Understanding Numba

Valentin Haenel

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

Friday 04 March 2022

Version: 2022-03-04-turchin-seminar https://github.com/esc/numba-talk

whoami

- Compiler Engineer at Anaconda
- Working on Numba full-time
- Doing this since three years

Outline

Numba

Numba

Numba in a Nutshell

- A compiler that might make your code faster
- Requires importing a decorator
- 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(nopython=True)  # 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 all boolean positions that are one
    return np.nonzero(numbers)[0][2:]
```

Example

```
In [5]: %timeit sieve.primes.py_func()
124 ms ± 2.72 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [6]: %timeit sieve.primes()
308 µs ± 8.93 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Open Source Status

- Several companies and many open source contributors
  - 5 FTE funded to contribute to Numba between Anaconda, Intel, Quansight, Nvidia, Bodo.ai and others
  - 7-12 non-core contributors per release (every 3 months)
- Slowly moving towards NumFocus application
- Issue and PR lists growing
- Very active community on GitHub, Discourse and Gitter

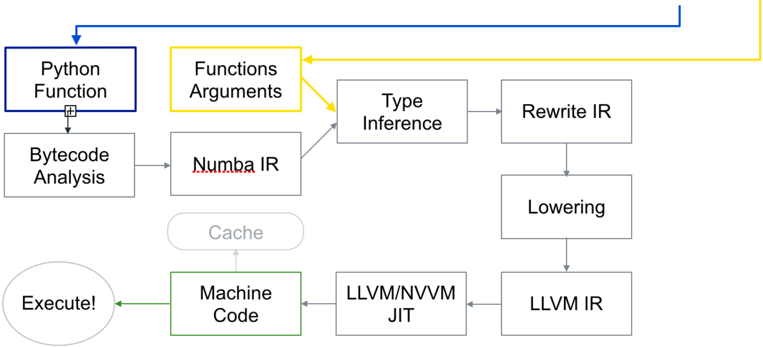
Community Usage

- Millions of Downloads per month
- On GitHub about 50k repositories have an `import numba`
- Several high-profile libraries use it:
  - PyData Sparse → sparse matrix implementation
  - UMAP → Uniform Manifold Approximation
  - Tardis → Super Nova Simulator
  - many, many more

Outline

Numba Flow

```
@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 all supported 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 to C++
- Algorithms are not primarily numerical, e.g. strings
- Exception: Numba can do pretty well at bit manipulation

Outline

Implementing Numpy

- Numba does not use any of the NumPy C implementations
- 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

```
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(nopython=True)`
- (Or `njit`)
- Prefer Numpy arrays for numerical data
- Use typed containers from `numba.typed` for nested data
- Use small functions, they are inlined
- `for` loops are fine
- Array expressions are fused, 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 Support

- Windows 7 and later, 32 and 64-bit
- macOS 10.9 and later, 64-bit
- Linux (most anything greater than RHEL 5), 32-bit and 64-bit

Python versions

- Python 3.7 - 3.10 (3.11 being worked on)
- NumPy 1.18 and later

Hardware Support

- x86, x86_64|AMD64 CPUs
- NVIDIA CUDA GPUs (Compute capability 5.3 and later, CUDA 11.2 and later)
- ARM 32-bit (Raspbery Pi) and 64-bit (Jetson TX2)
- POWER8|9
- Apple M1 Silison (in progress)

Packaging

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

Outline

inspect methods

- There are various inspect methods on compiled functions
  - inspect_types → Printout results from type-inference
  - inspect_llvm → Obtain LLVM IR representation
  - inspect_asm → Obtain assembly representation
  - inspect_cfg → Obtain Control Flow Graph
  - inspect_dissam_cfg → Reversed Control Flow Graph from 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

```
# 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

```
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

Outline

1. Introduction
2. Going Deeper
3. NumPy Support
4. Tips and Tricks
5. Compiler Toolkit
6. Summary?

Understood Numba?

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

What is Cooking?

- 3.11 and a new bytecode analyser
- Python $\rightarrow$ Assembly visualizer
- Automatic inspection of Numpy Support
- Scipy + Numba = `numba-scipy`

Getting in Touch

- https://numba.pydata.org
- Preferred: GitHub + Gitter
- Stuck? It can't hurt to ask. ;-)