Linking Python to C with CFFI

Why link Python to C?

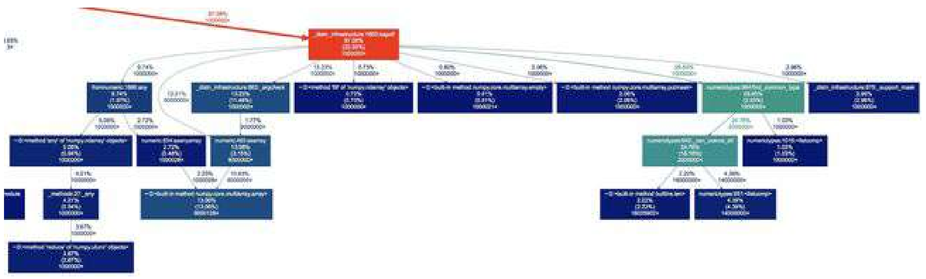
I hope that it is uncontroversial to state that Python is a great language that can suffer from occasional performance issues. This is especially true if Python is being used in heavy numerical computing environments like those at Gamalon. Gamalon is not the first to require using Python for numerical tasks. To meet this need, libraries like `NumPy`, `SciPy`, `Pandas` and others provide users with well tested implementations of most common numerical tasks. Most of these numerical tasks, such as matrix multiplication or special function evaluation among others, have reference implementations in C or Fortran that are linked to Python through many layers of indirection.

For rapid prototyping, this turns out to be a significant time saver, but what are the costs of these indirection layers? This can be answered by exploring the callgraph for the following code that simply evaluates the log of the probability density function of the Beta distribution with distributional parameters 1 and 2:

```python
from random import randint
from scipy.stats import beta

N = 1000000
for i in range(N):
    a = beta.logpdf(i, 1, 2)
```

This simple script has the following callgraph (click to zoom in):



We clearly see from the callgraph that a significant portion of our compute time is spent manipulating array shape, constructing arrays, and performing other associated operations. In fact, the underlying math for the distribution doesn't even make an appearance in the callgraph!

The story is certainly different if, instead of making repeated calls to the `logpdf` function with a scalar, we make a single call with the vector. In this situation, the call overhead of the array manipulation is swamped by the vectorized cost of the math itself. Overall runtime is reflected by:

Scalar	Vector
48.6 s	0.143 s

Vectorization, then, is the key to writing performant `NumPy` and `SciPy` based code.

Unfortunately, our specific use-cases typically involve lots of scalar calls, and thus, we encountered significant overhead. Can this cost be optimized away?

Beyond just the overhead, what happens when the user wants to use PyPy or some other tracing JIT to optimize the rest of their Python code (e.g. dictionary operations and the like)? Many of these libraries are simply not compatible with PyPy given PyPy's partial support of the C API. In fact, `NumPy` only recently passed all upstream tests when run under the PyPy interpreter, but PyPy was generally unable to provide any performance optimizations. Is it possible to still take advantage of the C and Fortran reference implementations in PyPy without rewriting them in Python?

Generally yes. This linking can be done with a tool called CFFI, or the C Foreign Function Interface, which provides light weight, PyPy compatible, Python bindings for C code. In the remainder, we will explore how to write the bindings, how to link those bindings to Python, and what their associated performance impacts are.

Writing and Building the C

For the sake of comparison, we will implement the Beta `logpdf` function in C. This function is given by:

$logpdf(x;\alpha,\beta) = \log\left(\Gamma(\alpha+\beta)\right) - \log\left(\Gamma(\alpha)\right) - \log\left(\Gamma\beta\right) + (\alpha-1)\log(x) + (\beta - 1)\log(1-x)$

which can be implemented in C like so:

```c
double beta_logpdf(double a, double b, double x){
  double prefactor;
  prefactor = lgamma(a + b) - (lgamma(a) + lgamma(b));
  return prefactor + (a - 1.) * log(x) + (b - 1.) * log(x - 1.);
}
```

Next, we can then start to explore writing a CFFI compatible build script. The first place to start is by constructing the associated header file, `beta.h`, for the above function. The entire contents of the header file are given by:

```
double beta_logpdf(double a, double b, double x);
```

This looks just like a normal header file, because it is. This header file is used to tell CFFI which functions you want to link to Python. Therefore, you should only add function prototypes for the functions you actually want to link.

With the header file and implementation in hand, we next turn our attention to implementing the build script that links with the API-level out-of-line linking paradigm. Generally, the CFFI works by taking the desired C code, automatically generating a C extension for the code, and then building that.

The script begins by import `CFFI` and creating a new `ffibuilder`. The `ffibuilder` is the object that will take care of the code generation and compilation.

```
import cffi
ffibuilder = cffi.FFI()
```

With the `ffibuilder` defined, we need to tell the `ffibuilder` where to find the file containing the method prototype for `beta_logpdf` and its implementation.

```
sourcefile = os.path.join('.', 'beta.h')
source = os.path.join('.', 'beta.c')
```

Next, we need to read in our files and build them. We first read in the header file and use the header file to inform the `set_source` method which functions to import. We also need to provide any necessary extra information about how to perform the code generation and provide any extra compilation arguments. Because our example is plain, we only need to tell the `ffibuilder` what to name the resulting module, `_beta` in our case, where to find the module and sources, and how to compile it. While not expressly necessary, I try and compile with '-O3', '-march=native', '-ffast-math' whenever it provides performance benefits. In this case, the move from `-O2` to `-O3` halves the run time.

```
with open(sourcefile) as f:
    ffibuilder.cdef(f.read())

ffibuilder.set_source(
    '_beta',
    '#include "{0}"'.format(sourcefile),
    sources=[source],
    library_dirs=['.'],
    extra_compile_args=['-O3', '-march=native', '-ffast-math'])
```

Finally, we build it. I prefer to build with `verbose=True` to ensure that I have all the information necessary to fix any problems that could occur.

```
ffibuilder.compile(verbose=True)
```

Putting this all together, we have:

```
import os
import cffi

ffibuilder = cffi.FFI()
sourcefile = os.path.join('.', 'beta.h')
source = os.path.join('.', 'beta.c')
with open(sourcefile) as f:
    ffibuilder.cdef(f.read())

ffibuilder.set_source(
    '_beta',
    '#include "{0}"'.format(sourcefile),
    sources=[source],
    library_dirs=['.'],
    extra_compile_args=['-O3', '-march=native', '-ffast-math'])

ffibuilder.compile(verbose=True)
```

which, when run, reports a successful build. The successful build gives us multiple files; `_beta.c` which is the emitted C extension module, the compiled object library, `_beta.o`, and `_beta.cpython-36m-darwin.so` which is the dynamic library that Python will actually load.

As a closing remark, it is possible to define the C inline as described in the CFFI documentation, but I have found that it gets tedious to do this for large C projects, like those that we use internally.

Linking with Python

Given the built module above, we can now import it into Python. The module will have the name prescribed by the first argument to the `set_source` method, which in our case was `_beta`. Because we're only interested in the library itself and not any of the other `FFI` periphery, such as memory allocation, we only need to import `lib`. Importing only the function that we actually used, our import statement is simply `from _beta.lib import beta_logpdf`. Note, the name of the function in C is the same as that in the module.

Putting this all together, we can call the linked function with:

```
from _beta.lib import beta_logpdf

beta_logpdf(1, 2, 1)
```

Performance Tests

Running the scripts detailed in the Scripts section below, we see the following results.

Method	Time [s]	Slowdown (C Reference)
Reference C	0.0283 s	-
`SciPy` Scalar	48.6 s	1720
`SciPy` Vector	0.143 s	5.05
`CFFI` in cPython 3.6	0.381 s	13.5
`CFFI` in PyPy3	0.0539 s	1.9

Where all timings were generated on my Mid-2017 Non-touchbar MacBook Pro with 2.3 GHz Intel Core i5. The `SciPy` scalar and vector results are exactly the same as those reported in the introduction. The reference C results were computed by using the `beta_logpdf` function reported above and timed using the system clock available in the C standard library. The C reference was compiled with GCC 7 with compiler flags `-O3`, `-march=native`, and `-ffast-math`. The reported CFFI numbers were computed using the exact build scripts linked above. We elected to test in both cPython 3.6 and PyPy3 to test the possible benefits that a tracing JIT can afford us in super simplistic scenarios like this. cPython 3.6 and PyPy3 are both from the bottle poured with Homebrew.

Interestingly enough, we observe that CFFI provides two order of magnitude improvement in the overhead for calling the `logpdf` function within cPython. This is already a drastic speedup, but we can do even better by running under PyPy, which provides only a factor of 2 slow-down versus the reference C.

What is the source of the remaining factor of two? We can explore the computational costs of an exceedingly simple function to further understand the overhead cost of the interpreter. To understand this, we will time a function that takes in three values and returns one:

```
double nop(double a, double b, double x){
    return x;
}
```

Linking in the method described above and then timing under both `PyPy3` and `cPython 3.6`, observe the following timings:

Interpreter	Time [s]	Slowdown (PyPy Reference)
cPython 3.6	0.248 s	15.1
PyPy3	0.0164 s	-

Interestingly enough, that means that, calling an empty function linked with

CFFI from cPython costs on average 248 ns, while the equivalent tasks costs 16 ns in PyPy. We can then see that the majority of the computational benefit that we have observed from moving from cPython to PyPy is the more efficient C interface.

Closing Remarks

Motivated by Knuth's oft-cited adage that "premature optimization is the root of all evil", the general advice that is given in these situations is to write working code, profile it, and then optimize hotspots. For Python, one of the common approaches to such an optimization is to translate the hotspots to C and then link them to Python.

In the above, we have explored specific techniques for such an optimization pathway that also permits the user to use interpreters such as PyPy. This allows for using a JIT to optimize other aspects of the codebase, such as heavy dictionary access, or string computations.

Combining `PyPy` with a `CFFI` linked module, it is possible to obtain performance within an order of magnitude to C while still operating within Python. Amazingly enough, `PyPy` + `CFFI` out performs even vectorized `SciPy` in this situation! Granted, this is a sample size of 1 but it is at least encouraging.

Of course, CFFI is not a magic bullet. Performing optimizations of this kind can significantly limit the flexibility of certain aspects of your code-base, and require developers to be more deliberate about their use of interfaces. It also adds maintenance costs because developers will now need to be able to support both C and Python.

Scripts

The callgraph was generated with

```
python -m cProfile -o prof.out beta.py
gprof2dot.py -f pstats prof.out | dot -Tsvg -o callgraph.svg
```

Run on `cPython 3.6`

```
from _beta.lib import beta_logpdf
from scipy.stats import beta
import time

N = 1000000
start = time.time()
for i in range(N):
    a = beta.logpdf(i, 1, 2)
end = time.time()
```

```
print(end - start)

start = time.time()
a = beta.logpdf(list(range(N)), 1, 2)
end = time.time()
print(end - start)

start = time.time()
for i in range(N):
    a = beta_logpdf(1, 2, i)
end = time.time()
print(end - start)
```

Run only under `PyPy3`

```
from _beta.lib import beta_logpdf
import time

start = time.time()
for i in range(N):
    a = beta_logpdf(1, 2, i)
end = time.time()
print(end - start)
```

Compiled with GCC 7 with command `gcc -O3 -march=native -ffast-math -finline-functions beta.c -lm`

```
#include <math.h>
#include <stdio.h>
#include <time.h>

double nop(double a, double b, double x){
    return x;
}

double beta_logpdf(double a, double b, double x){
  double prefactor;
  prefactor = lgamma(a + b) - (lgamma(a) + lgamma(b));
  return prefactor + (a - 1.) * log(x) + (b - 1.) * log(x - 1.);
}

int main(void){
  int N;
  clock_t start, end;
  double diff;
  N = 1000000;
  start = clock();
  for(int i = 0; i < N; i++){
    beta_logpdf(1.0, 2.0, i);
  }
  end = clock();
  diff = (double)(end - start) / (CLOCKS_PER_SEC);
  printf("Time in seconds %f", diff);
  return 0;
}
```