02_Speeding_Python

September 26, 2014

1 Speeding Python

2 Python in HPC Tutorial

2.1 Supercomputing 2014

Matt Knepley and Aron Ahmadia

2.2 About This Tutorial

2.2.1 PyHPC Tutorial on GitHub

These presentation materials are part of a continuously updated tutorial on Python for High Performance Computing. Future versions of this presentation will be found at:

https://github.com/pyhpc/pyhpc-tutorial

Please set all permanent bookmarks to this URL.

2.2.2

Checkout from git

```
git clone https://github.com/pyhpc/pyhpc-tutorial.git
git checkout sahpc2012
```

2.2.3 Viewing the read-only version of this presentation on nbviewer:

- http://nbviewer.ipython.org/urls/raw.github.com/pyhpc/pyhpc-tutorial/master/notebooks/02_Speeding_Python.ipynb

---

2.3 Interacting with the Tutorial Slides

This tutorial is an interactive worksheet designed to encourage you to try out the lessons during the demonstration. If you are looking at the PDF version of these slides, we encourage you to download the updated version (see previous slide) and try the interactive version.

To run the interactive version of this notebook, you will need a Python 2.7 environment including:

- IPython version >= 13.0
- numpy version >= 1.6

- scipy >= 0.10
- matplotlib >= 1.0.0

Move to the directory containing the tarball and execute:

```
$ ipython notebook --pylab=inline
```

If you are installing the packages yourself, Continuum Analytics provides both free community as well as professional versions of the Anaconda installer, which provides all the packages you will need for this portion of the tutorial. The installer is available from the Anaconda download page at Continuum Analytics.

You are also welcome to use Enthought's Python Distribution (free for Academic users), available from the EPD download page at Enthought.

2.4 How Slow is Python

Let's add one to a million number

```
In [2]: lst = range(1000000) # A pure Python list
        %timeit [i + 1 for i in lst]
        # A Python list comprehension (iteration happens in C but with PyObjects)
```

```
1 loops, best of 3: 194 ms per loop
```

2.5 Why is Python Slow?

Dynamic typing requires lots of metadata around variable.

- Python uses heavy frame objects during iteration

2.5.1 Solution:

- Make an object that has a single type and continuous storage.
- Implement common functionality into that object to iterate in C.

```
In [3]: arr = np.arange(1000000) # A NumPy list of integers
        %timeit arr + 1 # Use operator overloading for nice syntax, now iteration is in C with ints
```

```
100 loops, best of 3: 7.45 ms per loop
```

2.6 What makes NumPy so much faster?

- Data layout
- homogenous: every item takes up the same size block of memory
- single data-type objects
- powerful array scalar types
- universal function (ufuncs)
- function that operates on ndarrays in an element-by-element fashion
- vectorized wrapper for a function
- built-in functions are implemented in compiled C code

2.7 Numpy Data layout

- homogenous: every item takes up the same size block of memory
- single data-type objects
- powerful array scalar types

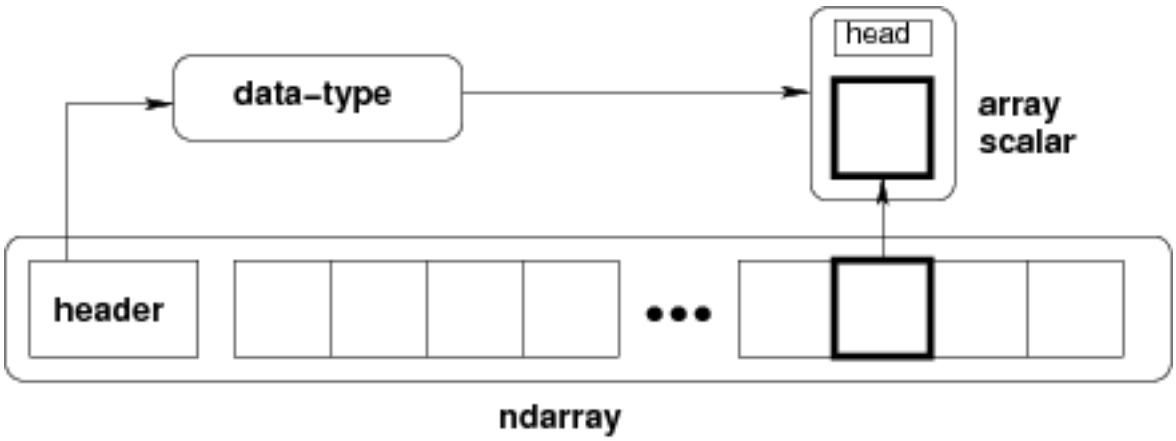


Figure 1: three fundamental

2.8 Numpy Universal Functions (ufuncs)

- function that operates on ndarrays in an element-by-element fashion
- vectorized wrapper for a function
- built-in functions are implemented in compiled C code

```
In [4]: %timeit [sin(i)**2 for i in arr]

1 loops, best of 3: 9.08 s per loop

In [5]: %timeit np.sin(arr)**2

10 loops, best of 3: 58.6 ms per loop
```

2.9 Other Numpy Features to be aware of

- Reshaping

```
In [6]: arr2 = arr.reshape((10,100000))
        print(arr2)

[[     0      1      2 ...,  99997  99998  99999]
 [100000 100001 100002 ..., 199997 199998 199999]
 [200000 200001 200002 ..., 299997 299998 299999]
 ...,
 [700000 700001 700002 ..., 799997 799998 799999]
 [800000 800001 800002 ..., 899997 899998 899999]
 [900000 900001 900002 ..., 999997 999998 999999]]
```

- Memory Views

```
In [7]: arr2.view?
```

- Index Slicing

```
In [8]: x = np.arange(0, 20, 2); y = x**2
        ((y[1:] - y[:-1]) / (x[1:] - x[:-1])) # dy/dx
```

```
Out[8]: array([ 2,  6, 10, 14, 18, 22, 26, 30, 34])
```

- Fancy Indexing

```
In [9]: evens = arr[arr%2 == 0]
        print(evens)
```

```
[     0      2      4 ..., 999994 999996 999998]
```

2.10 Compiling to C

C is faster, and Python is easier to write. We want both!

2.11 Cython

- a programming language based on Python
- uses extra syntax allowing for optional static type declarations
- source code gets translated into optimized C/C++ code and compiled as Python extension modules

---

2.12 Using Cython in IPython

In IPython we can make any cell call out to Cython via the cell magic. The user must first install Cython

```
$ pip install cython
```

Then load the extension

```
In [7]: %load_ext cythonmagic
```

Now use `%%cython` at the begining of a code cell to call out to Cython.

```
In [8]: %%cython
        def f_cython(int i):
            return i**4 + 3*i**2 + 10
```

Now use Cython function in code:

```
In [9]: f_cython(100)
```

```
Out[9]: 100030010
```

2.13 How much faster is Cython?

The more you are able to provide type information the better the compile. For example f without type information:

```
In [10]: %%cython
         def f_slow(i):
             return i**4 + 3*i**2 + 10

In [11]: %timeit f_slow(100)

1000000 loops, best of 3: 356 ns per loop

In [12]: %timeit f_cython(100)

10000000 loops, best of 3: 121 ns per loop
```

2.14 Declaring Cython variables for C level

If you use a variable or function only at the Cython level you can keep it in C via `cdef`:

```
In [13]: %%cython
         cdef f(double x):
             return x**2-x

         def integrate_f(double a, double b, int N):
             cdef int i
             cdef double s, dx
             s = 0
             dx = (b-a)/N
             for i in range(N):
                 s += f(a+i*dx)
             return s * dx

In [14]: %timeit integrate_f(1.0, 2.0, 1000)

10000 loops, best of 3: 57.1 μs per loop
```

The pure Python version:

```
In [15]: def f(x):
             return x**2-x

         def integrate_f(a, b, N):
             s = 0
             dx = (b-a)/N
             for i in range(N):
                 s += f(a+i*dx)
             return s * dx

In [16]: %timeit integrate_f(1.0, 2.0, 1000)

1000 loops, best of 3: 604 μs per loop
```

2.15 Using NumPy with Cython

You can also use fast accessors to NumPy arrays from Cython:

```
In [17]: %%cython
         import numpy as np
         # "cimport" is used to import special compile-time information
         # about the numpy module (this is stored in a file numpy.pxd which is
         # currently part of the Cython distribution).
         cimport numpy as np
         # We now need to fix a datatype for our arrays. I've used the variable
         # DTYPE for this, which is assigned to the usual NumPy runtime
         # type info object.
         DTYPE = np.int
         # "ctypedef" assigns a corresponding compile-time type to DTYPE_t. For
         # every type in the numpy module there's a corresponding compile-time
         # type with a _t-suffix.
         ctypedef np.int_t DTYPE_t
         # "def" can type its arguments but not have a return type. The type of the
         # arguments for a "def" function is checked at run-time when entering the
         # function.
         #
         # The arrays f, g and h is typed as "np.ndarray" instances. The only effect
         # this has is to a) insert checks that the function arguments really are
         # NumPy arrays, and b) make some attribute access like f.shape[0] much
         # more efficient. (In this example this doesn't matter though.)
         def naive_convolve(np.ndarray f, np.ndarray g):
             if g.shape[0] % 2 != 1 or g.shape[1] % 2 != 1:
                 raise ValueError("Only odd dimensions on filter supported")
             assert f.dtype == DTYPE and g.dtype == DTYPE
             # The "cdef" keyword is also used within functions to type variables. It
             # can only be used at the top indendation level (there are non-trivial
             # problems with allowing them in other places, though we'd love to see
             # good and thought out proposals for it).
             #
             # For the indices, the "int" type is used. This corresponds to a C int,
             # other C types (like "unsigned int") could have been used instead.
             # Purists could use "Py_ssize_t" which is the proper Python type for
             # array indices.
             cdef int vmax = f.shape[0]
             cdef int wmax = f.shape[1]
             cdef int smax = g.shape[0]
             cdef int tmax = g.shape[1]
             cdef int smid = smax // 2
             cdef int tmid = tmax // 2
             cdef int xmax = vmax + 2*smid
             cdef int ymax = wmax + 2*tmid
             cdef np.ndarray h = np.zeros([xmax, ymax], dtype=DTYPE)
             cdef int x, y, s, t, v, w
             # It is very important to type ALL your variables. You do not get any
             # warnings if not, only much slower code (they are implicitly typed as
             # Python objects).
             cdef int s_from, s_to, t_from, t_to
             # For the value variable, we want to use the same data type as is
             # stored in the array, so we use "DTYPE_t" as defined above.
```

```
            # NB! An important side-effect of this is that if "value" overflows its
            # datatype size, it will simply wrap around like in C, rather than raise
            # an error like in Python.
            cdef DTYPE_t value
            for x in range(xmax):
                for y in range(ymax):
                    s_from = max(smid - x, -smid)
                    s_to = min((xmax - x) - smid, smid + 1)
                    t_from = max(tmid - y, -tmid)
                    t_to = min((ymax - y) - tmid, tmid + 1)
                    value = 0
                    for s in range(s_from, s_to):
                        for t in range(t_from, t_to):
                            v = x - smid + s
                            w = y - tmid + t
                            value += g[smid - s, tmid - t] * f[v, w]
                    h[x, y] = value
            return h
```

```
In [18]: N=100
         f = np.arange(N*N, dtype=np.int).reshape((N,N))
         g = np.arange(81, dtype=np.int).reshape((9, 9))
         %timeit -n2 -r3 naive_convolve(f, g)
```

```
2 loops, best of 3: 2.86 s per loop
```

---

3 Numba – A Python Compiler for Numpy arrays

The user must install the Numba packages. If you're not using anaconda, you will need LLVM with RTTI enabled (See https://github.com/llvmpy/llvmpy for the most up-to-date instructions)

First compile LLVM 3.3

```
$ wget http://llvm.org/releases/3.3/llvm-3.3.src.tar.gz
$ tar zxvf llvm-3.3.src.tar.gz
$ cd llvm-3.3.src
$ ./configure --enable-optimized --prefix=LLVM_BUILD_DIR
$ # It is recommended to separate the custom build from the default system
$ # package.
$ # Be sure your compiler architecture is same as version of Python you will use
$ #  e.g. -arch i386 or -arch x86_64.  It might be best to be explicit about this.
$ REQUIRES_RTTI=1 make install
```

Then install llvmpy and numba

```
$ LLVM_CONFIG_PATH=LLVM_BUILD_DIR/bin/llvm-config pip install llvmpy numba
```

```
In [1]: import numpy as np
        from numba import autojit, jit, double
```

Numba provides two major decorators: `jit` and `autojit`.

The `jit` decorator returns a compiled version of the function using the input types and the output types of the function. You can specify the type using `out_type(in_type, ...)` syntax. Array inputs can be specified using `[:,:]` appended to the type.

The `autojit` decorator does not require you to specify any types. It watches for what types you call the function with and infers the type of the return. If there is a previously compiled version of the code available it uses it, if not it generates machine code for the function and then executes that code.

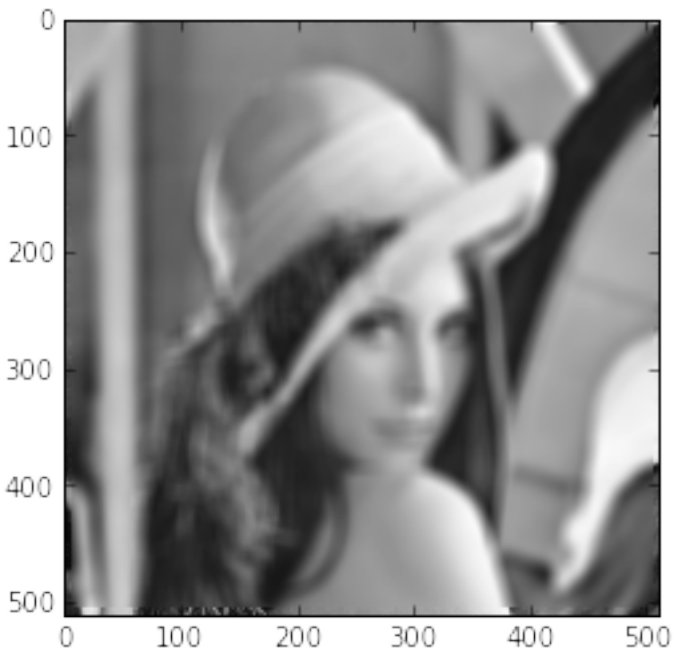
```
In [2]: def sum(arr):
            M, N = arr.shape
            sum = 0.0
            for i in range(M):
                for j in range(N):
                    sum += arr[i,j]
            return sum
        fastsum = jit('f8(f8[:,:])')(sum)
        flexsum = autojit(sum)

In [3]: arr2d = np.arange(600,dtype=float).reshape(20,30)
        print sum(arr2d)
        print fastsum(arr2d)
        print flexsum(arr2d)
        print flexsum(arr2d.astype(int))

179700.0
179700.0
179700.0
179700.0

In [4]: %timeit sum(arr2d)

1000 loops, best of 3: 623 µs per loop

In [5]: %timeit fastsum(arr2d)

1000000 loops, best of 3: 1.75 µs per loop

In [6]: 623 / 1.75 # speedup

Out[6]: 356.0

In [7]: %timeit arr2d.sum()

100000 loops, best of 3: 5.14 µs per loop

In [8]: 5.14 / 1.75 # even provides a speedup over general-purpose NumPy surm

Out[8]: 2.937142857142857
```

The speed-up is even more pronounced the more inner loops in the code. Here is an image processing example:

```
In [14]: @jit('void(f8[:,:],f8[:,:],f8[:,:])')
         def filter(image, filt, output):
             M, N = image.shape
             m, n = filt.shape
             for i in range(m//2, M-m//2):
                 for j in range(n//2, N-n//2):
                     result = 0.0
                     for k in range(m):
                         for l in range(n):
```

```
                        result += image[i+k-m//2,j+l-n//2]*filt[k, l]
                output[i,j] = result

        from scipy.misc import lena
        import time
        image = lena().astype('double')
        filt = np.ones((15,15),dtype='double')
        filt /= filt.sum()
        output = image.copy()
        filter(image, filt, output)
        gray()
        imshow(output)
        start = time.time()
        filter(image[:100,:100], filt, output[:100,:100])
        fast = time.time() - start
        start = time.time()
        filter.py_func(image[:100,:100], filt, output[:100,:100])
        slow = time.time() - start
        print "Python: %f s; Numba: %f ms; Speed up is %f" % (slow, fast*1000, slow / fast)
```

```
Python: 4.389855 s; Numba: 7.434130 ms; Speed up is 590.500176
```

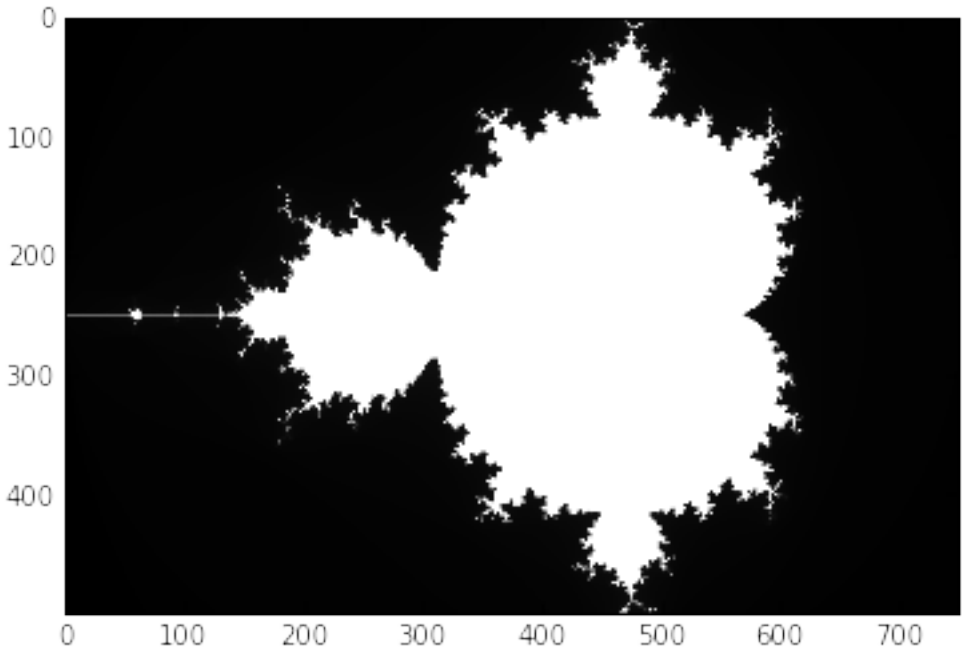


You can call Numba-created functions from other Numba-created functions

```
In [28]: @autojit
         def mandel(x, y, max_iters):
             """
             Given the real and imaginary parts of a complex number,
             determine if it is a candidate for membership in the Mandelbrot
             set given a fixed number of iterations.
             """
```

```
        i = 0
        c = complex(x, y)
        z = 0.0j
        for i in range(max_iters):
            z = z**2 + c
            if abs(z)**2 >= 4:
                return i

        return 255

@autojit
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image

image = np.zeros((500, 750), dtype=np.uint8)
imshow(create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20))
```

Out[28]: <matplotlib.image.AxesImage at 0x10790a8d0>



```
In [16]: %timeit create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
```

```
10 loops, best of 3: 40.2 ms per loop
```

```
In [17]: %timeit create_fractal.py_func(-2.0, 1.0, -1.0, 1.0, image, 20)
```

```
1 loops, best of 3: 403 ms per loop
```

```
In [18]: 403/ 40.2 # speedup of compiling outer-loop (inner-loop mandel call is still optimized)
```

```
Out[18]: 10.024875621890546
```

==========

Numba works very well for numerical calculations and infers types for variables. You can over-ride this inference by passing in a locals dictionary to the autojit decorator. Notice how the code below shows both Python object manipulation and native manipulation

```
In [20]: class MyClass(object):
             def mymethod(self, arg):
                 return arg * 2

         @autojit(locals=dict(mydouble=double)) # specify types for local variables
         def call_method(obj):
             print obj.mymethod("hello")    # object result
             mydouble = obj.mymethod(10.2) # native double
             print(mydouble * 2)           # native multiplication

         call_method(MyClass())
```

```
hellohello
```

```
40.8
```

Complex support is available as well.

```
In [25]: @autojit
         def complex_support(real, imag):
             c = complex(real, imag)
             return (c ** 2).conjugate()

         c = 2.0 + 4.0j
         complex_support(c.real, c.imag), (c**2).conjugate()
```

```
Out[25]: ((-12-16j), (-12-16j))
```

The roadmap for Numba includes better error-handling, support for almost all Python syntax which gets compiled to code that either uses machine instructions or else the Python library run-time, improved support for basic types, and the ability to create objects easily.

The commercial product NumbaPro includes additional features:

- ability to create ufuncs (fast-vectorize)
- ability to target the GPU via CUDA
- ability to target multiple-cores

- array-expressions (ability to write NumPy-like code that gets compiled)