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 based kernel-independent point fast multipole method implementation, built on the success of the ExaFMM project, to answer the question of whether it is 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], originally developed by Greengard and Rokhlin [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 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 $i \in [1, ..., N]$, each with a charge q_i, can be written as,

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

here $K(\cdot, \cdot)$ is called the Green's function, or kernel function, which for electrostatic problems in three dimensions is,

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

where ϵ_0 is the permittivity of free space. This kernel function is often referred to as the Laplace kernel. Without a loss of generality, we consider the set of sources and set of targets to correspond to the same 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 proscribed error bounds, making the simulation of physically realistic systems, potentially involving millions of degrees of freedom, tractable.

Algorithm

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. This truncation then 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 expansion representations can be done analytically, and are critical in the development of the $\mathcal{O}(N)$ algorithm. This approximation is only valid where the kernel function of a problem exhibits appropriate 'decay' characteristics, allowing us to assume that clusters of points far away from a region of interest can be well approximated with just a few representative expansion points.

In the FMM, the problem domain is described by a 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, the box is partitioned into four equal sized parts in two dimensions (or 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. This is referred to as the *local-to-local* [L2L] operation. 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] step. As there are a fixed number of points per leaf node, the complexity of this direct near field 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 by considering their respective potential generated in the far field.

The Motivation for Building PyExaFMM

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 restricted 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
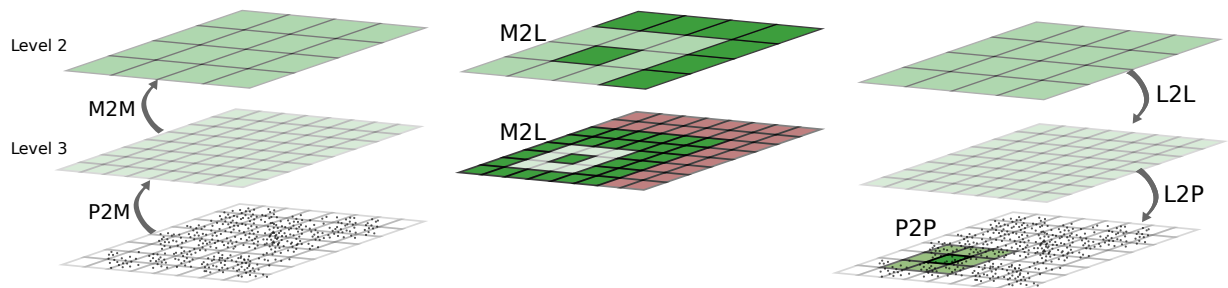
Figure 1. The FMM illustrated for a two dimensional problem with a uniform octree. The left column illustrates the first two steps of the upward pass, beginning with the P2M operation to find the multipole expansions at the leaf level, followed by an M2M operation. The central and right columns illustrate 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 it's near field of adjacent boxes, directly via the P2P operation.

computing [HPC] software.

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-based 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 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]. Therefore optimizing methods that loop over arrays. Furthermore, 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 Python code.

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 languages [10]. Furthermore, Numba allows users to decorate functions to be run on GPUs, allowing for users to develop heterogenous applications from a single Python source. By using Numba in conjunction with Numpy, it is possible to achieve compiled-language performance for a variety of numerical problems from within Python.

Therefore, 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].

In developing PyExaFMM we demonstrate the feasibility of building a high-performance implementation of a non-trivial algorithm solely in Python. We achieve runtime performance within an order of magnitude of a comparable state of the art compiled language implementation. 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 3,300 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 introducing Numba and discussing how it is used by PyExaFMM to accelerate computations, as well as program efficient data structures. We continue by discussing the mathematical and software based optimizations used by PyExaFMM for performance. Specifically how we efficiently pre-compute, cache, and apply FMM operations, and we provide an overview of the software design used in order to implement Numba optimizations effectively. We conclude with a discussion comparing the 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].

The Kernel Independent FMM

Consider the P2M operation illustrated in figure (2a). A cubic *equivalent* surface, S_e is used to enclose a box from the tree containing set of points, sharing the same center. It is discretized with quadrature points placed evenly across 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 in the far field, S_c,

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

Written in matrix form, we can solve for the equivalent densities, q_e,

$$K q_e = \phi_c \qquad (4)$$
$$q_e = K^{-1} \phi_c \qquad (5)$$

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

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

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,

$$K_c q_e^{child} = K_p q_e^{parent} \qquad (7)$$

where K_c and K_p are matrices with elements,

$$K_{i,j} = K(x_i, x_j), \quad x_j \in S_c, \ x_i \in S_e \qquad (8)$$

where S_c and S_e denote the relevant check and equivalent surfaces respectively. To find the

child's contribution to the parent's upward equivalent density, and hence multipole expansion, we solve,

$$q_e^{parent} = K_p^{-1} K_c q_e^{child} \tag{9}$$

we refer to $K_p^{-1} K_c$ as the *M2M operator*. 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.

PyExaFMM uses adaptive octrees, and as a result the near and far fields are have more

As originally developed in [4], 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.

- Interaction lists, and how these result from adaptivity. - Concept of balanced octrees, and why they help with computing interaction lists - List the difficulties in developing good KIFMMs (Ill Conditioning of c2e matrix, sparsification of the M2L operation) - The most intensive computations are P2M and M2L, explain why.

TECHNIQUES FOR ACHIEVING PERFORMANCE

Tree Construction
Tree construction routines on Morton coordinates. Multithreading of tree construction, as well as P2M evaluation. Experiment to demonstrate the speed of kernel evaluation, with caveat that data must be pre-organised.

Precomputing Operators
Transfer vectors, hashing, HDF5, loading into memory.

Compressing the M2L operator
Introduce [4]. Numerical bounds on error out of scope, but show how experiments demonstrate that the FMM error dominates the SVD error.

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

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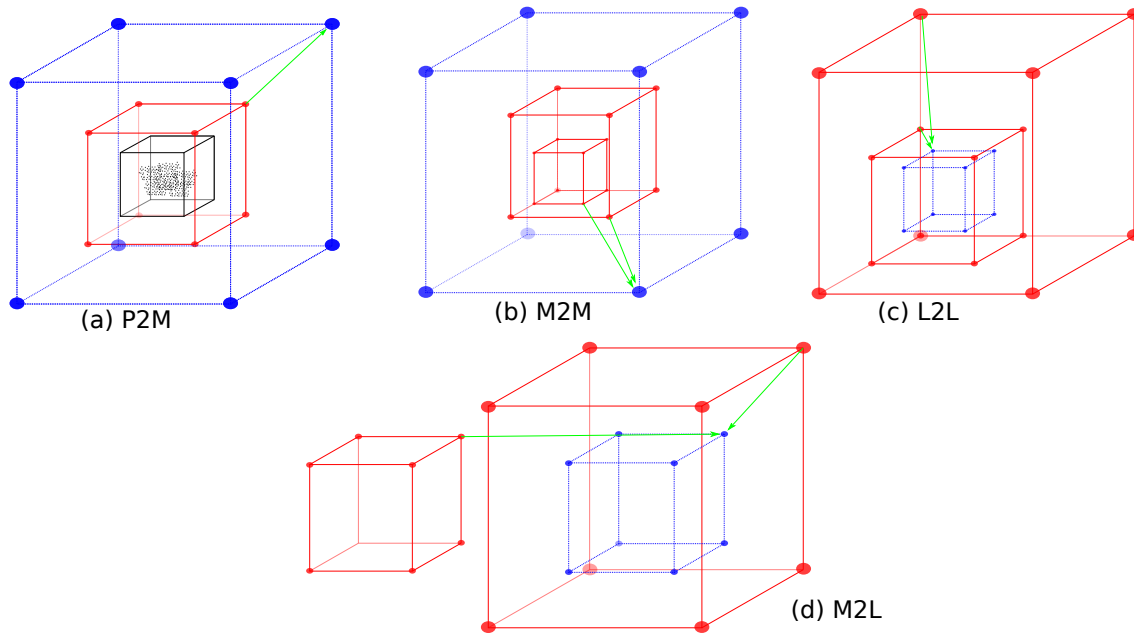
Figure 2. The operators of the KIFMM. Equivalent surfaces are denoted in red, check surfaces in blue, and the charged points are denoted in black. Surfaces are plotted with 8 quadrature points, for clarity.

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

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.

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

Department Head

Aerospace Engineering at the George Washington
University. Contact her at labarba@email.gwu.edu.