Speeding up Python with Cython

Kurt W. Smith
(presented by Robert Grant)
Enthought, Inc.

Cython by example

PYTHON

```
def fib(n):
    a,b = 0,1
    for i in range(n):
        a, b = a+b, a
    return a
```

C / C++

```
int fib(int n)
{
    int tmp, i, a=0, b=1;
    for(i=0; i<n; i++) {
        tmp = a; a += b; b = tmp;
    }
    return a;
}
```

Cython by example

PYTHON

```
def fib(n):
    a,b = 0,1
    for i in range(n):
        a, b = a+b, a
    return a
```

C / C++

```
int fib(int n)
{
    int tmp, i, a=0, b=1;
    for(i=0; i<n; i++) {
        tmp = a; a += b; b = tmp;
    }
    return a;
}
```

CYTHON

```
def fib(int n):
    cdef int i, a, b
    a,b = 0,1
    for i in range(n):
        a, b = a+b, a
    return a
```

Cython by example

PYTHON 1x

```
def fib(n):
    a,b = 0,1
    for i in range(n):
        a, b = a+b, a
    return a
```

C / C++ 70x faster

```
int fib(int n)
{
    int tmp, i, a=0, b=1;
    for(i=0; i<n; i++) {
        tmp = a; a += b; b = tmp;
    }
    return a;
}
```

CYTHON 70x faster

```
def fib(int n):
    cdef int i, a, b
    a,b = 0,1
    for i in range(n):
        a, b = a+b, a
    return a
```

For the record...

HAND-WRITTEN EXTENSION MODULE

```
#include "Python.h"

static PyObject* fib(PyObject *self, PyObject *args)
{
    int n, a=0, b=1, i, tmp;
    if (!PyArg_ParseTuple(args, "i", &n))
        return NULL;
    for (i=0; i<n; i++) {
        tmp=a; a+=b; b=tmp;
    }
    return Py_BuildValue("i", a);
}

static PyMethodDef ExampleMethods[] = {
    {"fib", fib, METH_VARARGS, ""},
    {NULL, NULL, 0, NULL}        /* Sentinel */
};

PyMODINIT_FUNC
initfib(void)
{
    (void) Py_InitModule("fib", ExampleMethods);
}
```

For the record...

HAND-WRITTEN EXTENSION MODULE — 40x faster

```
#include "Python.h"

static PyObject* fib(PyObject *self, PyObject *args)
{
    int n, a=0, b=1, i, tmp;
    if (!PyArg_ParseTuple(args, "i", &n))
        return NULL;
    for (i=0; i<n; i++) {
        tmp=a; a+=b; b=tmp;
    }
    return Py_BuildValue("i", a);
}

static PyMethodDef ExampleMethods[] = {
    {"fib", fib, METH_VARARGS, ""},
    {NULL, NULL, 0, NULL}        /* Sentinel */
};

PyMODINIT_FUNC
initfib(void)
{
    (void) Py_InitModule("fib", ExampleMethods);
}
```

What is Cython?

Cython is a Python-like language that:

- **Improves Python's performance:** 1000x speedups not uncommon
- **wraps external code:** C, C++, Fortran, others...

The `cython` command:

- generates optimized C or C++ from Cython source,
- the C/C++ source is then compiled into a Python extension module.

Other features:

- built-in support for NumPy,
- integrates with IPython,
- Combine C's performance with Python's ease of use.

`http://www.cython.org/`

Cython in the wild

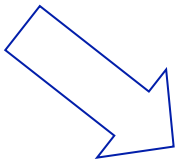
Project	Cython files	KLOC
sage	761	420
numpy	14	5.6
scipy	28	19
pandas	21	24
sympy	12	12 (cythonized python)
scikits-learn	35	10
scikits-image	48	10
mpi4py	48	12

Projects master branches as of July 1, 2013

Speed up Python

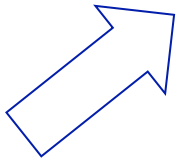
PYTHON

```
def fib(n):
    a,b = 0,1
    for i in range(n):
        a, b = a+b, a
    return a
```



CYTHON

```
def fib(int n):
    cdef int i, a, b
    a,b = 0,1
    for i in range(n):
        a, b = a+b, a
    return a
```



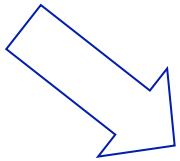
GENERATED C

```
static PyObject
*__pyx_pf_5cyfib_cyfib(PyObject *__pyx_self,
int __pyx_v_n) {
  int __pyx_v_a; int __pyx_v_b;
PyObject *__pyx_r = NULL; PyObject *__pyx_t_5
= NULL;
const char *__pyx_filename = NULL;
...
  for (__pyx_t_1=0; __pyx_t_1<__pyx_t_2;
__pyx_t_1+=1) {
    __pyx_v_i = __pyx_t_1;
    __pyx_t_3 = (__pyx_v_a + __pyx_v_b);
    __pyx_t_4 = __pyx_v_a;
    __pyx_v_a = __pyx_t_3;
    __pyx_v_b = __pyx_t_4;
  }
...
}
```

Wrap C / C++

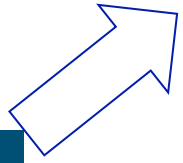
C / C++

```
int fact(int n)
{
    if (n <= 1)
      return 1;
    return n * fact(n-1);
}
```



CYTHON

```
cdef extern from "fact.h":
    int _fact "fact"(int)

def fact(int n):
    return _fact(n)
```



GENERATED WRAPPER

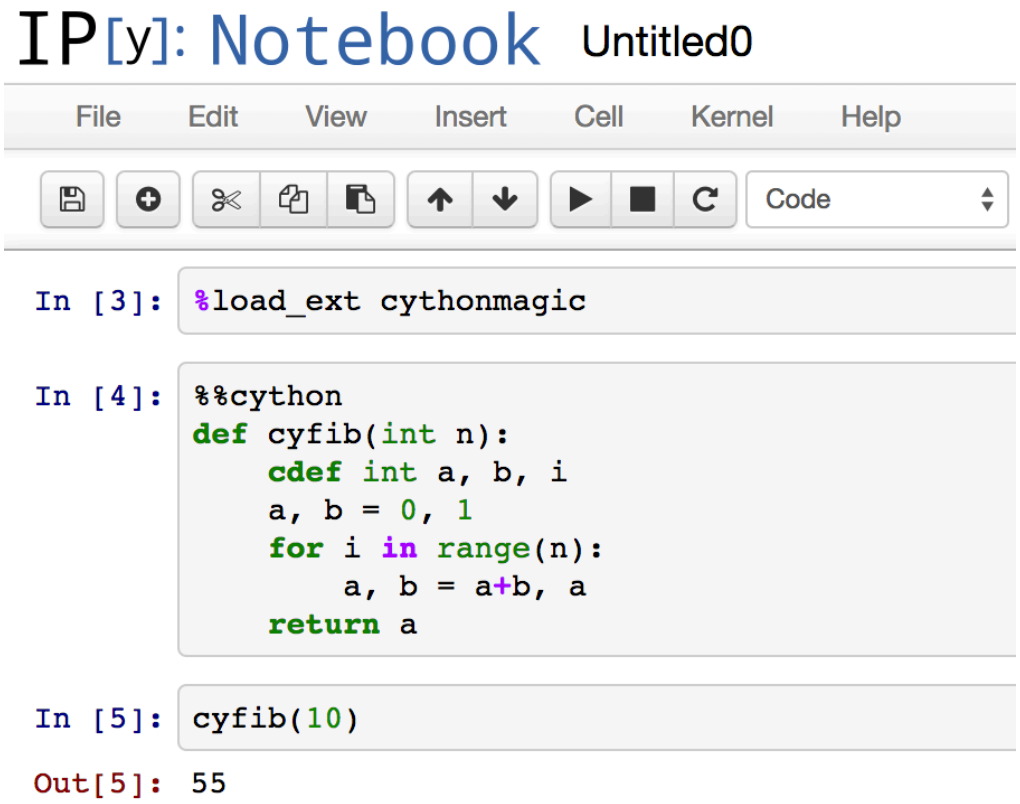
```
static PyObject
*__pyx_pf_5cyfib_cyfib(PyObject *__pyx_self,
int __pyx_v_n) {
  int __pyx_v_a; int __pyx_v_b;
PyObject *__pyx_r = NULL; PyObject *__pyx_t_5
= NULL;
const char *__pyx_filename = NULL;
...
  for (__pyx_t_1=0; __pyx_t_1<__pyx_t_2;
__pyx_t_1+=1) {
    __pyx_v_i = __pyx_t_1;
    __pyx_t_3 = (__pyx_v_a + __pyx_v_b);
    __pyx_t_4 = __pyx_v_a;
    __pyx_v_a = __pyx_t_3;
    __pyx_v_b = __pyx_t_4;
  }
...
}
```

Cython + IPython

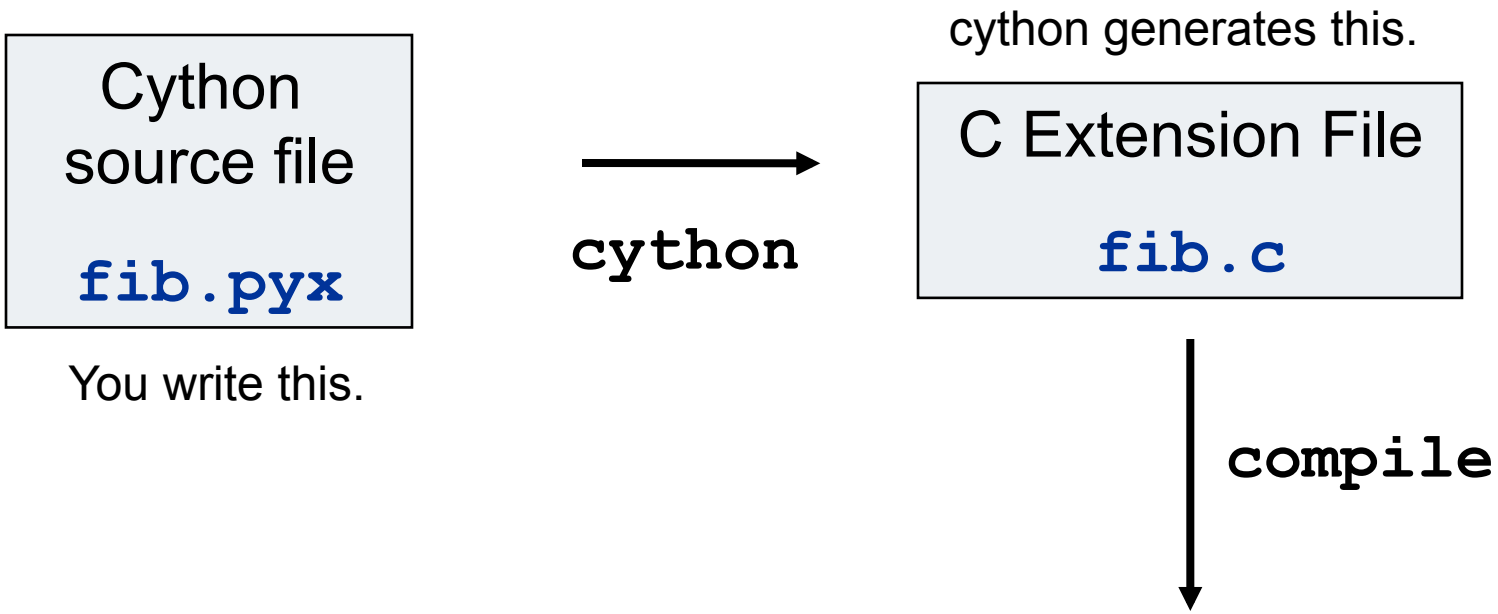
IPython provides cython magic commands, the most useful of which is **`%%cython`**.

IPYTHON / IPYTHON NOTEBOOK

```
In [10]: %load_ext cythonmagic

In [11]: %%cython
    ....: def cyfib(int n):
    ....:     cdef int a, b, i
    ....:     a, b = 0, 1
    ....:     for i in range(n):
    ....:         a, b = a+b, a
    ....:     return a
    ....:

In [12]: cyfib(10)
Out[12]: 55
```



Cython `pyx` files



Compiling with `distutils`

FIB.PYX

```
# Define a function. Include type information for the argument.
def fib(int n):
    ...
```

SETUP.PY

```
# Cython has its own "extension builder" module that knows how
# to build cython files into python modules.
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext = Extension("fib", sources=["fib.pyx"])

setup(ext_modules=[ext],
      cmdclass={'build_ext': build_ext})
```

Compiling an extension module

CALLING FIB FROM PYTHON

```
# Mac / Linux
$ python setup_fib.py build_ext --inplace

# Windows
$ python setup_fib.py build_ext --inplace -c mingw32

$ python
>>> import fib
>>> fib.fib()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 1 argument (0 given)
>>> fib.fib("dsa")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: an integer is required
>>> fib.fib(3)
2
```

pyximport

pyximport: import a Cython source file as if it is a pure Python module.

- Detects changes in Cython file, recompiles if necessary, loads cached module if not.
- Great for simple cases.

RUN_FIB.PY

```
import pyximport
pyximport.install() # hooks into Python's import mechanism.

from fib import fib # finds fib.pyx, automatically compiles.

print fib(10)
```

`cdef`: declare C-level object

LOCAL VARIABLES

```
def fib(int n):
    cdef int a, b, i
    ...
```

EXTENSION TYPES

```
cdef class Particle(object):
    cdef float psn[3], vel[3]
    cdef int id
```

C FUNCTIONS

```
cdef float distance(float *x,
                    float *y,
                    int n):
    cdef:
        int i
        float d = 0.0
    for i in range(n):
        d += (x[i] - y[i])**2
    return d
```

Typed function arguments are declared without **`cdef`**.

def, cdef, cpdef

DEF FUNCTIONS: AVAILABLE TO PYTHON + CYTHON

```
def distance(x, y):
    return np.sum((x-y)**2)
```

CDEF FUNCTIONS: FAST, LOCAL TO CURRENT FILE

```
cdef float distance(float *x, float *y, int n):
    cdef:
        int i
        float d = 0.0
    for i in range(n):
        d += (x[i] - y[i])**2
    return d
```

CPDEF FUNCTIONS: LOCALLY C, EXTERNALLY PYTHON

```
cpdef float distance(float[:] x, float[:] y):
    cdef int i
    cdef int n = x.shape[0]
    cdef float d = 0.0
    for i in range(n):
        d += (x[i] - y[i])**2
    return d
```

Profiling with annotations

FIB_ORIG.PYX: NO CDEFS

```
def fib(n):
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

CREATE ANNOTATED SOURCE

```
$ cython -a fib_orig.pyx
$ open fib_orig.html
```

FIB_ORIG.HTML

Raw output: fib_orig.c

```
1: def fib(n):
2:     a,b = 1,1
3:     for i in range(n):
4:         a, b = a+b, a
5:     return a
```

The darker the highlighting, the more lines of C code are required for the given line of Cython code.

Profiling with annotations

Raw output: fib_orig.c

```
1: def fib(n):
2:     a,b = 1,1
3:     for i in range(n):
4:         a, b = a+b, a
```

```
    /* "fib_orig.pyx":4
*     a,b = 1,1
*     for i in range(n):
*         a, b = a+b, a               # <<<<<<<<<<<<<<
*     return a
*/
    __pyx_t_1 = PyNumber_Add(__pyx_v_a, __pyx_v_b); if (unlikely
    __Pyx_GOTREF(__pyx_t_1);
    __pyx_t_5 = __pyx_v_a;
    __Pyx_INCREF(__pyx_t_5);
    __Pyx_DECREF(__pyx_v_a);
    __pyx_v_a = __pyx_t_1;
    __pyx_t_1 = 0;
    __Pyx_DECREF(__pyx_v_b);
    __pyx_v_b = __pyx_t_5;
    __pyx_t_5 = 0;
  }
  __Pyx_DECREF(__pyx_t_2); __pyx_t_2 = 0;
```

```
5:     return a
```

Profiling with annotations

FIB.PYX: WITH CDEFS

```
def fib(int n):
    cdef int i, a, b
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

CREATE ANNOTATED SOURCE

```
$ cython -a fib.pyx
$ open fib.html
```

FIB.HTML

Raw output: fib.c

```
1: def fib(int n):
2:     cdef int a, b, i
3:     a, b = 1, 1
4:     for i in range(n):
5:         a, b = a+b, a
6:     return a
```

`cimport`: access C stdlib functions

```
# uses Python's sin implementation
# Incurs Python overhead when calling
from math import sin as pysin

# NumPy's sin ufunc: fast for arrays, slower for scalars
from numpy import sin as npsin

# uses C stdlib's sin from math.h: no Python overhead
from libc.math cimport sin

# other headers are supported
from libc.stdlib cimport malloc, free
```

`cimport` and `pxd` files

To reuse Cython code in multiple files, create a **`pxd`** file of declarations for a corresponding **`pyx`** file and **`cimport`** it elsewhere.

FIB.PXD – LIKE C HEADER FILE

```
cdef int fib(int n)
```

FIB.PYX – LIKE C IMPL. FILE

```
cdef int fib(int n):
    cdef int a, b, i
    a, b = 1, 1
    for i in range(n):
        a, b = a+b, a
    return a
```

USES_FIB.PYX

```
# Access fib() in fib.pyx
from fib cimport fib

def uses_fib(int n):
    print "calling fib(%d)" % n
    res = fib(n)
    print "fib(%d) = %d" % (n, res)
    return res
```

PXD FILES PROVIDED WITH CYTHON

```
from libc.stdlib cimport malloc, free # C std library
cimport numpy as cnp # numpy C-API
from libcpp.vector cimport vector # C++ std::vector
```

Wrapping C

TIME_EXTERN.PYX

```
cdef extern from "time.h":
    # Declare only what is necessary from `tm` structure.
    struct tm:
        int tm_mday # Day of the month: 1-31
        int tm_mon  # Months *since* january: 0-11
        int tm_year # Years since 1900

    ctypedef long time_t
    tm* localtime(time_t *timer)
    time_t time(time_t *tloc)

def get_date():
    """ Return a tuple with the current day, month, and year."""
    cdef time_t t
    cdef tm* ts
    t = time(NULL)
    ts = localtime(&t)
    return ts.tm_mday, ts.tm_mon + 1, ts.tm_year
```

CALLING FROM PYTHON

```
>>> extern_time.get_date()
(8, 4, 2011)
```

Python classes, extension types

PYTHON / CYTHON PARTICLE CLASS

```
class Particle(object):  # Inherits from object; can use multiple inh.

  def __init__(self, m, p, v): # attributes stored in instance __dict__
    self.m = float(m)   # creating / updating attribute allowed anywhere.
    self.vel = np.asarray(v) # All attributes are Python objects.
    self.pos = np.asarray(p)

  def apply_impulse(self, f, t): # can be defined in or out of class.
    newv = self.vel + t / self.m * f
    self.pos = (newv + self.vel) * t / 2.
    self.vel = newv

  def speed(self):
    ...
```

Python classes, extension types

PARTICLE EXTENSION TYPE IN CYTHON

```
cdef class Particle:          # Creates a new type, like list, int, dict
  cdef float *vel, *pos       # attributes stored in instance's struct
  cdef public float m         # expose variable to Python.

  def __cinit__(self, float m, p, v): # allocate C-level data,
    self.m = m                        # called before __init__()
    self.vel = malloc(3*sizeof(float))
    self.pos = malloc(3*sizeof(float))
    # check if vel or pos are NULL...
    for i in range(3):
        self.vel[i] = v[i]; self.pos[i] = p[i]

  cpdef apply_impulse(self, f, t): # methods can be def, cdef, or cpdef.
    ...
  def __dealloc__(self): # deallocate C arrays, called when gc'd.
    if self.vel: free(self.vel)
    if self.pos: free(self.pos)
```

Cython, NumPy, memoryviews

Typed memoryviews allow efficient access to memory buffers (such as NumPy arrays) without any Python overhead.

TYPED MEMORYVIEWS

```
def sum(double[::1] a):  # a: contiguous 1D buffer of doubles.
    cdef double s = 0.0
    cdef int i, n = a.shape[0]
    for i in range(n):
        s += a[i]
    return s
```

USE JUST LIKE NUMPY ARRAYS

```
In[1]: from mysum import sum
In[2]: a = arange(1e6)
In[3]: %timeit sum(a)
1000 loops, best of 3: 998 us per loop
In[4]: %timeit a.sum()
1000 loops, best of 3: 991 us per loop
```

Cython, NumPy, memoryviews

ACQUIRING BUFFERS

```
cdef int[:, :, :] mv # a 3D typed memoryview, can be assigned to...

# 1: a C-array:
cdef int a[3][3][3]

# 2: a NumPy-array:
a = np.zeros((10,20,30), dtype=np.int32)

# 3: another memoryview
cdef int[:, :, :] a = b
```

USING MEMORYVIEWS

```
# indexing like NumPy, but faster, at C-level.
mv[1,2,0] # → integer

# Slicing like NumPy, but faster.
mv[10] == mv[10, :, :] == mv[10,...] # → a new memoryview.
```

Cython, NumPy, memoryviews

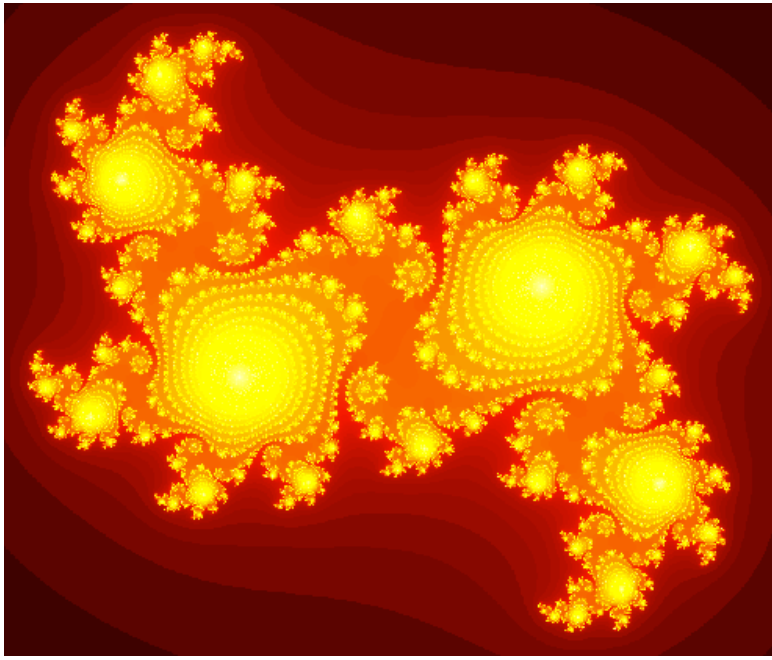
STRIDED AND CONTIGUOUS MEMORYVIEWS

```
# uses strided lookup when indexing
cdef int[:, :, :] strided_mv

# can acquire buffer from a non-contiguous np array.
strided_mv = arr[::2, 5:, ::-1]

# faster than strided, but only works with C-contiguous buffers.
cdef int[:, :, ::1] c_contig

c_contig = np.zeros((10, 20, 30), dtype=np.int)

c_contig = arr[:, :, :5] # non-contiguous, so ValueError at runtime.

# faster than strided, only works with Fortran-contiguous.
cdef int[::1, :, :] f_contig

f_contig = np.asfortranarray(arr)
```

Capstone demo: compute Julia set

PURE-PYTHON VERSION

```
# julia_pure_python.py
def kernel(z, c, lim):
  count = 0
  while abs(z) < lim:
    z = z * z + c
    count += 1
  return count

def compute_julia(cr, ci, N, bound, lim):
  julia = np.empty((N, N), dtype=np.uint32)
  grid_x = np.linspace(-bound, bound, N)
  grid_y = grid_x * 1j
  c = cr + 1j * ci
  for i, x in enumerate(grid_x):
    for j, y in enumerate(grid_y):
      julia[i,j] = kernel(x+y, c, lim)
  return julia
```



Add Type Information

```
def abs_sq(float zr, float zi):
    ...
def kernel(float zr, float zi,
           float cr, float ci,
           float lim, double cutoff):
    cdef int count
    ...
def compute_julia(float cr, float ci, int N,
                  float bound, float lim, double
  cutoff):
    ...
```

Use Cython C Functions

```
cdef float abs_sq(...):
    ...
cdef int kernel(...):
    ...
```

Use typed memoryviews

```
def compute_julia(...):
  cdef int[:,::1] julia # 2D, C-contiguous.
  cdef float[::1] grid  # 1D, C-contiguous.
  ...
  julia = empty((N,N), dtype=int32)
  grid = array(linspace(...), dtype=float32)
  # all array accesses and assignments faster.
```

Add Cython directives

```
cimport cython
...

# don't check for out-of-bounds indexing.
@cython.boundscheck(False)
# assume no negative indexing.
@cython.wraparound(False)
def compute_julia(...):
    ...
```

Parallelization using OpenMP

```
from cython.parallel cimport prange

cdef float abs_sq(...) nogil:
    ...

cdef int kernel(...) nogil:
    ...

def compute_julia_parallel(...):
    ...
    # release the GIL and run in parallel.
    for i in prange(N, nogil=True):
        ...
```

Parallelization using OpenMP

```
Extension("julia_cython", ["julia_cython.pyx"],
          extra_compile_args=["-fopenmp"],
          extra_link_args=["-fopenmp"])
```

Conclusion

Solution	Time (s)	Speed-up
Pure Python	630	x 1
Cython (Step 1)	2.8	x 230
Cython (Step 2)	2.0	x 320
Cython+Numpy (Step 3)	0.40	x 1600
Cython+Numpy+prange (Step 4)	0.24	x 2600

Timing performed on a 2.3 GHz Intel Core i7 MacBook Pro with 8GB RAM using a 2000x2000 array.

[July 20, 2012]