

PyExaFMM: Designing a high-performance point fast multipole solver in Python with Numba

S. Kailasa

Department of Mathematics, University College London

T. Betcke

Department of Mathematics, University College London

T. Wang

Department of Mechanical and Aerospace Engineering, The George Washington University

L. A. Barba

Department of Mechanical and Aerospace Engineering, The George Washington University

Abstract—PyExaFMM is a Python implementation of the kernel-independent point fast multipole method. Building on the success of the ExaFMM project, we aimed to answer the question of whether it was possible to develop a highly-performant scientific code without resorting to a lower level language. The computational experience of domain specialists is often restricted to interpreted languages, such as Python or Matlab. Therefore the ability to develop high-performance software in a high-level language is appealing, reducing the barrier to entry for potential contributors, as well as making deployment to new architectures or platforms easier. The FMM is a good case study for understanding the maturity of Python for developing high-performance software, due its reliance on a complex hierarchical tree data structure. In this article we explore the software engineering and mathematical techniques used to extract performance for PyExaFMM, and we report that we achieve runtimes within $\mathcal{O}(10)$ of the state of the art C++ implementation, with comparable accuracy, for three dimensional electrostatic problems in single precision.

■ **THE FAST MULTIPOLE METHOD** [FMM], [1], approximates the solution of the so called N body problem, in which one seeks to calculate

pairwise interactions between N objects. This problem arises in numerous contexts in science and engineering, for example in the calculation of electrostatic or gravitational potentials due to a set of charged or massive source objects. This is an important calculation, as with an estimate of potential one is able to describe the corresponding electric, or gravitational field due to these objects, and hence the forces they exert. In this article we use the electrostatic potential generated by point charges in three dimensions as our model to explore the FMM. Here, the potential, $\phi(x_j)$, for a given point at position x_j , or ‘target’, due to N points at positions x_i , or ‘sources’, where $i \in [1, \dots, N]$, each with a charge q_i , can be written as,

$$\phi(x_j) = \sum_{i=1}^N K(x_i, x_j) q_i, \quad (1)$$

here $K(\cdot, \cdot)$ is called the Green’s function, or kernel function. For electrostatic problems in three dimensions it is,

$$K(x, y) = \frac{1}{4\epsilon_0\pi|x - y|}, \quad (2)$$

where ϵ_0 is the permittivity of free space, and is referred to as the Laplace kernel. Without loss of generality, we consider the set of sources and set of targets to correspond to the same set of points.

Intuitively, (2) can be understood as encoding a ‘decay’ relationship between two given points. The closer they are the higher their proportionate effect on each other’s potential. Attempting to evaluate the sum in (1) directly for N points, results in algorithm of $\mathcal{O}(N^2)$ runtime complexity, as the sum will have to be repeated N times. However the FMM is able to approximate (1) with just $\mathcal{O}(N)$ runtime complexity, with prescribed error bounds. This makes the simulation of granular physically realistic systems tractable.

The FMM

The key idea behind the FMM is to encode the potential in the far field due to a cluster of points with a representative analytic series expansion centered on the cluster, referred to as a *multipole expansion*. This expansion can be

truncated to tune for a desired accuracy, which allows one to approximate the sum in (1) with fewer calculations. Alternatively, we can center the expansion far away from the cluster, the encoded potentials are then referred to as a *local expansion*. Translations between multipole and local expansions can be done analytically, and are critical in the development of the $\mathcal{O}(N)$ algorithm. However, this approximation is only valid where the kernel function of a problem exhibits appropriate ‘decay’ characteristics, allowing us to assume that potential due to clusters of points far away from a region of interest can be approximated with a truncated representative expansion.

In the FMM, the problem domain is described by a cubic box enclosing all targets and sources, which is hierarchically partitioned into a structure known as a quadtree in two dimensions, and an octree in three dimensions. The partitioning is done recursively. Initially it is partitioned into four equal parts in two dimensions (eight in three dimensions), known as its children. Correspondingly the unrefined box is referred to as their parent. The children are each subsequently refined in a similar fashion. The final tree consisting of a series of hierarchically nested boxes.

The size of the box is described by its ‘level’, with level zero denoting the unrefined box, level one denoting the level of its children, and so forth. One can define the maximum level of refinement through a parameter dictating the maximum allowable number of points contained within the finest box covering a portion of the domain. These finest boxes that remain after refinement are known as leaves. Note that the set of all leaves cover the entire domain, without overlap. One can choose to refine *adaptively*, such that the leaves may be of different sizes, reflective of the underlying point distribution, or *non-adaptively*, such that the leaves are all of a uniform size.

The FMM then consists of two sequential traversals of this tree, where boxes are considered level by level. Initially, during the *upward pass*, multipole expansions for points contained in boxes at the leaf level are formed. This is referred to as the *point-to-multipole* [P2M] operation. In order to obtain the multipole expansions for their parents, the expansion centers of a given box’s children are shifted to the center of their shared

parent box, and their coefficients are summed. This is referred to as the *multipole-to-multipole* [M2M] operation. This is repeated bottom-up, level by level, until one is left with the multipole expansions of all boxes in the tree.

Subsequently, during the *downward pass*, the multipole expansions of non-adjacent boxes of a given box are defined as being in its far field. In order to evaluate their contribution towards the potential for points within a given box, one translates their multipole expansion to a local expansion centered at the given box. This is referred to as the *multipole-to-local* [M2L] operation. The local expansion is then transferred to the children of the given box by shifting the expansion center to the center of the child boxes, and updating the coefficients of the child boxes' local expansions. Proceeding top-down, considering boxes level by level, one is left with the local expansions for each leaf box. These are referred to as *local-to-local* [L2L] operations.

Crucially the extent of the far field not already encapsulated in the local expansion shrinks as we descend down the levels of the tree. This is because the far field of a box's parent is already captured in the child's local expansion. These local expansions compress all of the contribution from the far field towards the potential for targets within a given leaf. Therefore the far field's effect on the potential of targets within a leaf box is entirely described by the local expansion.

The evaluation of the local expansion at the target points in a given leaf box is known as the *local-to-point* [L2P] operation. Finally, the near fields of a given leaf box are evaluated directly using (1), this is referred to as the *point-to-point* [P2P] operation. As there are a fixed number of points per leaf node, the complexity of this direct evaluation is bounded. The operations in this scheme are illustrated for a two dimensional problem, with a uniform quadtree, in figure (1).

The $\mathcal{O}(N)$ algorithmic runtime complexity can be roughly seen to be the result of the ability to exactly translate between the multipole and local expansion representations for all boxes deemed to be in the far field of a given box, as well as the recursive procedure of the FMM. As a result of this, each box must only consider its interaction with a constant number of other boxes. As there are $\mathcal{O}(N)$ boxes in a given tree, the

entire algorithm can be seen to be bounded by a runtime complexity of $\mathcal{O}(N)$.

Kernel Independent FMM

The coefficients of a given multipole expansion are kernel-dependent in the sense that they will depend on the form of (2), software implementations have to be rewritten for each specific physical model. Fast multipole methods for generic kernel functions have been developed [2], however there are fewer methods that rely only on numerical kernel evaluations, rather than analytic series expansions, examples include [3], [4]. The latter approaches are often referred to as kernel-independent fast multipole methods [KIFMM].

PyExaFMM utilizes the approach first presented by Ying et. al [4]. This KIFMM represents the multipole expansions due to a cluster of charges as set of equivalent charge densities supported on a surface enclosing the cluster. The fields generated by the charges are matched with an equivalent field generated by these equivalent densities via a least-squares fitting of the potential they generate at another surface in the far field. This method generalizes the FMM to non-oscillatory second-order elliptic partial differential equations with constant coefficients, for example the Laplace or Stokes equations. This includes many common problems, such as our model electrostatic problem.

The Software Context

At present, numerous high-quality open source implementations exist to solve FMM problems, both using analytic expansions for various kernels [5], as well as using the KIFMM of Fong et. al [6], [7], and Ying et. al [8], [9]. Some of these implementations are able to scale on supercomputing clusters, and solve problems involving millions of degrees of freedom [9]. Others are designed to be run on a single workstation [6], [8]. These softwares are developed in compiled languages, such as Fortran or C++, often making use of numerous non-standard libraries and language features. Compiled languages are often accompanied by complex build processes, making deployment to different platforms or architectures non-trivial. Many domain specialists who may want to use the FMM to aid their research usually have limited computational skill sets, often re-

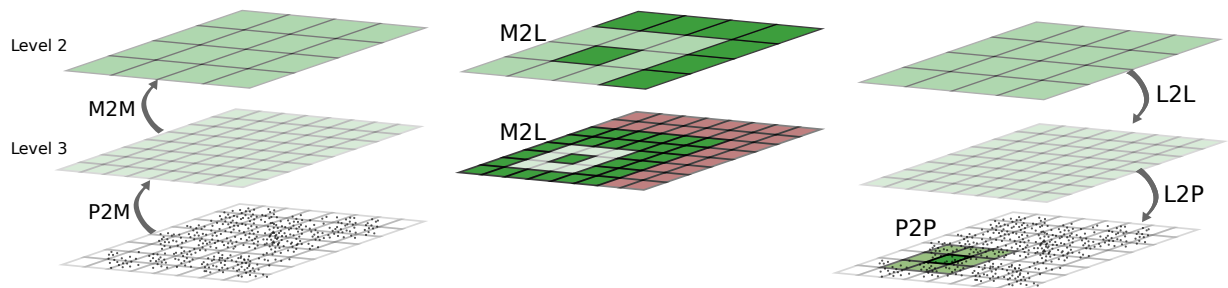


Figure 1. The FMM illustrated for a two dimensional problem with a uniform quadtree. The left column illustrates the first two steps of the upward pass, beginning with the P2M operation to find the multipole expansions at the leaf level, followed by an M2M operation. The central and right columns illustrate the downward pass. In the central column, the far-field of a given box is highlighted, we observe that at level three the far field highlighted in red is already encapsulated in the local expansion of the given box due to the L2L translation from its parent. In the right column we see how the algorithm concludes with a given box evaluating its near field of adjacent boxes, directly via the P2P operation.

stricted to subsets of Python or Matlab. Therefore it is not feasible for many users of to contribute to, or extend, many of these existing libraries without a significant investment in acquiring new skills. This problem is not unique to FMM implementations, but is general across high-performance computing [HPC] software.

Python for High Performance

In recent years, HPC libraries that aid in bypassing the limitations of the Python interpreter have emerged. The most pertinent example is Numba, a ‘just in time’ [JIT] compiler for CPython, a popular C implementation of the Python interpreter. JIT refers to the fact that the compiler waits until a Numba enhanced object, either a function or a class, is instantiated at runtime, before running optimizations to convert the Python source code into efficient machine code. Numba achieves performance by focussing solely on optimizing operations on ndarrays, the multi-dimensional array objects provided by Numpy . These objects have become the basis for numerical libraries written in Python as Numpy binds to widely used HPC libraries for numerical computing written in C or Fortran, such as LAPACK and BLAS.

Numba extends Numpy by removing inefficient layers of indirection for indexing into Numpy arrays, and instead translating array index operations into direct load and store operations from pointers [10]. Furthermore, it of-

fers simple multithreading support through a ‘prange’ statement that allows you to loop over processes, mirroring OpenMP’s parallel for loops. Numba even supports automatic single-instruction-multiple-data [SIMD] vectorization. Adapting to the instruction set architecture of your CPU, whether it supports Advanced Vector Extensions [AVX], AVX-512, or Streaming SIMD Extensions [SSE]. Therefore with Numba it’s easy to heavily optimize methods that loop over arrays. Finally, Numba allows one to decorate functions to be run on GPUs, allowing the development of heterogenous applications with a single Python source.

Importantly, Numba is a ‘drop-in’ library. It’s simply installed via package manager, and objects are marked to be optimized by a single decorator. If Numba is unable to perform an optimization, it fails silently, deferring to the ordinary Python runtime. Therefore users are able to seamlessly incorporate Numba into new or existing projects. Previously, achieving similar behavior in Python would have required the usage of a hybrid language such as Cython, or by writing custom C extensions on top of CPython.

Numba has been extended with optimized versions of many common array operations, as well as common linear algebra routines available through the Numpy library. These are called automatically when used in a marked object. For common array operations, Numba is able to achieve comparable performance with compiled

Table 1. Laplace kernel evaluations over 100,000 randomly distributed particles. Coordinate components, and source charge densities chosen in interval $[0, 1)$. Repeated seven times for statistics.

Runtime (s)	
Numba (PyExaFMM)	4.71 ± 0.29 s
C++ (ExaFMM-T)	$1.99s \pm 0.01$ s

languages. See table (1), for a comparison of direct Laplace kernel evaluations, using (1), for 100,000 points between a Numba modified PyExaFMM, and ExaFMM-t, both of which are multi-threaded and SIMD vectorized, or [10] for further examples.¹

The Motivation for Building PyExaFMM

The question arises of whether it is feasible to develop a complex HPC application entirely within Python. The vision being that a domain specialist may be able to iterate from a prototype, to a high-performance application, using a language they are comfortable with. Python has the additional benefit of simple cross platform build tools, such as Conda, making it easy for developers to code once and deploy everywhere. An FMM implementation is an excellent case study to understand the maturity of Python for HPC development. As Numba is built to optimize operations on arrays, it represents a challenge to develop efficient representations and operations for the tree on which the algorithm is based.

PyExaFMM was developed to answer the above question. It is currently a single node implementation designed for three-dimensional problems, implementing parallel strategies on multicore architectures, similar to [6], [8]. We demonstrate the feasibility of building a high-performance implementation of a non-trivial algorithm solely in Python. As with all Python applications, performance is degraded by portions of code that are unavoidably interpreted. However, this represents a compromise between usability and performance. Where by usability, we mean the ease by which non-software specialists can understand, alter, or extend the codebase to their needs. Using lines-of-code [LOC] as a rough metric of usability, PyExaFMM consists of approximately 5,800 LOC, compared with

approximately 20,000 LOC and 50,000 LOC for the comparable C++ implementations, ExaFMM-T [8] and TBFMM [6] respectively.

In this article we begin by providing an overview of the mathematical formulation behind the KIFMM in [4], before discussing how Numba is used by PyExaFMM to accelerate computations. We continue by examining other mathematical and software based optimizations used to extract performance. Specifically, the efficient pre-computation, and application of FMM operations, and an overview of the software’s design. We conclude with a comparison of performance, in terms of accuracy, runtimes, and memory footprint, with the comparable state-of-the-art C++ implementation from the ExaFMM project, ExaFMM-T [8].

Formulating the Kernel Independent FMM

Consider the P2M operation illustrated in figure (2a). A cubic *equivalent* surface, sharing the same center, S_e is used to enclose a leaf from the tree containing set of points. It is discretized with evenly spaced quadrature points placed its faces, each of which supports an equivalent charge density. In order to find these equivalent densities, we match the potential generated by the points directly, ϕ_c , calculated using (1) with that generated by the equivalent densities at the *check* surface, S_c , in the far field, also centered on the box, which encloses both the box and the corresponding equivalent surface. Using (1) we find,

$$\sum_{x_i \in S_e, x_j \in S_c} K(x_i, x_j) q_i = \phi_c \quad (3)$$

where x_i and x_j are the quadrature points for the equivalent and check surfaces respectively, q_i is the equivalent charge density supported at x_i . Written in matrix form, we can write the equivalent densities, q_e , as the solution of the following linear system,

$$K q_e = \phi_c \quad (4)$$

$$q_e = K^{-1} \phi_c \quad (5)$$

¹Note that all benchmarks are reported on a six core Intel i7-9750H CPU

where K denotes a matrix with elements,

$$K_{i,j} = K(x_i, x_j), \quad x_j \in S_c, \quad x_i \in S_e \quad (6)$$

We identify these equivalent densities as a multipole expansion. The order of this expansion, p , is related to the number of the quadrature points, n , by,

$$n = 6(p - 1)^2 + 2 \quad (7)$$

Equation (5) is in general ill-conditioned, therefore we use a truncated singular value decomposition [SVD] of K , to calculate a pseudo-inverse. We cut off components corresponding to singular values smaller than machine epsilon in single precision.

As shown in figure (2a), the equivalent surface is enclosed by the check surface for the P2M operation. When this is the case we refer to the surfaces as the *upward* equivalent and check surface respectively. We find that the ratio of the side-length of the upward equivalent and check surfaces with respect to the tree node as 1.05 and 1.90 work well in practice.

We perform a similar calculation for the M2M operation, illustrated in figure (2b). Here the upward equivalent density of a child box, is matched to that of its parent,

$$K_c q_e^{child} = K_p q_e^{parent} \quad (8)$$

where K_c and K_p are matrices with elements as defined by (6), where S_c and S_e denote their relevant check and equivalent surfaces. To find the child's contribution to the parent's upward equivalent density, and hence multipole expansion, we solve,

$$q_e^{parent} = K_p^{-1} K_c q_e^{child} \quad (9)$$

we refer to $K_p^{-1} K_c$ as the *M2M operator*, with four unique operators in two dimensions (eight in three dimensions) corresponding to all the children of a given box. To find the corresponding *L2L operators* and *M2L operators*, we simply repeat the procedure with the appropriate check and equivalent surfaces. For the L2L operation, the check surface is placed inside the equivalent surface. This reflects the region in which we expect the local expansion to valid.

Here they are referred to as the *downward* check and equivalent surfaces respectively. For the M2L operation (fig. 2d) the upward equivalent surface of the source box, is matched to the potential generated by the downward equivalent surface at the target box.

We choose the upward equivalent surface to correspond to the downward check surface, and the downward equivalent surface to correspond to the upward check surface, for a given level. We then notice that the quantity K^{-1} in (5) can be re-used in the calculation of the M2M operator, as well as the M2L operator. Additionally, the M2M and L2L operators remain the same for all boxes in the tree.

As PyExaFMM uses adaptive octrees, there are further permutations in the composition of the near and far field we must describe. Consider a target box T , all source boxes S directly neighboring T (sharing an edge, face, or corner) are in T 's *U list*. Denoting a neighbor of a box at the same level as its *colleague*. The children of T 's colleagues which are not adjacent to T are in its *W list*, and the children of the colleagues of T 's parent which are not adjacent to T are in its *V list*. Finally, the *X list* of T consists of boxes for which T is in their *W list*. The *X* and *W* list are only calculated for leaf boxes. Collectively these lists are known as the *interaction lists* of T , and are illustrated for a two dimensional problem in figure (3).

Greater specification of the interaction lists allows for a more efficient algorithm implementation. Considering a source box in T 's *W list*, the multipole expansion of the source box is valid despite not being in T 's far field, therefore reducing the cost of calculating the near field contribution. For source boxes in T 's *X list*, the multipole expansion is not valid, as T is not in the far-field of the source box. However, the source charges can be evaluated directly at the quadrature points on T 's downward equivalent surface. PyExaFMM also enforces a *2:1 balance constraint* on the leaves, such that neighboring leaves may at most differ by one level. This constraint ensures small interaction lists, in the case of particle distributions with rapidly varying densities.

Implementing efficient KIFMMs relies on efficiently caching and loading the matrices that

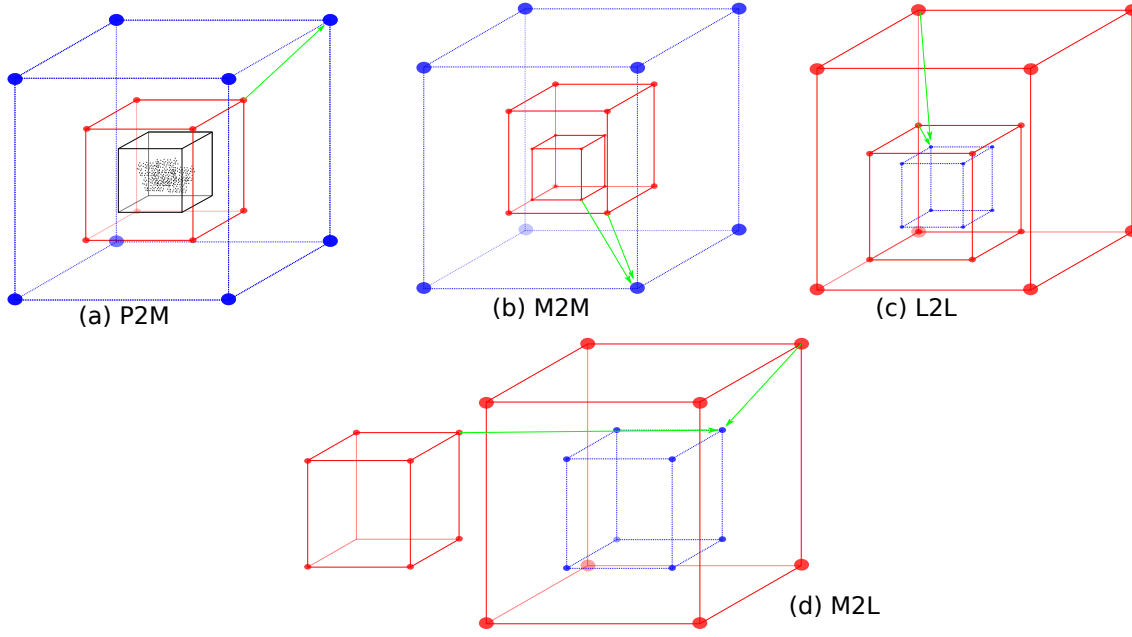


Figure 2. The operators of the KIFMM. Equivalent surfaces are denoted in red, check surfaces in blue, and the charged points are denoted in black. Surfaces are plotted with 8 quadrature points, for clarity.

X	X				V	V	
					V	V	
X	W	W	W	W	U	V	
	W	U	U	U			
	U		T		U	V	
X	U	U	U	U	U	V	
	V	V	V	V	V	V	

Figure 3. Interaction Lists for a target leaf box, T . Unmarked nodes are not considered to interact with T .

represent the P2M, M2M, L2L and M2L operators. As the algorithm consists of a series of matrix-vector products, one has to find a way to represent the coefficients of the multipole and local expansions, as well as tree nodes, to take

advantage of the cache-locality and vectorization offered by modern CPUs, or GPUs. The P2M and M2L steps are the largest bottlenecks in implementing fast KIFMMs [9]. The M2L operator has to be applied for each box in a given target's V list, and in three dimensions the maximum possible occupancy of a box's V list is 189. The P2M operator involves calculating the potential due to all particles in all leaves, at their check surfaces. However, the P2M operator is embarrassingly parallel over leaves, as is the M2L operator over a given V list.

TECHNIQUES FOR ACHIEVING PERFORMANCE

Tree Construction

Box positions in the tree are represented with a single 64-bit Morton encoded key [11]. A box at a given level, l , is said to have a displacement along a given axis in the range $[0, 2^l)$. In three dimensions, the index bits are interleaved as,

$$x_1 y_1 z_1 \dots x_n y_n z_n | l, \quad (10)$$

where x_i is the i^{th} bit in an index along the x axis, and the level is appended to the tail. We observe that we can find the parent of a given

Table 2. Common Morton operations

Method	Numba+Numpy	Numpy
Neighbor Finding	$1.041 \pm 0.003 \mu s$	$63 \pm 1 \mu s$
Parent Finding	$171 \pm 1 ns$	$573 \pm 3 ns$
Encoding a Point	$685 \pm 19 ns$	$6.170 \pm 0.001 \mu s$
Decoding a Key	$381 \pm 7 ns$	$16.500 \pm 0.002 \mu s$

key by removing its last three bits. Noticing the displacement of bits is consistent in (10), we optimize Morton encoding by storing the correct displacement bits of each index, up to values of 256, in a lookup table. We then consider 8 bits at a time of each index of a given anchor, using only bitwise operations to construct the encoding. Numba is then used to optimize a loop through a two dimensional array containing a set of anchors.

This strategy, of representing Morton operations in terms of lookups or bitwise operations and optimizing their application via Numba on a loop of values, is repeated for all common operations. This is because Numba is most efficiently able to translate basic instructions, with as little use of the Python object model as possible, as well as operations over Numpy arrays. Table (2) demonstrates the impact of Numba on selected operations in terms of runtime.

To find a linear representation of the tree, we assign Morton keys to each point, which is then stored as a vector. This is done with a top-down approach, considering batches of points sharing the same key, and refining to ensure that the maximum-particle-per-leaf constraint is met. We use Numba to implement multithreading over each batch. To ‘complete’ the tree from a linear representation of its leaves, we simply find its ancestors by repeatedly applying the parent finding function to a key. Balancing is then achieved by considering all keys level by level, bottom-up. If a given key’s parent, and it’s parent’s neighbors are not in the tree, we add them. We then remove overlaps from this final collection, favoring the finest box that covers a given area. This enforces a 2:1 balance. Numba allows for a typed-versions of Python datatypes, as well as for user created type-aliases. Therefore, we are able to represent this algorithm naturally using operations on sets. As these types are built on top of ndarrays we are able to easily convert to

Table 3. Building and balancing trees with N randomly distributed points, maximum depth of d , leading to M leaves.

N	M	PyExaFMM	ExaFMM-T
10,000	512	$19 \pm 2 ms$	$8.1 \pm 0.1 ms$
100,000	4096	$213 \pm 6 ms$	$64 \pm 2 ms$
1,000,000	32768	$2.76 \pm 0.04 s$	$0.64 \pm 0.01 s$

a linear array representation of the balanced, and complete tree.

Storing the tree in this way allows us to traverse it via index lookups, for example to find the parent or neighbors of a given key. These indices are precomputed and stored in a typed Numba dictionary. The coefficients of the multipole and local expansions, as well as the target potentials, for all boxes are each stored in a single vector, which are aligned using the index lookup tables. This scheme is designed for maximum cache-reuse. Table (3) illustrates benchmarks for tree construction, and balancing compared to ExaFMM-T.

Efficient Operators

Matrices for the unique M2M and L2L operators, as well that used in the P2M operation, are precomputed and cached in HDF5 database, and are loaded into memory at runtime.

The M2L operator is handled differently. We notice that there are at most $7^3 - 3^3 = 316$ unique M2L matrices for a given level in three dimensions corresponding to source and target box pairs, these are described uniquely with a *transfer vector* between a source and target [3]. We can collate them in a single matrix representing the M2L matrices for a given level,

[Illustration of matrix]

This can then be compressed using an SVD to find a low-rank representation.

[LOW RANK REPRESENTATION]

Which is then cached and stored on a level-by-level basis. When we encounter a specific M2L interaction, we simply compute the transfer vector, and lookup the appropriate compressed M2L matrix. Compression allows for faster application of M2L matrices ...

- Transfer vectors for M2L
- hashing, and why Numba is sometimes hard to use.
- Introduce RSVD algorithm [12].

- Demonstrate that SVD error dominated by FMM error.
- Demonstrate impact on runtime and storage.
- How to organise multithreaded computations in Numba

Software Design

The key is separating routines to be accelerated, and organising data ready to run. The data organisation part (and it's slowness) should be demonstrated as a bottleneck using experiment.

PERFORMANCE

Description of main experiments, and how they are conducted. What are the main results? Are they expected from theory? What conclusions can be drawn about developing HPC codes entirely in Python, is it worth it?

CONCLUSION

What have we learned, what will we be working on next?

ACKNOWLEDGMENT

SK is supported by EPSRC Studentship 2417009.

REFERENCES

1. L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, no. 2, pp. 325–348, 1987.
2. Z. Gimbutas and V. Rokhlin, "A generalized fast multipole method for nonoscillatory kernels," *SIAM Journal on Scientific Computing*, vol. 24, no. 3, pp. 796–817, 2003.
3. W. Fong and E. Darve, "The black-box fast multipole method," *Journal of Computational Physics*, vol. 228, pp. 8712–8725, dec 2009.
4. L. Ying, G. Biros, and D. Zorin, "A kernel-independent adaptive fast multipole algorithm in two and three dimensions," *Journal of Computational Physics*, vol. 196, no. 2, pp. 591–626, 2004.
5. Gimbutas, Z and Greengard, Leslie, "Fmmlib3d."
6. B. Bramas, "TBFMM: A C++ generic and parallel fast multipole method library," *Journal of Open Source Software*, vol. 5, no. 56, p. 2444, 2020.
7. E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for heterogeneous architectures," *Concurrency Computation*, vol. 28, pp. 2608–2629, jun 2016.
8. T. Wang, R. Yokota, and L. A. Barba, "Exafmm: a high-performance fast multipole method library with c++ and python interfaces," *Journal of Open Source Software*, vol. 6, no. 61, p. 3145, 2021.
9. I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast multipole method on heterogeneous architectures," *Communications of the ACM*, vol. 55, pp. 101–109, may 2012.
10. S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a LLVM-based Python JIT compiler," *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*, pp. 1–6, 2015.
11. H. Sundar, R. S. Sampath, and G. Biros, "Bottom-up construction and 2:1 balance refinement of linear octrees in parallel," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2675–2708, 2007.
12. N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM review*, vol. 53, no. 2, pp. 217–288, 2011.

Srinath Kailasa is a graduate student at University College London. He is currently pursuing a PhD in Computational Mathematics, having received an MPhys in Physics (2017) and an MSc Scientific Computing (2020) from the University of Durham, and University College London respectively. His research interests are in high-performance and scientific computing. Contact him at srinath.kailasa.18@ucl.ac.uk.

Timo Betcke is a Professor of Computational Mathematics at University College London. Contact him at t.betcke@ucl.ac.uk.

Tingyu Wang is a PhD student in Mechanical Engineering at the George Washington University. Contact him at twang66@email.gwu.edu.

Lorena. A. Barba is a Professor of Mechanical and Aerospace Engineering at the George Washington University. Contact her at labarba@email.gwu.edu.