Department: Head
Editor: Name, xxxx@email

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

S. Kailasa
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

T. Betcke
Department of Mathematics, University College London

***Abstract*—The particle fast multipole method is a good case study for understanding the efficacy of Python for developing high-performance software for non-trivial algorithms, due its reliance on a hierarchical tree data structure. In this paper we describe the mathematical and software engineering techniques used to extract performance for PyExaFMM, a Python based solver for the particle fast multipole method, accelerated with Numba, designed to be run on single-node multicore architectures. We report that we achieve runtimes within $\mathcal{O}(10)$ of the state of the art C++ implementation, with comparable accuracy and memory footprint for three dimensional problems in double precision.**

CPYTHON IS THE ORIGINAL AND MOST popular implementation of the highly productive, dynamically typed and interpreted, Python programming language, and is written in C. CPython was designed with safety and developer productivity in mind, rather than high-performance computing [HPC]. Pythons's dynamic typing forces objects to be passed through the interpreter loop at runtime, and applications are restricted to a single thread via a 'global interpreter lock' [GIL] to ensure thread safety. However, Python's popularity has lead to the development of numerous open-source tools for HPC with CPython, that allow users to bypass the interpreter, develop mul-

 Published by the IEEE Computer Society

tithreaded applications, and even deploy source code written in Python to GPUs.

PyExaFMM[1] is a solver for the three-dimensional particle fast multipole method [FMM], designed to be run on single-node multi-core architectures. It was designed to test the efficacy of Numba, a 'just-in-time' [JIT] compiler, for developing HPC applications in CPython. Building on the success of the ExaFMM project's comparable C++ implementation [1], we wanted to test whether we could retain the productivity benefits of working in Python while achieving performance comparable to compiled languages. The FMM consists of a recursive loop through a hierarchical data structure. Representing computations and data efficiently while retaining performance is challenging, therefore it offers a good benchmark for studying the efficacy of Python and Numba for developing efficient software for complex algorithms.

Numba appears to offer excellent tools for overcoming the performance problems of Python. It bypasses the interpreter for operations involving loops over Numpy arrays and numeric scalars, translating Python source code into efficient platform-dependent machine code using the LLVM infrastructure. LLVM applies hardware dependent optimizations such as single-instruction multiple data [SIMD] vectorization over loops, to the intermediate bytecode representation [IR] produced by Numba. Bypassing the Python interpreter in this way can make Numba compiled functions competitive with compiled languages such as C++ and Fortran. However, we note that Numba is not able to map all operations specified in Python to efficient machine instructions, or vectorize loops, if they contain operations outside of the subset of numeric Python it optimizes for. Furthermore, Numba still requires interaction with the Python interpreter to pass data to Numba compiled functions, as well as to interact with non-optimized parts of the codebase involved in data-organization or calls to incompatible libraries. Therefore, developers have to be careful to organize their code in such a way as to restrict interaction with the Python interpreter, as well as ensure that their source code can be optimized by Numba, to ensure that they experience a performance benefit from using it.

Numba is a 'drop-in' tool, functions or classes are marked for JIT compilation with a decorator, and 'polymorphic dispatching' picks up input and output type data from the objects arguments at runtime - compiling the machine-code for the function for this type signature if it does not already exist in cache. The function itself is called from a Python wrapper which forms the interface between the Python runtime and a Numba compiled function. Therefore Numba fits easily into existing Python projects. Numba has been extended with efficient implementations of many of the array manipulation and linear algebra operations offered by Numpy, as well as multithreading functionality via iterations over a parallel range iterator, reminiscent of OpenMP's parallel for loops. Furthermore, Numba supports the writing of Python kernels for AMD and NVidia GPUS, and is fully integrated with the CuPy library.

Numba therefore provides a 'framework' for developing heterogenous, cross-platform, applications using only Python source code. Numba's framework impacts the design of HPC software as performance is dictated by the interaction between the Python interpreter and Numba optimized functions. Developers have to be careful to design Numba functions that are vectorizable, and multithreaded functions that are cache-optimal. Furthermore, care has to be taken to design robust software that limits usage of the Python objects, and instead designs operations in a 'data-centric' manner, around Numpy arrays. In this paper we begin by briefly summarizing the kernel independent FMM [KIFMM] algorithm used by PyExaFMM, before proceeding to describe the mathematical optimizations, and software design strategies we used for achieving performance with Numba. We discuss in detail how we parallelize FMM operations to maximize cache-reuse, our approach to designing functions for Numba compilation, as well as how we architected PyExaFMM to minimize interactions between Numba compiled functions and the Python interpreter. We conclude with benchmarks comparing the memory usage, runtimes and accuracy of the software with respect to the comparable state-of-the-art C++ implementation from the ExaFMM

[1] https://www.github.com/exafmm/pyexafmm

project, ExaFMM-T [1].

THE FAST MULTIPOLE METHOD

The particle FMM [2] is an algorithm for approximating the N-body problem, in which one aims to calculate the pairwise interactions between N particles. Consider a problem domain $D \subset \mathbb{R}^3$, containing a set of 'source' particles at positions x_i, and their interaction with a 'target' particle at position y_j where $i \in [1, ..., N]$, where $x_i, y_j \in \mathbb{R}^3$. The pairwise interaction of the target with all sources can be written as,

$$\phi_j = \sum_{i=1}^{N} K(x_i, y_j) q_i \tag{1}$$

where $K(.,.)$ is called the 'kernel' or Green's function, and the interactions are weighted by q_i. This calculation appears in numerous contexts across science and engineering. For example, if we interpret q_i as a charge, and take the Green's function to be,

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

we recognize (1) as the calculation of the electrostatic potential ϕ_j at y_j due to source particles at positions x_j. Without loss of generality, we can consider the sources and targets to correspond to the same set of particles, we take this as our benchmark problem. A naive calculation of (1) at N target positions results in an algorithm of $\mathcal{O}(N^2)$ runtime. The FMM is able to approximate this in $\mathcal{O}(N)$, with proscribed error bounds. It works by partitioning D using a hierarchical tree. Level 0 of the tree corresponds to a cube containing all source and target particles, it is then recursively partitioned into 8^l cubes where l is a given level. The maximum level l, or *leaf level*, of partitioning in a subset of D is set by a user defined constant for the maximum number of particles allowed in a given tree node. Refinement proceeds until this condition is satisfied. Optionally, the domain can be refined in a *non-uniform* manner such that the leaf level consists of boxes of different sizes, reflecting non-uniform particle distributions. Figure (1b) illustrates the relationship between a given leaf box T and its neighbors, termed *interaction lists*, for a non-uniform tree in $\mathbb{R}^2$. The U list consists boxes adjacent to T - i.e. sharing an edge or vertex (or face in $\mathbb{R}^3$), known as its *neighbors*. Neighbors at the same level of discretization are known as *colleagues*. The V list consists of child boxes of the colleagues of T's parent box, which are not-adjacent to T. The W list consists of child boxes of T's colleagues, which are not-adjacent to T, and the X list consists of boxes for which T is in the W list. The X and W lists only occur in non-uniform trees, are only formed for leaf boxes.

The FMM consists of six operators: P2M, M2M, M2L, L2L, L2P and P2P, where a given operator is read as 'X to Y. 'P' stands for particle(s), 'M' for multipole expansion and 'L' for local expansion. The operators can be seen to correspond to translations between expansion representations. A multipole expansion represents the aggregation of charge located within a given box in the tree, where the expansion center is set to coincide with the box center, and can be truncated to an 'expansion order', p, for tunable accuracy. A multipole expansion can be translated into a local expansion centered on another box, inside of which it is valid, and it represents the aggregation of charge corresponding to the multipole expansion. This is the M2L operator. Similarly the expansion centers of local or multipole expansions can be shifted, which are the L2L and M2L operators respectively. The P2M operator is act of forming a multipole expansion for a set of particles. L2P and P2P refer to direct evaluations using (1) between local expansion coefficients, or source particles, with a set of target particles, respectively.

Figure (1) illustrates the FMM for a problem in $\mathbb{R}^2$. Figure (1a) shows the recursive partition, and the first step of the algorithm - the *upward pass*. The tree is traversed bottom-up, in which multipole expansions are found for boxes at the leaf level (P2M), the centers of which are shifted to the center of the corresponding 'parent box' (M2M) and expansion coefficients summed in order to find the parent's multipole expansion. Figure (1c) and Figure (1d) show the second step - the *downward pass*. The tree is traversed top-down, starting at level 2. M2L transfers for nodes in a box's V list are performed, followed by L2L transfers. Resulting in the local expansion for

each leaf box. These local expansions compress the far field component of potential for targets within a given leaf. The far field is defined by a box and its ancestors' V lists. The near field, defined by the U, W and X lists, is evaluated directly for boxes at the leaf level (P2P). For non-uniform trees the P2P step can be made more efficient by taking advantage of the X and W lists. Consider a leaf box T, we can use the multipole expansion of boxes in its W list to evaluate their contribution towards potential for target points in T (M2P). For boxes in T's X list, we can evaluate the source charges directly at the points supporting the local expansion (S2L), before performing L2P.

As we traverse down the tree in the downward pass, more and more of the far field contribution to potential is compressed in the local expansion for a given box. We find this far field component by performing $M2L$ transfers for a given box, over its V list, the size of which is bounded by a constant. Therefore we see the complexity of the FMM to be dictated by the number of nodes in the tree, which due to the maximum particle per node constraint will be restricted to $\mathcal{O}(N)$.

In the KIFMM [3], we use evaluations of the kernel function, rather than analytic expansions, to find the multipole and local expansions and translate between them. Consider the P2M operator for a given box illustrated in figure (2a). The multipole expansion is described by equivalent charges placed at n quadrature points evenly spaced on an *equivalent surface* enclosing the box. Where n is related to the expansion order p by,

$$n = 6(p-1)^2 + 2 \tag{3}$$

The sum (1), for these equivalent charges is matched via least squares fitting to the *check potential* ϕ^c calculated from the charges in the box directly at quadrature points on an *check surface* that encloses the equivalent surface and the box. Each component of ϕ^c, corresponding to a point on the check surface y_j, is calculated as,

$$\phi^c_j = \sum_{i=1}^{N_{box}} K(x_i, y_j) q_i \tag{4}$$

where there at most N_{box} particles in the box, with charges q_i and at positions x_i. We find the equivalent charges q^e_i corresponding to points x_i on the equivalent surface with,

$$\sum_{i=1}^{N_e} K(x_i, y_j) q^e_i = \phi^c \tag{5}$$

Where N_e is the number of quadrature points on the equivalent surface. In matrix form,

$$K q_e = \phi^c \tag{6}$$

$$q_e = K^{-1} \phi^c \tag{7}$$

The matrix on the right hand side of (7) is taken to be the P2M operator. This logic is repeated to form the other KIFMM operators, the required check and equivalent surfaces and matchings are illustrated in figure (2). For the M2M and L2L operators, the check potential is formed with the child/parent equivalent charges respectively. For the M2L operator, it is formed with the equivalent charges of boxes in the V list. The L2T, M2T, S2L and P2P operators are evaluated directly using (1). We note that large values of p lead to poor conditioning in the inversion of the matrix K.

TECHNIQUES FOR ACHIEVING PERFORMANCE

Compressing M2L Operator with Randomized SVD

- Introduce randomized compression in 500 words. Demonstrate by applying directly to M2L matrix, rather than a formal intro due to space constraints.

Software Design

- (1) Function composition. PyExaFMM consists of (compositions of) simple functions that make use of: bitwise operations, simple arithmetic. Example of Morton encoding functions, and how we don't pay further cost for passing into Numba once already there. Describe how parametrization of operators works in PyExaFMM (i.e. how data is copied to operators and the relative costs involved). Should mention that tree building library is separated from FMM library.

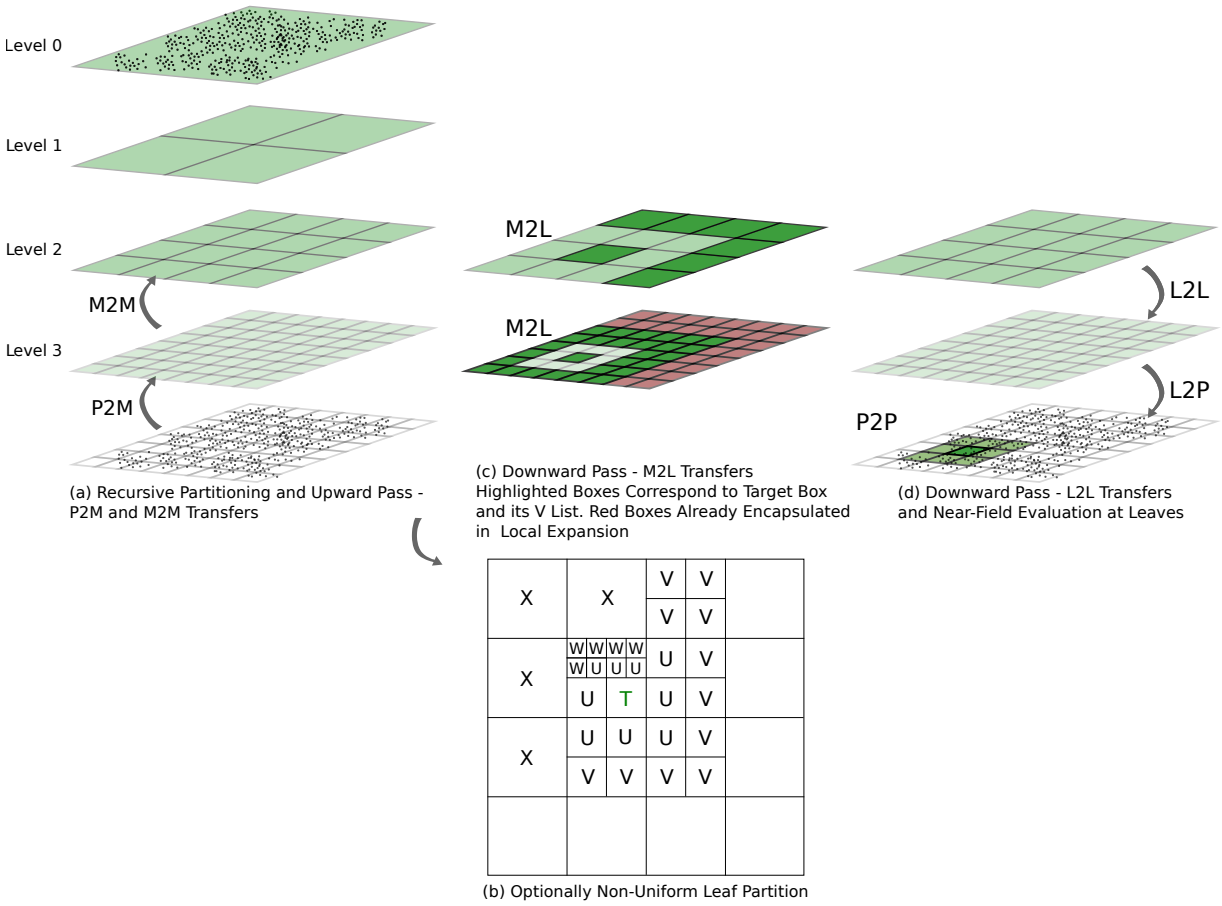

Figure 1. FMM algorithm and operator actions, including tree construction, illustrated for problem in $\mathbb{R}^2$.

- (2) Data oriented programming and API. API exposed via FMM class is a thin wrapper around numpy arrays containing expansion coefficients, potentials/gradients, and particle positions/charges. FMM loop logic is coded in Python, however actual floating point operations take place in numba compiled functions - separated via a 'backend' interface. Storage of coefficients and results in vectors. Lookup tables for indices.

- (3) Operator and tree precomputations. Technology used (HDF5). User configuration for experiments. Offer time benchmark for precomputations wrt to C++.

- (4) Conciseness of library. Python's expressiveness allows PyExaFMM to be a concise library, consisting of ∼3,000 lines of code, in comparison to comparable single-node C++ implementations ExaFMM-T (∼7000 lines) [1] or TBFMM (∼20,000) [4]. This makes it easy for domain specialists to hack for their needs, without having to traverse a large/complex codebase.

Parallelization Strategies for FMM operators

- (0) Carefully set threading layer/settings of Numba.

- (1) Allocate memory (if possible) aligning targets/sources - and run kernel function over them. Maximizes cache re-use (P2M, L2P, Near Field)

- (2) Simple parallel for loop over interactions - S2L (not enough computation for it to be worth it) M2T (too expensive)

- (3) Threads restricted to simple blas l3 operations and lookups, except M2L operator. This requires more than simple lookup. Comment on need for fast hash function compatible with Numba. Comment on how this is (a) an example of function composition of simple functions (b) an example of a potential pitfall for naive users,

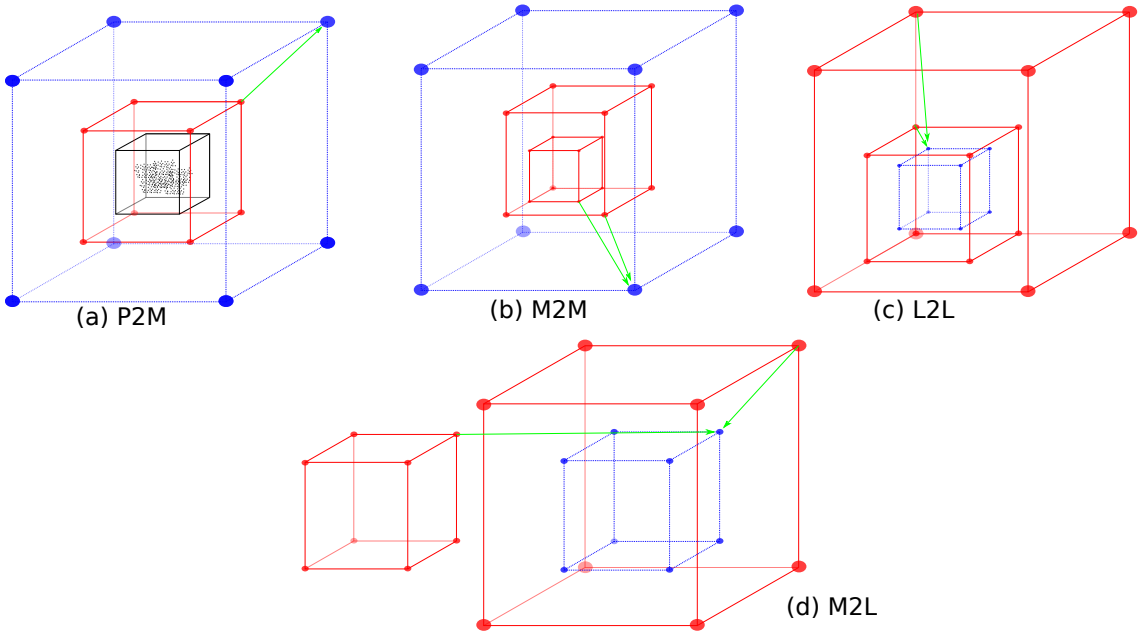

Figure 2. KIFMM operator calculations for order $p = 2$ expansions. Equivalent charges placed at quadrature points on the (red) equivalent surfaces, are matched at quadrature points on the (blue) check surfaces. Charges are plotted in black, and green arrows are used to indicate least-squares fittings.

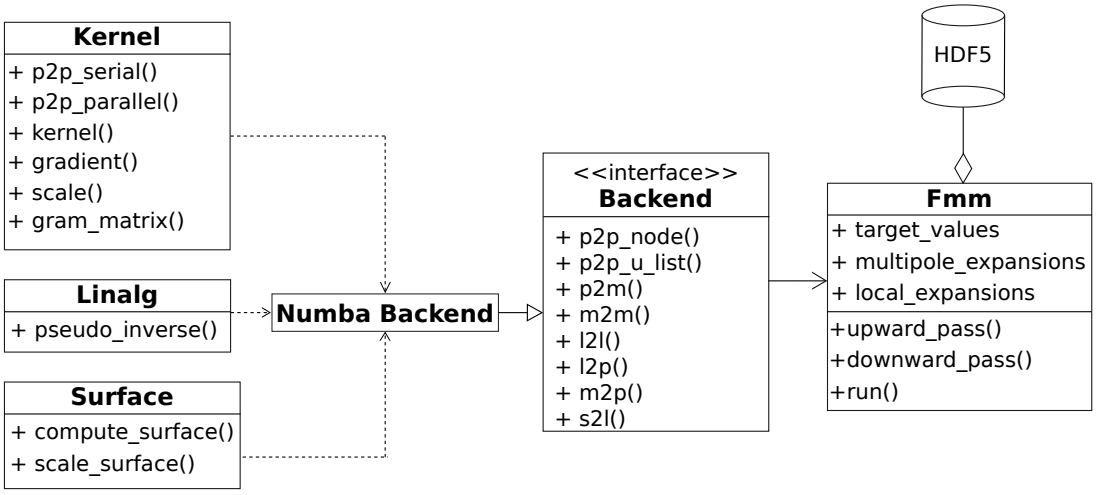

Figure 3. Simplified UML model of all PyExaFMM components. Trees and operators are precomputed and stored in the HDF5 database. Except for the 'Fmm' object which acts as the user interface, all other components are modules consisting of simple functions.

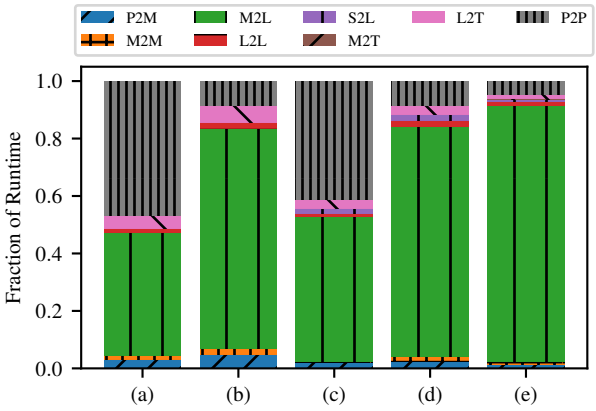

Figure 4. Foo bar

and will lead to slow performance if you use a native hashing function.

Benchmarks

- (1) Mention that C++ lib doesn't do M2L compression, instead does FFT based convolution.

- (1) Comment on fractional runtimes of operators. Relative impact of compression (minimal), and how this shows that the complexity of the two softwares must be similar, and that the difference must be explained due to the Python/Numba interface.

ACKNOWLEDGMENT

SK is supported by EPSRC Studentship 2417009.

REFERENCES

1. 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.
2. L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, no. 2, pp. 325–348, 1987.
3. 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.
4. B. Bramas, "TBFMM: A C++ generic and parallel fast multipole method library," *Journal of Open Source Software*, vol. 5, no. 56, p. 2444, 2020.

Srinath Kailasa is a Graduate Student at UCL, currently pursuing a PhD in Computational Mathematics. He completed an MPhys in Physics (2017) and an MSc in Scientific Computing (2020) from Durham University and UCL respectively, interspersed with time as a Software Engineer in industry. His research interests are in high-performance and scientific computing. Contact him at srinath.kailasa.18@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.

Timo Betcke is Professor of Computational Mathematics at University College London. Is the lead investigator of the Bempp project, an open-source boundary element library. He studied Engineering in Germany as Undergraduate and then completed a PhD in Oxford in Numerical Analysis. From 2005 to 2006 he had various research positions until he became a Lecturer at UCL in 2011. Since 2018 he is a full Professor in the Department of Mathematics at UCL. Contact him at t.betcke@ucl.ac.uk.

Table 1. Relative error, runtime and peak memory consumption in comparison to the SOTA. Experiments run with $N = 1,000,000$ points tested in two geometries: (1) distributed randomly in a cubic unit box, (2) distributed randomly on the surface of a sphere with unit radius, leading to M leaves in their respective geometries, with a maximum of 150 points per leaf, multipole and local expansions of order p, and a compression rank k for PyExaFMM. Charge densities are chosen in the interval $[0, 1)$. Runtimes exclude tree building time. Reported to 3 significant figures after a single run.

				Runtime		Peak Memory		Relative Error	
k	M	p	Geometry	PyExaFMM	ExaFMM-T	PyExaFMM	ExaFMM-T	PyExaFMM	ExaFMM-T
10	17,017	6	Sphere	10.6 ± 0.1 s	0.41 ± 0.04 s	2.96 GB	2.34 GB	1.00e-4	8.75e-5
	32,768		Random	13.2 ± 0.2 s	0.41 ± 0.05 s	4.93 GB	2.98 GB	8.75e-5	7.66e-5
		10	Sphere	57.0 ± 0.1 s	1.78 ± 0.04 s	3.09 GB	3.22 GB	2.00e-6	2.86e-6
			Random	131 ± 2 s	2.11 ± 0.06 s	4.93 GB	3.88 GB	1.71e-6	3.84e-6
100	17,017	6	Sphere	10.6 ± 0.1 s	0.41 ± 0.04 s	2.96 GB	2.34 GB	1.00e-4	8.75e-5
	32,768		Random	13.2 ± 0.2 s	0.41 ± 0.05 s	4.93 GB	2.98 GB	8.75e-5	7.66e-5
		10	Sphere	57.0 ± 0.1 s	1.78 ± 0.04 s	3.09 GB	3.22 GB	2.00e-6	2.86e-6
			Random	131 ± 2 s	2.11 ± 0.06 s	4.93 GB	3.88 GB	1.71e-6	3.84e-6
Full Rank	17,017	6	Sphere	10.6 ± 0.1 s	0.41 ± 0.04 s	2.96 GB	2.34 GB	1.00e-4	8.75e-5
	32,768		Random	13.2 ± 0.2 s	0.41 ± 0.05 s	4.93 GB	2.98 GB	8.75e-5	7.66e-5
		10	Sphere	57.0 ± 0.1 s	1.78 ± 0.04 s	3.09 GB	3.22 GB	2.00e-6	2.86e-6
			Random	131 ± 2 s	2.11 ± 0.06 s	4.93 GB	3.88 GB	1.71e-6	3.84e-6