Just in Time Compilation

K. Cooper[1]

[1]Department of Mathematics
Washington State University

2020

Python Interpreted

Python is an interpreted language. Each line of code is translated into machine language at run time. That translation takes a lot of time, compared to execution time.

- Of course, there is a byte-compiled file (the .pyc).
- Of course, there are compiled libraries to use (such as numpy).

Still, it seems like compilation might speed up and/or vectorize our clumsy programming.

JIT also does optimization. It might also give us access to certain libraries designed for compilers.

Just In Time Compilation

- Research by Sun Computing in the 1980s
- Java
- Now (or soon) available for Javascript, PHP...

JIT is available for Python – Numba package, and PyPy

- Numba developed by Nvidia, aimed at parallel and CUDA
- PyPy alternate implementation of Python

Just In Time

There is never any question about translation to machine instructions.
Ordinary interpreters do this basically one line at a time

Loops can thus be translated millions of times, to the point that the translation takes more time than executing the machine instructions.

The point of JIT is to compile larger blocks of code, and reuse those machine instructions; not just one line.

Simplest Example

```
result = log(exp(2.781))
result = log(exp(2.781))
result = log(exp(2.781))
                ⋮
result = log(exp(2.781))
result = log(exp(2.781))
```

10000 lines like this take .0340 seconds.

With Loop

```
for i in range(10000):
    result = log(exp(2.781))
```

This takes .0345 seconds.
About the same as running line by line.

With JIT

```
def theLoop():
    for i in range(10000):
        result = log(exp(2.781))
```

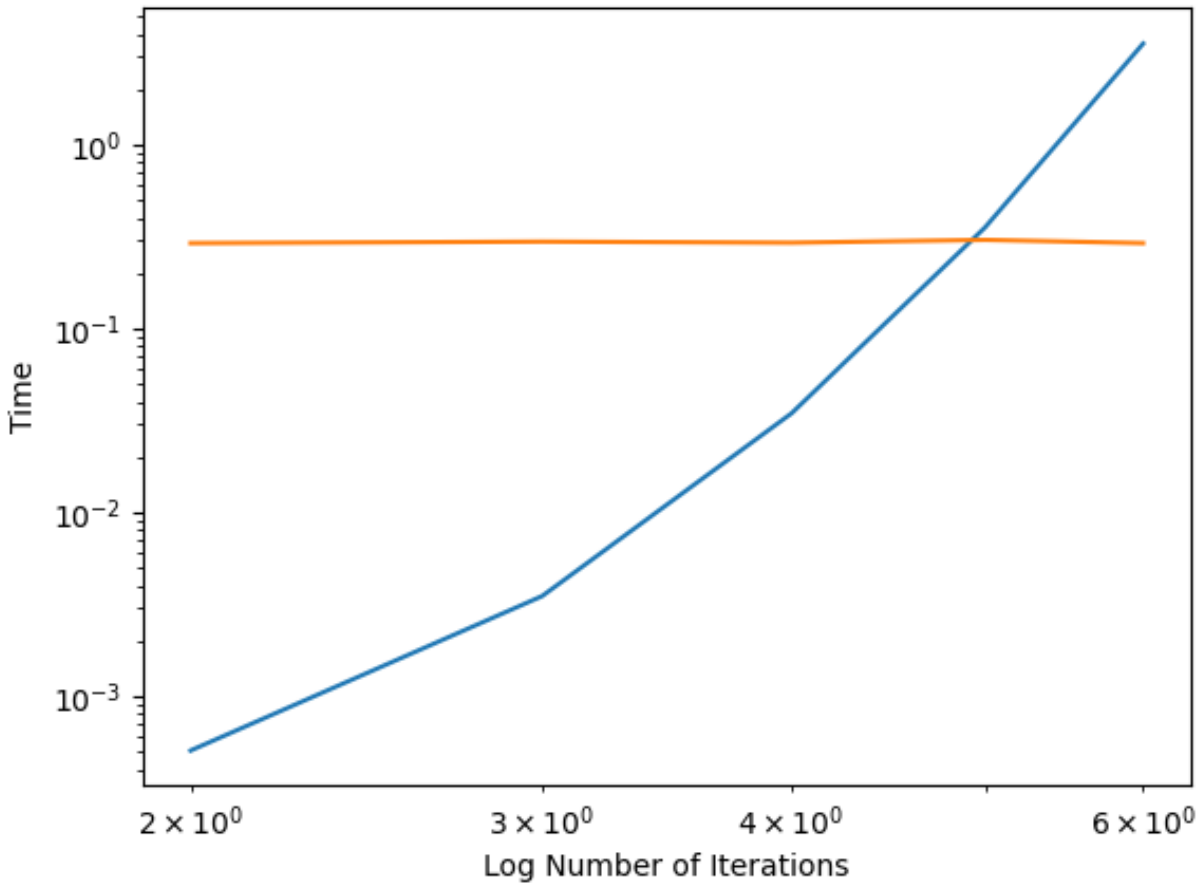
This takes .2900 seconds.

Note in passing that we had to put the loop in a function, and then apply JIT compilation to the whole function.

Numba JIT works on functions.

Comparison

The time for the code with JIT is orange.



Remarks

Most of the time to run Python is interpreter.

Compiling takes a lot of time.

Advantage of JIT is reuse of compiled code.

When computation changed to `x***x` (so no numpy), then 10000 lines takes 0.0 seconds.

Bytecode optimizer no longer confused by numpy. . .

“Vector Code”

We talk about “vectorization” and “Vector Code”.
This does not mean parallelization.
Instead, it means writing code using array operations, which can be handed off to compiled code.

```
for i=range(n):
    A[i] = x[i]*y[i]
```

Scalar Code

```
A = x*y
```

Vector Code

The Numba Package

Fundamentally about compiling Python code.

- JIT - just-in-time compilation
- Vectorization of scalar code
- Pyculib - BLAS etc. for CUDA/Python
- cuda.jit - full access to CUDA API

Using Numba

```
from numpy import log,exp
from numba import jit,void,int64

@jit@jit(void(int64))
def theLoop(n):
    for i in range(n):
        result = log(exp(2.781))
```

Takes 0.33 secondsTakes 0.0 seconds

Even with a return value, the optimization kicks it.

Illustration

Example shamelessly borrowed from
https://ipython-books.github.io/52-accelerating-pure-python-code-with-numba-and-just-in-time-compilation/

```
def mandelbrot_python(size, iterations):
  m = np.zeros((size, size))
  for i in range(size):
    for j in range(size):
      c = (-2 + 3. / size * j +
          1j * (1.5 - 3. / size * i))
      z = 0
      for n in range(iterations):
        if np.abs(z) <= 10:
          z = z * z + c
          m[i, j] = n
        else:
          break
  return m
```

```
def mandelbrot_numpy(size, iterations):
  m = np.zeros((size,size))
  c = np.zeros((size,size),dtype=complex)
  z = np.zeros((size,size))
  for i in range(size):
    for j in range(size):
      c[i,j] = (-2 + 3. / size * j +
              1j * (1.5 - 3. / size * i))
  for n in range(iterations):
    z = (np.abs(z)<=10)*(z*z+c) +
        (np.abs(z)>10)*z
    m += (np.abs(z)<=10)
  return m
```

Times

Mandelbrot program:

- No JIT: 10.5-11.5 seconds
- numpy rewrite: 2.1 seconds
- @jit: 0.63 seconds
- @jit with hints: 0.06 seconds

vectorize

```
@vectorize(['float64(float64,float64,float64)'],
           target='cpu')
def euler(y0,T,h):
    nSteps = int(T/h)
    for i in range(nSteps):
       y0 = (1.-h)*y0
    return y0
```

- If y0 is array, then numpy uses Numpy compiled functions for array computations
- Numba sees scalar function
- vectorize compiles it, if target='cpu'

  New compile, including loop implementation

Euler Vectorize

```
@vectorize(['float64(float64,float64,float64)'],
        target='cpu' target='parallel'
target='cuda' )
```

- Without vectorize: 13.2 seconds
- cpu: 27.4 seconds

  Numpy had already vectorized it – the new vectorization is redundant
- parallel: 6.9 seconds
- cuda: 2.4 seconds

CUDA

A word about CUDA is in order. . .

- API for GPU computing
- Available for C and Fortran
- Specialized functions, data types, certain variables
- Requires a different way of thinking

CUDA.JIT

What if we use a JIT compiler that understands CUDA (GPU programming)?

```
@cuda.jit('void(float64[:],float64,float64)')
def euler(y0,T,h):
  i = cuda.blockIdx.x*cuda.blockDim.x+cuda.threadIdx.x
  nSteps = int(T/h)
  for j in range(nSteps):
    y0[i] = (1-h)*y0[i]
```

- ♣ Requires understanding of CUDA
- ♣ Point is that compiling gives access to CUDA libraries.
- ♣ Runs in 0.42 seconds

Heat Equation

One other: heat equation on 200 space intervals, tiny time step.
Each new point depends on three points at earlier time step.
Not perfectly parallel.
scalar

43 seconds

```
@jit
```

0.66 seconds

```
@jit(float64[:](float64[:],float64[:],float64,float
```

0.084 seconds

```
@cuda.jit('void(float64[:],float64[:],float64,float
```

0.055 seconds

Conclusions

- If we are writing little demo programs, small computations, don’t worry about it.
- If there is one function with several loops, concentrated time usage, @jit might help.
- Usually worthwhile to give some information about arguments.
- If large arrays, then @vectorize might be good.
- If it is "production code", possibly worthwhile to use @cuda.jit. Much more complicated.
- Each of these steps is an significantly more "fiddly".