/

## Article / Book Information

| | |
|---|---|
| Title(English) | An FMM based on dual tree traversal for many-core architectures |
| Authors(English) | Rio Yokota |
| Citation(English) | Journal of Algorithms and Computational Technology, Vol. 7, No. 3, pp. 301–324 |
| Pub. date | 2013, 9 |

# An FMM Based on Dual Tree Traversal for Many-core Architectures

**R. Yokota**

King Abdullah University of Science and Technology

### Abstract

The present work attempts to integrate the independent efforts in the fast N-body community to create the fastest N-body library for many-core and heterogenous architectures. Focus is placed on low accuracy optimizations, in response to the recent interest to use FMM as a preconditioner for sparse linear solvers. A direct comparison with other state-of-the-art fast $N$-body codes demonstrates that orders of magnitude increase in performance can be achieved by careful selection of the optimal algorithm and low-level optimization of the code. The current N-body solver uses a fast multipole method with an efficient strategy for finding the list of cell-cell interactions by a dual tree traversal. A task-based threading model is used to maximize thread-level parallelism and intra-node load-balancing. In order to extract the full potential of the SIMD units on the latest CPUs, the inner kernels are optimized using AVX instructions.

## 1  Introduction

N-body simulations have traditionally been used in fields such as astrophysics and molecular dynamics, where the physics is naturally described by pairwise interaction of discrete bodies. However, applications for N-body solvers can be found in many other areas of physics including elastics, fluid dynamics, electromagnetics, acoustics, and quantum mechanics. This is made possible by transforming the partial differential equation governing the continuum field into an integral equation over discrete quadrature points. Therefore, N-body solvers are used in numerous scientific application codes and are considered as one of the key algorithmic components in scientific computing.

The fast multipole method (FMM) is a fast algorithm that reduces the complexity of the N-body problem from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. It is widely regarded as one of the top 10 algorithms in scientific computing [5], along with FFT and Krylov subspace methods. It is quite common that algorithms with low Byte/flop (dense linear algebra) have high complexity $\mathcal{O}(N^3)$, and algorithms with low complexity (FFT, sparse linear algebra) have high Byte/flop. The FMM has an exceptional combination of $\mathcal{O}(N)$ complexity and a Byte/flop that is even lower than matrix-matrix multiplication [38]. In other words, it is an *efficient* algorithm that is *compute* bound, which makes it an interesting alternative algorithm for many elliptic PDE solvers, on architectures of the future that will most likely have low Byte/flop.

The recent trend in computer architecture is shifting towards less and less Byte/flop ratios. A summary of the Byte/flop of flagship architectures from each vendor (as of August 2012) is shown in Table 1. From this table, one can see that the Byte/flop will soon fall below 0.2, which makes it very difficult for most algorithms to extract a high percentage of peak flop/s from these architectures. According to the roofline study by Williams *et al.* [34], sparse matrix-vector multiplication has about $4 - 5.88$ Byte/flop, stencil calculations have about $2 - 3$ Byte/flop, and 3-D FFT has about $0.61 - 0.92$ Byte/flop. It is clear how much the modern architectures are *off balance* compared to the requirements of these common algorithms. This trend will most likely continue, so it is necessary to rethink the underlying algorithms in scientific simulations. Matrix-free methods seem to have an advantage in this respect, and FMM can be viewed as one of them.

In a recent study we have shown that FMM becomes faster than FFT when scaling to thousands of GPUs [40]. The comparative efficiency between FMM and FFT can be explained from the asymptotic amount of communication. On a distributed memory system with $P$ nodes, a 3-D FFT requires two global transpose communications between $\sqrt{P}$ processes so the communication complexity is $\mathcal{O}(\sqrt{P})$. On the other hand, the hierarchical nature of the FMM reduces the amount of communication to $\mathcal{O}(\log P)$. A preliminary feasibility study for Exascale machines [15] indicates that the necessary bandwidth for FFT could only be provided by a fat-tree or hypercube network. However, constructing such network topologies for millions of nodes is prohibitive in terms of cost, and the current trend of using torus networks is likely to continue. Therefore, network topology is another area where the trend in hardware is deviating from the requirements of common algorithms. Hierarchical methods are promising in this respect, and FMM is undoubtedly one of them.

It is clear from the above arguments that FMM could be an efficient alternative algorithm for many scientific applications on Exascale machines. One common objection is that FMM requires much more operations than other fast algorithms like multigrid and FFT, and therefore is much slower. However, as future microarchitectures move towards less and less Byte/flop, the asymptotic constant of the *arithmetic* complexity becomes less of a concern. Therefore, the advantage in the *communication* complexity trumps the disadvantages as mentioned in the previous paragraph.

## 2 Related Work

FMM is a relatively new algorithm compared to well established linear algebra solvers and FFT, and has ample room for both mathematical and algorithmic improvement, not to mention the need to develop highly optimized libraries. We will briefly summarize the recent efforts in this area by categorizing them into; CPU optimization, GPU optimization, MPI parallelization, algorithmic comparison, auto-tuning, and data-driven execution.

Table 1: Byte/flop of modern microprocessors. GByte/s is the theoretical peak of the memory bandwidth, and Gflop/s is the theoretical peak of the double-precision arithmetic throughput when fused-multiply-add units are fully utilized, along with maximum clock frequency by turbo boost (if available).

| Vendor | Microarchitecture | Model | GByte/s | Gflop/s | Byte/flop |
|--------|------------------|-------|---------|---------|-----------|
| Intel | Sandy Bridge | Xeon E5-2690 | 51.2 | 185.6 | 0.276 |
| Intel | Knights Corner | Xeon Phi 5110P | 320 | 1010 | 0.317 |
| AMD | Piledriver | Opteron 6386 SE | 51.2 | 179.2 | 0.286 |
| AMD | Southern Islands | Radeon HD7970 (GHz Ed.) | 288 | 1010 | 0.285 |
| NVIDIA | Kepler | Tesla K20X | 250 | 1310 | 0.191 |
| IBM | PowerPC | PowerPC A2 (BG/Q) | 42.6 | 204.8 | 0.208 |
| Fujitsu | SPARC64 | SPARC64 IXfx (FX10) | 85 | 236.5 | 0.359 |

In terms of CPU optimization, Chandramowlishwaran *et al.* [6] have done extensive work on thread-level parallelization with NUMA-aware optimizations and manual SIMD vectorization. They performed benchmarks on four different microarchitectures; Harpertown, Barcelona, Nehalem-EP, and Nehalem-EX, and exploit the simultaneous multithreading features of the Nehalem processors. With the various optimizations, they were able to double the performance on Harpertown and Barcelona processors, while obtaining a 3.7x speed-up on Nehalem-EX, compared to their previous best multithreaded and vectorized implementation. Some of the optimizations are specific to the kernel-independent FMM (KIFMM) [36], and do not apply to FMMs based on Cartesian, spherical harmonics, or planewave expansions. It would be interesting to compare highly optimized kernels for these different types of expansions (including KIFMM) to determine at what order of expansion they cross over.

For GPU optimization for uniform distributions, Takahashi *et al.* [29] developed a technique to turn the originally memory bound M2L kernel into a compute bound GPU kernel by exploiting the symmetry of the interaction stencil. There are alternative techniques to make the M2L kernel compute bound on GPUs, by recalculating the M2L translation matrix on-the-fly [37]. However, the fact that the technique by Takahashi *et al.* precalculates the M2L translation matrix and still allows the M2L kernel to be compute bound, means it is that much more efficient. Increasing the number of particles per leaf [39] or calculating the M2L kernel on the CPU [20] is an easy optimization strategy for heterogenous architectures. The value of this work lies in the fact that they tackle the GPU-M2L optimization problem head on, and actually succeed in making it compute bound. One limitation of their method is that the particle distribution must be somewhat uniform in order to exploit the symmetry of the M2L translation stencil. Though, classical molecular dynamics is an important application that *will* be able to benefit from this method.

For GPU optimization of non-uniform distributions for low accuracy, Bedorf *et al.* [4] developed a highly optimized treecode `bonsai`[1] that runs entirely on GPUs. For three significant digits of accuracy in the force calculation for the Laplace kernel, their code is an order of magnitude faster than any other treecode or FMM on GPUs. Their code takes less than 50 milliseconds to calculate one million particles (including the tree construction) on an NVIDIA Fermi GF110 series GPU for the aforementioned accuracy, whereas other codes take about 500 milliseconds to achieve the same accuracy for one million particles. By using warp unit execution and `__ballot()` functions they have eliminated all synchronization barriers from their CUDA kernel. Furthermore, by using mask operations and prefix sums, they have eliminated all conditional branching from their code. However, an application which requires high accuracy may find this code difficult to use, since the order of expansion is not adjustable past $p > 3$. Nonetheless, anyone who has experience optimizing fast N-body codes knows that optimizing for low accuracy is much more challenging than optimizing high accuracy codes, so the impact of this work is remarkable.

For MPI parallelization of an adaptive tree, Winkel *et al.* [35] propose a request based point-to-point communication scheme that dedicates a thread for communication on each node, while the evaluation is done using the remaining threads simultaneously, in their code `PEPC`[2]. Meanwhile, Jetley *et al.* [21] used `CHARMM++` to dynamically load-balance on large scale heterogenous environments, obtaining large gains by task aggregation and memory pooling, in their code `ChaNGa`[3]. These efforts to provide an alternative to bulk-synchronous communication models could eventually become advantageous, but the lack of comparison against current state-of-the-art bulk-synchronous methods [26, 40] that also scale to the full node of the largest supercomputers, makes it difficult to predict exactly when they will become advantageous (if they do at all).

---

[1]https://github.com/treecode/Bonsai
[2]https://trac.version.fz-juelich.de/pepc
[3]http://software.astro.washington.edu/nchilada

As another aspect of MPI parallelization of an adaptive tree, Winkel *et al.* use a 64-bit global Morton key and partition the domain to equally distribute the number of keys, while using the traversal time of the previous step as weights. This technique will only allow a maximum depth of the global tree of 21 levels before the 64-bit key overflows. Current large scale simulations already exceed this depth [26], and the method of Winkel *et al.* will require the use of multiple integers to store the key in such cases. Looking into the future, it would seem that the use of ever more integers to store the key will slow down the sorting of these keys, but the fact that Bedorf *et al.* [4] can sort over a billion 96-bit keys per second using the `back40computing` library[4] means that this remains a viable option. It would be interesting to compare this approach with alternative techniques that eliminate the use of global keys altogether [40].

For algorithmic comparison, Fortin *et al.* [14] compare the BLAS based adaptive FMM by Coulaud *et al.* [8], `NEMOtree1`[5] implementation of the Barnes & Hut treecode [3], the `GADGET-2`[6] code by Springel *et al.* [28] (without using the treePM part), and `falcON`[7] code by Dehnen [9]. The comparison is for 6 different datasets with various non-uniformity, and the time-to-solution and memory consumption was compared for different problem sizes and various number of cores, while the accuracy was set to roughly 3 significant digits for the force. There was over an order of magnitude difference in the speed between these codes, `falcON` being consistently the fastest for all types of non-uniform distributions, and exhibiting perfect $\mathcal{O}(N)$ complexity throughout the examined range of $N$. A major drawback of `falcON` is that it is not parallel. In the present work, we extended the `falcON` framework to MPI, pthreads, OpenMP, Intel TBB, SSE, and AVX based parallelism, as shown later in this article.

An example of auto-tuning efforts in FMMs is given by Yokota & Barba [38], where the dual tree traversal approach by Dehnen [9] was combined with an auto-tuning mechanism by selecting between M2L and M2P kernels to produce a treecode-FMM hybrid code, and by selecting between M2L and P2P kernels to optimize the number of particles at the leaf at which P2P is performed. This is an advantage when developing a black-box library that runs on both CPUs and GPUs without having to worry about selecting the optimal number of particles per leaf for each architecture. This code – `exaFMM`[8] – inherits all the techniques of `falcON` such as the dual tree traversal, mutual interaction, error aware local optimization of the opening angle $\theta$, and a multipole acceptance criterion based on $B_{max}$ and $R_{max}$ [9]. At the same time, it adds features such as arbitrary order of expansion by using template metaprogramming, periodic boundary conditions, CUDA version of all FMM kernels, and a scalable MPI implementation. The present work is a further extension of this code to thread-level and SIMD parallelism.

Last but not least, there have been a few recent attempts to use event-driven runtime systems for optimizing the data flow of parallel FMM codes. Ltaief & Yokota [24] used `QUARK`[9] to schedule the threads dynamically according to a directed acyclic graph that represents the data flow of `exaFMM`'s dual tree traversal. Dekate *et al.* take a similar approach but using `ParalleX`[10] for the runtime system and a classical Barnes-Hut treecode [3] for the fast $N$-body solver. Agullo *et al.* use `StarPU`[11] along with the black-box FMM [13] on a CPU+GPU environment. Pericás *et al.* use `OmpSs` [12] with `exaFMM`. These are all preliminary results, and large gains from using runtime systems

---

[4]http://code.google.com/p/back40computing
[5]http://bima.astro.umd.edu/nemo
[6]http://www.mpa-garching.mpg.de/gadget
[7]http://bima.astro.umd.edu/nemo/man_html/gyrfalcON.1.html
[8]https://bitbucket.org/rioyokota/exafmm-dev
[9]http://icl.cs.utk.edu/quark/index.html
[10]http://stellar.cct.lsu.edu/tag/parallex
[11]http://runtime.bordeaux.inria.fr/StarPU
[12]http://pm.bsc.es/ompss

have yet to be reported.

In summary, the present work aims to provide the best implementation of the best algorithm for fast $N$-body solvers on many-core and heterogeneous architectures based on the thorough investigation of existing algorithms and implementations mentioned above. Our results indicate that many orders of magnitude in acceleration can be achieved by integrating many of these techniques that have not been previously combined. Upon integrating these various techniques, it was necessary to first reverse engineer many existing codes and to systematically investigate the individual contributions of each algorithmic improvement and each optimization trick that was not mentioned in the accompanying publication. Then, the value of these individual contributions had to be reevaluated based on predictions of how they will preform on many-core co-processors and GPUs. Finally, careful design decisions had to be made on which techniques to adopt, and what standardized constructs could be used to implement them in the simplest and most portable way in the integrated code. This ongoing work is being made available as an open source FMM library for Exascale – `exaFMM`, which is currently hosted on bitbucket[13]. Although the current work is influenced by many other predecessors, the work cited in this section are the ones that influenced the design decision of the final integrated code the most.

# 3 Present Method and Results

There are many variants of fast $N$-body methods, as described in the previous section. The argument in the previous section is strengthened by providing a more detailed description of the underlying techniques. The significance of the present work can be understood in more depth if these subtle differences in the algorithms and their implications on future architectures are explained clearly. Each subsection within this section revisits certain aspects of fast $N$-body methods in light of many-core and heterogenous architectures. In the end, we will compare 5 major implementations of treecodes and FMMs to show the relative performance of the resulting code.

---

[13]https://bitbucket.org/rioyokota/exafmm-dev

Table 2: Asymptotic complexity of the storage and arithmetic with respect to the order of expansion $p$ for the different expansions in fast $N$-body methods (3-D). There are two references for each expansion, the first is the original work describing the use of each expansion in the context of FMM, while the second has the simplest mathematical description of the translation operations.

| Type of expansion (+M2L acceleration) | Storage | Arithmetic |
|---|---|---|
| Cartesian Taylor [2, 30] | $\mathcal{O}(p^3)$ | $\mathcal{O}(p^6)$ |
| Cartesian Chebychev [11, 13] | $\mathcal{O}(p^3)$ | $\mathcal{O}(p^6)$ |
| Spherial harmonics [18, 7] | $\mathcal{O}(p^2)$ | $\mathcal{O}(p^4)$ |
| Spherial harmonics+rotation [33, 7] | $\mathcal{O}(p^2)$ | $\mathcal{O}(p^3)$ |
| Spherial harmonics+FFT [12, 22] | $\mathcal{O}(p^2)$ | $\mathcal{O}(p^2 \log^2 p)$ |
| Planewave [17, 22] | $\mathcal{O}(p^2)$ | $\mathcal{O}(p^3)$ |
| Equivalent charges [1, 25] | $\mathcal{O}(p^2)$ | $\mathcal{O}(p^4)$ |
| Equivalent charges+FFT [36, 23] | $\mathcal{O}(p^3)$ | $\mathcal{O}(p^3 \log p)$ |

## 3.1  Choice of Series Expansions

The accuracy of fast $N$-body algorithms is controllable by adjusting the order of multipole/local expansions. There are many series expansions that can be used for approximating Green's functions, *e.g.* Cartesian, spherical harmonics, planewaves, and equivalent charges. These series expansions have different characteristics in terms of Byte/flop, parallelism, and asymptotic complexity. A majority of the existing techniques to accelerate FMMs are based on old programming paradigms from back when arithmetic operations were the bottleneck. For example, pre-computation of translation matrices will reduce the arithmetic workload of *computing* the matrix entries multiple times, but requires extra bandwidth for *loading* the matrix every time. This goes against the current trend in hardware, where Byte/flop is decreasing at every level of the memory hierarchy. Furthermore, it may even be necessary to reconsider the choice of series expansions based on how regular the data access patterns are, and how parallel the operations are.

The asymptotic complexity of the storage and arithmetic operation for the M2L kernel for each expansion is shown in Table 2. Cartesian Taylor expansions are most widely used in treecodes, while other types of expansions are only used in FMMs. The common (but misleading) claim that treecodes are faster than FMMs for low accuracy, is actually a consequence of the choice of series expansions. Dehnen [9] used Cartesian expansions for FMM, and showed that it was an order of magnitude faster than the FMM using spherical harmonics or planewaves by Cheng *et al.* [7] for low accuracy, but was slower for higher accuracy. This result clearly demonstrates that it is not the use of treecodes that makes the difference, but rather the Cartesian Taylor expansion that increases the relative performance for lower accuracy. The FMM is always faster than treecodes for *any* accuracy if the only difference was cell-cell interactions vs. cell-particle interactions, and all other conditions (choice of expansion, particles per leaf cell, opening angle $\theta$) remain the same [38].

Cartesian expansions have higher arithmetic complexity but smaller asymptotic constants, whereas methods with lower arithmetic complexity tend to have larger asymptotic constants. For example, White and Head-Gordon [33] compare the spherical harmonics expansion with rotation $\mathcal{O}(p^3)$ and without rotation $\mathcal{O}(p^4)$, and observe only a 2 fold speed up even for $p = 21$. When comparing the break-even point between different series expansions, it is necessary to consider the low order terms as well. For example a $\mathcal{O}(p^6)$ method actually has a complexity of $ap^6 + bp^5 + cp^4 + dp^3 + ep^2 + fp + g$. If the coefficients of the lower order terms are large, a $\mathcal{O}(p^6)$ method may exhibit lower order asymptotic behavior for practical accuracy.

In order to demonstrate this behavior, we plot the calculation time of a single M2L kernel for a pair of cells against the order of expansion $p$ for the $\mathcal{O}(p^3)$ rotation based spherical harmonics expansion and $\mathcal{O}(p^6)$ Cartesian Taylor expansion in Figure 1. The time is shown in microseconds, and the runs were performed on a single socket of a Xeon X5650 (Westmere-EP, 2.67 GHz, 6 physical cores). The timings are measured using an actual FMM calculation with $N = 1,000,000$ particles, while timing the M2L kernels and counting the number of M2L kernel calls. The timing per M2L kernel is obtained by dividing the total M2L time by the number of M2L kernel calls. Even though the leading order term of the complexity of these methods are $\mathcal{O}(p^3)$ and $\mathcal{O}(p^6)$, they are closer to $\mathcal{O}(p^{2.5})$ and $\mathcal{O}(p^{4.5})$ in the practical accuracy range. Furthermore, the present implementation of the rotation based spherical harmonics expansion seems to crossover with the Cartesian Taylor expansion at around $p \approx 12$. Our Cartesian Taylor kernels are highly optimized using template meta-programming with specialization for low $p$. On the other hand, our rotation based spherical harmonics kernels are not optimized to the same extent, and simply use precalculated Wigner rotation matrices and translation matrices. Both versions use OpenMP but no SIMD intrinsics.

It would be useful to the FMM community to complete Figure 1 with the other entires in Table 2, and for a broader range of $p$. However, such results are not only hardware dependent, but also
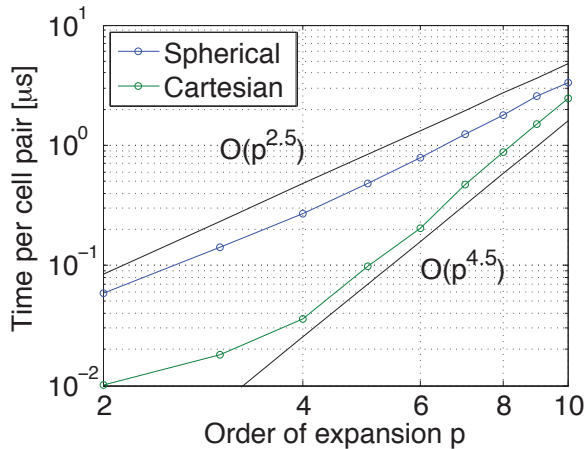
Figure 1: Calculation time of M2L translation between a single pair of cells for different orders of expansion $p$. The time is shown in microseconds. This data is calculated by performing a full FMM calculation and timing the M2L kernel while counting the number of cell pairs that call the M2L kernel. Calculations are performed on a single socket of a Xeon X5650 (2.67GHz, 6 physical cores).

highly implementation dependent. The tests would have to be run on a wide range of architectures, while tuning each kernel for each architecture. Furthermore, low level optimization of kernels of this complexity require a significant effort, and the absence of highly tuned open source implementations of these kernels makes it difficult to perform an exhaustive comparison between different expansions. Therefore, completing Figure 1 with the remaining series expansions is an ongoing investigation, which we hope to report in a future publication.

In the current work, we will use Cartesian Taylor expansions and focus on low accuracy FMMs. High accuracy FMMs with larger order of expansions will yield more fine grained parallelism, and will naturally offer better performance on many-core CPUs and GPUs. The real challenge is to optimize low accuracy FMMs on these architectures, and the reward will be substantial if this is possible. Recent efforts to view the FMM as a general elliptic PDE solver [23] have opened the possibility to use it as a preconditioner for even a broader range of applications. If the FMM is to be used as a preconditioner the required accuracy would not be high, which is the reason why we are interested in low accuracy FMM on future architectures. In terms of relative performance compared to multigrid preconditioners, the treecode by Bedorf *et al.* [4] can solve 8 levels in roughly 50 milliseconds on 1 GPU [14], while the multigrid code by Goddeke *et al.* also takes about 50 milliseconds for 8 levels on 1 GPU [16] to achieve the same accuracy.

## 3.2 Treecodes and FMMs

The two variants of hierarchical $N$-body methods – treecodes and FMMs – have followed very different paths of evolution over the past two decades. The primary reason for this divergence is the difference in the communities that adopted them. Treecodes have been used predominantly in the astrophysics community, where the distribution of stars are highly non-uniform, variation of local time scales is large, and the required accuracy is relatively low. On the other hand, FMMs received more attention from the applied mathematics community, and more emphasis was placed on asymptotically faster techniques with respect to the order of expansion $p$, extension to other

---

[14]The latest code from https://github.com/treecode/Bonsai is much faster than the one reported in their paper

kernels besides Laplace, and other generalizations to make them applicable to a wider range of problems. The variety of series expansions mentioned in the previous subsection is one example where FMMs are more advanced. Conversely, an area where treecodes are more sophisticated is the handling of adaptive tree structures for highly non-uniform distributions, which was an absolute requirement for their primary application. There are many interesting design decisions that arise when merging these orthogonal efforts into a unified framework.

The key differences between typical treecodes and FMMs are summarized in Table 3. The first item is the difference in how the particles are grouped into a tree structure. Treecodes commonly use rectangular grouping which squeeze the bounds of each cell[15] to exactly fit the residing particles [10], while FMMs always use cubic cells. When combined with the second item in the table for a sophisticated definition of "well-separated" cells, the use of tight rectangular cells results in an efficient error bounding mechanism. On the other hand, the use of cubic cells assumes a worst case scenario where the farthest particle from the cell center is at the corner of the cell. This may be a good approximation at coarser levels of a uniform tree, but is an extremely crude assumption at the leaf level of an adaptive tree. This assumption, along with the rigid definition of well-separated cells to be simply the non-neighboring cubes, results in an inefficient mechanism to achieve a given accuracy. This effect is much more prominent for low accuracy calculations, where the number of particles per leaf cell becomes small.

The multipole acceptance criterion (MAC) for treecodes has many variants as described by Salmon and Warren [27]. The concept of MACs have also been extended to FMM by the same authors [32] where they provide error bounds for the MAC based FMM. The different types of MAC are shown in Figure 2. The Barnes-Hut MAC is the original one proposed by Barnes and Hut [3]. It sets the threshold for accepting the multipole expansion according to the ratio between the cell size and distance between the target particle and center of expansion. Cells that are not accepted are further divided into their child cells and reevaluated. Therefore, the accuracy of treecodes can be controlled by adjusting the opening angle $\theta$. Though, decreasing $\theta$ too much will ultimately turn the treecode into a $\mathcal{O}(N^2)$ direct summation method (with no error by definition). The $B_{max}$ MAC gives a tighter error bound [27] and the same concept can be extended to FMMs [32].

Using the notation in Figure 2, the leading error term in the series expansion for the M2L kernel is of the order $\mathcal{O}(\left(\frac{B_i+B_j}{R}\right)^p)$. Therefore, the FMM has a simple error bound of $\mathcal{O}(\theta^p)$ [9].

---

[15]In our descriptions, the terminology "nodes" and "cells" essentially mean the same thing, where "node" refers to the elements in the tree data structure (from a computer science perspective), while "cell" refers to the corresponding geometric object in physical space (from a scientific computing perspective).

Table 3: Key differences between typical treecodes and FMMs.

|  | Treecode | FMM |
|---|---|---|
| Octree node shape | rectangle | cube |
| Definition of "well-separated" | multipole acceptance criterion | non-neighboring cubes |
| Center of expansion | center of mass | geometric center of cell |
| Accuracy control | opening angle $\theta$ | order of expansion $p$ |
| Data structure of tree | octree | Morton key |
| Finding interacting nodes | tree traversal | parent's neighbor's children that are non-neighbors |
| Domain decomposition | orthogonal recursive bisection | Morton key partitioning |

**Barnes-Hut MAC**
center of expansion
$R$
$L_j$
$\dfrac{L_j}{R} < \theta$
target particle
source cell

**FMM MAC**
$L_i$
$R$
$L_j$
$\dfrac{L_i + L_j}{R} < \theta$
target cell
source cell

**Bmax MAC**
$B_{max}$
$R$
$\dfrac{B_{max}}{R} < \theta$
target particle
source cell

**FMM Bmax MAC**
$B_j$
$R$
$B_i$
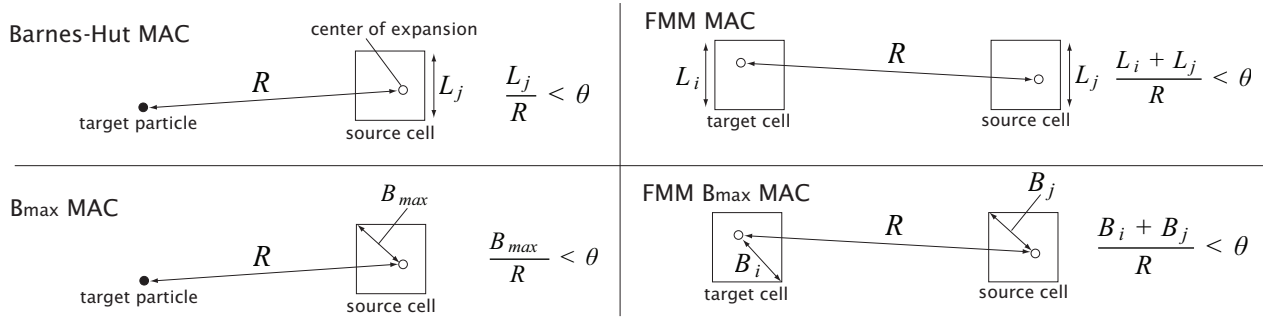$\dfrac{B_i + B_j}{R} < \theta$
target cell
source cell

Figure 2: Illustration of the different types of multipole acceptance criteria (MAC). Barnes-Hut MAC is the original one proposed by Barnes and Hut [3], $B_{max}$ MAC is a tighter MAC proposed by Salmon and Warren [27], and FMM MAC is the extension of the $B_{max}$ MAC to FMM [32].

This error bound is easily extendable to treecodes by assuming $B_i = 0$, which is equivalent to assuming that the target cell is just one particle and has no size. As the third item in Table 3 indicates, treecodes increase the accuracy by decreasing $\theta$, while keeping $p$ constant ($p = 3$ in most cases). Conversely, FMMs increase the accuracy by increasing $p$, while keeping $\theta$ constant (most commonly $\theta = 1$ according to the definition of "FMM MAC" in Figure 2). At every level of the tree, the M2L interaction list forms a constant-thickness layer on the surface of a sphere with radius $\frac{B_i+B_j}{\theta} < R < 2\frac{B_i+B_j}{\theta}$, which means the lists size grows as $\mathcal{O}(\theta^{-2})$ (surface of sphere with normalized radius $1/\theta$). If one were to use a $\mathcal{O}(p^3)$ method for the M2L kernel, the complexity with respect to $p$ and $\theta$ will be $\mathcal{O}(\theta^{-2}p^3)$. There should exist an optimal balance between $p$ and $\theta$ that minimizes the $\mathcal{O}(\theta^{-2}p^3)$ work, while keeping the $\mathcal{O}(\theta^p)$ error constant.

The various combinations of $p$ and $\theta$ for a given accuracy are shown in Table 4. This table was generated by first setting a target accuracy (relative $L^2$ norm error of the force (not potential) for the Laplace kernel, shown as $Err$) and setting $p$. Then, we searched through the parameter space of $\theta$ to find the largest value that achieves the target accuracy. These, values of $\theta$ are shown in the table along with the amount of time it takes to perform the dual tree traversal for each case. The

Table 4: Balance between order of expansion $p$ and opening angle $\theta$ for Cartesian Taylor expansions to achieve a specific error tolerance. $Err$ is the relative $L^2$ norm error of the force (not potential) for the Laplace kernel, where the result from a direct summation is used as the reference value. The time is measured for the dual tree traversal (M2L kernels+P2P kernels+list construction) for $N = 10^5$ particles for one time step. The maximum number of particles per leaf cell is set to $N_{crit} = 30$ for all cases.

| | $p = 3$ | $p = 4$ | $p = 5$ | $p = 6$ |
|---|---|---|---|---|
| $Err = 10^{-2}$ | $\theta = 1.00$ | $\theta = 1.18$ | $\theta = 1.23$ | $\theta = 1.24$ |
| | $time = 0.016s$ | $time = 0.012s$ | $time = 0.015s$ | $time = 0.026s$ |
| $Err = 10^{-3}$ | $\theta = 0.67$ | $\theta = 0.78$ | $\theta = 0.91$ | $\theta = 0.94$ |
| | $time = 0.036s$ | $time = 0.027s$ | $time = 0.024s$ | $time = 0.038s$ |
| $Err = 10^{-4}$ | $\theta = 0.30$ | $\theta = 0.49$ | $\theta = 0.62$ | $\theta = 0.70$ |
| | $time = 0.22s$ | $time = 0.085s$ | $time = 0.071s$ | $time = 0.073s$ |
| $Err = 10^{-5}$ | $\theta = 0.12$ | $\theta = 0.20$ | $\theta = 0.36$ | $\theta = 0.45$ |
| | $time = 1.38s$ | $time = 0.59s$ | $time = 0.21s$ | $time = 0.21s$ |

tree construction, P2M, M2M, L2L, L2P kernels are not included in these timings. The number of particles is $N = 10^5$, and the maximum number of particles per leaf cell is set to $N_{crit} = 30$ for all cases. Here, the definition of $\theta$ is a more advanced version than the ones shown in Figure 2, and uses both $B_{max}$ and $R_{max}$ [9] but without error optimization of $R_{max}$. The optimal $p$ shifts to larger values when more accuracy is required. According to the current definition, $Err = 10^{-5}$ means that 5 significant digits match with the results of a direct summation, which is sufficient for many applications in structural mechanics, fluid dynamics, and molecular dynamics. By comparing with Figure 1, one can conclude that Cartesian Taylor expansions are much faster than spherical harmonics with rotation for this range of $p$.

## 3.3  Dual tree traversal

As summarized in Table 3, the underlying data structure in treecodes and FMMs is quite different. Treecodes usually construct a tree data structure, where each node is expressed as a structure which contains a pointer to it's children. By looping over it's children in a recursive function call (or by using a stack or queue), all nodes can be visited once in a topdown manner, *i.e.* a tree traversal. The multipoles in the well-separated cells are easily found by applying the MAC during the tree traversal. This procedure is shown in Algorithm 1.

On the other hand, most FMM codes do not construct links between the tree nodes, nor do they traverse a tree. Instead, they bin the particles based on Morton/Hilbert keys, and loop over all the target cells in the tree (level wise), and explicitly construct a list of source cells that are well-separated. This procedure does not require a linked tree structure. The definition of well-separated cells is usually the "target cell's parent's neighbor's children that are not direct neighbors of the target cell". Parents and children are found by multiplying and dividing the Morton key by 8. Neighbors are found by deinterleaving the bits in the key into a three-dimensional index, incrementing/decrementing in a certain direction, and interleaving them back to a single key. In adaptive trees these keys are mapped to a unique index by hashing (or simply arrays with zeros for the empty boxes), and this unique index is used to load/store the corresponding multipole/local expansion coefficients. This procedure is shown in Algorithm 2.

There are more advanced techniques for finding the interaction lists in FMMs. Cheng *et al.* extended the originally non-adaptive interaction list with only 2 categories "well-separated" and "neighbor", to lists for adaptive trees using 4 categories [7]. Gumerov and Duraiswami proposed a way to reduce the number of interactions for the "well-separated" cells of the original non-adaptive interaction list from 189 to 119 without sacrificing the accuracy [19]. However, both of these techniques have the same structure of using the "parent's neighbors" to define the well-separated cells, and finding neighbors in this way have several disadvantages. First, special treatment of

Table 5: Different types of tree traversals and interaction list generation.

|  | Treecode | FMM | FMM (adaptive) | FMM (DTT) |
|---|---|---|---|---|
| Data structure | octree | array | array | octree |
| Complexity | $\mathcal{O}(N \log N)$ | $\mathcal{O}(N)$ | $\mathcal{O}(N)$ | $\mathcal{O}(N)$ |
| Interaction list | implicit | explicit | explicit | implicit |
| Cell shape | rectangular | cubic | cubic | rectangular |
| $\theta$ adjustable | yes | no | no | yes |
| Parallelism | for loop | for loop | for loop | task-based |

cells on the boundaries is required since they will not have a full list of 26 neighbors. Second, the definition of "well-separatedness" cannot be flexibly controlled, which makes a large difference as we have demonstrated in Table 4. Third, only cubic cells can be used because the cell's dimensions are used as units to measure well-separatedness, whereas rectangular cells have been preferred in treecodes for highly non-uniform distributions. This third disadvantage also has implications for load-balancing adaptive global trees, which is an important issue that is beyond the scope of the present work.

It is possible to overcome the aforementioned disadvantages by using a dual tree traversal [32, 9], where the target tree and source tree are traversed simultaneously to efficiently create the interaction list. A pseudocode for the dual tree traversal is shown in Algorithms 3 and 4. Table 5 summarizes the characteristics of four different ways to systematically evaluate the far field and near field contributions in fast $N$-body methods. The dual tree traversal inherits the flexibility and adaptiveness of the tree traversal used in treecodes, while achieving $\mathcal{O}(N)$ complexity. The "well-separatedness" can be defined flexibly and the use of rectangular cells are allowed. The target tree and source tree can be completely independent tree structures, since the distance between cells is identified from the coordinates of the center of expansion at each cell. The use of Morton/Hilbert keys is not a requirement, but can be used to accelerate the tree construction process. One drawback of the dual tree traversal could be the loss of obvious parallelism at the outermost loop, as can be seen by comparing Algorithms 1, 2, and 3. Treecodes have an outer loop over all target leafs, while conventional FMMs have an outer loop over all target cells. However, the dual tree traversal does not have any "for loop" structures that could be readily parallelized by, for example a `#pragma omp parallel for` directive. This may or may not be an issue since standards such as OpenMP 3.0 and Intel Thread Building Blocks (TBB) now provide task-based threading models, which do not require "for loops" for parallelization. In this model, tasks are spawned by the user and dispatched to idle threads dynamically during runtime. Extracting thread level parallelism from the dual tree traversal on NUMA architectures is an interesting challenge for data-driven execution models such as `QUARK`, `StarPU`, `OmpSs`, and `ParalleX`.

The lack of "for loop" type parallelism in dual tree traversals makes it difficult to simply assign a certain part of the work to a thread. Therefore, on GPUs and many-core co-processors, it is crucial to use task-based parallelization tools available in CUDA 5.0, OpenMP 3.0 and Intel TBB. However, the lack of "for loop" type parallelism in dual tree traversals itself is not a limitation, since these task-based parallelization tools now exist on the latest compilers. It is important to stay updated on the advancement in compiler technology and to continuously consider all variants of FMMs that can take advantage of them.

## 3.4 Code optimization

A highly optimized code must take advantage of SIMD instructions, to extract the full potential of modern microprocessors. For the current purpose of optimizing FMM for low accuracy, single-precision is sufficient. For this case SSE and AVX can offer a maximum of $\times 4$ and $\times 8$ acceleration, respectively. Furthermore, $N$-body kernels involve a reciprocal square root operation, which can be accelerated by an intrinsic function `rsqrt_ps()` in SSE/AVX. Furthermore, by reordering the instructions to interleave the load operations with fused-multiply-add operations, one can achieve flop/s that is very close to the theoretical peak, given the low Byte/flop ratio of these $N$-body kernels.

We first demonstrate the performance increase in a Laplace P2P kernel when SSE and AVX intrinsics are used. The runs are on a Intel Xeon E5-2643 (Sandy Bridge, 3.3 GHz, 4 physical cores) with `g++-4.7 -O3 -mavx -fopenmp -ffast-math -funroll-loops`. The execution time

---

**Algorithm 1** EvaluateTreecode()

---

**for** all target leafs **do**
   push source root cell to stack
   **while** stack is not empty **do**
     pop stack
     **for** all children of cell **do**
       **if** child cell is a leaf **then**
         call P2P kernel
       **else**
         **if** child cell satisfies MAC **then**
           call M2P kernel
         **else**
           push child cell to stack
         **end if**
       **end if**
     **end for**
   **end while**
**end for**

---

---

**Algorithm 2** EvaluateFMM()

---

**for** all target cells **do**
   find parent cell by dividing Morton key
   **for** all children of parent cell's neighbor **do**
     call M2L kernel
   **end for**
**end for**
**for** all target leafs **do**
   **for** all neighbor's **do**
     call P2P kernel
   **end for**
**end for**

---

---

**Algorithm 3** EvaluateFMMUsingDualTreeTraversal()

---

push pair of root cells $(A,B)$ to stack
**while** stack is not empty **do**
   pop stack to get $(A,B)$
   **if** target cell is larger then source cell **then**
     **for** all children $a$ of target cell $A$ **do**
       Interact$(a,B)$
     **end for**
   **else**
     **for** all children $b$ of source cell $B$ **do**
       Interact$(A,b)$
     **end for**
   **end if**
**end while**

---

---

**Algorithm 4** Interact($A$,$B$)

---

   **if** $A$ and $B$ are both leafs **then**
     call P2P kernel
   **else**
     **if** $A$ and $B$ satisfy MAC **then**
       call M2L kernel
     **else**
       push pair ($A$,$B$) to stack
     **end if**
   **end if**

---

and Gflop/s are shown in Table 6. The calculation is a direct summation between $N = 10^5$ particles for both potential and force, which results in 20 floating-point operations per interaction. In all three cases, the `#pragma omp parallel for` directive is used in the outermost loop to utilize all cores. We obtain a speedup of $\times 3.88$ with SSE and $\times 6.59$ with AVX. These SSE and AVX kernels are available from `https://bitbucket.org/rioyokota/exafmm-dev` as P2P kernels in our FMM. In this code, predefined macros will detect the availability of AVX/SSE and turn them on if the compiler supports them. Furthermore, the loop counter is carried across AVX, SSE, and non-SIMD kernels so that the loop is first processed in increments of 8, then remainders of 8 are processed with SSE, and finally remainders of 4 are processed with non-SIMD kernels.

## 3.5   Comparison between different FMM codes.

In order to clarify the relative performance of the major open source treecodes and FMMs, we perform a benchmark of 5 different codes on the same computer. The benchmark is for a Laplace kernel with randomly distributed particles in a cube. The number of particles is changed from $N = 10^4$ to $10^6$, while setting the accuracy to 3 significant digits for the force (not potential). Note that this setting will yield a little more than 4 significant digits for the potential. For codes that do not calculate the force, we tried our best to match the corresponding accuracy for the potential. These benchmarks are performed on a single socket of a Xeon X5650 (Westmere-EP, 2.67GHz, 6 physical cores). `exaFMM` is the latest version of our code[16]. `KIFMM` [6] is the latest version directly obtained from the developers. `falcON` [9] is publicly available as part of the package `NEMO`[17]. `ufmmlap` [17] is publicly available as `FMM-Laplace-U` from `fastmultipole.org`[18]. `PEPC` [35] is the

---

[16]https://bitbucket.org/rioyokota/exafmm-dev
[17]http://bima.astro.umd.edu/nemo
[18]http://fastmultipole.org/Main/FMMSuite

---

Table 6: Execution time and Gflop/s of Laplace P2P kernel with SSE and AVX. The calculation is a direct summation between $N = 10^5$ particles for both potential and force, which results in 20 floating-point operations per interaction.

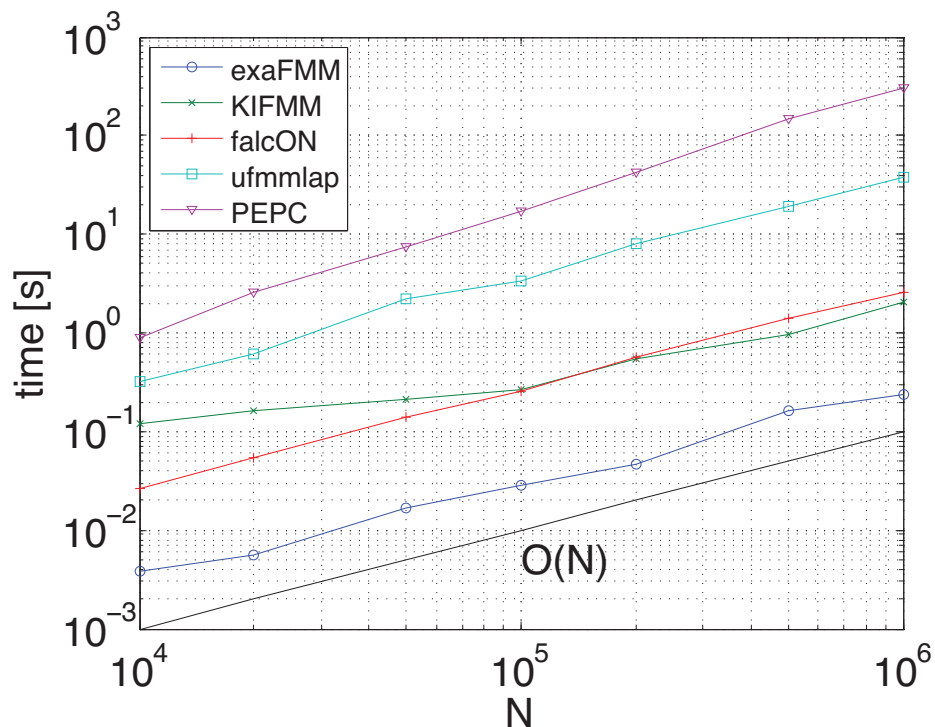|  | no SIMD | SSE | AVX |
|---|---|---|---|
| CPU time [s] | 1.75 | 0.486 | 0.274 |
| Gflop/s | 48.7 | 189 | 321 |
| Speedup | $\times 1$ | $\times 3.88$ | $\times 6.59$ |

Figure 3: Comparison between different FMM codes. The benchmark is for a Laplace kernel with randomly distributed particles in a cube. These benchmarks are performed on a single socket of a Xeon X5650 (Westmere-EP, 2.67GHz, 6 physical cores). Parameters in each code are adjusted to obtain 3 significant digits of accuracy in the force for all codes. Timings include the tree construction time.

latest publicly available version[19].

The characteristics of the 5 different codes are summarized in Table 7. PEPC is a treecode, while the other four are FMMs. PEPC inherits most of it's features from Salmon and Warren's treecode

---

[19]https://trac.version.fz-juelich.de/pepc

Table 7: Characteristics of the different codes in Figure 3. DTT : dual tree traversal, STT : single tree traversal, EC : equivalent charges, TBB : thread building blocks

|  | exaFMM | KIFMM | faclON | ufmmlap | PEPC |
|---|---|---|---|---|---|
| Language | C++ | C++ | C++ | F90 | Fortran 2003 |
| Tree data structure | octree | array | octree | array | octree |
| Interaction list | DTT | U,V,W,X [36] | DTT | U,D,N,S,E,W [17] | STT |
| Series expansion | Cartesian | EC + FFT | Cartesian | planewave | Cartesian |
| Accuracy control | $p$ & $\theta$ | $p$ | $\theta$ | $p$ | $\theta$ |
| MPI | yes | yes | no | no | yes |
| Threading model | TBB | OpenMP | N/A | N/A | pthreads |
| SIMD | AVX | SSE | N/A | N/A | N/A |

[31], and has the structure of a typical treecode. On the other hand, `ufmmlap` can be thought of as a typical FMM code, so the contrast in the tree data structure, interaction list, series expansion, and accuracy control are most drastic between these two. `KIFMM` is similar to `ufmmlap` in the sense that it also uses a typical FMM framework, except the series expansion is unique. The high performance of `KIFMM` mostly comes from the implementation (OpenMP+SSE with low level optimizations), and on a 12 (hyper-threaded) core machine this results in a big difference between the single core non-SIMD implementation of `ufmmlap`. Although `falcON` is also a single core non-SIMD implementation, it achieves the same speed as `KIFMM` just from purely algorithmic improvements. These algorithmic improvements include the dual tree traversal, mutual interaction, error aware local optimization of the opening angle $\theta$, and a multipole acceptance criterion based on $B_{max}$ and $R_{max}$ [9]. Finally, `exaFMM` combines the algorithmic improvements of `falcON` with a highly parallel implementation using MPI, Intel TBB, and AVX. Accuracy can be controlled by both $p$ and $\theta$, the dual tree traversal is parallelized efficiently with task-based parallelism that can either use OpenMP 3.0 or Intel TBB, and the kernels are optimized with AVX to yield 300 Gflop/s on a single CPU socket. With the combination of these techniques, `exaFMM` can calculate the Laplace potential+force for $N = 10^6$ particles to 3 digits of accuracy for the force in approximately 0.2 seconds on a single CPU socket, which exceeds the performance of most GPU implementations of FMM.

## 4  Summary

The present work attempts to integrate the independent efforts in the fast N-body community to create the fastest N-body library for many-core architectures of the future. Focus is placed on low accuracy optimizations, in response to the recent interest to use FMM as a preconditioner for sparse linear solvers. A direct comparison with other state-of-the-art fast $N$-body codes demonstrates that orders of magnitude increase in performance can be achieved by careful selection of the optimal algorithm and low-level optimization of the code.

The low Byte/flop of future architectures affects the choice of series expansions and translation methods, and our focus on low accuracy perturbs this decision even further. For example, the Cartesian Taylor expansion was shown to be much faster than spherical harmonics with rotation in the range of accuracy of interest. Furthermore, simultaneous control over the order of expansion $p$ and opening angle $\theta$ was also shown to yield a significant increase in performance over conventional treecodes and FMMs that only control either of them.

The dual tree traversal offers an interesting alternative to conventional interaction list construction, and modern compilers now provide the necessary tools to parallelize them efficiently. The fact that dual tree traversals allow rectangular cells, enables great flexibility in the partitioning/load-balancing on distributed memory architectures. There has been little work in this promising area, because dual tree traversal has not been used in the FMM community until now. Although the scope of the present article is focused on single node performance, we would like to point out that the current framework enables various interesting possibilities for partitioning/load-balancing FMMs.

The performance of the FMM code described in this article is made available to the general public[20]. The developer has made a significant effort to make the code readable by carefully designing the code structure to make it as simple as possible, while retaining the performance. One can easily see by comparing the source code with the other 4 codes in the benchmark study, that `exaFMM` is more than an order of magnitude smaller in terms of lines of code, while being orders of magnitude faster.

---

[20]https://bitbucket.org/rioyokota/exafmm-dev

# References

[1] C. R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM Journal on Scientific and Statistical Computing*, 13(4):923–947, 1992.

[2] J. Applequist. Traceless cartesian tensor forms for spherical harmonic functions: new theorems and applications to electrostatics of dielectric media. *Journal of Physics A: Mathematical and General*, 22:4303–4330, 1989.

[3] J. Barnes and P. Hut. O(NlogN) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[4] J. Bedorf, E. Gaburov, and S. Portegies Zwart. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *Journal of Computational Physics*, 231:2825–2839, 2012.

[5] J. Board and K. Schulten. The fast multipole algorithm. *Computing in Science and Engineering*, 2(1):76–79, 2000.

[6] A. Chandramowlishwaran, 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, 2010.

[7] H. Cheng, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155(2):468–498, 1999.

[8] O. Coulaud, P. Fortin, and J. Roman. High performance BLAS formulation of the multipole-to-local operator in the fast multipole method. *Journal of Computational Physics*, 227:1836–1862, 2008.

[9] W. Dehnen. A hierarchical O(N) force calculation algorithm. *Journal of Computational Physics*, 179(1):27–42, 2002.

[10] Z.-H. Duan and R. Krasny. An adaptive treecode for computing nonbonded potential energy in classical molecular systems. *Journal of Computational Chemistry*, 22(2):184–195, 2001.

[11] A. Dutt, M. Gu, and V. Rokhlin. Fast algorithms for polynomial interpolation, integration, and differntiation. *SIAM Journal on Numerical Analysis*, 33(5):1689–1711, 1996.

[12] W. D. Elliott and J. A. Board. Fast fourier transform accelerated fast multipole algorithm. *SIAM Journal on Scientific Computing*, 17(2):398–415, 1996.

[13] W. Fong and E. Darve. The black-box fast multipole method. *Journal of Computational Physics*, 228:8712–8725, 2009.

[14] P. Fortin, E. Athanassoula, and J.-C. Lambert. Comparisons of different codes for galactic N-body simulations. *Astronomy and Astrophysics*, 531:A120, 2011.

[15] H. Gahvari and W. Gropp. An introductory exascale feasibility study for ffts and multigrid. In *IEEE International Symposium on Parallel and Distributed Processing*, Atlanta, GA, April 19-23 2010.

[16] D. Goddeke and R. Strzodka. Cyclic reduction tridiagonal solvers on gpus applied to mixed precision multigrid. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):22–32, 2011.

[17] L. Greengard and V. Rokhlin. A new version of the fast multipole method for the laplace equation in three dimensions. *Acta Numerica*, 6:229–269, 1997.

[18] L. Greengard and Rokhlin V. On the efficient implementation of the fast multipole algorithm. Research Report RR-602, Yale University, 1988.

[19] N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227:8290–8313, 2008.

[20] Q. Hu, N. A. Gumerov, and R. Duraiswami. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[21] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kale, and T. R. Quinn. Scaling hierarchical N-body simulations on GPU clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, 2010.

[22] J. Kurzak and B. M. Pettitt. Fast multipole methods for particle dynamics. *Molecular Simulation*, 32(10):775–790, 2006.

[23] H. Langston, L. Greengard, and D. Zorin. A free-space adaptive FMM-based PDE solver in three dimensions. *Communications in Applied Mathematics and Computational Science*, 6(1):79–122, 2011.

[24] H. Ltaief and R. Yokota. Data-driven execution of fast multipole methods. *arXiv:1203.0889v1*, 2012.

[25] J. Makino. Yet another fast multipole method without multipoles- pseudoparticle multipole method. *Journal of Computational Physics*, 151(2):910–920, 1999.

[26] A. Rahimian, I. Lashuk, K. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, 2010.

[27] J. K. Salmon and M. S. Warren. Skeletons from the treecode closet. *Journal of Computational Physics*, 111:136–155, 1994.

[28] V. Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, 2005.

[29] T. Takahashi, C. Cecka, W. Fong, and E. Darve. Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units. *International Journal for Numerical Methods in Engineering*, 89:105–133, 2012.

[30] P. B. Visscher and D. M. Apalkov. Simple recursive implementation of fast multipole method. *Journal of Magnetism and Magnetic Materials*, 322:275–281, 2010.

[31] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21, 1993.

[32] M. S. Warren and J. K. Salmon. A portable parallel particle program. *Computer Physics Communications*, 87:266–290, 1995.

[33] C. A. White and M. Head-Gordon. Rotating around the quartic angular momentum barrier in fast multipole method calculations. *Journal of Chemical Physics*, 105(12):5061–5067, 1996.

[34] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications in Applied Mathematics and Computational Science*, 52(4):65–76, 2009.

[35] M. Winkel, R. Speck, H. Hubner, L. Arnold, R. Krause, and P. Gibbon. A massively parallel, multi-disciplinary barnes–hut tree code for extreme-scale N-body simulations. *Computer Physics Communications*, 183:880–889, 2012.

[36] L. Ying, G. Biros, and D. Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2):591–626, 2004.

[37] R. Yokota and L. A. Barba. *Treecode and Fast Multipole Method for N-body Simulation with CUDA*, chapter 9. GPU Computing Gems. Morgan Kaufmann, emerald edition edition, 2011.

[38] R. Yokota and L. A. Barba. Hierarchical N-body simulations with auto-tuning for heterogeneous systems. *Computing in Science and Engineering*, 14(3):30–39, 2012.

[39] 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*, 180:2066–2078, 2009.

[40] R. Yokota, T. Narumi, K. Yasuoka, and L. A. Barba. Petascale turbulence simulation using a highly parallel fast multipole method on gpus. *arXiv:1106.5273v2*, 2012.