ABSTRACT

Title of dissertation: Scalable Fast Multipole Method on Heterogeneous Architecture

Qi Hu, Doctor of Philosophy, 2013

Dissertation directed by: Professor Ramani Duraiswami
Department of Computer Science
Professor Nail A. Gumerov
University of Maryland Institute for Advanced Computer Studies

The N-*body* problem appears in many computational physics simulations. At each time step the computation involves an all-pairs sum whose complexity is quadratic, followed by an update of particle positions. This cost means that it is not practical to solve such dynamic N-body problems on large scale. To improve this situation, we use both algorithmic and hardware approaches. Our algorithmic approach is to use the Fast Multipole Method (FMM), which is a divide-and-conquer algorithm that performs a fast N-body sum using a spatial decomposition and is often used in a time-stepping or iterative loop, to reduce such quadratic complexity to linear with guaranteed accuracy. Our hardware approach is to use heterogeneous clusters, which comprised of nodes that contain multi-core CPUs tightly coupled with *accelerators*, such as graphics processors unit (GPU) as our underline parallel processing hardware, on which efficient implementations require highly non-trivial re-designed algorithms.

In this dissertation, we fundamentally reconsider the FMM algorithms on

heterogeneous architectures to achieve a significant improvement over recent/previous implementations in literature and to make the algorithm ready for use as a workhorse simulation tool for both time-dependent vortex flow problems and for boundary element methods. Our major contributions include:

1. Novel FMM data structures using parallel construction algorithms for dynamic problems.
2. A fast hetegenenous FMM algorithm for both single and multiple computing nodes.
3. An efficient inter-node communication management using fast parallel data structures.
4. A scalable FMM algorithm using novel Helmholz decomposition for Vortex Methods (VM).

The proposed algorithms can handle non-uniform distributions with irregular partition shapes to achieve workload balance and their MPI-CUDA implementations are highly tuned up and demonstrate the state of the art performances.

SCALABLE FAST MULTIPOLE METHOD ON HETEROGENEOUS ARCHITECTURE

by

Qi Hu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2013

Advisory Committee:
Professor Ramani Duraiswami, Chair/Advisor
Professor Nail A. Gumerov
Professor Dave Mount
Professor Howard Elman
Professor J. Gordon Leishman

© Copyright by
Qi Hu
2013

Acknowledgments

My graduate study at University of Maryland, College Park is fruitful and cherish. My life over those years would not be the same without many people to whom I owe my sinsere gratitude.

First and foremost I would like to thank my advisors, Professor Ramani Duraiswami and Professor Nail A. Gumerov for giving me an invaluable opportunity to work on challenging and extremely interesting projects on Fast Multipole Methods over the past four years. It is my great honor to work with those world leading talented researchers. I really appeciate their knowledge teaching, encouragement on my research, and contributions of time, ideas as well as fundings to make my Ph.D. study productive. Without their extraordinary theoretical and computational expertise, this thesis would have been a distant dream. I am grateful to their invaluable advice and great help on both research as well as on my future career.

I would also like to thank Professor J. Gordon Leishman, who is the principal investigator of my funding project and provides generous funding resources to support my research. Thanks are due to Professor Dave Mount, Professor Howard Elman for agreeing to serve both on my Ph.D. preliminary examination and dissertation defense committee and for sparing their invaluable time and effort reviewing the manuscript.

The staffs at UMIACS help desk deserve a special mention with my great appreciation. They always helped me effectively on hardware trouble shootings with the most earliest responses and provide continuously reliable IT support. Without their outstanding skills, I would not be able to finsih many experiments hence the research

papers on time.

I owe my deepest gratitude to my family - my mother and father who have always stood by me, given me strong support and cheered me up over the years.

I have my dearest friends to thank. It is them that made my graduate life at Maryland been colorful, joyful and memorable. I'd like to express my sincere gratitude to my best friend and mentor, Qiu Qiang, who shared me with lots of valuable perceptions of life and insights of technologies and truth. I want to say thank you very much to Chen Daozheng, Guo Huimin, Liu Bo and Shi Bing for their greatest help on my job hunting and career. Although many others' names are not mentioned, I want to say that they are the charming gardeners who make the world around blossom.

I would like to acknowledge the financial support from the AFOSR under MURI Grant W911NF0410176 and Department of Computer Science University of Maryland for all the projects discussed herein.

Lastly, many thanks to all!

Table of Contents

List of Tables vii

List of Figures viii

List of Algorithms x

List of Abbreviations xi

1 Introduction 1
- 1.1 Fast Multipole Method 2
- 1.2 Hardware Accelerators 7
  - 1.2.1 Graphics Processing Unit 7
- 1.3 Related Work and Contributions 11
- 1.4 Organization of the Dissertation 13

2 Parallel Algorithms for Constructing Data Structures for Fast Multipole Methods 15
- 2.1 Well-separated Pair Decomposition 16
- 2.2 Treecode and Its Data Structures 17
- 2.3 Multi-Level FMM Data Structures 18
- 2.4 Motivation for Fast Data Structure Algorithms 21
- 2.5 Parallel FMM Data Structure Algorithm for GPU 22
  - 2.5.1 Pseudo-Sort Using Fixed-Grid-Method in Linear Time 24
  - 2.5.2 Interaction Lists 26
  - 2.5.3 Parallel Data Structure Construction 29
  - 2.5.4 GPU Implementation Considerations 34
  - 2.5.5 Complexity 35
- 2.6 Experimental Results 35
- 2.7 Summary 38

3 GPU Accelerated Fast Multipole Methods for Dynamic N-Body Simulation 39
- 3.1 Interactions between Vortices and Particles 41
- 3.2 GPU-Based Fast Multipole Method 42
  - 3.2.1 Real Representation 43

3.2.2 Adaption to the Biot-Savart 3D Kernel 46
3.2.3 The Recurrence Relations for Gradients 48
3.2.4 Test and Error Analysis 50
3.3 Vortex Ring and Particle Interaction Simulation 53
3.3.1 Smoothing Kernel 54
3.3.2 Interactive Computational Visualization 55
3.3.3 Experimental Result 57
3.4 Summary 61

4 The Scalable FMM on the Heterogenous Architecture 62
4.1 Single Heterogeneous Node 64
4.2 Several Heterogeneous Nodes 67
4.2.1 Simplified Algorithm 69
4.3 Implementation Issues 72
4.4 Performance Tests 73
4.4.1 Single Heterogeneous Node 74
4.4.2 Double Precision GPU Performance 76
4.4.3 Multiple Heterogeneous Nodes 78
4.5 Performance Assessment 83
4.6 Summary 85

5 Parallel Algorithms for FMM Data Structures on Multiple Nodes 87
5.1 Global Data Structure and Partitioning 89
5.2 FMM Algorithm on Multiple Nodes 94
5.2.1 Global Partition 95
5.2.2 The Distributed FMM Algorithm 97
5.2.3 Source Box Type 101
5.2.4 Complexity 104
5.3 Experimental Results 108
5.4 Summary 114

6 Scalable Vortex Methods Using Fast Multipole Methods 116
6.1 FMM for Vortex Methods 120
6.1.1 Lamb-Helmholtz Decomposition 120
6.1.2 Scalable Heterogeneous FMM Algorithm 123
6.1.3 On A Single Node 123
6.1.4 Multiple Nodes 125
6.2 Vortex Core Function Evaluation 128
6.3 Performance Tests 132
6.3.1 Hardware 133
6.3.2 Single Node Performance 134
6.3.2.1 Accuracy 136
6.3.3 Multiple-Node Performance 138
6.3.3.1 Weak Scalability 139
6.3.3.2 Strong Scalability 140

6.3.3.3 Algorithm Overheads 142
6.3.3.4 Billion-Size Run 143
6.4 Summary 145

7 Conclusions and Future Work 147
7.1 Single Node Data Structures 150
7.2 Automatic Balanced FMM Algorithms 152
7.3 Partition Methods and Dynamic Data Structure Updates 153

Bibliography 155

List of Tables

2.1 The Time Comparison of FMM Data Structure Computation 36

3.1 The Time Comparison (on single precision) between the Coulomb and Bio-Savart Kernels 50

4.1 Performance on A Single Heterogeneous Node 75
4.2 Performance and Error Results for Single and Double Precision Computations 77
4.3 Performance for P Heterogeneous Nodes with $N/P = 2^{23}$ 79

5.1 Description of Equitation 5.1 104
5.2 Description of Equitation 5.3 106
5.3 Performance for P Heterogeneous Nodes with $N/P = 2^{23}$ 112

6.1 Time Comparisons between with Cutoff and without Cutoff Functions . . 131
6.2 The Single-Node Performance Profiling 133
6.3 Performance Profiling for P Heterogeneous Nodes with $N/P = 2^{23}$ (8M) 138

List of Figures

1.1 The FMM vs Direct Method 4
1.2 A Flow Chart of the Standard FMM 5
1.3 GPU Computation Power [1]. 8
1.4 GPU Memory Bandwidth [1]. 8
1.5 CUDA Thread and Memory Hierarchy 9

2.1 A Well Separated Pair 16
2.2 The E_1, E_2, E_3, E_4 Neighborhoods of Dimension 2 18
2.3 Bookmark List 25
2.4 Neighbor Lists 26
2.5 The Naïve Prefix Sum (Scan) Algorithm 34
2.6 The Data Structure Construction Time for Non-Uniform Distribution . . . 37

3.1 Vortex Rings and Particles Interaction 40
3.2 RCR Translation for the Fast Multipole Method 45
3.3 The Profiling of FMM for Biot-Savart Kernel 51
3.4 The Time Comparisons for Coulomb Kernel Between GPU-Based FMM and Direct Methods 52
3.5 Relative Error Analysis 53
3.6 Collision of Two Vortex Rings 56
3.7 The Leap-Frog of Two Rings with 2^{15} Particles Rendered 57
3.8 The Leap-Frog of Two Rings with 2^{18} Particles Rendered 57
3.9 The Leap-Frog of Two Rings with 2^{20} Particles Rendered 58
3.10 The Accuracy of Vortex Ring Simulation Using FMM for Each Time Step 59
3.11 The Simulation Snapshots of A Vortex Ring Colliding with Ground Plane 60

4.1 The Relative Cost and Speedup of Different Steps of the FMM 65
4.2 Flow Chart of the FMM on A Single Heterogeneous Node 66
4.3 Illustration of Separation of the M2M and M2L Translation Jobs between Two Nodes 68
4.4 A Flow Chart of the FMM on Two Heterogeneous Nodes 70
4.5 The Wall Clock Time for the heterogeneous FMM Running on A Single Node with One and Two GPUs 76

4.6 Contribution of the CPU/GPU Parallel Region Time and the Overhead to the Total Run Time for Two GPUs per Node . . . 80
4.7 The Results of the Strong Scalability for One and Two GPUs per Node . . 81
4.8 The Wall Clock Time for the Heterogeneous FMM Running on 32 Nodes Using One or Two GPUs per Node . . . 82
4.9 A Comparison of the Simplified Distributed FMM Implementation for N-Body Force+Potential Calculations with the 2009 Gordon Bell Prize Winner [2]for Velocity+Stretching Calculations . . . 83

5.1 Problems Due to Partition . . . 88
5.2 Global Data Structure with K Trees with Roots at Level 2 and Partitioned Data Structure with Roots at Any Level . . . 90
5.3 Source Box Type . . . 92
5.4 The Heterogenous CPU-GPU Computing Architecture of Chimara at UMIACS . . . 94
5.5 A Simple Multiple Node 2D FMM Algorithm Illustration ($l_{crit} = 2$) . . . 100
5.6 The CPU/GPU Concurrent Region Time and the Overhead of the Full FMM Algorithm against the Simplified FMM . . . 108
5.7 The Time Comparisons between the Full and Simplified FMM Algorithm 109
5.8 The Results of the Strong Scalability Test for 1 and 2 GPUs per Node of the Testing Cluster . . . 110
5.9 The Data Manager and Data Structure Processing Time against the Total Run Time . . . 111

6.1 The Single-Node FMM Algorithm . . . 124
6.2 An Overview of the Distributed FMM Algorithm . . . 126
6.3 Time Comparisons among Different Cutoff Implementations . . . 129
6.4 The Relative Error Comparisons among Different Cutoff Implementations with the Truncation Number $p = 16$ . . . 130
6.5 Time Comparisons between the Potential and Velocity+Stretching Computations Using 2 GPUs . . . 135
6.6 The Relative Errors of Velocity and Stretching Computation . . . 137
6.7 The Weak Scalability for $N = M = 2^{23}$ Particles per Node . . . 139
6.8 The Strong Scalability Test . . . 141
6.9 Overheads vs. the Total Time . . . 142
6.10 The Total Run Time on 32 Nodes Using 1 or 2 GPUs Each . . . 144

List of Algorithms

1 Parallel Pseudo-Sort 24
2 Access Source E_2 Neighborhood 28
3 Get Bookmark and Box Index 30
4 Get E_2 Neighborlist and Bookmark 31
5 Get E_2 Neighborlist and Bookmark Continued 32
6 Build FMM Data Structures 33
7 Get Source Box Type 102
8 Get Source Box Type Continued 103

List of Abbreviations

APU	Accelerated Processing Unit
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
ECC	Error Correcting Code
FMM	Fast Multipole Methods
GLEW	OpenGL Extension Wrangler
GPGPU	General Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
GRAPE	Gravity Pipe
LET	Local Essential Tree
MPI	Message Passing Interface
MVP	Matrix Vector Product
PCI	Peripheral Component Interconnect
SFC	Space Filling Curve
SLI	Scalable Link Interface
SMVP	Sparse Matrix Vector Product
UMIACS	University of Maryland Institute for Advanced Computer Study
VM	Vortex Methods
WSPD	Well Separated Pair Decomposition
`Idx`	Index
`Nei`	Neighbor
`Recv`	Receiver
`Src`	Source

Chapter 1: Introduction

The N-*body* problem and its corresponding matrix vector product problems arise in various applications, such as molecular dynamics, gravitation and fluid dynamics. Many physics based simulations can be efficiently and accurately performed using particle methods which, using analysis and geometric data structuring, focus computational resources at the location of sources or discontinuities (particles), and allow evaluation of relevant fields only at locations of interest. Compared to methods that use meshes which result in large discretizations, particle methods are extremely efficient.

In general form, N-body problems evaluate the weighted sum of a kernel function $\Phi(\mathbf{y}, \mathbf{x}_i)$ centered at N source locations $\{\mathbf{x}_i\}$ for all M receiver locations $\{\mathbf{y}_j\}$ with the strengths q_i (Eq. 1.1). They can also be viewed as dense $M \times N$ matrix vector products. Direct evaluation has quadratic $O(NM)$ complexity. Such direct evaluations cannot be scaled to large sizes required by high fidelity simulations.

$$\phi(\mathbf{y}_j) = \sum_{i=1}^{N} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i), \quad j = 1, 2, \ldots, M, \quad \mathbf{x}_i, \mathbf{y}_j \in \mathbb{R}^d. \tag{1.1}$$

While large numbers of particles may still be necessary for high fidelity, which makes normal direct computation methods impractical, hardware accelerations, such as [3] using GPU parallelism and [4] using specially constructed hardware called the "Gravity Pipe"(GRAPE), are often used but can only speedup the sum to some extent without

reducing its quadratic complexity.

An alternative way to evaluate such sums for particular kernels is to use *fast approximation algorithms*, for example, the Fast Multipole Method [5], the Barnes-Hut Method [6] and the Particle-Mesh Methods [7], which have lower asymptotic complexity when they are applicable. Because the FMM can achieve linear complexity while guarantee the approximation accuracy up to machine precision, we only focus on this method in this thesis, however the data structure techniques developed here may find application in the other fast algorithms also, and indeed wherever computations involve particles. Together with efficient parallel implementations on heterogeneous architecture of GPUs and multi-core CPUs, we are targeting at the state-of-art N-body simulation performance. Because of our collaboration with Prof. Leishman's group, the application problem is in fluid mechanics simulations using vortex methods. In this chapter, we will introduce the FMM and hardware accelerators briefly.

1.1 Fast Multipole Method

The FMM was first introduced by Greengard and Rokhlin in [5] and has been identified as one of the ten most important algorithmic contributions in the 20th century [8]. Since the publication of [5], hundreds of papers on the FMM and applications have been published, and the theory of the FMM has already been well established and developed. The main purpose of FMM is to speed up the matrix-vector product Eq. 1.1. In the context of vortex methods, Φ corresponds to the Biot–Savart kernel [9]. The FMM acceleration is achieved because of the sum is computed approximately, but with a

guaranteed specified accuracy, which may be set, e.g. to the machine precision.

The main idea in the FMM is to split the sum in Eq. 1.1 into a near and a far field as

$$\phi(\mathbf{y}_j) = \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i) + \sum_{\mathbf{x}_i \in \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i), \tag{1.2}$$

for $j = 1, 2, \ldots, M$ where Ω is the set of points that neighbor the point $\mathbf{y}_j$. Computations related to the second sum are said to be *near-field*, while those arising from the first sum are *far-field*.

The most expensive computations are related to the first sum, which involves $O(N^2)$ interactions. The far-field sum term on the right hand side of Eq. 1.2 can be computed exactly at $O(N)$ cost given a fixed cluster size, i.e., the maximal number of data points inside any neighborhood domain. To approximate it, the kernel function is factored into a convergent sum, which is truncated at p terms using an error bound according to the required accuracy, by using singular (multipole) spherical basis functions, S_l, and regular (local) spherical basis functions R_l (Eq. 1.3). We call p the *t*runcation number, which is a function of the specified tolerance $\epsilon = \epsilon(p)$ with the max degree of spherical harmonics $p - 1$. These factorizations can be used to separate the kernel computations involving the points sets either $\{\mathbf{x}_i\}$ or $\{\mathbf{y}_j\}$, and consolidate operations for many points as

$$\begin{aligned} \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i \Phi(\mathbf{y}_j - \mathbf{x}_i) = & \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i \sum_{l=0}^{p-1} S_l(\mathbf{y}_j - \mathbf{x}_*) R_l(\mathbf{x}_i - \mathbf{x}_*), \\ = & \sum_{l=0}^{p-1} S_l(\mathbf{y}_j - \mathbf{x}_*) \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} q_i R_l(\mathbf{x}_i - \mathbf{x}_*), \\ = & \sum_{l=0}^{p-1} C_l S_l(\mathbf{y}_j - \mathbf{x}_*), \end{aligned} \tag{1.3}$$

where the p coefficients C_l for all $\mathbf{x}_i$ are built in pN operations and then they can be used in the evaluation at all $\mathbf{y}_j$ in pM operations. This approach reduces the cost of evaluating

the far-field contributions as well as the memory requirement to $O(N + M)$.

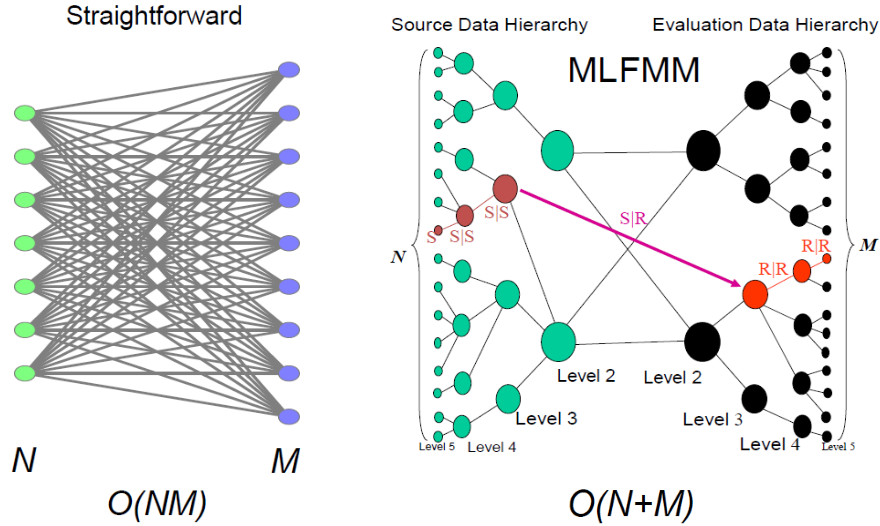


Figure 1.1: The Multi-Level FMM vs Direct Method. The straightforward method the number of operations is $O(MN)$. The number of operations (denoted by the connecting lines) can be reduced to $O(M + N)$ by using the FMM.

Because the factorization in Eq. 1.3 is not global, the split between the near and far-fields must be managed, which requires appropriate data structures and a variety of representations. This 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) [10], which is itself useful for solving a number of geometric problems [11, chapter 2]. Assume all the data points are already scaled into a unit cube. This WSPD is recursively performed to subdivide this cube into sub-cubes via *octree*s 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

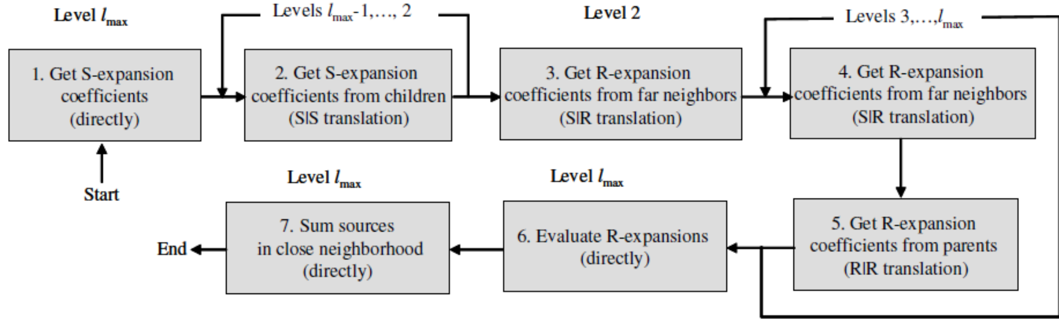


Figure 1.2: A Flow Chart of the Standard FMM.

completed at cost $O(N + M)$ to be consistent with the overall FMM cost.

The FMM puts sources into hierarchical space boxes and translates the consolidated interactions of sources into receivers. For the 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 containing four main parts: the initial expansion, the upward pass, the downward pass and the final summation.

1. **Initial expansion (P2M)**:

   (a) In the finest level l_{max}, all the source data points 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 source points within the same boxes are consolidated into a single expansion at each box center.

2. **Upward pass (M2M)**: For levels from l_{max} to 2, the $\mathcal{M}$ expansion coefficients for each box are evaluated via the multipole-to-multipole ($\mathcal{M}|\mathcal{M}$) translations from the

source boxes to their parent source box. 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 also 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 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 $\mathcal{L}$ 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 perform a local direct sum of nearby source points within their neighborhood domains.

Note that the evaluations of the nearby source point direct sums 4 (L2P) 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 computation efficiency and proper scaling. Reference [12] gives more details on the algorithm, Ref. [13] describes the translation theory, and Ref. [14] discusses the different translation methods.

1.2 Hardware Accelerators

There has been a revolution underway over the past decade or so in the use of accelerators, such as graphics inspired hardware for accelerating general purpose computation (GPUs, APUs) or INTEL Xeon PHI. Due to the current graphics processing unit (GPU) tremendous evolution in terms of increased computation power at decreased related energy consumption, the general purpose computing on GPU (GPGPU) is getting more and more popular and well-accepted by the high performance computing community. Nowadays, almost all high-end workstations or high performance computing clusters are equipped one or more such highly parallel, many-core GPU accelerators. Generally speaking, data on the many-core accelerators are processed as *warps*, i.e, a group of threads executing the same instruction at the same time and thousands of threads are spawn to run in parallel. Hence the parallel algorithms presented in this dissertation are designed for performance efficiency under this massive thread executing context. We particularly focus on the parallel programming on NVIDIA GPUs and CUDA. Nevertheless, our algorithms could be implemented similarly by using OpenCL [15] or OpenACC [16] on different many-core accelerators such as AMD GPU or INTEL Xeon PHI.

1.2.1 Graphics Processing Unit

A graphics 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

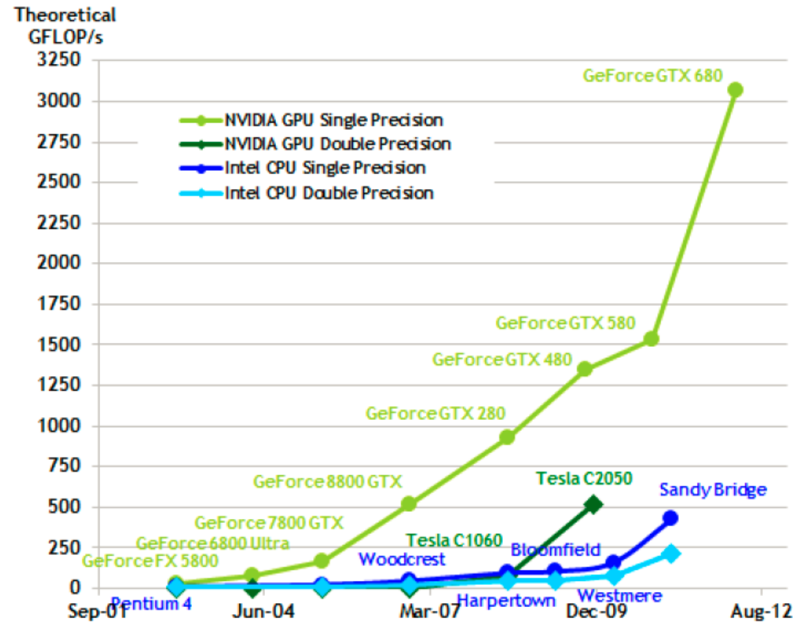


Figure 1.3: GPU Computation Power [1].

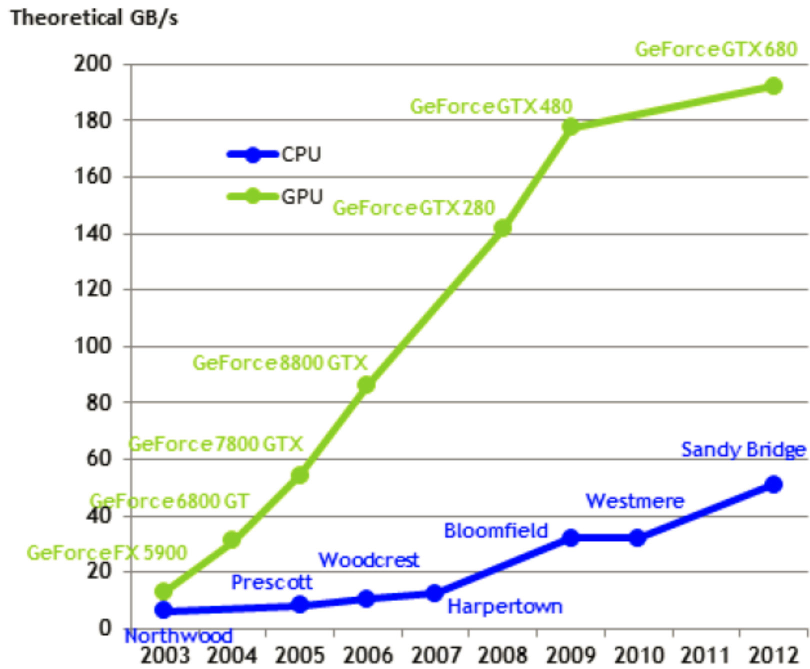
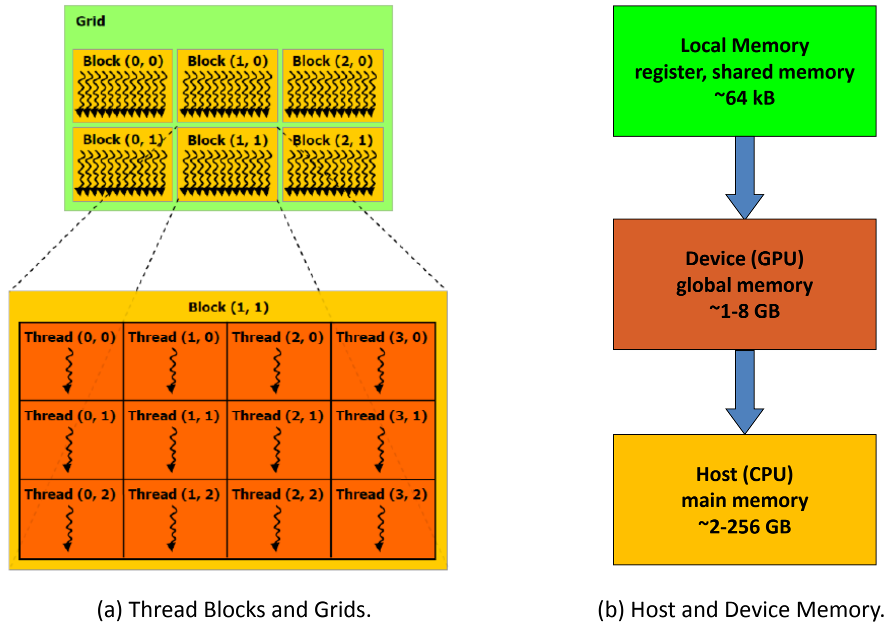


Figure 1.4: GPU Memory Bandwidth [1].



(a) Thread Blocks and Grids.

(b) Host and Device Memory.

Figure 1.5: CUDA Thread and Memory Hierarchy.

data (SIMD). Hence, more transistors on a GPU chip are devoted to data processing rather than data caching and flow control. Modern GPUs are capable of both single and double precision computations up to Tera-FLOPs on a single accelerator. Over the years, GPU architectures have evolved tremendously (see Fig. 1.3 and 1.4). Because GPUs are attached to the host (CPU) via PCI-Express bus, processing data on those accelerators requires data transfer between host and device (GPU). The on-chip memories are hierarchical and the programming focus is to best use these hierarchical memories (see Fig. 1.5 (b)) in the threaded model efficiently given the trade-off between speed and size [1, 17]. Moreover, these host-device memory communications are expensive compared to 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.

It is important to understand the hierarchical memory architecture on GPU to develop efficient codes. In the current NVIDIA Fermi architecture [18], on which the simulations in the thesis were performed, 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 and communicate with the host main memory.

2. *Constant memory*: 64 KB read only constant cache shared by all the threads. This is mainly used 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*: 32 KB fast on-chip registers for each SM. They are the fastest memory among all the memory hierarchy and are mainly used to hold instructions, input operands and values.

General purpose computing on graphics processing units (GPGPU) has become popular in the scientific community since 2000. However, programming on the GPU remained a technical barrier because it required deep computer graphics knowledge. In 2006, NVIDIA released a general purpose parallel computing architecture called CUDA so that a programmer can more easily manipulate the GPU and use a large number of parallel executing threads. Its scalable programming model allows the CUDA architecture to span a wide market range by simply scaling the number of processors and memory

partitions.

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. 1.5 (a))
2. Different kinds of memory
3. Synchronization mechanisms

such that the working problem can be divided into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. The state of the art for GPGPU applications is discussed in [19]. Besides CUDA, OpenCL (Open Computing Language) [15], DirectCompute by Microsoft [20] and OpenACC are other ways to program GPUs using similar libraries and APIs. One more thing worth noting is that OpenCL and OpenACC [16] are two different general frameworks on which users could write programs that can execute across heterogeneous platforms consisting of CPUs, GPUs, and other accelerator processors.

1.3 Related Work and Contributions

Since the invention of FMM, there are many research papers published regarding its optimization, distributed algorithms on parallel hardware and applications to different physical problems. The GPU parallelization of the FMM was first achieved in [21] and continued in several papers published in SC 2009 and 2010, including the 2009 Gordon

Bell prize winner [2]. 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. This is fine for single time step computation or iterative methods which use the same FMM data structures over multiple iterations. For dynamic problems, we need to seek more efficient way to re-build data structures every time step. There are several different versions of distributed FMM algorithms in the literature, such as [22–25]. The FMM was considered on a cluster of GPUs in [2, 26], and the benefits of architecture tuning on networks of multi-core processors or GPUs was considered in [27–29]. In those papers including the recent Peta-Scale turbulence simulation using FMM [30], the distributed FMM and tree code algorithms are mostly related to more coarse-grained parallel algorithms and use homogeneous computing, in which the potential powerful computing accelerators are not optimally utilized. Moreover those algorithms commonly use the *local essential tree* (LET [27, 31]) to manage communication among all the computing nodes. However, their implementation details for import or export data via LETs are not explicitly described.

In this dissertation, our purpose is to develop efficient data structures and distributed FMM algorithms with fine-granularity on massive parallel processing hardware to solve dynamic problems. Our main contributions can be summarized as:

- Fast FMM data structures for both single and multipole nodes: Those data structures include spatial data re-arrangements, interaction list constructions and data distribution as well as communication management (for multiple compute

nodes). Those data structures can be built on GPUs with 1 to 2 order of magnitude speedup and substantially reduced algorithm overheads.

- Highly efficient heterogeneous FMM algorithms: Based on our algorithm profiling analysis, we developed the first heterogeneous FMM algorithm by distributing different FMM parts to different computing hardware to achieve the state of the art performance.

- We further extend this algorithm to several distributed versions, which are capable to solve billion size N-body on a 32-node cluster.

Publications related to this dissertation can be found in [32–37], in which [33] was a best student paper finalist of SC'11. We will elaborate details in the following chapters.

1.4 Organization of the Dissertation

In Chapter 2, we introduce the FMM parallel data structures on GPU for a single computing node. This fast data structure enables the efficient solutions of large dynamic problems. Next, in Chapter 3, we present a simulation case study using the FMM and the novel data structures on GPU with on-the-fly rendering. Based on the parallel data structure, in Chapter 4, we will look at how to implement the FMM on the heterogeneous architectures efficiently by performing translations and local direct sums to CPUs and GPUs respectively as well as how to develop a practical distributed version for very large size problems. In Chapter 5, the scalable FMM data structures for multiple computing nodes communication management are developed. We then construct a scalable distributed FMM algorithm using those structures and demonstrate the reduced

communications and overheads compared with a simple distributed FMM algorithm. In Chapter 6, we show another distributed FMM algorithm using Helmholtz decompositions for vortex methods. The performance gain of this algorithm combines improvements from the mathematics (Lamb-Helmholtz decomposition), algorithm (fast multipole method) and implementations (highly optimized MPI-CUDA code). Finally, we conclude this dissertation and discuss the future research work in Chapter 7.

Chapter 2: Parallel Algorithms for Constructing Data Structures for Fast Multipole Methods

The complexity reduction of Eq. 1.1 is because of the summation splitting and kernel decomposition. However, this factorization in Eq. 1.3 is not global, hence the split between the near- and far-fields must be managed, which requires appropriate data structures and the use of a variety of representations for the function. The efficiency with which the data structures are constructed is very important for dynamic problems since the source and receiver points change their positions at every time step.

Recall the fact that the local direct sum is independent of the far-field expansions and translations, thus may be scheduled on different computing hardware concurrently for high performance efficiency. Moreover, it is important to balance costs between these pairwise kernel sums and the hierarchical translations to achieve high computation throughput and proper scaling. Besides those algorithmic considerations, there is another vital factor to achieve such desired high efficiency: low data addressing latency. In our implementation, both translations and local direct sums have their special auxiliary *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.

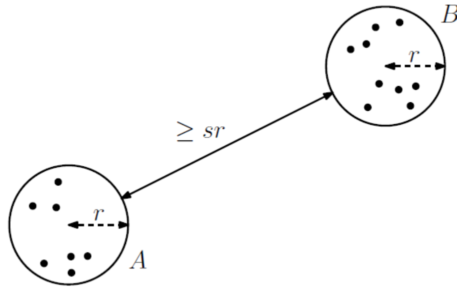


Figure 2.1: A Well Separated Pair.

2. Interaction lists for fast data addressing.

3. The communication management structures.

The construction of these data structures must be done via algorithms that have the same overall complexity with the summation. Previous work on the FMM often does not provide details of the data structures used, or their construction algorithms. A purpose of this chapter is to provide these details. Our second goal is to develop novel parallel algorithms for the data structures for both single and multiple heterogeneous nodes on GPUs.

2.1 Well-separated Pair Decomposition

The need to construct spatial data structures arise from a need to provide an error controlled translation of the FMM function representations (discussed below in Section 2.3). This is achieved by using a *well-separated pair decomposition* (WSPD) (see Fig. 2.1), which is itself useful for solving a number of other geometric problems [11, chapter 2]. In the context of FMM, given the distance between the two sphere centers d,

with radii are r_A and r_B respectively, the translation error ε (from c_A to c_B) is bounded by

$$\varepsilon(p) < C\eta^p(r_A, r_B), \quad \eta(r_A, r_B) = \frac{\max(r_A, r_B)}{d - \min(r_A, r_B)} \tag{2.1}$$

where p is the truncation number. Note that the p is determined based on the worst case, i.e., when $\eta(r_A, r_B)$ achieves its minimal value this ε still satisfies the prescribed error bound. Refer to [13, 38] for a discussion on the optimization of multi-level FMM for the very details.

2.2 Treecode and Its Data Structures

Similar to the FMM, there is also another well-known fast N-body simulation algorithm, *Barnes-Hut-Method* [6], which uses the similar spatial data structures as FMM and is often called a *treecode*. As in the FMM, the whole space is hierarchically subdivided via an octree. Each spatial box has an pseudo-particle that contains the total mass in the box located at the center of mass of all the particles it contains. Whenever force on a particle is required, the tree is traversed from the root. If a certain box is far away from that particle, the pseudo-particle is used to approximate the force induced by that box, otherwise it is subdivided again or is processed particle–by–particle directly. The complexity of treecode is in $O(N)$. However, unlike FMM, the control on the accuracy is less precise.

Treecode requires the octree structures to re-arrange spatial data, which are similar to part of our FMM data structures. In the most recent GPU treecode development [39], algorithms for the octree traversing, particle sorting and data compaction (skip empty boxes) on GPU based on the *CUDA scan* algorithm [40]. Such algorithms, are similar to

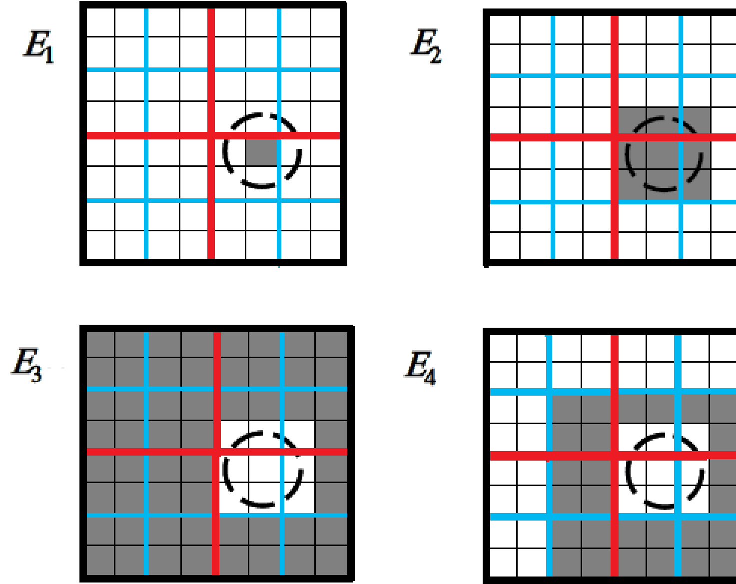


Figure 2.2: The E_1, E_2, E_3, E_4 Neighborhoods of Dimension 2. Red division at level 1; blue division at level 2; black division at level 3. The shaded box in E_1 sub-figure has its Morton index as 40 at level 2.

the approaches in Section 2.5.1 and which we first presented in [32]. The other work in the treecode space that is similar, is [41], in which a GPU-based construction of space filling curves (SFC) and octrees were presented. Although those two data sorting algorithms in [39, 41] are similar to our grid sorting (Alg. 1 described later), they are not designed in the context of FMM, and thus lack the costly constructions of neighbor interaction lists, which will described in the following sections.

2.3 Multi-Level FMM Data Structures

Assume all of the data points are already scaled into a unit cube. The WSPD is recursively performed by subdividing the cube into sub-cubes (spatial boxes) via an octree

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 sums and the far-field translations can be balanced to the extent possible). To guarantee separation of spatial data points by these sub-cubes and their minimal bounding spheres, we need to introduce several different space neighborhood domains [13]. Given each spatial cubic box with the Morton index [11,42] $n = 0, \dots, 2^{ld}$ at level $l = 0, \dots, l_{max}$ in d dimensions,

1. $E_1(n,l) \subset \mathbb{R}^d$ denotes the spatial points inside the box n at level l. We call these boxes as source or receiver box with index n at level l.

2. $E_2(n,l) \subset \mathbb{R}^d$ denotes the spatial points in the neighborhood of the box with index n at level l ("neighborhood" means all its immediate neighbor boxes). This list is used for local direct summations for $E_1(n,l)$.

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. This is the complement of $E_2(n,l)$.

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 which do not belong to the neighborhood of box n at level l. These are *interaction boxes* whose contributions are accounted for by M2L translations for $E_1(n,l)$.

Consider any box B with Morton index n at level l (see Fig. 2.2). All the translation operations are performed box by box so the source data have to be viewed as spatial boxes but not individual points. All the receiver data points inside B can not be well separated with all the source boxes inside $E_2(n,l)$. Hence $E_2(n,l)$ is used to compute the near-

field sum. Due to hierarchical translations, all the source boxes outside $E_4(n, l)$ have already been translated to B's center at the previous level. Thus only the influence of the remaining source data needs to be translated. These are located in its $E_4(n, l)$ domain, which corresponds to the most time consuming $M|L$ translation to B.

While there are several implementations of the FMM and the closely related treecode algorithms in the literature, the papers usually focus on reporting results of the summation, and do not describe the details of the FMM data structures. Our focus is on describing fast parallel algorithms for the multi-level FMM. In [38] and [13], they described those FMM related octree data structures and their implementations in details. Similar work on such hierarchical spatial data structures can be found [43] and [44]. In [32, section 5], we developed the parallel algorithms to construct FMM data structures using the GPU but only for a single computing node. Basic concepts and operations in these works are the fundamentals of our parallel GPU algorithms developed in this chapter.

In the literature, the data structure research mainly focuses on load balancing and data partition. In [45], several opportunities for parallelism in the FMM were discussed and it was shown that it is possible to apply FMM on both shared memory or distributed architectures. Compared to later work, the data distribution method in this pioneering paper was simple, perhaps not practical in many applications. In [23], an efficient parallel adaptive FMM with a "costzones" partition technique was developed based on data locality. A multi-threaded tree construction was implemented in [46]. However, in these papers the data structures were not built in parallel, i.e, the local tree of each node was built by a single processor. In [31] and [26] they separated the computation and

communication to avoid synchronization during the evaluation passes. Ref. [27] extended the work of [31] by providing a new parallel tree construction and a novel communication scheme, which scaled up to billion size problem on 65K cores. But all the GPUs were only used for kernel evaluation, i.e. direct local sum and part of translations, while the data structures alone were sequentially constructed within a single node on CPUs. In contrast, our approach provides parallel algorithms to build data structures not only on the node level, but also at a much finer granularity within a node, which allows their construction algorithms to be efficiently mapped on SIMD architectures of GPUs. There are also many other works focusing on a complementary problem: of partitioning the FMM data across multiple processors, such as [47], which shown a provably and efficient good partition as well as a load balancing algorithm, and [48], which presented a partition strategy based on pre-computed parameters. Those optimal global partition strategy could be used in our multiple node heterogeneous FMM algorithm given different application requirements.

2.4 Motivation for Fast Data Structure Algorithms

Because all the neighborhood relations are independent of FMM kernel evaluations, but only determined by the initial source and receiver locations, all computations for specifying neighborhoods can be implemented by pre-computing several *interaction lists* for all non-empty spatial boxes, which can directly address the right data of their E_2 or E_4 neighbor domains when it is needed. Hence, the major task of FMM data structure constructions is to re-arrange the source and receiver data and compute all these interaction lists efficiently with linear cost. This initial setup procedure can be treated

as a separate module in any FMM algorithm, and by Amdahl's law its cost should be consistent with the cost of the FMM kernel evaluation.

Several papers in the literature as we mentioned before ([39, 41], etc.) have been published on fast Kd-tree and octree data structures that 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. 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 [21]. For large dynamic problems (the particle positions change every time step), this data structure construction cost would dominate the overall cost by Amdahl's law, especially when the FMM kernel evaluation is significantly speeded up. Re-implementing the CPU algorithm for the GPU would not achieve the kind of acceleration we sought. The reason is that the conventional FMM data structures algorithm employs sorting of large data and operations such as set-intersection and searching, that require random access to the global memory, cannot be implemented efficiently on current GPU architectures.

2.5 Parallel FMM Data Structure Algorithm for GPU

The basic FMM data structures in our implementations are based on the octree [11, Chapter 2]. At different octree levels, the unit cube containing all the spatial points is hierarchically divided into sub-cubes via an octree and each spatial box is assigned a global *Morton index* [42]. Basic concepts and operations on the octree data structures

include: finding neighbor boxes, assigning indices and finding coordinates of the box center via interleaving/deinterleaving, particle location (box index) query, etc. Refer to [13, 38] for details of these basic concepts, operations and algorithms.

The algorithm is based on use of occupancy histograms (i.e., the counts of particles in each box), assigning particles to their grid cells, and parallel scans [49]. A disadvantage of this approach is the fact that the histogram requires temporary allocation of an array of size $8^{l_{\max}}$. 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. In this case accelerations up to two orders of magnitude compared to CPU were achieved. Note that the histogram is only needed at the time of data structure construction, all the empty box information is skipped in the final data structure outputs, which are passed to the real FMM kernel evaluation engine, to achieve both high memory and subsequent summation efficiency.

We would first like to establish some notation. First of all, all the integers in our implementation, such as box indices, histograms, are stored as *unsigned int*. We use `Src`/`Recv` to represent source points/receiver points respectively. We define non-empty source/receiver boxes as those boxes that contain at least one source/receiver data point respectively, while empty source/receiver boxes have no points inside. Note that an empty source box may contain receiver points and vice versa.

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]
```

2.5.1 Pseudo-Sort Using Fixed-Grid-Method in Linear Time

To build the FMM data structures, we first need to reorganize the data points (both source and receiver) into a tree structure according to their spatial locations, such that at the finest level each octree box holds at most a prescribed number of points, the *cluster size*. By adjusting the cluster size, we can try to ensure that the costs of the near-field direct sums and the far-field approximated sums are roughly balanced (or take the same time). Given the cluster size, we could determine the maximal level l_{max} of the octree. The data reorganization is realized by a "Fixed-Grid-Method" algorithm, in which all data points are rearranged according to their Morton box indices at level l_{max} but only with linear computation cost, since the order of data points within a box, which share the same Morton index, is irrelevant to the algorithm correctness. We do not use the word "sort" here is because this pseudo-sort is a non-deterministic algorithm. In our GPU implementation, the final sort order is determined by the run-time global memory access order of CUDA threads.

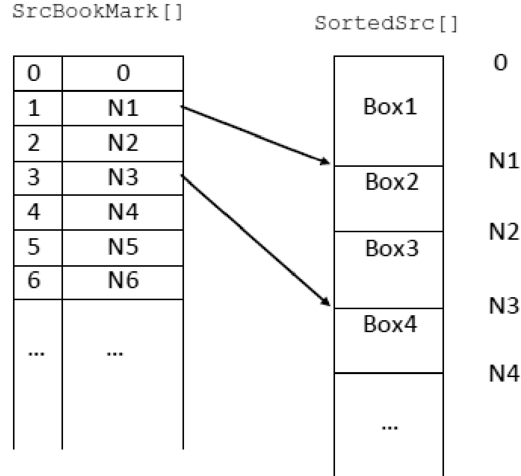


Figure 2.3: Bookmark List. Address the sorted data points by using bookmarks.

Since we have to pseudo-sort both source and receiver points, we use 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, there is a *histogram* array `Bin[]` 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 [1]. Let the number of data points be M. Then the pseudo-code 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. After this

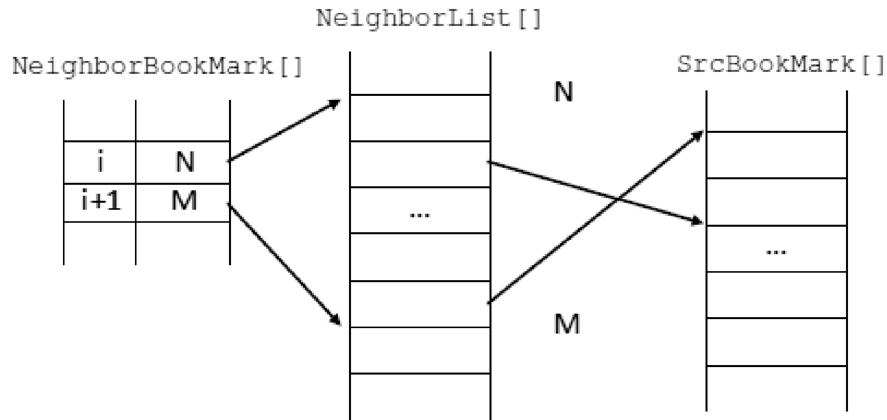


Figure 2.4: Neighbor Lists. The mapping relation among the neighbor bookmark, the neighbor list and the source bookmark

pseudo-sort, all the data points are copied into a new sorted array according to their `sortIdx` and their corresponding bookmark arrays (a pointer array described in details in Sec. 2.5.2), which is used to find the data points given a box Morton index, are constructed. Note that the cost to move data and write the pointer address into the bookmarks are also linear and moving data on the device can take advantage of high GPU memory bandwidth. We denote the pseudo-sorted source/receiver point arrays as `SortedSrc[]`/`SortedRecv[]` respectively.

2.5.2 Interaction Lists

To access data efficiently, we use several pre-computed arrays, which are also constructed by using parallel algorithms on the GPU: `SrcBookmark[]`, `RecvBookmark[]`, `NeighborList[]`, `SrcNonEmptyBoxIndex[]`, and `NeighborBookmark[]`. We call these *interaction lists*. Define `numSrcNEBox` and

numRecvNEBox be the number of non empty source and receiver boxes respectively. We describe these interaction lists below:

- SrcBookmark[]: its ith entry points to the first sorted source data point in the ith source non-empty box in SortedSrc[] Its length is numSrcNEBox $+ 1$.
- RecvBookmark[]: its jth entry points to the first sorted receiver data point in the jth receiver non-empty box in SortedRecv[]. Its length is numRecvNEBox $+ 1$.
- SrcNonEmptyBoxIndex[]: its ith entry stores the Morton index of the ith non-empty source box. Its length is numSrcNEBox.
- NeighborBookmark[]: to perform the local direct sum of the jth non-empty receiver box, its E_2 neighbor information can be retrieved from the NeighborBoo- kmark[j]th to the NeighborBookmark[j+1]-1th entries in the list Neighbo- rList[].
- NeighborList[]: given two indices i and j such that NeighborBookmark[j] $\leq i <$ NeighborBookm- ark[j+1] and denote $k = i -$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 SrcNonEmpty-BoxIndex[]. See figure 2.4.
- RecvPermutationIdx[]: its ith entry means the original position of data point SortedRecv[i] in Recv[] is Rec- vPermutationIdx[i].

Algorithm 2 ACCESS-SOURCE-E_2-NEIGHBORHOOD (`SortedSrc[]`, `SrcBookmark[]`, `NeighborList[]`, `NeighborBookmark[]`, `i`): an algorithm to extract all the source data within the E_2 neighborhood of the ith (non-empty) receiver box.

Input: the ith receiver box and other interaction lists

Output: the source data `tempSrcNei[]` within its E_2 neighborhood

```
1: count←0
2: tempSrcNei← ∅
3: B←NeighborBookmark[i+1]-1
4: for j=NeighborBookmark[i] to B do
5:     v←NeighborList[j];
6:     C←SrcBookmark[v+1]-1
7:     for k=SrcBookmark[v] to C do
8:         tempSrcNei[count++]←SortedSrc[k]
```

Given the bookmark array, the data point can be accessed directly from the sorted data list. For each receiver non-empty box, the source data points within its E_2 neighborhood can be accessed as Alg. 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 passed to the FMM kernel evaluation engine. The last auxiliary array `RecvPermutationIdx[]` is used to retrieve the input order of the original receiver data points.

2.5.3 Parallel Data Structure Construction

In our implementation, the bookmark for the source/receiver box is the rank of its first source/receiver point among all source/receiver points. The bookmark provides a pointer to the data of any non-empty box among all boxes without search. A reduction operation is needed to compute the entries of the bookmark arrays. The highly efficient parallel *prefix sum* (or scan) [40] is used in our implementation. Given the `Bin[]` obtained from Alg. 1, the `Bookmark[]` can be computed by removing the repeated elements (corresponding to empty boxes) in the prefix sum of `Bin[]` using Alg. 3. The same idea can also be used to address any non-empty source/receiver box among all source/receiver boxes if we mark non-empty boxes by 1 and empty boxes by 0 and apply the scan operation. With `Bookmark[]` and `SortIdx[]`, data points are copied to into a new sorted list. `SrcNonEmptyBoxIndex[]` is used to construct `NeighborBookmark[]` and `NeighborList[]` in parallel as Alg. 5: initially a thread computes the E_2 neighbor box indices of a non-empty receiver box

Algorithm 3 GET-BOOKMARK-AND-BOX-INDEX (`Bin[]`): an algorithm to compute the bookmark and the non-empty box index of source/receiver boxes.

Input: the pseudo-sorted index array `Bin[]` of source/receiver boxes

Output: the bookmark array `Bookmark[]` and the Morton index array `NonEmptyIdx[]` of source/receiver boxes ▷ *array indices depend on implementations*

1: perform parallel scan on `Bin[]` to obtain its prefix sum `ScannedBin[]`

2: **for** `i=0` to `Bin[].length-1` **parallel do**

3:     **if** `Bin[i]>0` **then**

4:         `Rank[i]←1`

5:     **else**

6:         `Rank[i]←0`

7: perform parallel scan on `Rank[]` to obtain its prefix sum `ScannedRank[]`

8: allocate memory for `Bookmark[]` and `NonEmptyIdx[]` ▷ *their lengths can be derived from* `Bin[]`

9: `Bookmark[0]←0`

10: `Bin[-1]←0`

11: **for** `i=0` to `Bin[].length-1` **parallel do**

12:     **if** `Bin[i]>Bin[i-1]` **then**

13:         `Bookmark[ScannedRank[i]]←Bin[i]`

14:         `NonEmptyIdx[ScannedRank[i]]←i`

Algorithm 4 GET-E_2-NEIGHBOR-LIST-AND-BOOKMARK(`Scanned- Rank[]`, `RecvNonEmptyBoxIdx[]`, `numNonEmptyRecvBox`): an algorithm to extract the E_2 neighborhood for all the (non-empty) receiver boxes.

Input: the `ScannedRank[]` from Alg. 3 for source, the receiver box index array `RecvNonEmptyBoxIdx[]` with its length `numNonEmptyRecvBox`

Output: The E_2 neighbor list array `NeighborList[]` (for receiver boxes) and its bookmark `NeighborBookmark[]`

1: allocate a temporary array `RecvE2NeiNEBoxIdx[]` to store neighbor box indices ▷ *each box can have 27 neighbors at most in 3D*

2: **for** `i=0` to `numNonEmptyRecvBox-1` **parallel do**

3: n_i ←`0`

4: `k`←`RecvNonEmptyBoxIdx[i]`

5: **for** all its non-empty E_2 source neighbor box j **do**

6: `NeighborIdx[27i+(`n_i`++)]`← j

7: `NumRecvE2NeiNEBox[i]`← n_i

8: **for** `j=0` to $n_i - 1$ **do**

9: `RecvE2NeiNEBoxIdx[27i+j]=ScannedRank[NeighborIdx[j]-1]`

Algorithm 5 GET-E_2-NEIGHBOR-LIST-AND-BOOKMARK (CONTINUED)

10: perform parallel scan on `NumRecvE2NeiNEBox[]` to obtain its prefix sum `NeighborBookmark[]`

11: allocate `NeighborList[]` ▷ *itself and its length can be derived from `RecvE2NeiNEBoxIdx[]` and `NeighborBookmark[]` respectively*

12: **for** `i=0` to `numNonEmptyRecvBox-1` **parallel do**

13: **for** `j=0` to `NumRecvE2NeiNEBox[i]-1` **do**

14: `count ← NeighborBookmark[i]+j`

15: `NeighborList[count]←RecvE2NeiNEBoxIdx[27i+j]`

and checks whether these source neighbor boxes are empty or not. Then this thread increases the local non-empty source box count accordingly for its assigned receiver box and store the neighbor indices temporarily. Finally after another parallel scan call, the temporary neighbor indices are compressed and written to `NeighborList[]`, where the target address is obtained by reading the non-empty source box index from `SrcNonEmptyBoxIndex[]`. Algorithm 6 summarizes all the steps to build the data structures for a single computing node.

All the octree operations needed in Alg. 6, can be found in [38]. By using the *interleave* and *deinterleave* operations, we can derive a 3D coordinate for any given Morton index. Given this 3D vector, we can increase or decrease its coordinate component to compute its neighbors' 3D coordinates. Therefore, the algorithms of E_2 and E_4 neighbor queries can be easily obtained. Accordingly, they are not presented as separate algorithms. Note that, given any spatial box, the computations of its neighbors' coordinates and Morton indices are independent of other boxes and executed in parallel.

Algorithm 6 BUILD-FMM-DATA-STRUCTURES (`Src[], Recv[]`): the single-node algorithm to pseudo-sort data points and construct all the needed interaction lists on GPU.

Input: the source/receiver data `Src[]`/`Recv[]`

Output: all interaction lists and the pseudo-sorted source/receiver data `SortedSrc[]`/`SortedRecv[]`

1: pseudo-sort `Src[]` by Alg. 1

2: get `SrcNonEmptyBoxIndex[]` and `SrcBookmark[]` by Alg. 3

3: copy sorted source data points to `SortedSrc[]`

4: pseudo-sort `Recv[]` by Alg. 1

5: get `RecvPermutationIdx[]` and `RecvBookmark[]` by Alg. 3

6: copy sorted receiver data points to `SortedRecv[]`;

7: build `NeighborBookmark[]` and `NeighborList[]` by Alg. 2 and Alg. 5

8: pass `SortedSrc[]`, `SrcBookmark[]`, `SortedRecv[]`, `RecvBookmark[]`, `RecvPermutationIdx[]`, `NeighborBookmark[]` and `NeighborList[]` to FMM kernel evaluation engine;

9: free all other allocated device memory;

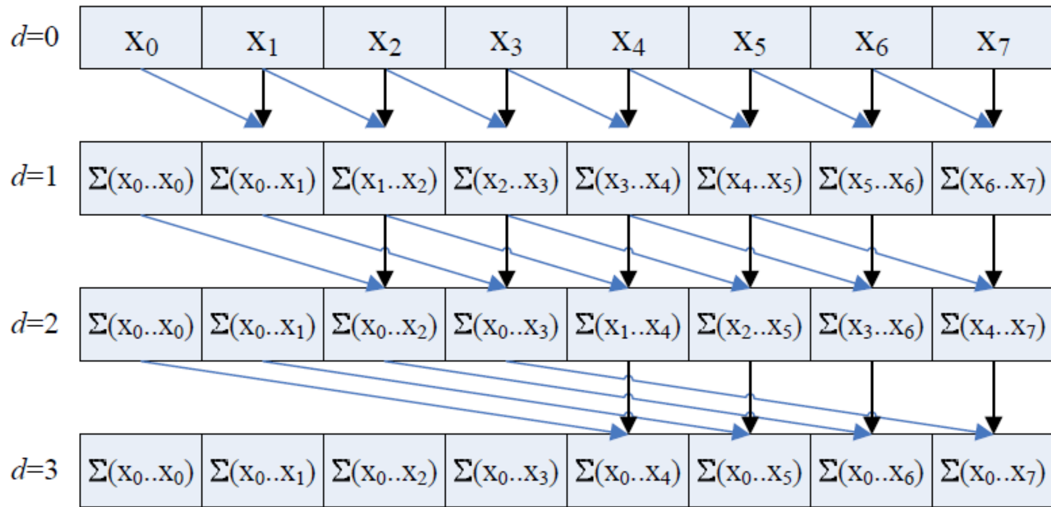


Figure 2.5: The Naïve Prefix Sum (Scan) Algorithm.

2.5.4 GPU Implementation Considerations

Basic octree operations, such as box index query, box center query, box index interleave/deinterleave and parent/children query, and more complex neighbor query operations, such as E_2 and E_4 neighbor index query, are all implemented as inline CUDA `__device__` functions. For efficiency, we minimize the use of global memory and local memory accessing. Once input data is loaded into these device functions, we only use local fast registers, or coalesced local memory if data can not fit into registers, to store intermediate results. Moreover, we manually unroll many loops to further optimize the code. Results shows that even for the costly computation of E_4 neighbors, its total running time can be neglected in comparison with the kernel evaluation time in the FMM. When we perform the FMM translation part on GPU, only these basic operations are used, in which all other necessary translation data structures are constructed on the fly. Refer to

section 2.6 for the experiment results.

2.5.5 Complexity

The complexity of these data structure algorithms is determined by the number of source points N, the number of receiver points M and the maximal octree level l_{max}. Since we use histograms, we can avoid all the searching operations on the device, which makes our implementation fast and efficient. However, there is a memory consumption trade-off for the processing speed since the size of histogram increases exponentially as l_{max}. For the bucket sort Alg. 1, its complexity is linear $O(N + M)$. All other algorithms are related to the octree boxes, which total number is $N_{box} = 2^{3l} = 8^l$. Since we use the canonical scan algorithm (see Fig. 2.5), Alg. 2 to Alg. 5 are in $O(N_{box} \log N_{box} + N + M) \sim O(l8^l + N + M)$. If we interpret the maximal level l_{max} as a prescribed constant, then our parallel data structure construction Alg. 6 for single node is linear with respect to particle size, i.e. in $O(N + M)$.

2.6 Experimental Results

To test the data structure performance for a single node, we fix the problem size to 1 million and use the uniform distributed source and receiver. Here the source and receiver points are different. The computation hardware used here are: NVIDIA GTX480 GPU and Intel Xeon X5560 quad-core CPU running at 2.8GHz.

We firstly test the performance on the uniformly distributed data in a unit cube. Note that, this would be most time consuming case since almost all the spacial boxes

$l_{\max}$	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 2.1: The Time Comparison of FMM Data Structure Computation. Tests are performed on 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.

are non-empty. The timing results are summarized in Table 2.1, in which 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.

In the second experiment, we generate all the source and receiver data on a sphere surface and test how the algorithm scales and the performance gain. In Figure 2.6, we show both the CPU and GPU time across the number of data points, which ranges from

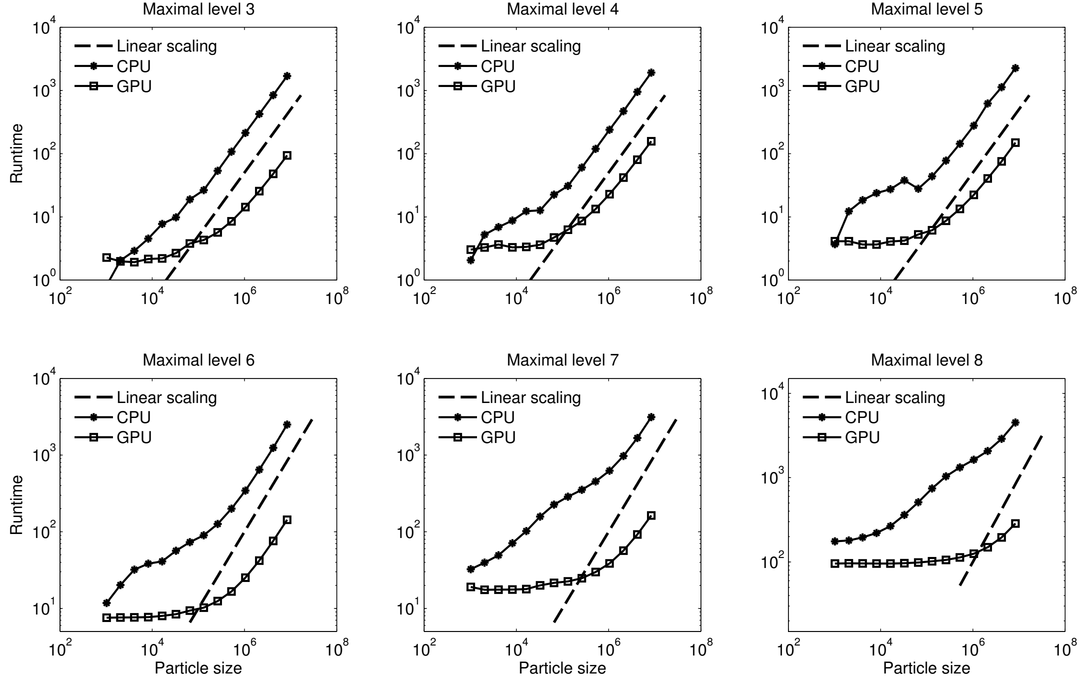


Figure 2.6: The Data Structure Construction Time for Non-Uniform Distribution. All the source and receiver are distributed on a sphere. Each sub-figure corresponds to a maximal level setting. All the tests are performed on a single node using one GPU.

1024 up to 8 millions, for different octree maximal levels. As the l_{max} increases, there are more spatial boxes occupied which *super linearly* increases the overall costs. However, once the number boxes become relative stable, i.e., increasing the number of particles only changes the number of spatial boxes a little bit, the overall cost increases linearly. This is because that all the boxes related constructions is more or less the same as a constant and the particle related computation, such as bit interleaving and the fixed-grid-method pseudo-sort that linearly scales as the amount of particle data, now dominates the overall costs. In a whole, for this non-uniform distribution, our data structure algorithms also demonstrate their linear complexity and our fast parallel implementations can achieve

15-20 times speed-ups against the CPU performance.

As a conclusion of these tests on a single node, the FMM data structure step is reduced to a small part of the computation time again, which provides substantial overhead reductions and makes our algorithm suitable to solve dynamic problems.

2.7 Summary

Our algorithm makes generation of data structures on GPUs very efficient, which are based on the use of occupancy maps, bin sorting, and parallel scans. Comparison of the GPU and CPU times for the same algorithm show accelerations in the range 20-100 times. This shows the feasibility of the use of GPUs for data structure construction, which satisfyingly reduce the data-structure step to a small part of the FMM overall computation time.

For the single-node FMM, we are able to device a new algorithm, which also has the advantage that it achieves the FMM data structure in $O(N)$ time, bringing the overall complexity of the FMM to this level for a given accuracy. Comparison of the GPU and CPU times for the same algorithm show accelerations in the range 20–100 times. This shows the feasibility of the use of GPUs for data structure construction, which satisfyingly reduce the data-structure step to a small part of the FMM overall computation time.

Chapter 3: GPU Accelerated Fast Multipole Methods for Dynamic N-Body Simulation

The fast multipole method can be used to accelerate N-body simulations and matrix vector products arising in various applications. These include fluid simulations [50,51] as well as in scientific computing: in fitting implicit functions to point based representations using radial-basis functions [52,53]; in radiosity computations [54] 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, [3] developed a fast GPU-based parallel implementation, however, its computation complexity is still quadratic. Recent work on particle methods using direct

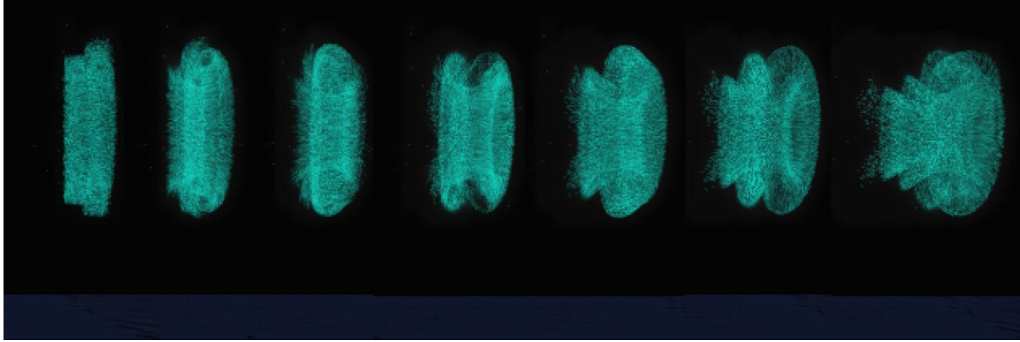


Figure 3.1: Vortex Rings and Particles Interaction.

computation has been shown in [55].

We use FMM as an alternative way to solve such N-body problems based on particle methods. 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.

To simulate interactions between vortices and particles, we apply particle methods with the FMM. In [21, 26, 48], different GPU-based FMM implementations were developed. Particularly, Ref. [56] compared the performances between *treecode* and FMM on GPUs for a similar leapfrogging vortex ring simulations. However, in those implementations, all the data structures need by FMM were built on the CPU, which is too expensive for dynamic problems, where particle locations change every step. Ref. [57] developed a CPU-GPU-Hybrid treecode to accelerate the computation, but its overall performance does not outperform the implementation presented in [21], while

Ref. [58] discussed the auto-tuning techniques of N-body simulations on heterogeneous systems. In this chapter, we use the data structure on the GPU presented in Chapter 2 and fully GPU-based FMM to simulate dynamic N-body problems 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 [52] [53]. 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.

3.1 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 (refer to [59] for details) 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

$$\mathbf{v}(\mathbf{y}) = \sum_{i=1}^{N} \mathbf{v}_i(\mathbf{y}), \quad \mathbf{v}_i(\mathbf{y}) = \frac{\boldsymbol{\omega}_i \times (\mathbf{y} - \mathbf{x}_i)}{|\mathbf{y} - \mathbf{x}_i|^3} = \nabla \times \frac{\boldsymbol{\omega}_i}{|\mathbf{y} - \mathbf{x}_i|}. \tag{3.1}$$

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

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}|_{\mathbf{x}=\mathbf{x}_i}, \quad \frac{d\boldsymbol{\omega}_i}{dt} = \boldsymbol{\omega}_i \cdot \nabla\mathbf{v}|_{\mathbf{v}=\mathbf{v}_i}, \quad \mathbf{v}(\mathbf{x}_i; t) = \sum_{j \neq i} \mathbf{v}_j(\mathbf{x}_i; t). \tag{3.2}$$

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. The velocity field in Eq. 3.1 can

also be modified by using a smoothing kernel $K(|\mathbf{y} - \mathbf{x}_i; a)$ as

$$\mathbf{v}_i(\mathbf{y}; a) = \frac{\boldsymbol{\omega}_i \times (\mathbf{y} - \mathbf{x}_i)}{|\mathbf{y} - \mathbf{x}_i|^3} K(|\mathbf{y} - \mathbf{x}_i|; a), \tag{3.3}$$

where a is the radius of the vortex core and the smoothing kernel only has effect in the near field of $\mathbf{x}_i$. As for the particles, their induced velocities are determined by the vortex elements which is also described by Eq. 3.1. 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 $O(n^2 + nm)$.

3.2 GPU-Based Fast Multipole Method

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 [21]. Recall the fact that re-implementing 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.

First, we generate the FMM data structure on the GPU in $O(N)$ time as in Chapter 2, bringing the overall complexity of the FMM to $O(N)$ for a given accuracy. Second, we determine the interacting source boxes in the neighborhood of the receiver boxes. In a whole, we rely on the histograms and parallel scan to achieve parallel processing goal. This technique enables fast neighbor determination **without sort, search, or set intersection operations.** Refer to Chapter 2 or [36] for the details of

interaction lists and their parallel constructing algorithms.

3.2.1 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 [21, (12)]) by

$$\widetilde{B}_n^m(\mathbf{r}) = \begin{cases} \text{Re}\{B_n^m\}, m \geq 0, \\ \text{Im}\{B_n^m\}, m < 0. \end{cases} \tag{3.4}$$

It is already known that the basic kernel function

$$\Phi(\mathbf{r}) = \sum_{n=0}^{p} \sum_{m=-n}^{n} c_n^m B_n^m(\mathbf{r}) \tag{3.5}$$

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

$$\Phi_n(\mathbf{r}) = \sum_{m=-n}^{n} c_n^m B_n^m(\mathbf{r}) = \widetilde{\Phi}_n(\mathbf{r}) = \sum_{m=-n}^{n} d_n^m \widetilde{B}_n^m(\mathbf{r}). \tag{3.6}$$

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

$$d_n^{-m} = c_n^{-m} + c_n^m, \quad d_n^m = \mathbf{i}(c_n^{-m} - c_n^m). \tag{3.7}$$

The elementary solutions of the Laplace equations in 3D are

$$R_n^m(\mathbf{r}) = \alpha_n^m r^n Y_n^m(\theta,\varphi), \;\; S_n^m(\mathbf{r}) = \beta_n^m r^{-n-1} Y_n^m(\theta,\varphi), \tag{3.8}$$

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

$$\begin{aligned} \alpha_n^m &= (-1)^n \sqrt{4\pi/[(2n+1)(n-m)!(n+m)!]} \\ \beta_n^m &= \sqrt{4\pi(n-m)!(n+m)!/(2n+1)}, \end{aligned} \tag{3.9}$$

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

$$\Phi(\mathbf{r},\mathbf{r}_*) = \frac{1}{|\mathbf{r}-\mathbf{r}_*|} = \sum_{n=0}^{+\infty} \sum_{m=-n}^{n} (-1)^n R_n^{-m}(\mathbf{r}_*) S_n^m(\mathbf{r}). \tag{3.10}$$

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)

$$\Phi(\mathbf{r},\mathbf{r}_*) = \sum_{n=0}^{p} \sum_{m=-n}^{n} (-1)^n d_n^{-m}(\mathbf{r}_*) \widetilde{S}_n^m(\mathbf{r}) + Err_t. \tag{3.11}$$

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

$$\begin{aligned} & d_0^0 = 1, \;\; d_1^1 = -\frac{1}{2}x, \;\; d_1^{-1} = \frac{1}{2}y, \\ & d_{|m|}^{|m|} = -\frac{x d_{|m|-1}^{|m|-1} + y d_{|m|-1}^{-|m|+1}}{2|m|}, \;\; m = 2,3,\dots, \\ & d_{|m|}^{-|m|} = \frac{y d_{|m|-1}^{|m|-1} - x d_{|m|-1}^{-|m|+1}}{2|m|}, \;\; m = 2,3,\dots, \\ & d_{|m|+1}^{m} = -z d_{|m|}^{m}, \qquad\qquad m = 0, \pm 1, \pm 2, \dots, \\ & d_n^m = -\frac{(2n-1) z d_{n-1}^m + r^2 d_{n-2}^m}{n^2 - m^2}, \quad \begin{array}{l} n = |m|+2, |m|+3, \dots, \\ m = -n, \dots, n. \end{array} \end{aligned} \tag{3.12}$$

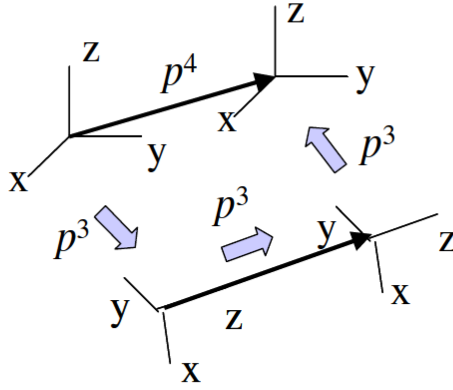


Figure 3.2: RCR Translation for the Fast Multipole Method. It replaces one $O(p^4)$ translation with two $O(p^3)$ rotations and one $O(p^3)$ coaxial translation.

Our implementation uses the $o(p^3)$ RCR decomposition [60] (Fig. 3.2)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 [14]. 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 presentation 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

$$
\begin{aligned}
\widehat{d}_n^{-m} &= \sin(m\alpha)d_n^m + \cos(m\alpha)d_n^{-m}, \ \ m = 1, \ldots, n. \\
\widehat{d}_n^{m} &= \cos(m\alpha)d_n^m - \sin(m\alpha)d_n^{-m}, \ \ m = 1, \ldots, n.
\end{aligned} \tag{3.13}
$$

For β-rotation, let

$$
\begin{aligned}
f_n^m &= 1/2\sqrt{(n-m)(n+m+1)}, \ \ m = 0, 1, \ldots, n. \\
f_n^{-m} &= 1/2\sqrt{(n+m)(n-m+1)}, \ \ m = 1, \ldots, n.
\end{aligned} \tag{3.14}
$$

$$
H_n^{m',0}(\beta) = (-1)^{m'}\sqrt{\frac{(n-|m'|)!}{(n+|m'|)!}}\mathbf{P}_n^{|m'|}(\cos\beta) \tag{3.15}
$$

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:

$$f_n^{m-1} H_n^{m-1,m'} - f_n^m H_n^{m+1,m'} = f_n^{m'-1} H_n^{m,m'-1} - f_n^{m'} H_n^{m,m'+1}. \tag{3.16}$$

Then, the β rotation can be computed by:

$$\begin{aligned}
\widehat{d}_n^{-m} &= \sum_{m'=1}^{n} d_n^{-m'} (H_n^{-m,-m'} - H_n^{-m,m'}), \quad m = 1, \ldots, n. \\
\widehat{d}_n^{m} &= \sum_{m'=0}^{n} d_n^{m'} (H_n^{m,m'} + H_n^{m,-m'}), \quad m = 1, \ldots, n. \\
\widehat{d}_n^{0} &= \frac{1}{2} \sum_{m'=0}^{n} d_n^{m'} (H_n^{0,-m'} + H_n^{0,m'}).
\end{aligned} \tag{3.17}$$

Finally, for the *coaxial translation* [14, (27)], the only modification is to change the sign for different m as

$$\widehat{d}_n^{m} = (-1)^m \sum_{n'=|m|}^{n} (S|R)_{n,n'}^m (t) d_{n'}^m. \tag{3.18}$$

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

3.2.2 Adaption to the Biot-Savart 3D Kernel

The baseline FMM computes the Coulomb kernel Phi defined by Eq. 3.19, which is defined as

$$\Phi(\mathbf{y}, \mathbf{x}) = \begin{cases} \dfrac{1}{|\mathbf{y} - \mathbf{x}|}, & \text{if } \mathbf{x} \neq \mathbf{y}, \\ 0, & \text{if } \mathbf{x} = \mathbf{y}. \end{cases} \tag{3.19}$$

So the possibly minimal modifications are preferred to adapt to the Biot-Savart kernel by Eq. 3.1 based on the baseline codes. Notice that

$$\begin{aligned}\nabla_{\mathbf{y}} \times \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|} &= (\nabla_{\mathbf{y}} \frac{1}{|\mathbf{y}-\mathbf{x}_i|}) \times \mathbf{q}_i \\ &= -(\frac{\mathbf{y}-\mathbf{x}_i}{|\mathbf{y}-\mathbf{x}_i|^3}) \times \mathbf{q}_i \\ &= \frac{\mathbf{q}_i \times (\mathbf{y}-\mathbf{x}_i)}{|\mathbf{y}-\mathbf{x}_i|^3}.\end{aligned} \tag{3.20}$$

So rewrite Eq. 3.1 using Eq. 3.20 as

$$V(\mathbf{y}) = \sum_{i=1}^{n} \frac{\mathbf{q}_i \times (\mathbf{y}-\mathbf{x}_i)}{|\mathbf{y}-\mathbf{x}_i|^3} = \sum_{i=1}^{n} \nabla_{\mathbf{y}} \times \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|}. \tag{3.21}$$

Based on Eq. , 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:

$$\begin{aligned}\{d_n^{(x),m}\} : \quad \sum_{i \notin \Omega_{\mathbf{c}}} \frac{q_i^{(x)}}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} d_n^{(x),m} R_n^m(\mathbf{y}-\mathbf{y_c}), \\ \{d_n^{(y),m}\} : \quad \sum_{i \notin \Omega_{\mathbf{c}}} \frac{q_i^{(y)}}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} d_n^{(y),m} R_n^m(\mathbf{y}-\mathbf{y_c}), \\ \{d_n^{(z),m}\} : \quad \sum_{i \notin \Omega_{\mathbf{c}}} \frac{q_i^{(z)}}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} d_n^{(z),m} R_n^m(\mathbf{y}-\mathbf{y_c}),\end{aligned} \tag{3.22}$$

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

$$\{\mathbf{d}_n^m\} : \sum_{i \notin \Omega_{\mathbf{c}}} \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|} = \sum_{n=0}^{p} \sum_{m=-n}^{n} \mathbf{d}_n^m R_n^m(\mathbf{y}-\mathbf{y_c}) \tag{3.23}$$

Therefore,

$$\begin{aligned}\nabla_{\mathbf{y}} \times \sum_{i \notin \Omega_{\mathbf{c}}} \frac{\mathbf{q}_i}{|\mathbf{y}-\mathbf{x}_i|} &= \sum_{n=0}^{p} \sum_{m=-n}^{n} \nabla_{\mathbf{y}} \times [\mathbf{d}_n^m R_n^m(\mathbf{y}-\mathbf{y_c})] \\ &= \sum_{n=0}^{p} \sum_{m=-n}^{n} \nabla_{\mathbf{y}} R_n^m(\mathbf{y}-\mathbf{y_c}) \times \mathbf{d}_n^m.\end{aligned} \tag{3.24}$$

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 (3.24). However, by differentiating Eq. 3.30 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.

3.2.3 The Recurrence Relations for Gradients

Denote

$$d_n^{(x),m} = \frac{\partial d_n^m}{\partial x}, \quad d_n^{(y),m} = \frac{\partial d_n^m}{\partial y}, \quad d_n^{(z),m} = \frac{\partial d_n^m}{\partial z}. \tag{3.25}$$

By skipping all the derivation details and using the same notations as Sec. 3.2.2 the recurrence relations of gradient coefficients in Eq. 3.22 are given by:

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

$$
\begin{aligned}
d_{|m|}^{(x),|m|} &= -\frac{d_{|m|-1}^{|m|-1}}{2|m|} - \frac{x d_{|m|-1}^{(x),|m|-1} + y d_{|m|-1}^{(x),-|m|+1}}{2|m|}, \ |m| = 2, 3, \ldots, \\
d_{|m|}^{(x),-|m|} &= -\frac{d_{-|m|+1}^{|m|-1}}{2|m|} + \frac{y d_{|m|-1}^{(x),|m|-1} - x d_{|m|-1}^{(x),-|m|+1}}{2|m|}, \ |m| = 2, 3, \ldots, \\
d_{|m|+1}^{(x),m} &= -z d_{|m|}^{(x),m}, \qquad m = 0, \pm 1, \pm 2, \ldots, \\
d_n^{(x),m} &= -\frac{2x d_{n-2}^{m}}{n^2 - m^2} - \frac{(2n-1) z d_{n-1}^{(x),m} + r^2 d_{n-2}^{(x),m}}{n^2 - m^2}, \\
& \quad n = |m| + 2, |m| + 3, \ldots, \\
& \quad m = -n, \ldots, n.
\end{aligned}
\tag{3.26}
$$

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

$$
\begin{aligned}
d_{|m|}^{(y),|m|} &= -\frac{d_{-|m|+1}^{|m|-1}}{2|m|} - \frac{x d_{|m|-1}^{(y),|m|-1} + y d_{|m|-1}^{(y),-|m|+1}}{2|m|}, \ |m| = 2, 3, \ldots, \\
d_{|m|}^{(y),-|m|} &= -\frac{d_{|m|-1}^{|m|-1}}{2|m|} + \frac{y d_{|m|-1}^{(y),|m|-1} - x d_{|m|-1}^{(y),-|m|+1}}{2|m|}, \ |m| = 2, 3, \ldots, \\
d_{|m|+1}^{(y),m} &= -z d_{|m|}^{(y),m}, \qquad m = 0, \pm 1, \pm 2, \ldots, \\
d_n^{(x),m} &= -\frac{2y d_{n-2}^{m}}{n^2 - m^2} - \frac{(2n-1) z d_{n-1}^{(y),m} + r^2 d_{n-2}^{(y),m}}{n^2 - m^2}, \\
& \quad n = |m| + 2, |m| + 3, \ldots, \\
& \quad m = -n, \ldots, n.
\end{aligned}
\tag{3.27}
$$

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

$$
\begin{aligned}
d_{|m|}^{(z),|m|} &= -\frac{x d_{|m|-1}^{(z),|m|-1} + y d_{|m|-1}^{(z),-|m|+1}}{2|m|}, \ |m| = 2, 3, \ldots, \\
d_{|m|}^{(z),-|m|} &= +\frac{y d_{|m|-1}^{(z),|m|-1} - x d_{|m|-1}^{(z),-|m|+1}}{2|m|}, \ |m| = 2, 3, \ldots, \\
d_{|m|+1}^{(z),m} &= -z d_{|m|}^{(z),m} - z d_{|m|}^{(z),m}, \qquad m = 0, \pm 1, \pm 2, \ldots, \\
d_n^{(z),m} &= -\frac{(2n-1) d_{n-1}^{m} + 2z d_{n-2}^{m}}{n^2 - m^2} \\
& \quad -\frac{(2n-1) z d_{n-1}^{(z),m} + r^2 d_{n-2}^{(z),m}}{n^2 - m^2}, \\
& \quad n = |m| + 2, |m| + 3, \ldots, \\
& \quad m = -n, \ldots, n.
\end{aligned}
\tag{3.28}
$$

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 3.1: 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.

3.2.4 Test and Error Analysis

As mentioned in Sec. 3.2.2, 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

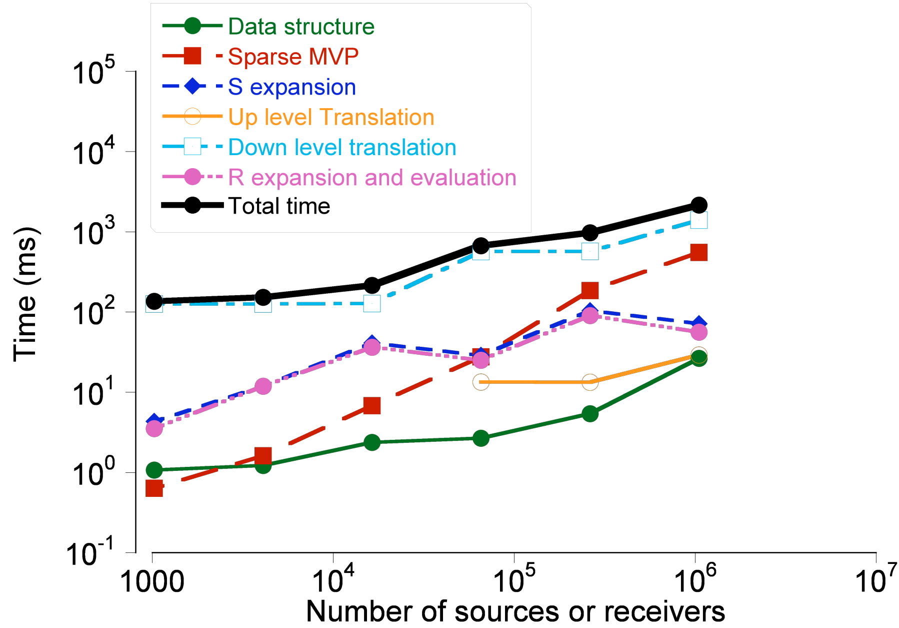


Figure 3.3: The Profiling of FMM for Biot-Savart Kernel.

Table 3.1) 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. 3.3. 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. 3.4. 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. 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

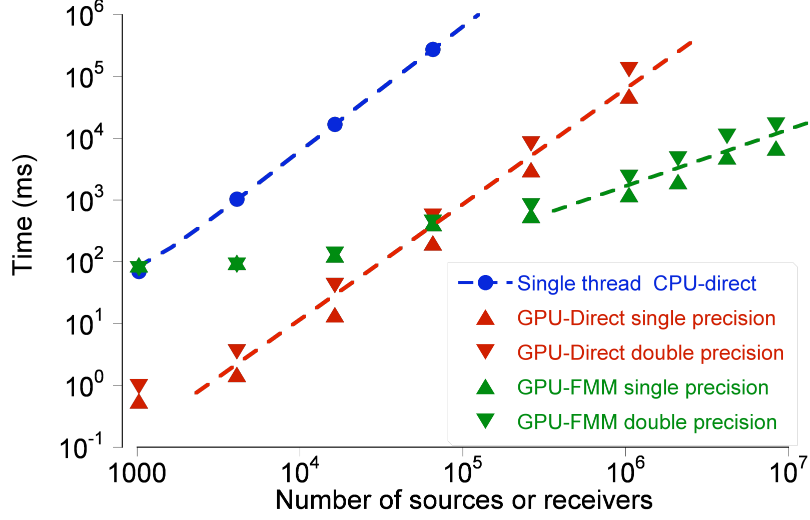


Figure 3.4: The Time Comparisons for Coulomb Kernel Between GPU-Based FMM and Direct Methods.

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 [13]. In this experiment, we will validate our GPU implementation satisfy the accuracy requirement controlled by the truncation number. The relative error is defined as

$$\epsilon = \sqrt{\frac{\sum_{j=1}^{k} |\phi_{exact}(\mathbf{y}_j) - \phi_{approx}(\mathbf{y}_j)|^2}{\sum_{j=1}^{k} |\phi_{exact}(\mathbf{y}_j)|^2}} \tag{3.29}$$

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. In Fig. 3.5, we show the relative errors on both single and double precision 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

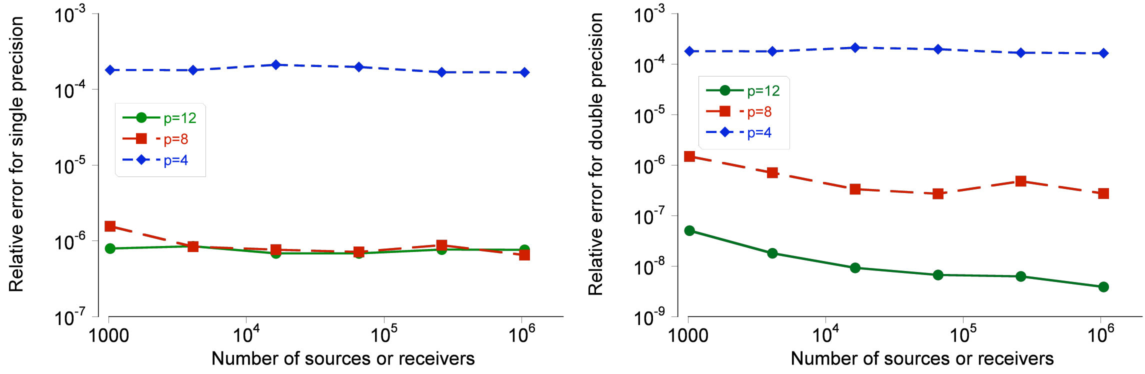


Figure 3.5: Relative Error Analysis. The relative errors on both single (on the left) and double (on the right) precision for Coulomb kernel.

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. 3.5 shows no accuracy improvement from $p = 8$ to $p = 12$. However, for the double precision, the accuracy of can be improved by adding extra multipole expansion terms.

3.3 Vortex Ring and Particle Interaction Simulation

The mathematical model to simulate vortex ring and particle interactions is based on vortex particle method [61]. In [50], 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.

3.3.1 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

$$V(\mathbf{y}) = \sum_{i=1}^{n} \frac{\mathbf{q}_i \times (\mathbf{y} - \mathbf{x}_i)}{|\mathbf{y} - \mathbf{x}_i|^3} K(|\mathbf{y} - \mathbf{x}_i|, \varepsilon), \tag{3.30}$$

Mathematically, the smoothing kernel is used to preserve the algorithm propagation stability. Physically, it corresponds to the vortex blob cut-off. Different smoothing kernel implementations are discussed in [61] and an algebraic implementation, where

$$K(d, \varepsilon) = \begin{cases} \dfrac{d^2}{\varepsilon^2} & \text{if} \quad d \leq \varepsilon, \\ 1 & \text{if} \quad d > \varepsilon. \end{cases} \tag{3.31}$$

is used in this chapter because it is fast to compute on the GPU while it has little effect on the results. 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 chapter used the Euler method.

3.3.2 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 the GPU (without data transfer between GPU and CPU) by the CUDA OpenGL *interoperability* [1].

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*

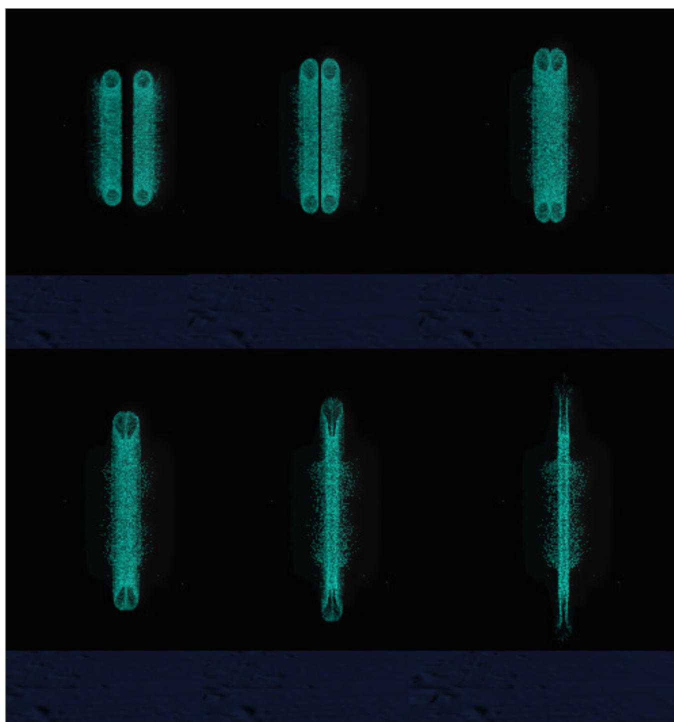


Figure 3.6: Collision of Two Vortex Rings.

[62, 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. 2.5.1 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.

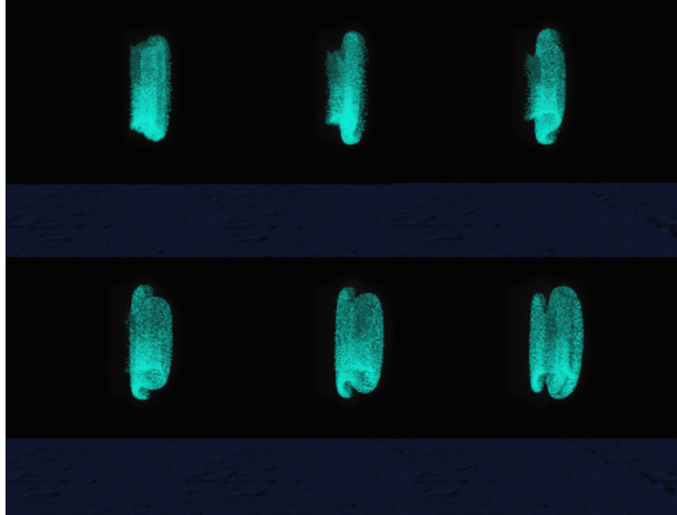


Figure 3.7: The Leap-Frog of Two Rings with 2^{15} Particles Rendered.

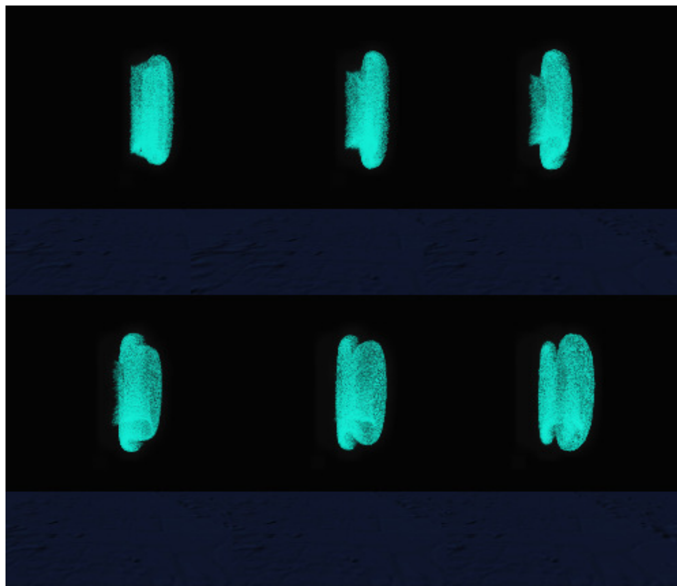


Figure 3.8: The Leap-Frog of Two Rings with 2^{18} Particles Rendered.

3.3.3 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

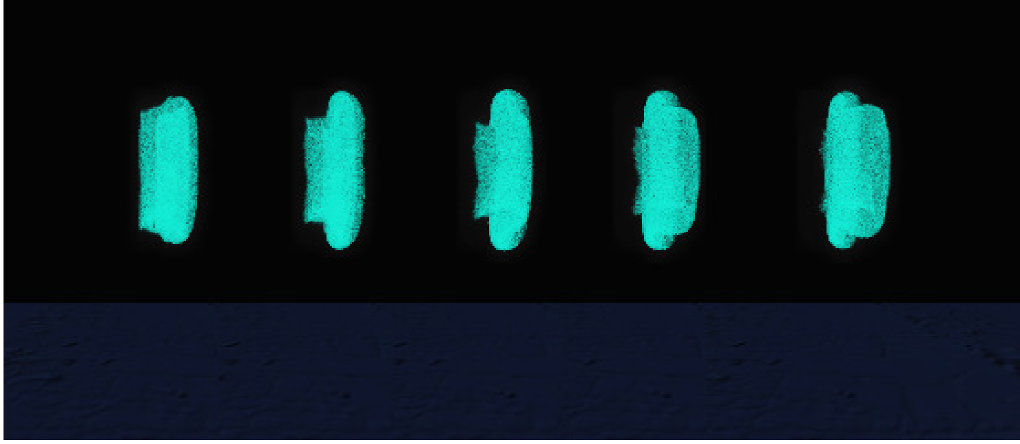


Figure 3.9: The Leap-Frog of Two Rings with 2^{20} Particles Rendered.

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 3.1 and Fig. 3.6 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 3.1 shown the leap-frog of two vortex rings while Fig. 3.6 shown the collision of two rings. In Fig. 3.7 and Fig. 3.8, 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. 3.9, 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. 3.2.4. But the total running time for each frame is still around $1.4 \sim 2.5$ seconds on double precision data

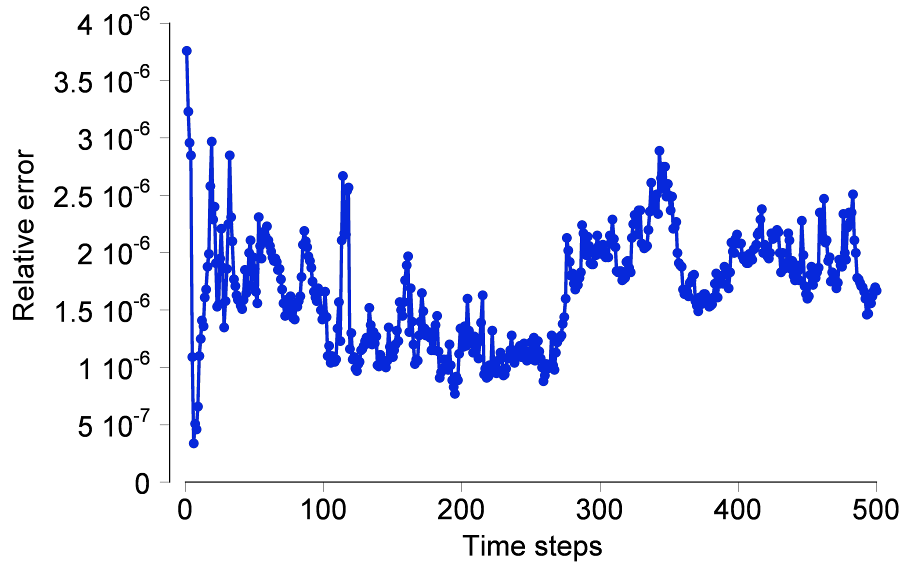


Figure 3.10: The Accuracy of Vortex Ring Simulation Using FMM for Each Time Step.

along the whole simulation process.

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 vertical 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. 3.11. For each time step, it was found to take around 1.2–2.5 seconds for the computation in which both FMM

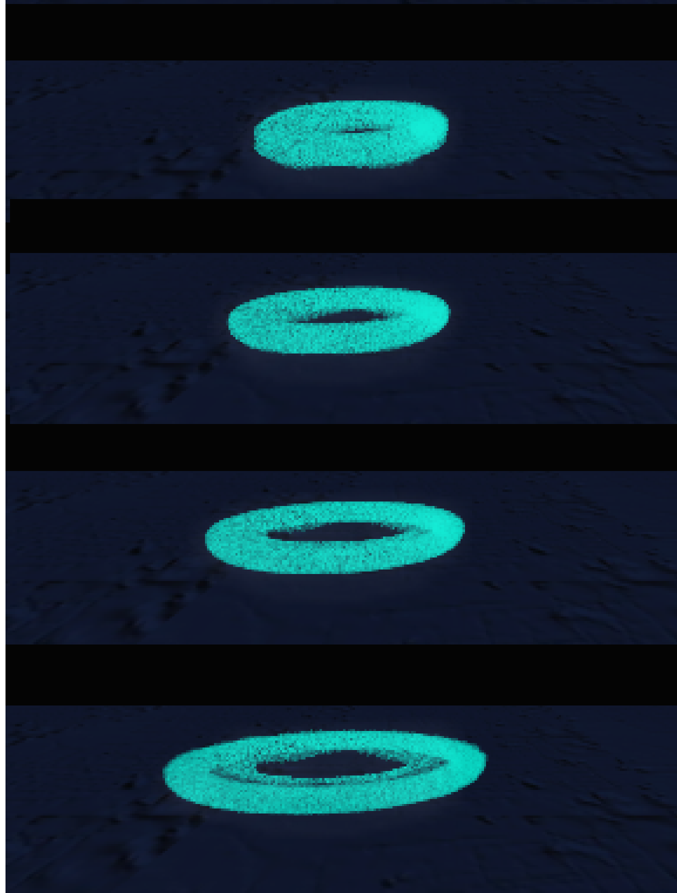


Figure 3.11: The Simulation Snapshots of A Vortex Ring Colliding with Ground Plane.

computation and data visualization share the same GPU hardware resource, compared to around 90 seconds on the CPU.

Moreover, 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 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. 3.10. All of the computations used double precision.

3.4 Summary

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 chapter, 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.

Chapter 4: The Scalable FMM on the Heterogenous Architecture

Because of the popularity of heterogeneous CPU-GPU architecture over the past decades, we will reconsider the FMM algorithm on such system with multi-core CPU(s) and one or more GPU accelerators, as well as on an interconnected cluster of such nodes to achieve a significant improvement over recent implementations and to make the algorithm ready for use as a workhorse simulation tool for both time-dependent vortex flow problems and for boundary element methods. In this chapter, our goal is to perform large rotorcraft simulations [32] using coupled free vortex methods with sediments and ground effect [63] as well as develop fast methods for simulation of molecular dynamics [64], micro and nanoscale physics [26], and astrophysics [65].

We consider essentially similar hardware of the same power consumption characteristics as discussed in recent papers, and our results show that the performance can be significantly improved. Since GPUs are typically hosted in PCI-express slots of motherboards of regular computers (often with multi-core processors), an implementation that just uses GPUs or multi-core CPUs separately wastes substantial available computational power. Moreover, a practical workstation configuration currently is a single node with one or more GPU-accelerator cards and a few CPU sockets with multi-core processors. With a view to providing simulation speed-ups for both a single

node workstation and a cluster of such interconnected nodes, we develop a heterogeneous version of the algorithm.

To achieve an optimal split of work between the GPU and CPU, we performed a cost study of the different parts of the algorithms and communication (see Fig. 4.1). A distribution of work that achieves best performance by considering the characteristics of each architecture is developed. More exactly, using the observation that the local summation and the analysis-based translation parts of the FMM are independent, we map these respectively to the GPUs and CPUs. The FMM in practical applications is required to handle both the uniform distributions often reported on in performance testing and the more clustered non-uniform distributions encountered in real computations. We are developing strategies to balance the load in each of these cases.

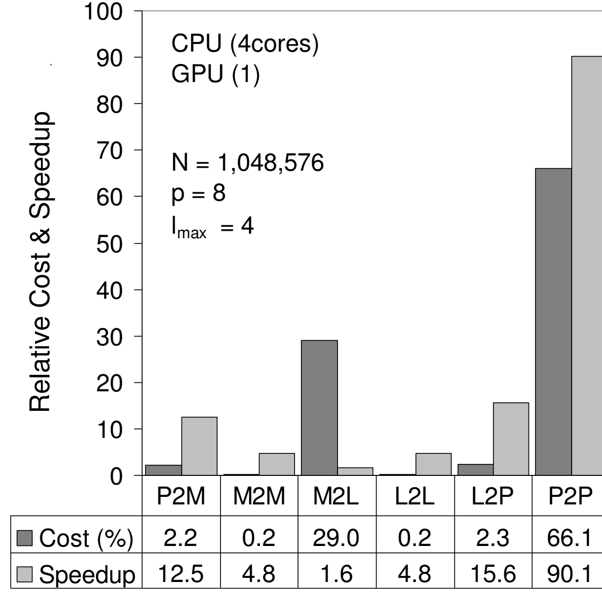
Since our algorithm is to be used in time-stepping where the data distribution changes during the simulation, we wanted to improve scalability in the algorithms for creating the FMM data-structures and interaction lists by using algorithms presented in Chapter 2. In this chapter, we first develop a single node version where the CPU part is parallelized using OpenMP and the GPU version via CUDA. Next, we present an simplified approach for clusters consisting of several heterogeneous nodes. This algorithm is sufficient for the number nodes we have access to. We also briefly consider the case of even larger numbers of nodes.

4.1 Single Heterogeneous Node

Our efficient algorithm is based on use of occupancy histograms (i.e., the counts of particles in all boxes), bin sorting, and parallel scans [49]. We have pointed out a potential disadvantage of this approach in Chapter 2 while we can achieve accelerations up to two orders of magnitude compared to CPU, which accounts for our preference of it. For problems that required greater octree depth, we developed a distributed multi-GPU version of the algorithm, where the domain is divided via octrees spatially and distributed to the GPUs, each GPU performs independent structuring of data residing in its domain, and global indexing is provided by applying prefixes associated with each GPU. The problem decomposition needed for load balancing can also be done with this data structure.

Different stages of the FMM have very different efficiency when parallelized on the GPU (Fig. 4.1). The lowest efficiency (due to limited GPU local memory) is for translations. On the other hand, computation of $\mathbf{\Phi}^{(sparse)}\mathbf{q}$ on the GPU is very efficient (making use of special instructions for the reciprocal square root and multiply-and-add operations), as well as the generation of M-expansions and evaluation of L-expansions. In fact, anything having to do with particles is very efficient on the GPU, and translations are relatively efficient on the CPU.

Figure 4.2 illustrates the work division between the CPU cores and GPU(s) on a single node. The large source and receiver data sets are kept in GPU global memory, and operations related to particles are performed only on the GPU. This makes dynamic simulations (where particles change location) efficient, since particle update can be done



	P2M	M2M	M2L	L2L	L2P	P2P
Cost (%)	2.2	0.2	29.0	0.2	2.3	66.1
Speedup	12.5	4.8	1.6	4.8	15.6	90.1

Figure 4.1: The Relative Cost and Speedup of Different Steps of the FMM. We tested uniform data on a GPU (NVIDIA GeForce 8800GTX) vs a 4 core CPU (Intel Core 2 extreme QX, 2.67GHz). The relative cost of steps is given for the GPU realization ($l_{\max} = 4$, $p = 8$, $N = M = 2^{20}$). The CPU wall clock time is measured for the same settings as for GPU (not necessarily optimal for the CPU [21]).

efficiently on the GPU minimizing CPU-GPU data transfer. The GPU does the jobs that it can do most efficiently, specifically, generating the data structure, generating M-expansions for source boxes at the finest level, performing the sparse MVP, evaluating L-expansions for the receiver boxes at the finest level, and producing final results. Because the result is obtained on the GPU, it can be used immediately for the next time step.

The CPU performs all work related to operations with boxes. It receives as input the box data structure from the GPU, which is used to generate a translation data structure (note that we use the reduced translation stencils described in [21], instead of the standard

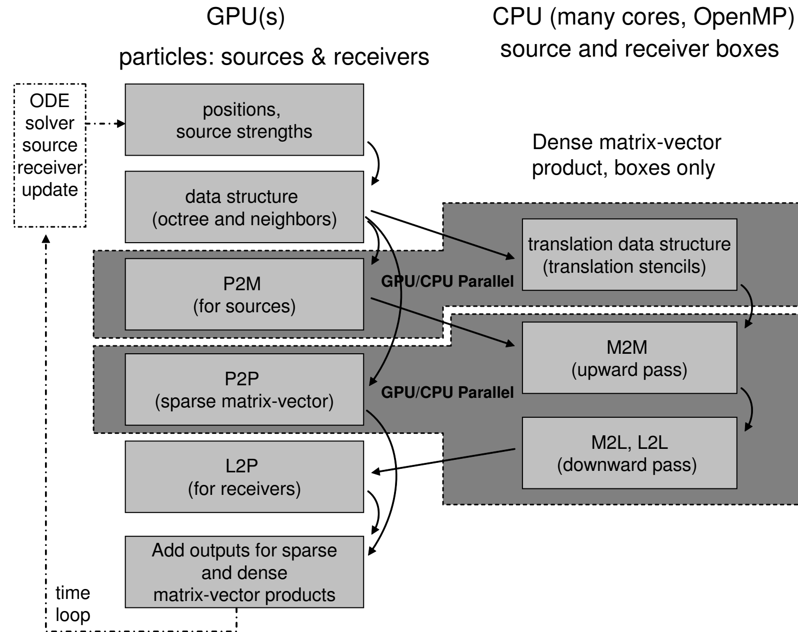


Figure 4.2: Flow Chart of the FMM on A Single Heterogeneous Node. Such node is equipped a few GPUs and several CPU cores. Steps shared in dark gray are executed in parallel on the CPU and the GPU.

189 per box), and M-expansion coefficients for the non-empty source boxes at the finest level. Then the CPU performs the upward and downward passes of the FMM and returns L-expansions for the non-empty receiver boxes at the finest level.

This strategy has several advantages

1. The CPU and GPU are tasked with the most efficient jobs they can do.
2. The CPU is not idle during the GPU-based computations and our tests show the loads on CPU and on GPU are reasonably balanced.
3. Data transfer between the CPU and GPU includes only p^2 expansion coefficients for each non-empty box, which usually is smaller than the particle data.

4. The CPU code can be better optimized as it may use more complex data structures, e.g., for complex translation stencils. More efficient automatic compilers are available for the CPU.

5. We use double precision without much penalty on the CPU. This is helpful since translation operations are more sensitive to round off.

6. If the required precision is below 10^{-7} single precision can be used for GPU computations. If the error tolerances are more strict, then double precision can be used on GPUs that support them.

7. The algorithm is efficient for dynamic problems.

8. Visualization of particles for computational steering is easy, as all the data always reside in GPU RAM.

4.2 Several Heterogeneous Nodes

For distributed heterogeneous nodes, the above algorithm can be efficiently parallelized by using the spatial decomposition property inherent to the FMM (see Fig. 4.3). The separation of jobs between the CPU cores and GPUs within a node remains the same. The M2L translations usually are the most time consuming and take 90% or more of the CPU time in the single node implementation. However, if we have P nodes and each node serves only N/P sources located compactly in a spatial domain $\Omega_j^{(s)}$, $j = 1, ..., P$ covered by $N_j^{(s)} \approx N^{(s)}/P$ source boxes at level $l_{\max}$ such that $\Omega_j^{(s)}$ do not intersect, then the number of the M2M and M2L translations for all receiver boxes

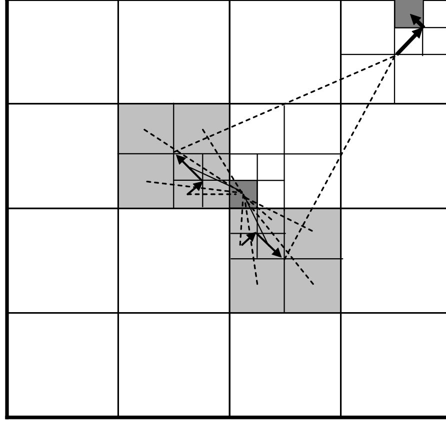


Figure 4.3: Illustration of Separation of the M2M and M2L Translation Jobs between Two Nodes. Two nodes handle sources allocated in two light gray source boxes and compute L-expansions for the darker gray receiver boxes. Solid lines with arrows show M2M translations, the dashed lines show M2L translations, and bold solid lines with arrows show L2L translations. M2M and M2L steps do not overlap, and the same (adaptive, empty box skipping) algorithm with different inputs can be executed on each node. L2L translations for the same boxes are duplicated by each node for the simplified algorithm in Sec 4.2.1 and are not duplicated in the general algorithm proposed in Sec 5.2.2.

due to sources in $\Omega_j^{(s)}$ is approximately $\left(C_{M2M} + C_{M2L}\right)/P$, where C_{M2L} and C_{M2M} are the numbers of all M2L and M2M translations for the entire domain, respectively, and $N^{(s)}$ the number of source boxes. This is achieved because our implementation of the FMM skips empty source boxes.

To achieve scalability for the L2L translations, we introduce an intermediate communication step between the nodes at levels $l = 2, ..., l_{\max}$. This is between the CPUs alone, and does not affect the parallel computation of the sparse MVP on the GPU. For

each node we subdivide the receiver boxes handled by it into 4 sets. These are built based on two independent criteria: belonging to the node, and belonging to the neighborhood containing all source boxes assigned to the node. "Belonging" means that all parents, grandparents, etc. of the box at the finest level for which the sparse MVP is computed by a given node also belong to that node. Receiver boxes that do not satisfy both criteria are considered "childless" and the receiver tree for each node is truncated to have that boxes as leaves of the tree.

During the downward pass a synchronization instruction is issued after computations of the L-expansion coefficients for receiver boxes in the tree at each level. The nodes then exchange only information about the expansions for the leaf boxes, and each node sums up only information for the boxes which belong to it. These steps propagate until level $l_{\max}$ at which all boxes are the leave boxes and information collected by each node is passed to its GPU(s), who evaluate the expansions at the node source points and sum the result with the portion of the sparse MVP computed by this node.

4.2.1 Simplified Algorithm

If the number of nodes is not very large, the above algorithm can be simplified to reduce amount of synchronization instructions and simplify the overall data structure. In a simplified algorithm, each node performs an independent job in the downward pass to produce the L-expansion coefficients for *all* receiver boxes at level $l_{\max}$. These coefficients are not final, since they take into account only contribution of the sources allocated to a particular node. To obtain the final coefficients, all expansions for a given

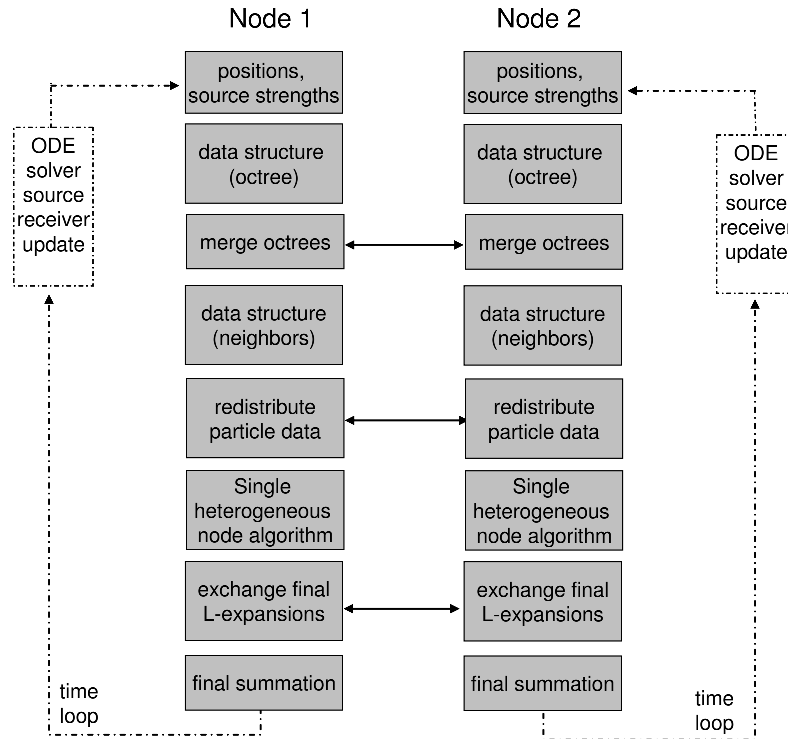


Figure 4.4: A Flow Chart of the FMM on Two Heterogeneous Nodes. The single heterogeneous node algorithm (see Fig. 4.2) is executed in parallel on CPU cores and GPUs available to the node.

receiver box must be summed up and sent to the node that computes the sparse MVP for that receiver box. This process for all nodes can be efficiently performed in parallel using hierarchical (golden) summation according to the node indices.

In contrast to the general algorithm, in the simplified algorithm some L2L translations are duplicated (see Fig. 4.3), which deteriorates the algorithm scalability and increases data transfer between the nodes. However, even in the worst case when all L2L translations are repeated on all nodes, the effect of L2L-translation duplication may have a substantial effect on the overall complexity only if the number of all L2L translations

violates the restriction $C_{L2L} \ll (C_{M2M} + C_{M2L})/P$. This inequality holds for moderate clusters. Indeed, if we use a scheme with a maximum of 119 or 189 M2L translations per box and one L2L translation then for $P \lesssim 100$ the scheme is acceptable, though sufficient memory per node is needed to keep L-expansion coefficients for all boxes in the tree. Our tests show that satisfactory performance for $N \lesssim 10^9$, which is comparable with the number of particles used in any FMM realizations we are aware of.

For illustration, the flow chart in Fig. 4.4 shows the algorithm for two heterogeneous nodes. Each node initially has in memory the sources and receivers assigned (randomly) to the node. Based on this, each node builds an octree. After that, all nodes are synchronized and receiver hierarchy data is scattered/gathered, so each node has complete information about all receiver boxes.

Since initial source/receiver data is redistributed between the nodes, each node takes care of a spatially compact portion of particle data. This distribution is done by prescribing weights to each box at a coarse level of the tree and splitting the tree along the Morton-curve to achieve approximately equal weights to each part. Then the single node heterogeneous algorithm described in the previous section is executed with some small modifications. The sparse MVP is computed only for the receiver boxes handled by a particular node, and the dense MVP is computed only for source boxes allocated on that node but for all receiver boxes (i.e. influence of a portion of the source boxes on all receivers is computed). The data on the L-expansions at the finest level is then consolidated for the receiver boxes handled by each node. The final summation consists of evaluation of the L-expansions and summation with partial sparse MVPs.

4.3 Implementation Issues

We use OpenMP for intra-node CPU computation (as implemented in Intel FORTRAN/C v.11) and MPI for inter-node communication. As shown in Fig. 4.4, on each computing node we launch a single process, within which multiple OpenMP threads are used to control multiple GPUs and compute FMM translations on the CPU. MPI calls for communicating with other nodes are made from the master thread of this process.

CUDA 3.2, which we used, allows only one active GPU device in a single CPU thread. Hence, to parallelize the multi-threaded CPU translation and multi-GPU direct sum, OpenMP threads have to be divided into two different groups. To avoid performance degradation due to the nested OpenMP parallel regions performing quite different computational tasks, the threads that control GPUs are spawned first. After a thread launches its GPU kernel function call, it immediately rejoins the master thread. At this point the threads for CPU translations are spawned, while the GPUs perform the local direct summation in the mean time.

After initial partition, each node has its own source and receiver data. The receiver data are mutually exclusive among all the nodes; however, the same source data might repeatedly appear on many nodes since they belong to the interaction neighborhood of many receiver boxes. As shown in Fig. 4.2, the main thread spawns multiple threads to copy data onto different GPUs and performs initial data structure building. Then each thread sends information on the receiver boxes to the master thread. The master thread computes the global receiver box information and broadcasts it to all other nodes. Based on the previous data structures and the global receiver box information, each GPU

then builds its own data structure for the CPU translations, performs initial multipole expansions, and copies them to CPU. Next, the CPU translation and GPU direct sum are performed simultaneously using the scheme described above.

For the simplified algorithm described above, the master thread collects the local expansion coefficients as a binary tree hierarchy within l rounds given 2^l processes. When it finishes, the master process has the local expansion data for all receiver boxes and sends the corresponding data to all other processes.

4.4 Performance Tests

Partitioning times: Our algorithm takes some time for global partition of the data. However, in a dynamic problem, this step is only needed at the initial step, and this time is amortized over several time steps, after which the global repartitioning may again be necessary. Accordingly, we report the partitioning time separately from the total run time. The run time, however, includes the cost of generating the entire FMM data structure on each node since this will have to be done at each time step. In the tests we measured the time for potential + force (gradient) computations and also for a faster version where only the potential was computed.

Most cases are computed with $p = 8$, which is sufficient to provide single precision accuracy (relative errors in the L^2-norm are below 10^{-5}). When comparisons are made with [2], we set $p = 10$ to match their choice. The benchmark cases include random uniform distribution of particles inside the cube and on the surface of a sphere (spatially non-uniform). In our wider tests we varied the number of sources and receivers N and

M. In all the reported cases, $N = M$, while the source and receiver data are different.

4.4.1 Single Heterogeneous Node

The algorithm for a single node was extensively tested. The first test was performed on a single node of the Chimera cluster using one and two GPUs with spatially uniform random particle distributions. Table 2 shows the measured performance results in optimal settings (in terms of the tree depth $l_{\max}$) for potential + force computations (the total run time for potential only computations is also included). Fig. 4.5 plots data only for potential computations in optimal settings. Even though these timing results appear to outperform the results of other authors on similar hardware which we are aware of, one may question whether the algorithm is scalable with respect to the number of particles and number of GPUs since the run time changes quite non-uniformly. An explanation of the observed performance is that the GPU sparse MVP has optimum performance for certain data cluster size $s_{opt}^{(sparse)}$ [21], and when clusters with $s < s_{opt}^{(sparse)}$ are used, this increases both the GPU and CPU times (due to increase of $l_{\max}$). Cluster sizes with $s > s_{opt}^{(sparse)}$ can be optimal, and this can be found from the balance of the CPU and GPU times. Since $l_{\max}$ changes discretely and the CPU time depends only on the number of boxes, or $l_{\max}$ (for uniform distributions), the CPU time jumps only when the level changes. Increase of $l_{\max}$ by one increases the CPU time eight times, and such scaling is consistent with the observed results. On the other hand, the GPU time for fixed $l_{\max}$ and $s > s_{opt}^{(sparse)}$ is proportional to N^2. So there is no way to balance the CPU and GPU times for a fixed $l_{\max}$, except perhaps increasing the precision of CPU computation.

Time (s) \ N	1,048,576		2,097,152		4,194,304		8,388,608		16,777,216	
Num of GPUs	1	2	1	2	1	2	1	2	1	2
CPU wall clock	0.13	0.13	1.06	1.08	1.07	1.11	1.02	1.10	8.53	8.98
C/G parallel region	0.58	0.30	1.06	1.08	1.58	1.11	4.38	2.21	8.55	8.98
Force+Potential total run	0.71	0.39	1.23	1.22	1.96	1.34	5.11	2.63	10.3	10.1
Potential total run	0.40	0.24	1.16	0.89	1.27	1.25	2.94	1.52	9.76	6.30
Partitioning	–	0.14	–	0.32	–	0.58	–	1.14	–	3.09

Table 4.1: Performance on A Single Heterogeneous Node. Force with potential (best settings). For potential computations, only the total run time is provided.

If the GPU time dominates, then use of the second GPU reduces the time, as seen for the cases $N = 2^{20}$ and $N = 2^{23}$. Note also that the parallelization efficiency for 2 GPUs is close to 100%. On the other hand, if the CPU time dominates, then the second GPU does not improve performance if $l_{\max}$ remains the same (see case $N = 2^{22}$ for the potential). Cases $N = 2^{21}$ and $N = 2^{24}$ show a reduction of the time due to the use of the second GPU because of a different reason. For these cases optimal $l_{\max}$ is different when using one or two GPUs. This causes reduction of the CPU time and increase of the single GPU time for two GPUs compared to the case with one active GPU.

We also performed tests with spatially non-uniform distributions (points or the sphere surface). This requires deeper octrees to achieve optimal levels (usually increase of $l_{\max}$ by 2) and provides more non-uniform loads on the GPU threads, which process the data box by box (the number of points in the non-empty boxes varies substantially).

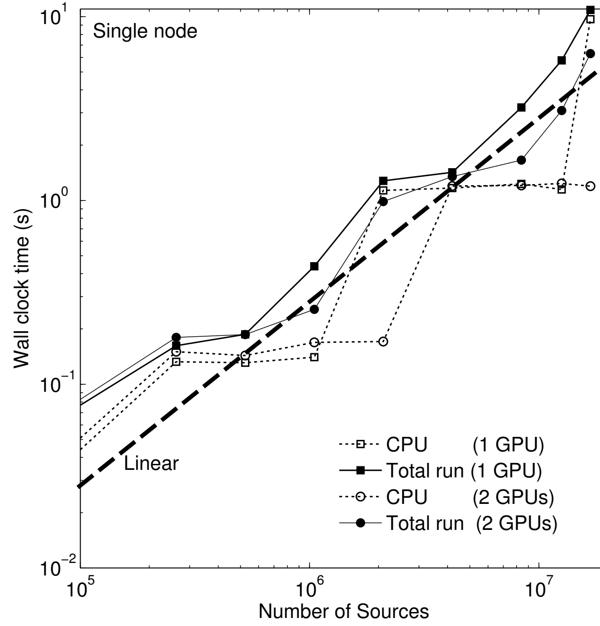


Figure 4.5: The Wall Clock Time for the heterogeneous FMM Running on A Single Node with One and Two GPUs. The computation is only for potential. The CPU part (using 8 CPU cores) is plotted by the thin dashed lines. The thick dashed line shows linear scaling.

This results in the increase of both CPU and GPU times. In some cases this increase is not substantial (e.g., for $N = 2^{20}$ the CPU/GPU parallel region time is 0.44 s and the total run time is 0.56 s), while we never observed increase more than 2.5 times (e.g., for $N = 2^{22}$ the CPU/GPU parallel region time is 2.82 s and the total run time is 3.26 s for potential only computations) (compare with Table 4.1).

4.4.2 Double Precision GPU Performance

Accepted wisdom in scientific computing often requires double precision computations for many problems. We accordingly demonstrate our algorithm with double

	Prec	$p = 4$	8	12	16
Time (s)	S	0.37	0.62	1.48	2.92
	D	1.36	1.40	1.49	2.95
Error	S	$2.8{\cdot}(-4)$	$1.4{\cdot}(-6)$	$2.5{\cdot}(-7)$	$1.2{\cdot}(-7)$
	D	$1.6{\cdot}(-4)$	$6.9{\cdot}(-7)$	$4.3{\cdot}(-8)$	$4.3{\cdot}(-9)$

Table 4.2: Performance and Error Results for Single and Double Precision Computations. Tests are performed on the GPU, where "$(-m)$" means "10^{-m}".

precision on the CPU, and both single and double precision on the GPU. Table 3 shows results of our tests for potential computation for $N = 2^{20}$ using different truncation numbers p and algorithms with different GPU precision. We used a workstation with a Tesla C2050 and a 4 core Intel E5504 2.00 GHz CPU (which explains the difference in run times in comparison with Tables 4.1 and 4.2). The error was measured in the relative L^2-norm using 100 random comparison points. Direct double precision computations were used to provide the baseline result. It is seen that for $p \leqslant 8$ there is no need to use double precision (which slows down the GPU by at least a factor of two). The CPU time grows proportionally to p^3, and the GPU time depends on p as $A + Bp^2$ with a relatively small constant B. This changes the CPU/GPU balance of the heterogeneous algorithm and reduces the optimal tree depth $l_{\max}$ as p increases.

4.4.3 Multiple Heterogeneous Nodes

On heterogeneous clusters we varied the numbers of particles, nodes, GPUs per node, and the depth of the space partitioning to derive optimal settings, taking into account data transfer overheads and other factors.

The weak scalability test is performed by fixing the number of particles per node to $N/P = 2^{23}$ and varying the number of nodes (see Table 4.3 and Fig. 4.6) for a simplified (small cluster) parallel algorithm. For perfect parallelization/scalability, the run time in this case should be constant. In practice, we observed an oscillating pattern with slight growth of the average time. Two factors affect the perfect scaling: reduction of the parallelization efficiency of the CPU part of the algorithm and the data transfer overheads. The results in Table 4.1 for 2 GPUs were computed with $l_{\max} = 5$ for cases $P = 1$ and 2 and $l_{\max} = 6$ for cases $P = 4, 8, 16$. In the case of the ideal CPU algorithm the time should reduce by a factor of two when the number of nodes is doubled at constant $l_{\max}$ and increase eight-fold when $l_{\max}$ increases by one. In our case, P is constant, so when $l_{\max}$ increases and P doubles the CPU time should increase by factor of 4. Qualitatively this is consistent with the observed pattern, but the simplified algorithm that we used is not perfectly scalable due to overheads (e.g., due to L2L-translations and related unnecessary duplication of the data structure) becoming significant at large sizes. The deficiency of the simplified algorithm also shows up in the data transfer overheads. The amount of such transfers depends on the number of boxes and the table clearly shows that the overhead time increases with $l_{\max}$. Figure 4.6 shows that for relatively small number of nodes this imperfectness is acceptable for practical problems.

Time (s) \ N (P)	8,388,608 (1)		16,777,216 (2)		33,554,432 (4)		67,108,864 (8)		134,217,728 (16)	
Num of GPUs	1	2	1	2	1	2	1	2	1	2
CPU wall	1.02	1.10	4.49	0.61	2.71	2.80	1.41	1.75	0.85	1.22
CPU/GPU	2.45	1.23	4.49	2.91	2.71	2.80	2.67	1.75	6.25	3.17
Overhead	0.50	0.30	1.03	0.36	0.95	0.85	1.12	0.96	1.31	1.07
Total run	2.95	1.53	5.46	3.27	3.66	3.65	3.79	2.71	7.56	4.24

Table 4.3: Performance for P Heterogeneous Nodes with $N/P = 2^{23}$. Results are for potential only.

We also performed the strong scalability test, in which N is fixed and P is changing (Fig. 4.7). The tests were performed for $N = 2^{23}$ and $P = 1, 2, 4, 8, 16$ with one and two GPUs per node. The deviations from the perfect scaling can be explained as follows. In the case of one GPU/node, the scaling of the CPU/GPU parallel region is quite good. We found that in this case the GPU work was a limiting factor for the parallel region. This is consistent with the fact that the sparse MVP alone is well scalable. In the case of two GPUs, the CPU work was a limiting factor for the parallel region. Scalability of the algorithm on the CPU is not as good as for the sparse MVP part because of the reasons explained above when the number of nodes increase. However, we can see approximate correspondence of the times obtained for two GPUs/node to the ones with one GPU/node, i.e. doubling of the number of nodes with one GPU or increasing the number of GPUs results in approximately the same timing. This shows a reasonably good balance between the CPU and GPU work in the case of 2 GPUs per node (so this is more or less the optimal

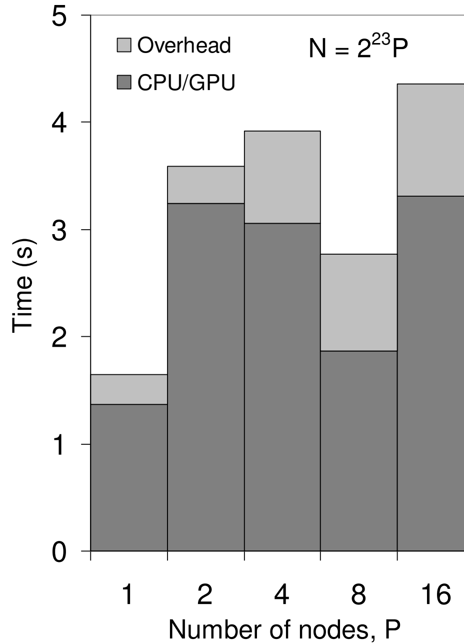


Figure 4.6: Contribution of the CPU/GPU Parallel Region Time and the Overhead to the Total Run Time for Two GPUs per Node. The overhead is due to data transfer between the nodes and CPU/GPU sequential region. The data size increases proportionally to the number of nodes. The time is measured for computations of potential only on UMIACS Chimera cluster.

configuration for a given problem size). More significant imperfections are observed for the total run time at increasing P, which is related to the data transfer overheads between the nodes (we also saw that in the weak scalability tests).

We did similar tests on the Lincoln cluster, which produced almost the same results. These best results are obtained for optimal settings, when the GPU(s) reach their peak performance (in terms of the particle cluster size). For such sizes the UMIACS cluster is limited by $N = 2^{28}$. For larger problems a sub-optimal performance is obtained; however,

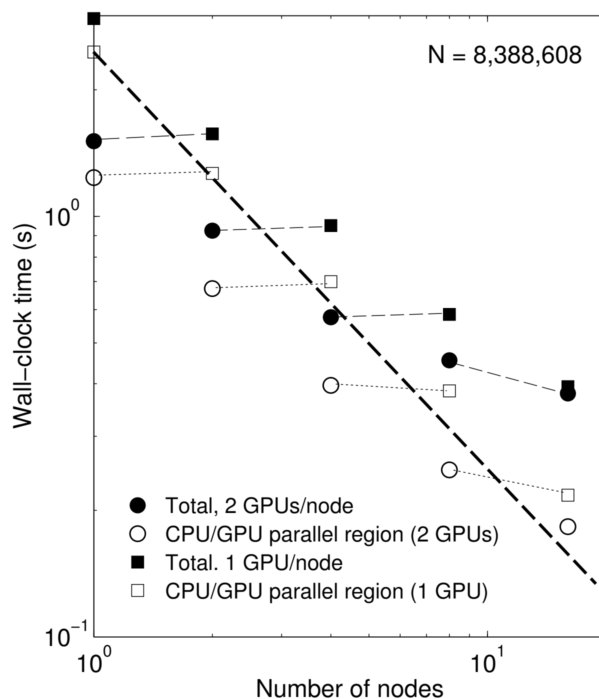


Figure 4.7: The Results of the Strong Scalability for One and Two GPUs per Node. The thin lines connect pairs of data points for which the same total number of GPUs was used. The thick dashed line shows perfect scalability $t = O(1/P)$. The time is measured for potential only computations on the UMIACS Chimera cluster.

such cases are still of practical importance, as solving billion-size problems in terms of seconds per time step is quite useful. Fig. 4.8 presents the results of the run employing 32 nodes with one or two GPUs per node, which shows a good scaling with N (taking into account that the FMM has jumps when the level changes). In the figure, we plot the best of one-GPU and two-GPU run times (as it was shown above, the use of the second GPU in the heterogeneous algorithm is not always beneficial and may create additional overhead). The largest case computed in the present study is $N = 2^{30}$ for 32 two-GPU

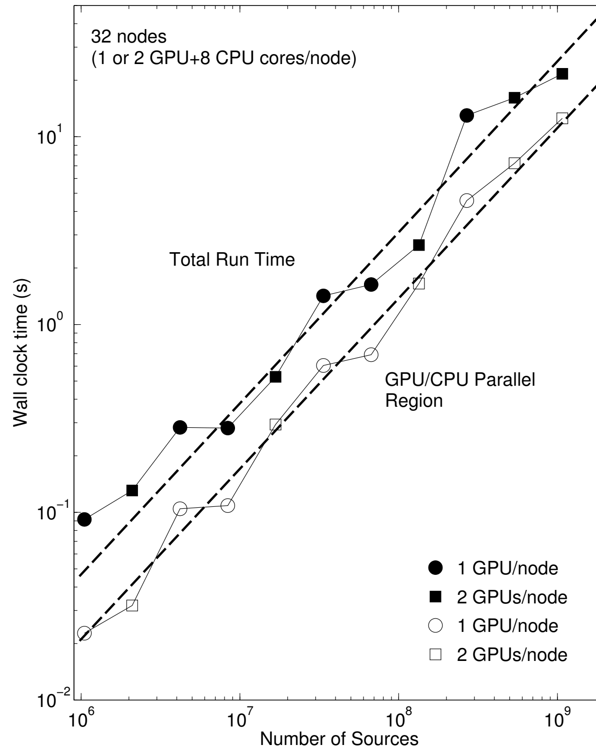


Figure 4.8: The Wall Clock Time for the Heterogeneous FMM Running on 32 Nodes Using One or Two GPUs per Node. The number of GPUs per node is optimized according to the load balance between CPU and GPU(s) in the parallel region. The thick dashed lines show the linear scaling and the computation results are for potential only.

nodes. For this case, the CPU/GPU parallel region time was 12.5 s and the total run time 21.6 s.

Figure 4.9 compares the wall clock time required by the potential+force computations for the present algorithm with the velocity+stretching timing reported in [2]. The present algorithm (Sec. 4.2.1) was executed with two GPUs per node for 2 and 8 nodes and one GPU for a single node (the total number of GPUs employed is shown in

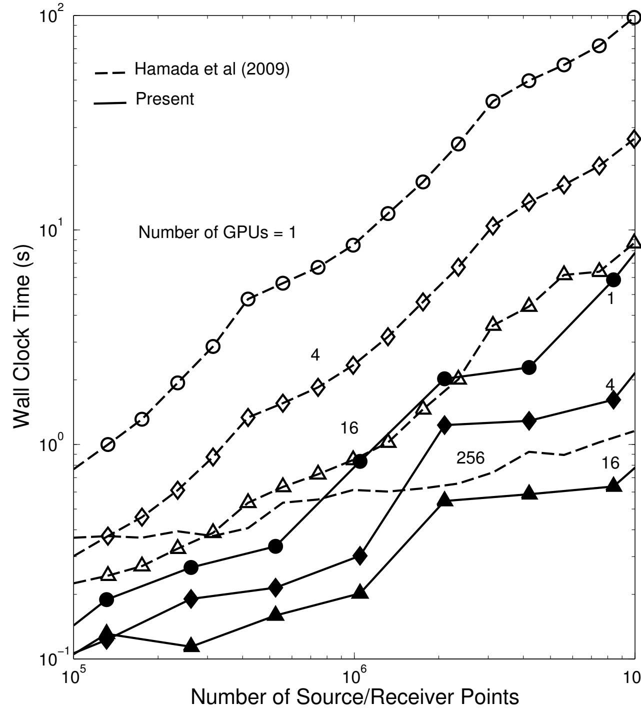


Figure 4.9: A Comparison of the Simplified Distributed FMM Implementation for N-Body Force+Potential Calculations with the 2009 Gordon Bell Prize Winner [2]for Velocity+Stretching Calculations. Velocity+stretching computation theoretically requires about 5.5 times more computation for the same N (see Fig. 12 there) than Force+Potential computations. In all cases the truncation number $p = 10$.

the figure).

4.5 Performance Assessment

There are several ways to assess the performance of our algorithm (see `http://folding.stanford.edu/English/FAQ-flops` for a discussion). One way to assess this performance is to look at the actual number of operations performed in

the FMM. This yields much more modest numbers of flops, but may be more useful in comparing the performance of different implementations of the FMM. The most arithmetically intensive part of the FMM is performed by the GPU during the sparse MVP. For uniform distributions and cluster size s, the number of operations at max. tree depth $l_{\max}$ is approximately $N_{op}^{(sp)} = 2^{3l_{\max}} 27 s^2 x$, where x is the number of operations per direct evaluation and summation of the potential or potential+force contribution. For a cluster of size $s = 2^8$ and $l_{\max} = 5$ we have $N_{op}^{(sp)} = 2^{31} 27x$. This corresponds to $N = 2^{3l_{\max}} s = 2^{23}$. Our tests show a very consistent GPU/CPU parallel region time per GPU for this problem size, which is strongly dominated by GPU, $t^{(sp)} = 2.45$ s (two GPUs do the same job for 1.23 s). This means that a single GPU performs at peak rate not less than $R = N_{op}^{(sp)} / \left(2^{30} t^{(sp)}\right) = 22x$ GFlops. The estimation of the x is rather tricky, and somewhat controversial, since a larger value tends to indicate better performance for one's algorithm. For potential only computations it can be estimated either as $x = 9$ in terms of GPU instructions, or as $x = 27$ in terms of CPU instructions, which is the commonly accepted way of counting. The latter value, which is consistent with the tests of the algorithms using standard CPU counters provides $R_{\max} = 594$ GFlops while $x = 9$ results in $R_{\min} = 198$ GFlops. Both these numbers are within the limit of the GPU used (933 GFlops peak performance reported by NVIDIA). Similar numbers can be computed for the potential+force computations by using $x = 38$ or $x = 15$, and the timing data from Table 2.

The contribution of the CPU part of the algorithm in the parallel region improves this marginally. We count all M2M, M2L, and L2L translations, evaluated the number of operations per translation, and used the measured times for the GPU/CPU parallel region

at $l_{\max} = 5$. That provided as a estimate of 27 GFlops per node for 8 CPU cores. Thus we can bound the single heterogeneous node performance for two GPUs between 423 and 1215 GFlops. We used at most 32 nodes with two GPUs each, and our cluster's performance bounds are [13.2, 38] TFlops.

Following [66], several authors, used a fixed flop count for the original problem and computed the performance that would have been needed to achieve the same results via a "brute-force" computation [2, 67–70]. For $N = 2^{30}$ sources/receivers and a computation time of 21.6 s (total run), which we observed for 32 nodes with two GPUs per node, the brute-force algorithm would achieve a performance of **1.25 Exaflops**.

4.6 Summary

Perhaps our biggest contribution is the use of the neglected resource (CPU) in heterogeneous CPU/GPU environments, allowing a significant improvement in hardware utilization. A user with a gaming PC worth less than $3000 (2 graphics cards and two-quad-core CPUs plus 16 GB RAM) can achieve FMM performance comparable with the 2009 Bell prize winner, at one percent of the cost and power, and a performance of 405 MFlops/dollar. Our algorithm can tackle two orders of magnitude larger problems of interest in fluid, molecular and stellar dynamics due to vastly improved handling of data structures and algorithmic improvements compared to current state-of-the-art algorithms. It also scales well, allowing the user to add more CPU cores and GPU cards to further improve the price/performance ratio. The Chimera machines cost $220K and achieve a performance of 177 MFlops/dollar—the best performance reported so far on the FMM

reported in [2].

We believe that the achieved total run times, up to $2^{30} \gtrsim 1$ billion particles was performed with this algorithm on a mid-size cluster (32 nodes with 64 GPUs), are among the best performance results reported for the FMM for the sizes of the problems considered (e.g, comparing with [2, 8, 28, 67, 71]).

Chapter 5: Parallel Algorithms for FMM Data Structures on Multiple Nodes

Since all the data structures are constructed based on the locations of source and receiver data points, there are two main issues for distributed systems with multiple nodes. First, for problems running on a single computing node, source and receiver data points for direct sum and translation are the same. However, on multiple nodes using the algorithm of [33, 35], only receiver data points are mutual exclusively distributed. Source points which are in the halo regions have to be distributed on several nodes, i.e. source points which are in the boundary layers of partitions have to be repeated among several computing nodes because of the direct sum region overlap. Hence on each computing node, the source data points for direct sum and translation are no longer the same. When we build the translation data structure, these repeated source data should be guaranteed to translate only once among all the nodes.

Second, since any translation stencils may require coefficients from many source boxes which are on other nodes, there will be many translation coefficient communications among different nodes. Moreover, due to the partition, from a certain level on, the octree box coefficients on a single node might be incomplete up to the root of the octree. This is because if one of any box's children is distributed on one or several

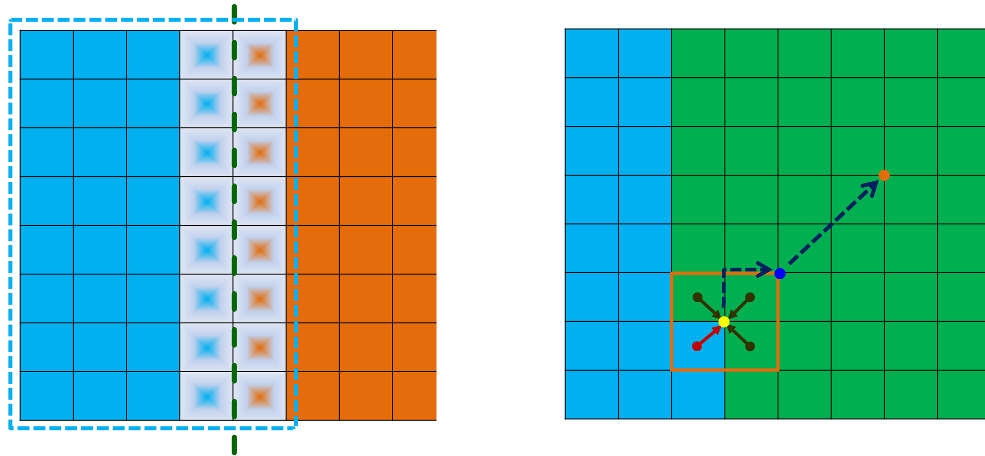


Figure 5.1: Problems Due to Partition. Left: blue boxes are Partition I and orange boxes are Partition II; dark green line shows the partition boundary line and light blue dash line shows the source regions which are needed by Partition I. The boxes with gradient orange color are in Partition II but they have also to be included in Partition I to compute full local direct sum. Right: blue boxes belong to Partition I and green boxes belong to Partition II. The S-data of the yellow box center bounded by the orange line is incomplete due to one alien child box. Hence its parents (the blue and orange dots) in the tree up to the minimal level are all incomplete.

different nodes, all its ancestors coefficients are incomplete. In figure 5.1, we show such an example. Lastly, all information related to these boxes are stored in a compressed way because we skip all empty boxes. Therefore it is a non-trivial task to pack all non-empty boxes' data and address any requested box to fetch its coefficients efficiently in which many searching and rearranging data operations are needed. Good partition and data communication algorithms are crucial to reduce the communication overhead in terms of both data transfer size and data packing time.

5.1 Global Data Structure and Partitioning

In [33] FMM algorithms for the heterogeneous CPU-GPU architecture were explored and it was concluded that a good strategy is to distribute the FMM computation components between CPUs and GPUs: expensive but highly parallizable particle related computations (direct sums) are assigned to the GPU, while the extensive and complex space box related computations (translations) are assigned to CPU. This way one can take the best advantages of both CPU and GPU hardware architecture, we design our data structures for this mapping. Below are our requirements/ideas for the distributed data structure:

1. There are P computing nodes. Each node handles a certain amount of sources and receivers. There is a master process which manages all computations and data distribution/transfer.

2. Algorithms for data partitioning/redistribution (based on the cost balance) are available, and these provide size of the computational cube D, octree of depth l_{max} and global Morton indexing of all boxes in the source and receiver trees. This algorithm also determines the halo regions induced by the space partition and takes care of data distributions of these halo regions.

3. Such algorithm is parallel, so that each computing node handles some part of the data, generates sub-trees of data belonging to it. The master process manages exchange of information, prefixing of local node data, and tools for global Morton indexing.

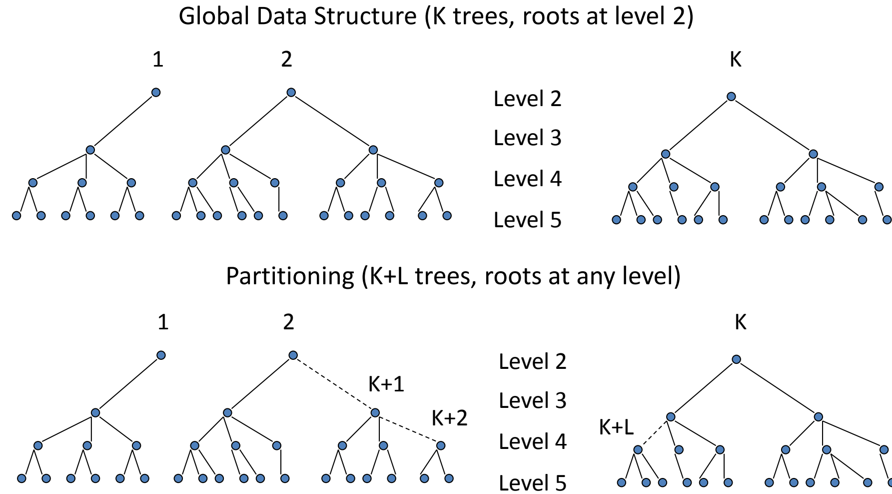


Figure 5.2: Global Data Structure with K Trees with Roots at Level 2 and Partitioned Data Structure with Roots at Any Level.

4. The spatial distribution of data affects the amount of communications between the nodes. However, the algorithm works correctly even if the distribution is substantially non-uniform.

5. We assume also that the partitioning is done in the way that each node has information about the sources and receivers it needs to compute the local sum. We assume also that tools to generate M-expansions and evaluate L-expansions for the boxes handled by each node are available. So communication between the nodes includes only exchange of the information about expansions related to the M and L boxes.

The FMM data structure is based on two data hierarchies for sources an receivers. Figure 5.2 shows only one of them. This can be viewed as a forest of K trees with roots at level 2 and leaves at level l_{max}. In the case of more or less uniform data distributions and

number of nodes less than $K \leq 64$ (for the octree), each node may handle one or several trees. If the number of nodes are more than 64 and/or data distributions are substantially non-uniform, partitioning based on the work load balance can be performed by splitting the trees at levels > 2. Such partitioning can be thought as breaking of some edges of the initial graph. This increases the number of the trees in the forest, and each tree may have a root at an arbitrary level $l = 2, ..., l_{max}$. Each node then takes care for computations related to one or several trees. At this point we assume that there exists some work load balancing algorithm which provides an efficient partitioning. At this point we also do not put any constraint to interaction between the receiver and source trees, so formally this can be considered as two independent partitions.

For presentation purposes, we define two important concepts although they are related to each other in our implementation:

- **Partition level** l_{par}: at this level, the whole space are partitioned among different computing nodes. On a local node, all the sub-trees at this level or below are totally complete, i.e., no box at level $\geq l_{par}$ is on other nodes.
- **Critical level** l_{crit}: at this level, all the box coefficients are broadcasted such that all boxes at level $\leq l_{crit}$ can be treated as local boxes, i.e., all the box coefficients are complete after broadcasting. In our implementation, $l_{crit} = \max(l_{par} - 1, 2)$.

One thing to notice here is that: normally the number of computing nodes in current high performance clusters are in the order of hundreds or even thousands, that is considered much smaller than the number of spatial boxes of the global octree. Hence, the partition level is usually quite low, such as $l_{par} = 2, 3, 4$. Hence broadcasting the coefficients at the

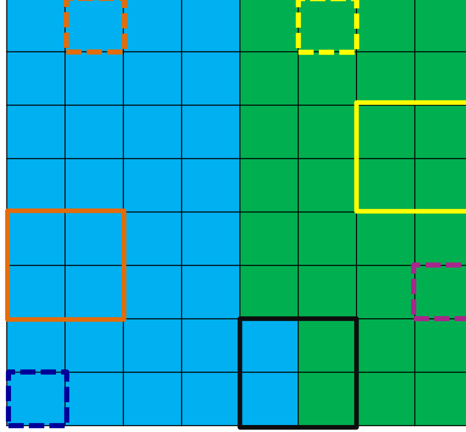


Figure 5.3: Source Box Type. Blue boxes are Partition I and green boxes are Partition II. The partition level is 3 and critical level is 2. Solid line boxes correspond to level 2 and dash line boxes correspond to level 3. As partition II, boxes with yellow boundary lines are export boxes. Boxes with orange boundary lines are import boxes. Box with black boundary lines are root box. Box with pink boundary lines are domestic box. Box with dark blue boundary lines are other box.

critical level l_{crit} only requires a small amount of data communication with neglectable overheads given the major cost of kernel evaluations.

To better organize data communications, on each node, we classify all the source boxes into five categories and store them into a box type array `SrcNonEmptyBoxType[]`. From the finest level l_{max} to 2, we list all the global source box Morton indices in an increasing order. Note that some local empty source boxes might be in the list since this box may contain source points which are located on other nodes and this global source boxes information is obtained from the initial global octree

construction. Given those box types, we could determine which boxes need to import or export their M-coefficients. For any node, say J, these five box types are:

1. **Domestic Boxes**: The box and all its children are on J. All domestic boxes are organized in trees with roots located at level 1. All domestic boxes are located at levels from l_{max} to 2. The roots of domestic boxes at level 1 are not domestic boxes (no data is computed for such boxes).

2. **Export Boxes**: These boxes need to send data to other nodes. At l_{crit}, the M-data of export boxes may be incomplete. At level $> l_{crit}$, all export boxes are domestic boxes of J and their M-data are complete.

3. **Import Boxes**: Their data are produced by other computing nodes for importing to J. At l_{crit}, the M-data of import boxes may be incomplete. At level $> l_{crit}$, all import boxes are domestic boxes of nodes other than J and their M-data are complete there.

4. **Root Boxes**: These are boxes at critical level, which need to be both exported and imported. For level $> l_{crit}$ there is no root box.

5. **Other Boxes**: Boxes which are included in the data structure but do not belong to any of the above types, e.g. all boxes of level 1, and any other box, which for some reason is passed to the computing node (such boxes are considered to be empty and are skipped in computation, so that affects only the memory and amount of data transferred between the nodes).

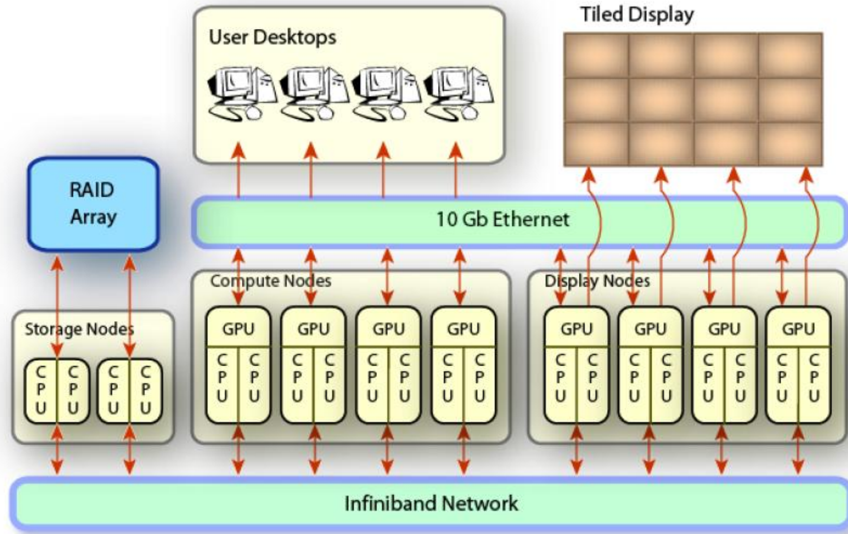


Figure 5.4: The Heterogenous CPU-GPU Computing Architecture of Chimara at UMIACS.

Refer to Fig. 5.3 for an example. Note that there are no import or export boxes at levels from $l_{crit} - 1$ to 2. All boxes at these levels are either domestic boxes or other boxes after the broadcast and summation of incomplete M-data at l_{crit}. In our algorithm, we only need compute M-data and box types from level l_{max} to l_{crit} and exchange the information at l_{crit}. After that we compute the M-data for all the domestic boxes up to level 2 then produces L-data for all receiver boxes at level l_{max} handled by the computing node.

5.2 FMM Algorithm on Multiple Nodes

Our multiple node algorithm involves three main parts:

1. Global source and receiver data partition: the partition should keep work balance among all the computing nodes.

2. Single node evaluation: a single node performs the translations upward/downward, compute the export and import data and the local summations.

3. Multiple node data exchange: The data communication manager collects and distributes the data from/to all the computing nodes accordingly.

Parts (2) and (3) are mutually inclusive because the translations on a single node require the missing data from other nodes while the data communication requires import and export information from each computing node. Part (1) depends on the application. For dynamic problems, the FMM evaluation is performed for every time step and the data distribution can be derived from the previous time step. It is very likely that all the nearby data are stored on the same node, in which case the partition to keep work balance only requires a small amount of data communications. For problems only performing a single FMM evaluation, the data appear on each node might be dependent on some geometric properties but it is also possible that the initial data on each node is totally random, in which case a large amount of the inter-node communications is inevitable. In our implementation, we assume the worst case that all the data on each node are random.

5.2.1 Global Partition

The architecture of our computing system (see figure 5.4), and perhaps of most current and near future systems, is heterogeneous. Each computing node has several multi-core CPUs and many core GPUs. While the CPU cores on the same node share the

main memory, each GPU has its own dedicated device memory, which are connected to host via PCI-Express bus. To perform the computations on GPUs, the data for a single node have to be divided again for each GPU. As mentioned before, we perform direct sums on GPUs and FMM translation on CPUs, hence we need two level partition: divide the data for nodes first (translation) then further divide data of each node for each GPU (direct sum). Given the prescribed cluster size, we construct the global octree then split it, i.e., the partition of all the data is performed by boxes but not by particles.

Assume the same number g of GPUs on each computing node, then we implement this two level partition as follow: we assign a unique global ID $(ig + j)$ to the jth GPU on the node i and compute our finer partition with respect to those GPUs. From $l_{par} = 2$, our algorithm tries to distribute all the boxes at level l_{par} among GPUs such that the amount of receiver points satisfy the prescribed balance conditions. It increases the l_{par} by 1 until the work load balance is roughly achieved. Once this finer partition is done, we automatically obtain the balanced coarse partition with respect to computing nodes (this is because each node has the same number of GPUs). To identify all the box locations, we use an auxiliary array `BoxProcId[]`, in which ith entry stores the GPU ID where the ith box at l_{par} resides in. Dividing `BoxProcId[i]` by g, we can obtain the node ID where the ith box resides in. Note that, for any given box at any level, we can easily answer its location query by shifting that box's Morton index and examining `BoxProcId[]`, i.e. by checking its ancestor/children's location. Initially, we use GPUs and Alg. 1 to pre-process data, i.e. to get the number of receiver points in each box. Then all nodes send their `Bucket[]` array to the master node. The master node then computes the balanced partition and derives the value of l_{par} such that each GPU is assigned several spatial boxes

at l_{par} with consecutive Morton indices. Finally, the master node broadcasts this partition information to all the nodes and each nodes distributes its own source and receiver data based on the partition to others.

5.2.2 The Distributed FMM Algorithm

Assuming that the balanced global data partition and distributed data are available. On one hand, the data structure constructions of local neighbor interaction lists for direct sum are the same as Chapter 2. On the other hand, the data structures of translations are for the coarse partition (with respect to node), hence they need to be recomputed by merging the octree data structures obtained from multiple GPUs on the same node. The merging steps are conducted as follow

1. Extract all the global source box information across all the computing nodes: after all GPU calls the data structure construction call of section 2.5, each node collects these non-empty source box indices from all its GPU, merges to one list and send to the master node. Then the master node merges all the lists to one global non-empty source box array and broadcasts to all nodes.

2. Extract the local receiver box information for each node: each node collects these non-empty receiver box indices from all its GPU, merges to one list. Because each GPU deals with consecutive receiver boxes, this merging is actually equivalent to copy operations.

Once these two box index arrays are available, we can construct the necessary data structures, i.e. the interaction lists for translation stencils, in parallel on GPU. Except

the source box type, of which the algorithm will be described later, all other needed information arrays, such as neighbor lists/bookmark etc, can be obtained by using the similar algorithms presented in section 2.5.

Given the global partition and the node-wise local merging algorithms, now lets assume all the necessary data structures for the translations, such as box's global Morton indices, the neighbor, export/import box type lists and those initial M-expansion data, are available, then each node J executes the following translation algorithm:

1. **Upward translation pass**:
   (a) Get M-data of all domestic source boxes at l_{max} from GPU global memory.
   (b) Produce M-data for all domestic source boxes at levels $l = l_{max} - 1, \ldots, \max(2, l_{crit})$.
   (c) Pack export M-data, the import and export box indices of all levels. Then send them to the data exchange manager.
   (d) The master node, which is also the manager, collects data. For the incomplete root box M-data from different nodes, it sums them together to get the complete M-data. Then according to each node's export/import box indices, it packs the corresponding M-data then sends to them.
   (e) Receive import M-data of all levels from the data exchange manager.
   (f) If $l_{crit} > 2$, consolidate S-data for root domestic boxes at level l_{crit}. If $l_{crit} > 3$, produce M-data for all domestic source boxes at levels $l = l_{crit} - 1, \ldots, 2$.
2. **Downward translation pass**:

(a) Produce L-data for all receiver boxes at levels $l = 2, \ldots, l_{max}$.

(b) Output L-data for all receiver boxes at level l_{max}.

(c) Redistribute the L-data among its own GPUs.

(d) Each GPU finally consolidates the L-data, add the local sums to the dense sums and copy them back to the host according to the original inputting receiver's order.

Here all boxes mean all boxes handled by each node. A simple illustration of this algorithm for a 2D problem is shown in Fig. 5.5. Because $l_{crit} = 2$, no further $M|M$ translations are needed therefore the direct $M|L$ and $L|L$ translation are performed directly after the data exchanging.

Packing the M-data inside the manager is simple. When each node outputs its import box index array, they are listed in an increasing order from l_{max} to l_{crit}. The manager processes the requesting import box index array one after another. Given the ith requested source box index `ImportSrcIdx[i]`, the manager first figures out its level l_*. If $l_* < l_{crit}$, the manager derives `ImportSrcIdx[i]`'s ancestor in the partition level and check the array `BoxProcId[]` to find which node it belongs to. If $l_* == l_{crit}$, the manager will check all its children's node address (l_* can not exceed l_{crit} since all the boxes above the critical level are marked as domestic box). Once the manager identifies the node ID, where that box belongs to or its children belong to, it searches the export box index array from that node for `ImportSrcIdx[i]` at level l_* then makes an copy of the corresponding M-data in the sending buffer.

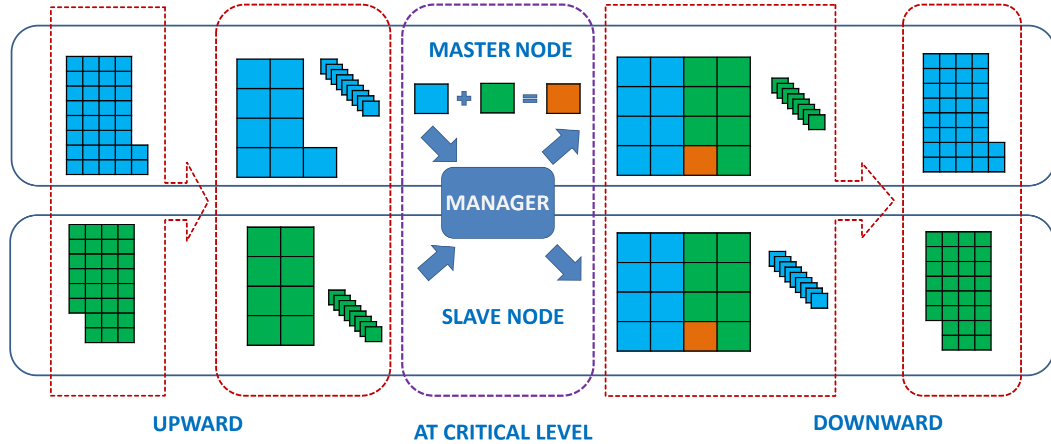


Figure 5.5: A Simple Multiple Node 2D FMM Algorithm Illustration ($l_{crit} = 2$). The top rounded rectangular is the MASTER NODE which is also chosen as the data exchange manager. The bottom rounded rectangular is the SLAVE NODE. From left to right is how the algorithm proceeds. Each node perform upward M-translations from l_{max} to l_{crit}. At critical level the manager collect and distribute data from and to all the nodes. Those isolated boxes in this figure are export boxes. After this communication, all the nodes perform downward $M|L$ and $L|L$ translations only for its own non-empty receiver boxes.

Even though each node handles a large number of spatial boxes, the amount information exchanged with the manager is actually small since only boxes on the partition boundary layers (for example, the total number is proportional to $O(4^l)$ in the case of the uniform distribution) need to be transferred back and forth. In our current implementation, the manager is responsible for all the collecting and redistributing M-data work, which involves searching operations, the total run time is still smaller by comparing with the communication scheme of [33], where all the box's L-data ($O(p^2 8^l)$ in the case of uniform distribution) at the finest level l_{max} are broadcast from the master

node.

5.2.3 Source Box Type

The type of a source box $\mathbf{k}$ is determined by the M2M and M2L translation because its children or neighbors might be missing due to the data partition, which have to be requested from other nodes. However, once the parent box M-data is complete, the L2L translations for its children are always complete. Hence, based on this observation, we can summarize the key idea of Alg. 8, which computes the type of each box, as follows:

- At the critical level, we need all boxes to perform upward M2M translations. If one child is on a node other than J, its M-data is either incomplete or missing, hence we mark it an import box. We also check its neighbors required by M2L translation stencil. If any neighbor is not on J, then the M-data of these two boxes have to be exchanged.

- For any box at the partition level or deeper levels, if this box is not on J, then it is irrelevant to this node, in which case it is marked as other box. Otherwise we check all its neighbors required by M2L translations. Again if any neighbor is not on J, these two boxes' M-data have to be exchanged.

We compute all box types in parallel on the GPU. For each level from l_{max} to 2, a group of threads on the node J are spawned and each thread is assigned by one source box index at that level. After calling Alg. 8, all these threads have to be synchronized before the final box type assignment in order to guarantee no race conditions. Note that some "if-then" conditions in Alg. 8 can be replaced by `OR` operations so that thread "divergent branches"

Algorithm 7 GET-SOURCE-BOX-TYPE(`BoxIndex[],Partition,`J): the algorithm to compute the source box types on the Node J given the partition

Input: a source box index `BoxIndex[i]`$=$ $\mathbf{k}$ at level l, the partition information and the node ID J

Output: `BoxType[i]`

1: `isOnNode`$\leftarrow$`isImportExport`$\leftarrow$`isExport`$\leftarrow$`FALSE`
2: **if** l<l_{crit} **then**
3:     `BoxType[i]`$\leftarrow$`DOMESTIC`
4: **else if** l=l_{crit} **then**
5:     **for** any $\mathbf{k}$'s child $\mathbf{c}_i$ at partition level **do**
6:         **if** $\mathbf{c}_i$ is not on J **then**
7:             `isImportExport`$\leftarrow$`TRUE`
8:         **else**
9:             `isOnNode`$\leftarrow$`TRUE`
10:     **if** `isOnNode=FALSE` **then**
11:         `BoxType[i]`$\leftarrow$`IMPORT`
12:     **else**
13:         **for** any $\mathbf{k}$'s neighbor of M2L translation $\mathbf{n}_i$ **do**
14:             **if** one of $\mathbf{n}_i$'s children at l_{crit} is not on J **then**
15:                 `isExport`$\leftarrow$`TRUE`   ▷ *update the type of a different box*
16:                 $\mathbf{n}_i$'s box type $\leftarrow$`IMPORT`

Algorithm 8 GET-SOURCE-BOX-TYPE (CONTINUED)

```
17: else
18:     if k's ancestor at l_crit is not on J then
19:         BoxType[i]←OTHERS
20:     else
21:         for any k's neighbor of M2L translation n_i do
22:             if the ancestor of n_i at l_crit is not on J then
23:                 isExport←TRUE                ▷ update the type of a different box
24:                 n_i's box type ←IMPORT
25: synchronize all threads
26: if isImportExport=TRUE then
27:     BoxType[i]←ROOT
28: else if isExport=TRUE then
29:     BoxType[i]←EXPORT
30: else
31:     BoxType[i]←DOMESTIC
```

Term	Description
$a_0(N_i + M_i)$	each node derives the local box Morton indices based its source and receiver points and send
$a_1 B^{recv}_{i,l_{max}}$	each node sends its receiver box indices at the finest level to the master node
$a_2 B^{recv}_{all,l_{max}}$	The master node collects all the receiver box indices and builds the global indices
$a_3 B^{recv}_{all,l_{max}} \log P$	The master node broadcasts the global receiver box indices and the partition information
$a_4(N + M)$	All nodes exchange the source and receiver data according to the partition, which includes a node-wide synchronization

Table 5.1: Description of Equitation 5.1

can be reduced.

5.2.4 Complexity

Lets assume that we have P computing nodes and each node has g GPUs. We only count non-empty boxes here and all symbols are used as follow: $B^{src}_{i,l}$ and $B^{recv}_{i,l}$ are the numbers of local source and receiver boxes at level l on the ith node respectively; $B^{src}_{all,l}$ is the number of global source boxes at level l; N_i and M_i are the numbers of source and receiver points on the ith node respectively; N and M are the total numbers across all the nodes.

Firstly, we estimate the running time of our global partition given the worst case (the initial data are totally random):

$$T_1 = a_0(N_i + M_i) + a_1 B^{recv}_{i,l_{max}} + a_2 B^{recv}_{all,l_{max}} + a_3 B^{recv}_{all,l_{max}} \log P + a_4(N + M). \quad (5.1)$$

Each term within the equation (5.1) is described in table 5.1. Note that, moving $O(N+M)$ data points is inevitable if the initial data on each node are random, however in many applications, this communication cost can be avoided or reduced substantially if this initial distribution is known. Also there are many publications on this initial partition (tree generation) and optimized communication in the literature such as [31,47,48], which can be used for different applications accordingly.

Secondly, Alg. 8 examines the occupancy status of each source box's E_4 neighbors. Note that the number of E_4 neighbors for any octree box is upper bounded by 27 and for each neighbor such check operation is in constant time. Hence the total cost to compute all source box types can be estimated as:

$$T_2 = b_0 B^{src}_{all,l_{max}} + b_1 B^{src}_{all,l_{max}-1} + \ldots + b_{l_{max}-2} B^{src}_{all,2} \leq b_* B^{src}_{all,l_{max}}. \quad (5.2)$$

Lastly, let's denote $B^{src}_{all} = \sum_{j=2}^{l_{max}} B^{src}_{all,j}$. There is no clean mathematics model to derive the number of sending export boxes and receiving import boxes, and the cost of the master node finding and packing M-data for each node. However, the boundary layers are nothing but the surface of some 3D object. Given the uniform distribution case, in which we can simplify the model, it is reasonable to estimate the exchanged box number as $\mu 2^{2l_{max}} = \mu 4^l_{max}$ with some constant μ for all the nodes ([27] estimate this number as $O((B^{recv}_{i,l_{max}})^{2/3})$, which is similar as ours). Therefore, at the critical level l_{crit}, the

Term	Description
$c_0 B_{all}^{src}$	each node examines its source box types and extracts import and export source box indices
$c_1 \mu 4^{l_{max}} p^2$	each node sends the export box's M-data to the master node
$c_2 \mu 4^{l_{max}} \log(\mu 4^{l_{max}})$	the master node addresses all the requested import box indices of each node
$c_3 8^{l_{crit}} p^2$	the master node packs all the import M-data and sends to the corresponding node

Table 5.2: Description of Equitation 5.3

exchanging data cost can be estimated as:

$$T_3 = c_0 B_{all}^{src} + c_1 \mu 4^{l_{max}} p^2 + c_2 \mu 4^{l_{max}} \log(\mu 4^{l_{max}}) + c_3 8^{l_{crit}} p^2. \tag{5.3}$$

Each term in the equation (5.3) is explained in table 5.2. By combining some constant coefficients, we can further simplify T_3 as

$$\begin{aligned} T_3 &= c_0 B_{all}^{src} + c_1 \mu 4^{l_{max}} p^2 + c_2 \mu l_{max} 4^{l_{max}} \log(4\mu) + c_3 8^{l_{crit}} p^2, \\ &= d_0 B_{all}^{src} + d_1 l_{max} 4^{l_{max}} + d_2 (4^{l_{max}} + d_3 8^{l_{crit}}) p^2. \end{aligned} \tag{5.4}$$

Because all the above three sections are executed in the sequential order, the total cost T of our multiple node data structure related computation can be obtained by summing them and rewritten as

$$T = T_1 + T_2 + T_3 = T_{gpu} + T_{cpu} + T_{comm}, \tag{5.5}$$

where

$$
\begin{aligned}
T_{gpu} &\leq a_0 M_i + a_1 B^{recv}_{i,l_{max}} + b_* B^{src}_{all,l_{max}} \\
&= a_0 \frac{M}{P} + a_1 \frac{B^{recv}_{all,l_{max}}}{P} + b_* B^{src}_{all,l_{max}}, \\
T_{cpu} &= d_0 B^{src}_{all} + d_1 l_{max} 4^{l_{max}}, \\
T_{comm} &= d_2(4^{l_{max}} + d_3 8^{l_{crit}})p^2 + a_2 B^{recv}_{all,l_{max}} + a_3 B^{recv}_{all,l_{max}} \log P \\
&\quad + a_4(N+M).
\end{aligned}
\tag{5.6}
$$

The ideal algorithm should have all the cost proportional to $1/P$, i.e. all the computations are evenly distributed. In our case, only the particle related terms but not all the box related terms are amortized among all the computing nodes. However, in practice, l_{max} can not be very large which is viewed as a constant in most cases. More sophisticated schemes can be used so that these theoretical non-scalable terms can be amortized among all the nodes. However, given our efficient parallel implementation, even the box number could be large, the constant coefficients for each term are so small, hence T_{gpu} and T_{cpu} are neglectable by comparing with kernel evaluations.

As for the communication part, except the last term $a_4(N+M)$, all other terms are relative small given the limitation of our GPU implementation ($l_{max} \leq 8$). For the M-data, even though each box has p^2 coefficients, the total number of exchanging boxes in the halo regions is small, hence this communication time is acceptable by comparing with kernel evaluations. The real killing communication comes from the last term $a_4(N+M)$ since we target on billion scale problems. Exchanging all these particle data requires much more time than the real kernel evaluations. However, this term is not necessarily in our algorithm because it is obtained from the worst case, that is the totally random distribution. In most application, based on the physical or geometric properties, this initial

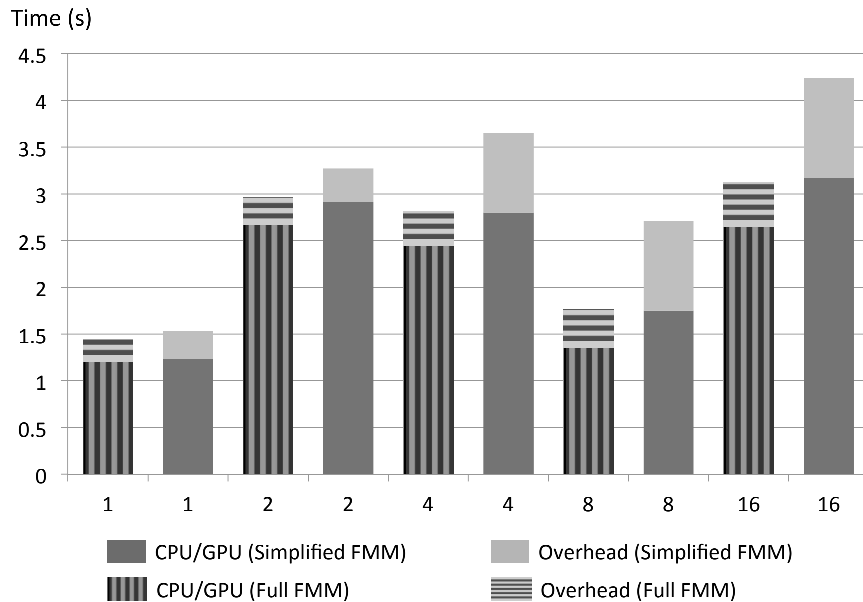


Figure 5.6: The CPU/GPU Concurrent Region Time and the Overhead of the Full FMM Algorithm against the Simplified FMM. The overhead is due to data transfer between the nodes and CPU/GPU sequential region. Tests are performed by using two GPUs per node and the testing case size increases proportionally to the number of nodes (8M particles per node). The time is measured for computations of the potential only.

data distribution can be configured such that only small amount of particle communication is needed. Moreover, as mentioned before, we could use the parallel partition methods in literature to minimize this cost.

5.3 Experimental Results

There is a *simplified* multi-node algorithm in Sec. 4.2.1 (also in [33]), which we use as baseline to compare to the *full* FMM algorithm present in Sec. 5.2.2 (the same

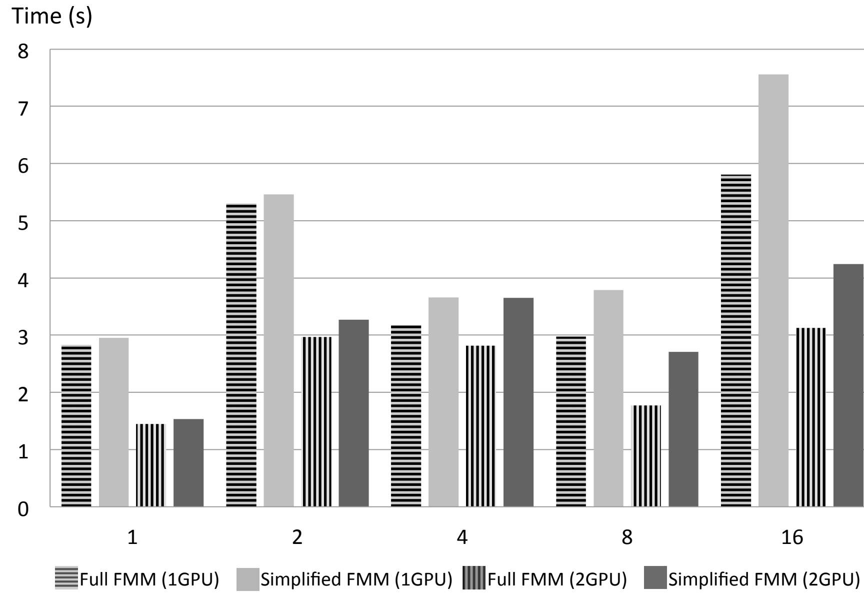


Figure 5.7: The Time Comparisons between the Full and Simplified FMM Algorithm. The testing case has 8M particles and run on 1, 2, 4, 8, 16 nodes using 1 or 2 GPUs on each node.

terms are used in all the figures of this section). The major difference between them is the data communication scheme. In the simplified algorithm, the partition boundary information is not available for computing nodes. Hence the master node has to collect and broadcast all the box's L-data in the end, which introduces big overheads. While in the present algorithm, the global communication requires light-weight pre-computations but exchanges much less amount of data. In this section, we will show that both the overheads and scalability of our new algorithm can be well improved with the aid of those new data structures. We define *concurrent region* here as the period when the GPU(s) computes local summation and the CPU cores compute translation simultaneously.

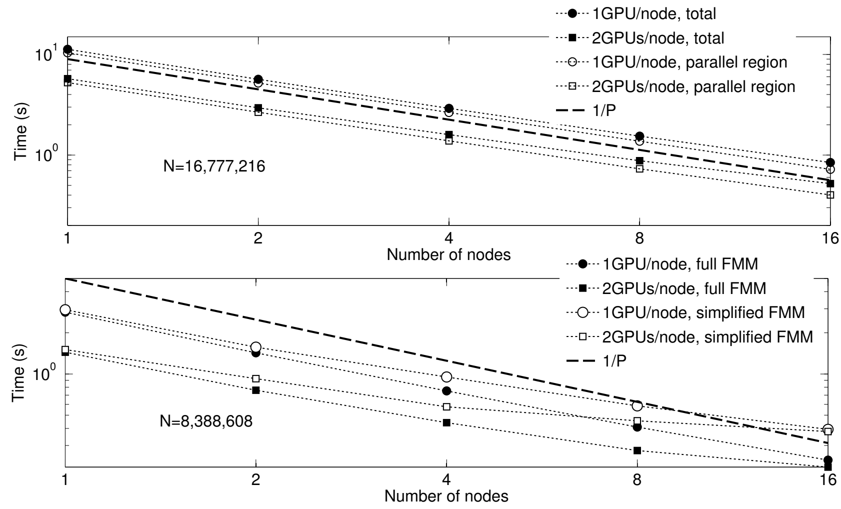


Figure 5.8: The Results of the Strong Scalability Test for 1 and 2 GPUs per Node of the Testing Cluster. The thick dashed line shows perfect scalability $t = O(1/P)$. The time is measured for potential only computations. In the top figure, it shows the present algorithm's performance and the problem size is fixed to be 16M running on 1 to 16 nodes. In the bottom figure, it shows the strong scalability comparison (total run time) between the simplified and the full FMM algorithm . The problem size is fixed to be 8M running on 1 to 16 nodes.

Firstly, the weak scalability of our algorithm was tested by fixing the number of particles per node to $N/P = 2^{23}$ and varying the number of nodes (see Table 5.3). In Fig. 5.6, we show our overhead vs. concurrent region time against the baseline algorithm performance. For perfect parallelization/scalability, the run time in this case should be constant. In practice, we observed an oscillating pattern with slight growth of the average time. In [33], two factors were explained which affect the perfect scaling: reduction of the parallelization efficiency of the CPU part of the algorithm and the data transfer overheads,

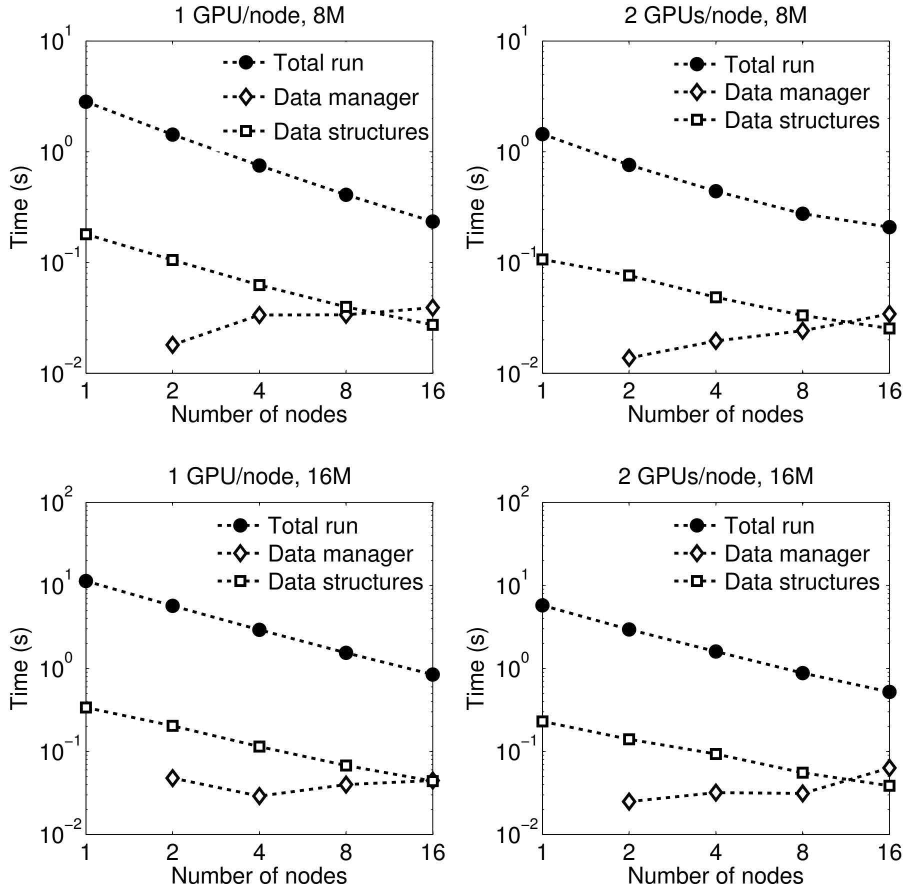


Figure 5.9: The Data Manager and Data Structure Processing Time against the Total Run Time. The problem sizes are fixed to be 8M (top) and 16M (bottom) running on 1 to 16 nodes. Each node uses 1 (left) or 2 (right) GPUs. The time is measured for potential only computations.

which also applies to our results. We distribute L2L-translations among nodes and avoid the unnecessary duplication of the data structure, which would become significant at large sizes. Since our import/export data of each node only relates to the boundary surfaces, we improve the deficiency of their simplified algorithm that also shows up in the data transfer overheads, which increases with $l_{\max}$.

Time (s) \ N (P)	8M (1)		16M (2)		32M (4)		64M (8)		128M (16)	
Num of GPUs	1	2	1	2	1	2	1	2	1	2
CPU wall	1.07	1.07	4.71	0.60	2.53	2.44	1.39	1.35	0.83	0.81
CPU/GPU	2.38	1.20	4.71	2.66	2.53	2.44	2.43	1.35	5.22	2.65
Overhead	0.45	0.24	0.59	0.30	0.64	0.37	0.54	0.42	0.58	0.47
Total run	2.83	1.44	5.30	2.96	3.17	2.81	2.97	1.77	5.80	3.12

Table 5.3: Performance for P Heterogeneous Nodes with $N/P = 2^{23}$. Results are for potential only.

In Fig. 5.6, our new FMM algorithm shows almost the same parallel region time for the cases with similar particle density (the average number of particles in a spatial box at l_{max}). Moreover, the overheads of this algorithm only slightly increases in contrast to the big jump seen the baseline algorithm when l_{max} changes. Even though the number of particles on each node remains the same, the problem size increases hence results in the deeper octree and more spatial boxes to handle, which also contributes to such overhead increase (besides communication cost). As for the total run time comparisons, we summarize the improvements in Fig. 5.7. Generally speaking, as the problem size and octree depth increase, our algorithm shows substantial savings against the baseline algorithm, which implies the much improved weak scalability.

Secondly, we also performed the strong scalability test, in which N is fixed and P is changing (Fig. 5.8). The tests were performed for $N = 2^{23}, 2^{24}$ and $P = 1, 2, 4, 8, 16$

with one and two GPUs per node. Even though, our algorithm demonstrates superior scalability compared with the baseline algorithm, we still observe the slight deviations from the perfect scaling for the 8M case. For 16M case, the total run time of both 1 and 2 GPU shows the well scaling because the GPU work was a limiting factor of CPU/GPU concurrent region (the dominant cost). This is consistent with the fact that the sparse MVP alone is well scalable. For 8M case, in the case of two GPUs, the CPU work was a limiting factor for the parallel region. However, we can see approximate correspondence of the times obtained for two GPUs/node to the ones with one GPU/node, i.e. doubling of the number of nodes with one GPU or increasing the number of GPUs results in approximately the same timing. This shows a reasonably good balance between the CPU and GPU work in the case of 2 GPUs per node, which implies this is more or less the optimal configuration for a given problem size.

Lastly, to validate the reduced cost of our communication scheme and the computation of box type, we compare the data manager processing time including M-data exchange time and the overall data structure construction time with the total running time in Fig. 5.9. Given the problem size and truncation number fixed, our communication increases as the number of nodes (roughly $P^{1/3}$ in Eq. 5.3). In our strong scalability tests, such time is in the order of 0.01 seconds while the wall clock time is in the order of 1 or 0.1 seconds (contribute $1\% \sim 15\%$ of overall time), even though GPUs are not fully occupied in some cases. This implies such cost can be neglected in larger problems, in which the kernel evaluations keep all GPUs fully loaded. Our implementation incorporate the box type computation with other data structures, such as octree and translation neighbors,

hence it makes more sense to report the total data structure cost. From Fig. 5.9 we observe that our data structure time decrease similarly as the wall clock time (as $1/P$) and shows good strong scalability.

However, it could be problematic if each node is only assigned a small number of boxes, which would occur given a large number of nodes. Eventually the subdivision of the domain would result in the number of boxes in the boundary region of each subdomain is more or less the same as that of domain itself. In this case, the number of boxes to exchange is almost the same as the total global spatial boxes. Note that although the data manager processes the box data searching and consolidating, its main cost comes from the communication but not those processing. Hence, all the traffic (each box has $2p^2$ coefficients) that must go through the master node will become the bottleneck of the entire system. However, this communication traffic issue is intrinsic to the splitting of the global octree. One possible mitigation might be implementing a many-to-many communication model. In fact, in our current implementation, each node is capable of computing the sending/requesting address (node IDs) of each export/import box and this further improvement by investigating communication cost is left for future work.

5.4 Summary

The need for a fast code for data structures is a manifestation of Amdahl's law. In a serial FMM code, generation of basic data structures usually takes a small portion of the total algorithm execution time. In some applications of the FMM, such as for iterative solution of large linear systems, this step is even less important as it is amortized over

several iterations. The typical way of doing the data structures is via an $O(N \log N)$ algorithm that uses sorting, which is usually done on the CPU [21]. However, for large dynamic problems, when the particle positions change every time step, the cost of this step would dominate, especially when the FMM itself is made very fast. The novel parallel data structures developed here are designed to resolve this issue and enable the FMM applications to large dynamic simulations.

The multiple node data structures developed here can handle non-uniform distributions and achieve workload balance. In fact, our algorithm splits the global octree among all the nodes and processes each sub-tree independently. Such a split can be treated as an isolated module which is free to use different methods based on different applications, to estimate workload. Moreover, since each node constructs its own sub-tree independently, the limitation of the depth of octee constructed by GPU only applies to the local tree, which implies such algorithm can handle deeper global octrees. Our approach using import and export box concepts only exchange necessary box data hence substantially reduces the communication overheads. We develop parallel algorithms to determine the import and export boxes in which the granularity is spatial boxes. Their parallel GPU implementations are shown to have very small overhead and good scalability.

Chapter 6: Scalable Vortex Methods Using Fast Multipole Methods

Fluid dynamics is commonly simulated on a stationary mesh (Eulerian), but can also be treated by particle methods (Lagrangian). Among the Lagrangian methods, vortex element methods (VEM) have the advantage of being able to directly capture the dynamics of vortex structures by using vortex elements as the basis of discretization. Lagrangian methods solve the convection term in the Navier-Stokes equation by explicitly moving the points, instead of updating the value at that point. Therefore, they are free from traditional time-integration constraints due to the Courant-Friedrichs-Lewy (CFL) condition [72], and free from numerical diffusion and dispersion (Note that time integration constraints are due to stability issues, and not diffusion/dispersion issues). Unlike implicit methods, there is no need to solve a linear system so the algorithm is highly parallel.

Another difference between vortex methods and conventional Computational Fluid Dynamics (CFD) methods is in the formulation, where vortex methods use the vorticity-velocity formulation, while conventional CFD use a pressure-velocity formulation. For flows around bodies, the vorticity-velocity formulation results in large savings since the vorticity has a much more compact support than the pressure. This is due to the fact that the vorticity is confined to a thin region near the wall and possibly a vortical wake.

Despite the advantages mentioned above, vortex methods have received only a small amount of attention from the CFD community in general due to the lack of validation in even the simplest turbulent flows. Vortex methods are a convergent method, but have multiple and little understood sources of numerical errors. First, even if a high-order diffusion scheme is used, there are often additional low-order sources of error. For example, the effect of the particle overlap ratio and the frequency of remeshing cannot be neglected. Another source of error is the stretching term calculation, which is highly sensitive to the spatial and temporal resolution due to its non-linearity. Also, for the regions with high shear, the stretching becomes intense and the flow field becomes highly anisotropic. This may impose additional constraints on the spatial and temporal resolution.

According to Helmholtz laws the vortex elements move with the local velocity of the fluid. The velocity at locations $\mathbf{y}_j, j = 1, \ldots, M$ induced by vortex elements at locations $\mathbf{x}_i, i = 1, \ldots, N$ can be computed as

$$\mathbf{v}_j = \mathbf{v}(\mathbf{y}_j) = \sum_{i=1}^{N} \frac{\boldsymbol{\omega}_i \times (\mathbf{y}_j - \mathbf{x}_i)}{|\mathbf{y}_j - \mathbf{x}_i|^3} = \nabla \times \frac{\boldsymbol{\omega}_i}{|\mathbf{y}_j - \mathbf{x}_i|}. \tag{6.1}$$

This equation is derived from the definition of vorticity $\boldsymbol{\omega} = \nabla \times \mathbf{u}$ and the incompressibility condition $\nabla \cdot \mathbf{u}$, so calculating the velocity in this manner ensures conservation of mass. For viscous flows, this singular kernel is smoothed by a Gaussian for the vorticity field, leading to the modified form of Eq. 6.1

$$\mathbf{v}_j = \sum_{i=1}^{N} \frac{\boldsymbol{\omega}_i \times (\mathbf{y}_j - \mathbf{x}_i)}{|\mathbf{y}_j - \mathbf{x}_i|^3} K_{\sigma_i}, \tag{6.2}$$

where the *cutoff function* $K(\mathbf{y}_j, \mathbf{x}_i)$ is defined by

$$K_\sigma(\mathbf{y}_j, \mathbf{x}_i) = \text{erf}\left(\sqrt{\frac{r_{ij}^2}{2\sigma_i^2}}\right) - \sqrt{\frac{4}{\pi}}\sqrt{\frac{r_{ij}^2}{2\sigma_i^2}}\exp\left(-\frac{r_{ij}^2}{2\sigma_i^2}\right), \tag{6.3}$$

where $r_{ij} = |\mathbf{y}_j - \mathbf{x}_i|$.

As mentioned earlier, convection is handled by moving the vortex elements, and diffusion is calculated by spreading the Gaussian at a rate that satisfies the analytical solution of the diffusion equation. As for the stretching term we insert Eq. 6.2 into

$$\frac{d\boldsymbol{\omega}_j}{dt} = \boldsymbol{\omega}_j \cdot \nabla \mathbf{v}_j, \tag{6.4}$$

which results in another N-body computation. Note that this involves the contraction of the vorticity pseudo-vector with the velocity gradient tensor, and this step has taken some previous authors a time equivalent to six scalar potential computations [26].

Direct evaluation of Eq. 6.1 or Eq. 6.4 on all $\mathbf{y}_j$ yields $O(N^2)$ cost, which cannot scale to large size simulations in practice. However, since the Biot-Savart kernel is composed of dipole solutions of Laplace equation, we can use FMM to approximate these sums to any precision ϵ at $O(N + M)$ cost [5,32]. The FMM itself is a divide-and-conquer approximation algorithm via *well separated pair decomposition* (WSPD) [10] data structures. It divides the kernel sum (Eq. 6.1) into a far-field and near-field term. The far-field term is approximated while the near-field term is evaluated directly. Such approximation can achieve up to machine precision accuracy, controlled by the truncation number p.

Regarding the FMM itself, the arithmetic intensity of the inner kernels and the hierarchical communication patterns allow it to scale well to large size clusters. Efficient parallel FMM algorithms [23,29,31] presented in the literature, including the 2009 winner

of the Gordon Bell prize [2]. A scalable FMM using heterogeneous architectures was developed which can calculated billion size problems on a small cluster (32 nodes) in [33]. References [2, 26, 30] present FMM algorithms on GPUs for vortex element methods to simulate isotropic turbulence in large scale.

Comparing with the Laplace potential kernel $1/|\mathbf{y}_j - \mathbf{x}_i|$, the computation of Eq. 6.1 and Eq. 6.4 require more operations. In this paper, we focus on the computational part of vortex methods by developing an efficient FMM algorithm for "velocity+stretching" evaluation on heterogeneous clusters, using the efficient new formulation.

We used FMM to speedup this computation and developed a new distributed heterogeneous FMM algorithm which can compute "velocity+stretching" by taking advantages of both algorithmic and hardware accelerations. This work combined the mathematics (Lamb-Helmholtz decomposition), algorithm (heterogeneous FMM), programming (highly optimized MPI-CUDA codes) and application (vortex element method) to extract the full compute capability of large GPU-based systems for physics simulations.

First, given the incompressibility constraint

$$\nabla \cdot \mathbf{v} = 0, \tag{6.5}$$

we provide an efficient FMM translation method to calculate "velocity+stretching" at a cost of only two Laplace potential kernels by using Lamb-Helmholtz decomposition [73, 74]. Based on this, we were able to develop a new heterogeneous FMM algorithm, which can efficiently map the new expansion and translation procedures on the distributed heterogeneous architectures.

Second, earlier FMM implementations for dynamic problems on distributed architecture shown both relatively large data structure and communication penalty. To separate the computation and communication to avoid synchronization during GPU computations, we developed new data structures build on the *local essential tree* (LET) [29,31] concept but have a novel parallel construction algorithm, in which the granularity is at the level of the spatial boxes (which allows finer parallelization than at the single-node level). Together with the algorithms for FMM octree data structures and interaction lists' constructions [36], we are able to reduce all data structure-related overhead substantially.

Finally, we improved the computation for Gaussian blob (used in the local sums) based on comprehensive timing and accuracy analysis. There are several classically used transcendental functions involved in the sum of the cutoff functions (Eq. 6.3) which are expensive. We developed their accurate and inexpensive approximations on GPU and demonstrate a large performance gain. Moreover, since Gaussian computations on GPUs arise in other contexts, our contributions are likely useful widely.

6.1 FMM for Vortex Methods

6.1.1 Lamb-Helmholtz Decomposition

If the velocity field is not constrained, each Cartesian component of the velocity is an independent harmonic function, which requires three harmonic FMMs in total to determine the velocity field. However, given the divergence constraint in Eq. 6.5, the total

velocity field can be described by only two harmonic scalar potentials, ϕ and χ as

$$\mathbf{v}(\mathbf{r}) = \nabla\phi(\mathbf{r}) + \nabla \times (\mathbf{r}\chi(\mathbf{r})), \tag{6.6}$$

where $\nabla^2\phi = 0$ and $\nabla^2\chi = 0$. The theory of such decomposition and translation is established in [73]. The FMM based on such decomposition is similar to normal harmonic potential FMM. For presentation purposes, we define source/receiver box as the spatial box containing at least one source/receiver points.

Initially, all source and receiver points are spatially grouped via *octree* space division recursively (as spatial boxes) and sorted according to their locations (*Morton index*). For each receiver point $\mathbf{y}_j$, its kernel sum is split into near and far-field terms given a neighborhood $\Omega(\mathbf{y}_j)$

$$\mathbf{v}(\mathbf{y}_j) = \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} \nabla \times \frac{\boldsymbol{\omega}}{|\mathbf{y}_j - \mathbf{x}_i|} + \sum_{\mathbf{x}_i \in \Omega(\mathbf{y}_j)} K_\sigma \frac{\boldsymbol{\omega}_i \times (\mathbf{y}_j - \mathbf{x}_i)}{|\mathbf{y}_j - \mathbf{x}_i|^3}. \tag{6.7}$$

The neighborhood $\Omega(\mathbf{y}_j)$ corresponds to the box containing $\mathbf{y}_j$ at the maximal level l_{max}. The latter term is called local direct sum (P2P) and evaluated at l_{max} directly for each $\mathbf{y}_j$.

For the former term, we compute its approximation by using expansions and translations over spherical basis functions in four steps: data structure construction, initial expansion, upward pass, downward pass, and final sum. We elaborate the stages of the FMM in the following 7 steps:

1. **Data structure construction**: Build all interaction lists and translation stencils for source/receiver boxes.

2. **Multipole expansion of the Laplace Green's function (P2M)**: At the finest level l_{max}, all source data points are expanded to p^2 terms at their box centers.

The M-expansion coefficients $\{M_n^m\}$ from all source points in the same box are consolidated into a single expansion.

3. **Multipole expansion conversion**: Each source box converts $\{M_n^m\}$ to $\{\phi_n^m, \chi_n^m\}$ at level l_{max}(see [73]).

4. **Upward pass (M2M)**: For levels from l_{max} to 2, each child source box performs the multipole-to-multipole translation on $\{\phi_n^m, \chi_n^m\}$ to the parent box and all children boxes' contributions are consolidated into a single M-expansion.

5. **Downward pass (M2L, L2L)**: For levels from 2 to l_{max}, local or L-expansions corresponding to $\{\phi_n^m, \chi_n^m\}$ are both created at each receiver box. Each source box performs multipole-to-local (M2L) translations to receiver boxes according to the special stencil (see [21]) which satisfies the WSPD. Each receiver box performs Local-to-local translation (L2L) to its child boxes. All the L-expansion coefficients are consolidated at each receiver box center.

6. **Local expansion conversion**: Each receiver box computes the coefficients of the velocity local expansions $\{_x v_n^m, _y v_n^m, _z v_n^m\}$ for all three dimensions from $\{\phi_n^m, \chi_n^m\}$. Each receiver box also computes local expansions coefficients of the gradient of velocity from $\{_x v_n^m, _y v_n^m, _z v_n^m\}$. Note that this new way to calculate the gradient coefficients for a box is much less expensive than previous approaches.

7. **Final summation (L2P, P2P)**: Evaluate the velocity and stretching components by $\{_x v_n^m, _y v_n^m, _z v_n^m\}$ and their corresponding gradient expansion coefficients for all receiver points at level l_{max}.

6.1.2 Scalable Heterogeneous FMM Algorithm

Since the two most time-consuming parts in FMM, the local summation (P2P) and multipole-to-local translations (M2L) can be performed concurrently, the FMM algorithm can be accelerated by mapping these on different parts of a typical CPU-GPU heterogeneous architecture and communication minimized. In [33], it was shown that high efficiency can be achieved by assigning boxes-related computations (i.e., the translations which require complex data structures and irregular memory access pattern) to multi-core CPUs and the particle-related computations (i.e., the local sums of receivers) to many-core GPUs. Because our FMM for vortex element methods is equivalent to performing two potential FMMs, which has already been studied intensively in [33], we can use the same strategy as there to distribute the algorithm components to CPUs and GPUs for the best performance.

6.1.3 On A Single Node

A single heterogeneous computing node consists of one or more multi-core CPUs and connects to one or more GPUs over a PCI-Express bus. Figure 6.1 shows how each component of the algorithm in Sec. 6.1.1 is mapped onto the hardware. We define the *CPU/GPU concurrent region* as the overlapped blocks where GPUs and CPUs process their own data concurrently and the wall clock time is determined by the slower task. The bottom concurrent region shown in Fig. 6.1 is the most time-consuming part and reflects how well the overall algorithm performs. GPUs keep all the source/receiver data in global memory and are responsible for time-step updates. All

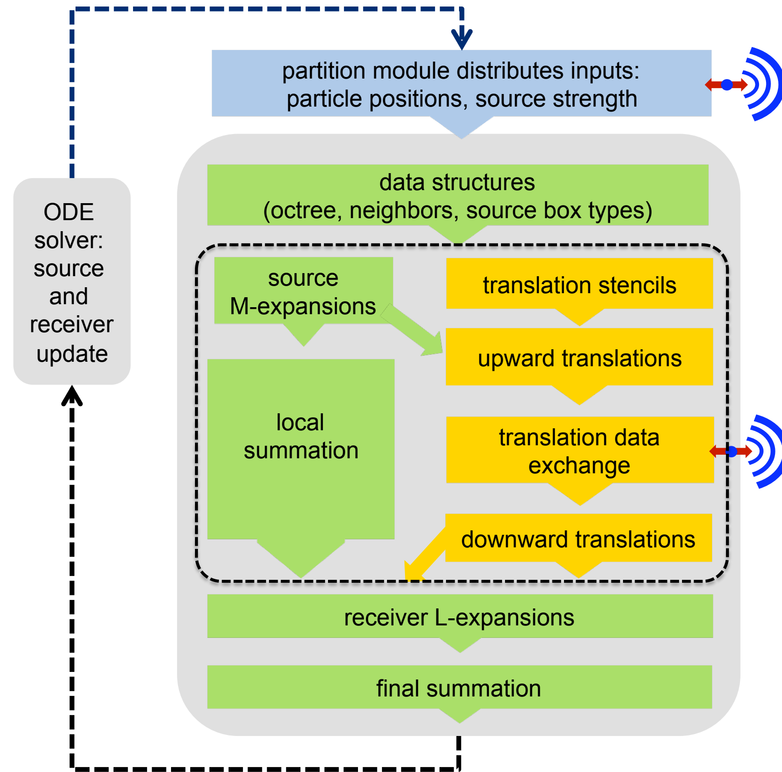


Figure 6.1: The Single-Node FMM Algorithm. The light gray parts of the algorithm are computed on the GPU while the dark gray parts are computed on the CPU. The rectangle with dash lines represents the concurrent region, where concurrent steps are performed on the CPU and GPU.

the data structures and interaction lists (such as neighbor lists for local summations), M/L-expansions, expansion conversions and receivers' local summation are performed in parallel on GPUs. Computations on GPUs use single precision, since this precision in the final results is often sufficient. CPUs only request the box information, such as expansion coefficients and translation stencil list, and provide the translated coefficients from/to GPUs before/after the concurrent region, which minimizes the cost of CPU-GPU data transfer. Once these coefficients are in host main memory, CPUs execute

translations in parallel on multi-threaded cores. Note that the CPU performs double precision computations, which is helpful since the translations are more sensitive to round-off errors.

6.1.4 Multiple Nodes

The essence of the FMM is to group source/receiver points and process them as hierarchical groups. This inherent spatial decomposition property allows us to distribute the FMM algorithm on multiple nodes efficiently. This distributed algorithm splits the global octree structures based on global workload balance and uses a Master-Slave model (Fig. 6.2), in which there is a one-time (initial) data distribution among all the nodes. All the heavy computation steps are performed independently on each node without any further inter-node communications. We use the concepts of *partition level, critical level, box types*, introduced in Chapter 5.

These box types can be computed efficiently on GPUs with very small overheads, once the partition is determined. See [35] for the algorithmic details. Note that this classification enables each node to export and import only necessary data. More precisely, only boxes on the boundaries of the partitions—which are 2D surfaces (in contrast to 3D volumes)—need to be sent/received. There is also a *data manager* on the master node. Its function is to collect/redistribute all the exported coefficients from/to slave nodes. Since we skip all empty boxes, addressing a particular box's data requires its identification (using the box Morton index) and searching operation. We can avoid searching and thread divergence by constructing a histogram [33], and implement this algorithm efficiently

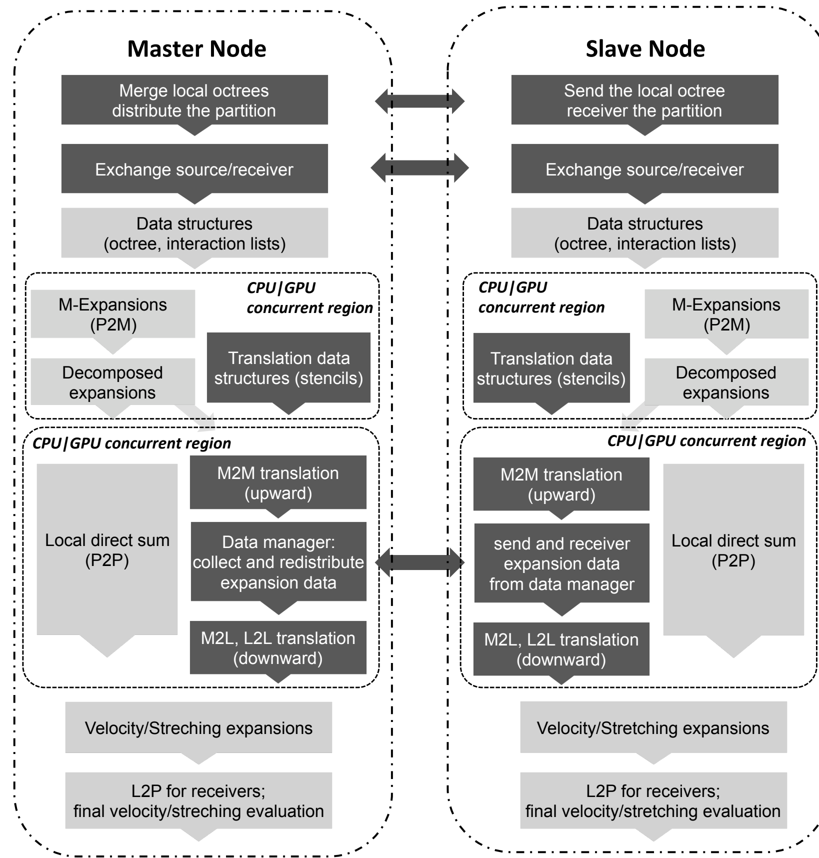


Figure 6.2: An Overview of the Distributed FMM Algorithm. The parts processed on the GPU are shaded light gray, while the parts processed on CPU are shaded dark gray. The two parts on the top represent the partitioning algorithm, where the temporary local trees are also constructed on GPUs. This initial partitioning is independent of the FMM algorithm and one can use any partitioning algorithm based on the application requirements. There is a one time initial data exchange during the upward pass.

on GPUs. However, since the amount of data is quite small, it is counterproductive to implement this data manager on GPUs given the data transfer cost between CPU and GPU. Now assume all the necessary data structures for the translations, such as the box's global Morton indices, the neighbor lists, export/import box type lists and M-expansion

data, are available on the CPUs. Then, each node J executes the following translation algorithm:

1. **Upward translation pass**:

   (a) Get the M-data (initial $\{\phi_n^m, \chi_n^m\}$) of all domestic source boxes at l_{max} from global memory.

   (b) Perform M2M translations for all domestic source boxes at levels $l = l_{max} - 1, \ldots, \max(2, l_{crit})$.

   (c) Pack M-data of export boxes and the import/export box indices of all levels. Then send them to the data exchange manager.

   (d) The master node, which is also the manager, collects data. For the incomplete root box M-data from different nodes, it sums them together to get the complete M-data. Then according to each node's export/import box indices, it packs the corresponding M-data and sends them back.

   (e) Receive import M-data of all levels from the data exchange manager.

   (f) If $l_{crit} > 2$, consolidate M-data for root domestic boxes at level l_{crit}. If $l_{crit} > 3$, produce M-data for all domestic source boxes at levels $l = l_{crit} - 1, \ldots, 2$.

2. **Downward translation pass**:

   (a) Produce L-data (translated $\{\phi_n^m, \chi_n^m\}$) for all receiver boxes at levels $l = 2, \ldots, l_{max}$.

   (b) Output L-data for all receiver boxes at level l_{max}.

(c) Redistribute the L-data on GPU.

(d) Each GPU consolidates the L-data, adds the local sums to the dense sums and copies them back to the host according to the original input order.

Each node separates the computations between CPU cores and GPUs as in the single node case, except there is a one-time communication with the data manager during the upward pass. Assume we have P nodes and the particle cluster size is c_s. This cluster size is the maximum number of particles at the finest level. On one hand, the partition algorithm assigns N/P sources, which are located compactly in a domain $\Omega_j^{(s)}, j = 1, \ldots, P$, covered by $N_j^{(s)} \approx N/(c_s P)$ spatial boxes at level l_{max} to each node J. In this algorithm, the node J only performs M2M, M2L and L2L translations for its own assigned domain $\Omega_j^{(s)}$, i.e. $N_j^{(s)}$ octree spatial boxes. On the other hand, the node J is also assigned M/P receivers and needs to evaluate the kernel function directly for approximately $27c_s M/P$ source-to-receiver interaction pairs. Hence, there are no repeated operations among nodes, and the total computation cost for each node, dominated by either $O(N/P)$ or $O(M/P)$, is proportional to $1/P$, which makes it truly scalable.

6.2 Vortex Core Function Evaluation

The vortex blobs are often represented by Gaussian distributions with a core radius σ_i. The cutoff function $K(\mathbf{y}_j, \mathbf{x}_i)$ for velocity induced by the superposition of these Gaussian distributions is in Eq. 6.3 while the cutoff function $G(\mathbf{y}_j, \mathbf{x}_i)$ for the stretching

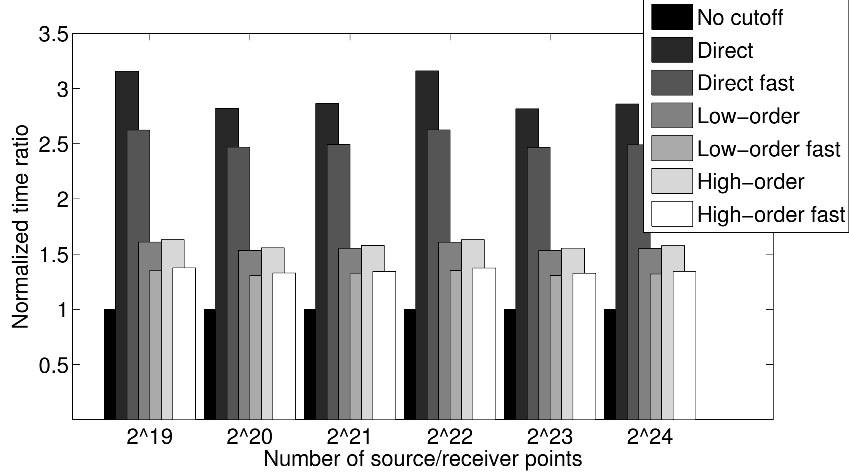


Figure 6.3: Time Comparisons among Different Cutoff Implementations. All times are normalized by the time of local summations without cutoff functions. "Direct" means cutoff functions are implemented using the erf CUDA device-function-call on GPUs; Low-order means the approximation by Eq. 6.9 while high-order means the approximation by Eq. 6.10; "fast" appendix means using the less accurate but faster compilation option.

term is

$$\begin{aligned} G(\mathbf{y}_j, \mathbf{x}_i) &= 3\text{erf}\left(\sqrt{\frac{r_{ij}^2}{2\sigma_i^2}}\right) - \left(2\frac{r_{ij}^2}{2\sigma_i^2} + 3\right)\sqrt{\frac{2r_{ij}^2}{\pi\sigma_i^2}} e^{-r_{ij}^2/2\sigma_i^2} \\ &= 3K(\mathbf{y}_j, \mathbf{x}_i) - 2\frac{r_{ij}^2}{2\sigma_i^2}\sqrt{\frac{2r_{ij}^2}{\pi\sigma_i^2}} e^{-r_{ij}^2/2\sigma_i^2} \end{aligned} \tag{6.8}$$

where $r_{ij} = |\mathbf{y}_j - \mathbf{x}_i|$. $K(\mathbf{y}_j, \mathbf{x}_i)$ and $G(\mathbf{y}_j, \mathbf{x}_i)$ need to be computed during the local summation for velocity and stretching calculation. On the GPU, direct evaluations of the transcendental function erf(x) in $K(\mathbf{y}_j, \mathbf{x}_i)$ and $G(\mathbf{y}_j, \mathbf{x}_i)$ are very expensive as shown in Fig. 6.3.

In our efficient implementation of these terms, we use two computational schemes to reduce this cost. Let us denote the more general form of $K(\mathbf{y}_j, \mathbf{x}_i)$ as $C(x) = \text{erf}(x) -$

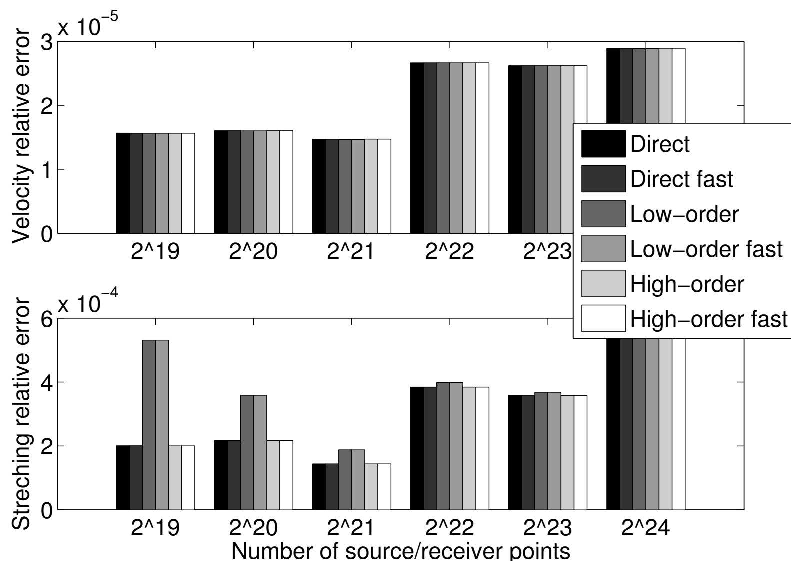


Figure 6.4: The Relative Error Comparisons among Different Cutoff Implementations with the Truncation Number $p = 16$. The legends are the same as Fig. 6.3. The top plot shows the velocity errors while the bottom plot shows the relative errors of the stretching term which is more singular.

$e^{-x}\sqrt{4x/\pi}$. Firstly, since we only implement the single precision calculation on GPU, there is no need to distinguish 1.0f from any function value $v \in (0.999999, 1.000001)$ due to round-off errors. Consider that $C(x)$ is a monotonically increasing function with $C(+\infty) \to 1$ and $1.0 - C(4.0) < 10^{-6}$. We can set $C(x) = 1$ for all $x > 4.0$f, so that most function calls in Eq. 6.3 and Eq. 6.8 can be omitted. Secondly, instead of directly evaluating erf(x), we can approximate it by using a low degree polynomial plus an exponential term. Two options are shown in Eq. 6.9 and Eq. 6.10 with their error bounds and all the coefficients are listed in [75]. Since we use CUDA, there is also a compiler option `use-fast-math` to speedup the exponential and square root functions by calling their intrinsic counterparts with less accuracy.

	Velocity only		Velocity+stretching	
Number of Particles	Direct	Cutoff	Direct	Cutoff
262144	0.492	1.177	0.814	1.787
524288	0.356	0.537	0.436	0.818
1048576	0.506	1.330	0.843	1.989
2097152	1.700	4.931	2.819	7.376
4194304	2.545	3.611	2.780	5.569
8388608	3.291	9.317	5.574	13.98
16777216	11.70	35.53	19.78	53.24

Table 6.1: Time Comparisons between with Cutoff and without Cutoff Functions. In all the cases with cutoff function evaluations, it is the GPU local summation dominates the overall time. However, in the case of $N = 2^{22}$, the maximal level increases from 4 to 5. In the direct kernel evaluation version, it is the CPU translation that dominates the overall time, which explains the time decreasing (compare with the $N = 2^{21}$ case).

$$\mathrm{erf}(x) = 1 - t(a_1 + t(a_2 + ta_3))e^{-x^2} + \epsilon(x) \tag{6.9}$$

where $t = 1/(1 + d_1 x)$ and $|\epsilon(x)| \leq 2.5 \times 10^{-5}$.

$$\mathrm{erf}(x) = 1 - t(b_1 + t(b_2 + t(b_3 + t(b_4 + tb_5))))e^{-x^2} + \epsilon(x) \tag{6.10}$$

where $t = 1/(1 + d_2 x)$ and $|\epsilon(x)| \leq 1.5 \times 10^{-7}$.

From Fig. 6.3, we can see that the approximations reduce the overhead caused by the transcendental functions substantially, and the fast compilation option can further

save about 15%. However, such fast evaluation methods are seen not to compromise the overall accuracy. Figure 6.4 shows that all these approximations even with the less accurate compilation option have no effect on the velocity accuracy. For the stretching term computation, due to its higher singularity, the low-order approximation, Eq. 6.9, introduces larger errors but the high-order approximation produces almost the same accuracy as direct cutoff function evaluation. Given the small processing time difference between Eq. 6.9 and Eq. 6.10 according to Fig. 6.3, our implementation uses Eq. 6.10 together with the fast GPU device function compilation option to evaluate $K(\mathbf{y}_j, \mathbf{x}_i)$ and $G(\mathbf{y}_j, \mathbf{x}_i)$ during the local summation.

6.3 Performance Tests

In all of our tests, we assume the worst case, where all initial data is globally distributed randomly. In such a case, exchanging $O(N + M)$ data among nodes can not be avoided. This assumption makes the initial partitioning expensive. However, in most applications or dynamic problems, large inter-node communications only occur at the initial step. Therefore, the timings of our algorithm exclude this initial partitioning time. The kernel defined by Eq. 6.1 and Eq. 6.4 have larger singularity and based on our accuracy tests, using $p = 12$ or larger yields the accepted accuracy for our applications. Hence in most tests, we set the truncation number to $p = 12$. Even though our implementation treats sources and receivers as different sets, our performance benchmarks assume that they have the same size, i.e. $N = M$. To make the presentation succinct, we abbreviate "potential computation" as potential, "velocity computation" as

Time (s) \ N	1,048,576		2,097,152		4,194,304		8,388,608		16,777,216		33,554,432	
Num of GPUs	1	2	1	2	1	2	1	2	1	2	1	2
CPU wall clock	0.66	0.65	0.66	0.66	5.87	5.85	5.87	5.87	5.85	5.86	–	49.84
C/G concurrent region	1.43	0.75	5.64	2.96	5.87	5.86	10.98	5.87	43.07	49.84	–	49.84
Velocity+Stretching	2.11	1.18	6.93	3.73	7.82	6.93	14.64	7.85	50.01	25.47	–	61.13
Velocity	1.42	0.91	4.80	2.62	6.91	6.44	10.04	6.94	35.25	17.86	–	54.66
Potential	0.46	0.43	1.47	0.86	3.14	2.96	3.49	3.15	10.78	5.51	26.64	25.26

Table 6.2: The Single-Node Performance Profiling. All results are obtained in their best settings with $p = 12$. The CPU translation and parallel region timings are for velocity+stretching. The last three rows are for total run time.

velocity and "velocity+stretching computation" as velocity+stretching. We also define the relative ratio $r_p(T)$ of the velocity or velocity+stretching computation time T to the potential computation time T_p as $r_p(T) = T/T_p$.

6.3.1 Hardware

We used a 32-node heterogeneous cluster ("Chimera") at the University of Maryland to perform tests. Each node has dual quad-core Intel Xeon X5560 2.8 GHz processors, 24 GB of RAM per node, and two Tesla C1060 accelerators each with 4 GB of RAM, interconnected via Infiniband.

6.3.2 Single Node Performance

Compared to the potential+force, evaluations of the velocity+stretching expansion coefficients have a noticeable increase in computational cost. However, the performance of the heterogeneous algorithm is still dominated by the concurrent region (defined in Sec. 6.1.3). The time profile by using one or two GPUs is summarized in Table 6.2, where the results are reported in the optimal settings (in terms of the tree depth l_{max}). Note that only when the local sums dominate the total time, using the second GPU can achieve high parallelization efficiency, which implies the necessary condition to scale the number of GPUs. Moreover, the detailed analysis in [33] can be applied here to show linear scalability with respect to the problem size. We also test the algorithm for non-uniform distributions, which requires deep tree depths and results in non-uniform loads for GPU threads (This is not preferred by current hardware architectures in terms of performance. However, in future, this non-uniform load perhaps provides possible power saving on the green computing hardware, which could automatically turn on/off the cores based their workload). However, the performances of both the velocity and stretching kernels remain consistent with the potential kernel in [33] with respect to the time increase.

The second test is used to demonstrate the computational cost of velocity and stretching by comparing with the potential. Figure 6.5 summarizes their relative ratios to the potential where the timings are obtained by using two GPUs. Note that there is no cutoffs for potential and its heterogeneous algorithm cost is also dominated by either the local sum or translations.

In the simple circumstance when both FMMs (potential vs. velocity/stretching)

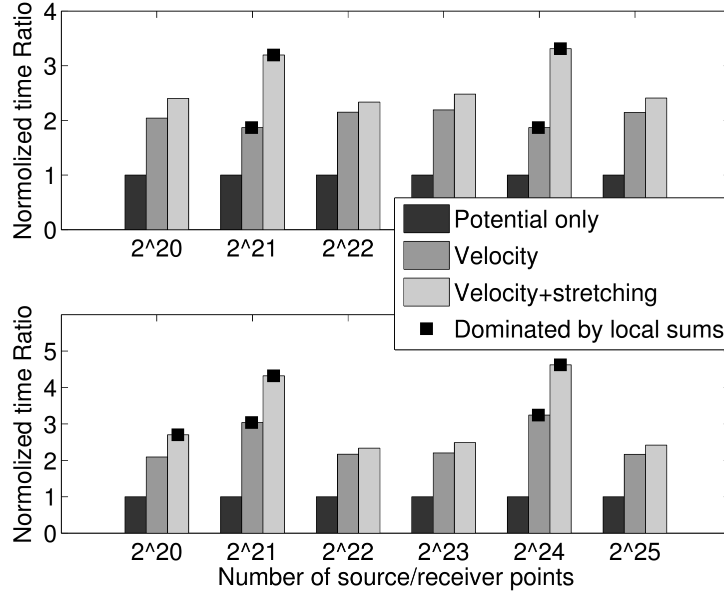


Figure 6.5: Time Comparisons between the Potential and Velocity+Stretching Computations Using 2 GPUs. All times are shown as the relative ratios to the scalar potential FMM (no cutoff function involved), which is shown in the dark block with height 1.0. In the top figure, the velocity and stretching are computed without cutoff functions. In the bottom figure, the computation times include cutoff functions of Eq. 6.3 and Eq. 6.8. The cases where the overall time is dominated by the local summations are denoted specially by the solid square blocks since their performance behaviors are complicated. In other cases where the time is dominated by the translations, the cost jump ratio of velocity+stretching is only as high as 2.5 times a single potential calculation even with cutoff function calculations.

are dominated by translations, we expect the velocity/stretching cost to be twice that of the potential plus some overhead for the coefficients' conversion. This is confirmed in Fig. 6.5. For velocity, the relative ratio r_p is around 2 without cutoff computation, while

with it this ratio jumps to 3. For velocity+stretching, this algorithm is still roughly as twice slow as that for the potential kernel. Given the extra cost of the computation of expansion coefficients, the relative ratio r_p is about 2.4 regardless using the cutoff computations or not. However, when local summation dominates the concurrent region, the behavior of the algorithm is complicated. Recall that the GPU sparse matrix vector product (SMVP) has an optimum performance for a certain data cluster size [21]. Hence, relative costs vary depending on the kernel, the cluster size, and the hardware. The best performance requires computation fine tuning. For example, kernels with and without cutoffs can be considered as different which might result in different optimal settings. In Fig. 6.5, we observe that under the current best settings without cutoffs, the cost increase can be 1.8 times for velocity or 3.3 times for velocity+stretching in the worst case. When calculating the cutoff functions, the cost ratio to potential can be as high as 3.2 for velocity or 4.7 for velocity+stretching, respectively. Note that the cases where the cluster sizes are the same (such as $N = 2^{21}, 2^{24}$ or $N = 2^{22}, 2^{25}$) have a similar cost jump ratio.

6.3.2.1 Accuracy

To test the accuracy of the algorithm, we vary the truncation number p and the problem size. Note that the data partitioning among nodes does not introduce any error, hence it is sufficient to perform the error analysis on a single node. For each receiver point, the "true value" is computed by directly summing the kernel values using double precision on the CPU. The error of each testing case is measured by the averaged L_2 relative norm based on 100 randomly selected receiver points (this number of samples is

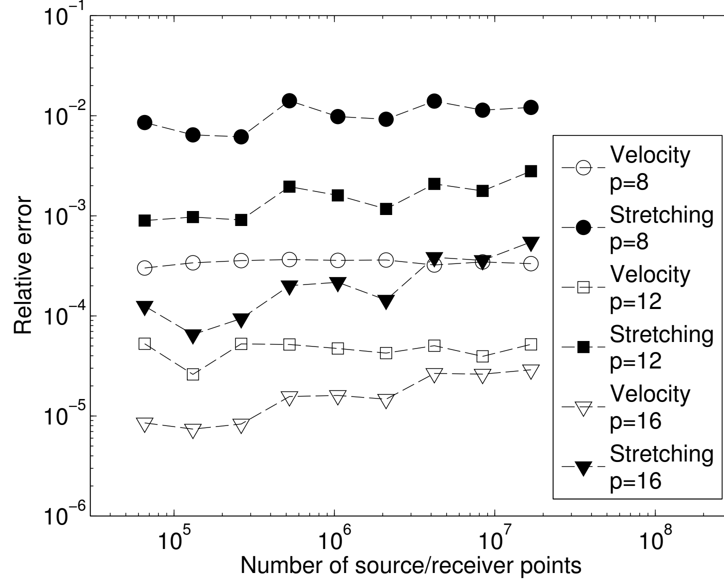


Figure 6.6: The Relative Errors of Velocity and Stretching Computation. All computations on GPUs use single precision.

sufficient for the L_2-norm error according to [12]).

The relative L_2-norm error of both velocity and stretching are summarized in Fig. 6.6 with the high-order cutoff approximation. Their singularities require much larger truncation numbers than the Coulomb kernel for potential ($p = 8$ is sufficient to make the relative error 10^{-6} [33]). It is clearly seen in Fig. 6.6 that the amount of error differs between Eq. 6.1 and Eq. 6.4, and that the accuracy improves by increasing p. Furthermore, we also observe the translation error propagation as the maximal level increases. For example, in the case of stretching using $p = 16$, there is a discrete jump whenever l_{max} changes (in our best settings, $l_{max} = 4, 5$ and 6 for $N = \{2^{16}, 2^{17}, 2^{18}\}$, $\{2^{19}, 2^{20}, 2^{21}\}$ and $\{2^{22}, 2^{23}, 2^{24}\}$). Note that this phenomena can only occur for the high-accuracy case, in which the error introduced by the initial expansion truncation is much smaller than errors

Time (s) \ N (P)	8M (1)		16M (2)		32M (4)		64M (8)		128M (16)		256M (32)	
Num of GPUs	1	2	1	2	1	2	1	2	1	2	1	2
CPU wall	5.86	5.87	3.13	3.09	13.38	13.38	7.13	7.14	4.04	4.07	18.03	17.86
CPU/GPU	10.91	5.87	21.36	10.69	13.38	13.38	11.89	7.14	23.61	11.81	18.03	17.86
Velocity+stretching	14.56	7.87	25.05	12.95	17.28	15.54	15.72	9.28	27.49	14.29	22.36	20.41
Velocity	9.98	6.90	17.61	9.02	15.50	14.74	10.85	8.35	19.43	10.08	20.27	19.83
Potential	3.47	3.11	5.89	3.08	7.09	6.79	4.21	3.92	5.92	3.21	9.74	9.46

Table 6.3: Performance Profiling for P Heterogeneous Nodes with $N/P = 2^{23}$ (8M). The CPU translation and concurrent region timings are for velocity+stretching. Results are obtained in their best settings using $p = 12$.

caused by translations.

6.3.3 Multiple-Node Performance

The performance evaluations in this section consist of four parts: weak scalability test, strong scalability test, overhead analysis, and large size runs. In all tests, we tune performance to obtain the best settings by varying the tree depths, problem sizes and only results at the best settings are reported. Although our algorithm can handle non-uniform distributions, the analysis of cost would be complex and problem-dependent but provide less insight. Hence we assume the uniform distribution for all the tests with $N = M$. For this case, the cost of each node can be estimated as

$$T = g_0 \frac{N}{P} + g_1 8^l + g_2 \frac{8^l}{P} + \max\left(c_0 \frac{8^l}{P} + T_c, g_3 \frac{N}{8^l} \frac{N}{P} \right) \tag{6.11}$$

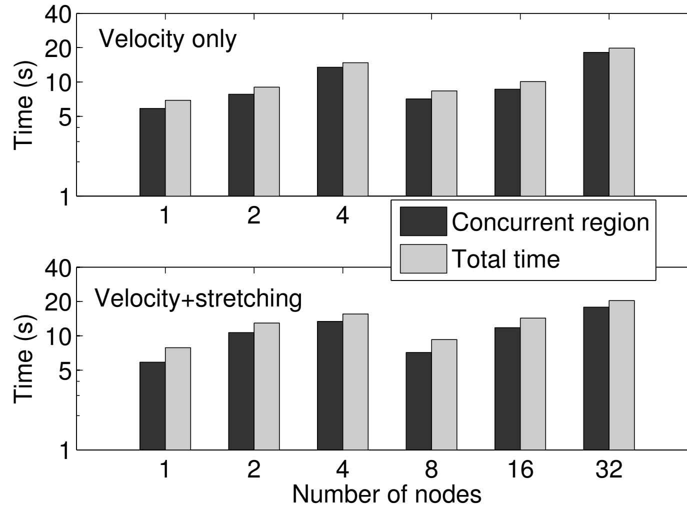


Figure 6.7: The Weak Scalability for $N = M = 2^{23}$ Particles per Node. The top figure shows the timing results of velocity only, while the bottom figure shows the results of velocity+stretching.

where g_i and c_j represent the cost ratio on GPU and CPU, respectively. T_c is the communication cost of the data manager and its cost can be estimated by $T_c = (8^{l_{max}}/P)^{2/3}$ (it is proportional to the 2D surface area of a 3D body). Note that the first three terms in Eq. 6.11—the particle-related process (such as sort, expansion), global tree construction and coefficient conversions—only take a small amount of the overall cost. The max function represents the dominant term—the concurrent region cost.

6.3.3.1 Weak Scalability

In the weak scalability test, the number of source/receiver points on each node is fixed to $N = M = 2^{23}$ and P varies from 1 to 32. The detailed profiling data is shown in Table 6.3. The ideal case should have a constant run time for any P. However, from

Fig. 6.7, we can observe time fluctuation caused by three factors according to Eq. 6.11. Firstly, the global problem size increases as P, which also requires larger tree depth l_{max}. Although each node is only in charge of a sub-domain (roughly $8^{l_{max}}/P$ boxes at l_{max}), the tree has to be constructed globally (top-down approach). The only difference is that many empty boxes are skipped with less memory consumption but they have to be processed with costs. Therefore, the overhead of spatial data structure constructions increases with the number of nodes. Secondly, the translation cost is determined by the number of non-empty boxes ($T_t = c_0 8^{l_{max}}/P$). When the maximal level increases by 1 the number of cells increases 8 times while P is only doubled. This will increase T_t by 4 times. However, further increase in P will reduce translation costs and change the time dominance (from translation to the local sum). Thirdly, while the maximal level remains the same, if we double P, the global size of the problem N is also doubled. As a consequence, the cost of the local summation on each node $T_s = g_3 N^2/(8^l P)$ is doubled. All these factors result in the time fluctuation shown in Fig. 6.7.

6.3.3.2 Strong Scalability

To test strong scalability, we use two different problem sizes $N = M = 2^{24}$ and 2^{25} and vary the number of nodes P from 1 to 32 using 1 or 2 GPUs per node. Since a single GPU cannot handle more than 32M particles' velocity updates because of its memory size, the timing of 1 node using 1 GPU is skipped. The timing results of $N = 2^{24}$ are summarized in Fig. 6.8 for both velocity and velocity+stretching. Note that, except for the data manager (which has very small asymptotic constant), the workload on each

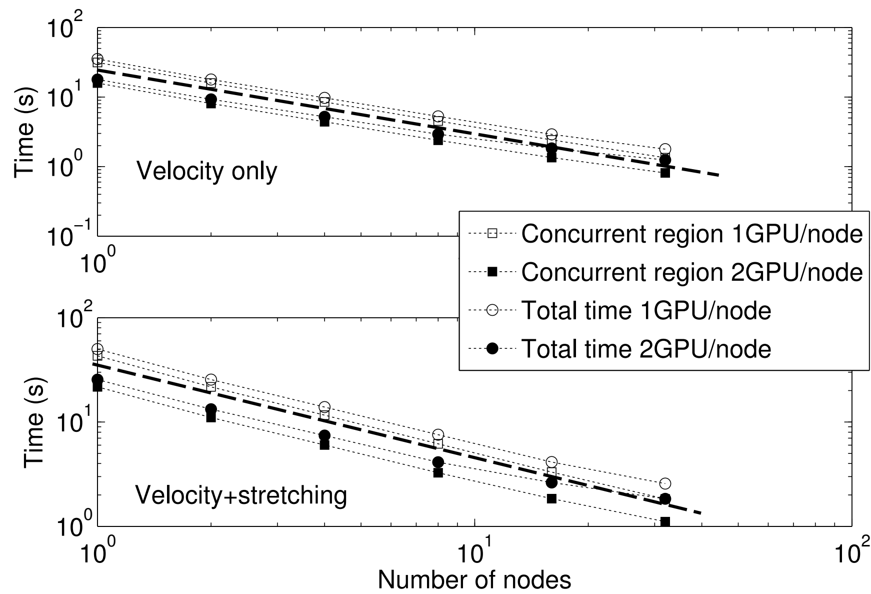


Figure 6.8: The Strong Scalability Test. The problem size N is fixed to be 2^{24} and the number of nodes vary from 1 to 32 using 1 or 2 GPUs each. In the top figure, it shows the concurrent region and the total run time for velocity, while the bottom figure is for the velocity+stretching.

node is proportional to $1/P$ since the maximal level and problem size are fixed. In the case of $N = 2^{24}$, the local sum dominates the total cost, hence, using the second GPU reduces the overall cost by around 50%. Note this nearly 100% parallel efficiency can only be achieved when the hardware is fully loaded. As P doubles, its total time also reduces by around half, which is consistent with the fact that the local summation alone is scalable. For $N = 2^{25}$, translations dominate the overall cost. Hence the utilization of the second GPU does not reduce the total time very much and the savings only come from computations of the expansions and their efficient conversions. However as P increases, the parallel efficiency deceases because of the increased workload of the data manager

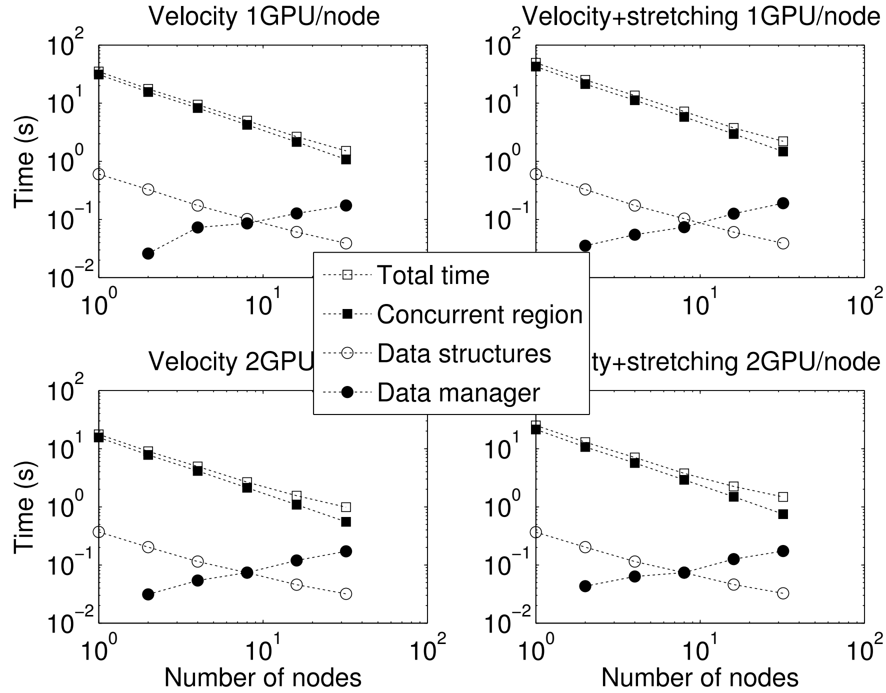


Figure 6.9: Overheads vs. the Total Time. The problem size N is fixed to be 2^{24}. The data structure and data manager overheads are plotted to compare with the concurrent region and the total run time. The two figures in the left column are for velocity, while the figures in the right column are for velocity+stretching.

and the inter-node communications. In the case of 32 nodes, the parallel efficiency drops to 76%. Even though its effect on the overall performance is small, further research efforts should be devoted to this issue.

6.3.3.3 Algorithm Overheads

We use a master-slave model to manage the communication among all the nodes. However, this scheme perhaps will raise many questions about scalability. In [35], we found that this communication overhead has little effect on the scalability and its cost is

shown in $O(P^{1/3})$. Moreover, since the data manager collects and distributes box data in the translation module, its overheads can be hidden if the overall time is dominated by the local summations on GPUs, therefore it has no effect on the scalability. In the Fig. 6.9, the low cost of the data manager is confirmed (where it is still less than 10% in the worst case) in comparison to the overall time.

However, it could be problematic if each node is only assigned a small number of boxes, which would occur given a large number of nodes. Eventually the subdivision of the domain would result in the number of boxes in the boundary region of each sub-domain is more or less the same as that of domain itself. In this case, the number of boxes to exchange is almost the same as the total global spatial boxes. Note that although the data manager processes the box data searching and consolidating, its main cost comes from the communication but not those processing. Hence, all the traffic (each box has $2p^2$ coefficients) that must go through the master node will become the bottleneck of the entire system. However, this communication traffic issue is intrinsic to the splitting of the global octree. One possible mitigation might be implementing a many-to-many communication model. In fact, in our current implementation, each node is capable of computing the sending/requesting address (node IDs) of each export/import box and this further improvement by investigating communication cost is left for future work.

6.3.3.4 Billion-Size Run

By fully employing 32 nodes of Chimera, we vary the problem size $N = M$ from 2^{20} to 2^{30} and compare the performance of potential, velocity and velocity+stretching.

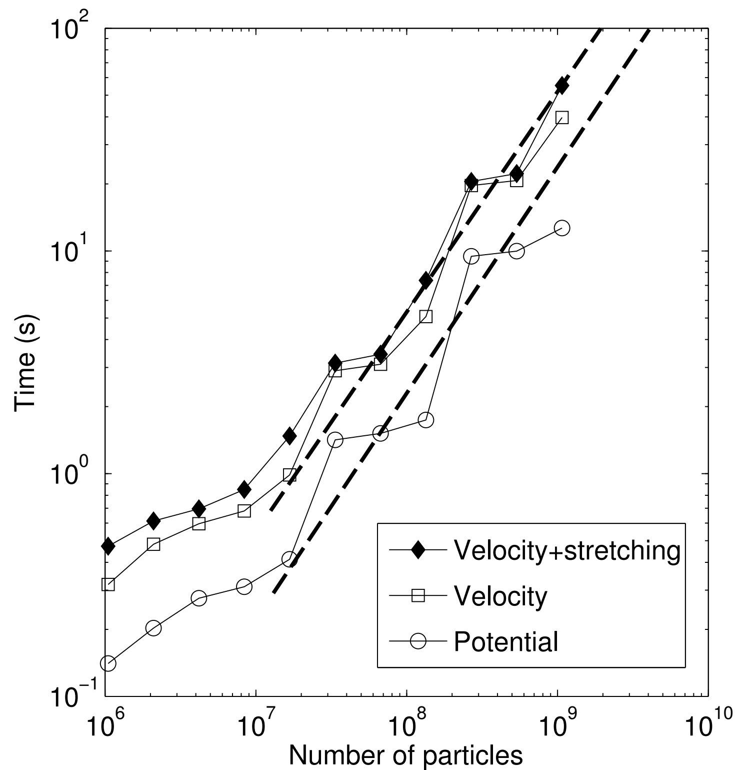


Figure 6.10: The Total Run Time on 32 Nodes Using 1 or 2 GPUs Each. Computations are performed for potential only, velocity only and velocity+stretching with $p = 12$. For each individual problem, only the best time between 1 and 2 GPUs is reported. The largest size we run is 1 billion particles, in which case source and receiver points are different.

The timing results are summarized in Fig. 6.10, in which only the best time between using 1 and 2 GPUs for each N is shown. As N increases, the wall clock time increases linearly in all cases, while the run time has a jump when l_{max} is increased. Unlike for the potential, where for the same l_{max} the time increases more or less at the same rate, both velocity and velocity+stretching have a big time jump before l_{max} changes. This is due to their local summations (with cutoffs) requiring much more floating point operations than for a single potential. For $N = M = 2^{30}$, the velocity takes 39.9 s and the

velocity+stretching takes 55.9 s, while the potential itself uses 12.6 s. Note that these timings and their corresponding relative ratios to the potential are consistent with our single node analysis (see Fig. 6.5). To estimate the floating operations, we use the same operation counts as [26] for velocity (70) and stretching (115). In the billion-particle case, the local summation dominates the total run time and the cost of each receiver can be estimated by $(70 + 115) \times 27N/8^{l_{max}}$. Therefore, this yields 49.12 Tflop/s on 32 nodes with 64 GPUs (933 Gflop/s peak performance reported by NVIDIA) for the 1 billion velocity+stretching computation in total. The soundness of FMM-based VEM using GPUs was validated in [26] and we run the same isotropic turbulence simulation as there and observe the similar statistics to validate our implementation.

6.4 Summary

The FMM originally solves potential problems with free-field boundary conditions. However, the method can be extended to handle periodic boundary conditions by placing periodic images around the original domain and clustering them into large cells, [76]. There are two reasons why periodic FMMs add almost no computational overhead to the original FMM. The first is the fact that further domains are clustered into larger and larger cells, so the extra cost of considering another layer of periodic images is constant. The second reason is that only the sources need to be duplicated and the evaluation points exist only in the original domain. Since the work load for considering the periodicity is independent of the number of particles, it becomes negligible as the problem size increases. We have confirmed in an earlier study that periodic boundary conditions adds

3 % more calculation time for a million particles [77]

In this chapter, we present both algorithmic and implementation improvements of FMM for vortex element methods. First, the Lamb-Helmholtz decomposition allows us to reduce the translation complexity in FMM to only two scalar potentials, which enables calculation of velocity or velocity+stretching by only doubling the cost of a potential FMM if the translation dominates the overall performance. This saving also allows us to use a larger truncation number p for higher accuracy far-field approximation at a smaller cost. Second, we develop an economical way to compute the cutoff functions without compromising accuracy. In the case when the local summation dominates the overall time, we reduce the cost of velocity+stretching from 6, which is reported in literature [26], to 4.8 times at most. Finally, we extend this vortex element method FMM algorithm to multiple heterogeneous nodes. We compute the communication data structure with box level granularity in parallel on GPU and only necessary data are exchanged among nodes with small overheads. Based on our performance tests, this multiple-node algorithm demonstrates both good strong and weak scalability.

The heterogeneous model developed in the paper enables utilization of the current popular multi-core high-performance computer hardware. A single workstation is capable of calculating the velocity+stretching up to 32 million particles within 1 minute while a small cluster with 32 nodes can compute 1 billion-particles in 55.9 s. Finally, we demonstrate the state of the art performance of FMM for vortex method and the capability of using this fast algorithm together with small or moderate heterogeneous clusters to solve large-size practical problems in many fields.

Chapter 7: Conclusions and Future Work

Algorithmic improvements presented in this dissertation substantially extend the scope of application of the FMM for large N-body simulation problems, especially for dynamic problems. In these problems, the fast algorithm for data structures are required otherwise it would be a performance bottleneck since the entire source and receiver points change their positions at every time step. Our proposed data structures in Chapter 2 which can be constructed in parallel efficiently on accelerators, such as GPUs or many-core processors, enable application of FMM to high fidelity dynamic simulations. The overall construction time of data structures was reduced dramatically by comparing with the normal CPU sequential implementation. Moreover, the approaches for fast spatial data hashing and interaction list constructions could be also useful in many other problems.

Based on those data structures, we developed a fully GPU-based FMM implementation, which was illustrated a dynamic simulation of the interactions between vortex rings in Chapter 3. Except for initial setup, our approach processed all the computations and updates on a single GPU, which simultaneously providing interactive visualization of the simulation as it proceeded. Since resources from OpenGL and Direct3D could be mapped into the address space of CUDA, this pure GPU-based approach could fully utilize graphics interoperability by dumping the computation results

on to screen with little host (CPU) interruption overheads. In multiple GPU environments, Scalable Link Interface (SLI) interoperability could allow one GPU render other GPU's resources via SLI (without data transfer via PCI-Express bus and host interruption) hence provided possible graphics rendering performance gain.

The single GPU algorithm was extended to heterogeneous architectures in Chapter 4. This was done by substantially simplifying the GPU job and offloading some tasks to the CPU. By decoupling translations and local direct sum, which are the two main time consuming parts of FMM, to the CPU and the GPU separately. We could achieve state of the art N-body computation performance. This also provided a full load on the CPUs, enabled efficient double precision computations, and brought other benefits of parallel use of the CPU cores and GPUs. With the present algorithm, dynamic problems of the order of ten million particles per GPU can be solved in a few seconds per time step, extending computation capabilities of single workstations equipped with one or several GPUs and relatively small (several node) low-cost heterogeneous clusters.

The distributed heterogeneous cluster solution with special data structures, developed in Chapter 5 can handle non-uniform distributions and achieve workload balance. In fact, our algorithm splits the global octree among all the nodes and processes each sub-tree independently. Such a split can be treated as an isolated module which is free to use different methods based on different applications, to estimate workload. Moreover, since each node constructs its own sub-tree independently, the limitation of the depth of octee imposed by GPU memory only applies to the local tree, which implies this algorithm can handle deeper global octrees. Our approach using the newly introduced import and export box concepts only exchanges necessary box data hence substantially

reduces the communication overhead. We developed parallel algorithms to determine the import and export boxes in which the granularity corresponded to spatial boxes. Their parallel GPU implementations were shown to have very small overhead and good scalability.

Based on the work developed in Chapters 2–5, we proposed a new approach to use FMM to evaluate the most time-consuming kernels—the Biot-Savart equation and stretching term of the vorticity equation for incompressible flows simulations. We used a mathematically reformulation in [73] so that *only two* Laplace scalar potentials are used *instead of six*. This automatically ensured divergence-free far-field computation. Based on this formulation, we developed a new FMM-based vortex method on heterogeneous architectures, which distributed the work between multi-core CPUs and GPUs to best utilize the hardware resources and achieve excellent scalability. The algorithm uses new data structures which can dynamically manage inter-node communication and load balance efficiently, with only a small parallel construction overhead. This algorithm can scale to large-sized clusters showing both good strong and weak scalability. Careful error and timing trade-off analysis was also performed for the cutoff functions induced by the vortex particle method. This implementation can perform one time step of the velocity+stretching calculation for one billion particles on 32 nodes in 55.9 seconds, which yields 49.12 Tflop/s.

In the following sections, we briefly describe the possible future research directions and discuss other potential effective approaches regarding FMM data structures and implementations.

7.1 Single Node Data Structures

In Chapter 2, the parallel data structures use histograms which keep counts for all the boxes (including empty boxes) at the maximal level. Also histograms are used to retrieve the rank of any box among non-empty boxes directly without searching, which keep an value for each empty box too. Since the number of boxes grows exponentially as the level increases, this technique of using histograms cannot deal with data distributions, which require $l_{max} > 8$, on current GPU hardware. In applications, the adaptive FMM can go to level as deep as 15 or more. However, the data structure construction time or efficiency analysis has not been reported and no parallel algorithms are developed either.

In our implementation, we use 32-bits integer to store the Morton index (space filling curve). Therefore, in 3D problems, the maximal level we can compute the Morton index is 10 (due to bit-interleaving). However, we might use long integers to store the Morton index which allows the maximal level as large as 21. The ideal data structure algorithm for FMM should also work for such a deep maximal level.

An interesting project would be to use other structures or re-implement histogram by using both sequential and parallel algorithms such that we can eliminate this limitation but not compromise performance. One possible idea is to develop a two-level algorithm:

- At a coarse level, which can be handled by GPU using histograms, we compute all data structures.
- For each non-empty box at the coarse level, we further divide it into smaller boxes at deeper levels.

In such kind of two-level algorithms, resolving storage issues is straightforward while counting or retrieving the rank of a box would become tricky, which may require complicated structures. Other data integrating problems such as merging box coefficients from different partitions (in the multi-GPU context) also need to be considered. It is still not clear that whether this method can address needed data quickly. Much more tuning and test experiments have to be performed. Such work should then be extended to multiple computing nodes as well.

Another way to overcome the huge memory consumption of histograms is to explore adaptive data structures and the corresponding adaptive FMM algorithms. To the best of our knowledge, such data structures have only been implemented by using sequential algorithms on CPU. The key idea of adaptive algorithm can be summarized as: given a unit cube and prescribed cluster size s, the adaptive algorithm recursively divides the cube into boxes. If certain boxes contain less than s data points, no further divisions of the sub-tree are performed.

Even though GPUs are not good at recursive algorithms, we could still perform such recursive calls on the CPU level by level, then let GPUs efficiently perform the data structure computation below a certain level. However, some technical barriers of such method can be expected and must be overcome:

- Many data transfers between CPU and GPU might be required, which might reduce the efficiency.

- The neighbor domains are no longer uniform, i.e., neighbor boxes might be at different levels of the tree. This might need quite different data structures from

those presented in Chapter 2. Hence, big modifications or totally different data structure algorithms are expected.

- How to quickly address the rank of non-empty boxes in parallel without search is still unclear if we skip all empty boxes.

7.2 Automatic Balanced FMM Algorithms

The heterogeneous algorithm in [33] has a pre-defined maximal level. In that case, the balance of CPU translation and GPU direct sum is not known to the algorithm itself but is passed to the algorithm as an input parameter. The user needs to perform the some experiments to determine the level such that CPU and GPU work are balanced, i.e., manually tune the input parameter such that optimal performance is achieved.

To our best knowledge, there is no work in literature has been done on automatic work balance in FMM. Existing algorithms pre-define one cluster size, and then determine the maximal level at which the direct local sums are performed. This is equivalent to pre-defining the maximal level in [33]. For uniform distributions, such automatic balancing can be realized by a naive method: starting from a prescribed level, for example level 3, we can compute the data structure similar to Chapter 2 to obtain box information. Once the non-empty receiver box and source box information are available, we can estimate the work of translation as well as the local direct sum. If they differ much, we go to next level until some maximal level (for example 10 if we use 32 bits to store the Morton index) or the rough balance is achieved.

However, for non-uniform distributions, finding the balanced parameter is not

trivial. For instance, it is likely that in some region the density of data points is high while it is low in some other regions. In that case, the level to perform the local direct sum might be different. Hence, this requires adaptive data structures, which we proposed in the above section. In future work, we would like to explore current available adaptive fast multipole methods, such as [13,23,78], and their corresponding data structures for possible solution in our framework. More exactly, based on these existing methods, we might be able to develop novel adaptive data structures and automatic balancing algorithms, and further parallelize them on GPU or distributed systems.

7.3 Partition Methods and Dynamic Data Structure Updates

The multiple node data structures in Chapter 5 are designed in the context of FMM, where it does not deal with the initial partitions. The partition method presented there is naive and only for the worst case. The profiling results in [33] show that such partition takes even more time than the FMM evaluation for very large size problems. This big overhead is due to two factors: one is distributing the data points to different nodes; the other is MPI data exchange. Even though the later factor has no big improvement based on our current hardware, we might implement a fast packing function to reduce this overhead. Possible approach can be developing fast parallel algorithm for GPU to pack the data, where we can take advantage of its high memory bandwidth and large number of cores to accelerate packing. However, similar histograms as Chapter 2 have to be used since the reduction operations are needed. An interesting problem is to develop both speed and memory efficient approaches to save this partition time.

As for the practical simulations, in which the positions of data points or particles are dynamically updated every time step, the data distributions actually do not change very much, i.e., only a relative small number of particles leave or enter each box. Note that, our FMM data structures are mainly lists of box related information. Once certain particles are within the same box, their order or positions in a box are irrelevant to the correctness of algorithms, which suggests that the algorithm should only care about whether the particles leave or enter a box but not how they appear in that box. Hence, it is very likely to balance the work among the nodes by exchanging only a small amount of data given the fact that the majority of particles stay in the same region of the previous step. Also within a node, such small changes will not affect the overall data structures much, which implies possible saving due to cheap data structure update (but not reconstruction). Since all the particle related information, such as position coordinates and its mass, are stored in a compact format as described in Chapter 2, it is however not clear whether the insert or delete operations can be performed sequentially or in parallel better than our current fast reconstruction scheme. Such performance validation requires fine-tuning experiments, code optimization and comprehensive analysis. Therefore, we can expect significant future work on such efficient dynamic data structure updating techniques to minimize this data exchange and construction overhead. Moreover, the ideal algorithm should be able to determine when to update the data structure and when to reconstruct it based on computation cost estimation. For this topic, we would like to consider the partition methods in literature first, then look for effective dynamic updating algorithms for vortex/particle simulations.

Bibliography

[1] NVIDIA, *NVIDIA CUDA C Programming Guide*, 5.0 ed., October 2012.

[2] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, "42 TFlops hierarchical n-body simulations on GPUs with applications in both astrophysics and turbulence," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 1–12, ACM, 2009.

[3] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with CUDA," in *GPU Gems 3* (H. Nguyen, ed.), ch. 31, pp. 677–695, Addison Wesley Professional, August 2007.

[4] A. Gualandris, S. P. Zwart, and A. Tirado-Ramos, "Performance analysis of direct n-body algorithms for astrophysical simulations on distributed systems," *Parallel Comput.*, vol. 33, pp. 159–173, April 2007.

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

[6] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, December 1986.

[7] T. Darden, D. York, and L. Pedersen, "Particle mesh Ewald: An $n \log(n)$ method for ewald sums in large systems," *J. Chem. Phys.*, vol. 98, pp. 10089–10092, June 1993.

[8] E. Darve, C. Cecka, and T. Takahashi, "The fast multipole method in parallel clusters, multicore processors and graphic processing units," *Comptes Rendus Mecanique*, vol. 339, pp. 185–193, 2011.

[9] G. K. Batchelor, *An Introduction to Fluid Dynamics*. Cambridge University Press, February 2000.

[10] P. B. Callahan and S. R. Kosaraju, "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *J. ACM*, vol. 42, pp. 67–90, January 1995.

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

[12] N. A. Gumerov and R. Duraiswami, "Fast multipole method for the biharmonic equation in three dimensions," *J. Comput. Phys.*, vol. 215, no. 1, pp. 363–383, 2006.

[13] N. A. Gumerov and R. Duraiswami, *Fast Multipole Methods for the Helmholtz Equation in Three Dimensions*. Oxford: Elsevier, 2004.

[14] N. A. Gumerov and 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, April 2005.

[15] NVIDIA, *OpenCL Programming Guide for the CUDA Architecture*, 3.1 ed., 2010.

[16] CAPS Enterprise, CRAY Inc, The Portland Group Inc (PGI), and NVIDIA, "Directives for Accelerators." http://www.openacc-standard.org/, 2013.

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

[18] NVIDIA, *Fermi Compute Architecture Whitepaper*, 2011.

[19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, pp. 80–113, March 2007.

[20] NVIDIA, "Microsofts DirectCompute API for GPU computing." https://developer.nvidia.com/directcompute, 2013.

[21] N. A. Gumerov and R. Duraiswami, "Fast multipole methods on graphics processors," *J. Comput. Phys.*, vol. 227, no. 18, pp. 8290–8313, 2008.

[22] L. Greengard and W. D. Gropp, "A parallel version of the fast multipole method," *Computers Mathematics with Applications*, vol. 20, pp. 63–71, 1990.

[23] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta, "A parallel adaptive fast multipole method," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, (New York, NY, USA), pp. 54–65, ACM, 1993.

[24] J. P. Singh, J. L. Hennessy, and A. Gupta, "Implications of hierarchical n-body methods for multiprocessor architectures," *ACM Trans. Comput. Syst.*, vol. 13, pp. 141–202, May 1995.

[25] S.-H. Teng, "Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation," *SIAM J. Sci. Comput.*, vol. 19, pp. 635–656, March 1998.

[26] R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, and K. Yasuoka, "Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence," *Computer Physics Communications*, vol. 180, no. 11, pp. 2066–2078, 2009.

[27] I. Lashuk, A. Chandramowlishwaran, H. Langston, T. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 58:1–58:12, ACM, 2009.

[28] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc, "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures," *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, pp. 1–12, 2010.

[29] I. Lashuk, A. Chandramowlishwaran, H. Langston, T. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast multipole method on heterogeneous architectures," *Commun. ACM*, vol. 55, pp. 101–109, May 2012.

[30] R. Yokota, T. Narumi, L. A. Barba, and K. Yasuoka, "Petascale turbulence simulation using a highly parallel fast multipole method," *ArXiv e-prints*, June 2011.

[31] L. Ying, G. Biros, D. Zorin, and H. Langston, "A new parallel kernel-independent fast multipole method," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, (New York, NY, USA), pp. 14–30, ACM, 2003.

[32] Q. Hu, M. Syal, N. A. Gumerov, R. Duraiswami, and J. G. Leishman, "Toward improved aeromechanics simulations using recent advancements in scientific computing," in *Proceedings 67th Annual Forum of the American Helicopter Society*, May 3–5 2011.

[33] Q. Hu, N. A. Gumerov, and R. Duraiswami, "Scalable fast multipole methods on distributed heterogeneous architectures," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 36:1–36:12, ACM, 2011.

[34] Q. Hu, N. A. Gumerov, and R. Duraiswami, "GPU accelerated fast multipole methods for dynamic n-body simulation," in *24th International Conference on Parallel Computational Fluid Dynamics (ParCFD 2012)*, (Atlanta, USA), May.

[35] Q. Hu, N. A. Gumerov, and R. Duraiswami, "Scalable distributed fast multipole methods," in *Proceedings of the 14th International Conference on High Performance Computing and Communications (HPCC-2012)*, HPCC '12, ACM, 2012.

[36] Q. Hu, N. A. Gumerov, and R. Duraiswami, "Parallel algorithms for constructing data structures for fast multipole methods," *ArXiv e-prints*, Jan 2013.

[37] Q. Hu, N. A. Gumerov, L. Barba, R. Yokota, and R. Duraiswami, "Scalable fast multipole for vortex methods," un-published.

[38] N. A. Gumerov, R. Duraiswami, and Y. 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, April 2003.

[39] J. Bédorf, E. Gaburov, and S. Portegies Zwart, "A sparse octree gravitational n-body code that runs entirely on the GPU processor," *Journal of Computational Physics*, vol. 231, pp. 2825–2839, Apr. 2012.

[40] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3* (H. Nguyen, ed.), ch. 39, pp. 851–876, Addison Wesley, August 2007.

[41] P. Ajmera, R. Goradia, S. Chandran, and 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, (New York, NY, USA), pp. 10:1–10:1, ACM, 2008.

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

[43] F. E. Sevilgen and S. Aluru, "A unifying data structure for hierarchical methods," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '99, (New York, NY, USA), ACM, 1999.

[44] B. Hariharan and S. Aluru, "Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods," *Parallel Comput.*, vol. 31, pp. 311–331, March 2005.

[45] L. Greengard and W. D. Gropp, "A parallel version of the fast multipole method," *Computers & Mathematics with Applications*, vol. 20, no. 7, pp. 63–71, 1990.

[46] H. Mahawar, V. Sarin, and A. Grama, "Parallel performance of hierarchical multipole algorithms for inductance extraction," in *Proceedings of the 11th international conference on High Performance Computing*, HiPC'04, (Berlin, Heidelberg), pp. 450–461, Springer-Verlag, 2004.

[47] S.-H. Teng, "Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation," *SIAM J. Sci. Comput.*, vol. 19, pp. 635–656, March 1998.

[48] F. A. Cruz, M. G. Knepley, and L. A. Barba, "PetFMM dynamically load-balancing parallel fast multipole library," *Int. J. Numer. Meth. Engng.*, vol. 85, no. 4, pp. 403–428, 2010.

[49] G. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, pp. 1526–1538, 1987.

[50] A. Selle, N. Rasmussen, and R. Fedkiw, "A vortex particle method for smoke, water and explosions," *ACM Trans. Graph.*, vol. 24, pp. 910–914, 2005.

[51] F. Losasso, J. Talton, N. Kwatra, and R. Fedkiw, "Two-way coupled SPH and particle level set fluid simulation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, pp. 797–804, July 2008.

[52] R. K. Beatson, J. B. Cherrie, and D. L. Ragozin, "Fast evaluation of radial basis functions: Methods for four-dimensional polyharmonic splines," *SIAM J. Math. Anal.*, vol. 32, pp. 1272–1310, 2001.

[53] N. A. Gumerov and R. Duraiswami, "Fast radial basis function interpolation via preconditioned Krylov iteration," *SIAM J. Scientific Computing*, vol. 29, no. 5, pp. 1876–1899, 2007.

[54] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg, "A progressive refinement approach to fast radiosity image generation," *SIGGRAPH Comput. Graph.*, vol. 22, pp. 75–84, June 1988.

[55] D. Groen, S. P. Zwart, T. Ishiyama, and J. Makino, "High-performance gravitational n-body simulations on a planet-wide-distributed supercomputer," *Computational Science & Discovery*, vol. 4, no. 1, p. 015001, 2011.

[56] R. Yokota and L. A. Barba, "Comparing the treecode with fmm on GPUs for vortex particle simulations of a leapfrogging vortex ring," *Computer & Fluids*, vol. 45, pp. 155–161, 2011.

[57] M. J. Stock and A. Gharakhani, "Toward efficient GPU-accelerated n-body simulations," in *46th AIAA Aerospace Sciences Meeting, AIAA 2008-608*, January 2008.

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

[59] A. J. Chorin, "Numerical study of slightly viscous flow," *Journal of Fluid Mechanics*, vol. 57, pp. 785–796, 2 1973.

[60] C. A. White and M. Head-Gordon, "Rotating around the quartic angular momentum barrier in fast multipole method calculations," *The Journal of Chemical Physics*, vol. 105, pp. 5061–5067, September 1996.

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

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

[63] D. A. Griffiths, S. Ananthan, and J. G. Leishman, “Predictions of rotor performance in ground effect using a free-vortex wake model,” *Journal of the American Helicopter Society*, vol. 49, October 2005.

[64] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, “Accelerating molecular modeling applications with graphics processors,” *Journal of Computational Chemistry*, vol. 28, no. 16, pp. 2618–2640, 2007.

[65] J. Makino, T. Fukushige, M. Koga, and K. Namura, “GRAPE-6: Massively-parallel special-purpose computer for astrophysical particle simulations,” *Publication of Astronomical Society of Japan*, vol. 55, pp. 1163–1187, December 2003.

[66] M. S. Warren, J. K. Salmon, D. J. Becker, M. P. Goda, and T. Sterling, “Pentium Pro inside: I. a treecode at 430 Gigaflops on ASCI Red, II. price/performance of $50/Mflop on Loki and Hyglac,” in *Proc. Supercomputing 97*, in CD-ROM. IEEE, 1997.

[67] R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, and K. Yasuoka, “Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence,” *Computer Physics Communications*, vol. 180, no. 11, pp. 2066–2078, 2009.

[68] M. S. Warren, T. C. Germann, P. Lomdahl, D. Beazley, and J. K. Salmon, “Avalon: An Alpha/Linux cluster achieves 10 gflops for $150k,” in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp. 1–10, 1998.

[69] A. Kawai, T. Fukushige, and J. Makino, “$7.0/Mflops astrophysical n-body simulation with treecode on GRAPE-5,” in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing ’99, (New York, NY, USA), pp. 1–6, ACM, 1999.

[70] A. Kawai and T. Fukushige, “$158/GFLOPS astrophysical n-body simulation with reconfigurable add-in card and hierarchical tree algorithm,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC ’06, (New York, NY, USA), pp. 1–8, ACM, 2006.

[71] A. Chandramowlishwarany, K. Madduri, and R. Vuduc, “Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’10, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2010.

[72] R. Courant, K. Friedrichs, and H. Lewy, “On the partial difference equations of mathematical physics,” *IBM J. Res. Dev.*, vol. 11, pp. 215–234, Mar. 1967.

[73] N. A. Gumerov and R. Duraiswami, “Efficient FMM accelerated vortex methods in three dimensions via the Lamb-Helmholtz decompositions,” *ArXiv e-prints*, January 2012.

[74] H. Lamb, *Hydrodynamics*. Cambridge, UK: Cambridge University Press, 1932.

[75] M. Abramowitz, *Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables*. Dover Publications, Incorporated, 1974.

[76] C. G. Lambert, T. A. Darden, Jr., T. A. Darden, and J. A. Board, “A multipole-based algorithm for efficient calculation of forces and potentials in macroscopic periodic assemblies of particles,” *J. Comput. Phys*, vol. 126, pp. 274–287, 1995.

[77] R. Yokota, T. K. Sheel, and S. Obi, “Calculation of isotropic turbulence using a pure Lagrangian vortex method,” *J. Comput. Phys.*, vol. 226, pp. 1589–1606, Oct. 2007.

[78] L. Ying, G. Biros, and D. Zorin, “A kernel-independent adaptive fast multipole algorithm in two and three dimensions,” *J. Comput. Phys.*, vol. 196, pp. 591–626, May 2004.