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 an 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,

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

called the 'Laplace kernel', we recognize (1) as the calculation of the electrostatic potential ϕ_j at y_j due to source particles at positions x_i with charges q_i. 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 cubic box containing all source and target particles, it is then recursively partitioned into 8^l boxes 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. The level of non-uniformity in the final tree can be resricted by a *balance condition*, which dictates the maximum difference in size between two neighboring boxes. 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 M2M operators respectively. The P2M operator is the 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 topdown, 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, and are evaluated directly at the target particles (L2P). The far field is defined by a box and its ancestors' V lists. The near field, defined by its 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 to the *check potential* ϕ^c calculated from the charges in the box directly at quadrature points on a *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). We note that for the L2L and M2L operators, the equivalent surface around the target box encloses the check surface, which itself encloses the target box. This is reflective of the region of validity of local expansions. The check potentials are then formed with the child/parent equivalent charges for the M2M and L2L operators respectively, and with the equivalent charges of boxes in the V list of a given box, for its corresponding M2L operator. The L2P, M2P, 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, and in practice we solve this by taking the *backward-stable pseudo-inverse* of K, introduced by Malhotra et. al [4].

TECHNIQUES FOR ACHIEVING PERFORMANCE

Compressing the M2L Operator with SVD

The maximum size of a V list in three dimensions is 189. Therefore naively forming M2L matrices, and performing inversions for each box during the downward pass is costly. Instead, we label each unique V list interaction at a given tree level with a *transfer vector* [5]. Note that there are at most $7^3 - 3^3 = 316$ transfer vectors at

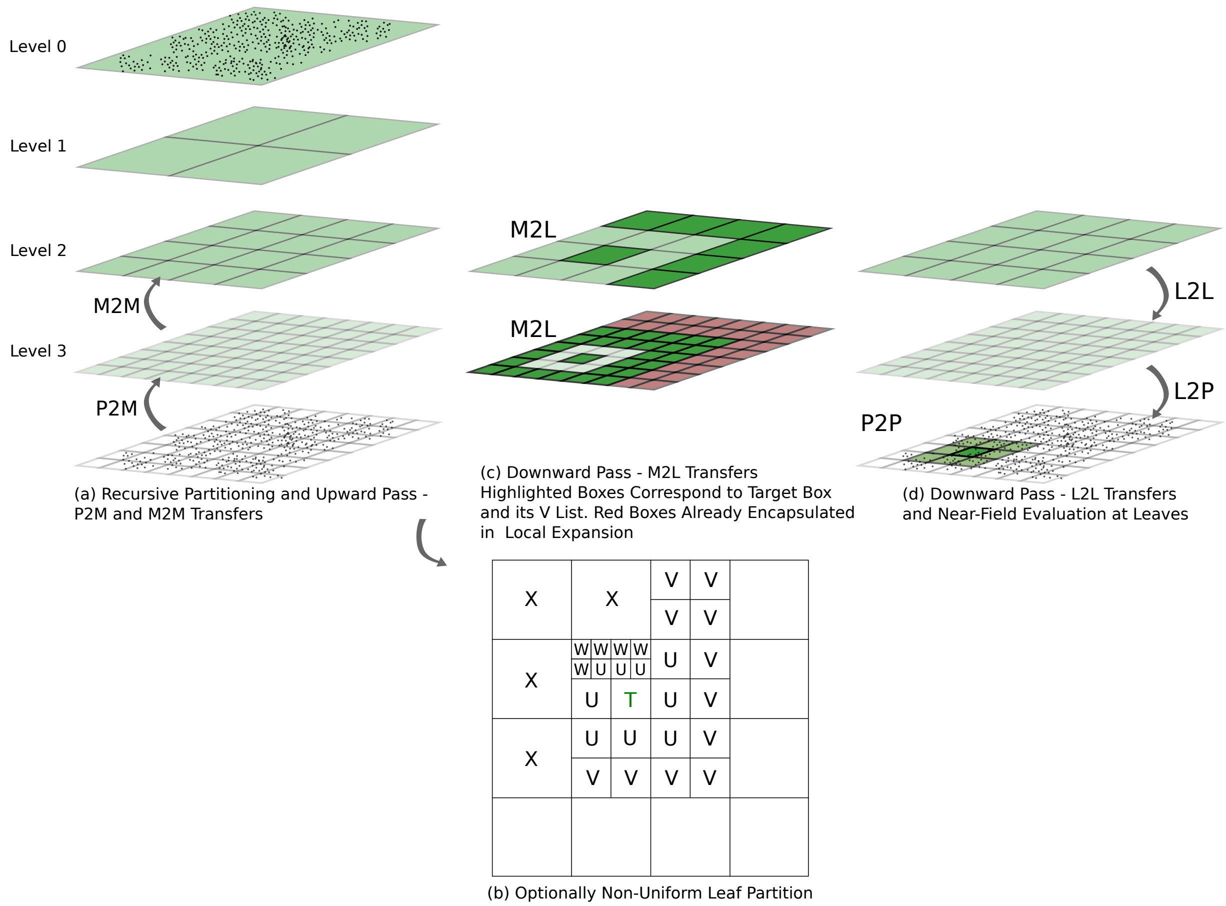

Figure 1. FMM algorithm and operator actions, including tree construction, illustrated for problem in $\mathbb{R}^2$.

a given level. As usual we seek the equivalent charge around a target box due to a source box in its V list, by matching the kernel evaluations at a check surface enclosing the target box and its equivalent surface,

$$K^{se2tc} q_e^s = K^{te2tc} q_e^t \tag{8}$$

where K^{se2tc} is the matrix with elements calculated using (2), between the source box's equivalent surface, and a target box's check surface, K^{te2tc} is calculated using the target box's equivalent and check surfaces, q_e^s is the known equivalent charge at the source box, and q_e^t is the unknown equivalent charge at the target box. We form the M2L matrix by inverting K^{te2tc},

$$q_e^t = \underbrace{(K^{te2tc})^{-1} K^{se2tc}}_{M2L} q_e^s \tag{9}$$

We can form an M2L matrix at level l, for source boxes corresponding to all unique transfer vectors for a single target box, and concatenate them,

$$M2L_l = \begin{pmatrix} M2L_1 & M2L_2 & \dots & M2L_{316} \end{pmatrix} \tag{10}$$

Note that $M2L_l \in \mathbb{R}^{N_c \times 316 N_e}$, where N_c and N_e are the number of quadrature points on the check and equivalent surfaces, respectively. When we encounter an M2L interaction during the downward pass, we simply compute the corresponding transfer vector and look up the submatrix of (10) that corresponds to its M2L matrix. Equation (10) can be pre-computed and cached. PyExaFMM further reduces the application and storage costs of concatenated M2L matrices for a given level by applying *SVD compression* to (10). Consider taking the SVD of $M2L_l$

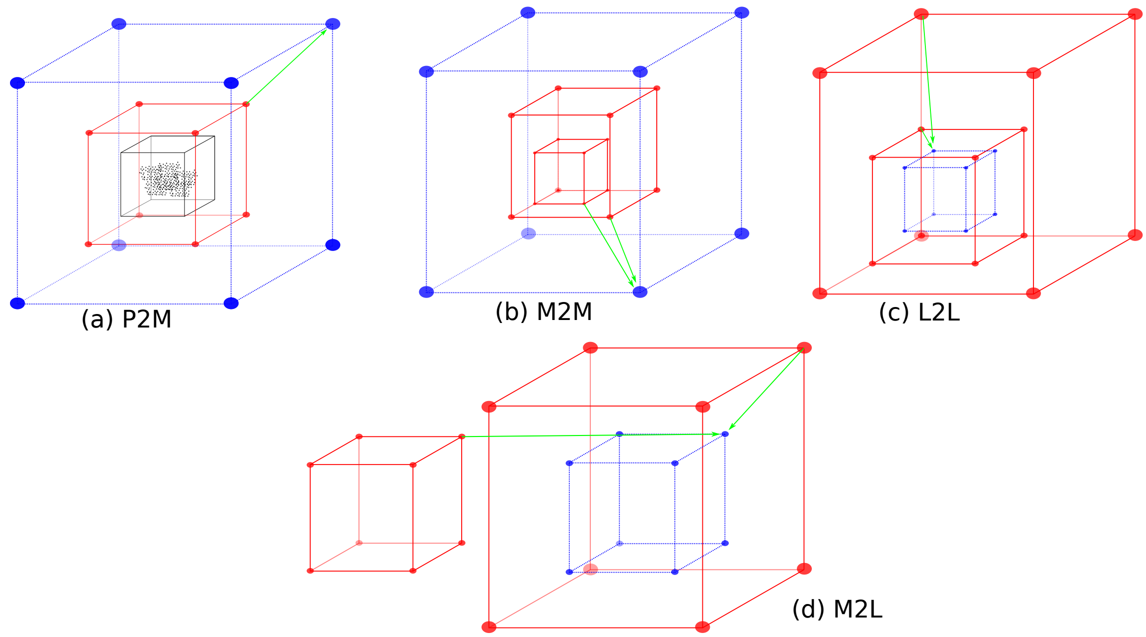

Figure 2. KIFMM operator calculations for order $p = 2$ expansions. Equivalent charges placed at quadrature points on the (red) equivalent surfaces, are matched at quadrature points on the (blue) check surfaces. Charges are plotted in black, and green arrows are used to indicate least-squares fittings.

$$M2L_l = \sum_{j=1}^{r} \sigma_j u_j v_j^T \qquad (11)$$

where the row-rank $r = \text{rank}(M2L_l) = N_c$, u_j and v_j are the left and right singular vectors respectively, and the singular values σ_j are arranged in weakly decreasing order. We can cutoff this sum at a *compression rank* $k \leq N_c$ to approximate $M2L_l$. PyExaFMM implements the *randomized SVD* of Halko et. al [6]. The algorithm can be decomposed into a series of BLAS level 2 and 3 operations, making it fast to compute.

Efficiently Representing the Tree

Particle coordinates are discretized into tree boxes via a *Morton encoding* [7]. The relative position of a box at a given level l, can be described in terms of integer displacements along each coordinate axis in the range $[0, 2^l)$, with respect to a corner of the root box chosen to be the 'origin'. The bits of these indices are interleaved as,

$$x_1 y_1 z_1 x_2 y_2 z_2 ... x_n y_n z_n | l \qquad (12)$$

where x_i, y_i and z_i are the i^{th} bit of the respective index, and the level is appended to the end. This is known as a *Morton key*. We can map between physical particle coordinates and Morton keys at a given level by scaling the physical coordinates with respect to the boxes at this level, and finding their relative displacement from the origin. Morton keys are stored as signed 64 bit integers[2], with 15 bits for the level l, and 16 bits for the index along a given axes. They allow for simple calculation of tree relationships. For example, the parent of a given box - by removing the last three bits and subtracting one from the level; its children - by appending the appropriate combinations onto the final three bits and adding one to the level; or its neighbours - which are defined by algebraic displacements of the index bits. These operations are efficiently represented by Numba, making use of only arithmetic and bit shifts.

PyExaFMM represents trees linearly, as an array of Morton keys. Furthermore, it enforces a *2:1 balance*, such that neighboring boxes differ by at most 1 level. Containers for the multipole

[2] Python doesn't support all bitwise shifts on unsigned integers.

and local expansions at each box, as well as for calculated potential and potential gradients at each target, are stored in linear arrays. They are linked to the tree via lookup tables, stored as Numba compatible dictionaries, where the Morton key can be used to look up the associated index in a given data array. Tree traversal is then a matter of computing the required key, and looking up associated data via its index. Simple array-based data structures are chosen to retain compatibility with Numba, without having to extend it by writing custom containers. Additionally, they encourage positive cache behavior at the CPU level. Furthermore, as trees are defined by particle data they can be precomputed before the FMM loop.

Precomputing FMM Operators

For boxes at a given level l, the equivalent surface for a multipole expansion can be set to be the same as the check surface for a local expansion, and the check surface for a multipole expansion can be set to be the equivalent surface for a local expansion. Figures (2b) and (2d) illustrate how this could be the case for the target boxes of each respective operator. We therefore refer to these surfaces more generally as *inner* and *outer* surfaces. The inverse of the matrix with elements calculated using the Laplace kernel[3] (2) between the inner and outer surfaces can be cached and re-used for FMM operator computations, and is scaled when applied at different tree levels. Using this inverse, we can pre-compute and cache the eight unique M2M and L2L matrices. PyExaFMM stores the computed M2M, L2L and M2L operators in a HDF5 database, HDF5 is chosen due to compatibility with Numpy arrays and data types. Operators are loaded into memory at runtime. The runtime then only involves data allocation for computed results, data organization to look up operators and expansion data and pass their references to Numba compiled functions, and the application of pre-computed FMM operators via BLAS level 2 and 3 operations.

[3]Other kernels may have different scaling relationships, and will need to be precomputed for each tree level, rather than simply being scaled.

Software Design

Numba interacts with the CPython interpreter via a Python wrapper function, which dispatches arguments to a compiled function with a matching type signature. As this occurs from within Python there is a parsing cost for copying function arguments, and unboxing Python objects into primitive types. There is an additional overhead in returning control to the Python interpreter, as the Numba compiled function has to box return values in a corresponding Python type, and handle errors. Indeed, the Assembly code of a Numba compiled function can be considerably more complex than the equivalent produced by a compiled language. In order for Numba compilation to have an effect on runtime, the cost of computation must outweigh these overheads.

We minimize interaction between the CPython interpreter and Numba compiled functions in PyExaFMM via a thin 'backend' interface, exposed using a Python dictionary, see figure (3). A compatible backend needs to implement each FMM operator. We implement a Numba backend, though it's possible to implement backends based on other technologies. Within the Numba backend, all functions are Numba compiled, and therefore we pay the interaction cost with the interpreter only once per operator call. Furthermore, we design the operators to be called as few times as possible. The P2M, P2P, L2P and M2P operators are called once, and act over all leaf boxes. During the downward pass the M2L operator is called over all boxes at a given level. The L2L and M2M operators are called for each box at a given level, however both involve just a single BLAS level 3 operation with few operands, we don't pay a significant interpreter interaction cost.

PyExaFMM's design can be seen to be 'data oriented'. The API is exposed via the 'Fmm' object (fig. 3), which can be seen to be a thin wrapper that: (a) loads the precomputed tree and operators from the HDF5 database at runtime, (b) implements the logic of the FMM loop (c) dispatches data to a Numba compiled backend.

Python's expressiveness allows PyExaFMM to be a concise library, consisting of ∼5,000 lines of code, in comparison to comparable single-node C++ implementations ExaFMM-T (∼7000 lines) [1] or TBFMM (∼20,000) [8].

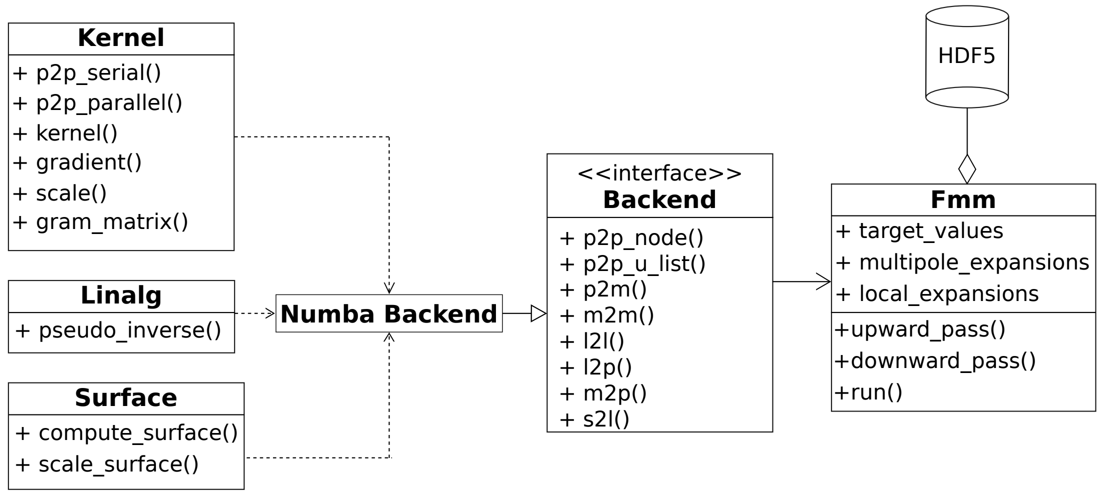

Figure 3. Simplified UML model of all PyExaFMM components. Trees and operators are precomputed and stored in the HDF5 database. Except for the 'Fmm' object which acts as the user interface, all other components are modules consisting of simple functions.

Parallelization Strategies for FMM operators

Numba implements its threading layer in an agnostic way, compatible with OpenMP and Intel TBB backends, it also provides a generic cross-platform implementation. We choose the OpenMP backend, as the size of the tasks performed by FMM operators are approximately uniform. TBB implements complex scheduling optimizations, more suited to unbalanced workloads. BLAS routines called from within threads are multithreaded by default, leading to *nested parallelism*. We avoid this oversubscription by restricting these operations to run on a single thread.

PyExaFMM implements two parallelization strategies for FMM operators. For the P2M, L2P and P2P operators we allocate aligned vectors of the sources and targets for each target box being considered at a given level, and run the kernel function, and its corresponding gradient function, in parallel over sources and targets batched by their corresponding target box. As we don't apriori know how many sources and targets a given target box will have for these operators, we have to allocate enough space to store the maximum allowable number of sources and targets, and use *index pointers* to lookup data for a given target box. Allocation cost is restricted by the constraint dictating the maximum particles per leaf node, and the size of the local and multipole expansions. For the P2P operator, cost is also restricted by the 2:1 balance condition, which limits the maximum size of the U list to 60 in three dimensions. This strategy is optimized for cache re-use due to the aligned vectors.

For the M2L and M2P operators, the maximum size of the V and W lists for all boxes at a given level make the above strategy too expensive in terms of memory to be run on most local workstations. In three dimensions, after imposing a 2:1 balance condition, a given target box has up to 189 source boxes in its V list, and 148 in its W list. For example, allocating an array to store the coordinates for source particles in double precision for boxes in the W list of a uniform tree with 5 levels, hence 32768 target leaf boxes, with at most 150 particles per leaf box, would take $\sim$ 17GB. Instead we perform a parallel loop over all target boxes at a given level. We don't optimize for CPU cache re-use as we have to look up data from an un-aligned global array. The maximum size of the X list for a given box is 19 in three dimensions, we observe empirically that this is too small to be dramatically impacted by the first parallelization strategy. Each thread, computing the S2L operation over a target box's X list, contains relatively little computation. Therefore, the S2L operator is computed in a similar way to the M2L and M2P operators.

We aimed to restrict computations within threads to BLAS operations. However, our implementation requires the computation of transfer vectors for each source box in a V list within the M2L operator. Furthermore, we needed to develop a 'hash' function for transfer vectors, to lookup the corresponding components of the SVD compressed M2L operator (11) at that level. Numba does offer an implementation of Python's native 'hash' function, however it is not optimized, as it falls out of the numerical remit of the compiler. This is an example of a potential pitfall in Numba development, where naive usage of native Python functions or objects impacts performance despite being supported. Instead we compute a checksum by projecting transfer vector components uniquely to the real integers, and concatenating them, thus using only simple bit shifts and arithmetic operations.

Benchmarks

Benchmarks are taken on an Intel i7-9750H mobile workstation, running Ubuntu 20.04.2, GCC 9.3.0 and OpenMP 4.5.

- (0) Mention machine used for computing benchmarks. Offer time benchmark for precomputations wrt to C++ (trees + operators).

- Add other benchmarks here:

- Operator precomputation benchmark

- M2L compression benchmark

- Tree precomputation benchmark

- Tree ops benchmark (child/neighbour/parent finding), compare without Numba

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

- (2) 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.

CONCLUSION

- Numba constraints software design, and has its own learning curve. Not simple to just drop in and expect performance boost.

- Performance debugging for more complex applications requires more knowledge than novice users can be expected to have.

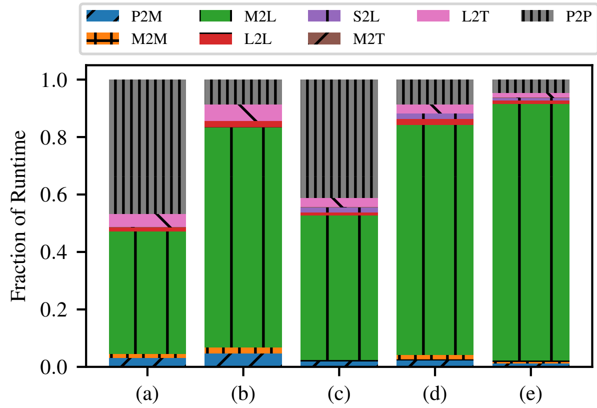

Figure 4. Foo bar

- Numba doesn't help much if there are big interpreter related costs (using large Numpy arrays) which add a large constant to runtime complexity.

- Numba is clearly insufficient for more complex HPC applications, however performance is still fairly remarkable. Not to mention other benefits of Numba (rapid prototyping with Python, cross-platform builds, GPU development etc.)

ACKNOWLEDGEMENT

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. D. Malhotra, G. Biros, D. Malhotra, and G. Biros, "PVFMM: A Parallel Kernel Independent FMM for Particle and Volume Potentials," *Commun. Comput. Phys.*, vol. 18, no. 3, pp. 808–830, 2015.
5. W. Fong and E. Darve, "The black-box fast multipole method," *Journal of Computational Physics*, vol. 228, pp. 8712–8725, dec 2009.
6. N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for

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

constructing approximate matrix decompositions," *SIAM review*, vol. 53, no. 2, pp. 217–288, 2011.

7. 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.
8. 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 PhD student in Mathematics at University College London. 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. Contact him at t.betcke@ucl.ac.uk.