Making PyExaFMM Performant

Srinath Kailasa

Department of Mathematics
University College London

May 5, 2021

Table of Contents

Table of Contents

Is Numba Up To It?

Problem: Laplace Kernel applied to interactions between 20,000 particles to find Gram Matrix

- Numba: $\sim$ 296ms
- Rust: $\sim$ 155ms

So at first glance, yes - Numba appears to be useful.

Is Numba Up To It?

Important caveats for benchmark,

- Numba targeting all CPUs via prange operator
- Data is already arranged ‘nicely’

Language Bottlenecks For Speed

- Native Python objects are slow for our purposes, with important implications for data access
- Python's numeric libraries (Numpy/Numba) etc haven't focussed on distributing data as much as computation. There are moves in this direction (Legate)

Language Bottlenecks For Speed

Problems faced by PyExaFMM:

- How do we store tree efficiently if we want to avoid native PyObjects?
- How do we access precomputed operators and coordinate data without dictionaries, or access to pointers?
- How can we accelerate computations on the tree structure?

Language Bottlenecks For Speed

Laplace Problem Benchmark: 1e6 randomly distributed points, max 100 particles per node, order 5 multipole expansions and order 6 local expansions, with M2L compression rank of 1.

- FMM takes 15s on my i7 processor.
- Precomputations of the tree and operators, accelerated with GPU, takes 9s, with my NVidia Quadro RTX-3000 GPU.

Table of Contents

Separating Concerns

- Separate trees into their own module as a dependency - AdaptOctree
- Separate precomputations into script, with CLI for interactive programming
- Minimal API, consists of a single object containing FMM loop and required data
- Separate compute 'backend' (e.g. choice between Numba, Cython etc) and 'frontend' consisting of the API

Separating Concerns

AdaptOctree is:

- Accelerated with Numba
- Tree construction is multithreaded, balancing and interaction list computation are single-threaded
- Benchmark: 1e6 particles distributed randomly, build time - 244 ± 6 ms, balance time - 97.5 ± 3 ms, interaction lists (slow) - 5.81 ± 0.07s

Separating Concerns

We use a single HDF5 file to store:

- Precomputed FMM operators
- Precomputed tree stored as a single vector, and tree parameters (physical center, radius etc)
- Precomputed index pointers tying together tree index and particle indices
- Precomputed interaction lists
- Benchmark: 1e6 randomly distributed particles, order 5 multipole expansion, order 6 local expansion, max 100 particles per node, target rank 1 in SVD, takes 9s.

Designing Around Data

Avoiding array creation/destruction leads to use of index pointers (linking indices between two arrays) e.g. between array of tree nodes, and their corresponding particles.

- Prioritise simple data structures, arrays
- Avoid sets and dicts if possible, these are partially supported by Numba - but add complications in terms of allowed return types, instatiation time
- All computational methods stripped down to bare loops to help Numba, and operator in place on data - looks UnPythonic and more C-like.

Designing Around Data

Numba Gotchas

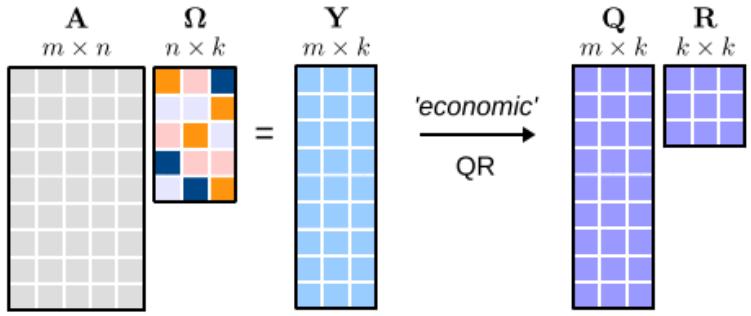
- Numba threading on the face of it simple: range $\rightarrow$ prange for a given for loop, picks up backend (TBB, OMP etc).
- Can lead to oversubscription of threads (silently) killing performance
- JIT'ing doesn't change allocation cost, so have to be careful to operate in place, this then feels like you are programming to an (unspecified) framework!
- Sometimes multithreading doesn't work as expected, especially when performing recuctions - rewriting to ensure that all operations are in place may help (i.e. avoiding allocations within prange loop).
- Numba works 'too well' - fails silently, and always produces output!

Table of Contents

Accelerating M2L

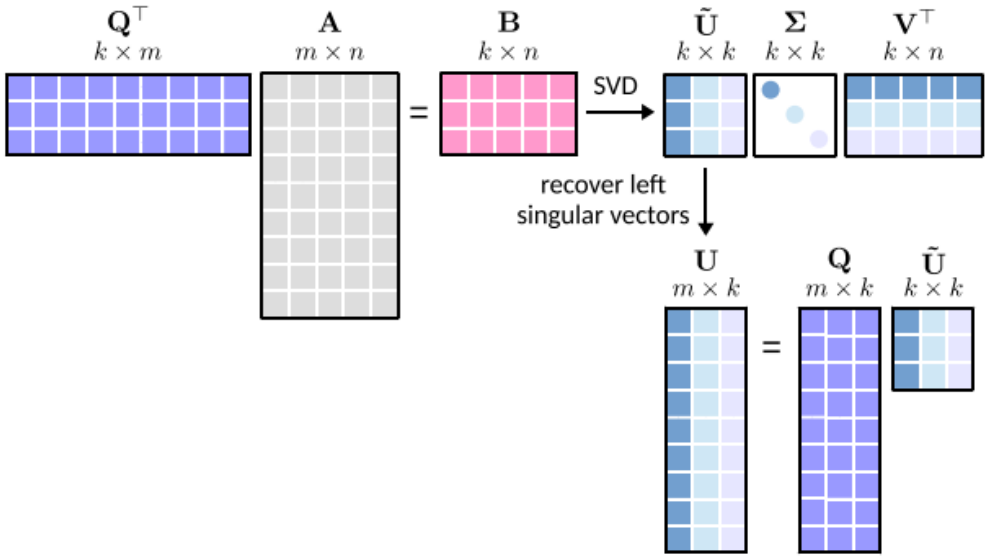
Offers a case study in utility of Numba for scientific computing, but also the pitfalls.

Randomised SVD Recap



Adapted from [1]

Randomised SVD Recap



Adapted from [1]

Compressing the M2L Gram Matrix Implicitly

- 'A' corresponds to Gram Matrix for unique M2L interactions for a given target node and up to 316 source nodes at each level
- We just store coordinates, and compute matrix elements implicitly, saving memory
- Use Cupy and Numba interoprability to transfer data to device, and compute SVD/QR algorithms - easy!
- Only compute unique interactions for a given level, referencing these with a 'transfer vector' [2]

Unexpected Problems

- Need to compute a hash corresponding to a given transfer vector on the fly, can't lookup using coordinate vector as these can't be used as keys in HDF5 (or PyDict).
- How does one efficiently hash a vector? Does Numba support hashes? Is it fast?
- Inability to treat functions as first class objects (for interoprability with Cupy functions) leads to massive code duplication for different kernels.
- At this point, the code is already fairly UnPythonic (no objects for encapsulating behaviour, no enforced interfaces) - why are we still using Python?

References I

[1] N. Benjamin Erichson, Sergey Voronin, Steven L. Brunton, and J. Nathan Kutz.
Randomized matrix decompositions using r.
Journal of Statistical Software, 89(11), 2019.

[2] William Fong and Erich Darve.
The black-box fast multipole method.
Journal of Computational Physics, 228, 2019.