

PyExaFMM: Designing a highly-performant particle 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 particle 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 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 the pairwise interactions between N objects. This problem arises in numerous contexts in science

and engineering, for example in the calculation of the resulting electrostatic potentials due to a set of charged particles. Considering the calculation of electrostatic potential as our model problem, the potential, $\phi(x_j)$, for a given charged particle at position x_j , or ‘target’, due to N particles 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, which for electrostatic problems in three dimensions is,

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

where ϵ_0 is the permittivity of free space. This kernel function is often referred to as the Laplace kernel.

[Find more scientific examples where the algorithm is useful for better context]

Attempting to evaluate the sum in (1) directly for N target particles at positions x_j where $j \in [1, \dots, N]$ due to N sources, results in algorithm of $\mathcal{O}(N^2)$ runtime complexity, however the FMM is able to approximate (1) with just $\mathcal{O}(N)$ runtime complexity, with proscribed error bounds.

The key idea underlying the FMM is to encode the potential in the far field due to a cluster of particles with a representative analytic *multipole expansion* centered on the cluster, which can be truncated to tune for desired accuracy. This truncation allows one to approximate the sum in (1) with fewer calculations. In regions centered far away from the cluster, their potentials can be encoded in 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.

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, such that at the first level of refinement the box is partitioned into four equal sized parts in two dimensions (or eight in three dimensions), known as it’s children, which are each subsequently refined in a similar fashion. The level of refinement is defined by a parameter dictating the maximum allowable number of particles contained within a box. The finest boxes

covering a portion of the domain that remain after refinement are known as leaves. One can choose to refine *adaptively*, such that the leaves may be of different sizes, reflective of the underlying particle distribution, or *non-adaptively*, such that the leaves are all of a uniform size.

The FMM then consists of two sequential traversals through this tree, where boxes are considered level by level, with level zero being the unrefined box which defines the domain, level one being the level of its child boxes, and so forth. Firstly, during the *upward pass*, multipole expansions for particles contained in boxes at the highest level of refinement, or leaf level, of the tree are formed. This is referred to as the *particle-to-multipole* [P2M] step. In order to obtain the multipole expansion at their parent level, the multipole expansions of a given box’s children at the leaf level have their expansion centre translated to the centre of their shared parent box, and the coefficients are summed. This is referred to as the *multipole-to-multipole* [M2M] step. 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 particles 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] step. The local expansion is then transferred to the children of the given box by shifting the expansion centre. 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] step. Crucially the extent of the far field not already encapsulated in the local expansion shrinks as we descend down the octree. 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-particle [L2P] step. Finally, the near fields of a given leaf box are evaluated directly using (1), this is referred to as the *particle-to-particle* [P2P] step. As there are a fixed number of particles per leaf node, the complexity of this direct near field evaluation is bounded. This scheme is 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 expansions for all boxes deemed to be in the far field, as well as the recursive procedure of the FMM. As a result of this, each box must only consider it's 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 $\mathcal{O}(N)$.

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 general kernels [2] have been developed, however there are fewer methods that rely only on numerical kernel evaluations, rather than analytic series expansions, examples include [3], [4], [5]. 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 densities supported on a surface enclosing the cluster, with the fields generated by the charges matched with the equivalent field via a least-squares fitting in the far field. 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. Further work has extended the method to include oscillatory problems [6]. Alternatively, Fong et. al [3] use a Chebyshev based interpolation scheme to approximate the action of the kernel function in the far-field.

At present, numerous high-quality open source implementations exist to solve FMM problems, both using analytic expansions for various kernels [7], [8], as well as using the KIFMM of Fong et. al [9], [10], [11], and Ying et. al

[12], [13]. Some of these implementations are able to scale on supercomputing clusters, and solve problems involving millions of degrees of freedom [13]. Others are designed to be run on a single workstation [9], [10], [12]. These softwares are often developed in compiled languages, such as Fortran or C++, often making use of numerous non-standard libraries and features. Furthermore, compiled languages such as C++ are often accompanied by complex build processes, making deployment to different platforms or architectures non-trivial. On the other hand, many domain specialists usually have limited computational skill-sets, often restricted to subsets of Python, for data-analysis or simple numerical computing, 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.

In recent years, HPC libraries that aid in bypassing the limitations of the Python interpreter have emerged. The most most pertinent example is Numba, a 'just in time' [JIT] compiler for CPython, the most commonly used implementation of Python interpreter written in the C language. JIT refers to the fact that the compiler waits until a Numba enhanced function or class is called 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 the multi-dimensional array objects provided by Numpy, the ndarray. Numpy already provides a Python binding to HPC libraries that are 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 [14]. Furthermore, Numba is a 'drop-in' library. It's simply installed via package manager, and functions or classes are marked to be optimized by a single decorator. If Numba is unable to optimize the marked function or class, 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 the ability to interface with linear algebra routines, available through the Numpy library, which it calls automatically when used in a marked function or class. For common array operations, Numba is able to achieve comparable performance with compiled languages [14]. Furthermore, Numba allows users to decorate functions to be run on GPGPUs, 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 HPC application entirely within Python. The vision being that a domain specialist may be able to trivially iterate from a prototype, to a high-performance application, using a language they are comfortable with. Python has the additional benefit of easy to use cross platform build tools, such as Anaconda [15], 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 [9], [12].

In developing PyExaFMM we have demonstrated the possibility of building a high-performance implementation of a non-trivial algorithm purely in Python, achieving runtime performance within an order of magnitude of a 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. Using lines-of-code [LOC] as a rough metric of usability,

by which we mean the ability to understand and alter the codebase, PyExaFMM consists of approximately 3,300 LOC, compared with approximately 20,000 LOC and 50,000 LOC for the comparable C++ implementations, ExaFMM-T [12] and TBFMM [9] 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, and provide an overview of the software design used in order to implement 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 [12].

THE KERNEL-INDEPENDENT FAST MULTIPOLE METHOD

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, and provide an overview of the software design used in order to implement 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 [12].

Algorithm TECHNIQUES FOR ACHIEVING PERFORMANCE

What is Numba?

What is it, how is it useful? Where do we use it, and why there. How much difference can it make in an idealized routine. What doesn't work, why doesn't it work. Where to be careful. Programming to an (invisible) framework ...

Where is numba used heavily? 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 step with a randomised SVD

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

Software Architecture

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 COMPARISON WITH STATE OF THE ART

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. P. G. Martinsson and V. Rokhlin, "An accelerated kernel-independent fast multipole method in one dimension," *SIAM Journal on Scientific Computing*, vol. 29, no. 3, pp. 1160–1178, 2007.
6. B. Engquist and L. Ying, "Fast directional multilevel algorithms for oscillatory kernels," *SIAM Journal on Scientific Computing*, vol. 29, no. 4, pp. 1710–1737, 2007.
7. Gimbutas, Z and Greengard, Leslie, "Fmmlib3d."
8. Yokota, Rio and Barba, Lorena A., "Exafmm."
9. B. Bramas, "TBFMM: A C++ generic and parallel fast multipole method library," *Journal of Open Source Software*, vol. 5, no. 56, p. 2444, 2020.
10. E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Mennner, and T. Takahashi, "Task-based FMM for multicore architectures," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. 66–93, 2014.
11. E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Mennner, and T. Takahashi, "Task-based FMM for heterogeneous architectures," *Concurrency Computation*, vol. 28, pp. 2608–2629, jun 2016.
12. 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.
13. 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.
14. 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.
15. Continuum Analytics, "Anaconda software distribution,," 2016-11-01.

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.

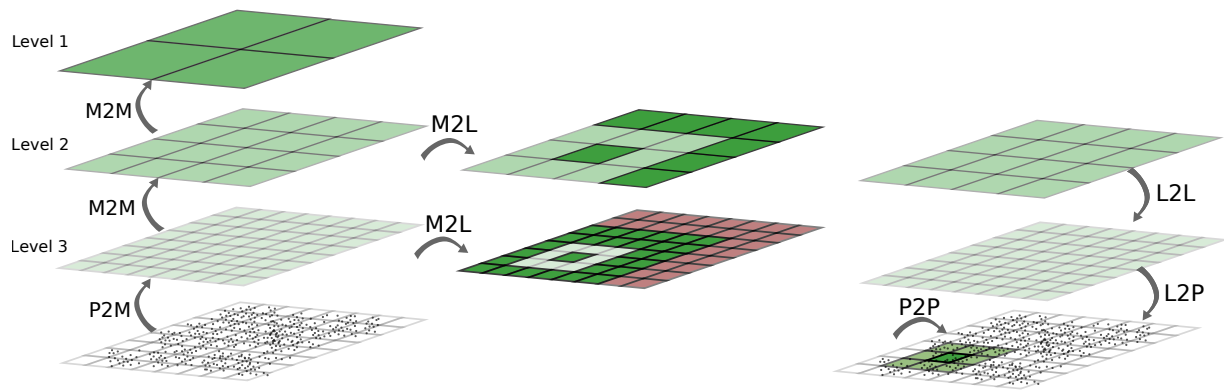


Figure 1. FMM illustrated for a two dimensional problem.

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.