Understanding Numba

Valentin Haenel

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

LLVM Social Berlin - Wednesday 23 October 2024

Version: 2024-10-23-llvm-berlin-meetup-3-g542a4f87c7 https://github.com/esc/numba-talk

whoami

- Senior Software Engineer at Anaconda
- Working on Numba full-time
- Doing this for over 5 years
- Actively involved in:
    - Typed containers
    - Release and community management
    - Compiler frontend design and implementation

Outline

Numba

Numba

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 [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 and others
  - 7-12 non-core contributors per release
- Issue and PR lists growing
- Very active community on GitHub, Discourse and Matrix/Gitter

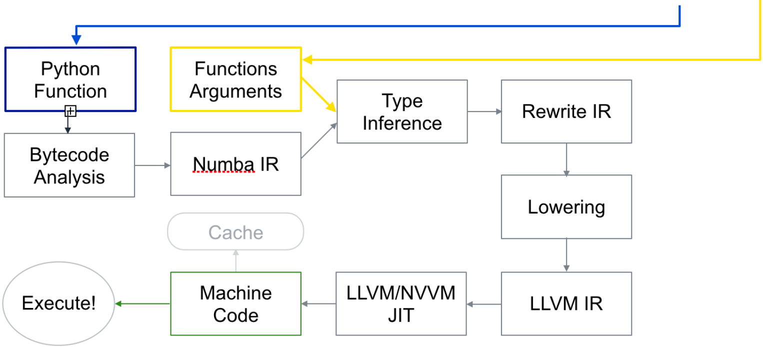
Community Usage

- 10s of Millions of Downloads per month
- On GitHub about 120k repositories are listed as dependents (Oct 2024)
- Several high-profile libraries use it:
  - PyData Sparse → sparse matrix implementation
  - UMAP → Uniform Manifold Approximation
  - Tardis → Super Nova Simulator
  - Pandas → Dataframe Library
  - Open AI Whisper → Speech-to-Text
  - NVIDIA RAPIDS → GPU Library
  - many, many more, we eventually stopped keeping track

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 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

`llvmlite`

- Numba's LLVM binding package
- A "lightweight" Python wrapper
- Not exactly intended as a standalone package, but a few people use it anyway
- Separation of concerns, easier to develop this way
- Uses the C++ API via C – easier to bind to from Python
- Stability is key, we only upgrade LLVM if necessary, don't chase the bleeding edge

`llvmlite` releases

- Released in lock-step with Numba
- About two to four times a year
- Each release only supports one (sometimes two) LLVM versions
  - $\rightarrow$ several disgruntled users
- Currently we support 14 but transitioning to 15
- ZERO-based versioning
- Patch releases only to fix severe regressions
- No backports

llvmlite packages

- Numba team creates two types of packages, wheels and conda packages
- The Numba team statically distributes LLVM
- Python wheels for distribution via PyPI
  - `20 * 30-40 MB`
- Conda packages for distribution via the `numba` channel on anaconda.org
  - `25 * 20-40 MB`
- Other distributors choose to link dynamically
- (This often creates issues for distributors, since they also only want to support one LLVM version, usually not the one we support but the latest)
- Hot take: LLVM is not a shared library – but something to be vendored

Why not `llvmpy`

- llvmlite comes from an era of LLVM 3.something
- Stability was more important, Numba is the main consumer
- Only the JIT part of LLVM needed (MCJIT)
- For example: our builder-API is entirely in Python and string based
- Better to control the bindings ourselves

Builder-API Example

```python
from llvmlite import ir

# Create some useful types
double = ir.DoubleType()
fnty = ir.FunctionType(double, (double, double))

# Create an empty module...
module = ir.Module(name=__file__)
# and declare a function named "fpadd" inside it
func = ir.Function(module, fnty, name="fpadd")

# Now implement the function
block = func.append_basic_block(name="entry")
builder = ir.IRBuilder(block)
a, b = func.args
result = builder.fadd(a, b, name="res")
builder.ret(result)

print(module)
```

Builder-API Example

```
; ModuleID = "examples/ir_fpadd.py"
target triple = "unknown-unknown-unknown"
target datalayout = ""

define double @"fpadd"(double %".1", double %".2")
{
entry:
  %"res" = fadd double %".1", %".2"
  ret double %"res"
}
```

Code Generation Example

```python
from __future__ import print_function

from ctypes import CFUNCTYPE, c_double

import llvmlite.binding as llvm


# All these initializations are required for code generation!
llvm.initialize()
llvm.initialize_native_target()
llvm.initialize_native_asmprinter()  # yes, even this one
```

Code Generation Example

```
llvm_ir = """
   ; ModuleID = "examples/ir_fpadd.py"
   target triple = "unknown-unknown-unknown"
   target datalayout = ""

   define double @"fpadd"(double %".1", double %".2")
   {
   entry:
     %"res" = fadd double %".1", %".2"
     ret double %"res"
   }
   """
```

Code Generation Example

```python
def create_execution_engine():
    """
    Create an ExecutionEngine suitable for JIT code generation on
    the host CPU.  The engine is reusable for an arbitrary number of
    modules.
    """
    # Create a target machine representing the host
    target = llvm.Target.from_default_triple()
    target_machine = target.create_target_machine()
    # And an execution engine with an empty backing module
    backing_mod = llvm.parse_assembly("")
    engine = llvm.create_mcjit_compiler(backing_mod, target_machine)
    return engine
```

Code Generation Example

```python
def compile_ir(engine, llvm_ir):
    """
    Compile the LLVM IR string with the given engine.
    The compiled module object is returned.
    """
    # Create a LLVM module object from the IR
    mod = llvm.parse_assembly(llvm_ir)
    mod.verify()
    # Now add the module and make sure it is ready for execution
    engine.add_module(mod)
    engine.finalize_object()
    engine.run_static_constructors()
    return mod
```

Code Generation Example

```python
engine = create_execution_engine()
mod = compile_ir(engine, llvm_ir)

# Look up the function pointer (a Python int)
func_ptr = engine.get_function_address("fpadd")

# Run the function via ctypes
cfunc = CFUNCTYPE(c_double, c_double, c_double)(func_ptr)
res = cfunc(1.0, 3.5)
print("fpadd(...) =", res)
```

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` → `min(abs(x))`
  - `0` → `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`
- 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

- Windows 7 and later 64-bit only
- macOS 10.9 and later, `x86_64` (Intel) and `arm64` (M1, M2 etc...)
- Linux `x86_64` and `aarch64` (no `ppc64le` anymore)
- NVIDIA CUDA GPUs (Compute capability 5.3 and later, CUDA 11.2 and later)
  - → currently being refactored into separate package `numba-cuda`

Python versions

- Python 3.10 - 3.13 as of Numba 0.62 and llvmlite 0.43

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

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?

- AST/Source based frontend
- RVSDG based lambda-calculus intermediary representation
- Hindley-Milner style type inference
- egglog/Equality saturation
- PIXIE based backend, toolchain and executable format
- Transition from MCJIT $\rightarrow$ ORCJIT
- Compiler modularization: ‘numba-cuda‘ and ‘numba-numpy‘
- Refactoring to support NumPy 2.0 type system

- Effectively: complete project overhaul

Getting in Touch

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