Extending Numba

FOSDEM 2019

Joris Geessels // @jolos

LUCEDA

JIT – compiler (LLVM)


```python
@numba.jit(nopython=True)
def go_fast(a):
    trace = 0
    for i in range(a.shape[0]):
        trace += numpy.tanh(a[i, i])
    return a + trace

x = numpy.arange(100).reshape(10, 10)
print(go_fast(x))
```

accelerates Python

numpy support ++

Simulation of a PIC

```
class WaveguideModel(CompactModel):
```

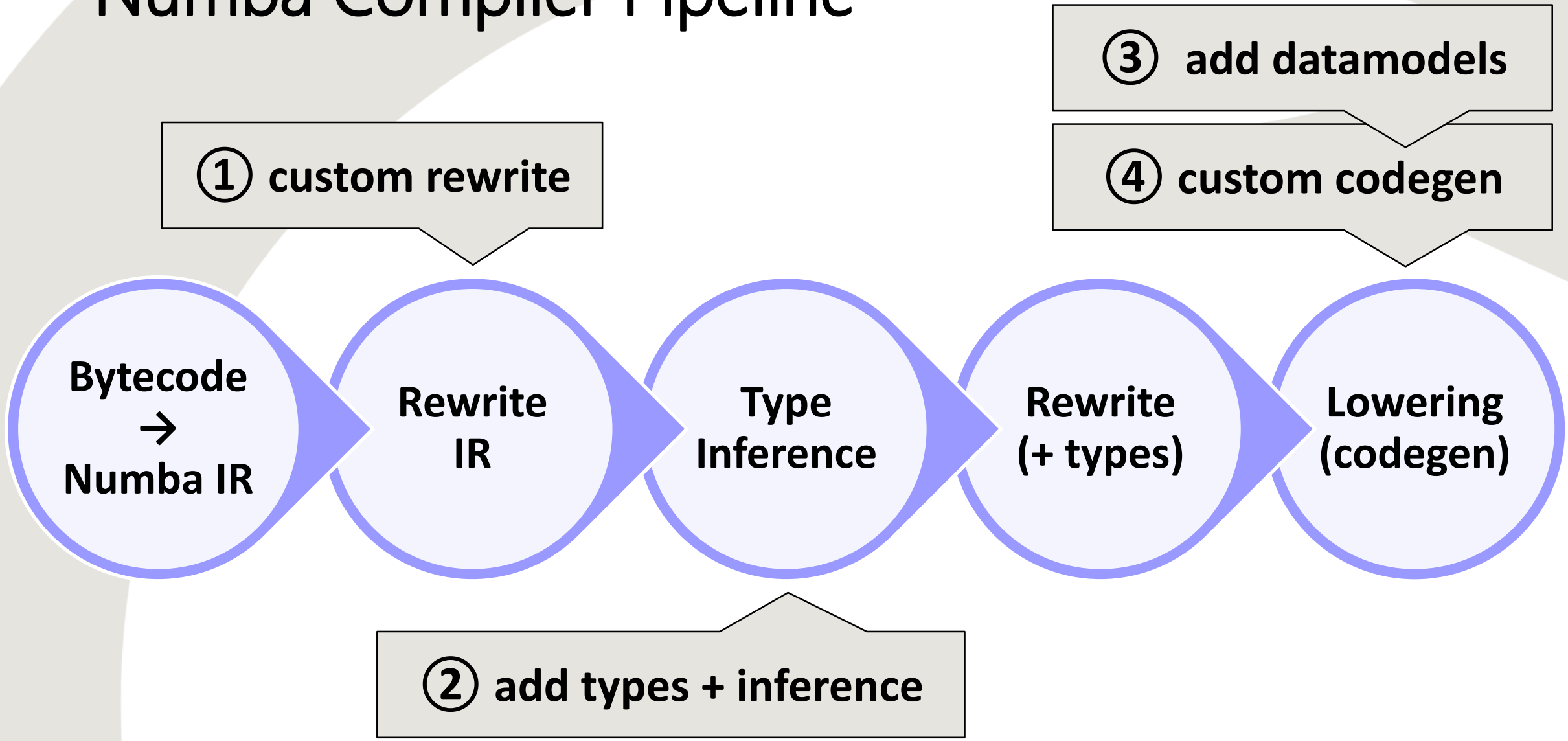
Called by simulator?

C++ objects?

```
def calculate_smatrix(params, env, S):
    phase = 2 * np.pi / env.wavelength * params.n_eff * params.length
    A = 0.99
    S['in', 'out'] = S['out', 'in'] = A * np.exp(1j * phase)
```

dict support?

Numba Compiler Pipeline



① Numba IR Rewrite

```
from numba import ir
from numba.rewrites import Rewrite, register_rewrite

# 'before-inference' or 'after-inference'
@register_rewrite('before-inference')
class MyRewrite(Rewrite):
    def match(self, func_ir, block, typemap, calltypes):
        # search for expressions to rewrite,
        # return True when match
        return True

    def apply(self):
        # return a new function 'block'
        return new_block
```

② Type Inference

```
# 2 public decorators to register your custom typers
from numba.extending import type_callable, typeof_impl

class MyPointType(numba.types.Type):
    # A custom type to represent a point
    # used during inference
    def __init__(self):
        super(MyPointType, self).__init__(name='Point')

@type_callable(MyPoint)
def type_MyPoint(context):
    def typer(x, y):
        # your_func returns a point
        return MyPointType()
    return typer
```

instantiate & return your custom type

③ Lowering Types

Lowering = generating LLVM intermediate representation (IR)

```
from numba.extending import register_model, models

@register_model(MyPointType)
class MyPointModel(models.StructModel):
    def __init__(self, dmm, fe_type):
        members = [
            ('x', types.int64),
            ('y', types.int64),
        ]
        models.StructModel.__init__(self, dmm,
                                    fe_type, members)
```

④ Lowering callables, setattr, getattr, ...

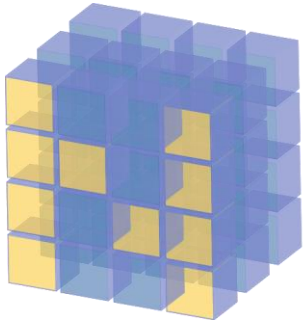
```
from numba.extending import lower_builtin
from numba import cgutils # llvm codegen utils

@lower_builtin(MyPoint, types.Integer, types.Integer)
def impl_point(context, builder, sig, args):
    typ = sig.return_type
    assert isinstance(typ, MyPointType)
    x, y = args
    point = cgutils.create_struct_proxy(typ)(context, builder)
    point.x = x
    point.y = y
    return point._getvalue()
```

Types of arguments

Codegen utilities for StructModel

Integration with C/C++

```
import numpy.ctypeslib
```



NumPy

Numba

```
from numba import carray, cfunc
import cffi
```

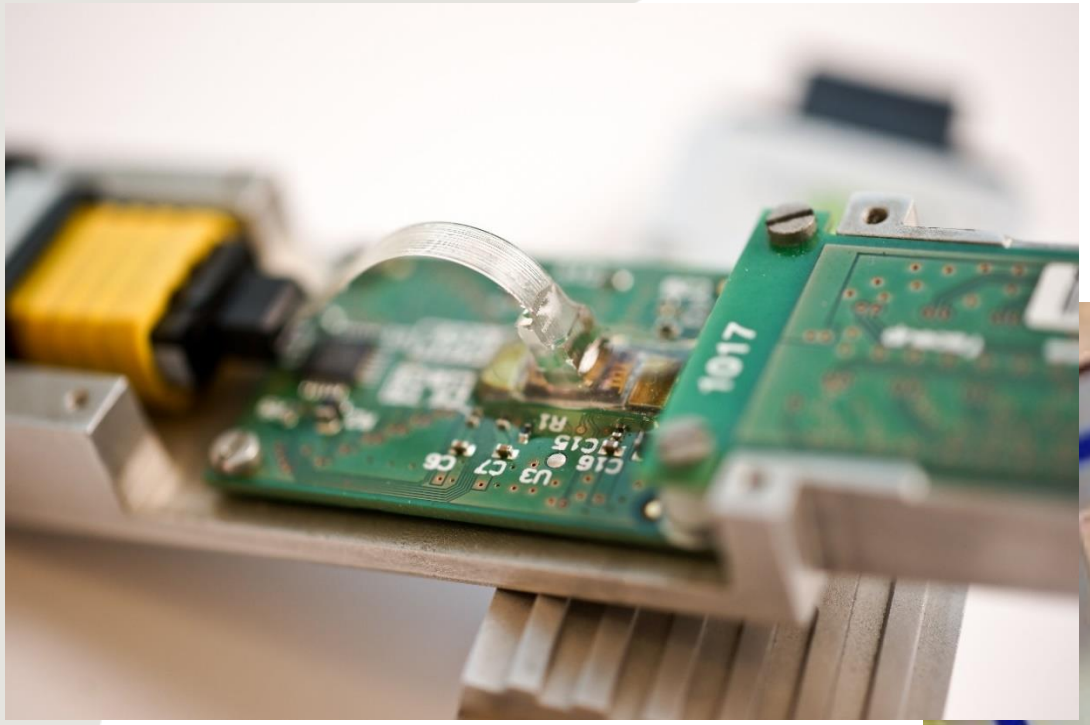
THE C PROGRAMMING LANGUAGE

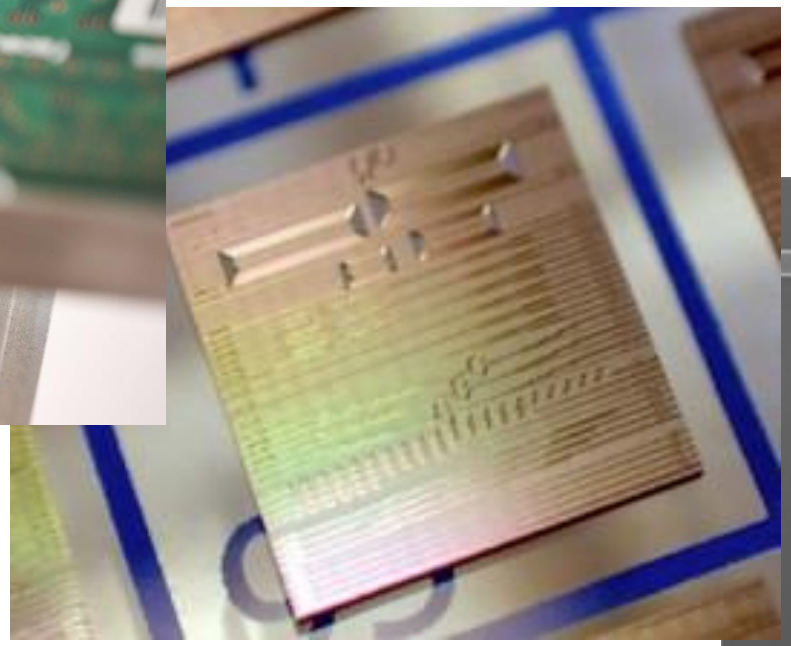
`sys.exit(0)`

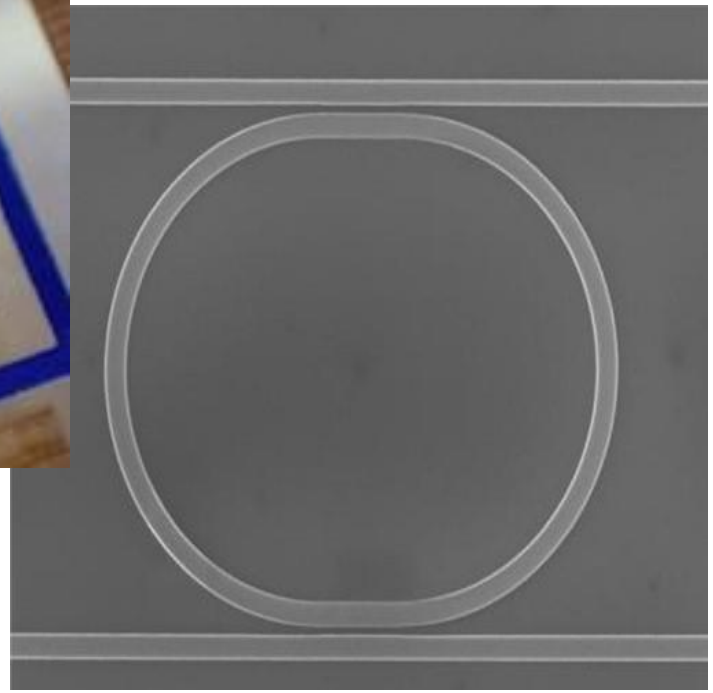
References & Docs:

- Good start: http://numba.pydata.org/numba-doc/latest/extending/interval-example.html
- Numpy ctypeslib: https://docs.scipy.org/doc/numpy/reference/routines.ctypeslib.html
- Numba uses Numba, so look at its code for examples: https://github.com/numba/numba
- Friendly introduction to LLVM: https://www.aosabook.org/en/llvm.html
- Examples: https://fosdem.org/2019/schedule/event/python_extending_numba/

Photonics Integrated Circuit (PIC)?







LUCEDA PHOTONICS

Interop with C++ classes?

1. Write **C – wrapper** for your C++ code
2. Create a Numba type and model to store **a C++ obj pointer**
3. Implement attributes using Numba's extension architecture
4. Pass the pointer to the C++ object to your **`numba.cfunc`**
5. Segfault ;-)