jitpy Documentation

Release 0.1

Maciej Fijałkowski

July 13, 2016

Contents

CHAPTER 1

What it jitpy?

jitpy is a hack to embed PyPy inside CPython: the goal is to let PyPy optimize selected functions and call them from CPython. The provided interface is limited: you can pass only simple builtin immutable types and numpy arrays. In other words, you cannot pass lists, dicts, and instances of custom classes.

The usage pattern is similar to numba, though it does have very different characteristics. A simple example

```
from jitpy import setup
setup('<path-to-pypy-home>')
from jitpy.wrapper import jittify

@jittify([int, float], float)
def func(count, no):
    s = 0
    for i in range(count):
       s += no
    return s

func(100000, 1.2)
```

This function will be executed by the underlying PyPy, thus yielding a significant speed benefit (around 50x in my own measurements).

CHAPTER 2

Motivation

The idea behind `jitpy` is to lower the barrier of entry to PyPy. A lot of people have complicated dependencies that don't work under PyPy, yet they want some way to speed up numeric computations. This is where `jitpy` comes to play.

CHAPTER 3

Installing

You can install jitpy using `pip install jitpy` in your CPython installation..

You also need to download and unpack a very recent PyPy (newer than 2nd of Dec 2014), which can be e.g. downloaded from PyPy nightlies.

An example of usage:

Download 64bit binary or 32bit binary for linux:

```
~$ wget http://buildbot.pypy.org/nightly/trunk/pypy-c-jit-74798-f1b314da580e-linux64.tar.bz2
~$ tar xjf pypy-c-jit-74798-f1b314da580e-linux64.tar.bz2
~$ export PYPY_HOME=`pwd`/pypy-c-jit-74798-f1b314da580e-linux64/bin/
~$ pip install jitpy
```

And you should be able to run examples (**NOTE:** since jitpy takes source code via `inspect` module, you can't run jittify on functions typed from python interactive prompt)

CHAPTER 4

Using jitpy

jitpy is not magic - what it does is to move code across the boundary between the two different Python implementations. It means that while PyPy and CPython don't share any data, you can pass `ints`, `floats`, `strings` and `numpy arrays` without copying, since it's done in-process. It's also faster compared to out-of-process solutions, like `multiprocessing`. However, one needs to remember that the global state and the namespaces of the two interpreters are separate, which means that the functions and classes declared on CPython won't be automatically available in PyPy, and viceversa. Moreover, if you import the same module in both interpreters, the module will be actually imported twice, which can make a difference in case of modules which have side-effects when imported.

The API looks like this:

- `jitpy.setup(pypy_home=None)` - has to be called before anything in order to point to the correct PyPy build directory. `pypy_home` points to the directory of pypy checkout/installation. If None is passed, it'll default to `PYPY_HOME` environment variable.
- `jitpy.wrapper.jittify(argtypes, restype=None)` - a wrapper that's passed argument types as a list and restype as on of the:
  - `int, float, string` - immutable types. Additionally `None` can be used for return type
  - `'array'` - a numpy array, can only be used as an argument, not a return value. Also only simple types are supported for now (no compound dtypes, no string, unicode) or object dtypes
- `jitpy.extra_source(source)` - executes `source` inside PyPy. The classes and functions defined there will be visible by the functions decorated with `@jittify`. For example:

```
jitpy.extra_source("""
class X:
    def __init__(self, x):
        self.x = x
""")

class Y(object):
    pass

@jitpy.wrapper.jittify([], int)
def func():
    return X(42).x

func()
```

this will work, however trying to reference `Y` from inside the `func` will result in a `NameError` exception.

Differently than numba, you can use all Python constructs inside jitted functions, including the most dynamic ones like `import`, `pdb`, `sys._getframe`, `ex ec`, etc. However note that `sys.path` is **not** inherited: if you want to include extra directories in `sys.path`, you need to modify it explicitly using `jitpy.extra_source`.

CHAPTER 5

Limitations

The API is limited to builtin types, because it's easy to see how the boundary looks like. Numpy arrays can be shared, because the data is visible as a pointer in C on the low level. `sys.path` has to be initialized separately, but will respect all the libraries installed in the underlying `pypy`.

CHAPTER 6

Benchmarks

Everyone loves benchmarks. The way one presents benchmarks is very important. I'm going to compare on a limited set of benchmarks various tools designed for a specific purpose – speeding up Python in pieces or in whole without learning a new language. That means that tools like Cython, C, Fortran are out of scope of this comparison. I'm going to compare CPython, jitpy, numba and to some extent PyPy.

The basic benchmark measures the overhead of calling through the layer. The first example is empty function, the second loops ten times to do three additions, in order to run **any** python code.

benchmark	pure python	jitpy	numba
return 1	0.09s (1.0x)	0.58s (6.4x slower)	0.36s (4x slower)
loop 10	0.95s (1.0x)	0.8s (1.2x faster)	0.39s (2.4x faster)

While this is an interesting data point, this generally points out you should not write very tiny functions using those layers, but as soon as there is any work done, CPython is just very slow. For a comparison, running those benchmarks under PyPy gives, respectively, 0.003s (30x speedup) and 0.11s (8.6x speedup), which means that if you have a high granularity of functions that can't be nicely separated, a wholesome solution like PyPy gives more benefits.

The array benchmark gives insight into passing arrays into the functions and doing more advanced things. The benchmarks do, in order:

- pass 1d array, walk it for a sum (equivalent to `sum(a)`)
- pass 2d array, walk it for a sum (equivalent to `sum(a)`)
- pass 2d array, walk it, create tuple of size two and count the length
- pass 2d array, walk it, create an instance of a class and read it's attribute

Benchmarks grow in complexity as what sort of stuff is done in them (and also grow in silliness). Results are as follows. Notes:

- we do 10x less iterations with CPython just because of how bloody slow it is
- because we don't cross boundary much, the numbers for jitpy should be very similar to what you would get running pure PyPy

benchmark	pure python	jitpy	numba
1d array	12.7s (1.0x)	0.28s (45x faster)	0.21s (60x faster)
2d array	16s (1.0x)	0.35s (46x faster)	0.22s (73x faster)
2d + tuple	33.5s (1.0x)	0.30s (104x faster)	64.5s (1.9x slower)
2d + instance	48.4s (1.0x)	0.30s (161x faster)	53.9s (1.1x slower)

The benchmark results might look very confusing, but here are my takeaways:

- CPython is slow at numerics
- if everything is perfect for numba to emit optimize LLVM code, LLVM does a very good job

- PyPy (and jitpy) is slightly to moderately slower than numba for simple cases
- PyPy (and jitpy) is vastly better for complicated cases that involve more of Python semantics.

After all, it makes sense - numba is a specific tool that does not try to be fast on all Python code, while PyPy runs all Python code and tries to be fast on it.

PyPy (and jitpy) also supports more of Python (in fact all), so it's possible to get tracebacks, `try:`, `except:` clauses, `imports` etc. etc. that are simply not supported by numba.

However, your mileage may vary, try tools before jumping into conclusions.

- genindex
- search