Speeding up scientific Python code using Cython

Advanced Python Summer School, Zurich

Stéfan van der Walt

September 2013
Example Code

https://python.g-node.org/wiki/cython
Introduction
Motivation

- Example Code
- Introduction
  - Motivation
  - Motivation (continued)
  - Use Cases
  - Tutorial Overview
- From Python to Cython
- Handling NumPy Arrays
- Parallelization
- Wrapping C and C++ Libraries

Diagram:
- Ease of Use: (Your Time)
- Speed: (Computer Time)
- Python
- Cython
- NumPy
- Fortran
- Tea Break

Zurich 2012
Motivation (continued)

- Cython allows us to cross the gap
- This is good news because
  - we get to keep coding in Python (or, at least, a superset)
  - but with the speed advantage of C
- You can’t have your cake and eat it. *Or can you?*
- Conditions / loops approx. 2–8x speed increase, 30% overall; with annotations: hundreds of times faster
Use Cases

- Optimize execution of Python code (profile, if possible! – demo)
- Wrap existing C and C++ code
- Breaking out of the Global Interpreter Lock; openmp
- Mixing C and Python, but without the pain of the Python C API
For this quick introduction, we’ll take the following approach:

1. Take a piece of pure Python code and benchmark (we’ll find that it is too slow)
2. Run the code through Cython, compile and benchmark (we’ll find that it is somewhat faster)
3. Annotate the types and benchmark (we’ll find that it is much faster)

Then we’ll look at how Cython allows us to

- Work with NumPy arrays
- Use multiple threads from Python
- Wrap native C libraries
Introduction

From Python to Cython

- Benchmark Python code
- More Segments
- Benchmark Python Code
- Compile the code with Cython
- Compile generated code
- Benchmark the new code
- Providing type information
- Benchmark
- Expense of Python Function Calls
- The Last Bottlenecks
- Integrating Arbitrary Functions (callbacks)
- Handling NumPy Arrays

Parallelization

Wrapping C and C++ Libraries

Zurich 2012
Our code aims to compute (an approximation of) \( \int_a^b f(x) \, dx \)
More Segments

- Example Code

Introduction

From Python to Cython

- Benchmark Python code
- More Segments

- Benchmark Python Code
- Compile the code with Cython
- Compile generated code
- Benchmark the new code
- Providing type information
- Benchmark
- Expense of Python Function Calls
- The Last Bottlenecks

Integrating Arbitrary Functions (callbacks)

Handling NumPy Arrays

Parallelization

Wrapping C and C++ Libraries
Benchmark Python Code

```
from __future__ import division

def f(x):
    return x**4 - 3 * x

def integrate_f(a, b, N):
    """Rectangle integration of a function."

    Parameters
    ----------
    a, b : int
        Interval over which to integrate.
    N : int
        Number of intervals to use in the discretisation.
    ""

    s = 0
dx = (b - a) / N
for i in range(N):
s += f(a + i * dx)
return s * dx
```
Compile the code with Cython

- `cython filename.[py|pyx]`
- What is happening behind the scenes? `cython -a filename.[py|pyx]`
  - Cython translates Python to C, using the Python C API (let’s have a look)
- This code has some serious *bottlenecks*. 
Compile generated code

Example Code

Introduction
From Python to Cython
Benchmark Python code
More Segments
Benchmark Python Code
Compile the code with Cython
Compile generated code
Benchmark the new code
Providing type information
Benchmark
Expense of Python Function Calls
The Last Bottlenecks
Integrating Arbitrary Functions (callbacks)
Handling NumPy Arrays
Parallelization
Wrapping C and C++ Libraries

$ gcc -02 -fPIC -I/usr/include/python2.7 -c integrate.c -o integrate_compiled.o

Easier yet, construct setup.py:

from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [
        Extension("integrate", ["integrate.pyx"]),
    ]
)

Run using python setup.py build_ext -i. This means: build extensions «in-place».
Benchmark the new code

- Use IPython’s %timeit (could do this manually using from timeit import timeit; timeit(...))
- Slight speed increase ($\approx 1.4 \times$) probably not worth it.
- Can we help Cython to do even better?
  - Yes—by giving it some clues.
  - Cython has a basic type inferencing engine, but it is very conservative for safety reasons.
  - Why does type information allow such vast speed increases?
Providing type information

```python
from __future__ import division

def f(double x):
    return x**4 - 3 * x

def integrate_f(double a, double b, int N):
    """Rectangle integration of a function."
    ...
    """
    cdef:
        double s = 0
        double dx = (b - a) / N
    Py_ssize_t i

    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```
Benchmark...
Expense of Python Function Calls

```python
def f(double x):
    return x**4 - 3 * x

def integrate_f(double a, double b, int N):
cdef:
    double s = 0
    double dx = (b - a) / N
    size_t i

    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```

![Diagram illustrating the expense of Python function calls]

**Python layer**
- `integrate_f(0, 1, 10000)`

**C layer**
- `_pyx_integrate_f(...)`
  - for (i=0; i<10000; i++)
- `_pyx_f(x)`
  - s is updated
The Last Bottlenecks

Example Code

Introduction

From Python to Cython

* Benchmark Python code
* More Segments
* Benchmark Python Code
* Compile the code with Cython
* Compile generated code
* Benchmark the new code
* Providing type information
* Benchmark
* Expense of Python Function Calls

The Last Bottlenecks

Integrating Arbitrary Functions (callbacks)

Handling NumPy Arrays

Parallelization

Wrapping C and C++ Libraries

---

```python
# cython: cdivision=True
cdef double f(double x):
    return x**4 - 3 * x

def integrate_f(double a, double b, int N):
cdef:
    double s = 0
    double dx = (b - a) / N
    Pyssize_t i

    for i in range(N):
        s += f(a + i * dx)

    return s * dx
```

---
Benchmark!
Integrating Arbitrary Functions (callbacks)

Example Code

```
# cython: cdivision=True

cdef class Integrand:
    cdef double f(self, double x):
        raise NotImplementedException()

cdef class MyFunc(Integrand):
    cdef double f(self, double x):
        return x*x*x*x - 3 * x

def integrate_f(Integrand integrand, double a, double b, int N):
    cdef double s = 0
    cdef double dx = (b - a) / N
    cdef Py_ssize_t i
    for i in range(N):
        s += integrand.f(a + i * dx)
    return s * dx
```
Exploring Cython Further
Handling NumPy Arrays

Example Code

Introduction

From Python to Cython

Handling NumPy Arrays
- Declaring the MemoryView type
- Declaring the Numpy Array type
- Matrix Multiplication
- Our Own MatMul

Parallelization

Wrapping C and C++ Libraries

Handling NumPy Arrays
Declaring the MemoryView type

```python
import numpy as np

def foo(double[:, :, 1] arr):
cdef double[:, :, 1] = np.zeros_like(arr)
cdef size_t i, j
for i in range(arr.shape[0]):
    for j in range(arr.shape[1]):
        out[i, j] = arr[i, j] * i + j

return np.asarray(out)
```
Declaring the Numpy Array type

An alternative to the MemoryView syntax that corresponds more closely with ndarray dtypes:

```python
import numpy as cp
import numpy as np

def foo(arr):
    cdef cp.ndarray[cp.float64, ndim=2] out = np.zeros(arr.shape)
    cdef size_t i, j
    for i in range(arr.shape[0]):
        for j in range(arr.shape[1]):
            arr[i, j] = i + j

    return out
```

Different types are defined in Cython/Includes/numpy.pxd.
Matrix Multiplication

```python
rows_A, cols_A = A.shape[0], A.shape[1]
rows_B, cols_B = B.shape[0], B.shape[1]

out = np.zeros(rows_A, cols_B)

# Take each row in A
for i in range(rows_A):
    # And multiply by each column in B
    for j in range(cols_B):
        s = 0
        for k in range(cols_A):
            s = s + A[i, k] * B[k, j]

        out[i, j] = s
```

Our Own MatMul

We won’t even try this in pure Python (way too slow).

```python
def dot(A, B, out):
    cdef:
        Py_ssize_t rows_A, cols_A, rows_B, cols_B
        Py_ssize_t i, j, k
        double s

    rows_A, cols_A = A.shape[0], A.shape[1]
    rows_B, cols_B = B.shape[0], B.shape[1]

    # Take each row in A
    for i in range(rows_A):
        # And multiply by every column in B
        for j in range(cols_B):
            s = 0
            for k in range(cols_A):
                s = s + A[i, k] * B[k, j]
            out[i, j] = s
```
Parallelization

Example Code

Introduction

From Python to Cython

Handling NumPy Arrays

Parallelization

- Parallel Loops with «prange»

- Wrapping C and C++ Libraries

Parallelization
Parallel Loops with «prange»

```python
@cython.boundscheck(False)
@cython.wraparound(False)

cdef:
    Py_ssize_t rows_A, cols_A, rows_B, cols_B
    Py_ssize_t i, j, k
double s

rows_A, cols_A = A.shape[0], A.shape[1]
rows_B, cols_B = B.shape[0], B.shape[1]

with nogil:
    # Take each row in A
    for i in prange(rows_A):
        # And multiply by every column in B
        for j in range(cols_B):
            s = 0
            for k in range(cols_A):
                s = s + A[i, k] * B[k, j]

out[i, j] = s
```
Benchmark!
Wrapping C and C++ Libraries
We won’t be talking about that here, but Ondrej Certik has some excellent notes:

http://fortran90.org/src/best-practices.html#interfacing-with-python
Create a file, `trig.pyx`, with the following content:

```python
import math

cdef extern from "math.h":
    double cos(double x)
    double sin(double x)
    double tan(double x)

double M_PI

def test_trig():
    print 'Some trig functions from C:','\n    cos(0), cos(M_PI)
```

External Definitions
Build: Link Math Library

```python
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [
        Extension("trig",
            ["trig.pyx"],
            libraries=["m"],
        ),
    ]
)
```
C++ Class Wrapper

```cpp
namespace geom {
    class Circle {
        public:
            Circle(double x, double y, double r);
            ~Circle();
            double getX();
            double getY();
            double getRadius();
            double getArea();
            void setCenter(double x, double y);
            void setRadius(double r);
        private:
            double x;
            double y;
            double r;
    }
}
```
C++ Class Wrapper

Example Code

```
cdef extern from "Circle.h" namespace "geom":
cdef cppclass Circle:
    Circle(double, double, double)
    double getX()
    double getY()
    double getRadius()
    double getArea()
    void setCenter(double, double)
    void setRadius(double)
```
C++ Class Wrapper

cdef class PyCircle:
    cdef Circle *thisptr

def __cinit__(self, double x, double y, double r):
    self.thisptr = new Circle(x, y, r)

def __dealloc__(self):
    del self.thisptr

@property
def area(self):
    return self.thisptr.getArea()

@property
def radius(self):
    return self.thisptr.getRadius()

def set_radius(self, r):
    self.thisptr.setRadius(r)

@property
def center(self):
    return (self.thisptr.getX(), self.thisptr.getY())
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [
        Extension("circ", ["circ.pyx", "Circle.cpp"],
            language="c++"),
        Extension("trig", ["trig.pyx"],
            libraries=['m'],)
    ]
)

C++ Class Wrapper
In conclusion...

- Build functional and tested code
- Profile
- Re-implement bottlenecks (behavior verified by tests)
- Et voilà—high-level code, low-level performance. [It’s no silver bullet, but it’s still pretty good.]