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. Here we document 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 that would ordinarily be run through the Python interpreter. We explain how Numba influences the design and structure of software, as well as its major pitfalls. We find that Numba doesn't live up to its promise due to unavoidable interaction between the Python interpreter and Numba compiled code, and that our software remains up to 30 times slower than ExaFMM-T, the leading C++ implementation of the same algorithm. However, the dramatic speedup in comparison to vanilla Python [INSERT BENCH HERE], means that Numba is still an excellent choice for projects in which the portability and simplicity that Python offers are a greater priorities than raw performance.**

PYTHON[1], is designed for memory safety and developer productivity, not speed. Its main selling factor being that it allows Computational Scientists to spend more time exploring the science, and less time being confused by strange software quirks, infuriating memory errors, and the nightmare of incompatible dependencies, which conspire to drain productivity when working with lower level compiled languages.

Code is run through an interpreter and restricted to run on 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 and 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 fast compiled implementations of algorithms for numerical linear algebra, and mathematical solvers for problems from differential equations to statistics.

As the actual number-crunching itself happens outside of the GIL, it only becomes a bottleneck to performance if a program must repeatedly pass control between the Python interpreter and non-Python code. For example, if an optimized compiled language implementation of your algorithm doesn't exist in the Python Open Source, or if it requires a lot of data organization which must take place within the interpreter, and cannot be formed in terms of Numpy or SciPy routines to create the input for optimized functions from those libraries. Previously, an unlucky developer would have been forced to write a compiled implementation themselves and connect it to their Python package.

This is the context in which Numba was introduced. It specifically targets Python code written for Numpy's ndarray data structure, and 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[2], an implementation of the particle Fast Multipole Method [FMM]. Efficient three dimensional implementations of this algorithm are complicated by a recursively defined tree data structure and algorithm with a series of operators which require significant data organization and careful memory allocation in between steps in order to create the simple linear inputs for the Numpy and SciPy routines used. These features made PyExaFMM an excellent test case to see whether Numba could free Computational Scientists from the complexities of compiled languages.

We begin by introducing Numba, focussing on when and where its use is appropriate. After introducing the key concepts of the FMM in order to understand its data structure and the computations involved in its computation, we proceed to an overview of how we designed our software's data structures and API to optimally use Numba. We show how we optimized multithreaded functions for cache re-use, and proceed with a discussion on the main pitfalls we encountered with Numba. We conclude with two benchmarks. Firstly we test the impact of Numba in comparison to a vanilla Python implementation. Secondly, we test the performance of PyExaFMM's various operations in terms of memory consumption and runtime in comparison to ExaFMM-T[3], the leading C++ implementation of the same algorithm.

NUMBA

- Scope and aim of Numba. What is it designed to do, and what isn't it designed to do. Where it can be impactful. (ndarrays, fusing loops, autovectorization and what this is via llvm)

- diagram of how Numba works, including how the api works i.e. python wrapper function, that dispatches python objects to compiled numba functions.

- Explain how Numba can target multiple types of target due its generic design.

- how numba functions interacts with python interpreter, and the explanation of box-

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

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

[3]https://github.com/exafmm/exafmm-t

Published by the IEEE Computer Society

ing/unboxing and a metric for this cost vs function arguments.

- code snippet for numba in real life, and when/where it is not appropriate. Contextualization of the code snippet wrt to the diagram of how Numba works.

- overview of multithreading in scientific python and the current state of affairs and why this isn't appropriate for hpc apps in pure python, and multithreading api in numba, and make analogies with openmp where they exist. Thread oversubscription issues and their origin and how to avoid this.

THE FAST MULTIPOLE METHOD

- data structure - hierarchical, non-uniform leaves, results in 4 distinct interactions. (U, V, X, W). Explanation how this is represented linearly here in order to work with Numba. this can be illustrated with a picture. Can refer to a figure in another paper.

- Discretization needs to be illustratred, check surface and equivalent surfaces. Can refer to a figure in another paper.

- approximate potentials with equivalent expansions

- upward pass - post order traversal

P2M - parallel. loop over leaves. check potential = $O(n_l \cdot n_{crit} \cdot n_c)$ equivalent charge = $O(n_l \cdot n_e \cdot n_c)$.

M2M - serial. cannot parallelize over leaves, there are parallel writes to parent multipole expansion from siblings. Parallelizing over sibling leaves is hard due to linear representation of tree - have to perform expensive neighbours searches to find siblings to perform group by. equivalent charge = $O(8^l \cdot n_e \cdot n_c)$ at a given level l.

- downward pass - pre-order traversal

L2L. serial - interaction

M2L

S2L

P2P

Pictures: - illustrate expansion orders for approximating in a single box?

- tree figure, and list of kernels that are compueted at each stage, and exactly what the computations involve for kernel independent fmm. Offer complexity bounds on these calculations. Contextualize the computations required of each kernel. i.e. P2P is raw multithreading performance.

- Offer analysis of where it is suitable to parallelize.

- A short note on the decision for M2l via SVD for FMM experts.

DATA ORIENTED DESIGN

- Data oriented design. - How this is reflected in tree design - How this is reflected in kernel design - API design

- Software design diagram - Minimize the impact of running code in the interpreter.

MULTITHREADING IN NUMBA

- how kernels are multithreaded bearing in mind issues raise in Numba section, and data oriented design section. - Optimal design of multithreading approach relies on the actual constraints of the kernel in question, will need to get into specifics here of how each kernel was approached. - metric for portion of code that is run on single vs multiple threads, can actually time this at least roughly.

PITFALLS OF NUMBA

- Numba/Numpy API differences for numerical methods can be a pitfall (svd)

- it is not Python, and writing like python (OOP) will lead to failure, numba is a framework

- not everything that is supported is fast, e.g. hashing example, and how we got around this.

- it's really designed for numerical work alone, and that's where it shines.

BENCHMARKS

- contrast multithreading impact for each kernel

- compare and contrast pyexafmm vs exafmm for a given accuracy, try and find optimum compression parameter.(?)

CONCLUSION

foo bar

ACKNOWLEDGMENT

SK is supported by EPSRC Studentship 2417009.

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.