Department: Head
Editor: Name, xxxx@email

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. In this article we document our experience with Numba for developing PyExaFMM, a fully multithreaded Numba implementation of the Fast Multipole Method, an algorithm with a non-linear data structure and a significant amount of data organization. We find that designing highly performant Numba code for complex algorithms can be as challenging as writing in a compiled language.**

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

The catch is that code is run through an

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

 Published by the IEEE Computer Society

interpreter and restricted to a single thread via a software construction called the Global Interpreter Lock [GIL]. 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 have together helped to 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 equation solvers and machine learning.

As the actual number crunching 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 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, which are homogenously typed containers stored contiguously in memory. 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, an implementation of the three-dimensional Kernel Independent Particle Fast Multipole Method [FMM] [2], [3]. PyExaFMM can be found on GitHub[2], as can jupyter notebooks containing the experiments presented in this paper[3]. 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. After 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.

NUMBA

Numba is a compiler built with LLVM, a framework for building custom compilers, to target a subset of Python code that uses ndarrays. LLVM provides an API for generating machine code for different hardware architectures such as CPUs and GPUs and is also able to analyze code for hardware level optimizations such as auto vectorization, automatically applying them if they are available on the target hardware [4]. Additionally, LLVM generated code may be multithreaded - bypassing the issue of the GIL. Furthermore Numba is able to use the metadata provided by ndarrays describing their dimensionality, type and layout to generate code that takes advantage of the hierarchical caches available in modern CPUs [1]. Altogether, this allows code generated by Numba to run significantly faster than ordinary Python code, and often be competitive with code generated from compiled languages such as C++ or Fortran.

From a programmer perspective using Numba, at least naively, doesn't involve a significant rewrite. Python functions are simply marked for compilation with a special decorator, see listings (1), (2) and (3) for example syntax. This encapsulates the appeal of Numba. The ability to generate

[2]https://github.com/exafmm/pyexafmm
[3]https://github.com/betckegroup/pyexafmm-cise

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

```
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
```

high-performance code for different hardware targets from Python, and letting Numba worry about how to perform optimizations, would allow for significantly faster workflows than possible with a compiled language.

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 is 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 re-allocating memory, however pointers to memory locations have to be converted and placed in a type compatible with either Numba compiled code or Python.

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 unknown quantities as generic Python objects. To ensure a seamless experience Numba does this without reporting it to the user, unless explicitly marked to run in 'no Python' mode (see listing (2) and (3) for example syntax). However, object mode is often no faster than vanilla Python, putting the burden on the programmer to understand when and where Numba works. As Numba influences the way Python is written it's more akin to a programming framework rather than just a compiler.

An example of Numba's framework-like behavior arises when implementing algorithms that share data, and have multiple logical steps as in listing (2). This listing shows three implementations of the same logic, a function that generates a random matrix $A \in \mathbb{R}^{100\times100}$ and multiplies it with itself, and also writes an input vector $v \in \mathbb{R}^{100}$ to a Numba dictionary. Data of this size is chosen to reflect the typical amount of computation in a task parallelized by PyExaFMM. The implementations `algorithm1` and `algorithm2` aren't distinguished by the Numba compiler, and both pay a cost to call subroutines defined outside of their function body. However, `algorithm1` pays a small additional (un)boxing cost in order manipulate a globally defined Numba compatible dictionary, in comparison to a locally defined one in `algorithm2`. The *nested* function in `algorithm3` differs from the other two implementation, by defining its sub-routines within its function body, rather than calling externally defined functions. This is an example of an *inlining* optimization being taken by LLVM.

The runtimes of all three implementations are shown in table (1) for three contrasting problem sizes, and shows how inlining can have a significant impact on runtime for this algorithm.

[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

Table 1. Testing the effect of inlining and (un)boxing with dense matrix vector products in double precision using implementations from listing (2).

Algorithm	Matrix Dimension	Time (μs)
1	$\mathbb{R}^{1\times1}$	1.74 ± 0.01
1	$\mathbb{R}^{100\times100}$	308 ± 1
1	$\mathbb{R}^{1000\times1000}$	27100 ± 200
2	$\mathbb{R}^{1\times1}$	2.94 ± 0.01
2	$\mathbb{R}^{100\times100}$	306 ± 1
2	$\mathbb{R}^{1000\times1000}$	27100 ± 200
3	$\mathbb{R}^{1\times1}$	2.61 ± 0.01
3	$\mathbb{R}^{100\times100}$	2.64 ± 0.07
3	$\mathbb{R}^{1000\times1000}$	2.84 ± 0.07

Algorithm 1 and 2 are not inlined, in contrast to Algorithm 3. Algorithm 1 additionally pays a boxing and unboxing cost to pass between Numba and Python, as it operates on a result dictionary defined in the Python. Algorithm 2 and 3 only pay the boxing cost to pack a result obtained in Numba into a Python type as they creates a result dictionary within Numba. All experiments were taken on an AMD Ryzen Threadripper 3970X 32-Core processor running Python 3.8.5 and Numba 0.53.0

This example is designed to illustrate how small changes to writing style can impact the performance of an algorithm written with Numba. We emphasize that the speedup obtained from inlining is dependent on the size of the data being operated on as well as the program logic. Other factors such as memory latency for large data, or the passing of execution control between Python and Numba, with small data are may become more significant. Indeed the experiment with $A \in \mathbb{R}^{1000\times1000}$ and $v \in \mathbb{R}^{1000}$, we observe that inlining is still the dominant factor in performance difference and is even more prominent than with the smaller dataset. With $A \in \mathbb{R}^{1\times1}$ and $v \in \mathbb{R}^{1}$, we see that the instantiation of the result dictionary from within a Numba function is now a significant part of total runtime.

Nested functions have the tendency to grow long in performant Numba code, in order to minimize the number of interactions between Numba and Python. However this makes them more difficult to unit test. Indeed, performant Numba code can look decidedly un-Pythonic. Numba encourages fewer user created objects, performance critical sections written in terms of loops over simple array based data structures, and potentially long nested functions.

Furthermore, not every supported feature from Python behaves in a way an ordinary Python programmer would expect, which has an impact on program design. An example of this arises when using Python dictionaries, which are central to Python, but are only partially supported by 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.

Therefore, though Numba is advertised as an easy way of injecting performance into your program via a simple decorator, it can be seen to have its own learning curve. Achieving performance requires a programmer to be familiar with the internals of its implementation and potential discrepancies that arise when translating between Python and the LLVM generated code, which may lead to significant alterations in the design of algorithms and data structures.

Listing 2. Three ways of writing a trivial algorithm that passes around a vector, while performing some computations.

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

# Initialize in Python interpreter
data = numba.typed.Dict.empty(
    key_type=numba.core.types.unicode_type,
    value_type=numba.core.types.float64[:]
)

data['v'] = np.ones(100)

# Functions marked with 'njit' rather than
# 'jit' decorator, to force Numba to run in
# no Python mode, dissallowing the compilation
# of Python code it is not specialized for.

# Subroutine 1
@numba.njit
def step_1(data):
    # An example allocation & calculation
    a = np.random.rand(100, 100)
    b = a @ a
    data['step_1'] = data['v']

# Subroutine 2
@numba.njit
def step_2(data):
    # An example allocation & calculation
    A = np.random.rand(100, 100)
    B = A @ A
```

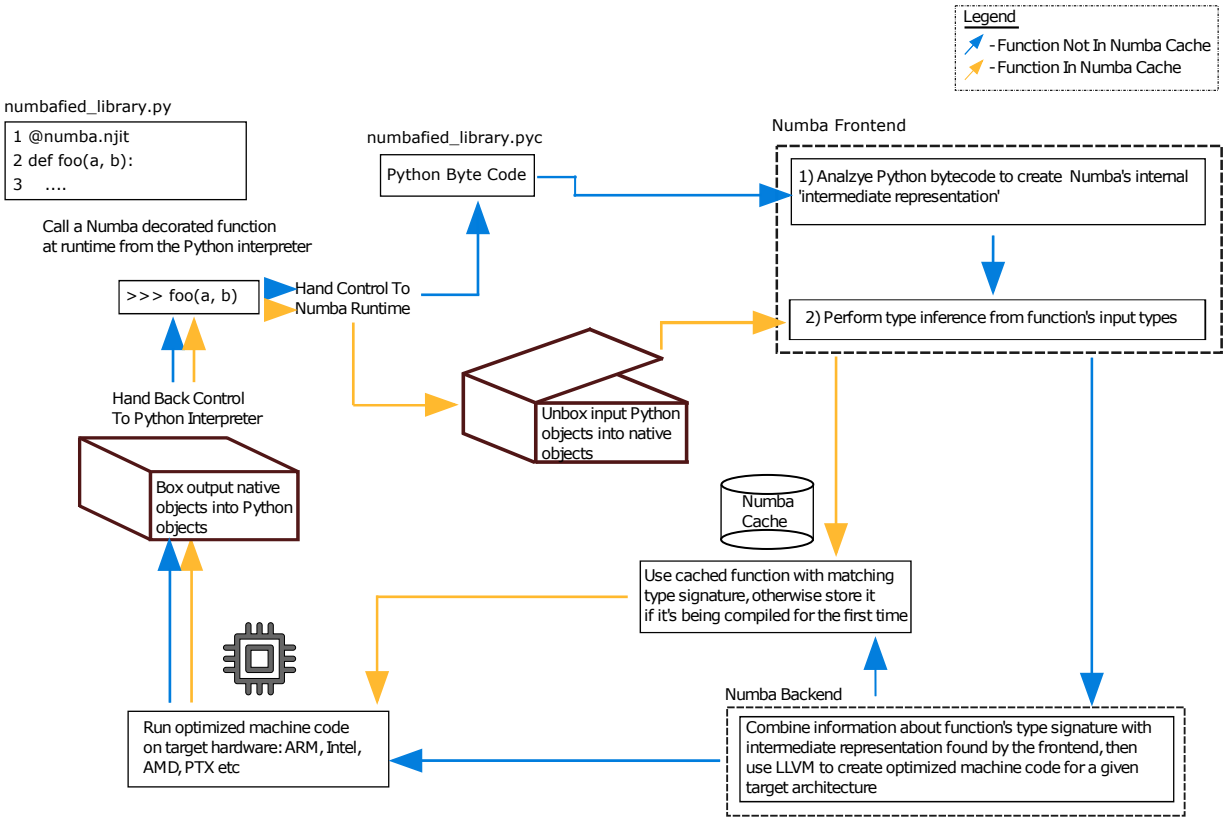

Figure 1. Simplified execution path when calling a Numba compiled function from the Python interpreter. The blue path is only taken if the function hasn't been called before. The orange path is taken if a compiled version with the correct type signature already exists in the Numba cache.

```
    data['step_2'] =  data['step_1']

@numba.njit
def algorithm1(data):
    # Pays the un(boxing) cost to exchange
    # data with interpreter
    step_1(data); step_2(data)

@numba.njit
def algorithm2():
    # Only pays the boxing cost to return a
    # Python type.
    data = dict()
    data['v'] = np.ones(100)
    step_1(data); step_2(data)
    return data

@numba.njit
def algorithm3():
    # Numba interprets subroutines as a
    # part of a **single** function body.
    data = dict()
    data['v'] = np.ones(100)

    def step_1(data):
        A = np.random.rand(100, 100)
        B = A @ A
        data['step_1'] = data['v']

    def step_2(data):
        A = np.random.rand(100, 100)
        B = A @ A
        data['step_2'] =  data['step_1']

    step_1(data); step_2(data)
    return data
```

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 approximate charge contained within subregions of the octree, and can be truncated with to a desired accuracy, defined by a parameter, p, called the expansion order. 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 an octree data structure to discretize the problem domain in 3D. Octrees place the region of interest in a cube, or 'root node', and subdivide it into 8 equal parts. These 'child nodes' turn are recursively subdivided until a user defined threshold is reached (see fig. 1 [6]). The FMM 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). The operators define interactions between a given 'target' node, and potentially multiple 'source' nodes from the octree. They are 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. $n_e = 6(p-1)^2 + 2$ [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 operate independently over the leaf nodes. The M2L and L2L operate independently on all nodes at a given level, from level 2 to the leaf level, during the top-down traversal. The M2M is applied to each node during the bottom-up 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; 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 in 3D contain at most 189, 19, 148 and 60 nodes, 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 internally 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 in-

volve 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 OF PYEXAFMM

Data oriented design is about writing code that operates on data structures with simple memory layouts, such as arrays, in order to optimally take advantage of modern hardware features. The idea being that it is easier for programmers to optimize for cache locality and parallelization if the data structures are easier to map to the hardware. This contrasts with object oriented design, where although code is organized around data the focus is on user created types or 'objects', where memory layout is obfuscated by the potential complexity of an object, which can contain multiple attributes of different types. This makes it harder to write code that takes 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 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 make software more complex, despite the data structures being simpler.

Figure (2) illustrates PyExaFMM's design. There is only a single Python object, 'Fmm', which acts as the API. 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. When sharing data we prefer nested functions, however we keep the operator implementations separate from each other, which allows us to unit test them individually. This means that we must have at least one interaction between Numba and the Python interpreter to call the near field, P2M, L2P, M2P and P2L operators, $d-2$ interactions to call the M2L and L2L operators, and d interactions for the M2M operator where d is the depth of the octree. The most performant implementation would be a single Numba routine that interacts with Python just once, however this would sacrifice other principles of clean software engineering such as modularity, and unit testing.

This structure has strong parallels with software designs that arise from traditional methods of achieving performance with Python by interfacing with a compiled language such as C or Fortran. The benefit of writing in Numba is that we can continue to write in Python. Though as seen above, performant Numba code may only be superficially Pythonic through its shared syntax.

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 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. This leads to *nested parallelism* when combined with a multithreaded region declared with Numba, as in listing (3). This is where a parallel region calls a function with another parallel region inside it. Threads created by the two functions are not coordinated

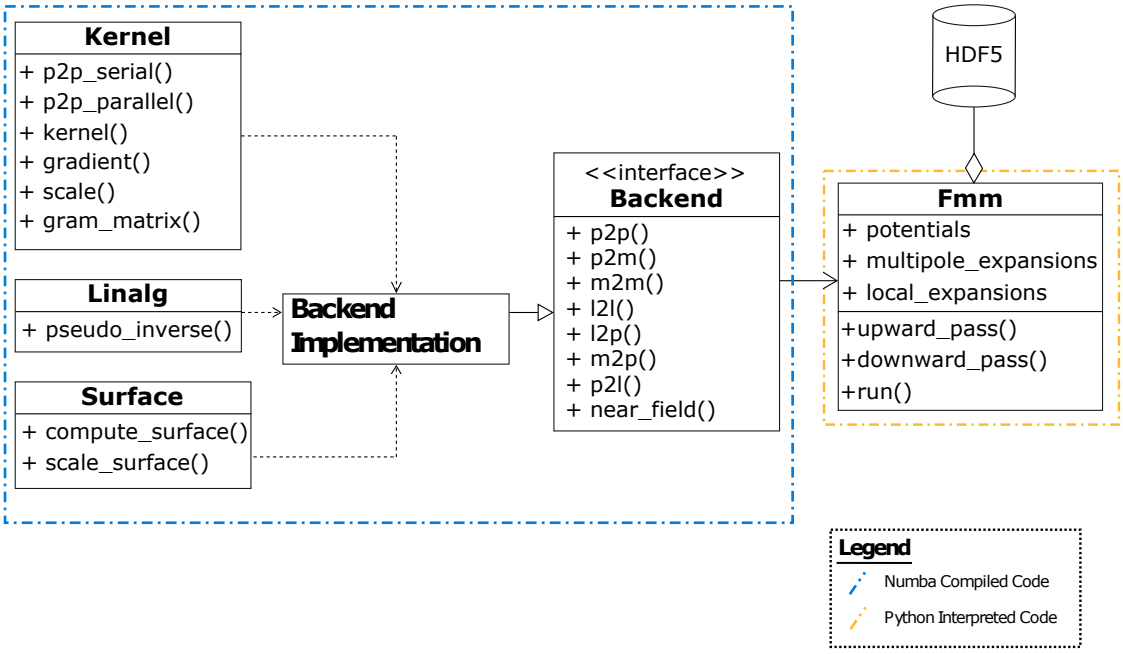

Figure 2. Simplified UML model of all PyExaFMM components. Trees and other precomputed quantities are stored in a HDF5 database. The 'Fmm' object acts as the user interface, all other components are modules consisting of methods operating on arrays.

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):
        B = A @ A
```

by Numba, and this leads to *oversubscription*, where 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 hanging threads threads, 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.

PARALLELIZATION STRATEGIES FOR FMM OPERATORS

The P2P, P2M, P2L, M2P, L2P and near field operators all rely on the P2P operator, as this computes 1 over their respective sources and targets, and are parallelized over their targets, the leaf nodes.

For the L2P operator 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 expansion coefficients, such that all the data required to apply an operator to single target node is adjacent in memory. 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 lookup the associated data to fill the cache local vectors.

The speedup achieved with this strategy, in comparison to a naive parallel iteration over the L2P's targets, increases with the number of calculations in each thread and hence the expansion

order p. In an experiment with 32768 leaves, $n_{crit} = 150$ and $p = 10$, our strategy is 13 % faster. This corresponds to a realistic FMM problem with approximately $1e6$ randomly distributed particles.

Due to their large interaction lists, the previous strategy is too expensive in terms of memory for the near field, M2L and M2P operators. For example, allocating an array large enough to store the maximum possible number of source particle coordinates in double precision for the M2P operator; with $|W| = 148$ and $n_{crit} = 150$, requires $\sim$ 17GB, and a runtime cost for memory allocations that exceeds the computation time. Instead, for the M2L we perform a parallel loop over the target nodes at each given level, and over the leaf nodes for the M2P and near field, looking up the relevant data from the linear tree as needed. The P2L interaction list of each target is at most 19 nodes, and the P2M must also calculate a check potential, cancelling out any speedup from cache locality for these operators.

The matrices involved in the M2M and L2L operators can be precomputed and scaled at each level [7], and their application is parallelized over all nodes at a given level.

Multithreading in this way means that we call the P2P, P2M, M2P, L2P and near field operators once during the algorithm, the M2L and L2L are called $d - 2$ times, and the M2M is called d times, where d is the depth of the octree. This is the minimum number of calls while keeping the operator implementations separate for unit testing.

Figure (3) compares the time spent within each Numba-compiled operator ('CPU time') to the total runtime ('wall time') of each operator. The results are computed over five trials over 32768 leaves, with $n_{crit} = 150$ and $p = 6$, for a random distribution of $1e6$ charges distributed on the surface of a sphere representing a typical FMM problem. The mean size of the interaction lists are $|U| = 11$, $|V| = 42$, $|X| = 3$, $|W| = 3$, and the entire algorithm is computed in $5.95 \pm 0.02s$, with an additional $9.00 \pm 0.01s$ for operator pre-computations for a given dataset, which is unachievable in ordinary single-threaded interpreted Python.

The wall time includes the time to (un)box data, organize inputs for Numba compiled functions, and pass control between Numba and Python. Except for the L2P which has a different parallelization strategy that requires significant data organization that must take place within the GIL restricted Python interpreter, the runtime costs are less than 5 % of each operator's total wall time, implying that we are nearly always running multithreaded code and utilizing all available CPU cores.

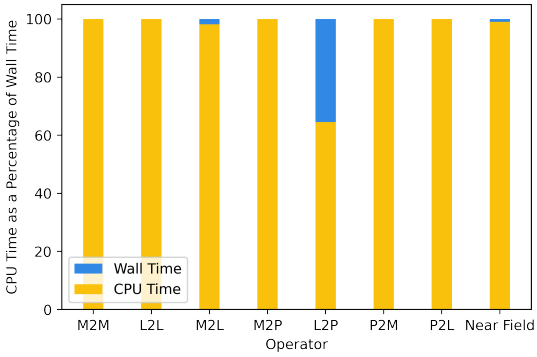

Figure 3. CPU time as a percentage of wall time for operators. CPU time is defined as the time in which the algorithm runs pure Numba compiled functions. Wall time is CPU time in addition to the time taken to return control to the Python interpreter.

CONCLUSION

Achieving optimal multithreaded performance with Numba requires careful consideration of the algorithm being accelerated, details of Numba's backend implementation, as well as a design that suits Numba's data oriented framework. The pitfalls illustrated above show how a user must potentially adapt their code significantly in order to achieve the best performance. Altogether, Numba accelerated code may look decidedly un-Pythonic despite using Python syntax. The complexities involved when using Numba to implement a non-trivial algorithm contrast with its advertisement as a simple way of injecting performance into Python code by applying a decorator. Significant software development expertise is needed in order to optimize a Numba implementation, and arguably more than many in its intended target audience can be expected to possess.

Despite this, Numba is a remarkable tool. Projects which value Python's expressiveness,

simple cross platform builds, as well as large open source ecosystem, and only contain a small number of isolated performance bottlenecks would benefit the most from a Numba implementation. Indeed, by writing only in Python our project size is kept minimal with the entire project running to just 4901 lines of code. Furthermore, we are able to deploy PyExaFMM cross platform trivially with Conda and distribute our software in popular Python channels without the need to create and maintain a separate Python interface as many compiled language packages for computational science do.

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.

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.