Optimizing Python code with Numba JIT compiler

Marco Fringuelli, tesina d'esame per "Calcolatori elettronici e Reti di calcolatori", AA 17/18.

The Challenge: *Is it true that Python won't ever compete with C++ performances?* In this work I have tried to partially answer this question and to change the result using a Python module introduced in 2012 called **Numba**. I have chosen two widely-used numerical calculus algorithms, that is, LU factorization and Runge-Kutta 4th order method.

1. Background

A brief presentation of the case study and the tools used in this project.

1.1 Languages

- **Python** is a very high-level programming language created by the researcher Guido Van Rossum in 1996. It is an object-oriented cross-platform scripting language, and its main characteristics are its simplicity and dynamicity, along with great potential in the majority of the fields of IT. Nowadays it is widely used for distributed applications, machine learning, mathematical computation. Although it is possible to learn how to use it very quickly, it has performances as its dark side, very low if compared with lower level languages. This comes from the fact that Python is commonly an interpreted language and it is dynamic (weakly) typed.
- **C++** is itself a high-level programming language, but older, and it has different characteristics that make it more difficult to understand but also much more suitable to many applications, e.g. dealing with hardware and operating systems. The fact is that C++ is a strongly-typed, compiled language that allows to use pointers and references. Compared to Python, C++ source may appear longer, more complicate and more criptic, but on the other hand it can be hundreds of times faster than its rival.

1.2 Algorithms

I have chosen two mathematical algorithms widely used in many applications, from graphics to control systems. Rather than taking simple matrix operations such as inversion, pivoting or vector addition, I decided to take something a little bit more complex. These methods follow more steps and, when coding, more functions are necessary to be defined (not just one) and they could be called hundreds of times at run time, depending on the size of the input operands. This situation better reflects a real case, when processes in execution call many times more than one function in their lifetime, and each of those has different execution time. Here are the numerical methods I have implemented to test and compare performaces between C++ and Numba Python:

- **LU Factorization**. The LU factorization is a matrix operation performed before the back resolution of a system with that matrix as associated matrix ($Ax = b$). It produces a Lower-triangular matrix (L), an Upper-triangular other (U) and a matrix P of pivoting so that $PA = LU$. U coincides with the matrix resulting from Gaussian elimination on A. After factoring, the system can be solved first solving the system Ux = y, and then Ly = b. Using this algorithm, a system can be solved with $O(n^2)$ operations rather than $O(n^3)$.

- **Runge-Kutta's 4th order method for solving differential equations (ODEs)**. The RK numerical methods offer a way to approximately determine the trend of a function by evaluating it in precise points. It derives from Taylor's formula, and the higher the order the higher the accuracy. The 4th order method has an accuracy of $O(h^5)$. It requires more computation than classic Euler's method, but it is much more precise, and this is the reason why it is used in real applications.

$$
\begin{aligned}
w_0 &= \alpha \\
k_1 &= hf(t_i, w_i) \\
k_2 &= hf(t_i + h/2, w_i + 1/2k_1) \\
k_3 &= hf(t_i + h/2, w_i + 1/2k_2) \\
k_4 &= hf(t_{i+1}, w_i + k_3) \\
w_{i+1} &= w_i + 1/6(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
$$

– *Runge-Kutta's 4th method. α represents the value of f at time zero. The w_i are the nodes.*

1.3 Tools

- **Profilers**. One of the greatest facts about Python is that almost everything can be easily done just by importing modules in the code. When it is about optimizing, the first thing to do is to locate bottlenecks. Assuming our code is sufficiently optimized from the algorithm p.o.v., (minimal computation complexity) we must the see how it works on the machine. Some function may be more "onerous" to be executed, some other less but it may be called repeatedly. For Python there are available tools that allow to profile the code, among of these I picked `cProfile` and `line profiler`. The purpose of the first one is to profile the entire code to roughly see how many times each function is called at run time and what is the mean execution time of it. The following goes deeper, since it analyzes, for a chosen function, all the lines of code contained in it and returns time evaluations with a bigger number of decimals.

- **NumPy**. NumPy is the most important package for scientific computing with Python. It contains lots of advanced features. It provides an array object and function working on it to do any algebraic and analytic computation. Other advanced features will not be taken into consideration for the purposes of this reports.

- **Numba.** Numba is a JIT (just-in-time) compiler for Python that allows to peed up your applications with high performance functions written directly in Python. Numba works by generating optimized machine code using the `LLVM` compiler infrastructure at import time, runtime, or statically (using the included pycc tool). Numba supports compilation of Python to run on either CPU or GPU hardware, and is designed to integrate with the Python scientific software stack. With adding *decorators* to functions (`@jit()`), array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++ and Fortran, without having to switch languages or Python interpreters. However it is not always usable efficiently: Numba is thought to work with maths-based code, and sometimes it does not work the right way. If, for a given function, it is able to infer the types of the arguments (it is NumPy-aware), the types of any object inside the function and of the returned values, great performance improvement can be obtained. When It works so, it is in `nopython` mode. If Numba is not

able to do this (for example with dictionaries or non-standard functions) it has to work in `object` mode and the time of execution may increase a lot. This is why we should "help" the jit decorator setting some optional parameter to assert it is working in nopython mode and the argument types are inferred correctly.

2. Profiling and optimizing Python code

From this point below, I will refer to tests/benchmarks I carried out on a machine with a 8GB Ram and a 2.7 GHz Intel Core i5®dual-core. Numba documentation states that performances may vary across different hardwares and different ISAs. Scripts used and other data are in this repository: `https://github.com/marcofrn23/Python-Benchmark`.

1.1 ODEs Solving

The Python code for solving differential equations has been organized in two functions: `func(t,x)`, where one can write the expression of the equation, and `solve(x0,t0,T,N)`, the function that applies the Runge-Kutta method.

Step 1. Profiling: The code execution can be first analyzed via the instruction (wrapper) `cProfile.run("solve(3,0,15,500)")`, and then more deeply with the `@profile` decorator (line profiler) put above the definition of the "solve" function in the code. Using conventional input data ($x0 = 3$, $t0 = 0$, $T = 15$, $N = 500$, it is indifferent for this matter) and any equation (equations may vary a lot, but if code results slow for a simple one, it is reasonable to think that it will be also with different ones) it is possible to quickly locate the bottlenecks of the program:

```
Total time: 0.006666 s
File: RungeKutta.py
Function: solve at line 22

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    22                                           @profile
    23                                           def solve(x0, t0, T, N):
    24         1         10.0     10.0      0.2      dt = (T-t0)/float(N)
    25         1         16.0     16.0      0.2      time = np.arange(t0, T, dt)    # create mesh
    26         1          4.0      4.0      0.1      sol = np.zeros((len(time),))
    27                                               #print (len(time),len(sol))
    28         1          2.0      2.0      0.0      sol[0] = x0
    29         1          1.0      1.0      0.0      mid = dt/2.0
    30       500        281.0      0.6      4.2      for i in range(1, len(time)):
    31       499        353.0      0.7      5.3          tau = time[i-1]
    32       499       1167.0      2.3     17.5          k1 = (func(tau, sol[i-1]))*dt
    33       499       1374.0      2.8     20.6          k2 = (func(tau+mid, sol[i-1]+k1/2))*dt
    34       499       1337.0      2.7     20.1          k3 = (func(tau+mid, sol[i-1]+k2/2))*dt
    35       499       1217.0      2.4     18.3          k4 = (func(tau+dt, sol[i-1]+k3))*dt
    36       499        903.0      1.8     13.5          sol[i] = sol[i-1] + (k1 + 2*k2 + 2*k3 + k4)/6.
    37         1          1.0      1.0      0.0      return sol
```

Figure 1: line-profiling the "solve" function with the shell command
`kernprof -l -v <nome>.py`

Step 2. Optimizing: we see that slow section is quite evident: given a N number of steps, the evaluation function must be called $5(N-1)$ times, at first to compute every k_i, and then to compute the solution at each step (lines 20-26, 90% of total time execution). The code can be optimized by adding: `@numba.jit('float64(float64, float64)', nopython = True)` above `func()` and `@numba.jit('float64[:](float64, float64, float64, int64)', nopython = True)` above `solve()`(even if the problem is mostly with "func").

The JIT decorator, in this example, takes 2 inputs: the first one is a string that it will check to infer the input arguments types and the returned arguments types, the second, "Nopython = True" represents a key flag that tells to the Numba compiler to work only if it can work in nopython mode. It is good practice, when using Numba, to always set nopython mode to "True" and, when possible, telling the arguments types. This ensures that if Numba is able to optimize the function execution, it will do it the right way.

1.2 LU Factorization

The Python code for LU factorization has been organized in three main functions: `dot(A,B)`, that performs matrix product (I have also considered the NumPy way of making the product: `A@B`, given two matrices A, B), `pivotmatrix(m)` to compute the pivoting matrix P , and finally `lu(A)` to perform the factorization.

Step 1. Profiling: The code execution can be first analyzed via the instruction (wrapper) `cProfile.run("lu(A)")`, and then more deeply with the `@profile` decorator (line profiler) put above the definition of the "lu" function in the code (cProfile is useful to look for the slowest function). Using a matrix of random-generated floating point numbers of any size (but not too small, in this example I have taken a 10x10 one) it is possible to quickly locate the bottlenecks of the program:

```
MBP-di-Marco:Progetto CER Marco$ kernprof -l -v LU.py
Wrote profile results to LU.py.lprof
Timer unit: 1e-06 s

Total time: 0.057082 s
File: LU.py
Function: lu at line 43

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    43                                           @profile  # line_profiler decorator
    44                                           def lu(A):
    45         1         11.0     11.0      0.0      n = len(A)
    46                                               # Initialization of empty L & U matrices
    47         1         47.0     47.0      0.1      L = np.identity(n)
    48         1         13.0     13.0      0.0      U = np.identity(n)
    49
    50         1        173.0    173.0      0.3      P = pivot_matrix(A)
    51                                               # A2 = dot_py(P,A)
    52         1      56031.0  56031.0     98.2      A2 = dot_numpy(P,A)
    53
    54        11         13.0      1.2      0.0      for j in range(n):
    55        10         14.0      1.4      0.0          L[j][j] = 1.0
    56        65         42.0      0.6      0.1          for i in range(j+1):
    57        55        261.0      4.7      0.5              s1 = sum(U[k][j] * L[i][k] for k in range(i))
    58        55         80.0      1.5      0.1              U[i][j] = A2[i][j] - s1
    59        65         42.0      0.6      0.1          for i in range(j, n):
    60        55        258.0      4.7      0.5              s2 = sum(U[k][j] * L[i][k] for k in range(j))
    61        55         96.0      1.7      0.2              L[i][j] = (A2[i][j] - s2) / U[j][j]
    62         1          1.0      1.0      0.0      return (L, U, P)
```

Figure 2: line-profiling the "lu" function with the shell command
`kernprof -l -v <nome>.py`

Step 2. Optimizing: we see that the slow section is here more evident than before: The 98.2% of the execution time is spent computing the matrix product between P and A (it is done only one time). If we are able to optimize the "dot" function, we optimize the entire code. This resulted not to be easy: even if Numba can work with NumPy array-matrix objects, it is not able to work in nopython mode with the product `@` implemented in NumPy (contained in the "dotnumpy" function). The first thing is to break out the product function and write it "by hand", like one would probably do in C++. After that, the JIT compiler can work rightly. (Decorator `@numba.jit('float64[:,:](float64[:,:], float64[:,:])', nopython = True)` above `lu()`).

Unfortunately, as we will see, the optimization obtained here is limited in comparison with the one of the RK method. This means that even if Numba is very handy and powerful, it sometimes can lead to wrong/inefficient modifications of the code if not controlled.

3. Comparisons and results

3.1 Benchmarks

After using Numba decorator to fasten the execution of the programs, I have tested the performances to check if there was any improvement. Finally, I have taken the perfomance data of the Naive Python execution (without Numba) and of C++ (the same algorithms).

ODEs solving: I have chosen some equation, trying to vary all the terms of input among them (initial values, T, steps, coefficients in the equation, complexity...). The test result consists of an array of time evaluations, each of which corresponding to an increasing number of steps (N from 500 to 5000, increasing by 300). Every evaluation is in reality the average of many evaluations of the same type (generally 20/30) to make sure the benchmark was consistent. (For the plots I will show I have taken the mean results between all the equations considered)
LU factorization: I have used a Python-coded random matrix generator (floating point) starting from size 10 to 25, then 30,40,50,60 and 100 (I have stopped because the results were already satisfactory). For each size, the matrix was re-generated at least 10 times, and for each one of these, the test has been repeated more times. The test result consists of an array of time evaluations, each of which corresponding to an increasing matrix size (NxN).

3.2 ODEs solving view

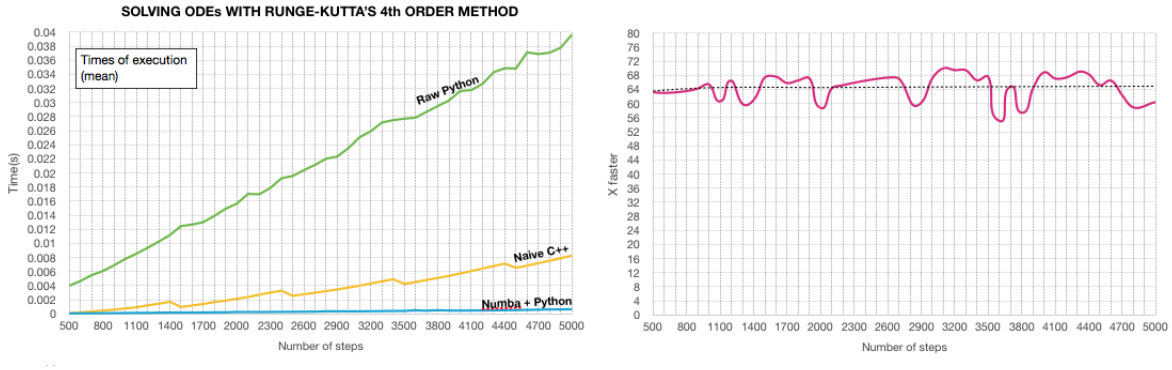

Figure 3: Results for ODEs solving benchmark, the right plot shows the trend of the number of times Numba-Python is faster than Naive Python

From these plots we see that Numba is very effective on this algorithm: the trend line on the right one shows that with the JIT decorator, the Python code gets 65x faster than before. It is also quite curious that it overtakes C++ performances: by increasing the number of equation evaluations, Numba-Python loses less efficiency than C++. Even though it may seem a wonderful result, there is to say that the C++ code used is completely naive, and it can be improved not too difficultly.
At this page, `https://murillogroupmsu.com/numba-versus-c/` there is a wider explanation of other cases in which Numba overtakes C/C++ in performances.

3.3 LU factorization view

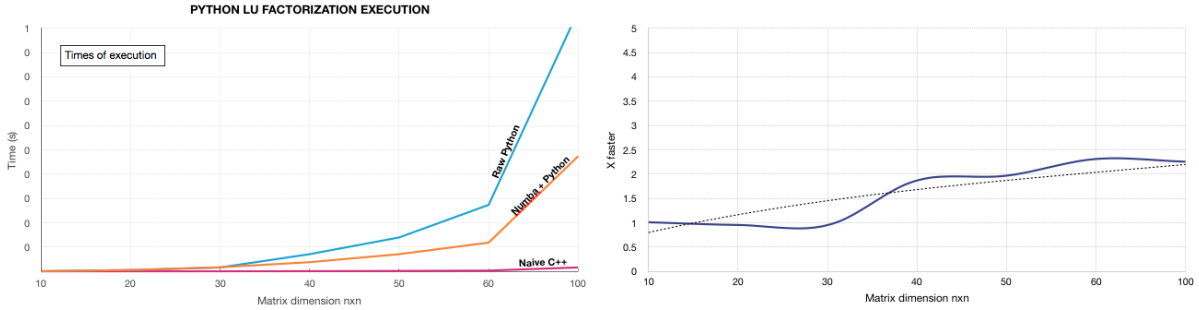

Figure 4: Results for LU factorization, the right plot shows the trend of the number of times Numba-Python is faster than Naive Python

As expected, in this case Numba is not very efficient. Python code gets almost 2.5x faster with matrices of $N \geq 10^4$ elements, which is not so bad, but remains far from C++ speed level. This is the consequence of the fact that forcing Numba to work in nopython mode reduces its possibilities, sometimes it is not appliable at all, but if we do not set the flag to true and it accidentally works in object mode, it can also slowen the code, as show in some example findable on the net (`https://stackoverflow.com/questions/21468170/numba-code-slower-than-pure-python`).

4. Advanced Numba settings

For the purposes of this report, I have only explained and showed the use of the Numba JIT decorator to just-in-time compile chunks of Python code, however this module has some other interesting feature. Here I briefly refer to the potentially most useful ones, also for non-trivial projects. For a complete documentation, go to `http://numba.pydata.org/numba-doc/0.38.0/index.html`.

4.1 JIT compilation options

- `nogil = True` to release the Global Interpreter Lock
- `cache = True` to avoid a re-compilation of a function with the JIT decorator when the program is reused, the result of compilation is stored into a file-based cache
- `parallel = True` (needs `nogil = True` and `nopython = True`) to enable a feature that automatically parallelizes operation with parallel semantic. Numba is not thought primarily to parallelize code, this feature is able to speed up code only in particular cases.

4.2 Environment variables

Some nice environment variables settable (modifiabile importing the `os.environ` function and setting them to zero/non-zero) :

- `NUMBA_DEBUG_TYPEINFER` prints information about type inference in compiling
- `NUMBA_DEBUG_CACHE` prints information about the JIT cache
- `NUMBA_DUMP_ASSEMBLY` dumps the native assembler code for the compiled functions

- `NUMBA_WARNINGS` prints out eventual Numba warnings.
- `NUMBA_OPT` (Positive number) Selects a level of optimization.

4.3 Advanced Features

Apart from the `@jit()` decorator, Numba contains other tools thought to boost the throughput of Python when working with math-heavy algorithms and arrays/matrices processing.

- `@stencil` is a decorator that allows to define a stencil kernel, a fixed pattern array elements are updated according to. When it is necessary to apply the kernel to an array, Numba generates the looping code for each element, also giving the advantage to make code much more compact.
- `@generated_jit()` is a JIT decorator appliable to function that have different behaviors depending on the types of the input argument
- CUDA/GPU support: Numba supports also CUDA programming, giving the possibility to compile a subset of the Python code into CUDA kernels following the CUDA execution model.