Accelerating Python. Example with sensitivity analysis

Sergey Rykovanov

Plan for today

1. Tools for acceleration of Python code
    - os $\rightarrow$ os.system(somebin) :)
    - `ctypes` call C from Python
    - Numba
    - Cython
    - Pybind11 call Python from C++, call C++ from Python!
    - Run snippets

About Python

Great advantages:

- Easy to learn
- Easy to get complicated libraries
- Easy to combine purpuses

Great minuses:

- Slow
- Interepreter causes troubles with shared-memory parallelism
- Runtime errors...

About C/C++

Great advantages:

- Fast
- Compile once and use long
- OpenMP

Great minuses:

- Study for all life
- SegFault every time :)
- Libraries require some skills

Big Dream.

- Easy to learn
- Easy to get complicated libraries
- Easy to combine purpuses
- Fast
- Compile once and use long
- OpenMP or smth like this

When it is a good idea?

- Need to accelerate particular parts of code
  E.g. a lot of work with matrices/tensors or many for-type loops
- Use of really tuned and large libraries
  NLopt, openGL, openCV
- os.system call binary – why not?

Data conversion

Main idea: Python int != C/C++ int

type	C/C++	Python
int	Fixed size	Arbitrary size[1]
float	Fixed precision	Arbitrary precision
complex	Built-in (but no built-in conversion)	Built-in
string	Built-in (but no built-in conversion)	Built-in
bool	Built-in (built-in conversion)	Built-in

[1] example here!

Our target Python procedure

```
def monte_carlo_pi_for(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Ok, in C it looks like this...

```c
double compute_pi(int nsamples)
{
    int i;
    int acc = 0;
    srand(time(NULL));
    double x, y, z;
    for (i = 0; i < nsamples; i++)
    {
      x = (double)rand() / RAND_MAX;
      y = (double)rand() / RAND_MAX;
      z = x * x + y * y;
      if (z <= 1)
          acc++;
    }
    return (4.0 * acc) / nsamples;
}
```

gcc -fPIC -fopenmp -O3 -shared -o libPI.so compute_pi.c

Ok, let us do binding with Ctypes

```python
import ctypes #
clibPI = ctypes.CDLL('./libPI.so') #
n = 10000000
answer = clibPI.compute_pi(n)
```

Ok, let us do binding with Ctypes

```
import ctypes #

clibPI = ctypes.CDLL('./libPI.so') #
n = 10000000
answer = clibPI.compute_pi(ctypes.c_int(n)) #
```

Ok, let us do binding with Ctypes

```
import ctypes #

clibPI = ctypes.CDLL('./libPI.so') #
clibPI.compute_pi.restype = ctypes.c_double #
n = 10000000
answer = clibPI.compute_pi(ctypes.c_int(n)) #
```

Easy path to Openmp now!

```c
double par_for(int nsamples)
{
    double x = 0.0;
    #pragma omp parallel for reduction(+:x)
    for (int i = 0; i < nsamples; i++)
    {
        if (i % 2 == 0)
            x += i * 0.5;
    }
    return x;
}
```

Easy path to Openmp now! (where is the bottleneck???)

```
double compute_pi(int nsamples)
{
  int i;
  int acc = 0;
  srand(time(NULL));
  double x, y, z;

#pragma omp parallel for reduction (+:acc) private(x,
  for (i = 0; i < nsamples; i++)
  {
    x = (double)rand() / RAND_MAX;
    y = (double)rand() / RAND_MAX;
    z = x * x + y * y;
    if (z <= 1)
        acc++;
  }
  return (4.0 * acc) / nsamples;
}
```

```python
@jit(nopython=True)
def jit_monte_carlo_pi_for(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
```

```python
from numba import njit
@njit
def f(n):
    s = 0.
    for i in range(n):
        s += sqrt(i)
    return s
```

Opportunities of numba

- Python functional known by numba and in particular
- Numpy functional known by numba [2]

More details

- Python lists
- Numpy arrays
- Tuples
- Dicts

What cannot be accelerated:

- pandas
- scipy
- many others

[2] habr pt 1

Useful options

There are also some more useful options[3]

- **nogil=True**
- **parallel=True**
- **cache=True**

[3] habr pt 2

Cython

- Types like in Python
- Definitions with C style
- Faster and cheaper way

example.pyx

```
from libc.math cimport pow

cdef double square_and_add (double x):
    """Compute x^2 + x as double."""
    return pow(x, 2.0) + x

cpdef print_result (double x):
    """This is a cpdef function
     that can be called from Python."""
    print("({} ^ 2) + {} = {}".format(x,
    x, square_and_add(x)))
```

setup.py

```
from distutils.core import Extension, setup
from Cython.Build import cythonize
# define an extension that
#will be cythonized and compiled
ext = Extension(name="example",
        sources=["example.pyx"])
setup(ext_modules=cythonize(ext))
```

python setup.py build_ext inplace

myscript.py

```
import example
A = example.double square_and_add(10)
print(A)
```

Back to computations

```
import random
cpdef double compute_pi (int nsamples):
    """
    compute pi
    parameter: nsamples -- integer
    """
    cdef int i
    cdef double x, y
    cdef int acc
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x **2 + y**2 <= 1):
            acc = acc + 1
    return 4.0 * acc / nsamples
```

PyBind11

pybind11.readthedocs.io/en/stable/basics.html

pybind11

stable

Search docs

Read the Docs v: stable

Docs » First steps

Edit on GitHub

First steps

This sections demonstrates the basic features of pybind11. Before getting started, make sure that development environment is set up to compile the included set of test cases.

Compiling the test cases

Linux/MacOS

On Linux you'll need to install the **python-dev** or **python3-dev** packages as well as **cmake**. On Mac OS, the included python version works out of the box, but **cmake** must still be installed.

After installing the prerequisites, run

```
mkdir build
cd build
cmake ..
make check -j 4
```

The last line will both compile and run the tests.

Windows

On Windows, only **Visual Studio 2015** and newer are supported since pybind11 relies on various

Before we start

```
git clone https://github.com/pybind/pybind11.git
cd pybind11
mkdir build
cd build
cmake ..
make install
```

conda install pybind11 works

Basic example

```cpp
// hello.cpp
#include <pybind11/pybind11.h>
#include <iostream>
using namespace std;
int add(int i, int j) {
    cout << "Hello from C++!" << endl;
    return i + j;
}
PYBIND11_MODULE(hello_world, m) {
    m.doc() = "pybind11 hello world plugin";
    m.def("add", &add,
    "A function which adds two numbers");
}
```

g++ -O3 -Wall -shared -std=c++11 -fPIC ‘python3 -m pybind11 –includes‘ hello.cpp -o hello‘python3-config –extension-suffix‘

Now try how it looks like..

PyBind11

We are also able to use

- Python objects as variables for C++ functions (see daxpy example)
- Use Openmp inside calls
- Setup and call Python-functions with lambda-functions inside of C++ code
- Use standard Python calls inside of C++ code

Run snippets

PyBind11

And also advanced things

- Custom data structures
- Binding custom data structures
- OOP bindings
- Work with STL containers
- Advanced Numpy bindings