A Whirlwind Tour of Numba

Valentin Haenel

Senior Software Engineer - Open Source
https://anaconda.com

Python Users Berlin - Thursday 19 Feb 2026

whoami

- Senior Software Engineer at Anaconda
- Working on Numba full-time
- Doing this for almost 7 years
- Actively involved in:
    - Typed containers
    - Release and community management
    - Structured Control Flow Graphs

Outline

Numba in a Nutshell

- A compiler that might make your code faster
- Requires importing a decorator: `@jit`
- And decorating functions with it
- Numba = NumPy + Mamba (fast snake)

Numba Explained

- Numba is a
    - just-in-time
    - type-specializing
    - function compiler
    - for accelerating numerically-focused Python

LLVM

- LLVM is a compiler toolkit
- Numba uses it as a compiler backend
- Access via `llvmlite`

Example

- Sieve of Erastothenes

Example

```python
import numpy as np
from numba import jit


@jit  # simply add the jit decorator
def primes(max=100000):
    numbers = np.ones(max, dtype=np.uint8)  # initialize the boolean sieve
    for i in range(2, max):
        if numbers[i] == 0:  # has previously been crossed off
            continue
        else:  # it is a prime, cross off all multiples
            x = i + i
            while x < max:
                numbers[x] = 0
                x += i
    # return all primes, as indicated by boolean positions that have value 1
    return np.nonzero(numbers)[0][2:]
```

Example

```
In [2]: %timeit sieve.primes.py_func()
11.8 ms ± 109 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [3]: %timeit sieve.primes()
177 µs ± 11.2 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Open Source Status

- Several companies and many open source contributors
  - several FTE funded to contribute to Numba
  - several non-core contributors per release
- Issue and PR lists growing
- Very active community on GitHub, Discourse

Community Usage

- Download rate via PyPi is ~21 Hz (audible by goldfish)
- GitHub: 163126 Repositories and 6513 Packages
- Several high-profile libraries use it:
  - PyData Sparse $\rightarrow$ sparse matrix implementation
  - UMAP $\rightarrow$ Uniform Manifold Approximation
  - librosa $\rightarrow$ Music analysis
  - Tardis-SN $\rightarrow$ Super Nova Simulator for Type 1a (White Dwarf Nova)
  - Pandas $\rightarrow$ Dataframe Library
  - NumbaDuck $\rightarrow$ Fully jitted DuckDB interface
  - numba-cuda, pyOMP, numba-hip $\rightarrow$ GPU Programming
  - Caterva-2 $\rightarrow$ Compressed Compute Engine based on Blosc
  - many, many more, we eventually stopped keeping track

Outline

1 Introduction

2 **Going Deeper**

3 NumPy Support

4 Tips and Tricks

5 Compiler Toolkit

6 Summary

```
@jit
def do_math(a, b):
    …
>>> do_math(x, y)
```

Internals
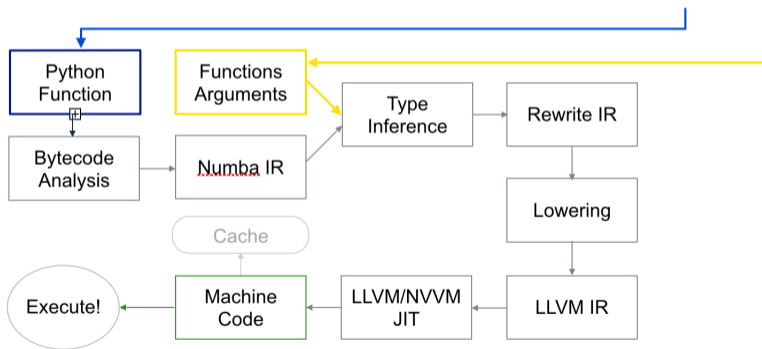
- Translate Python objects of supported types into representations with no CPython dependencies ("unboxing")
- Compile Python bytecode from your decorated function into machine code.
- Swap calls to builtins and NumPy functions for implementations provided by Numba (or 3rd party Numba extensions)
- Allow LLVM to inline functions, autovectorize loops, and do other optimizations you would expect from a C compiler
- Allow LLVM to exploit the instruction sets of your hardware (SSE, AVX)
- When calling the function, release the GIL if requested
- Convert return values back to Python objects ("boxing")

What Numba does not do

- Automated translation of CPython or NumPy implementations
- Automatic compilation of 3rd party libraries
- Partial compilation
- Automatic conversion of arbitrary Python types
- Change the layout of data allocated in the interpreter
- Translate entire programs
- Magically make individual NumPy functions faster

When is Numba unlikely to help?

- Whole program compilation
- Critical functions have already been converted to C or optimized Cython
- Need to interface directly with C++
- Algorithms that are not primarily numerical, e.g. string manipulation
- Exception: Numba can do pretty well at bit manipulation

Outline

Implementing NumPy

- Numba does not use much of the NumPy C implementations
- (Only some ufuncs are "borrowed".)
- We implement the NumPy API using Numba compatible/supported Python
- Treat NumPy as DSL for array oriented computing

Implementing Numpy

- Implement: `numpy.linalg.norm`
- For vectors, `ord` is:
    - inf $\rightarrow$ min(abs(x))
    - 0 $\rightarrow$ sum(x != 0)
- Implemented in: `numba.targets.linalg.py`

Implementing Numpy

```python
elif ord == -np.inf:
    # min(abs(a))
    ret = abs(a[0])
    for k in range(1, n):
        val = abs(a[k])
        if val < ret:
            ret = val
    return ret

elif ord == 0:
    # sum(a != 0)
    ret = 0.0
    for k in range(n):
        if a[k] != 0.:
            ret += 1.
    return ret
```

Outline

Tips and Tricks

- Always use @jit
- Prefer NumPy arrays for numerical data
- Use typed containers from numba.typed for nested data
- for loops are fine
- Array expressions are fused if on the same line, so those are fine too

Typed Containers

```python
from numba import njit
from numba.typed import List

l = List()  # instantiate a new typed list
l1 = List(); [l1.append(i) for i in range(5)]
l.append(l1)  # add the first sub-list
l2 = List(); [l2.append(i) for i in range(10)]
l.append(l2)  # add the second sub-list
print(l)

@njit
def func(my_list):
    # modify list in a compiled context
    for i in range(10):
        my_list[1][i] = 23

func(l)
print(l)
```

Typed Containers

```
$ python code/typed.py
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
[[0, 1, 2, 3, 4], [23, 23, 23, 23, 23, 23, 23, 23, 23, 23]]
```

Fused Expressions

```python
import numpy as np
from numba import njit

a, b = np.arange(1e6), np.arange(1e6)

@njit
def func(a, b):
    return a*b-4.1*a > 2.5*b
```

Fused Expressions

```
In [1]: from fused import a,b,func

In [2]: %timeit func.py_func(a,b)
4.68 ms ± 89.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [3]: %timeit func(a,b)
626 µs ± 22.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

OS and Hardware Support

- `osx-arm64`
- `linux-64`
- `linux-aarch64`
- `win-64`

Python versions

- Python 3.10 - 3.14 as of Numba 0.64 and llvmlite 0.46

Packaging

- You can depend on Numba to perform the heavy lifting!
- We run CI on most Python/NumPy/OS/Hardware combinations
- You can ship a single source package
  - PyPi
  - anaconda.org
- No need to pre-compile binaries for your users
- Reduced code needed for distribution

Testing

- Ported CI from custom server farm to GHA last year
- Testing matrix is a cross-product: Python x NumPy x Platform
- GHA runs for 12 hours on a single commit during release process

Outline

inspect methods

- There are various `inspect` methods on compiled functions
  - `inspect_types` → Prints out results from type-inference
  - `inspect_llvm` → Obtain LLVM IR
  - `inspect_asm` → Obtain print out of function assembly
  - `inspect_cfg` → Obtain Control Flow Graph
  - `inspect_dissam_cfg` → Control Flow Graph and disassembly from reversing generated ELF

Custom Compiler Passes

```python
from numba import njit
from numba.core import ir
from numba.core.compiler import CompilerBase, DefaultPassBuilder
from numba.core.compiler_machinery import FunctionPass, register_pass
from numba.core.untyped_passes import IRProcessing
from numbers import Number

# Register this pass with the compiler framework, declare that it will not
# mutate the control flow graph and that it is not an analysis_only pass (it
# potentially mutates the IR).
@register_pass(mutates_CFG=False, analysis_only=False)
class ConstsAddOne(FunctionPass):
    _name = "consts_add_one" # the common name for the pass

    def __init__(self):
        FunctionPass.__init__(self)
```

Custom Compiler Passes

```python
# implement method to do the work, "state" is the internal compiler
# state from the CompilerBase instance.
def run_pass(self, state):
    func_ir = state.func_ir # get the FunctionIR object
    mutated = False # used to record whether this pass mutates the IR
    # walk the blocks
    for blk in func_ir.blocks.values():
        # find the assignment nodes in the block and walk them
        for assgn in blk.find_insts(ir.Assign):
            # if an assignment value is a ir.Consts
            if isinstance(assgn.value, ir.Const):
                const_val = assgn.value
                # if the value of the ir.Const is a Number
                if isinstance(const_val.value, Number):
                    # then add one!
                    const_val.value += 1
                    mutated |= True
    return mutated # return True if the IR was mutated, False if not.
```

Custom Compiler Passes

```python
class MyCompiler(CompilerBase): # custom compiler extends from CompilerBase

    def define_pipelines(self):
        # define a new set of pipelines (just one in this case)
        pm = DefaultPassBuilder.define_nopython_pipeline(self.state)
        # Add the new pass to run after IRProcessing
        pm.add_pass_after(ConstsAddOne, IRProcessing)
        pm.finalize()
        return [pm]
```

Custom Compiler Passes

```python
@njit(pipeline_class=MyCompiler) # JIT compile using the custom compiler
def foo(x):
    a = 10
    b = 20.2
    c = x + a + b
    return c

print(foo(100)) # 100 + 10 + 20.2 (+ 1 + 1), extra + 1 + 1 from the rewrite!
```

Outline

Understood Numba?

- Numba is a
  - just-in-time
  - type-specializing
  - function compiler
  - for accelerating numerically-focused Python

What is Cooking?

- Currently in a PR cleanup, hoping to reduce number of open PRs
- Next gen: numbacc compiler based on Equality Saturation
- PIXIE based backend, toolchain and executable format

Getting in Touch

- https://numba.pydata.org
- https://github.com/numba
- https://numba.discourse.group/
- Preferred: GitHub + Discourse
- Stuck? It can't hurt to ask. ;-)