Introduction to Cython: Bridging Python to Compiled Code for High-Performance Computing

Yung-Yu Chen

yyc@solvcon.net

Python Hsinchu User Group

December 3, 2012

What Is Cython?

- Cython generates C code from Python.
- The generated code contains API calls to CPython runtime (VM).
  - Can interface to both the existing C and C++ code.
- Popular use cases:
  - Convert Python code into binary for protection (only a little bit speed-up).
  - Highly-tuned, high-performance C code.
  - Glue Python with C.
- `http://www.cython.org/`.

Installation

- The easiest way is to use your OS' package manager, e.g., under Debian/Ubuntu:

```
> apt-get install cython
```

- Use setuptools (`easy_install`):

```
> easy_install --upgrade cython
```

- I am focusing on the most recent version of Cython:

```
> cython --version
Cython version 0.17.2
```

- If you don't want to mess up your system Python, virtualenv can help you:

```
> virtualenv --system-site-package --setuptools cpython
```

Build Your First Cython Module

- Let's say we have a Python file `example1.py`:

```
def action():
    it = 0
    while it < 1000000:
        it += 1
    print it
```

- It can be compiled by Cython (and gcc):

```
> cp example1.py /tmp/example1c.py
> cython /tmp/example1c.py -o /tmp/example1c.c
> gcc `python-config --includes` -c /tmp/example1c.c -fPIC -
    o /tmp/example1c.o
> gcc `python-config --libs` /tmp/example1c.o -shared -o
    example1c.so
```

- The results are identical:

```
> python -c 'import example1; example1.action()'
> python -c 'import example1c; example1c.action()'
```

Source Code Are Now Hidden

- Python code becomes compiled API calls:

```
> file example1.py example1c.so
example1.py:  ASCII text
example1c.so: ELF 64-bit LSB shared object, ...
```

- But the runtime speed isn't improved much:

```
> python -m timeit -s 'from example1 import
    action' 'action()'
10 loops, best of 3: 49 msec per loop
> python -m timeit -s 'from example1c import
    action' 'action()'
10 loops, best of 3: 42.6 msec per loop
```

- It's still run on interpreter, so the speed-up (42.6 msec vs 49 msec) isn't impressive.

Cython is a Language

- Cython extends Python. It is a superset of Python.
  - When compiling `.py` files, of course only Python syntax can be used.
- If Cython-specific syntax is used, the source code can no longer be run by a vanilla Python VM.
  - It should be saved as a `.pyx` file and compiled by Cython compiler to binary.
- The Cython add-on helps to generate faster code and adapt to various libraries.

Faster Code? Type Information

- Just prefix "`cdef int`" to our counter:

```
def action():
    cdef int it = 0
    while it < 1000000:
        it += 1
    print it
```

and compile

```
> cython example1d.pyx -o /tmp/example1d.c
> gcc `python-config --includes` -c /tmp/example1d.c -fPIC -
    o /tmp/example1d.o
> gcc `python-config --libs` /tmp/example1d.o -shared -o
    example1d.so
```

- Then 16 times faster! (3.01 msec vs 49 msec)

```
> python -m timeit -s 'from example1d import
    action' 'action()'
100 loops, best of 3: 3.01 msec per loop
```

Cython Supports NumPy

- You can use NumPy arrays (example2d.pyx):

```
import numpy as np
def action():
    arr0 = np.empty([1000,1000], dtype='float64')
    arr0.fill(0)
    arr1 = np.empty([1000,1000], dtype='float64')
    arr1.fill(1)
    cdef int it = 1
    cdef int jt
    while it < 999:
        jt = 1
        while jt < 999:
            arr0[it, jt] += arr1[it-1, jt-1]
            arr0[it, jt] += arr1[it-1, jt  ]
            arr0[it, jt] += arr1[it-1, jt+1]
            arr0[it, jt] += arr1[it  , jt+1]
            arr0[it, jt] += arr1[it+1, jt+1]
            arr0[it, jt] += arr1[it+1, jt  ]
            arr0[it, jt] += arr1[it+1, jt-1]
            arr0[it, jt] += arr1[it  , jt-1]
            jt += 1
        it += 1
    assert 7968032 == arr0.sum()
```

Add Array Dimensions

- Let Cython use direct indexing (example2a.pyx):

```
import numpy as np
cimport numpy as cnp
def action():
    cdef cnp.ndarray[cnp.double_t, ndim=2] arr0 = np.empty(
        [1000,1000], dtype='float64')
    arr0.fill(0)
    cdef cnp.ndarray[cnp.double_t, ndim=2] arr1 = np.empty(
        [1000,1000], dtype='float64')
    arr1.fill(1)
    cdef int it = 1
    cdef int jt
    while it < 999:
        jt = 1
        while jt < 999:
            arr0[it, jt] += arr1[it-1, jt-1]
            arr0[it, jt] += arr1[it-1, jt  ]
            arr0[it, jt] += arr1[it-1, jt+1]
            arr0[it, jt] += arr1[it  , jt+1]
            arr0[it, jt] += arr1[it+1, jt+1]
            arr0[it, jt] += arr1[it+1, jt  ]
            arr0[it, jt] += arr1[it+1, jt-1]
            arr0[it, jt] += arr1[it  , jt-1]
            jt += 1
        it += 1
    assert 7968032 == arr0.sum()
```

Direct Indexing Is Way Faster

- Use NumPy API for indexing: 5.74 sec.

```
> python -m timeit -s 'from example2d import
    action' 'action()'
10 loops, best of 3: 5.74 sec per loop
```

- Direct indexing: 139 msec.

```
> python -m timeit -s 'from example2a import
    action' 'action()'
10 loops, best of 3: 139 msec per loop
```

- Direct indexing makes the code run 41 times faster.

Turn off Bound-Checking

- Get a little speed-up (example2b.pyx): 110 msec.

```
import numpy as np
cimport numpy as cnp
cimport cython
@cython.boundscheck(False)
def action():
    cdef cnp.ndarray[cnp.double_t, ndim=2] arr0
        = np.empty(
          [1000,1000], dtype='float64')
    arr0.fill(0)
    cdef cnp.ndarray[cnp.double_t, ndim=2] arr1
        = np.empty(
          [1000,1000], dtype='float64')
    arr1.fill(1)
    ...
```

We Just Want to Call C

- Let's say we have a C function (example2.c):

```
void caction(double arr0[][1000], double arr1
    [][1000]) {
     for (int it=1; it<999; it++) {
          for (int jt=1; jt<999; jt++) {
               arr0[it][jt] += arr1[it-1][jt-1];
               arr0[it][jt] += arr1[it-1][jt  ];
               arr0[it][jt] += arr1[it-1][jt+1];
               arr0[it][jt] += arr1[it  ][jt+1];
               arr0[it][jt] += arr1[it+1][jt+1];
               arr0[it][jt] += arr1[it+1][jt  ];
               arr0[it][jt] += arr1[it+1][jt-1];
               arr0[it][jt] += arr1[it  ][jt-1];
          };
     };
};
```

Cython Let You Call C

- example2c.pyx:

```
import numpy as np
cimport numpy as cnp
cdef extern:
    void caction(double* arr0, double* arr1)
def action():
    cdef cnp.ndarray[double, ndim=2] arr0 = np.
        empty(
         [1000,1000], dtype='float64')
    arr0.fill(0)
    cdef cnp.ndarray[double, ndim=2] arr1 = np.
        empty(
         [1000,1000], dtype='float64')
    arr1.fill(1)
    caction(&arr0[0,0], &arr1[0,0])
    assert 7968032 == arr0.sum()
```

Side Effect: Even Faster

- The C version is 51.4 msec, the fastest.

```
> python -m timeit -s 'from example2c import
    action' 'action()'
10 loops, best of 3: 51.4 msec per loop
```

- Compare to the original version, 5.74 sec / 51.4 msec = 106 times faster.
- Compare to the pure-Cython version, 110 msec / 51.4 msec = 2 times faster.
- C is the king.
  - The rule of thumb: Python is two orders of magnitude slower than C.
  - But after we replace the Python hotspot with C, it's OK.

Useful Syntax for Wrapping

- `cdef extern from "header.h":`
  - Allows Cython to check the declarations from outside headers.
- `cdef public:`
  - Allows Cython to generates C header files for inclusion in other C code.

Build Cython Code on the Fly

- For simple code, Cython can compile it on the fly by using "pyximport":

```
import pyximport
pyximport.install()
```

- Then standard `import` gets the `pyx` file compiled:

```
import example2b
```

- After that the code is ready for use:

```
example2b.action()
```

Use Cython's Distutils Helper

- Prepare a setup.py file:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
ext_modules = [Extension("example2s", ["
    example2b.pyx"])]
setup(
    name = 'example2',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules)
```

Build and Run the Extension

- Run the `setup.py`:

```
> python setup.py build_ext --inplace
```

- We then can use the built module:

```
> python -c 'from example2s import action;
  action()'
```

Distribute Compiled C Files

- Prepare the C file to be distributed:

```
> cython example2p.pyx -o example2p.c
```

- Prepare another `setup2.py` file:

```
from distutils.core import setup
from distutils.extension import Extension
setup(
    name = 'example2',
    ext_modules = [Extension("example2p", ["
        example2p.c"])])
```

Build and Run the Extension with Cython

- Run the `setup2.py`:

```
> python setup2.py build_ext --inplace
```

  But no Cython is needed at building.

- We then can use the built module:

```
> python -c 'from example2p import action;
    action()'
```

What Cython Is Good At

- Hide the Python source code by turning it into C.
  - Even byte code disappears. Only binary remains.
- By supplying type information, provide an order of magnitude of speed-up.
- When working with NumPy arrays, Cython supports direct indexing that boosts the performance for tens of times.
- Wrap self-written or 3rd-party libraries with great ease.

What You Shouldn't Do with Cython

- Write a complex class hierarchy and expect it to run very fast.
  - If you need high-performance code based on a complex class hierarchy, you should use C++ and wrap it around with Boost.Python, Cython, or something else.

Extended Resources

- Cython
- Boost.Python
- SWIG
- Distributing Python Modules
- Python C/API Reference Manual