PyExaFMM: an exercise in designing high-performance software with Python and 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*—Numba is a game changing compiler for high performance computing with Python. The machine code it produces runs outside of the single-threaded Python interpreter, and can therefore fully utilize the resources of modern CPUs. This means support for parallel multithreading and auto vectorization if available, just like in a compiled language such as C++ or Fortran. Here we present our attempt to use Numba to develop a fully multithreaded implementation of the Fast Multipole Method, an algorithm which relies on a non-linear data structure and contains a significant amount of data organization. We find that designing highly performant Numba code can be as challenging as writing in a compiled language, and that there is a significant runtime cost due to the unavoidable interaction between Numba compiled code and the Python interpreter.**

■ **PYTHON**[1], is designed for memory safety and programmer productivity. Its simplicity allows Computational Scientists to spend more time exploring the science, and less time being confused by strange software quirks, memory errors, and the nightmare of incompatible dependencies, which all conspire to drain productivity in lower

[1]We use ‘Python’ to refer to CPython, the popular C language implementation of Python, which is dominant in computational science.

level languages.

The catch is that code is run through an interpreter and restricted to run on a single thread via a software construction called the Global Interpreter Lock [GIL]. However, libraries for high performance computational science have traditionally bypassed the issue of the GIL by using Python's C interface to call extensions built in C or other compiled languages which can be multithreaded or compiled to target special hardware features. Popular examples of this approach include Numpy and SciPy, which together have helped propel Python's popularity in computational science by providing high performance data structures for numerical data as well as interfaces for compiled implementations of common algorithms from numerical linear algebra, to differential equations solvers and machine learning.

As the actual number-crunching itself happens outside of the interpreter, the GIL only becomes a bottleneck to performance if a program must repeatedly pass control between the interpreter and non-Python code. This is most often an issue when an optimized compiled language implementation of your desired algorithm doesn't exist in the Python Open Source, or if it requires a lot of data organization to form the input for an optimized Numpy or SciPy code, which must unavoidably take place within the interpreter. Previously, an unlucky developer would have been forced to write a compiled implementation to tackle these issues themselves and connect it to their Python package, relegating Python's role to an interface. However, not all Computational Scientists have the necessary software skills or research interest in developing and maintaining complex codebases that couple multiple languages.

This is the context in which Numba[2] was introduced [1]. It is a compiler that specifically targets and optimizes Python code written with Numpy's n-dimensional array, or 'ndarray', data structure. Its power comes from the ability to generate multithreaded architecture optimized compiled code while *only writing Python*. The promise of Numba is the ability to develop applications with speed that can rival C++ or Fortran, while retaining the simplicity and productivity of working in Python. We put this promise to the test by developing PyExaFMM[3], an implementation of the Kernel-Independent Particle Fast Multipole Method [FMM] [2], [3], in three dimensions. Efficient implementations of this algorithm are complicated by its reliance on a tree data structure and a series of operations which each require significant data organization and careful memory allocation. These features made PyExaFMM an excellent test case to see whether Numba could free us as Computational Scientists from the complexities of compiled languages.

We begin with an overview of Numba's design and its major pitfalls when. After briefly introducing the data structures and computations involved in the FMM, we provide an overview of how we implemented our software's data structures, algorithms and application programming interface [API] to optimally use Numba. With benchmarks to demonstrate the impact of Numba on vanilla Python implementations of the FMM's operators, and the cost of passing between the Python interpreter and Numba compiled code.

NUMBA

Numba targets code written using ndarrays, which are homogeneously typed and stored contiguously in memory. This means that adjacent elements are stored in adjacent memory locations. When loading data from a given memory location, modern CPUs cache data located at adjacent locations, therefore algorithms which iterate sequentially over arrays benefit from lower memory loading latencies as the CPU's cache is likely to contain the next element being operated on. This is known as *cache locality*. This is in contrast to Python lists, where subsequent elements need not be stored in adjacent memory locations and are found by following trails of pointers, this phenomena is known as *indirection*, which doesn't take advantage of CPU caching behavior at all. Ndarrays have fields describing their data's dimensionality, type, and layout, which Numba uses to generate machine code without any indirection. Instead it directly references the memory locations being referred to by the Python code at runtime. This allows Numba to index

[2]We use the latest stable release of Numba, 0.53.1 throughout this study.

[3]https://github.com/exafmm/pyexafmm

into ndarrays with performance that can match compiled languages.

Numba is built with LLVM [4], a framework for building custom compilers, that provides an API to generate machine code for different hardware architectures such as CPUs and GPUs. LLVM is also able to analyze the code for hardware level optimizations, such as auto vectorization, and automatically apply them if they are available on the target hardware.

From a programmer perspective using Numba naively doesn't involve a significant rewrite of code. Python functions are marked for compilation with a special decorator (see listings (1), (2) and (3) for example syntax). Figure (1) illustrates the program execution path when a Numba decorated function is called from the Python interpreter.

We see that Numba doesn't replace the Python interpreter. If a marked function is called at runtime, program execution is handed to Numba's runtime which compiles the function on the fly, with a type signature matching the input arguments. This is the origin of the term 'just in time' [JIT] to describe such compilers. The Numba runtime interacts with the Python interpreter dynamically, and control over program execution passed back and forth between the two. There is a cost to this interaction from having to 'unbox' Python objects into types compatible with the compiled machine code, and 'box' the outputs of the compiled functions back into Python compatible objects. This process doesn't involve reallocating memory, however pointers to memory locations have to be converted and placed in a type compatible with either Numba compiled code or Python. The cost of this can be seen in Table ([REFERENCE TO BOX BENCHMARK TABLE]), where we illustrate the cost of boxing and unboxing functions of with increasing numbers of array arguments. Table ([REFERENCE BOX BENCHMARK FOR DIFFERENT TYPES]) illustrates the cost of boxing different Python types into Numba compatible types. Software written with Numba has to account for this cost, and minimize the interactions between Python and Numba.

Listing 1. An example of using Numba in a Python function operating on ndarrays.

```python
import numba
import numpy as np

# optionally the decorator can take
# the option nopython=True, which
# disallows Numba from running in
# object mode
@numba.jit
def loop_fusion(a):
    """
    An example of loop fusion, an
    optimization that Numba is able
    to perform on a user's behalf.
    When it recognizes that they are
    operating similarly on a single
    data structure
    """
    for i in range(10):
        a[i] += 1

    for i in range(10):
        a[i] *= 5

    return a
```

PITFALLS OF NUMBA

Since its first release Numba has been extended to compile most functionality from the Numpy library, as well as the majority of Python's basic features and standard library modules[4]. However, if Numba isn't able to find a suitable Numba type for each Python type in a decorated function, or it sees a Python feature it doesn't yet support, it runs in 'object' mode, handling all unknowns as generic Python objects. Crucially Numba does this without reporting it to the user, with object mode often no faster than ordinary vanilla Python. This puts the burden on the programmer to understand when and where Numba works, and to achieve performance a programmer must be careful to limit the surface of interaction between Python code and Numba code. Therefore, as Numba influences the way in which you write Python in practice, it behaves more like a framework rather than just a compiler.

Additionally, not every supported feature behaves in a way a programmer would expect, which can have impacts on program design. An example of this arises when using Python dictionaries, which are partially supported by

[4]A full list of supported features for the current release can be found at: https://numba.pydata.org/numba-doc/dev/reference/pysupported.html

Numba. As they are untyped, and can have any Python objects as members, they don't neatly fit into a Numba compatible type. Programmers can declare a Numba compatible 'typed dictionary', where the keys and values are constrained to Numba compatible types, and pass it to a Numba decorated function at low cost. However, using a Numba dictionary from the Python interpreter is *always slower* than an ordinary Python dictionary due to the (un)boxing cost when getting and setting any item. A natural use case for a dictionary is as a place to store results during an algorithm. To avoid the (un)boxing cost in algorithms that involve multiple steps that operate on shared data, programmers are forced write a nested functions as in listing (2) that are called as minimally as possible. Nested functions have the tendency to grow long when writing performant Numba code, in order to minimize the interaction surface between Numba and Python. When pursuing performance with Numba code can end up looking quite un-Pythonic. Numba encourages fewer user created objects, performance critical sections written in terms of loops over simple array based data structures - reminiscent of C, and long nested functions which are more difficult to unit test.

Though Numba is advertised as an easy way of injecting performance into your program via a simple decorator it has its own learning curve. Achieving performance requires a programmer to be somewhat familiar with the internals of its implementation (fig. 1), and potentially have to radically change the design of their algorithms and data structures to avoid back and forth between the Numba runtime and the Python interpreter.

Listing 2. An example of using typed dictionaries in Numba.

```
import numpy as np
import numba
import numba.core
import numba.typed


# Dictionary instance created at runtime
results = numba.typed.Dict.empty(
    key_type=numba.core.types.unicode_type,
    value_type=numba.core.types.float64[:]
)

@numba.njit
def nested(arg, results):
    # This function expects a
    # typed dictionary as an
    # argument to store results

    def step1(a):
        # first step of algorithm
        # store results in dictionary
        ...

    def step2(a):
        # second step of algorithm
        # store results in dictionary
        ...

    # Interpreter doesn't know about
    # the changes in the dictionary
    # until Numba runtime hands back
    # control
    return results
```

THE FAST MULTIPOLE METHOD

The particle FMM is an approximation algorithm for N-body problems, in which N source particles interact with N target particles [3]. Consider the calculation of electrostatic potentials in 3D, which we use as our reference problem. A set of N charged particles at positions x_i. The potential, ϕ_j, at a given target particle at x_j due to all other particles, excluding self interaction, can be written as.

$$\phi_j = \sum_{i=1, i\neq j}^{N} \frac{q_i}{4\pi |x_i - x_j|} \tag{1}$$

where $\frac{1}{4\pi|x_i - x_j|}$ is called the kernel, or the Green's function, and q_i is the charge at x_i. The naive computation over all particles is $O(N^2)$, however the FMM compresses groups of interactions far away from a given particle using *expansions*, reducing the overall complexity to $O(N)$. Expansions are ways of describing the charge contained within subregions of the octree, and can be truncated with to a desired accuracy, described by a parameter, p, called the expansion order. The value of p is equal to the number of digits of precision of the final solution. Problems with this structure appear with such frequency in science and engineering that the FMM has been described as one of the ten most important algorithms of the twentieth century [5].

The algorithm relies on a hierarchical octree

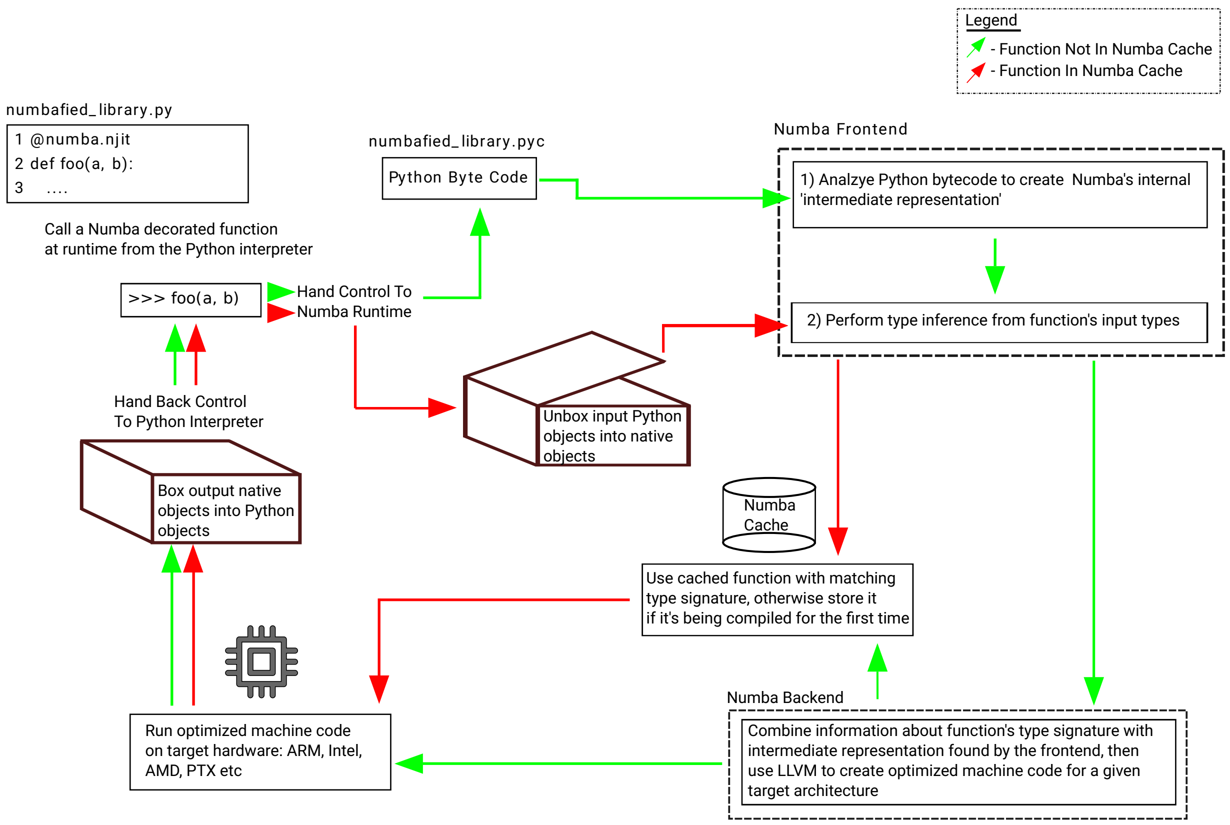
Legend
- Function Not In Numba Cache
- Function In Numba Cache
numbafied_library.py
1 @numba.njit
2 def foo(a, b):
3 ....
numbafied_library.pyc
Python Byte Code
Numba Frontend
1) Analzye Python bytecode to create Numba's internal 'intermediate representation'
2) Perform type inference from function's input types
Call a Numba decorated function at runtime from the Python interpreter
>>> foo(a, b)
Hand Control To Numba Runtime
Hand Back Control To Python Interpreter
Unbox input Python objects into native objects
Box output native objects into Python objects
Numba Cache
Use cached function with matching type signature, otherwise store it if it's being compiled for the first time
Numba Backend
Combine information about function's type signature with intermediate representation found by the frontend, then use LLVM to create optimized machine code for a given target architecture
Run optimized machine code on target hardware: ARM, Intel, AMD, PTX etc

Figure 1. Simplified execution path when calling a Numba compiled function from the Python interpreter. The green path is only taken if the function hasn't been called before. The red path is taken if a compiled version with the correct type signature already exists in the Numba cache.

data structure to discretize the problem domain in 3D (see fig. 1 [6]) and consists of eight operators: P2M, P2L, M2M, M2L, L2L, L2P, M2P and the 'near field', applied *once* to each applicable node over the course of two consecutive traversals of the octree (bottom-up and then top-down). An operator is read as 'X to Y', where 'P' stands for particle(s), 'M' for *multipole expansion* and 'L' for *local expansion*. The direct calculation of (1) is referred to as the P2P operator, and is used as a subroutine during the calculation of the other operators. The Kernel Independent FMM [KIFMM] [2], implemented by PyExaFMM, is a re-formulation of the FMM with a structure that favours parallelization. Indeed all of the operators can be decomposed into matrix-vector products, or multithreaded implementations of (1), which are easy to optimize for modern hardware architectures, and fit well with Numba's programming framework. We defer to the FMM literature for a more detailed discussion on the mathematical significance of these operators [2], [3].

COMPUTATIONAL STRUCTURE OF FMM OPERATORS

The computational complexities of KIFMM operators are defined by a user specified n_{crit}, which is the maximum allowed number of particles in a leaf node, n_e and n_c which are the numbers of quadrature points on the *check* and *equivalent* surfaces respectively (see sec. 3 [2]). The parameters n_e and n_c are quadratically related to the expansion order, i.e. $p \sim n_e^2$. Typical values for n_{crit} used are ~ 100. Notice the depth of the octree is defined n_{crit}, and hence by the particle distribution.

The near field, P2M, P2L, M2P, and L2P only operate on leaf nodes. The M2L operates during the top-down traversal from level two to the leaf level, on all nodes at a given level during each step. The M2M is applied to each node during the bottom-up traversal, and the L2L is applied to each node during the top-down traversal.

All operators, except the M2L M2M and L2L, rely on the P2P. The inputs for the P2P are vectors for the source and target positions, and the source charges or expansion coefficients (which approximate charges). The output is a vector of potentials.

The inputs to the M2L, M2P, P2L and near field operators are defined by 'interaction lists', called the V, W, X and U lists respectively. These interaction lists define the nodes a target node interacts with when an operator is applied to it. We can restrict the size of these interaction lists by demanding that neighboring nodes at the leaf level are at most twice as large as each other [6]. Using this 'balance condition', the V, X, W and U lists contains at most 189, 19, 148 and 60 nodes in three dimensions, respectively.

The near field operator applies the P2P between the charges contained in the target and the source particles of nodes in its U list, in $O(60 \cdot n_{crit}^2)$. The M2P applies the P2P between multipole expansion coefficients of source nodes in the target's W list and the charges it contains in $O(148 \cdot n_e \cdot n_{crit})$. Similarly, the L2P applies the P2P between a target's own local expansion coefficients and the charges it contains in $O(n_e \cdot n_{crit})$.

The P2L, P2M and M2L involve creating local and multipole expansions, and rely on a matrix vector product related to the number of source nodes being compressed, which for the P2L and M2L are defined by the size of the target node's interaction lists. These matrix vector products have complexities of the form $O(k \cdot n_e^2)$ where $k = |X| = 19$ for the P2L, $k = |V| = 189$ for the M2L, and $k = 1$ for the P2M. Additionally, the P2L and P2M have to calculate 'check potentials' (see sec. 3 [2]) which require $O(19 \cdot n_{crit} \cdot n_c)$ and $O(n_{crit} \cdot n_c)$ calculations respectively. The M2M and L2L operators both involve translating expansions between nodes their eight children, and rely on a matrix vector product of $O(n_e^2)$.

The structure of the FMM's operators expose natural parallelism. The P2P is embarrassingly parallel over each target. As are the M2L, M2P, P2L and near field operators over their interaction lists. The near field, L2P, M2P, P2L and P2M operators are also embarrassingly parallel over the leaf nodes, as are the M2L, M2M and L2L over the nodes at a given level.

DATA ORIENTED DESIGN

Data oriented design is about writing code using data structures with simple memory layouts, such as arrays, in order to optimally take advantage of modern hardware features, such as cache-locality, and are easy to parallelize. This contrasts with object oriented design, where although code is organized around data, the focus is on user created objects where the object's memory layout is hidden behind abstraction, making it harder to write code to take advantage of cache locality. Numba's focus on ndarrays strongly encourages data oriented design principles, which are reflected in the design of PyExaFMM's octrees as well as its API.

Octrees can either be 'pointer based' [7], or 'linear' [6]. A pointer based octree, as in uses objects to represent each node, with fields for a unique id, contained particles, associated expansion coefficients, potentials, and pointers to their parent and sibling nodes. This makes searching for neighbors and siblings easy, as one has to just follow pointers. The linear octree implemented by PyExaFMM represents nodes by a unique id stored in a 1D vector, all other data such as expansion coefficients, particle data, and calculated potentials, are also stored in 1D vectors. Data is looked up by creating indices to tie a node's unique id to the associated data. This is an example of how Numba can affect design decisions, and may make software more complex, despite the data structure being simpler.

Figure (2) illustrates PyExaFMM's design. There is only a single object, 'Fmm', which acts as the API from Python. It initializes ndarrays for expansion coefficients and calculated potentials, and its methods interface with Numba compiled functions for the FMM operators and their associated data manipulation functions.

MULTITHREADING IN NUMBA

Numba enables multithreading via a simple parallel for loop syntax (see listing (3)) reminiscent of OpenMP. Internally Numba can use either OpenMP or Intel TBB to generate multithreaded code. We choose OpenMP for PyExaFMM, as it's more suited to functions in which each thread has an approximately similar workload. The threading

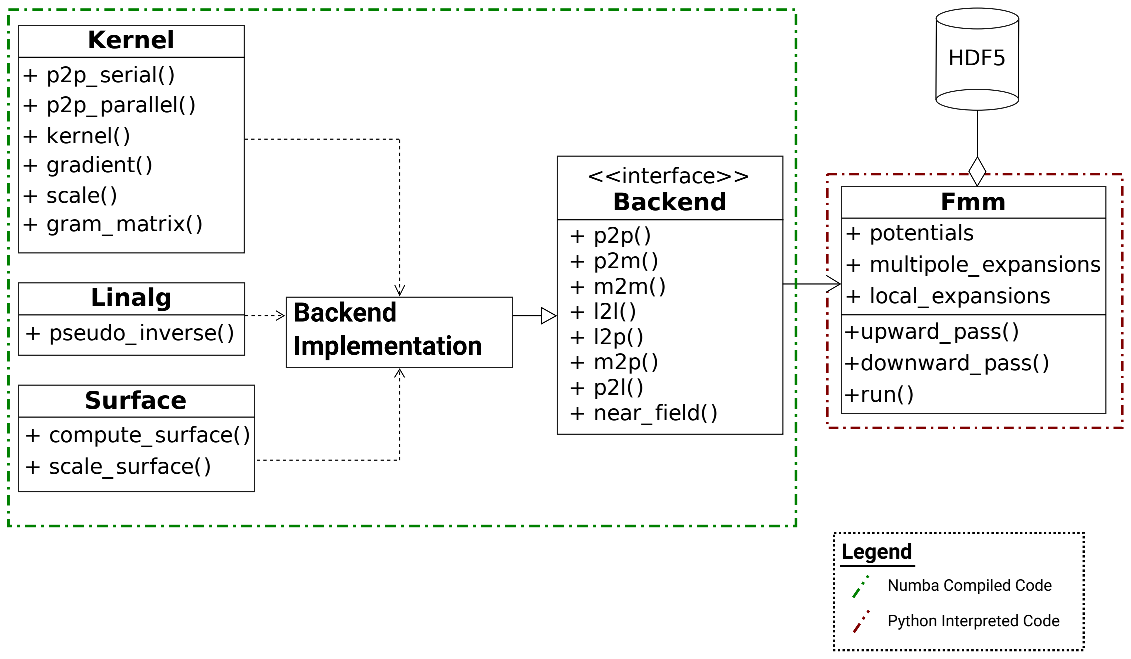

Figure 2. Simplified UML model of all PyExaFMM components. Trees and other precomputed quantities are stored in a HDF5 database. The 'Fmm' object acts as the user interface, all other components are modules consisting of methods on operating on arrays.

library can be set via the `NUMBA_THREADING_LAYER` environment variable.

Numerical libraries such as Numpy and SciPy implement many of their mathematical operations using multithreaded compiled libraries internally, such as OpenBLAS or IntelMKL. Numba compiled versions of these operations retains this internal multithreading. But when combined with a multithreaded region declared with Numba, as in listing (3), this leads to *nested parallelism*. Where a parallel region calls a function with another parallel region inside it. Threads created by the two functions are not coordinated, and this leads to *oversubscription*, where there the number of active threads exceeds the CPU's hardware capacity. Many threads are left idle while the CPU is forced to jump between threads operating on different data. This leads to broken cache locality and load imbalances where some threads are stuck waiting for others to finish [8]. We avoid this in PyExaFMM by explicitly setting Numpy operations to be single threaded, via the environment variable `OMP_NUM_THREADS=1`, before starting our program. This ensures that the only threads created are those explicitly declared using Numba.

Listing 3. An example of parallel multithreading.
```
import numba
import numpy as np

@numba.njit(cache=True, parallel=True)
def multithreading(a):
    # consider an 'a' as an nxn matrix
    # This is a situation that can lead
    # to thread oversubscription unless
    # Numpy is configured to run single
    # threaded.
    for _ in numba.prange(10):
        a @ a
```

PARALLELIZATION STRATEGIES FOR FMM OPERATORS

The P2P, P2M, P2L, M2P, L2P and near field operators are parallelized over their targets, the leaf nodes, and all rely on the P2P. For the L2P, M2P and near field operators we encourage cache locality for the P2P step, and keep the data structures passed to Numba as simple as possible, by allocating 1D vectors for the source positions, target positions and the source charges (or expansion coefficients - depending on the operator), such that all the data required to apply an operator to single target node is next to each other in memory in each vector. By storing a vector of 'index pointers', that bookend the data corresponding to each target in these 1D vectors, we can form parallel for loops over each target to compute the P2P that encourages cache-locality in the CPU. In order to to this, we have to first iterate through the target nodes and their interaction lists if relevant, and lookup their associated data to fill the vectors cache local vectors. Table [TABLE CONTAINING EXPERIMENT SHOWING THE EFFECT CACHE LOCALITY CAN HAVE].

The L2P and P2M operators only use each target node's associated data, whereas for the near field operator the source nodes are from its U list, the size of which is a priori unknown. Therefore we allocate enough memory to store up to 60 nodes worth of source charges, and source positions for each target, corresponding to the maximum size of the U list. This strategy is too expensive in terms of memory to repeat for the M2L and M2P operators, as their interaction lists can potentially be much larger, at 189 and 148 nodes respectively. For example, allocating an array large enough to store just potential coordinates for source particles in double precision for the M2P operator with $|W| = 148$ with an $n_crit = 150$ requires $\sim$ 17GB. For these operators, we simply perform a parallel loop over the target nodes at each given level for the M2L, and over the leaf nodes for the M2P, looking up the relevant data from the linear tree. We also don't implement the cache-optimal strategy for the P2L operator, the interaction list is at most 19 nodes, and we found in practice that it did not benefit from it significantly. The M2M and L2L operators are parallelized over all nodes at each given level. The the matrices involved in these operators can be precomputed and scaled at each level [7], therefore their application is highly optimized.

Multithreading in this way means that we call the P2P, P2M, M2P, L2P and near field operators just once during the algorithm, the M2L is called $d - 2$ times, and the M2M and L2L are called d times where d is the depth of the octree, which is the minimum number of times we can pass control between Python and Numba for this algorithm.

CONCLUSION

We've shown how to use Numba to create a performant application for an algorithm with a complex data structure. We've shown where Numba can constrain software design, and that developers must be careful in order to experience a performance benefit. Achieving performance for complex implementations requires more software knowledge than non-software specialists can be expected to have, and Numba can be seen to have its own learning curve. Despite this, Numba is a remarkable tool, allowing the development of fast, heterogenous, cross-platform numerical applications, all from Python. Projects which favour hackability or have a small number of performance bottlenecks would benefit the most from a Numba implementation.

ACKNOWLEDGMENT

SK is sported by EPSRC Studentship 2417009.

REFERENCES

1. 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.
2. 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.
3. L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, no. 2, pp. 325–348, 1987.
4. C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis amp; transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, 2004.
5. B. A. Cipra, "The Best of the 20th Century: Editors Name Top 10 Algorithms," *SIAM News*, vol. 33, no. 4, 2000.
6. 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.
7. 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.

8. A. Malakhov, “Composable Multi-Threading for Python Libraries,” *Proceedings of the 15th Python in Science Conference*, pp. 15–19, 2016.

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.