Toward Improved Aeromechanics Simulations Using Recent Advancements in Scientific Computing

Qi Hu Nail A. Gumerov Ramani Duraiswami
Department of Computer Science &
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Monica Syal J. Gordon Leishman
Department of Aerospace Engineering
Glenn L. Martin Institute of Technology
University of Maryland
College Park, MD 20742

Many types of aeromechanics simulations are very expensive to perform because the computational cost per time step is quadratic or cubic in the discretization. To improve this situation, it is shown how such simulations can be accelerated by using two approaches: 1. Utilizing powerful parallel hardware for high computational throughput; 2. Developing effective fast algorithms to reduce the computational complexity. The hardware method has been implemented by using parallel programming on graphic processors with hundreds cores, and the algorithmic method was realized by an efficient fast multipole method. An application is shown where the use of both methods is combined to give nearly two orders of magnitude reduction in computational cost.

Introduction

This paper presents details on recent advancements in scientific computing in terms of integrating new hardware and software to greatly enhance the computational efficiency of comprehensive rotorcraft analysis. The focus is on demonstrating the tremendous computational accelerations that are possible (i.e., orders of magnitude speed up) by using algorithmic developments in the form of the fast multipole method (FMM) combined with hardware accelerations by using graphics processors (GPUs). Even today, for many applications a comprehensive rotorcraft aeromechanical analysis may prove too computationally expensive (e.g., for several aspects of design and analysis) because of the significant computational requirements that are needed and the wall-clock times incurred. A good example is when a free-vortex wake analysis is exercised within such comprehensive rotorcraft codes, especially when the simulations are conducted for maneuvering flight conditions and/or over long time scales. To avoid high computational costs, the fidelity of the analysis may have to be compromised either by using prescribed wake or small disturbance linearized models instead (e.g., incarnations of dynamic inflow). These simpler models may also be tuned to predict solutions for specific flight conditions and may have inadequate predictive capabilities for other flight conditions.

Recent interest in the prediction of brownout dust clouds is a good example of a complex problem in aeromechanics that requires the use of comprehensive rotorcraft models. For such problems, which involve the simulation of rotorcraft maneuvering near the ground in dusty two-phase flow environments (Refs. 1–5), the use of a free-vortex wake method is required at a minimum because it is required to model the details of the vortical wake to predict the pickup and convection of the dust particles. However, because of the tracking of substantial number of vortex elements (N) in the flow simulation and a much larger number of particles (M) for the brownout simulations, the basic computational cost of such algorithms is $O(N^2 + NM)$ per time step. This cost may translate into weeks of computational time.

In this paper, two methods are discussed to deal with the practical implementation of such problems:

1. Development of efficient algorithms to reduce the computational cost.
2. Utilization of parallel hardware for high performance computing.

In particular, the fast multipole method (FMM) has complexity $O(N \log N)$ and is shown to be very appropriate for this type of free-vortex wake problem. In combination with parallel processing, this approach can become extremely fast. While each vortex element influences every other vortex element, the influence is mostly on nearby elements. The FMM uses this insight in a precise mathematical way. Given an acceptable tolerance, the FMM groups the vortex elements using hierarchical data structures, computes a "multipole" representation for them of a certain order (related to the tolerance),

Presented at the 67th Annual Forum of the American Helicopter Society, Virginia Beach, May 3–5, 2011.

and then further groups them to compute the effect on particles/elements that are even further away. Nearby interactions are computed directly. The use of data structures allows this grouping to be done automatically and efficiently. The hardware acceleration was implemented using graphics processors (GPUs) that have a low cost highly parallel architecture consisting of hundreds of cores. The results show the feasibility of this approach for a variety of potential applications in rotorcraft aeromechanics, and show that for some problems computational accelerations up to two orders of magnitude faster may be possible.

Brief Summary of Previous Work

The idea of utilizing parallel computing and multi-core computer architectures for aerodynamic computations is not new, with initial work to parallelize free-vortex wake simulations going back at least two decades (Ref. 6). In the context of the present approach, recent scientific computing developments related to the N-body problem are more relevant. Nyland and Harris (Ref. 7) developed a GPU implementation of the N-body simulation. However, their method was based only on hardware accelerations; they simply summed directly all pairwise interactions and so the cost of the algorithm remained $O(NM)$ and cannot be scaled to larger problems. Gumerov and Duraiswami (Ref. 8) presented the first realization of the FMM on a GPU, which was a scalable hierarchical algorithm. However, in their work only the most expensive run of the algorithm was implemented on the GPU, while the data structures required for the FMM were computed on the CPU and transferred to the GPU.

Stock and Gharakhani (Ref. 9) developed the CPU-GPU-hybrid treecode to accelerate the computations. However, according to their results its overall performance cannot surpass the FMM/GPU method implementation method presented in Ref. (Ref. 8). Based on the treecode work, Stock et al. (Ref. 10) developed a hybrid Eulerian-Lagrangian method on a multi-core CPU-GPU cluster to model rotor wakes, which can simulate high Reynolds number wake flows at reasonable cost. Stone et al. (Ref. 11) also presented a combined CPU/GPU parallel algorithm for a hybrid Eulerian-Lagrangian CFD code, where multilevel parallelism was demonstrated in the coupled OVERFLOW/VPM algorithm by using MPI, OpenMP, and CUDA.

GPU Architectures

A graphical processing unit (GPU) is a highly parallel, multithreaded, many-core processor with high computational power and memory bandwidth. GPUs were developed originally for graphical rendering to efficiently process single instructions on multiple data (SIMD). Hence, more transistors on a GPU chip are devoted to data processing rather than to

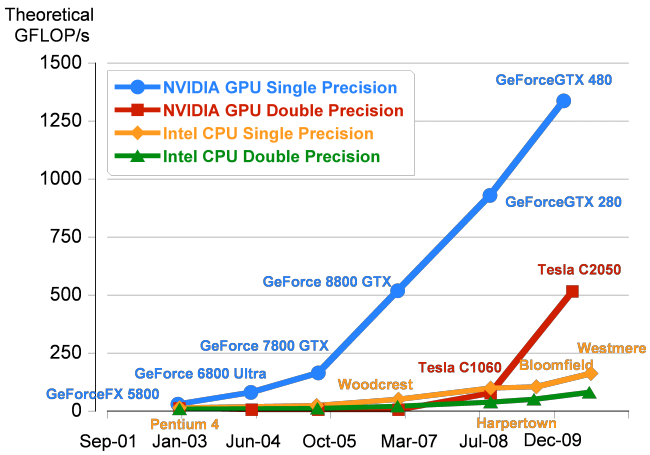


Fig. 1: GPU computation power (Ref. 13).

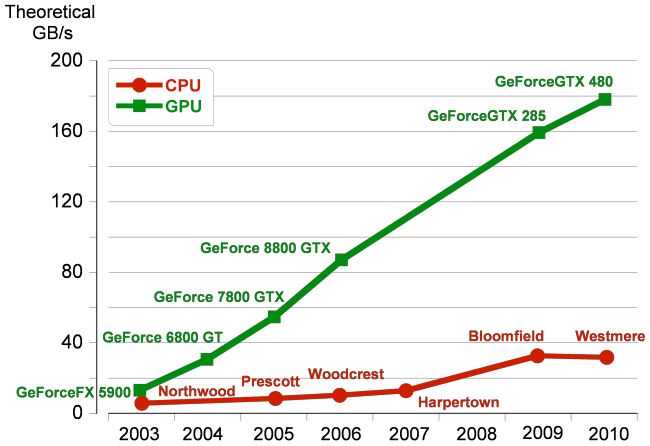


Fig. 2: GPU memory bandwidth (Ref. 13).

data caching and flow control. Over the years, GPU architectures have evolved tremendously (see Figs. 1 and 2) and current GPUs are capable of both single and double precision computations. GPUs employ a threaded parallel computational model. High-end GPUs designed for computations are already incorporated into many top high performance clusters to achieve computational speeds in the high Tera-FLOPs (Ref. 12).

Data on the host (CPU) are transferred to the device (GPU) memory back-and-forth during the entire computational process (Fig. 3). However, these host-device memory communications are expensive to perform compared to the GPU computations. Therefore, one GPU programming philosophy is to minimize the data transfer while processing the data on the GPU as much as possible per data transfer. The memory architecture on the GPU is hierarchical. In the current NVIDIA Fermi architecture (Ref. 14) on which the present simulations were performed using the FMM, there are four different kinds of memories:

1. *Global memory*: Device DRAM memory with slow accessing speed but large size. This is used to keep data

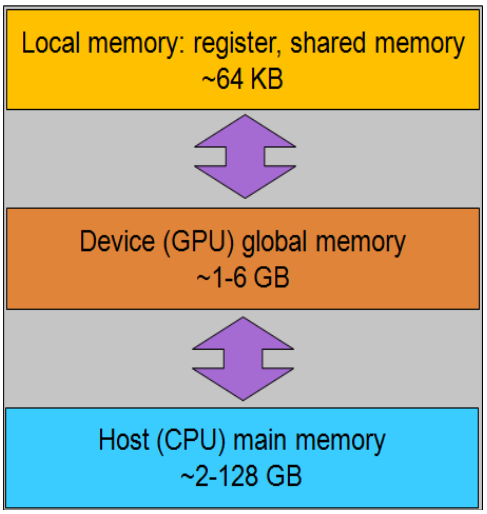


Fig. 3: Host and device memories.

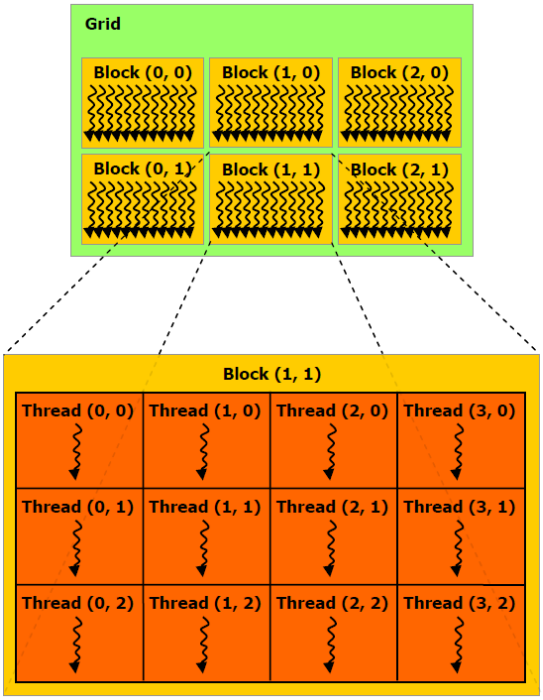


Fig. 4: Thread blocks and grids.

and communicate with the host main memory.

2. *Constant memory*: 64KB read only constant cache shared by all the threads. This is used mainly to store constant values shared by all the threads.

3. *Shared memory*: 64 KB of fast on-chip memory for each steaming multiprocessor (SM). It can be configured as 48 KB of Shared memory with 16 KB of L1 cache or as 16 KB of Shared memory with 48 KB of L1 cache. Accessing shared memory is much faster than global memory.

4. *Registers*: 32KB fast on-chip registers for each SM. They are the fastest memory among all the memory hierarchy and are mainly used to hold instruction's input operands and values.

One main GPU programming challenge is to efficiently use these hierarchical memories in the threaded model given the tradeoff between speed and size. *General Purpose GPU* (GPGPU) computations have become popular in the scientific community since 2000. However, programming on the GPU remained a technical barrier because it required a deep knowledge of computer graphic architectures. In 2006, NVIDIA released a general-purpose parallel computing architecture called CUDA so that a programmer can more easily manipulate the structure of the GPU and so use a large number of threads executing in parallel.

The CUDA programming is almost the same as C except that the programmer is given techniques to handle:

1. A hierarchy of threaded groups (Fig. 4);
2. Different kinds of memory;
3. Synchronization mechanisms.

The state of the art for GPGPU applications is discussed in Refs. 15 and 16. OpenCL (Open Computing Language) is another way to program GPUs (Ref. 17) using a similar library. In fact, OpenCL is a framework for writing programs that can execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. However, in the present paper the intention is to explain how to map fast algorithms onto the GPU using CUDA so that they can potentially achieve good performance gains for different rotorcraft aeromechanical simulations. These results should also extend to OpenCL.

Use of GPUs in Calculations

For demonstration in the present work, the quadratic cost brownout simulation methodology without the FMM as developed in by Syal et al. (Ref. 5) was implemented on the GPUs. The sediment particles that are otherwise stationary on the ground can be mobilized by the aerodynamic action of the rotor wake. These sediment particles are usually rather small (i.e., sizes varying between 1 μm and 100μm), and once entrained, they can remain suspended in the flow to produce obscurations that limit the ability for the pilots to see the ground. Therefore, the onset of brownout conditions poses a safety of flight issue.

One of the primary challenges involved in the modeling of brownout is in tracking of millions of particles over hundreds of rotor revolutions, which make it a computationally formidable problem. Syal et al. (Ref. 5) used a free-vortex method to model the rotor flow, so the cost of the brownout

simulation at each time step is $O(N^2) + O(NM)$, where N is number of free vortex segments (typically of the order of thousands) and M is number of dust particles (of the order of millions). If performed on a single CPU, the entire simulation in FORTRAN can take up to days or weeks. These computations can also be performed in parallel on the multi-core architecture of the GPUs, and can potentially provide huge performance gains. In the present case, the CUDA parallel codes were incorporated into the traditional FORTRAN codes using FLAGON (Ref. 18).

Figure 5 shows snapshots of the computed dust clouds during a landing maneuver for a representative single rotor helicopter with a 4-bladed rotor; see Ref. 5. A total of 1 million particles were used in this case to obtain the dust cloud. To obtain 35 seconds of actual simulation time (approximately 150 rotor revolutions) of the dust cloud development computed in a CPU only environment takes about 2–3 days of wall-clock time. If implemented on the GPUs, the same solution can be obtained in 2–3 hours, and without compromising accuracy. The performance gains obtained from the GPU depend upon the problem parameters and the desired precision. Figure 6 shows the computational performance gains obtained for such brownout simulations as implemented on both single-precision and double-precision GPU architectures for different particle counts. The performance gain is defined as a ratio of wall-clock times required to run a case using GPU architecture and the full CPU simulation. The results obtained are clearly very good: 250 times speed-up for single-precision and 25 times speed-up for double-precision.

The accuracy of both single and double-precision implementations of this problem on the GPU was compared with the double-precision CPU results for the brownout landing maneuver. The results showed that the double-precision GPU computations are as accurate as the CPU computations, whereas the accumulated errors in single-precision become unacceptable for larger times. Therefore, for the present application double-precision GPU computations must be used, although for some other problems single-precision may suffice. The maximum size of the calculations that can be performed on a single GPU are limited by the quadratic cost. To consider even larger problems, algorithmic accelerations such as provided by the FMM need to be provided, which is discussed next.

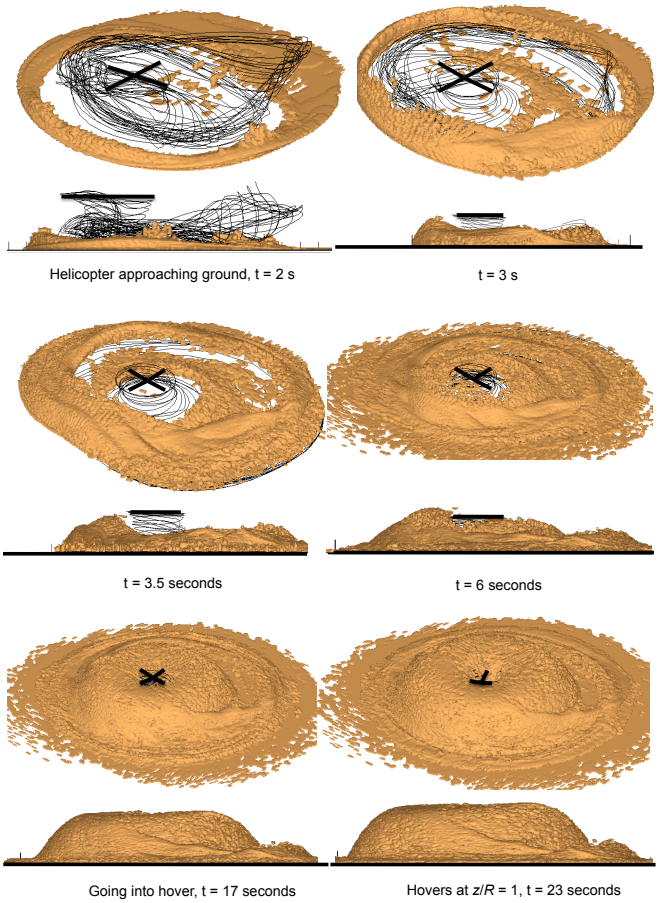


Fig. 5: Simulations of the development of a brownout dust cloud at six times during a simulated landing maneuver.

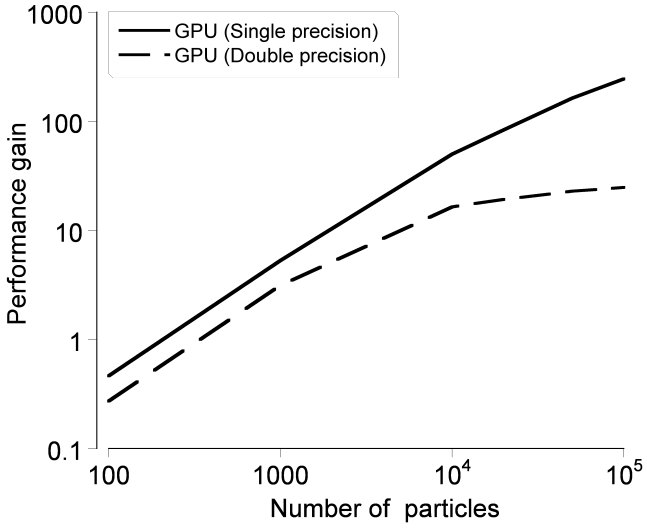


Fig. 6: Performance gain with GPU implementation of the dust cloud simulation.

Fast Multipole Method

The fast multipole method (FMM) was first introduced by Greengard and Rokhlin in Ref. 19 and has been identified as one of the ten most important algorithmic contributions in the 20th century (Ref. 20). Since this initial publication, hundreds of papers on applications of the FMM have been published , and the theory of the FMM has been well established and developed. In short, the main purpose of FMM is to speed up

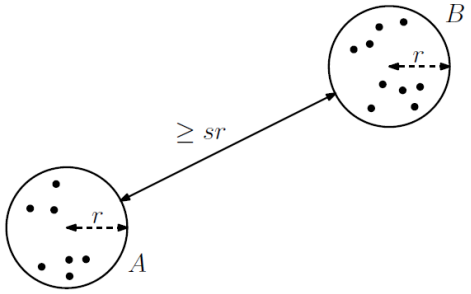


Fig. 7: A well separated pair.

the matrix-vector product

$$\phi(\mathbf{y}_j) = \sum_{i=1}^{N} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i), \quad j = 1,2,\ldots,M, \quad \mathbf{x}_i, \mathbf{y}_j \in \mathbb{R}^d, \quad (1)$$

where $\{\mathbf{x}_i\}$ are source points and $\{\mathbf{y}_j\}$ are receiver points of dimension d with the strengths q_i. In the context of the free-vortex analysis, the sources and receivers are equivalent to the vortex segments representing the rotor wake and the rotor blades. Correspondingly, for the particle tracking algorithm the sources will be vortex segments and receivers are particles. The quantity Φ used in this equation will correspond to the Biot–Savart kernel. Accelerations when using the FMM are achieved because the summation is computed approximately, but with a guaranteed specified accuracy, which may be set explicitly, e.g., to the machine precision.

The main idea in the FMM is to split the summation in Eq. 1 into a near and a far field solution as

$$\phi(\mathbf{y}_j) = \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i) + \sum_{\mathbf{x}_i \in \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i), \quad (2)$$

for $j = 1,2,\ldots,M$ where Ω is the neighborhood domain. The second sum here can be computed exactly at $O(N)$ cost. The most expensive computations are related to the first sum. In the FMM, factored approximate representations of the functions in the far-field are utilized. These representations come from local and multipole expansions over spherical basis functions, which are truncated to retain the p^2 terms. The truncation number p (the maximum degree of the spherical harmonics is $p-1$) is a function of the specified tolerance $\varepsilon = \varepsilon(p)$. Larger values of p provide better accuracy with the FMM, but also increases the computational time. Therefore, the selection of p is based on the overall accuracy requirements and the accuracy of the other components of a bigger code.

These factored representations for the far field are expressed in terms of singular (multipole) spherical basis functions, S_l, and regular (local) spherical basis functions R_l. These factorizations can be used to separate the computations involving the points sets $\{\mathbf{x}_i\}$ and $\{\mathbf{y}_j\}$, and consolidate operations for many points as

$$\begin{aligned} & \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i) \\ = & \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i \sum_{l=0}^{p-1} S_l(\mathbf{y}_j - \mathbf{x}_*) R_l(\mathbf{x}_i - \mathbf{x}_*), \\ = & \sum_{l=0}^{p-1} S_l(\mathbf{y}_j - \mathbf{x}_*) \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i R_l(\mathbf{x}_i - \mathbf{x}_*), \\ = & \sum_{l=0}^{p-1} C_l S_l(\mathbf{y}_j - \mathbf{x}_*), \end{aligned} \quad (3)$$

where the coefficients C_l for all $\mathbf{x}_i$ are built and then used in the evaluation at all $\mathbf{y}_j$. This approach reduces the cost of this part of the summation and memory to $O(N+M)$.

Because the factorization in Eq. 3 is not global, the split between the near- and far-fields must be managed, which requires appropriate data structures and a variety of representations. The geometric structure that encodes much of the FMM data information, such as grouping of points and finding neighbors, is called a *well-separated pair decomposition* (WSPD), see Fig. 7, which is itself useful for solving a number of geometric problems (Ref. 21, chapter 2). Assume all of the data points are already scaled into a unit cube. This WSPD is recursively performed to subdivide the cube into subcubes using *octrees* until the maximal level, l_{max}, or the tree depth, is achieved. The level l_{max} is chosen such that the computational costs of the local direct sum and far field translations can be balanced to the extent possible. Moreover, the total data structure construction must be completed at cost $O(N+M)$ to be consistent with the overall cost of the FMM.

The difference between the FMM and the straightforward method is shown in Fig. 8. Given N sources and M receivers, the computational cost of the straightforward algorithm is $O(NM)$, which is shown on the left. In contrast, the multilevel FMM (MLFMM) will put the sources (green points) into hierarchical space boxes and translate the consolidated interactions of sources into receivers (black points), as shown on the right. The red lines show the multipole to local translation $(\mathcal{S}|\mathcal{R})$.

For the convenience of expression, call the box containing the source points the *source box* and the box containing the receiver points the *receiver box*. Then the whole FMM algorithm, whose hierarchical execution procedures are summarized in Fig. 9, consists of four main parts: the initial expansion, the upward pass, the downward pass and the final summation.

1. **Initial expansion**:

   (a) In the finest level l_{max}, all the source data points are expanded at their box centers to obtain the far-field $\mathcal{S}$ expansions $\{C_n^m\}$ over p^2 spherical basis functions. The truncation number p is determined by the required accuracy.

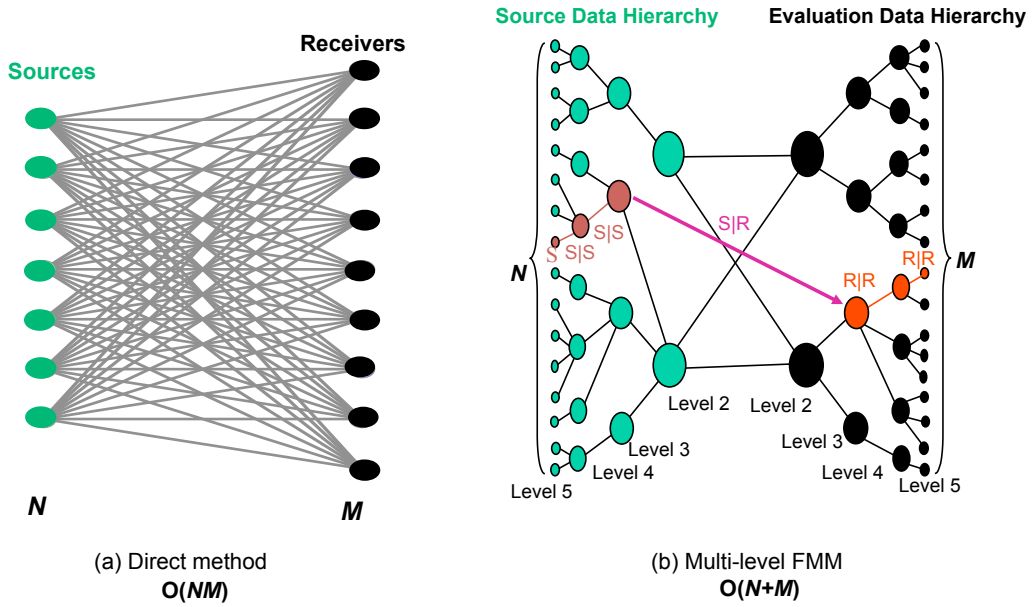


Fig. 8: FMM (multi-level) versus direct method. In the straightforward method the number of operations is $O(MN)$. The number of operations (connecting lines) can be reduced to $O(M+N)$ for the FMM with multi-levels. The maximal level l_{max} is level 5 in this case.

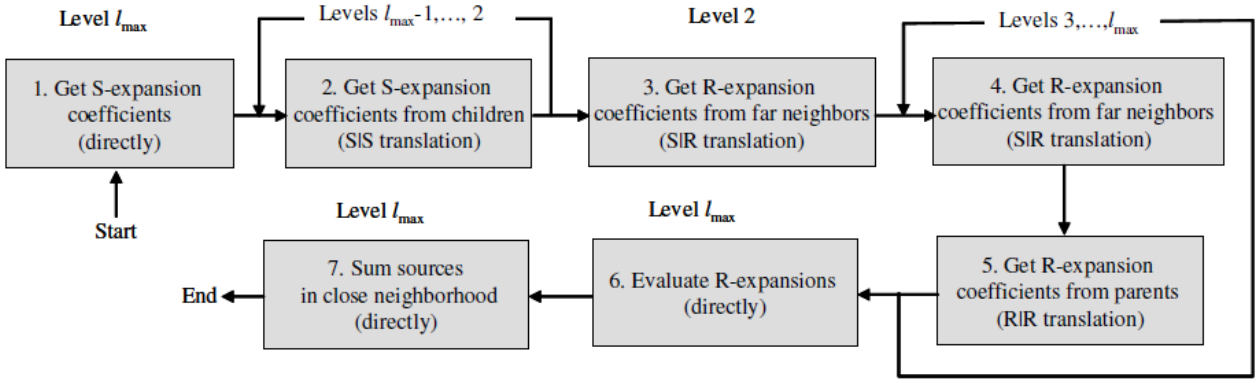


Fig. 9: A flow chart of the standard FMM.

(b) The obtained $\mathcal{S}$-expansions from all source points within the same boxes are consolidated into a single expansion.

2. **Upward pass**: For levels from l_{max} to 2, the $\mathcal{S}$ expansions for each box are evaluated via the multipole-to-multipole ($\mathcal{S}|\mathcal{S}$) translations from the children source boxes to their parent source box. All these translations are performed in a hierarchical order up the tree.

3. **Downward pass**: For levels from 2 to l_{max}, each receiver box also generates its local or $\mathcal{R}$ expansions in a hierarchical order from level to level.

   (a) Translate multipole $\mathcal{S}$ expansions from the source boxes of the same level belonging to the receiver box's parent neighborhood but not the neighborhood of that receiver itself, to local $\mathcal{R}$ expansions via multipole-to-local ($\mathcal{S}|\mathcal{R}$) translations then consolidate the expansions.

   (b) Translate the $\mathcal{R}$ expansion (if the level is 2, then these expansions are set to be 0) from the parent receiver box center to its child box centers and consolidate with the same level multipole-to-local translated expansions.

4. **Final summation**: Evaluate the $\mathcal{R}$ expansions for all the receiver points at the finest level l_{max} and performs a local direct sum of nearby source points within their neighborhood domains.

Notice that the evaluations of the nearby source point direct sums are independent of the far-field expansions and translations, and may be scheduled according to convenience. It is important to balance the costs of these sums and the translations to achieve better computational efficiency and proper scaling. Reference 22 gives more details on the algorithm, Ref. 23 describes the translation theory, and Ref. 24 discusses the different translation methods.

FMM Data Structure

In the previous FMM approach used on GPU implementations (Ref. 8), only the FMM computations were performed on GPUs while the FMM data structures were still developed on CPUs and then transferred to GPUs. This implementation is not the most optimal solution for most of types aeromechanics problems because of the frequent data transfers between the CPUs and GPUs, which as explained earlier should be minimized for optimal GPU performance. In the present work, a novel and efficient technique was developed to fully implement the FMM (including data structures) on the GPUs.

The basic FMM data structures are inherited from the *octree* data-structure (Ref. 21, chapter 2). For different levels, the unit cube, which contains all the spatial points are uniformly subdivided into subcubes using the octree (8 sub-boxes at level 2, 64 sub-boxes at level 3 and so on). Basic operations include specifying box indices, computing box centers, and performing query operations to find the box indices of particle locations, etc. at different levels; see Refs. 25 and 22(Section 5.3) for these basic operational details. To process near-field direct sums and hierarchical translations efficiently, different space neighborhood domains have to be defined appropriately (Ref. 22). Given each box with index $n = 0, \ldots, 2^{ld}$ at level $l = 0, \ldots, l_{max}$ in d dimensions:

1. $E_1(n,l) \subset \mathbb{R}^d$ denotes the spatial points inside the box n at level l.
2. $E_2(n,l) \subset \mathbb{R}^d$ denotes the spatial points inside the neighborhood of the box n at level l.
3. $E_3(n,l) = \overline{E_2(n,l)} \subset \mathbb{R}^d$ denotes spatial points outside the neighborhood of the box n at level l.
4. $E_4(n,l) = E_2(\texttt{ParentIndex}(n), l-1) \backslash E_2(n,l) \subset \mathbb{R}^d$ denotes spatial points inside the neighborhood of the parent box $\texttt{ParentIndex}(n)$ at level $l-1$ but do not belong to the neighborhood of box n at level l.

A 2d case is shown in Fig. 10 with the box index being n at level l. Although several papers have been published on fast Kd-tree and octree data structures that are similar to the spatial data structures used here, they cannot be directly applied to the FMM framework. A novel GPU parallel solution for the FMM data structure construction will now be described.

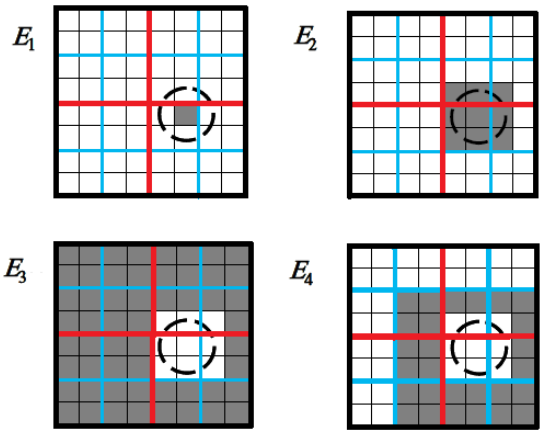


Fig. 10: E_1, E_2, E_3, E_4 neighborhoods of dimension 2: red division at level 1; blue division at level 2; black division at level 3.

Bin Sort in Linear Time To build the FMM data structures, empty boxes need to be skipped to organize the points into a tree structure with each box at the finest level holding at most a prescribed number of points. This process ensures that the costs of the local summation and the far-field summation are approximately balanced. This algorithm is called "Bin-sort." Let all the box indices be stored as an *unsigned int*, with `Src` representing source points and `Recv` representing receiver points. A nonempty source box means that the box contains at least one source data point, while a nonempty receiver box means that the box contains at least one receiver data point.

At the finest tree level, all data points stored in `P[]` are sorted according to their box indices with linear computational cost. This is called *bin-sort*. This is not a real sort because the order of the data points within a box is irrelevant, hence they are arranged by thread access order. Each data point has a 2d vector called `sortIdx`, where `sortIdx.x` stores its residing box index and `sortIdx.y` stores its rank within the box. There is an unsigned integer array `bin` allocated for the boxes in the maximal level. Its *i*th entry `bin[i]` stores the number of data points within the box *i*, which is computed by the `atomicAdd` (Ref. 13) in the GPU implementation. Let the number of data points be M. Then the pseudocode for the bin-sort is given by Algorithm 1.

Algorithm 1 GPU Bucket-Sort.

```
launch M threads;
for all threads (indexed by i) do
    reads P[i];
    performs SortIdx[i].x=BoxIndex(P[i]);
    performs atomicAdd[Bucket[SortIdx[i].x]];
    sets SortIdx[i].y=Bucket[SortIdx[i].x];
end for
synchronize threads;
```

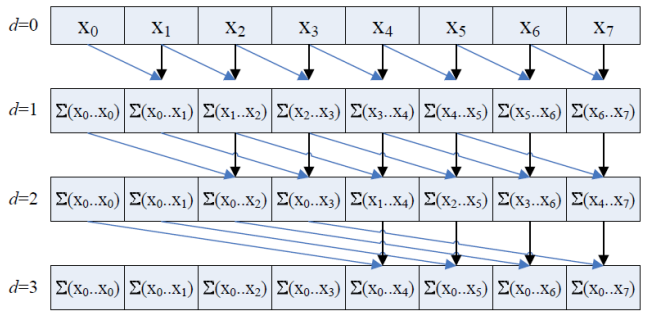


Fig. 11: The naïve prefix sum (scan) algorithm.

Auxiliary Arrays One significant challenge of using FMM data structures is to access the data efficiently. The use of pre-computed data structures is preferred because on GPUs memory access costs are higher than pure computations. Auxiliary pre-computed arrays used for this purpose are `SrcBookMark[]`, `RecvBookMark[]`, `SrcNonEmptyBoxIndex[]`, `NeighborBookMark[]`, `NeighborList[]`. The details are provided in Appendix A.

Parallel Data Structure Construction The bookmark of an element is actually the rank of that element, hence it requires a reduction operation on the `bin[]` array. More exactly, given the `bin[]` array of length n obtained from the data points (either source or receiver points), the `BookMark[]` array is computed as

$$\texttt{BookMark[j]} = \sum_{i=0}^{j} \texttt{bin[i]}, \quad for \;\; j = 0, \ldots, n+1, \qquad (4)$$

Recall that `bin[i]` stores the number of data points within the box i. Hence, there are `BookMark[j]` data points in the sorted data point array from box 0 to box j, which can be used to identify the data points of box j in the sorted data point array.

In fact, `BookMark[]` is the so-called *prefix sum (scan)* of `bin[]`. A naïve algorithm to compute the scan is shown in Fig 11. A highly efficient parallel scan (Ref. 26) is used in the present implementation to perform such kinds of operations.

Basic octree operations, such as box index query, box center query, E_2 and E_4 neighbor index query and parent/children query, are implemented as inline CUDA `__device__` functions. For computational efficiency, the device function either operates on local register or on the coalesced memory. The testing results shows that even for the costly computation of E_4 neighbors, the running time can be neglected in comparison to memory access. Within the translation part, only these basic operations are used. Hence, the necessary data structure for translations are constructed on the fly.

Data Structure Implementation Test To test the performance of data structure construction and the GPU FMM implementation, data of different sizes up to 10^7 were used. The source data points are different from the receiver data points but with the same size. The CPU codes used for comparisons are optimized, but only single-threaded without any streaming SIMD extensions (SSE) instructions.

Table 1: The data structure construction speedup.

Level	CPU time (ms)	GPU time(ms)	Speedup
3	223.42	7.686	29.068
4	272.254	13.949	19.517
5	430.616	12.959	33.229
6	1808.414	34.591	52.280
7	6789.339	70.847	95.831
8	7782.755	124.859	62.332

In the data structure comparison experiment (single precision), 1048576 source points and receiver points are generated uniformly within the unit. The computer hardware configurations used was GPU NVIDIA GTX480 versus CPU Intel Xeon Nehalem quad-core running at 2.8GHz. The experimental results are summarized in Table 1.

FMM GPU Implementation

The FMM has already been implemented on the GPU, as reported in the literature (Refs. 22, 27). Although recent work has already shown its application to real problems, their published levels performance cannot surpass the current implementation in respect to both timing and accuracy. Details and test results are provided in this section, and more comprehensive comparisons with FMM implementations on other aeromechanics problems will be shown in the future.

Theories and formulas of the FMM expansions and translations for the real valued Laplace kernel were derived by using complex arithmetic. However, these complex representations can result in extra costs by having to design complex number arithmetic routines and special functions that can use complex arguments. These kinds of computations are less efficient to parallelize on a GPU.

Alternatively, a real number version of these expansions and translations can be derived by using their symmetrical properties. One big advantage of the real number representations is that the GPU can process these real numbers much more efficiently than complex numbers. In the following discussion, both spherical coordinates (r, θ, φ) and Cartesian coordinates (x, y, z) are used to establish real FMM expansions and translations. These necessary real modifications include: local multipole expansions and RCR decomposition of the translations (Ref. 28) (Ref. 24, section 2.3).

Local Expansions Local expansions are derived using real numbers. These recurrence relations are the same as used in

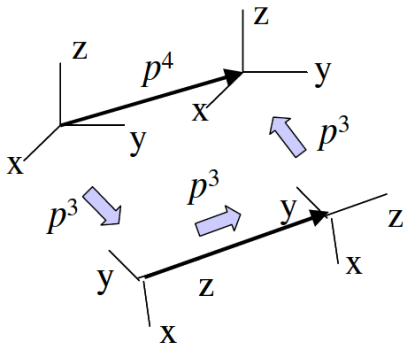


Fig. 12: RCR translation for the Fast Multipole Method replaces one $O(p^4)$ translation with two $O(p^3)$ rotations and one $O(p^3)$ coaxial translation.

Ref. 22. The details and the mathematical formulas are explained in Appendix B. The current implementation uses the GPU registers to optimize the performance of the expansion.

RCR Decompositions The implementation uses the $O(p^3)$ RCR decomposition (Fig. 12) to perform the $\mathcal{S}|\mathcal{S}$, $\mathcal{S}|\mathcal{R}$ and $\mathcal{R}|\mathcal{R}$ translations. Details of the translation and rotation formulas can be found in Ref. 24. All of the FMM expansions and translations can be performed as real with fast implementations on the GPU.

Adaption to the Biot–Savart 3D Kernel

One of the fundamental principles governing vortex motion is Helmholtz's theorem, which says that vortex elements move with the fluid particles (Ref. 29). At each time step, the induced velocity at the fluid/vortex element $\mathbf{y}$ is given by

$$V(\mathbf{y}) = \sum_{i=1}^{n} \frac{\mathbf{q}_i \times (\mathbf{y}-\mathbf{x}_i)}{|\mathbf{y}-\mathbf{x}_i|^3}. \tag{5}$$

As for the particles, their induced velocities are determined by the vortex elements which is also described by the Eq. 5.

The baseline FMM algorithm computes the Coulomb potentials generated by the source points. The kernel function Φ is defined by

$$\Phi(\mathbf{y},\mathbf{x}) = \begin{cases} \dfrac{1}{|\mathbf{y}-\mathbf{x}|}, & \text{if } \mathbf{x} \neq \mathbf{y}, \\ 0, & \text{if } \mathbf{x} = \mathbf{y}. \end{cases} \tag{6}$$

The intension is to minimally modify this equation to adapt to the preferred Biot–Savart kernel. Notice that

$$\begin{aligned} \nabla_{\mathbf{y}} \times \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|} &= (\nabla_{\mathbf{y}} \frac{1}{|\mathbf{y}-\mathbf{x}_i|}) \times \mathbf{q}_i \\ &= -(\frac{\mathbf{y}-\mathbf{x}_i}{|\mathbf{y}-\mathbf{x}_i|^3}) \times \mathbf{q}_i \\ &= \frac{\mathbf{q}_i \times (\mathbf{y}-\mathbf{x}_i)}{|\mathbf{y}-\mathbf{x}_i|^3}. \end{aligned} \tag{7}$$

Therefore, Eq. 5 can be written using Eq. 7 as

$$V(\mathbf{y}) = \sum_{i=1}^{n} \frac{\mathbf{q}_i \times (\mathbf{y}-\mathbf{x}_i)}{|\mathbf{y}-\mathbf{x}_i|^3} = \sum_{i=1}^{n} \nabla_{\mathbf{y}} \times \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|}. \tag{8}$$

Based on Eq. 8, the baseline FMM is applied three times with three coordinates components of the vector weights $\mathbf{q}_i = (q_i^{(x)}, q_i^{(y)}, q_i^{(z)})$ first. Then in the final evaluation step, the following $\mathcal{R}$-expansion coefficients for each non empty receiver box, which center is $\mathbf{c}$, are available:

$$\begin{aligned} \{d_n^{(x),m}\}: \quad \sum_{i \notin \Omega_{\mathbf{c}}} \frac{q_i^{(x)}}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} d_n^{(x),m} R_n^m(\mathbf{y}-\mathbf{y}_{\mathbf{c}}), \\ \{d_n^{(y),m}\}: \quad \sum_{i \notin \Omega_{\mathbf{c}}} \frac{q_i^{(y)}}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} d_n^{(y),m} R_n^m(\mathbf{y}-\mathbf{y}_{\mathbf{c}}), \\ \{d_n^{(z),m}\}: \quad \sum_{i \notin \Omega_{\mathbf{c}}} \frac{q_i^{(z)}}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} d_n^{(z),m} R_n^m(\mathbf{y}-\mathbf{y}_{\mathbf{c}}), \end{aligned} \tag{9}$$

which form the vector expansion coefficients $\mathbf{d}_n^m = (d_n^{(x),m}, d_n^{(y),m}, d_n^{(z),m})$ i.e.,

$$\{\mathbf{d}_n^m\}: \sum_{i \notin \Omega_{\mathbf{c}}} \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|} = \sum_{n=0}^{p} \sum_{m=-n}^{n} \mathbf{d}_n^m R_n^m(\mathbf{y}-\mathbf{y}_{\mathbf{c}}) \tag{10}$$

Therefore,

$$\begin{aligned} \nabla_{\mathbf{y}} \times \sum_{i \notin \Omega_{\mathbf{c}}} \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} \nabla_{\mathbf{y}} \times [\mathbf{d}_n^m R_n^m(\mathbf{y}-\mathbf{y}_{\mathbf{c}})] \\ &= \sum_{n=0}^{p} \sum_{m=-n}^{n} \nabla_{\mathbf{y}} R_n^m(\mathbf{y}-\mathbf{y}_{\mathbf{c}}) \times \mathbf{d}_n^m. \end{aligned} \tag{11}$$

While the direct summation is computed as the baseline FMM except by replacing Coulomb kernel by the Biot–Savart kernel, the (x,y,z) components of the gradient of the basis functions $R_n^m(\mathbf{y}-\mathbf{y}_{\mathbf{c}})$ need to be computed according to Eq. 11. However, by differentiating Eq. A9 in Appendix B with respect to x,y and z, these gradients can be obtained recursively. In fact, the recursions for the derivatives of the basis functions depend on the basis functions, while the recursion coefficients are very similar. In implementation, a simple routine can be used to compute all the four sets of the basis functions $\{d_n^m\}, \{d_n^{(x),m}\}, \{d_n^{(y),m}\}, \{d_n^{(z),m}\}$. The purpose of combining these calls is to hide the extra computation (compared with one call) by global memory accessing such that the extra computation can be performed for no cost.

Test and Error Analysis

As mentioned previously, three baseline FMM calls are integrated into one call because of the similarities of the recursion coefficients. A significant advantage of this implementation is that extra computational costs can be hidden from expensive global memory access. In the downpass translation steps,

Table 2: FMM running time on different kernels.

N	Coulomb kernel (ms)	Biot–Savart kernel (ms)
1048576	1074.1	2159.1
262144	565.7	975.4
65536	418.3	669.6
16384	129.1	215.7
4096	97.8	153.1
1024	89.8	136.1

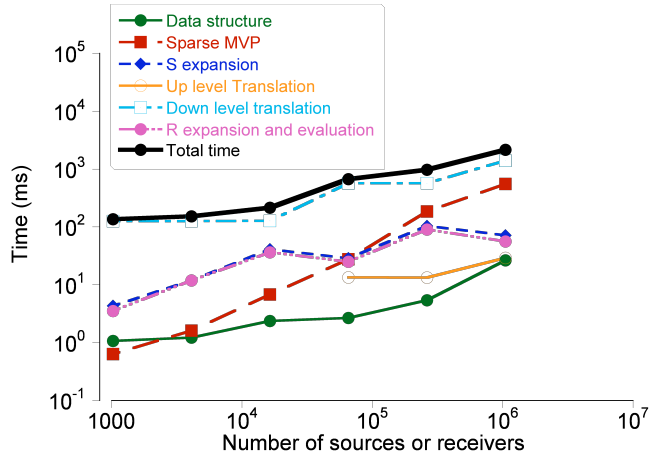
both the indices and the processing order of the E_4 neighbors for each receiver box are quite different among active threads. Therefore, it is impossible to make translation data accessing coalesced threads in the same warp, which results in significant data fetching time. However, combining three calls into one call reduces three memory accesses to one. Even though the data being fetched is the same, the total access time is reduced. Numerical experiments (single precision) summarized in Table 2 show that the full FMM computation time of the Biot–Savart kernel is not tripled but is even less than twice the baseline FMM running time.

The profiling of all parts of the FMM for the Biot–Savart kernel is shown on the top of Fig. 13(a). The main idea of profiling is to identify the most time consuming part of the current FMM algorithm and how each part scales as the number of data points increases. The results show that the down pass translation is the most computationally expensive part and the data structure construction time is so small that it can be neglected. Notice that all the parts show linear scaling, showing that the computation of the Biot–Savart kernel is now no longer quadratic.

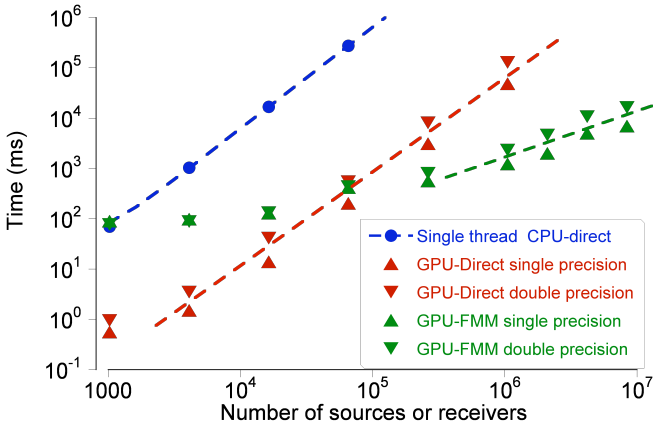
Another experiment was performed to compare the Biot–Savart kernel computation time when implemented directly on the CPU in double precision, GPUs (in both double and single precision), and using the FMM on GPUs (again, in both double and single precision). The results of this study are shown in Fig. 13(b). The GPU direct method implementation was optimized as shown in Ref. 7, but it was still of quadratic cost like the direct CPU implementation. Notice that the GPU-based FMM showed a linear computational cost for large numbers of data points. For these large numbers of data points, the overhead and latency can be neglected.

A comparison of time taken for the computation of the full interaction of 10 million particles using different methods reveals the benefits of the FMM on GPU computations over other methods. Notice that for 10 million receivers, the time taken on a single CPU was of the order of months (on CPU clusters it would be weeks), on a single GPU it was of the order of hours, whereas using the FMM on the GPUs it was only seconds. Therefore, this result shows that a tremendous improvement in run times can be achieved by using the FMM on GPUs for large number of receiver and source points.

This outcome is of great value in many rotorcraft aerome-



(a) FMM profiling



(b) CPU time comparison

Fig. 13: Top is the profiling of FMM for Biot–Savart kernel; bottom is the time comparisons with other methods.

chanics problems, because it not only means that the time required to obtain solutions will be much shorter, but also that the fidelity of the analysis can be increased because computational costs no longer remain limiting factor in actually solving the problem.

Realizing that huge speedups can be obtained by using the FMM, it is important to evaluate the error incurred by these approximations. The error introduced by the FMM is determined by the truncation number p. The theory on the FMM error analysis can be found in Ref. 22. In this experiment, the relative errors are defined as

$$\varepsilon = \sqrt{\frac{\sum_{j=1}^{k} |\phi_{exact}(\mathbf{y}_j) - \phi_{approx}(\mathbf{y}_j)|^2}{\sum_{j=1}^{k} |\phi_{exact}(\mathbf{y}_j)|^2}} \qquad (12)$$

which are computed by picking k =100 testing points for each of the cases. The *exact* values to measure the FMM error are computed by the direct method on CPU using double precision.

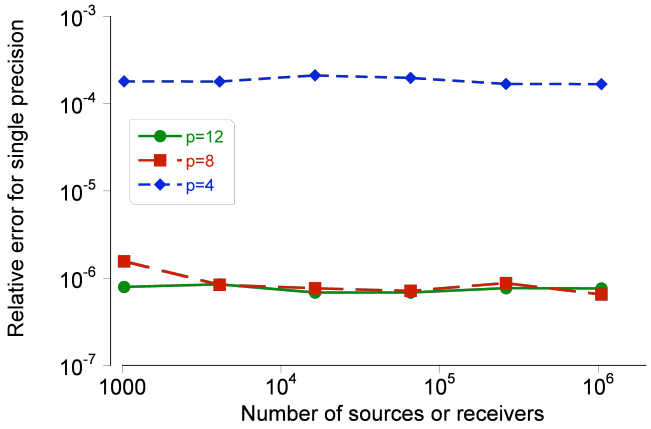


Fig. 14: Relative errors in single precision for the Coulomb kernel.

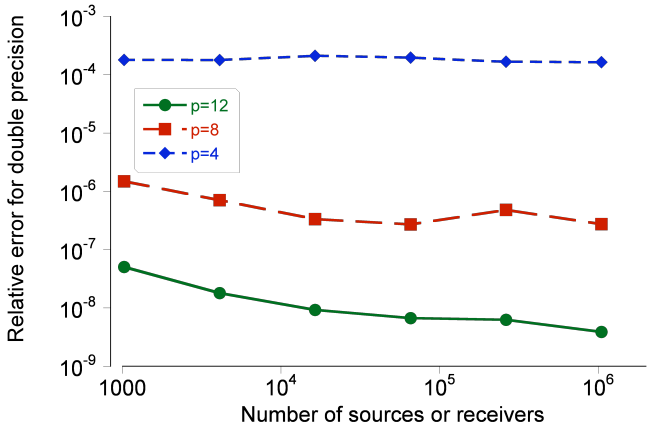


Fig. 15: Relative errors in double precision for the Coulomb kernel.

Figures 14 and 15 show the relative errors both on single and double precision for the Coulomb kernel. Because the single precision round-off errors are accumulated in the recursive calls, the extra $\mathcal{R}$ expansion coefficients obtained from p =8 to p =12, are no longer accurate. Moreover, the kernel evaluations within the neighborhood also introduce errors. Hence, the single precision case in Fig. 14 shows no accuracy improvement when the truncation is increased from 8 to 12. However, for double precision, the accuracy of can be improved by adding extra multipole expansion terms (by increasing p) which is shown in Fig. 15.

Smoothing Kernel and Dynamic Simulation

A big challenge in such simulations is the stability of the integration. In a large scale simulation, many particles are very close to each other, hence the roundoff errors of their distance are increased dramatically because of the kernel singularity, which makes the direct time step integration of the particle displacement unstable. For the Biot–Savart kernel, the vortices are dipole singularities, so the field grows as $1/|\mathbf{r}|^2$ near the source location. Based on the numerical experiments, even

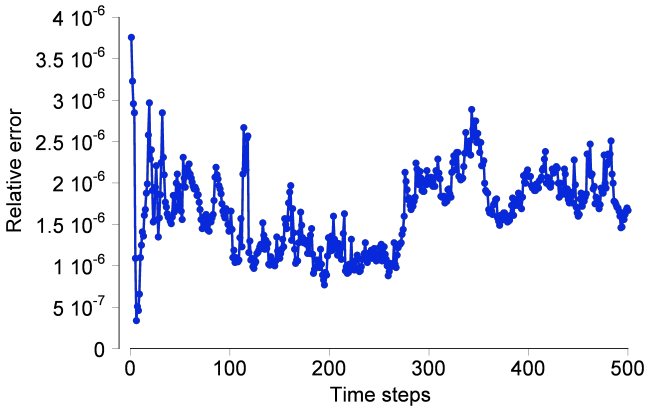


Fig. 16: Accuracy for each time step.

the direct CPU computation using double precision will diverge within several time steps on small size problems. An effective solution to this problem is to introduce *smoothing kernel* $K(d,\varepsilon)$ for the computation kernel as

$$V(\mathbf{y}) = \sum_{i=1}^{n} \frac{\mathbf{q}_i \times (\mathbf{y}-\mathbf{x}_i)}{|\mathbf{y}-\mathbf{x}_i|^3} K(|\mathbf{y}-\mathbf{x}_i|,\varepsilon), \tag{13}$$

where

$$K(d,\varepsilon) = \begin{cases} \dfrac{d^2}{c\varepsilon^2} & \text{if} \quad d \le \varepsilon, \\ 1 & \text{if} \quad d > \varepsilon. \end{cases} \tag{14}$$

for some constant c. Given the *minimal distance* d_{min} between source points and receiver points and the side length u of the box at the maximal level, the control threshold ε needs to satisfy $d_{min} \ll \varepsilon < u$. It guarantees that the error from the kernel singularity is cut off by enforcing $d_{min} \ll \varepsilon$ while it makes modifications of the FMM simple by setting $\varepsilon < u$; only the local kernel evaluations of the direct sum need to be modified. Other smoothing kernel functions can also be used, however, because the transcendental functions are implemented by a *special function unit* (SFU) (Ref. 13) on the NVIDIA GPUs, simple polynomial smoothing kernels are preferred for computational efficiency. Notice that viscous core models for vortices can be implemented in an analogous way.

As for the numerical integration, both Euler and the fourth order Runge–Kutta methods have been implemented. The Euler method with one FMM evaluation at each time step is fast and is less robust, while the Runge–Kutta method requires four FMM evaluations and is robust.

A simple test case of the accuracy of the FMM implementation for fluid dynamic problems has been studied where a convecting vortex ring comprising fluid particles has been simulated. In total, 16384 discretized ring elements and 32768 fluid particles were simulated. Because the CPU double precision data needed to be computed as the "exact" solution, it is not practical to use a large number of vortex elements. The relative errors were computed at each time step

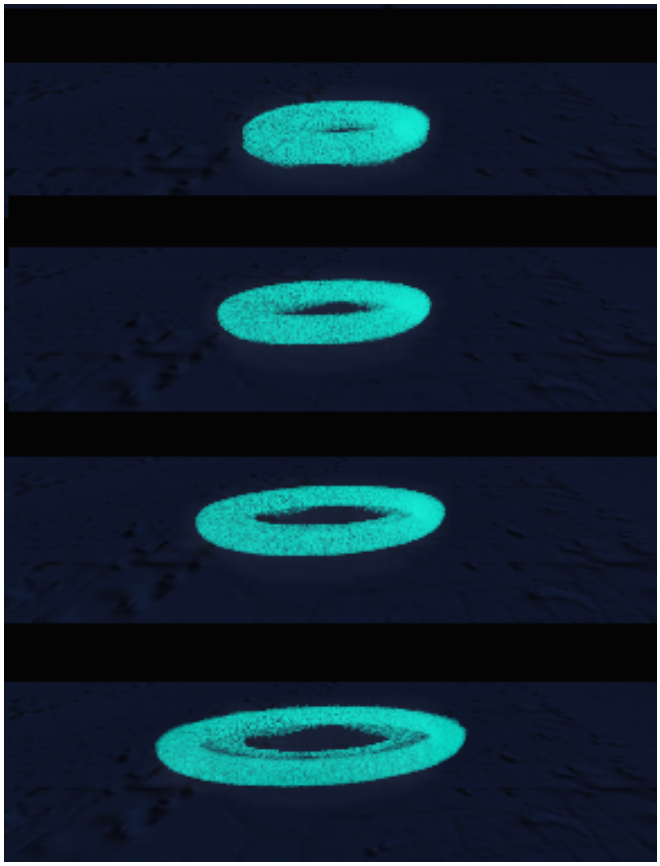


Fig. 17: The simulation snapshots of a vortex ring colliding with a ground plane at 4 different time steps.

by comparing the FMM GPU results with the CPU results. To test the errors, the whole simulation was run for 500 time steps with Euler integration and the results are summarized in Fig. 16. All of the computations used double precision.

In another test of the FMM implementation, a vortex ring colliding with a ground plane was simulated using double precision. This problem can be viewed as a highly simplified case of a rotor operating in ground effect where a vortical flow interacts with the flow at the wall, providing for the same essential features of radial vortex stretching and vortex/wall interaction that are produced below a rotor hovering in ground effect. The advantage of simulating this problem is that it produces a simplified but still highly representative test flow suitable for the evaluation of the FMM and GPU implementations, and the approach can also be extended to the two-phase flow environment.

In this case, 500000 fluid particles were computed with 32768 discretized vortex ring elements. The computational results were visualized by OpenGL, as shown in Fig. 17. For each time step, it was found to take around 1.2–2.5 seconds for the computation in which both FMM computation and data visualization share the same GPU hardware resource, compared to around 90 seconds on the CPU.

Concluding Remarks

By exploring the multi-core architecture of GPUs, the capability of GPU computations to give large performance gains has been demonstrated. This kind of hardware acceleration can also be applied to multi-GPUs and multi-CPU/GPU architectures by using OpenMP/MPI on large high performance clusters. The potential for the application of these methods to various rotorcraft aeromechanics simulations is enormous.

The FMM has complex data structures and translation schemes. It has been shown how to take advantage of GPU hardware by paralleling its computations on streaming multiprocessors and how it can achieve good speedup compared with other sequential implementations. Hence, based on the results shown, the FMM can be applied to problems of very large size on a single GPU equipped desktop, which currently can only be completed in practical times by using direct methods using expensive computer clusters.

The recent advances in both computer hardware and fast computational algorithms presented in this paper can be applied to various fluid mechanics and aeromechanics problems of large size. The novel GPU based FMM implementation developed here shows the best timing and error bounds compared to other published FMM implementations. The details of real expansions and translations and the adaption to Biot–Savart kernel are also gave high computational efficiency on the GPU. Successful simulations to long times with large numbers vortex elements and particles have also been demonstrated.

Acknowledgements

The Air Force Office of Scientific Research supported this work under a Multidisciplinary University Research Initiative, Grant W911NF0410176. The contract monitor was Dr. Douglas Smith. We would like to thank to NVIDIA for their equipment donation to the University of Maryland CUDA Center of Excellence and NSF award 0403313 for the computational clusters.

Appendix A: Auxiliary Arrays for FMM Data Structure

Let `numSrcNonEmptyBox` be the number of boxes containing at least one source data point in the maximal level and `numRecvNonEmptyBox` be the corresponding number for the receiver points. The functions of these arrays are described as the following:

1. `SrcBookMark[]`: its *i*th entry points to the first sorted source data point of the *i*th source non-empty box in `SortedSrc[]`. Its length is `numSrcNonEmptyBox` + 1.

2. `RecvBookMark[]`: its *j* entry points to the first sorted receiver data point of the *j*th receiver

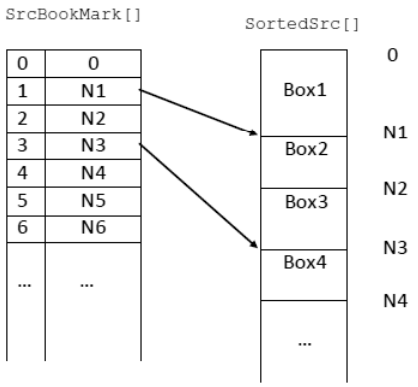


Figure A 1: Bookmark and sorted data points.

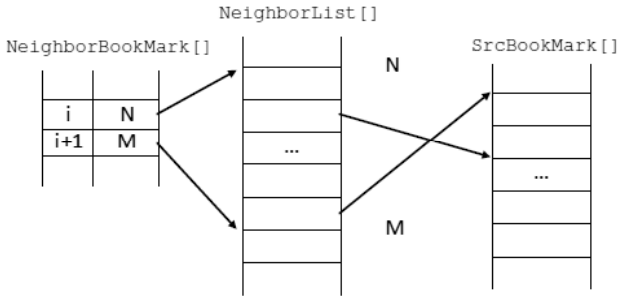


Figure A 2: Neighbor bookmark, neighbor list and source bookmark.

non-empty box in `SortedRecv[]`. Its length is `numRecvNonEmptyBox` + 1.

3. `SrcNonEmptyBoxIndex[]`: its i entry stores the index of the ith non-empty source box. Its length is `numSrcNonEmptyBox`.

4. `NeighborBookMark[]`: the function of its jth entry is to tell the thread which performs the nearby direct sum for the jth non-empty receiver box that values from `NeighborList[NeighborBookMark[j]]` to `NeighborList[NeighborBookMark[j+1]-1]` have to be retrieved.

5. `NeighborList[]`: for `NeighborBookMark[j]` $\leq i <$ `NeighborBookMark[j+1]`, let $k =$ $i-$`NeighborBookMark[j]`, then `NeighborList[i]` stores the index of the kth non-empty source box adjacent to the jth non-empty receiver box (E_2 neighborhood). Here the index means the rank of that non-empty source box in the `SrcNonEmptyBoxIndex[]` (Fig. 2).

6. `RecvPermutationIdx[]`: its ith entry means the original position of data point `SortedRecv[i]` in `Recv[]` is `RecvPermutationIdx[i]`.

Algorithm 2 Access source data points of E_2 neighborhood.

```
Given the i-th non-empty receiver box;
B=NeighborBookMark[i+1]-1
for j=NeighborBookMark[i] to B do
  reads v=NeighborList[j];
  C=SrcBookMark[v+1]-1
  for k=SrcBookMark[v] to C do
    reads SortedSrc[k]
  end for
end for
```

Given the bookmark array, the data point can be accessed directly from the sorted data list shown in Fig. 1. For each receiver non-empty box, the source data points within its E_2 neighborhood can be accessed using Algorithm 2. The bookmarks are only kept for non-empty boxes and the neighbor list is only kept for non-empty neighbors. No information of empty boxes are stored. However, `NeighborBookMark[]` and `NeighborList[]` can actually be replaced just by *scanned* `bin[]` mentioned before under the current implementation context. Although this paper targets at the GPU fast implementation, in the CPU-GPU hybrid implementation, the `sortIdx` is not allocated for all the boxes but only for the non-empty boxes. Using the adaptive FMM data structure, which are designed for non-uniform data distribution, the maximal level can be large such that just the allocated memory for `sortIdx` can consume all the available GPU global memory. Hence, in that case, `sortIdx` are only allocated for non-empty boxes while `NeighborBookMark[]` and `NeighborList[]` are constructed on CPU using multi-threading for to save memory. The last auxiliary array is `RecvPermutationIdx[]`, which is used to retrieve the input order of the original receiver data points.

Appendix B: Local Expansions Using Real Numbers

Let $\mathbf{r} = (r, \theta, \varphi)$ be a 3D vector, p be the truncation number, $B_n^m(\mathbf{r})$ be the complex basis function with coefficient c_n^m, and $\widetilde{B}_n^m(\mathbf{r})$ be the real basis function obtained from $B_n^m(\mathbf{r})$ with coefficient d_n^m. Then it is defined the same as (Ref. 8, (12)) by

$$\widetilde{B}_n^m(\mathbf{r}) = \begin{cases} \mathrm{Re}\{B_n^m\}, m \geq 0, \\ \mathrm{Im}\{B_n^m\}, m < 0. \end{cases} . \tag{A1}$$

It is already known that the basic kernel function

$$\Phi(\mathbf{r}) = \sum_{n=0}^{p} \sum_{m=-n}^{n} c_n^m B_n^m(\mathbf{r}) \tag{A2}$$

is real. If define $\Phi_n(\mathbf{r}) = \sum_{m=-n}^{n} c_n^m B_n^m(\mathbf{r})$, then by the conjugate property, $\Phi_n(\mathbf{r})$ is real, which implies

$$\Phi_n(\mathbf{r}) = \sum_{m=-n}^{n} c_n^m B_n^m(\mathbf{r}) = \sum_{m=-n}^{n} d_n^m \widetilde{B}_n^m(\mathbf{r}). \tag{A3}$$

From Eqs. A1 and A3, the coefficient relation between c_n^m and d_n^m can be derived as

$$d_n^{-m} = c_n^{-m} + c_n^m, \quad d_n^m = i(c_n^{-m} - c_n^m). \tag{A4}$$

The elementary solutions of the Laplace equations in 3D can be described as

$$R_n^m(\mathbf{r}) = \alpha_n^m r^n Y_n^m(\theta,\varphi), \;\; S_n^m(\mathbf{r}) = \beta_n^m r^{-n-1} Y_n^m(\theta,\varphi), \tag{A5}$$

where α_n^m, β_n^m are the normalization constants and $Y_n^m(\theta,\varphi)$ are the orthonormal spherical harmonics. To obtain the real representations, define the normalization constants as

$$\begin{aligned} \alpha_n^m &= (-1)^n \sqrt{4\pi/[(2n+1)(n-m)!(n+m)!]} \\ \beta_n^m &= \sqrt{4\pi(n-m)!(n+m)!/(2n+1)}, \end{aligned} \tag{A6}$$

then the following identity holds for Coulomb kernel in spherical coordinates system

$$\Phi(\mathbf{r},\mathbf{r}_*) = \frac{1}{|\mathbf{r}-\mathbf{r}_*|} = \sum_{n=0}^{+\infty} \sum_{m=-n}^{n} (-1)^n R_n^{-m}(\mathbf{r}_*) S_n^m(\mathbf{r}). \tag{A7}$$

Together with the local $\mathcal{R}$ expansions of receiver points in the final summation, Eq. A7 implies that the FMM only needs to compute $R_n^{-m}(\mathbf{r})$ for both source and receiver points. Now, shift to the truncated real number version of Eq. A7

$$\Phi(\mathbf{r},\mathbf{r}_*) = \sum_{n=0}^{p} \sum_{m=-n}^{n} (-1)^n d_n^{-m}(\mathbf{r}_*) \widetilde{S}_n^m(\mathbf{r}) + Err_t. \tag{A8}$$

Given Eq. A4, the following recurrence relations can be derived to compute real $\mathcal{R}$-expansions (multipole) $d_n^{-m}(\mathbf{r}_*)$:

$$\begin{aligned} & d_0^0 = 1, \;\; d_1^1 = -\frac{1}{2}x, \;\; d_1^{-1} = \frac{1}{2}y, \\ & d_{|m|}^{|m|} = -\frac{x d_{|m|-1}^{|m|-1} + y d_{|m|-1}^{-|m|+1}}{2|m|}, \;\; m = 2,3,\ldots, \\ & d_{|m|}^{-|m|} = \frac{y d_{|m|-1}^{|m|-1} - x d_{|m|-1}^{-|m|+1}}{2|m|}, \;\; m = 2,3,\ldots, \\ & d_{|m|+1}^{m} = -z d_{|m|}^{m}, \qquad\qquad m = 0,\pm1,\pm2,\ldots, \\ & d_n^m = -\frac{(2n-1) z d_{n-1}^m + r^2 d_{n-2}^m}{n^2-m^2}, \quad \begin{matrix} n = |m|+2, |m|+3,\ldots, \\ m = -n,\ldots,n. \end{matrix} \end{aligned} \tag{A9}$$

References

[1]Haehnel, R. B., Moulton, M. A., Wenren, W., and Steinhoff, J., "A Model to Simulate Rotorcraft-Induced Brownout," American Helicopter Society 64th Annual Forum Proceedings, Montréal, Canada, April 29–May 1, 2008.

[2]Phillips, C., and Brown, R., "Eulerian Simulation of the Fluid Dynamics of Helicopter Brownout," American Helicopter Society 64th Annual Forum Proceedings, Montréal, Canada, April 29–May 1, 2008.

[3]D'Andrea, A., "Numerical Analysis of Unsteady Vortical Flows Generated by a Rotorcraft Operating on Ground: A First Assessment of Helicopter Brownout," American Helicopter Society 65th Annual Forum Proceedings, Grapevine, TX, May 27–29 2009.

[4]Wachspress, D. A., Whitehouse, G. R., Keller, J. D., Yu, K., Gilmore, P., Dorsett, M., and McClure, K., "A High Fidelity Brownout Model for Real-time Flight Simulations and Trainers," American Helicopter Society 65th Annual Forum Proceedings, Grapevine, TX, May 27–29, 2009.

[5]Syal, M., Govindarajan, B., and Leishman, J. G., "Mesoscale Sediment Tracking Methodology to Analyze Brownout Cloud Developments," American Helicopter Society 66th Annual Forum Proceedings, Phoenix, AZ, May 10–13, 2010.

[6]Egolf, T. A., "Helicopter Free Wake Prediction of Complex Wake Structures Under Blade-Vortex Interaction Operating Conditions," American Helicopter Society 44th Annual Forum Proceedings, Washington, D.C., June 16–18, 1988.

[7]Nyland, L., Harris, M., and Prins, J., "Fast N-Body Simulation with CUDA," *GPU Gem 3*, Ed. Addison-Wesely, New York, NY, 2007.

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

[9]Stock, M. J., and Gharakhani, A., "Toward Efficient GPU-accelerated N-body Simulations," 46th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, January 7–10, 2008, pp. 608–621.

[10]Stock, M. J., Gharakhani, A., and Stone, C. P., "Modeling Rotor Wakes with a Hybrid Overflow-Vortex Method on a GPU Cluster," AIAA Applied Aerodynamics Conference, Chicago, IL, June 28–July 1, 2010.

[11]Stone, C. P., Hennes, C. C., and Duque, E. P. N., "A Hybrid CPU/GPU Parallel Algorithm for Coupled Eulerian and Vortex Particle Methods," *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, Rupak Biswas and NASA Ames Research Center, NASA Advanced Supercomputing Division, DEStech Publications, Inc.

[12]"China Grabs Supercomputing Leadership Spot in Latest Ranking of World's Top 500 Supercomputers," `http://www.top500.org/lists/2010/11/press-release`, November 11, 2010.

[13]NVIDIA, "NVIDIA CUDA C Programming Guide," Version 3.2, November 2010.

[14]NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009.

[15]Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kr´uger, J., Lefohn, A. E., and Purcell, T., "A Survey of General- Purpose Computation on Graphics Hardware," Computer Graphics Forum, Vol. 26, (1), March 2007, pp. 80–113.

[16]Davidson, A., and Owens, J. D., "Toward Techniques for Auto-Tuning GPU Algorithms," Para 2010: State of the Art in Scientific and Parallel Computing, June 2010.

[17]NVIDIA, "OpenCL Programming Guide for the CUDA Architecture," 3rd edition, 2010.

[18]Gumerov, N. A., Luo, Y., Duraiswami, R., DeSpain, K., Dorland, B., and Hu, Q., "FLAGON: Fortran-9X Library for GPU Numerics," GPU Technology Conference, NVIDIA Research Summit, San Jose, CA. Available at `http://sourceforge.net/projects/flagon/`, October 2009.

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

[20]Dongarra, J., and Sullivan, F., "The Top 10 Algorithms," *Computing in Science and Engineering*, Vol. 2, 2000, pp. 22–23.

[21]Samet, H., *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2005.

[22]Gumerov, N. A., and Duraiswami, R., *Fast Multipole Methods for the Helmholtz Equation in Three Dimensions*, Elsiever, Oxford, England, 2004.

[23]Epton, M. A., and Dembart, B., "Multipole Translation Theory for the Three-Dimensional Laplace and Helmholtz Equations," *SIAM Journal of Scientific Computing*, Vol. 16, July 1995, pp. 865–897.

[24]Gumerov, N. A., and Duraiswami, R., "Comparison of the Efficiency of Translation Operators Used in the Fast Multipole Method for the 3D Laplace Equation," Technical Report CS-TR-4701, UMIACS-TR-2005-09, University of Maryland Department of Computer Science and Institute for Advanced Computer Studies, April 2005.

[25]Gumerov, N. A., Duraiswami, R., and Y.A., B., "Data Structures, Optimal Choice of Parameters, and Complexity Results for Generalized Multilevel Fast Multipole Methods in d-Dimensions," Technical Report CS-TR-4458; UMIACSTR-2003-28, University of Maryland Department of Computer Science and Institute for Advanced Computer Studies, April 2003.

[26]Harris, M., Sengupta, S., and Owens, J. D., "Parallel Prefix Sum (Scan) with CUDA," *GPU Gems 3*, Addison Wesley, New York, NY, August 2007, pp. 851–876.

[27]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, 2009, pp. 2066–2078.

[28]White, C. A., and Head-Gordon, M., "Rotating Around the Quartic Angular Momentum Marrier in Fast Multipole Method Calculations," *The Journal of Chemical Physics*, Vol. 105, September 1996, pp. 5061–5067.

[29]Batchelor, G. K., *An Introduction to Fluid Dynamics*, Cambridge University Press, New York, NY, February 2000.