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 Python programming language, which is both dynamically typed and interpreted. CPython was designed with safety and developer productivity in mind, rather than high-performance computing [HPC]. 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 multithreaded applications,

 Published by the IEEE Computer Society

and even deploy source code written in Python to GPUs.

PyExaFMM[1] is a solver for the particle fast multipole method [FMM]. 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 an excellent development framework 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 makes Numba compiled functions competitive with compiled languages such as C++ and Fortran. 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 function for these types. 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 cross-platform, heterogenous, applications using only Python source code. However, developing applications in Numba still requires interaction with the Python interpreter to allocate data before passing to optimized functions, as well as to interact with non-optimized parts of the codebase involved in data-organization or calls to incompatible libraries.

- This paper consists of: - Brief intro to FMM algorithm (¡1000 words) - Math methods, software design and parallelization strategies for achieving performance with Numba. (2500 words) - Benchmarks wrt comparable C++ module (1500 words)

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

FAST MULTIPOLE METHODS TECHNIQUES FOR ACHIEVING PERFORMANCE

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) [2].

Larger figure

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. 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.

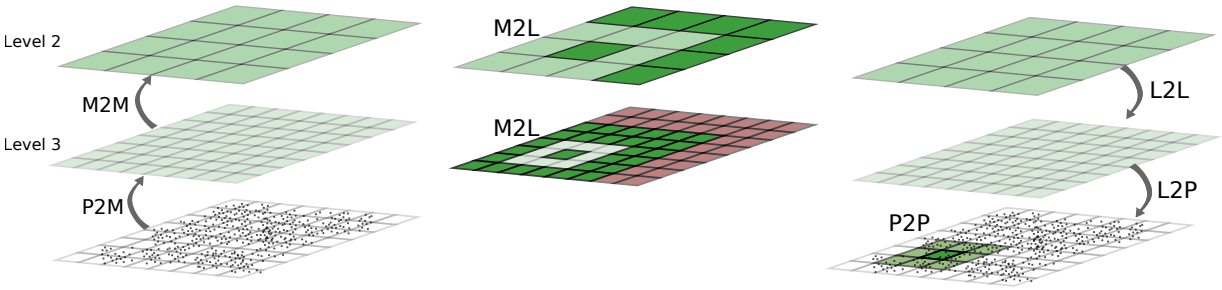

Figure 1. algorithm

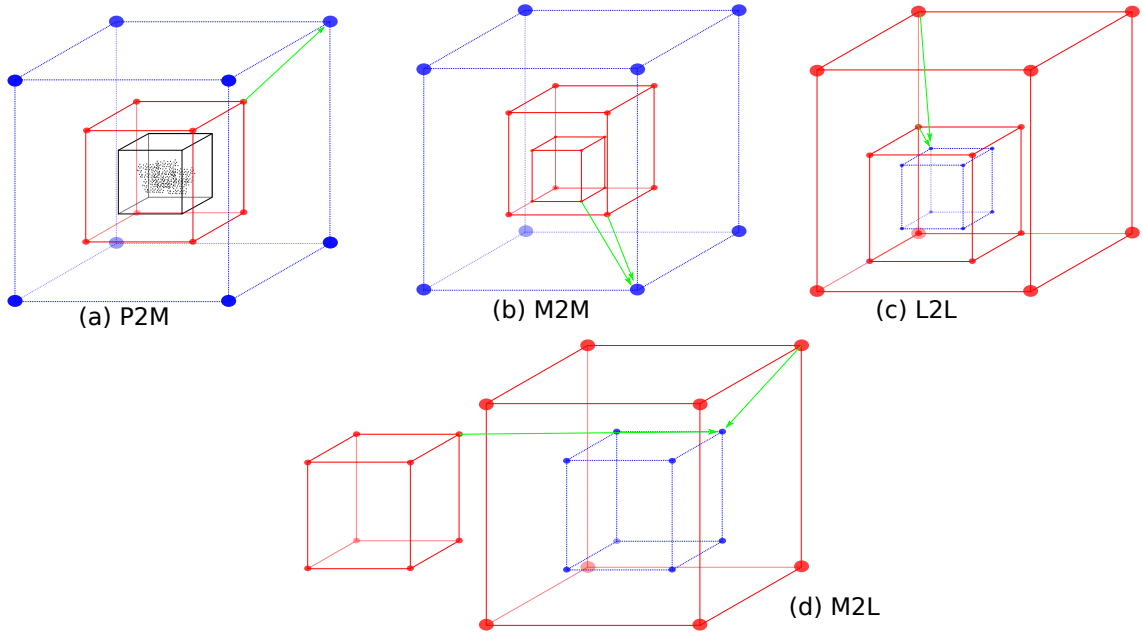

Figure 2. The operators of the KIFMM. Equivalent surfaces are shown in red, check surfaces in blue, and the charged points in black. Surfaces are plotted with 8 quadrature points, one at each vertex.

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

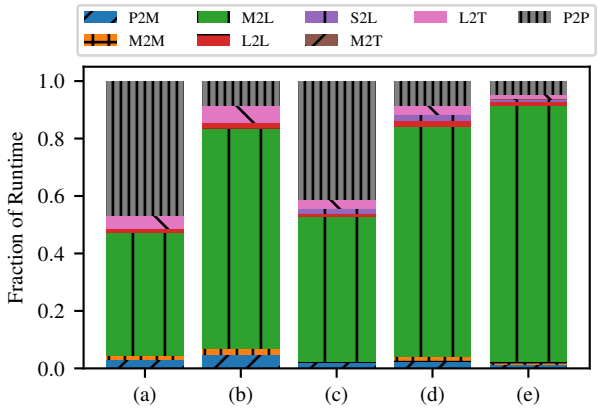

Figure 3. Foo bar

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.

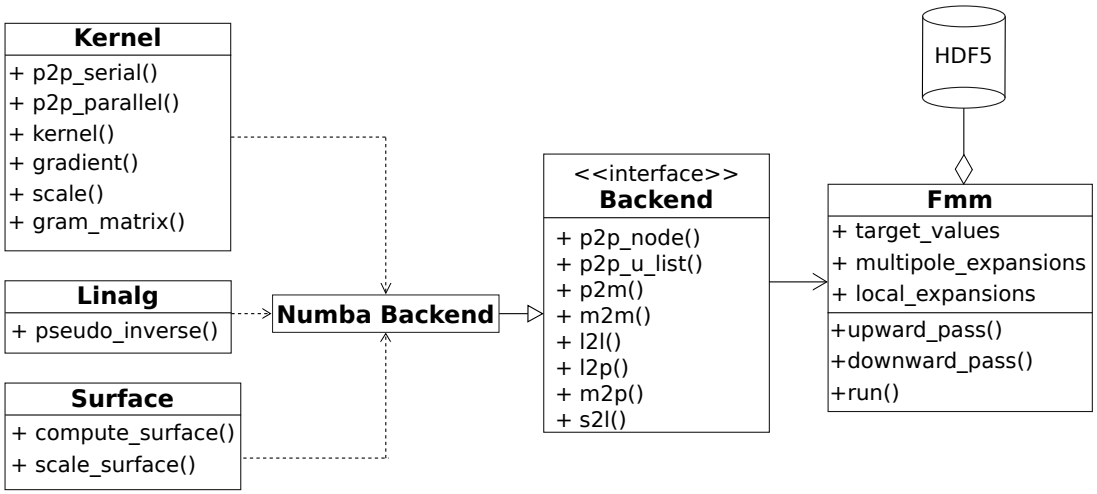

Figure 4. Simplified UML model of all PyExaFMM components. Trees and operators are precomputed and stored in the HDF5 database. Except for the 'Fmm' object which acts as the user interface, all other components are modules consisting of simple functions.