GPU Accelerated Fast Multipole Methods for Dynamic N-body Simulation

Qi Hu[a,b,1], Nail A. Gumerov[b,c], Ramani Duraiswami[a,b,c]

[a] *Department of Computer Science, University of Maryland, College Park*

[b] *University of Maryland Institute for Advanced Computer Studies (UMIACS)*

[c] *Fantalgo LLC, Elkridge, MD*

Abstract

Many physics based simulations can be efficiently and accurately performed using particle methods which focus computational resources at the location of sources or discontinuities (particles), and evaluation of relevant fields at locations of interest. These particle methods result in the so-called N-body problem. The N body problem also arises in interpolation using implicit functions, in simulation of molecular and stellar dynamics, and other areas. Fast and accurate N body simulations are the goal of this paper. The Fast Multipole Method (FMM) has been proposed for these. In this paper we provide efficient data-structures implemented on Graphical Processing Units (GPUs), and a novel parallel formulation of the FMM on GPUs to address this problem. As an example application, we simulate the interactions between vortex rings. Except for initial setup, our approach processes all the computations and updates on GPU. Further, we provide interactive visualization of the simulation as it proceeds. Where the

[1] please contact the corresponding author via email: `huqi@cs.umd.edu` or phone:`+13014051207` (fax: `+13013149658`)

Preprint submitted to Elsevier *May 1, 2013*

cost of direct simulation of the interaction of vortices and particles is $O(n^2 + nm)$ per time step, where n is number of vortex elements and m is the number of particles, our algorithm reduces it to O(n+m) cost.
Keywords: FMM, GPGPU, N-body Simulation, Vortex Ring

---

1. Introduction

The fast multipole method can be used to accelerate N-body simulations and matrix vector products arising in various applications. These include fluid simulations [1, 2] as well as in scientific computing; in fitting implicit functions to point based representations using radial-basis functions[3, 4]; in radiosity computations [5] and in computing the dynamics of attracting and repelling bodies such as those arising in molecular or stellar dynamics. In fluid simulation, compared to methods that use meshes which result in large discretizations, particle methods are extremely efficient. Despite this, large numbers of particles may be necessary for fidelity.

Although particle methods avoid large mesh discretizations, the interactions among particles appear for all pairs, which makes the computational complexity quadratic. Because of such $O(n^2)$ cost given n particles, simulations on large scale problems can not be completed within practical time. Generally speaking, without distributed systems such as high performance clusters, the direct method can only work for the problem size in the order of 10^4 on high end workstations. By parallelizing the computations on the multi-core architecture, [6] developed a fast GPU-based parallel implementation, however, its computation complexity is still in quadratic. Recent work on particle methods using direct computation has

been shown in [7].

An alternative way to solve such n-body problems based on particle methods is to use *fast algorithms*. For example, the Fast Multipole Method (FMM) only exactly computes near-field interactions but approximates far-field interactions to a specified tolerance ϵ in order to reduce the computation cost.

While our method can work for all applications of the FMM, for specificity, we will consider the case of the simulation of the dynamics of vortex rings. Readers may be familiar with the blowing of smoke rings by smokers. In these smoke rings the lips blow a vortex ring, which traps the smoke particles, which help its visualization. Particularly adept smokers can blow successions of vortex rings, which then may exhibit behaviors such as leap-frogging (a later ring accelerates through a previous ring).

To simulate interactions between vortices and particles, we apply particle methods with the FMM. In the FMM, the split between the near and far-fields must be managed by grouping nearby particles using the well-separated pair decomposition [8], which requires appropriate spatial data structures. In [9, 10, 11], different GPU-based FMM implementations were developed. Particularly, Ref. [12] compared the performances between treecode and FMM on GPUs for a similar leapfrogging vortex ring simulations. However, in these implementations, the data structures were built on the CPU, which is too expensive for dynamic problems, where particle locations change every step. Ref. [13] developed a CPU-GPU-Hybrid *treecode* to accelerate the computation, but its overall performance does not outperform the implementation presented in [9]. Recently, Ref. [14] achieved the state of the art performance by developing a hybrid FMM algorithm on the

heterogeneous architectures, only for the scalar Coulomb kernel, while Ref. [15] discussed the auto-tuning techniques of N-body simulations on heterogeneous systems.

In this paper, both parallel FMM and the data structure on the GPU are developed to solve the dynamic *n-body* problem with on-the-fly rendering. Although it is applied in the context of fluid flow, such fast FMM parallel implementation can also be used for molecular dynamics, stellar dynamics and RBF interpolation [3] [4]. The FMM translations and expansions we use employ real number representations as opposed to the usual complex spherical harmonic based representation. This allows for GPU computation efficiency. Best to our known, this is the first paper in which the entire vortex method using FMM with run time result visualizations is running on a single GPU.

1.1. Introduction to GPU

There is a revolution underway over the past decade or so in the use of graphics inspired hardware for accelerating general purpose computations. These accelerators allow access to tremendous computational power and have favorable energy consumption. The use of such accelerators, which started with general purpose GPU (GPGPU) for computing is getting more and more popular and well-accepted by the high performance computing community.

Graphical processing unit (GPU) is a highly parallel, multithreaded, many-core processor, developed originally for graphical rendering, but because of their high computation power and memory bandwidth, used extensively in all sorts of simulations. These processors achieve single instruction multiple data (SIMD)

computation with more transistors devoted to data processing rather than to data caching and flow control. Modern GPUs are capable of both single and double precision computations up-to Tera-FLOPs on a single accelerator. Since GPUs are attached to the host (CPU) via PCI-Express bus, processing data on those accelerators requires data transfer between host and device (GPU) back and forth. The on-chip device memory are hierarchical. In the current NVIDIA Fermi architecture [16], on which the present simulations were performed, there are four kinds of memories:

1. *Global memory*: Device DRAM memory with slow access but large size. This is used to keep data and communicate with the host main memory.
2. *Constant memory*: 64KB read only constant cache shared by all the threads, used mainly to store constant values.
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 used mainly used to hold instructions and values.

One main GPU programming challenge is to efficiently use these hierarchical memories in the threaded model given the trade-off between access speed and size [17, 18]. Programming on the GPU remained a technical barrier before 2006 because it required a deep knowledge of computer graphics. In 2006, NVIDIA released a general-purpose parallel computing architecture called CUDA so that

a programmer can more easily manipulate and 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;
2. Different kinds of memory;
3. Synchronization mechanisms.

OpenCL (Open Computing Language) is another way to program GPUs [19] 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. The present paper restricts itself to CUDA. These results should also extend to OpenCL. Refer to [20, 21] for the state of the art in GPGPU applications.

1.2. Interactions between vortices and particles

Our target application is to simulate the intensive interactions among vortex elements and fluid governed by the so-called Biot-Savart law. Given N vortex blobs of strength $\boldsymbol{\omega}_i,\ i = 1, \ldots, N$ located at $\mathbf{x}_i$ moving with the flow, the velocity field can be evaluated by

$$\mathbf{v}(\mathbf{y}) = \sum_{i=1}^{N} \mathbf{v}_i(\mathbf{y}), \quad \mathbf{v}_i(\mathbf{y}) = \frac{\boldsymbol{\omega}_i \times (\mathbf{y} - \mathbf{x}_i)}{|\mathbf{y} - \mathbf{x}_i|^3} = \nabla \times \frac{\boldsymbol{\omega}_i}{|\mathbf{y} - \mathbf{x}_i|}. \tag{1}$$

While the vortex elements move with flow, vortex field also evolves according to the vortex evolution equation. For inviscid flow, it can be described as

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}|_{\mathbf{x}=\mathbf{x}_i}, \quad \frac{d\boldsymbol{\omega}_i}{dt} = \boldsymbol{\omega}_i \cdot \nabla \mathbf{v}|_{\mathbf{v}=\mathbf{v}_i}, \quad \mathbf{v}(\mathbf{x}_i; t) = \sum_{j \neq i} \mathbf{v}_j(\mathbf{x}_i; t). \tag{2}$$

Here the right hand side for the vortex strength is the so-called vortex stretching term and requires the evaluations of the gradient of the velocity vector. The velocity field in Eq. 1 can also be modified by using a smoothing kernel $K(|\mathbf{y} - \mathbf{x}_i; a)$ as

$$\mathbf{v}_i(\mathbf{y}; a) = \frac{\boldsymbol{\omega}_i \times (\mathbf{y} - \mathbf{x}_i)}{|\mathbf{y} - \mathbf{x}_i|^3} K(|\mathbf{y} - \mathbf{x}_i|; a), \tag{3}$$

where a is the radius of the vortex core and the smoothing kernel only has effect in the near field of $\mathbf{x}_i$. Given n vortex elements and m fluid particles, we obtain an *n-body* problem to update all their space positions, and the total computation cost is in $O(n^2 + nm)$.

Choices of the smoothing kernels is well-discussed in the vortex element literature e.g. in [22]. Core-spreading algorithms and re-meshing techniques as well as approximations of the viscous term dropped in Eq. 2 for evolution of $\boldsymbol{\omega}_i$ are discussed as well. The focus of present paper is development of fast summation procedure for elementary velocity fields in Eq. 3 with arbitrary kernels $K()$ decaying fast enough in the far field. So, particular applications can use the method described below, while physical modeling requires combination of the present technique with standard vortex-element techniques mentioned above.

1.3. Introduction to Fast Multipole Method (FMM)

Fast multipole methods have been identified as one of the ten most important algorithmic contributions in the 20th century [23]. Its theory has already been well developed. Our baseline FMM algorithm computes the Coulomb potentials generated by source points $\{\mathbf{x}_i\}$ at receiver points $\{\mathbf{y}_j\}$ as

$$\phi(\mathbf{y}_j) = \sum_{i=1}^{n} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i), \quad j = 1, 2, \ldots, m, \quad \mathbf{x}_i, \mathbf{y}_j \in \mathbb{R}^3, \tag{4}$$

where q_i is the strengths. Here the kernel function Φ

$$\Phi(\mathbf{y}, \mathbf{x}) = \begin{cases} \frac{1}{|\mathbf{y} - \mathbf{x}|}, & \text{if } \mathbf{x} \neq \mathbf{y}, \\ 0, & \text{if } \mathbf{x} = \mathbf{y}. \end{cases} \tag{5}$$

The main idea in the FMM is to split the Eq. 4 into near and far fields

$$\phi(\mathbf{y}_j) = \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i) + \sum_{\mathbf{x}_i \in \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i), \tag{6}$$

for $j = 1, 2, \ldots, m$ where $\Omega(\mathbf{y}_j)$ is the neighborhood domain, then build factored approximate representations of the functions in the far-field which usually come from analytical series representations, and are truncated at some number of coefficients p, which is a function of the specified tolerance $\epsilon = \epsilon(p)$. The geometric structure that encodes much of these FMM data information, such as grouping points and finding neighbors, is called a *well-separated pair decomposition* (WSPD, see Fig. A.2), which is itself useful for solving a number of geometric problems [8, chapter 2]. Assume all data points are already scaled into an unit cube. The WSPD is recursively performed to subdivide this cube into sub-cubes via *octree* until the maximal level l_{max} is achieved; l_{max} is chosen such that the computation costs of the local direct sum and far field translations can be balanced.

For convenience of presentation, we call a box containing at least one source point a *source box* and a box containing at least one receiver point a *receiver box*. The FMM algorithm can be summarized as follows:

1. **Initial expansion (P2M)**:
   (a) At the finest level l_{max}, all sources are expanded at their box centers to obtain the far-field $\mathcal{M}$ expansion coefficients $\{\mathcal{C}_n^m\}$ over p^2 spherical basis functions.

(b) The obtained $\mathcal{M}$-expansion from all sources in the same boxes are consolidated into a single expansion.

2. **Upward pass (M2M)**: For levels from l_{max} to 2, the $\mathcal{M}$ expansion coefficients for each box are translated via multipole-to-multipole ($\mathcal{M}|\mathcal{M}$) translations from the source box centers to their parent source box center. All these translations are performed in a hierarchical order from bottom to top via the octree.

3. **Downward pass**: For levels from 2 to l_{max}, each receiver box generates its local or $\mathcal{L}$ expansion in a hierarchical order from top to bottom via the octree.

   (a) M2L: Translate multipole $\mathcal{M}$ expansion coefficients from source boxes at the same level in the receiver box's parent neighborhood, but not the neighborhood of that receiver itself, to a local $\mathcal{L}$ expansion via multipole-to-local ($\mathcal{M}|\mathcal{L}$) translations, then consolidate the expansion coefficients.

   (b) L2L: Translate the $\mathcal{L}$ expansion coefficients (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 (L2P**: Evaluate the $\mathcal{L}$ expansion coefficients 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.

The translation theory and other algorithmic details can be found in [24, 25, 26].

2. GPU-based Fast Multipole Method

2.1. FMM Data Structure

Efficient FMM algorithm requires both fast data structure construction and low data addressing latency. In our implementation, both translations and local direct sums have their spatial *interaction lists* used to address data directly. Therefore, the FMM algorithm requires the following special data structures:

1. Octree to ensure WSPD that ensures error bounds.
2. Interaction lists for fast data addressing.
3. Communication management structures.

The construction of these data structures must be done via algorithms that have the same overall complexity with the summation. The typical way of computing these data structures is via an $O(N \log N)$ algorithm, which is built upon spatial data sorting and is sequentially implemented on the CPU [9]. Reimplementing this CPU algorithm for the GPU would not have achieved the kind of acceleration we sought since the conventional FMM data structures algorithm employs sorting of large data sets and operations such as set intersection on smaller subsets, that require random access to the global GPU memory, which is not very efficient. Previous fast Kd-tree and octree data structures work [27, 28] look similar to the spatial data structures used here, however, they lack the functionality to construct these interaction lists for the specific neighbor and box query operations, hence cannot be directly applied in to the FMM framework. Other work on the distributed FMM algorithm often does not provide details of the data structures used, or their construction algorithms.

Hence, the first goal is to design a new parallelizable algorithm, which generates the FMM data structure in $O(N)$ time, bringing the overall complexity of the FMM to $O(N)$ for a given accuracy. Our algorithm is based on use of occupancy histograms (i.e., the counts of particles in all boxes), pseudo-sort using Fixed-Grid-Method, and parallel scans [29, 30]. A potential disadvantage of our approach is the fact that the histogram requires allocation of an array of size $8^{l_{\max}}$ where zeros indicate empty boxes. Nonetheless this algorithm for GPUs with 4 GB global memory enables of data structures up to a maximum level $l_{\max} = 8$, which is sufficient for many problems. For problems that required greater octree depth, we developed a distributed multi-GPU version of the algorithm, which is out of the scope of this paper. Note that we use a prescribed cluster size to determine the best value of l_{max} for different problem sizes. More exactly, this cluster size is obtained for a GPU with particular hardware specifications (refer to [9] for extensive discussions) and limits the maximal number of particles at each spatial octree boxes at the maximal level.

- *Determine the Morton index [8] for each particle*, after scaling all data to a unit cube using bit-interleaving (e.g. [31, 32]). On the GPU this does not require communication between threads.

- *Construction of occupancy histogram and pseudo-sort using Fixed-Grid-Method.* The histogram shows how many particles reside in each spatial box at the finest level. In this step, the box index is its Morton index. Bin sorting occurs simultaneously with the histogram construction. Note that there is no need to sort the particles inside the box — our pseudo-sorting just results in an arbitrary local rank for each particle in a given box.

Algorithm 1 PARALLEL-PSEUDO-SORT(`P[], M`): an algorithm to compute the sorted index of each particle using the Fixed-Grid-Method.

Input: a particle position array `P[]` with length `M`

Output: a 2D index array `sortIdx[]`

```
1: for i=0 to M-1 parallel do
2:     SortIdx[i].x←BoxIndex(P[i])
3:     atomicAdd[Bin[SortIdx[i].x]]
4:     SortIdx[i].y←Bin[SortIdx[i].x]
```

- *Parallel scan and global particle ranking.* Parallel scan is an efficient algorithm that provides a pointer to the particles in a given box in the final array. Particle global ranking is simply a sum of its global bookmark and local arbitrary rank.
- *Final filtering.* This process simply removes entries for empty boxes and compresses the array, again using a scan, so the empty boxes are emitted in the final array.
- *Final bin sorting.* Particle data is placed into the output array according to their global ranking.

We show how this linear pseudo-sorting works as follow: assume the term "data points" to refer to both, and denote the array storing these data points by `P[]`. Firstly, each data point `P[i]` has associated with a 2D vector called `sortIdx[i]`, where `sortIdx[i].x` stores the Morton index of its box and `sortIdx[i].y` stores its rank within the box. Secondly, the *histogram* array `Bin[]` is allocated for the boxes at the maximal level. Its ith entry `Bin[i]` stores the number of data points within the box i, which is computed by the

atomicAdd() function in the GPU implementation. This CUDA function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory [18]. Let the number of data points be M. Then the pseudocode to compute sortIdx[] and Bin[] is given in Alg. 1. Although atomicAdd() serializes those threads that access the same memory address, the parallel performance of our implementation is good on average. This is because most threads work on different memory locations at the same time. Using sortIdx and Bin with parallel scan, all the data points can be re-arrange according to their Morton indices.

The second part of the algorithm determines the interacting source boxes in the neighborhood of the receiver boxes. The histogram for the receivers can be deallocated while retaining the one for sources. We also keep the array A of source boxes obtained after the parallel scan (before compression). This enables fast neighbor determination **without sort, search, or set intersection operations.**

For a given receiver box i, its Morton index n is available as the ith entry of the array ReceiverBoxList. This index allows one to determine the Morton indices of its spatial neighbors. As a new neighbor index is generated, the occupancy map is checked. If the box is not empty, the corresponding entry in array A provides its global rank, which is stored as the index of the neighbor box. Computation of the parent neighborhoods and subdivision of the domains for translation stencils, which require a more complex data access pattern, is performed on the CPU, which creates the arrays ReceiverBoxList, SourceBoxList, NeighborBoxList and bookmarks (values indicating the starting and ending values of the particle number in a box). Refer to [33] for the details of interaction lists and their parallel constructing algorithms.

2.2. Real Representation

Although the FMM expansions and translations in the literature use complex valued spherical harmonic representations, this can result in extra costs and the use of special functions that use complex arguments. A real number version of these expansions and translations can be derived by using their symmetry properties. A big advantage of the real number representations is that GPU can process these real numbers much more efficiently. In the following discussion, we will use both spherical coordinates (r, θ, φ) and Cartesian coordinates (x, y, z) to establish real FMM expansions and translations. Let $\mathbf{r} = (r, \theta, \varphi)$, p be the truncation number and $B_n^m(\mathbf{r})$ be the complex basis function with coefficient c_n^m. Then $\widetilde{B}_n^m(\mathbf{r})$, the real basis function obtained from $B_n^m(\mathbf{r})$ with coefficient d_n^m, is defined (see [9, (12)]) by

$$\widetilde{B}_n^m(\mathbf{r}) = \begin{cases} \mathrm{Re}\{B_n^m\}, m \geq 0, \\ \mathrm{Im}\{B_n^m\}, m < 0. \end{cases} \tag{7}$$

It is already known that the basic kernel function

$$\Phi(\mathbf{r}) = \sum_{n=0}^{p} \sum_{m=-n}^{n} c_n^m B_n^m(\mathbf{r}) \tag{8}$$

is real. 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

$$\Phi_n(\mathbf{r}) = \sum_{m=-n}^{n} c_n^m B_n^m(\mathbf{r}) = \widetilde{\Phi}_n(\mathbf{r}) = \sum_{m=-n}^{n} d_n^m \widetilde{B}_n^m(\mathbf{r}). \tag{9}$$

From (5) and (7), the relation between c_n^m and d_n^m is

$$d_n^{-m} = c_n^{-m} + c_n^m, \quad d_n^m = \mathbf{i}(c_n^{-m} - c_n^m). \tag{10}$$

The elementary solutions of the Laplace equations in 3D are

$$R_n^m(\mathbf{r}) = \alpha_n^m r^n Y_n^m(\theta,\varphi), \quad S_n^m(\mathbf{r}) = \beta_n^m r^{-n-1} Y_n^m(\theta,\varphi), \tag{11}$$

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

$$\begin{aligned} \alpha_n^m &= (-1)^n \sqrt{4\pi/[(2n+1)(n-m)!(n+m)!]} \\ \beta_n^m &= \sqrt{4\pi(n-m)!(n+m)!/(2n+1)}, \end{aligned} \tag{12}$$

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

$$\Phi(\mathbf{r},\mathbf{r}_*) = \frac{1}{|\mathbf{r}-\mathbf{r}_*|} = \sum_{n=0}^{+\infty}\sum_{m=-n}^{n} (-1)^n R_n^{-m}(\mathbf{r}_*) S_n^m(\mathbf{r}). \tag{13}$$

Together with the local $\mathcal{R}$ expansions of receiver points in the final summation, (11) 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 (11)

$$\Phi(\mathbf{r},\mathbf{r}_*) = \sum_{n=0}^{p}\sum_{m=-n}^{n} (-1)^n d_n^{-m}(\mathbf{r}_*) \widetilde{S}_n^m(\mathbf{r}) + Err_t. \tag{14}$$

Given (8), the following recurrence relations can be derived to compute real $\mathcal{R}$-expansions (multipole) $d_n^{-m}(\mathbf{r}_*)$:

$$\begin{aligned}
&d_0^0 = 1, \quad d_1^1 = -\frac{1}{2}x, \quad d_1^{-1} = \frac{1}{2}y, \\
&d_{|m|}^{|m|} = -\frac{x d_{|m|-1}^{|m|-1} + y d_{|m|-1}^{-|m|+1}}{2|m|}, \quad m = 2,3,\ldots, \\
&d_{|m|}^{-|m|} = \frac{y d_{|m|-1}^{|m|-1} - x d_{|m|-1}^{-|m|+1}}{2|m|}, \quad m = 2,3,\ldots, \\
&d_{|m|+1}^{m} = -z d_{|m|}^{m}, \qquad m = 0,\pm 1,\pm 2,\ldots, \\
&d_n^m = -\frac{(2n-1) z d_{n-1}^m + r^2 d_{n-2}^m}{n^2 - m^2}, \quad n = |m|+2, |m|+3, \ldots, \; m = -n,\ldots,n.
\end{aligned} \tag{15}$$

Our implementation uses the $o(p^3)$ RCR decomposition [34] (Fig. A.3)to perform the $\mathcal{S}|\mathcal{S}$, $\mathcal{S}|\mathcal{R}$ and $\mathcal{R}|\mathcal{R}$ translations. Details of translation formula can be found in [25]. To move to real numbers, one actually only needs to provide modifications to α*-rotation* and β*-rotation* and an sign change for the *coaxial translation*. To keep the paper concise, we show the result. Let $\widehat{d_n^m}$ be the transformed real coefficients of d_n^m after rotation, then the α-rotation can be computed by

$$\begin{aligned} \widehat{d_n^{-m}} &= \sin(m\alpha)d_n^m + \cos(m\alpha)d_n^{-m}, \;\; m = 1, \ldots, n. \\ \widehat{d_n^m} &= \cos(m\alpha)d_n^m - \sin(m\alpha)d_n^{-m}, \;\; m = 1, \ldots, n. \end{aligned} \tag{16}$$

For β-rotation, let

$$\begin{aligned} f_n^m &= 1/2\sqrt{(n-m)(n+m+1)}, \;\; m = 0, 1, \ldots, n. \\ f_n^{-m} &= 1/2\sqrt{(n+m)(n-m+1)}, \;\; m = 1, \ldots, n. \end{aligned} \tag{17}$$

$$H_n^{m',0}(\beta) = (-1)^{m'}\sqrt{\frac{(n-|m'|)!}{(n+|m'|)!}}\mathbf{P}_n^{|m'|}(\cos\beta) \tag{18}$$

where $n = 0, 1, \ldots$, $m' = -n, \ldots, n$ and $\mathbf{P}_n^m(\mu)$ are the associated Legendre functions. $H_n^{m,m'}(\beta)$ satisfies the following relation:

$$f_n^{m-1}H_n^{m-1,m'} - f_n^m H_n^{m+1,m'} = f_n^{m'-1}H_n^{m,m'-1} - f_n^{m'}H_n^{m,m'+1}. \tag{19}$$

Then, the β rotation can be computed by:

$$\begin{aligned} \widehat{d_n^{-m}} &= \sum_{m'=1}^{n} d_n^{-m'}(H_n^{-m,-m'} - H_n^{-m,m'}), \;\; m = 1, \ldots, n. \\ \widehat{d_n^m} &= \sum_{m'=0}^{n} d_n^{m'}(H_n^{m,m'} + H_n^{m,-m'}), \;\; m = 1, \ldots, n. \\ \widehat{d_n^0} &= \frac{1}{2}\sum_{m'=0}^{n} d_n^{m'}(H_n^{0,-m'} + H_n^{0,m'}). \end{aligned} \tag{20}$$

Finally, for the *coaxial translation* [25, (27)], the only modification is to change the sign for different m as

$$\widehat{d}_n^m = (-1)^m \sum_{n'=|m|}^{n} (S|R)_{n,n'}^m(t) d_{n'}^m. \tag{21}$$

Using (13), (14), (18) and (19), all the FMM expansions and translations can be performed in real arithmetic with fast implementations on GPU.

2.3. Adaption to the Biot-Savart 3D Kernel

The baseline FMM computes the Coulomb kernel defined by Eq. 5, so the possibly minimal modifications are preferred to adapt to the Biot-Savart kernel by Eq. 1 based on the baseline codes. Notice that

$$\begin{aligned} \nabla_{\mathbf{y}} \times \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|} &= (\nabla_{\mathbf{y}} \frac{1}{|\mathbf{y}-\mathbf{x}_i|}) \times \mathbf{q}_i \\ &= -(\frac{\mathbf{y}-\mathbf{x}_i}{|\mathbf{y}-\mathbf{x}_i|^3}) \times \mathbf{q}_i \\ &= \frac{\mathbf{q}_i \times (\mathbf{y}-\mathbf{x}_i)}{|\mathbf{y}-\mathbf{x}_i|^3}. \end{aligned} \tag{22}$$

So rewrite (1) using (20) as

$$V(\mathbf{y}) = \sum_{i=1}^{n} \frac{\mathbf{q}_i \times (\mathbf{y}-\mathbf{x}_i)}{|\mathbf{y}-\mathbf{x}_i|^3} = \sum_{i=1}^{n} \nabla_{\mathbf{y}} \times \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|}. \tag{23}$$

Based on formula (21), apply the baseline FMM 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:

$$
\begin{aligned}
\{d_n^{(x),m}\}: \quad \sum_{i \notin \Omega_{\mathbf{c}}} \frac{q_i^{(x)}}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} d_n^{(x),m} R_n^m(\mathbf{y}-\mathbf{y_c}), \\
\{d_n^{(y),m}\}: \quad \sum_{i \notin \Omega_{\mathbf{c}}} \frac{q_i^{(y)}}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} d_n^{(y),m} R_n^m(\mathbf{y}-\mathbf{y_c}), \\
\{d_n^{(z),m}\}: \quad \sum_{i \notin \Omega_{\mathbf{c}}} \frac{q_i^{(z)}}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} d_n^{(z),m} R_n^m(\mathbf{y}-\mathbf{y_c}),
\end{aligned}
\tag{24}
$$

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

$$
\{\mathbf{d}_n^m\}: \sum_{i \notin \Omega_{\mathbf{c}}} \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|} = \sum_{n=0}^{p} \sum_{m=-n}^{n} \mathbf{d}_n^m R_n^m(\mathbf{y}-\mathbf{y_c}) \tag{25}
$$

Therefore,

$$
\begin{aligned}
\nabla_{\mathbf{y}} \times \sum_{i \notin \Omega_{\mathbf{c}}} \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} \nabla_{\mathbf{y}} \times [\mathbf{d}_n^m R_n^m(\mathbf{y}-\mathbf{y_c})] \\
&= \sum_{n=0}^{p} \sum_{m=-n}^{n} \nabla_{\mathbf{y}} R_n^m(\mathbf{y}-\mathbf{y_c}) \times \mathbf{d}_n^m.
\end{aligned}
\tag{26}
$$

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 $R_n^m(\mathbf{y}-\mathbf{y_c})$ needs to be computed according to (26). However, by differentiating Eq. 28 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) during the global memory access time such that the extra computation can be performed for no cost.

2.4. Test and Error Analysis

To test the performance of data structure construction and our GPU FMM implementation, data of different sizes up to 10 million are 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.

In the data structure comparison experiment (single precision), Table A.1 shows the time for data structures generation using a NVIDIA GTX480 and CPU Intel Xeon X5560 quad-core 2.8 GHz (a single core was used) for $N = 2^{20}$ source and $M = 2^{20}$ receiver points uniformly randomly distributed inside a cube. The octree depth was varied in the range $l_{\max} = 3, ..., 8$. Column 2 shows the wall clock time for a standard algorithm, which uses sorting and hierarchical neighbor search using set intersection (the neighbors were found in the parent neighborhood domain subdivided to the children level). Column 3 shows the wall clock time for the present algorithm on the CPU. It is seen that our algorithm is several times faster. Comparison of the GPU and CPU times for the same algorithm show further acceleration in the range 20-100. As a consequence, the data-structure step is reduced to a small part of the computation time.

As mentioned in Sec. 2.3, for the Biot-Savart kernel, three baseline FMM calls are integrated into one call to use the similarities of those recursion coefficients. A big advantage of this implementation is that extra computation costs can be hidden

from expensive GPU global memory access. In the downward-pass translation steps, both the indices and the processing order of E_4 neighbors for each receiver box are quite different among active threads. Therefore, it is impossible to make the access to translation data coalesced for threads in the same warp, which results in much reduced data fetching time. However, combining three calls into one call reduces three memory accesses to one. Even though the data fetched is the same, the total access time is reduced. Our experiments (see Table A.2) show that the full FMM computation time of Biot-Savart kernel is not tripled but less than doubled compared with the baseline FMM. The profiling of all parts of FMM for Biot-Savart kernel are also provided in Fig. A.4. Note that for the small number of data points, only one level of expansions and translations are performed.

Another experiment is performed to compare with normal direct methods on both single and double precision. The CPU implementation is double precision. The GPU direct method implementation is also optimized and the test results for Coulomb kernel are summarized in Fig. A.5. The GPU-based FMM shows the linear computation cost for large number of data points, in which case the overhead and latency can be neglected, while the direct methods on both CPU and GPU show the quadratic cost. The cross-over point, where GPU based FMM outperform other implementations using direct method, is at $N = M = 65536$. As for the Biot-Savart kernel, its GPU implementation has the similar performance, in which the evaluations of 10 million particle interactions takes about 7 and 16 seconds for the single and double precision respectively.

The error introduced by FMM is determined by the truncation number p. Theory on FMM error analysis can be found in [26]. In this experiment, we will validate our GPU implementation satisfy the accuracy requirement controlled by

the truncation number. The relative error is defined as

$$\epsilon = \sqrt{\frac{\sum_{j=1}^{k} |\phi_{exact}(\mathbf{y}_j) - \phi_{approx}(\mathbf{y}_j)|^2}{\sum_{j=1}^{k} |\phi_{exact}(\mathbf{y}_j)|^2}} \tag{27}$$

are computed by picking $k = 100$ testing points for each test cases. The *exact* values to measure the FMM error are computed by the direct method on CPU using double precision. We show the relative errors on both single (Fig. A.6) and double precision (Fig. A.7) for Coulomb kernel. Since 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 enough to improve the overall translation accuracy. Moreover, the kernel evaluations within neighborhood also introduce floating point number truncation errors. Hence the single precision case in Fig. A.6 shows no accuracy improvement from $p = 8$ to $p = 12$. Note that our simulation is done in a unit cube, and the number of sources and receivers is increased, and the error computed at 100 receiver locations. In a case where there are 10,000 receivers and sources, the probability of a source and a receiver being close is some value. In a case where there are 1,000,000 receivers and sources, this probability is higher. Our error is computed from Eq. 27. This shows that where the denominator has the norm of the solution. The numerator is not growing in this formula, while the denominator in a larger sized problem will have more large terms, leading to the decreased relative error for double precision. The same behavior is observed in the single precision case, but saturates due to round off errors.

3. Vortex Ring and Particle Interaction Simulation

The mathematical model to simulate vortex ring and particle interactions is based on vortex particle method [22]. In [1], they showed smoke, water and explosion visual effects using vortex particle method but the number of vortex elements used were only in order of hundreds or thousands. With the GPU-based FMM, the same kind of simulations can be scaled to large size problems in which there are $O(10^5)$ vortex elements and millions of particles.

3.1. Smoothing Kernel

A big challenge of the simulation is the integration stability. In the large scale simulation, many particles are very near to each other, hence the round-off errors of their distance are enlarged dramatically due to the kernel singularity, which makes the direct time step integration of the particle displacement not stable. For Biot-Savart kernel, the vortices have dipole singularities, so the field grow as $1/|\mathbf{r}|^2$ near the source location. Based on our experiments, even the direct CPU computation using double precision will blow up within several time steps on small size problems. An effective solution to this problem is to introduce *smoothing kernel* $K(d, \varepsilon)$ to reduce the computation kernel singularity as

$$V(\mathbf{y}) = \sum_{i=1}^{n} \frac{\mathbf{q}_i \times (\mathbf{y} - \mathbf{x}_i)}{|\mathbf{y} - \mathbf{x}_i|^3} K(|\mathbf{y} - \mathbf{x}_i|, \varepsilon), \tag{28}$$

where

$$K(d, \varepsilon) = \begin{cases} \dfrac{d^2}{c\varepsilon^2} & \text{if} \quad d \leq \varepsilon, \\ 1 & \text{if} \quad d > \varepsilon. \end{cases} \tag{29}$$

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 enlarged by kernel singularity is cut off by enforcing $d_{min} \ll \varepsilon$ while it still makes modifications of FMM simple, i.e., by setting $\varepsilon < u$, which means that only the local kernel evaluations of the direct sum need to be modified. Other smoothing kernel functions can also be used, however, since the transcendental functions computation are expensive on the GPU, simple polynomial smoothing kernels are preferred.

As for the numerically integration, both Euler and Runge-Kutta 4 methods are implemented. Euler method with one FMM evaluation at each time step is fast while Runge-Kutta 4 requiring four FMM evaluations is robust. The simulation results reported in this paper used the Euler method.

3.2. Interactive Computational Visualization

The visualization of the particle positions and movements during the simulation is realized via OpenGL and the OpenGL Extension Wrangler Library (GLEW). Since the computations are performed on the device using CUDA, the rendering can be performed directly on GPU (without data transfer between GPU and CPU) by the CUDA OpenGL *interoperability* [18].

To visualize the interactions, the particles are drawn as OpenGL points with certain size in a 3D cube. Vortex elements are only computed but not rendered. Although the simulation is performed on a large number of particles, it does not deliver a good visual effect to render them all. This is because that the number of pixels within the range of particles on the final screen is much less than the number of particles. In that case, rendering all the particles will result in a very bright

region, hence part of the depth and density visual effects will be lost. Instead, in our implementation, only part of particles are rendered with blending enabled and the full particle information are used in *ray tracing* [35, Chapter 10] to compute the opacity for each pixel, which is used to adjust the pixel brightness to reflect particle density for a better realistic visual effect.

Recall the array `bin[]` described in Sec. 2.1 in which its ith entry keeps the count of the number of data points in the box i. Given the ray from the eye to certain pixel, a thread keeps an particle count and samples k points along that ray to find which the boxes in the maximal level that intersect with the ray. Once a box index j is returned, the thread increases the count by `bin[j]`. After the opacities of all the pixels are obtained, they are further smoothed by averaging the opacities of nearby pixels. The opacity information is computed and smoothed by CUDA threads then is passed to the rendering function as a texture map. After rendering part of the particles as point, a *fragment shader* is used to reset the pixel values according to that opacity information.

3.3. Experimental Result

We used a workstation with INTEL Xeon E5504 CPU 2.0GHz, 12GB RAM and a single NVIDIA Tesla C2050 (it is capable for graphic rendering) to perform all the experiments. Our FMM is double precision (ECC disabled), and the truncation number was set to 12. All the data are generated within a unit cube and only Euler integration is used for simulations. The vortex rings are constructed with the radius being 0.3 and the fluid particles are generated randomly around these two vortex rings.

Figure A.1 and Fig. A.8 were captured frames from the demonstration video, in which two vortex rings with totally 2^{12} discretized elements and 2^{18} particles, in which 2^{14} particles were rendered. Figure A.1 showed the leap-frog of two vortex rings while Fig. A.8 showed the collision of two rings. In Fig. A.9 and Fig. A.10, there are 2^{15} vortex elements and 2^{18} fluid particles generated in total, we have 2^{15} and 2^{18} particles were rendered respectively. The last visualization was shown in Fig. A.11, in which we have 2^{15} vortex elements and 2^{20} fluid particles with all particles rendered. Since the computation and rendering share the same hardware and the overheating issue due to large time steps, the performance is much inferior compared with testing results Sec. 2.4. But the total running time for each frame is still around $1.4 \sim 2.5$ seconds on double precision data along the whole simulation process.

4. Conclusion

FMM has complex data structures and translation schemes. Using parallel algorithms allows this efficient but complicated algorithm to take advantage of the GPU hardware. It can achieve good speedup compared with other sequential implementations. The errors introduced by its approximation of far field interactions almost have no effect on the simulations. Problems of large size can be computed on a single GPU equipped desktop, which currently can only be completed otherwise in practical time on expensive clusters.

In this paper, the GPU-based FMM with parallel data structures are developed for dynamic problems. This enables us to fully off-load computations and visualizations to the GPU. The development of real coefficient representations and

the adaptation to the Biot-Savart kernel allows us to implement highly efficient FMM expansion and translation calls on the GPU. Our novel GPU implementation is capable of both single and double precision computation and demonstrates the superior timing and error bounds to direct methods for practical simulations on a desktop with single GPU. Successful visualizations to long times with large number of particles and vortex elements are also demonstrated.

Acknowledgements

Work partially supported by AFOSR under MURI Grant W911NF0410176 (PI Dr. J. G. Leishman, monitor Dr. D. Smith); Work also partially supported by Fantalgo, LLC.

Appendix A. The Recurrence Relations for Gradients

We use the same notations as Sec. 2.3 and denote

$$d_n^{(x),m} = \frac{\partial d_n^m}{\partial x}, \;\; d_n^{(y),m} = \frac{\partial d_n^m}{\partial y}, \;\; d_n^{(z),m} = \frac{\partial d_n^m}{\partial z}. \tag{A.1}$$

Then the recurrence relations of gradient coefficients in Eq. 24 are given by:

$$d_0^{(x),0} = 0, \;\; d_1^{(x),1} = -\frac{1}{2}, \;\; d_1^{(x),-1} = 0,$$

$$
\begin{aligned}
d_{|m|}^{(x),|m|} &= -\frac{d_{|m|-1}^{|m|-1}}{2|m|} - \frac{x d_{|m|-1}^{(x),|m|-1} + y d_{|m|-1}^{(x),-|m|+1}}{2|m|}, \ |m| = 2,3,\ldots, \\
d_{|m|}^{(x),-|m|} &= -\frac{d_{-|m|+1}^{|m|-1}}{2|m|} + \frac{y d_{|m|-1}^{(x),|m|-1} - x d_{|m|-1}^{(x),-|m|+1}}{2|m|}, \ |m| = 2,3,\ldots, \\
d_{|m|+1}^{(x),m} &= -z d_{|m|}^{(x),m}, \qquad m = 0, \pm 1, \pm 2, \ldots, \\
d_n^{(x),m} &= -\frac{2x d_{n-2}^{m}}{n^2 - m^2} - \frac{(2n-1) z d_{n-1}^{(x),m} + r^2 d_{n-2}^{(x),m}}{n^2 - m^2}, \\
& \quad n = |m| + 2, |m| + 3, \ldots, \\
& \quad m = -n, \ldots, n.
\end{aligned}
\tag{A.2}
$$

$$
d_0^{(y),0} = 0, \ \ d_1^{(y),1} = 0, \ \ d_1^{(y),-1} = \frac{1}{2},
$$

$$
\begin{aligned}
d_{|m|}^{(y),|m|} &= -\frac{d_{-|m|+1}^{|m|-1}}{2|m|} - \frac{x d_{|m|-1}^{(y),|m|-1} + y d_{|m|-1}^{(y),-|m|+1}}{2|m|}, \ |m| = 2,3,\ldots, \\
d_{|m|}^{(y),-|m|} &= -\frac{d_{|m|-1}^{|m|-1}}{2|m|} + \frac{y d_{|m|-1}^{(y),|m|-1} - x d_{|m|-1}^{(y),-|m|+1}}{2|m|}, \ |m| = 2,3,\ldots, \\
d_{|m|+1}^{(y),m} &= -z d_{|m|}^{(y),m}, \qquad m = 0, \pm 1, \pm 2, \ldots, \\
d_n^{(x),m} &= -\frac{2y d_{n-2}^{m}}{n^2 - m^2} - \frac{(2n-1) z d_{n-1}^{(y),m} + r^2 d_{n-2}^{(y),m}}{n^2 - m^2}, \\
& \quad n = |m| + 2, |m| + 3, \ldots, \\
& \quad m = -n, \ldots, n.
\end{aligned}
\tag{A.3}
$$

$$
d_0^{(z),0} = 0, \ \ d_1^{(z),1} = 0, \ \ d_1^{(z),-1} = 0,
$$

$$
\begin{aligned}
d^{(z),|m|}_{|m|} &= -\frac{xd^{(z),|m|-1}_{|m|-1} + yd^{(z),-|m|+1}_{|m|-1}}{2|m|}, \; |m| = 2, 3, \ldots, \\
d^{(z),-|m|}_{|m|} &= +\frac{yd^{(z),|m|-1}_{|m|-1} - xd^{(z),-|m|+1}_{|m|-1}}{2|m|}, \; |m| = 2, 3, \ldots, \\
d^{(z),m}_{|m|+1} &= -zd^{(z),m}_{|m|} - zd^{(z),m}_{|m|}, \qquad m = 0, \pm 1, \pm 2, \ldots, \\
d^{(z),m}_{n} &= -\frac{(2n-1)d^{m}_{n-1} + 2zd^{m}_{n-2}}{n^2 - m^2} \\
&\quad -\frac{(2n-1)zd^{(z),m}_{n-1} + r^2 d^{(z),m}_{n-2}}{n^2 - m^2}, \\
&\quad n = |m| + 2, |m| + 3, \ldots, \\
&\quad m = -n, \ldots, n.
\end{aligned} \tag{A.4}
$$

References

[1] A. Selle, N. Rasmussen, R. Fedkiw, A vortex particle method for smoke, water and explosions, ACM Trans. Graph. 24 (2005) 910–914.

[2] F. Losasso, J. Talton, N. Kwatra, R. Fedkiw, Two-way coupled SPH and particle level set fluid simulation, IEEE Transactions on Visualization and Computer Graphics 14 (2008) 797–804.

[3] R. K. Beatson, J. B. Cherrie, D. L. Ragozin, Fast evaluation of radial basis functions: Methods for four-dimensional polyharmonic splines, SIAM J. Math. Anal. 32 (2001) 1272–1310.

[4] N. A. Gumerov, R. Duraiswami, Fast radial basis function interpolation via preconditioned Krylov iteration, SIAM J. Scientific Computing 29 (5) (2007) 1876–1899.

[5] M. F. Cohen, S. E. Chen, J. R. Wallace, D. P. Greenberg, A progressive

refinement approach to fast radiosity image generation, SIGGRAPH Comput. Graph. 22 (4) (1988) 75–84.

[6] L. Nyland, M. Harris, J. Prins, Fast n-body simulation with CUDA, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley Professional, 2007, Ch. 31, pp. 677–695.

[7] D. Groen, S. P. Zwart, T. Ishiyama, J. Makino, High-performance gravitational n-body simulations on a planet-wide-distributed supercomputer, Computational Science & Discovery 4 (1) (2011) 015001.

[8] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann Publishers Inc., 2005.

[9] N. A. Gumerov, R. Duraiswami, Fast multipole methods on graphics processors, J. Comput. Phys. 227 (18) (2008) 8290–8313.

[10] F. A. Cruz, M. G. Knepley, L. A. Barba, PetFMMa dynamically load-balancing parallel fast multipole library, International Journal for Numerical Methods in Engineering 85 (4) (2011) 403–428.

[11] R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, K. Yasuoka, Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence, Computer Physics Communications 180 (2009) 2066–2078.

[12] R. Yokota, L. A. Barba, Comparing the treecode with fmm on GPUs for vortex particle simulations of a leapfrogging vortex ring, Computer & Fluids 45 (2011) 155–161.

[13] M. J. Stock, A. Gharakhani, Toward efficient GPU-accelerated n-body simulations, in: 46th AIAA Aerospace Sciences Meeting, AIAA 2008-608, 2008.

[14] Q. Hu, N. A. Gumerov, R. Duraiswami, Scalable fast multipole methods on distributed heterogeneous clusters, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '11, ACM, Seattle, WA, USA, 2011, pp. 1–12.

[15] R. Yokota, L. A. Barba, Hierarchical n-body simulations with autotuning for heterogeneous systems, Computing in Science and Engineering (CiSE) 14 (2012) 30–39.

[16] NVIDIA, NVIDIA's next generation cuda compute architecture: Fermi.

[17] D. B. Kirk, W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 1st Edition, Morgan Kaufmann, 2010.

[18] NVIDIA, NVIDIA CUDA C Programming Guide, 3rd Edition (2010).

[19] NVIDIA, OpenCL Programming Guide for the CUDA Architecture, 3rd Edition (2010).

[20] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (2007) 80–113.

[21] A. Davidson, J. D. Owens, Toward techniques for auto-tuning GPU algorithms, in: Para 2010: State of the Art in Scientific and Parallel Computing, 2010.

[22] G. H. Cottet, P. Koumoutsakos, Vortex Methods: Theory and Practice, Cambridge University Press, 2000.

[23] J. Dongarra, F. Sullivan, Guest editors' introduction: the top 10 algorithms, Computing in Science and Engineering 2 (2000) 22–23.

[24] M. A. Epton, B. Dembart, Multipole translation theory for the three-dimensional laplace and helmholtz equations, SIAM J. Sci. Comput. 16 (1995) 865–897.

[25] N. A. Gumerov, R. Duraiswami, Comparison of the efficiency of translation operators used in the fast multipole method for the 3D laplace equation, Tech. Rep. CS-TR-4701, UMIACS-TR-2005-09 (2005).

[26] N. A. Gumerov, R. Duraiswami, Fast Multipole Methods for the Helmholtz Equation in Three Dimensions, ELSEVIER, Oxford, 2004.

[27] J. Bédorf, E. Gaburov, S. Portegies Zwart, A sparse octree gravitational n-body code that runs entirely on the GPU processor, Journal of Computational Physics 231 (7) (2012) 2825–2839.

[28] P. Ajmera, R. Goradia, S. Chandran, S. Aluru, Fast, parallel, GPU-based construction of space filling curves and octrees, in: Proceedings of the 2008 symposium on Interactive 3D graphics and games, I3D '08, ACM, New York, NY, USA, 2008, pp. 10:1–10:1.

[29] G. Blelloch, Scans as primitive parallel operations, IEEE Transactions on Computers 38 (1987) 1526–1538.

[30] M. Harris, S. Sengupta, J. D. Owens, Parallel prefix sum (scan) with CUDA, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley, 2007, Ch. 39, pp. 851–876.

[31] G. M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, in: IBM Germany Scientific Symposium Series, 1966.

[32] N. A. Gumerov, R. Duraiswami, Y. A. Borovikov, Data structures, optimal choice of parameters, and complexity results for generalized multilevel fast multipole methods in d dimensions, Tech. Rep. CS-TR-4458; UMIACS-TR-2003-28 (2003).

[33] Q. Hu, N. A. Gumerov, R. Duraiswami, Parallel algorithms for constructing data structures for fast multipole methodsarXiv:1301.1704.

[34] C. A. White, M. Head-Gordon, Rotating around the quartic angular momentum barrier in fast multipole method calculations, The Journal of Chemical Physics 105 (1996) 5061–5067.

[35] P. Shirley, M. Ashikhmin, M. Gleicher, S. Marschner, E. Reinhard, K. Sung, W. Thompson, P. Willemsen, Fundamentals of Computer Graphics, Second Ed., A. K. Peters, Ltd., Natick, MA, USA, 2005.

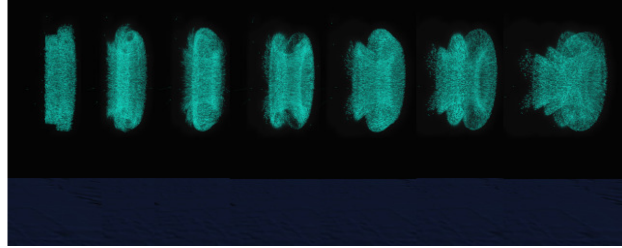

Figure A.1: *Vortex rings and particles interaction*

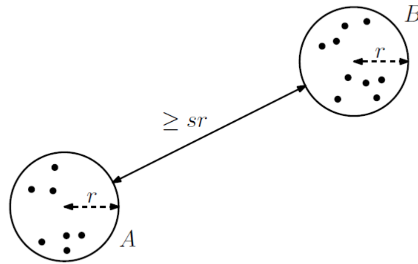

Figure A.2: *A well separated pair.*

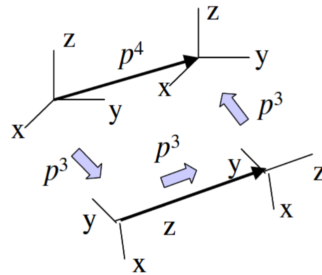

Figure A.3: *RCR translation for the Fast multipole method, replaces one $O(p^4)$ translation with two $O(p^3)$ rotations and one $O(p^3)$ coaxial translation.*

$l_{\max}$	CPU (ms)	Improved CPU (ms)	GPU (ms)
3	1293	223	7.7
4	1387	272	13.9
5	2137	431	13.0
6	8973	1808	34.6
7	30652	6789	70.8
8	58773	7783	124.9

Table A.1: *FMM data structure computation for 2^{20} uniform randomly distributed source and receiver particles using our original CPU $O(N \log N)$ algorithm, the improved $O(N)$ algorithm on a single CPU core, and its GPU accelerated version.*

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

Table A.2: *The time comparison (on single precision) between the Coulomb and Bio-Savart kernels. The total run time of Bio-Savart kernel is only doubled but not tripled by comparing with the baseline (Coulomb kernel) FMM.*

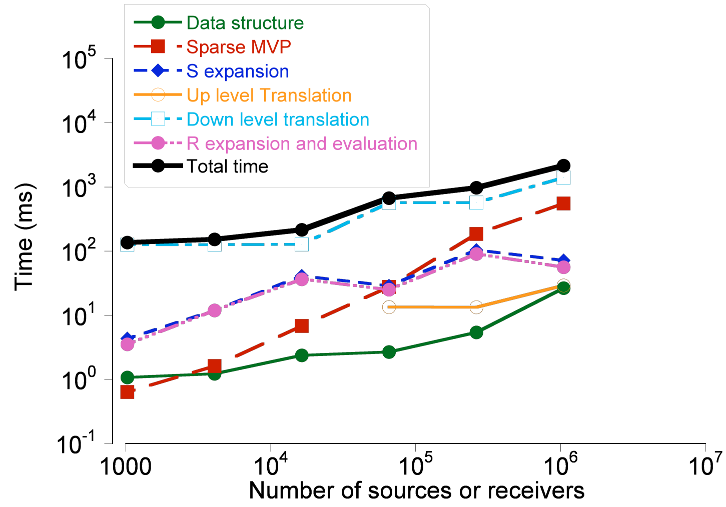

Figure A.4: *The profiling of FMM for Biot-Savart kernel*

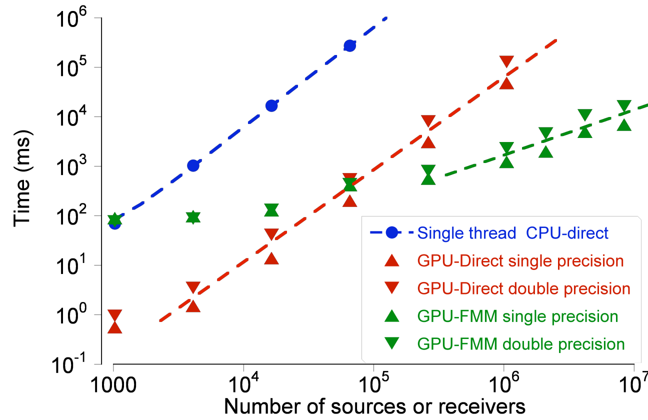

Figure A.5: *The time comparisons for Coulomb kernel between GPU-based FMM and direct methods*

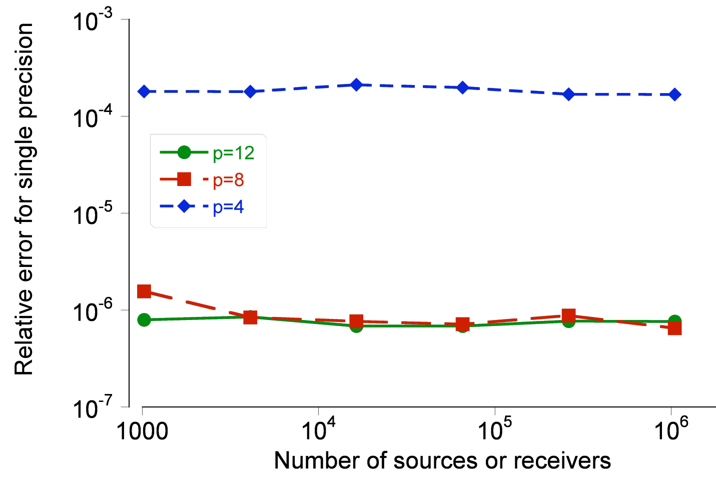

Figure A.6: *Relative errors in single precision for the Coulomb kernel.*

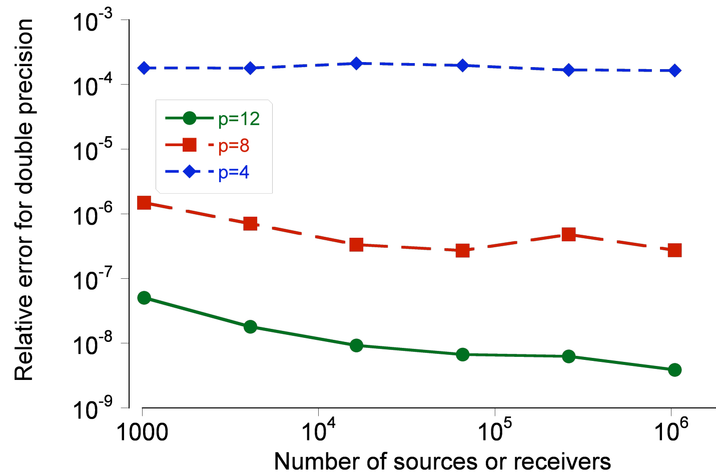

Figure A.7: *Relative errors in double precision for the Coulomb kernel.*

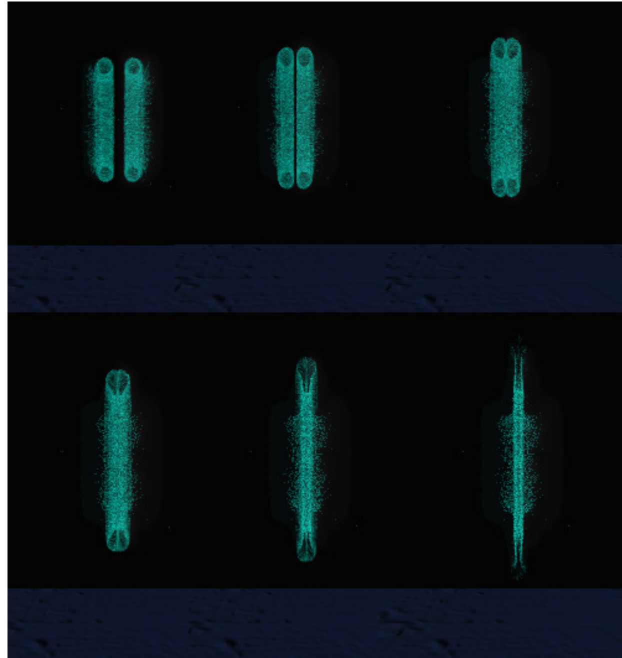

Figure A.8: *Collision of two vortex rings*

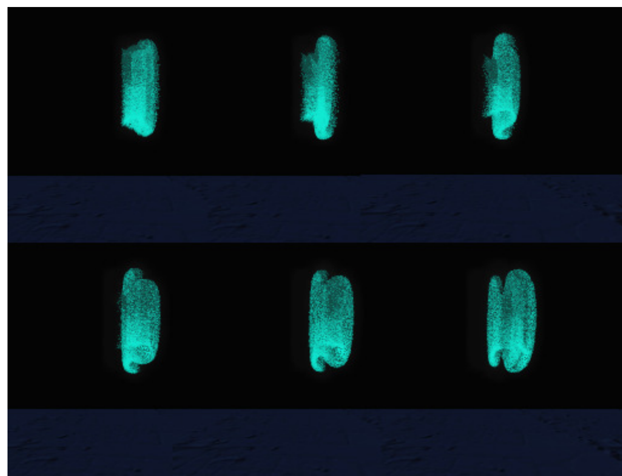

Figure A.9: *The leap-frog of two rings with* 2^{15} *particles rendered*

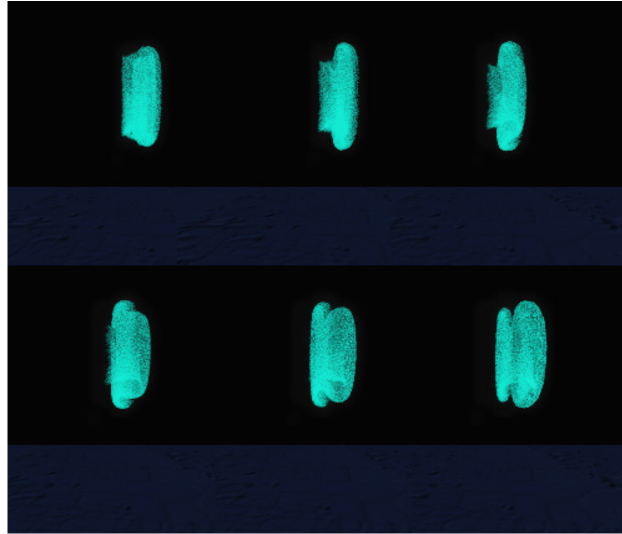

Figure A.10: *The leap-frog of two rings with* 2^{18} *particles rendered*

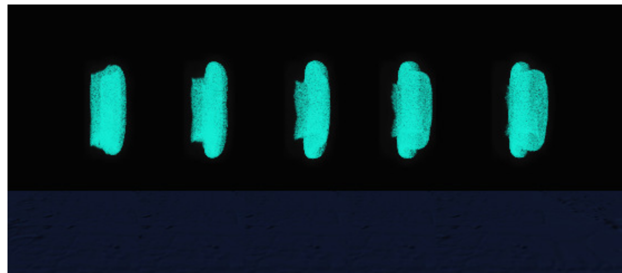

Figure A.11: *The leap-frog of two rings with* 2^{20} *particles*