of computers, or use faster specialized hardware.

Early research leveraged hierarchical clustering of the data to create "fast" methods for numerically solving the N-body problem. These include a method by Appel,[1] the treecode method,[2] and the Fast Multipole Method (FMM).[3] Each theoretically allows computation of the influence on every particle in $\mathcal{O}(N)$ to $\mathcal{O}(N \log N)$ time. When applied to desingularized vortex particle systems in three dimensions, though, these methods rarely require less than $\mathcal{O}(N^{1.1\rightarrow1.2})$ effort.[4–6]

The other algorithmic improvement to N-body performance comes from efficiently casting the problem onto non-shared-memory parallel computers. Salmon[7] used a binary tree and hierarchical information-sharing patterns to retain the $\mathcal{O}(N \log N)$ performance of the treecode for gravitational calculations on parallel computers. The ideas contained therein appear throughout the literature of parallel N-body methods.

The other route to improved performance of N-body simulations is to use specialized hardware, and in this category are custom hardware such as GRAvity PipelinE (GRAPE) and increasingly less specialized graphics processing units (GPUs). Computers with custom-designed and manufactured GRAPE[8] and MD-GRAPE[9] boards have repeatedly won Gordon Bell prizes for performance and price-per-performance, though the machines are not general-purpose computers. The boards are multi-pipelined parallel computers, and have essentially "compiled" the particle-particle interaction into hardware, allowing for very fast calculations. The cost of this extra speed is programmability, though the MD-GRAPE allows calculation of interactions

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

with arbitrary force-law relations such as vortex methods.[10] GRAPE hardware has been used with both parallel direct,[11] serial treecode,[12,13] and parallel treecode methods.[14,15]

In the spectrum of speed vs. programmability, graphics processing units (GPUs) lay between general-purpose computers and specialized GRAPE hardware. The advantage of GPUs is that they are COTS (consumer off-the-shelf) hardware, and thus the technology is updated very frequently and price-per-performance drops rapidly. As of late 2007, the price per GFlop/s for GPUs is under 1 USD, while CPUs are 10-15 USD. The rapid decrease in price-performance ratio combined with the increasingly programmable architectures make general-purpose programming with GPUs (GPGPU) an attractive approach for N-body simulations. Direct methods for gravitational[16–20] and molecular dynamics[21] applications have already been demonstrated on GPU hardware. Xue[22] ported the direct portion of a treecode method to a GPU, and Siddiqui[23] attempted to port the entire algorithm, from tree-building to velocity-finding, but both applications attacked problems that were too small for practical use and both encountered problems with efficiency or software stability. Belleman *et al.*[20] also present a treecode method for GPUs, but far-field interactions are accounted for using only monopole moments.

The present work makes the following contributions: a best-in-class treecode method for the N-vortex problem, a direct solver that achieves over 200 GFlop/s on a single GPU, and the first successful demonstration of a GPU-accelerated *multipole* treecode solver for large, dynamic N-body problems.

The remainder of this paper will be organized into four main sections. We will begin by reviewing the basic treecode method and the various calculations required to set up and solve the N-body problem for vortex particles. Following that, we will discuss the present state of GPU architecture and demonstrate an $\mathcal{O}(N^2)$ solver for vortex particle methods. Finally, we will cover porting the treecode method to GPU hardware and show results from this effort.

II. Vortex particle methods

The problem that we will address in this work is the kinematic velocity-vorticity relationship found in incompressible fluid dynamics, though the resulting methods are equally applicable to gravitational and electrodynamic computations. The vortex particle velocities $\bar{u}_\sigma$ and their gradients are smooth and are evaluated by convolving the Biot-Savart integral for velocities with a smoothing or core function g:

$$\bar{u}_i(\bar{x}_i) = \sum_{j=1}^{N_v} K_\sigma(\bar{x}_j - \bar{x}_i) \times \bar{\Gamma}_j \tag{1}$$

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

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

where $\bar{\Gamma}_j$ is the vectorial circulation (strength) of the particles and $g(r) = (3/4\pi)\exp(-r^3)$ is the core function with radius σ. The smooth velocity gradient is obtained by differentiating equation (1) directly. This calculation can be performed numerically with a doubly-nested loop—a method called *direct summation*—and results in a method that is $\mathcal{O}(N^2)$ in time.

II.A. Fast methods

The two most common algorithmic improvements to the direct method are the treecode[2] and Fast Multipole Method (FMM).[3] They are similar in that they both use hierarchical subdivision of the problem space to reduce computation, and they rely on multipole expansions to approximate the effect of far-away clusters. The present method uses a treecode because the algorithm is easier to parallelize and port to special hardware, as we shall see in §III-IV. As such, it is theoretically $\mathcal{O}(N\log N)$, though performance results below indicate $\mathcal{O}(N^{1.17})$ best-case behavior. Like FMM, the multipole treecode algorithm conducts the following four steps for every full velocity evaluation.

II.A.1. Create tree structure

For the following results, the tree structure used is a *VAMSplit k-d tree*[24] with post-construction box shrinking. This tree construction technique is designed to fill as many leaf boxes as possible to capacity, which becomes important when writing GPGPU code. The original intent of the method was to avoid wasting space on devices that needed to page memory to slower storage media, but is adapted at present to prevent inefficiency on data-parallel graphics processing units. All but one leaf node in this tree will have exactly N_b (called the "bucket size") particles.

II.A.2. Calculate multipole expansion coefficients

The details of the representation of clusters of vortex particles by multipole expansions are documented elsewhere,[3] and only the variations from the general method will be discussed here. The present method uses multipole expansion coefficients that are based on the real, and not the complex, spherical harmonic functions.[5] Doing this increases somewhat the CPU time required to calculate the coefficients, but reduces by roughly a factor of two the CPU time required to perform the multipole multiplications in the velocity evaluation loops. Both computations require $\mathcal{O}\left(p^2\right)$ calculations per particle per box and $\mathcal{O}\left(p^2\right)$ storage per box, where p is the order of the multipole expansion. We find that this method achieves $\mathcal{O}\left(10^{-4}\right)$ error most rapidly with $p \geq 9$.

II.A.3. Make interaction lists

Before or during the velocity calculation for a given point, the tree is traversed from the top down and tree nodes, representing boxes of particles, are categorized as "far" or "near." Far boxes are those for which a multipole-based approximation would generate acceptable error. Near boxes are first opened to see if any of their children qualify as a far box, otherwise they represent boxes that are too near for the approximation to be accurate. The box-opening criteria are similar to the original treecode method[2] but further modified by Warren and Salmon.[25] This procedure recurses downward through the tree until the influences of all particles are accounted for by either a multipole multiplication (box-wise) or direct (particle-wise) interaction. The present CPU treecode creates a unique interaction list for each individual particle, but supports creating lists for groups of particles. Highly data-parallel applications require groups of particles to share a common interaction list.[26]

II.A.4. Velocity calculation

The velocity and velocity gradients are calculated anew for each particle in the system (unlike FMM, which uses local multipole expansions). This procedure consists of finding the interaction list for either the individual particle or its immediate parent's box, and then looping through the boxes and particles within that list while summing their influences. An algorithm may compute the influences of near and far boxes together or in separate procedures.

II.B. Performance

Serial performance can be measured in several ways, first of which is the absolute time taken by the program to construct the tree and evaluate the velocities for every particle. The second is a more processor-independent method, which consists of comparing the raw CPU time to what would be required by the direct method; this result is commonly called the *speedup.*

The most common test case is that of finding the velocities without the velocity gradients for a group of particles distributed throughout a cubic volume. Nevertheless, no two references found contained exactly the same test. Figure 1 shows the raw performance of the method as well as the speedup vs. two other references. The method by Wang[5] was previously one of the fastest treecode methods and solved for the velocity of Gaussian-cored vortex particles placed randomly in a cube, though no core size was mentioned. The simulations by Strickland *et al.*[27] also used a Gaussian core function, but over a uniform distribution of particles. That reference specified a smoothly-varying vorticity throughout the cube, which we have observed returns smaller errors for the same simulation parameters. Cheng *et al.*[28] showed performance similar to the present method, but computed potentials (which are one derivative lower than the present method's velocities, but decay less rapidly with distance) for singular point sources (which require fewer calculations than 3-tuple vortex strengths) in single precision (as opposed to the current method's double precision), and are thus not directly comparable. The fast algorithm presented herein allows a speedup of nearly **1000 times** over the direct method for practical problems ($N_v \simeq 10^6$) on a serial computer.

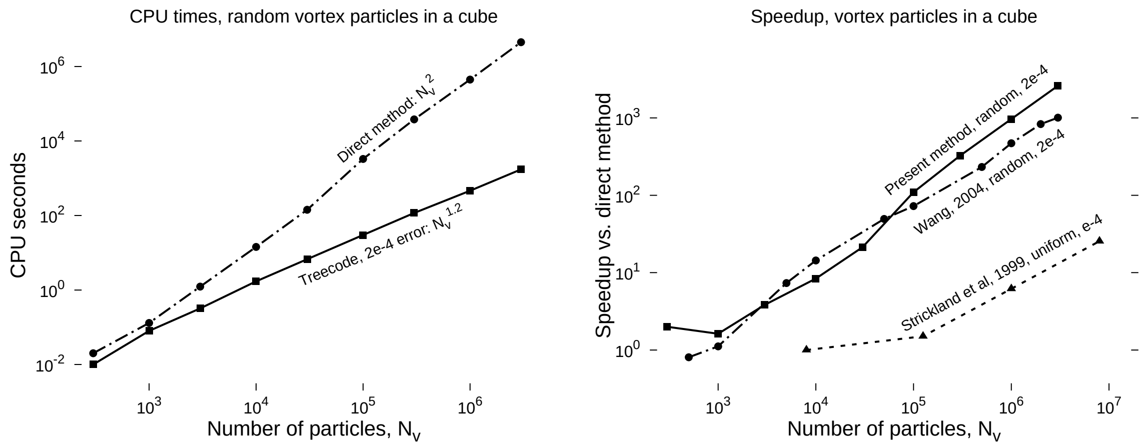

Figure 1. Raw CPU speed (left) and speedup (right) for particles in a cube (with mean error); data from Strickland *et al.*[27] are modified to account for parallelization inefficiencies; data for present method were created on an AMD Opteron 246 processor with version 5.2 of the *pgf90* compiler.

Other box-opening (or multipole acceptance) criteria were tested. It was found that for the same multipole order ($5 < p < 9$) and mean velocity error ($10^{-3} < e < 10^{-4}$), the best times achieved by the variable-multipole-order method[25] were within 5% of the best times achieved by the older treecode method.[2] For lower-accuracy calculations, the Barnes-Hut method is uniformly faster. The variable-expansion-order method[4] (Eqns. 3-5) proved least effective of the three methods for our implementation.

III. GPU-accelerated direct solver

Until recently, leveraging the performance of data-parallel graphics processing units required programming in graphics-centric languages and working with low-accuracy number representations. With the advent of single-precision IEEE-compliant math and higher-level programming layers like BrookGPU[29] and more recently NVIDIA's Compute Unified Device Architecture (CUDA), the field of general-purpose computing on graphics processing units (GPGPU) has blossomed. Now GPGPU offers order-of-magnitude better price-performance ratios than CPUs (under 1 USD per GFlop/s for the NVIDIA 8800 GTS compared with 10-15 USD for entry-level quad-core CPUs) with increasingly less restrictive programmability.

III.A. Recent GPU architecture changes

Previously, all GPUs on the market had *superscalar* architectures: compute cores that operate concurrently on packed structures of four 32-bit floating-point numbers. This would benefit traditional graphics applications which mainly operate on 4-component quaternions and RGBA colors, but any calculation that requires only one component either wastes the other three calculations or must be carefully hand-tuned to use them. In addition, these GPUs have separate processors for vertex processing, fragment processing, and texture processing. There are usually more fragment processors than others, so GPGPU programs used only those while the others remained idle.

In November 2006, NVIDIA released a new series of GPUs that operate with larger numbers of *unified scalar* processors. In this new 8000-series (and in projected future GPUs) each scalar processor core operates not on a packed sequence of four floats at once, but only one float at a time. In addition, there are no separate vertex, fragment, or texture processors: each unified processor can do any job and can switch jobs depending on the load. This benefits the gaming market because those applications increasingly require complex scalar arithmetic for procedural effects and in-game physics. It also benefits GPGPU, because many sections of code—even vector mathematics code—require scalar calculations; thus, the computational resources are used more efficiently.

The NVIDIA 8800 GTX hardware was chosen for the following tests because of its speed, scalar capability, and programmability via CUDA. The 8800 GTX has 128 unified scalar processing elements (PEs) organized into 16 multiprocessors, each with 8 PEs and 16 kB of on-chip shared memory. This shared memory can be accessed as fast as a register by any of the threads running on that chip as long as there are no bank conflicts. Though fetches to main GPU memory take 200-300 clock cycles, the latency is hidden by switching between thread groups.[30]

GPGPU is most often applied to very data-parallel problems: problems with a lot of elements and for which the computation of each element can be performed independently. The direct solution method of §II is an ideal problem for GPGPU, despite its $\mathcal{O}(N^2)$ operation count. Before moving to the more complex treecode method, we first created and tested a direct solver on a variety of GPU hardware.

III.B. GPU-direct method

GPGPU kernels were written for both the BrookGPU and CUDA compilers. For the Brook implementation, the particles are divided into two-dimensional blocks of stream elements. Each stream element reads the running sum for its respective target particle, adds the influence of a block of vortex particles, and overwrites the running sum upon output. Once all of the blocks have been called, a single read returns the data from the GPU. In the CUDA kernel, each thread is responsible for accumulating the influence of all other particles. Each group of threads iterates through the same group of 64 vortex particles at a time, reading those source particles into shared memory and operating on them locally (as suggested in the documentation[30] and previously implemented[17]). Because both the Brook and CUDA implementations support arbitrary numbers of particles, the final block must be filled with data that does not affect the outcome; we use zero strengths and unit radii.

Some difficulties were encountered while constructing these kernels. Extra logic had to be used in BrookGPU because kernel programs are limited to 65536 instructions. In addition, because BrookGPU on Linux uses the OpenGL drivers, each kernel invocation is limited to one set of `float4` data on output. As such, we had to run four separate kernels to compute the full velocity and velocity gradient at a target point (though it could feasibly be packed into three `float4`s). CUDA had neither of these limitations.

III.C. GPU-direct performance

The case used to test these kernels is that of computing the velocity and velocity gradient for particles distributed randomly in a unit cube and with core radius $r_p = N_v^{1/3}$. The strength of each particle is random, bounded in magnitude by $||\bar{\Gamma}_p|| \leq \frac{3}{2}\, r_p^2$. For comparison purposes, a multithreaded, single-precision CPU direct solver was created using Fortran 77 and compiled with gfortran with high optimization, but with no hand-coded machine language or explicit SSE instructions.

The particle-particle interaction rates for the CUDA and BrookGPU versions are compared with the pure CPU method in Fig. 2. The run times reported here and in the rest of the paper include both calculation

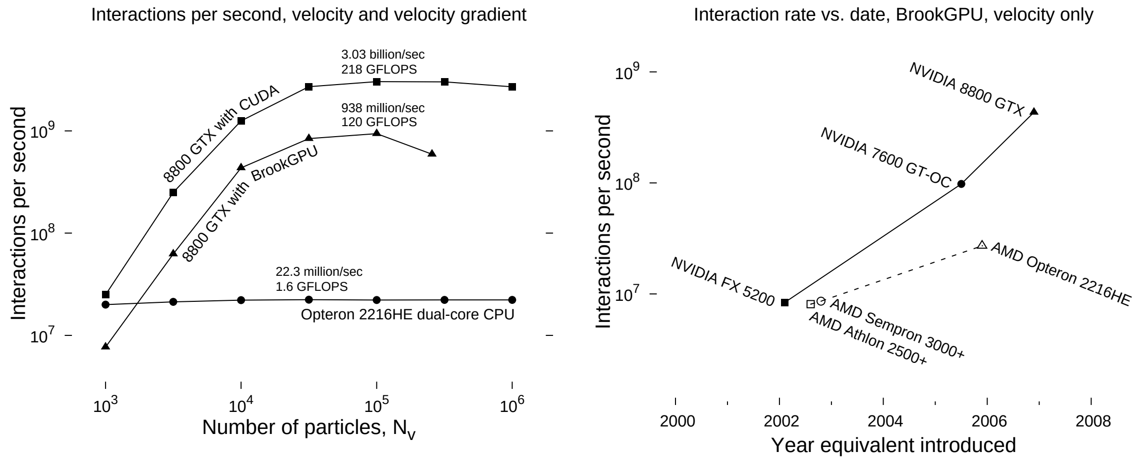

Figure 2. Left: number of velocity and velocity gradient particle-particle interactions per second for two GPU kernels (CUDA and Brook) vs. dual-core CPU optimized Fortran implementation. Right: performance of BrookGPU velocity-only kernel on various GPU hardware and of present Fortran implementation on various CPUs. Date is year equivalent hardware introduced.

time and data transfer time to and from the GPU. The interaction rate for the dual-core CPU version is a constant 22 million interactions per second and does not depend on the problem size. The two GPU implementations exhibit CPU-level performance for small problems and rapidly speed up as the problem size grows, only to retreat slightly for very large problems. The source of this performance degradation has not been isolated. The CUDA version produces results about three times faster than the Brook version, the difference being due to the limited number of outputs from the Brook kernel on Linux. This caused the number of floating-point operations to differ, too: 72 for the CPU and CUDA versions, 128 for Brook. At its peak, **the CUDA version conducted more than 140 times the number of interactions as the dual-core CPU version: 3.03 billion**. The MD-GRAPE implementation of vortex methods[10] also requires several calls to the special hardware and as such allowed only ∼60 million interactions per second.

The peak floating-point calculation rate, **218 GFlop/s**, is nearly two-thirds of the NVIDIA 8800 GTX's theoretical usable peak of 346 GFlop/s, and is nearly 140 times faster than the CPU's 1.6 GFlop/s performance on the same problem. This theoretical rate assumes uninterrupted multiply-add (MAD) instructions for all PEs. The observed deficit is most likely due to the fact that only half of the instructions in the tested kernel were MADs. Nevertheless, the performance of the proposed method attests to the high *arithmetic intensity* of the algorithm. Arithmetic intensity is proportional to the number of arithmetic operations per memory operation, and the present method requires only 7 `floats` (particle location, radius, strength) for every 72 instructions. Examples of applications with lower arithmetic intensity are the single-precision matrix-matrix multiply routine *SGEMM* from NVIDIA's CUBLAS library, which returned 91 GFlop/s on the 8800 GTX, and the *SGEMV* matrix-vector multiply routine, which achieved only 5 GFlop/s.

Note that previously reported performance for the calculation of gravitational forces on the same hardware (256 GFlop/s[17] and 340 GFlop/s[20]) assume 38 floating-point operations per interaction. This number artificially accounts for the extra work traditionally required by the reciprocal and `sqrt` functions, and does not correlate to actual floating point operations, of which there are 20. Counting 20 floating-point operations per interaction, the peak performance of the aforementioned methods are 135 and 179 GFlop/s, respectively. Nyland *et al.*[18] use the correct count and report 204 GFlop/s for a direct, gravitational, N-body method.

During the course of developing the BrookGPU kernel, similar tests were made on a variety of older hardware. The results, also in Fig. 2, show the direct summation performance (in interactions per second for the velocity-only BrookGPU kernel) vs. the first release date of equivalently-performing GPU hardware. The performance of the CPUs tested doubled roughly every two years, but the GPUs doubled their performance every 10 months. If this trend of enormous growth relative to CPUs continues, it will result in major performance boosts for any applications that can be arranged to run efficiently on GPU hardware.

IV. GPU-accelerated treecode

We have seen in the previous sections two methods which improve the performance of N-body simulations: fast treecode solvers that offer *several orders of magnitude improvement* over direct methods and GPU-accelerated direct summation methods that show *two orders of magnitude speed improvement* for the direct-only summations. Amdahl's law implies that we cannot just multiply these speed gains together, though the goal of this work is to see how effectively the methods cooperate. The implementation of the full multipole-accelerated treecode method is described below.

IV.A. GPU-treecode method

Recall that the accelerated treecode method of §II consists of several phases: building the tree, computing the multipole moments, finding the interaction lists, and traversing the lists while accumulating the velocity influences. That final step—finding the velocity influences—is composed of two kinds of actions: particle-particle interactions and box-particle interactions (where a whole source box affects a target point using multipole multiplication). These are, respectively, the near and far boxes in the interaction lists described in §II.A.3. The optimal CPU method computes these interaction lists for each particle just before using them to concurrently compute the near- and far-field interactions. In contrast, the GPU method will compute and save the interaction lists for entire boxes of target particles, then compute the near-field Biot-Savart interactions separately from the far-field multipole multiplications. The runtime breakdown for one time step of both of these methods run on a dual-core CPU is displayed graphically in Fig. 3. The present method's multipole-acceptance criteria result in longer interaction lists and more near-field interactions when the target volume is greater; this causes the increase in total computation time for the latter method. Figure 3 confirms that the majority of the computation time is spent in the velocity calculation routines, and supports the decision to port only those portions to the GPU.

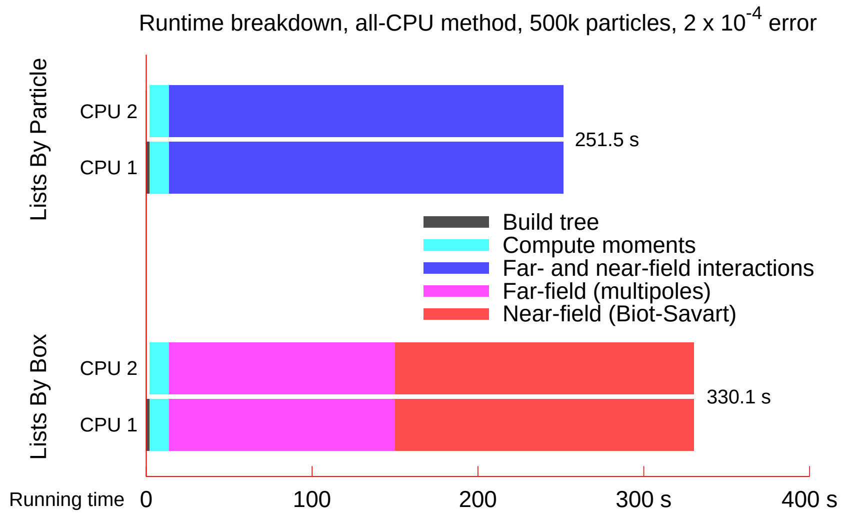

Figure 3. Runtime breakdown of all-CPU method showing time spent in each portion of the calculation; top: optimal CPU computation, which determines interaction lists for each particle individually; bottom: compartmentalized calculation, which determines interactions lists for whole boxes of particles before any velocity evaluation takes place.

In the present method, the tree, particle box multipole moments, and box-wise interaction lists are created on the CPU, and the far-field box-particle and near-field particle-particle interactions are sent to the GPU in separate kernels. Note that, of the previous efforts, one ported only the particle-particle interactions to the GPU;[22] the GRAPE methods[12,13] recast the box-particle interactions as particle-particle interactions; another used a monopole method for far-field interactions;[20] and only one attempted to port every stage of the fast multipole method.[23]

To make an algorithm more data-parallel—and thus more amenable to the current range of GPU hardware—the same operations must be performed on large numbers of elements, and the architecture of the underlying hardware may dictate the optimum number of elements in these blocks. The VAMSplit k-d tree construction method is selected to most efficiently map the velocity-finding computation to data-parallel GPU hardware. In this method all boxes but one have exactly the same number of particles, thus the bucket size N_b can be set to match the underlying hardware. In the case of the 8800 GTX, the documentation[30] suggests creating threads in multiples of 32. N_b is set to 64 to force two sets of 32 threads to be active on each PE (swapping between these sets helps hide costly main GPU memory latency).

Before the particle-particle influences are calculated, all particles are placed in GPU main memory in the order that their parent boxes appear in the tree. Source and target particles are stored in separate arrays, even though in normal use, each source particle is also a target. The method creates one thread per target particle, 64 threads per thread block (because there are 64 particles in each tree box), and one thread block for each leaf box in the tree. The box index for each particle/thread is calculated from the particle's index using integer division, and is then used to reference the appropriate precomputed interaction list for that box. This interaction list consists of integer indexes of source boxes whose influence on the target box's particles must be calculated. Each source box's particles are read into the PE's fast shared memory, 64 particles at a time (as in the GPU-direct method), and all threads in the thread block iterate through that chunk of data simultaneously. Each thread contains a running sum of its particle's velocity and nine-component velocity gradient tensor, which it writes to main GPU memory when the interaction list is exhausted.

The hardware programmability provided by CUDA allows computation of the far-field influences using the same multipole multiplication algorithm as in the CPU version. The threads and thread blocks are set up the same as for the near-field (direct) computation, and each thread block iterates through its interaction list of source boxes. Each source box's entire set of multipole moments is read into the PE's fast shared memory together, and all threads operate on that chunk of data simultaneously. As mentioned in §II.A.2, only real multipoles are used, and it was found that a multipole order of $p = 7$ returns better system performance than $p = 5$ or 9. Note that the use of spherical harmonics for the multipoles means that interactions very close to the pole axis are error-prone. The solution employed in the present method to deal with this involves excluding certain particles from the multipole calculation and adding them separately as

Biot-Savart summations. These are still done on the CPU (in parallel), and the time spent accounting for these "bad" particles is included in the subsequent runtime breakdown graphics.

In comparison, the GRAPE boards, being more rigidly preprogrammed, require the multipole description to be approximated with a reduced set of particles. Early work[12, 14] replaced each box with a single particle (monopole), but later work[13, 15] leveraged the pseudo-particle method[31] to replace the multipole description of each box of particles with a number of equivalent particles. This method allows *all* influences to be calculated by a single fast routine, but has been shown to be slower than FMM on CPUs for a given level of accuracy.[32]

IV.B. GPU-treecode performance

The new GPU-treecode was tested on the same problem as the GPU-direct method (§III.C)—finding the velocities and velocity gradients of a system of particles distributed randomly in a cube.

Whereas the direct GPU method was able to perform 140 times the number of particle-particle interactions as the CPU method (§III.C), tests indicate that the real multipole multiplications complete only 11-13 times faster than the CPU version (using both cores of an Opteron 2216HE). The runtime breakdown of the same calculation performed with an all-CPU method and the new hybrid CPU-GPU method appears in Fig. 4. For these tests, 7 orders of multipole moments were used, and the particles are in 2-way VAMSplit k-d trees with $N_b = 64$. This figure shows that moving the velocity calculation routines to the GPU results in

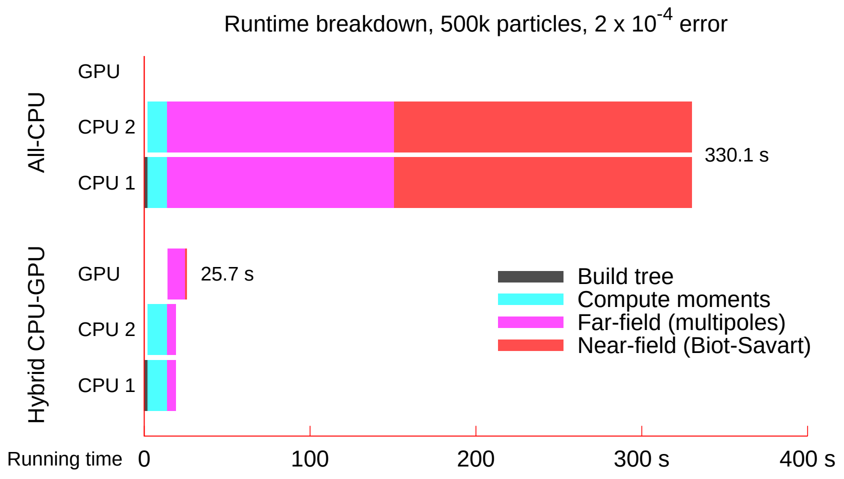

Figure 4. Runtime breakdown of all-CPU and hybrid CPU-GPU methods showing time spent in each portion of the calculation; same simulation parameters used for both methods.

a nearly **13-fold speed improvement** over the identical dual-core CPU method and 10-fold improvement over the ideal dual-core CPU method (not shown).

Note that in Fig. 4, the GPU computations are dominated by the far-field interactions. Since the GPU-CPU speedup for the far-field multipole multiplications is not as profound as the near-field Biot-Savart speedup, it would be globally beneficial to shift work from the far-field to the near-field interactions. The parameter that gives this sort of control is the bucket size (N_b). The effect of changing the bucket size is illustrated in Fig. 5. In this figure it is easy to see that the work spent computing direct summations increases significantly as N_b rises. The computational effort involved in the far-field interactions decreases as N_b increases, and the optimum bucket size is typically found where the slopes are opposite and of equal magnitude. An additional effect of increasing the bucket size is that the trees require fewer levels and boxes, which translates to less time spent in the costly CPU sections of the code: building the tree, computing the multipole moments, and determining the interaction lists. Future treecodes will surely perform those calculations on the GPU hardware as well.

As with other treecodes, the box-opening criterion and the number of multipole orders have a strong influence on speed and accuracy. Optimal performance for a mean velocity error of 2×10^{-4} over 500,000 particles was found with multipole order $p = 7$, a bucket size of 64 for the target points and 512 for the source particles, and 8-way VAMSplit trees. The box-opening criteria were two-fold: the ratio of the largest diagonal of the source box to the distance from the target box center to the nearest point on the source

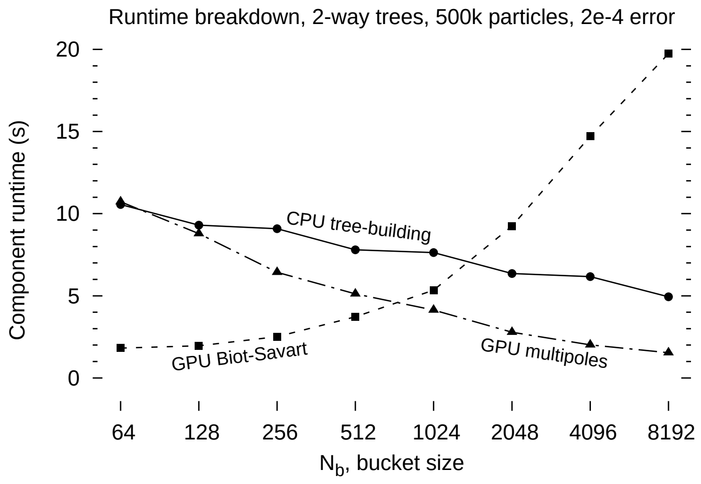

Figure 5. Runtime breakdown of hybrid CPU-GPU method with different bucket sizes, showing shift of computational effort between near-field Biot-Savart and far-field multipole multiplications; all trees are 2-way VAMSplit k-d trees.

box was 1/0.79; the ratio of the largest particle radius to the distance between the closest corners of the source and target boxes was 1.5. With these parameters, the fastest hybrid CPU-GPU run took 14.87 s, or **nearly 17 times faster than the best dual-core all-CPU run**. The runtime breakdown appears in Fig. 6. These parameters produced mean velocity gradient errors of less than 1×10^{-4} for most runs. A

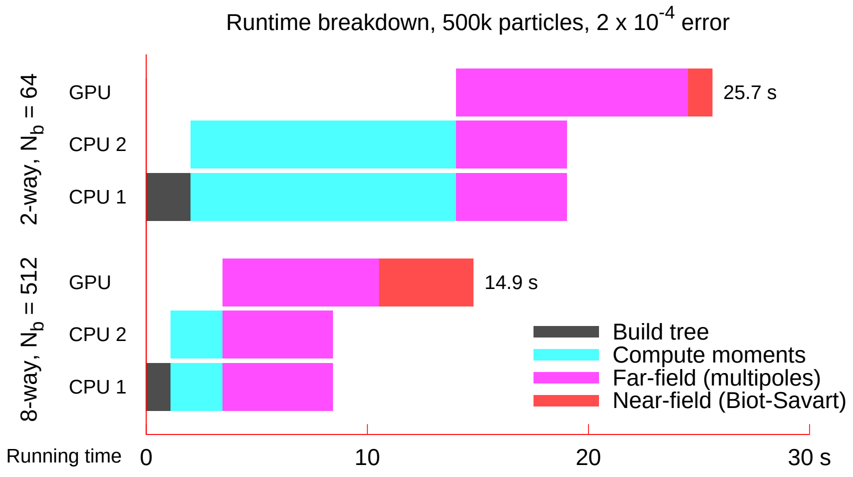

Figure 6. Runtime breakdown of hybrid CPU-GPU method with different tree parameters; top: $N_b = 64$, 2-way trees (the parameters that returned best all-CPU performance), bottom: $N_b = 512$, 8-way trees (best overall GPU performance).

table summarizing the performance breakdown of the various methods appears in Fig. 7.

There are a few improvements that can be made to the present method to increase its performance on the same hardware. First, because the multipole moments are not needed for the near-field summations, they could be performed on the CPU while the GPU begins the near-field summations. Once both are complete, the GPU would use the multipole moments to compute the far-field interactions. Secondly, the GPU kernels do not take into account differences in the length of the interaction lists of each target box. This means that some boxes require more calculations than others, which results in wasted GPU resources near the end of the computations. Finally, certain optimizations such as manual loop unrolling[18] and reducing register requirements have not been attempted. These optimizations are generally beneficial to GPU performance, but can make the code more complex, and are not guaranteed to have the same impact on future hardware.

Method	t_{total}	t_{tree}	$t_{multipole}$	$t_{velocity}$
CPU direct	11242 s	0 s	0 s	11242 s
CPU treecode	251.5 s	2.0 s	11.7 s	237.8 s
GPU direct	88.7 s	0 s	0 s	88.7 s
Hybrid CPU-GPU treecode	14.9 s	1.1 s	2.3 s	11.4 s

Figure 7. Peak computational speeds for calculating the velocity and 9-component velocity gradient for 500,000 exponential-cored vortex particles positioned randomly in a cube, treecode mean velocity errors are 2×10^{-4} compared to direct methods.

IV.C. Dynamic simulation

One step of the full fluid dynamic vortex method developed previously[6,33] requires two velocity evaluations, viscous diffusion via a Vorticity Redistribution Method (VRM),[34] and a boundary element method (BEM) solution. Because only the velocity evaluations have been moved to the GPU, the full method is not expected to perform 17 times faster. Nevertheless, to show that the method is robust and flexible, we present results from a GPU-assisted simulation of flow over a 10:10:1 discoid.

The simulation is that of a flattened ellipsoid composed of 46080 triangular panels oriented 60 degrees to an impulsively-started unit freestream. The Reynolds number based on the diameter is 1000, and $\Delta t = 0.02$. The interparticle spacing is set to $\Delta x = \sqrt{8\,\Delta t/\mathrm{Re}} = 0.012649$. The bucket size is 1024 for the source vortons and 64 for the target vortons, and 8-way VAMSplit trees are used for all trees in the GPU-accelerated velocity-finding calculation. The source vorton bucket size is larger than that for the runs in §IV.B because it was found that larger systems require larger N_b for optimum performance. For comparison, the all-CPU version uses two-way trees with $N_b = 64$ for all trees. The hybrid CPU-GPU simulation was performed on a dual-core Opteron 2216HE running at 2.4 GHz with one NVIDIA 8800 GTX GPU with a core speed of 575 MHz. The all-CPU version used only the dual-core processor.

The simulation ran for 185 time steps, to $t = 3.7$, and the results appear in Figs. 8-10. The simulation ended because available memory was exhausted. The mean velocity and velocity gradient errors were both $\simeq 8 \times 10^{-4}$ at the end of the simulation. Figure 8 shows that the velocity-finding algorithm timing scales as $N_v^{1.24}$, though a small unexplained abrupt jump appears at around $N_v = 2M$. In addition, the non-GPU-accelerated portions of the calculation take an inordinate fraction of the total time, with the diffusion scheme (VRM) always requiring more than twice the time as the GPU-accelerated velocity solution. Figure 9 shows the vorticity at various stages of the calculation, showing the large initial, downward-traveling vortex ring and a complex connection to the braided vortices trailing from the top surface of the discoid. Finally, Fig. 10 illustrates the three-dimensional structure of the wake vorticity, showing the downward-pointing loop and the distribution of vorticity around the object.

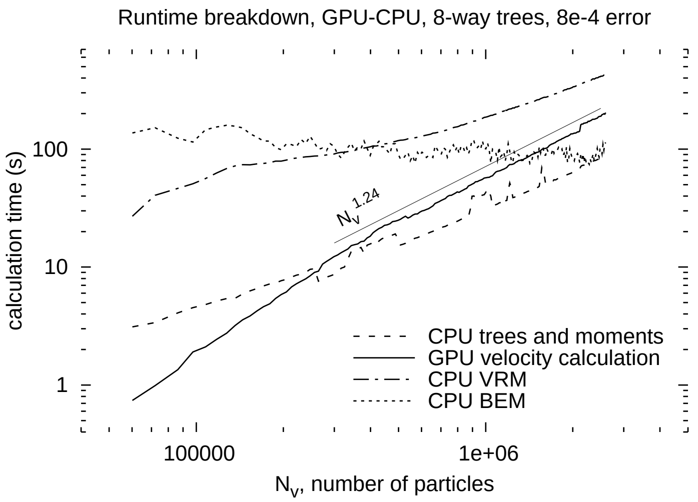

Figure 8. Runtime breakdown per time step of hybrid CPU-GPU method on dynamic case of flow over 10:10:1 discoid at $\theta = 60\,^{\circ}$. Trees are built five times, multipole moments are calculated three times, and velocities are evaluated twice per time step.

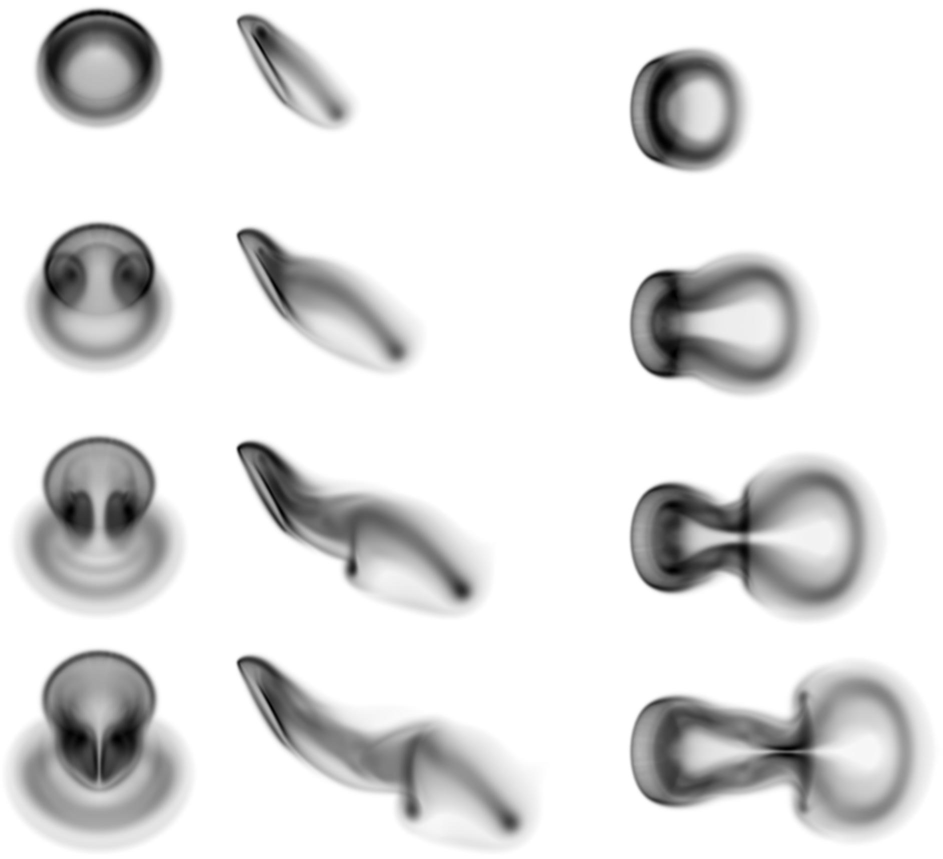

Figure 9. Integrated vorticity magnitudes along z (left), y (middle), and x (right) planes at times $t = 1, 2, 3, 3.7$ (top to bottom); hybrid CPU-GPU method.

The final velocity calculation (at step 185, $t = 3.7$, with $N_v = 2,522,130$) was performed using both the hybrid CPU-GPU and all-CPU methods at equivalent accuracy and using parameters that returned optimum performance. The total calculation times (including tree-building, determining the multipole moments, and computing the near- and far-field velocities) were 116.403 s for the hybrid method and 1117.75 s for the all-CPU method, for a speedup of 9.6. The speedups vs. direct methods were 28.1 for the hybrid, and 1428 for the all-CPU methods (timings for the direct method are based on full velocity evaluations for a subset of target particles).

V. Conclusion

It seems clear from the results above that the multipole treecode algorithm is well-suited to data-parallel computations on recent GPU hardware. Even though the hardware architecture forces significant changes in the structure of the treecode algorithm, the increased parallelism more than makes up for the difference, delivering **17 times the total performance** of an optimized dual-core CPU solution and **over 218 GFlop/s on one GPU** for the direct summation computation. Future work will address the most costly portions of the dynamic simulation (diffusion, BEM) with multi-threaded or data-parallel algorithms, and test the resulting method on distributed-memory hybrid CPU-GPU supercomputers.

Acknowledgments

This project was funded by NASA SBIR Phase I Contract Number NNX07CA24P. The GPU implementation of the far-field (multipole) component was partially supported by SBIR Phase II Grant Number R44RR024300 from the National Center For Research Resources (NCRR). The content is solely the responsibility of the authors and does not necessarily represent the official views of NASA, NCRR or NIH. The authors wish to thank Professor Eric Darve and Erich Elsen (Stanford University) for sharing their invalu-

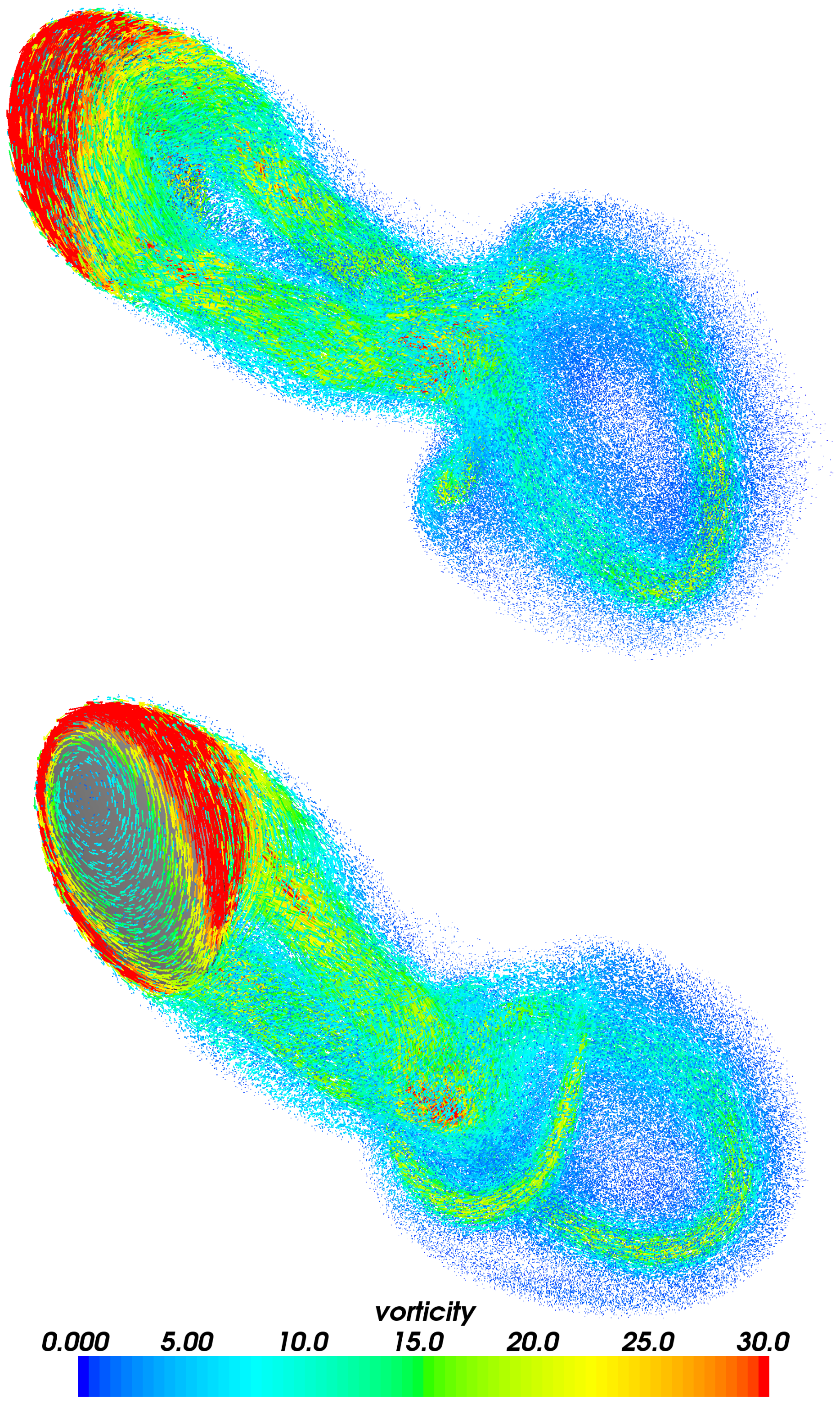

Figure 10. Partial sampling of vortex particles, perspective views from above and oblique (top) and from below and oblique (bottom), at $t = 3.7$; hybrid CPU-GPU method.

able experience in high performance computing with modern GPUs during the early stages of the algorithm development and implementation.

References

[1]Appel, A. W., "An efficient program for many-body simulation," *SIAM J. Sci. Stat. Comput.*, Vol. 6, 1985, pp. 85.

[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]Winckelmans, G. S., Salmon, J. K., Warren, M. S., Leonard, A., and Jodoin, B., "Application of fast parallel and sequential tree codes to computing three-dimensional flows with the vortex element and boundary element methods," *ESAIM Proc.*, Vol. 1, 1996, pp. 225–240.

[5]Wang, Q. X., "Variable order revised binary treecode," *J. Comput. Phys.*, Vol. 200, 2004, pp. 192–210.

[6]Gharakhani, A., Sitaraman, J., and Stock, M. J., "A Lagrangian vortex method for simulating flow over 3-D objects," *Proceedings of ASME FEDSM2005*, Houston, TX, June 19-23 2005.

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

[8]Sugimoto, D., Chikada, Y., Makino, J., Ito, T., Ebisuzaki, T., and Umemura, M., "A special-purpose computer for gravitational many-body problems," *Nature*, Vol. 345, 1990, pp. 33–35.

[9]Fukushige, T., Taiji, T., Makino, J., Ebisuzaki, T., and Sugimoto, D., "A highly-parallelized special-purpose computer for many-body simulations with an arbitrary central force: MD-GRAPE," *Astrophysical Journal*, Vol. 468, 1996, pp. 51–61.

[10]Sheel, T. K., Yasuoka, K., and Obi, S., "Acceleration of vortex method calculation using MDGRAPE-2: A special-purpose computer," *Proc. of the 3rd International Conference on Vortex Flows and Vortex Methods (ICVFM2005)*, Nov. 2005.

[11]Fukushige, T. and Makino, J., "N-body simulation of galaxy formation on the GRAPE-4 special purpose computer," *Proc. of Supercomputing '96 conference*, 1996.

[12]Makino, J., "Treecode with a Special-Purpose Processor," *Publ. of the Astronomical Society of Japan*, Vol. 43, 1991, pp. 621–638.

[13]Makino, J., "Yet another fast multipole method without multipoles—pseudoparticle multipole method," *J. Comput. Phys.*, Vol. 151, 1999, pp. 910–920.

[14]Athanassoula, E., Bosma, A., Lambert, J. C., and Makino, J., "Performance and accuracy of a GRAPE-3 system for collisionless N-body simulations," *Mon. Not. R. Astron. Soc.*, Vol. 293, No. 4, 1998, pp. 369–380.

[15]Spinnato, P. F., *Hybrid systems for N-body simulations*, Ph.D. thesis, Universiteit van Amsterdam, 2003.

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

[17]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.

[18]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.

[19]Elsen, E., Vishal, V., Houston, M., Pande, V., Hanrahan, P., and Darve, E., "N-body simulations on GPUs," 2007.

[20]Belleman, R. G., Bédorf, J., and Zwart, S. F. P., "High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA," *New Astronomy*, Vol. 13, No. 2, 2008, pp. 103–112.

[21]Kupka, S., "Molecular dynamics on graphics accelerators," *Central European Seminar on Computer Graphics for Students, 2006*, 2006.

[22]Xue, F., "Computing the dynamics of large multi-particle systems using Fast Multipole Method with multi-scale time stepping," Tech. rep., University of Maryland, 2006, AMSC664 Final Report.

[23]Siddiqui, M. F., *Hierarchical N-body problem on Graphics Processor Unit*, Master's thesis, Rochester Institute of Technology, 2006.

[24]White, D. A. and Jain, R., "Algorithms and strategies for similarity retrieval," Tech. rep., Visual Computing Laboratory, University of California, San Diego, 1996, Technical Report VCL-96-101.

[25]Warren, M. S. and Salmon, J. K., "Skeletons from the treecode closet," *J. Comput. Phys.*, Vol. 111, 1994, pp. 136–155.

[26]Barnes, J. E. and Hut, P., "A modified treecode: Don't laugh, it runs," *J. Comput. Phys.*, Vol. 87, 1986, pp. 161.

[27]Strickland, J. H., Gritzo, L. A., Baty, R. S., Homicz, G. F., and Burns, S. P., "Fast multipole solvers for three-dimensional radiation and fluid flow problems," *ESAIM Proc.*, Vol. 7, 1999, pp. 408–417.

[28]Cheng, H., Greengard, L., and Rokhlin, V., "A fast adaptive multipole algorithm in three dimensions," *J. Comput. Phys.*, Vol. 155, 1999, pp. 468–498.

[29]Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P., "Brook for GPUs: stream computing on graphics hardware," *ACM Trans. Graph.*, Vol. 23, No. 3, 2004, pp. 777–786.

[30]NVIDIA, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," Tech. rep., NVIDIA Corporation, Santa Clara, CA, June 2007, Version 1.0.

[31]Anderson, C. R., "An implementation of the fast multipole method without multipoles," *SIAM J. Sci. Stat. Comput.*, Vol. 13, 1992, pp. 923–947.

[32]Cottet, G.-H. and Koumoutsakos, P., *Vortex Methods: Theory and Practice*, Cambridge Univ. Press, Cambridge, UK, 1999.

[33]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.

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