

BCH2203 Python - 9. Performance

Ramses van Zon

16 March 2022

You've probably noticed in several occasions that your Python script may not be as fast as you'd like.

The real question is: is it as fast as reasonable possible?

Today, we'll look at ways how to improve the performance.

Python is interpreted

- Translation to machine language happens line-by-line as the script is read.
- Repeated lines are no faster.
- Cross-line optimizations are not possible.

Python is dynamic

- Types are part of the data: extra overhead
- Memory management is automatic. Behind the scene that means reference counting and garbage collection.
- All this also interferes with optimal streaming of data to processor, which interferes with maximum performance.

Fast development

- Python lends itself easily to writing clear, concise code.
- Python is very flexible: large set of very useful packages.
- Easy of use → shorter development time

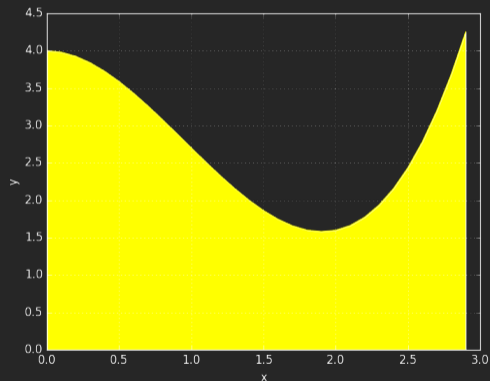
Performance hit depends on application

- Python's performance hit is most prominent on 'tightly coupled' calculation on fundamental data types that are known to the cpu (integers, doubles), which is exactly the case for the 2d diffusion.
- It does much less worse on file I/O, text comparisons, etc.
- Calls to compiled libraries and applications (e.g. BLAST) can remove worst performance pitfalls.

- Let's consider a code that numerically computes the following integral:

$$b = \int_{x=0}^3 \left(\frac{7}{10}x^3 - 2x^2 + 4 \right) dx$$

- Exact answer $b = 8.175$
- It's the area under the curve on the right.
- Method: sample $y = \frac{7}{10}x^3 - 2x^2 + 4$ at a uniform grid of x values (using `ntot` number of points), and add the y values.



C++

```
// auc.cpp
#include <iostream>
#include <cmath>
int main(int argc, char** argv)
{
    size_t ntot = atoi(argv[1]);
    double width = 3.0;
    double dx = width/ntot;

    double x = 0, y;
    double a = 0.0;

    for (size_t i=0; i<ntot; ++i) {
        y = 0.7*x*x*x - 2*x*x + 4;
        a += y*dx;
        x += dx;
    }
    std::cout << "The area is "
              << a << std::endl;
}
```

Fortran

```
program auc
    implicit none
    integer :: i, ntot
    character(64) :: arg
    double precision :: dx, width, x, y, a

    call get_command_argument(1,arg)
    read (arg,'(i40)') ntot
    width = 3.0
    dx = width/ntot
    x = 0.0
    a = 0.0
    do i = 1,ntot
        y = 0.7*x**3 - 2*x**2 + 4
        a = a + y*dx
        x = x + dx
    end do

    print *, "The area is ", a
end program
```

Python

```
# auc.py

import sys

ntot = int(sys.argv[1])
width = 3.0
dx = width/ntot

x = 0
a = 0.0

for i in range(ntot):
    y = 0.7*x**3 - 2*x**2 + 4
    a += y*dx
    x += dx

print("The area is %f"%a)
```

```
$ time python auc.py 30000000
The area is 8.17499995268473

real    0m21.000s
user    0m20.983s
sys     0m0.012s
```

If you compare this with the C++ and Fortran versions, Python is about $100\times$ slower.

We want better performance. Where do we start?

Measuring Python Performance (a.k.a. Profiling)

- Performance is about maximizing the utility of a resource.
- Several aspects:
 - ▶ cpu processing power,
 - ▶ memory,
 - ▶ network,
 - ▶ file I/O, etc.

The first order of business is to find out what part of the code makes it slow: the **bottlenecks**.

There is no point optimizing code that only run a small fraction of the total time.

Do not guess, measure it!

Profiling the whole process

- An application that's running is called a **process**.
- OS's have tools to know the cpu and memory usage of a process.
- E.g. Linux has the `time` and `top` commands.

Time Profiling by function

- To consider the computational performance of functions, but not of individual lines in your code, there is the package called `cProfile`.

Time Profiling by line

- To find cpu performance bottlenecks by line of code, there is package called `line_profiler`

Memory Profiling

- To find memory bottlenecks by line of code, there is package called `memory_profiler`

- Use cProfile to know in which functions your script spends its time.
- You usually do this on a smaller but representative case.
- The code should be using separate functions for different tasks, for cProfile to be useful.

Example

```
$ python -m cProfile -s cumulative auc.py 3000000
The area is 8.174999550082383
    4 function calls in 2.083 seconds
```

Ordered by: cumulative time

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	2.083	2.083	{built-in method builtins.exec}
1	2.083	2.083	2.083	2.083	auc.py:1(<module>)
1	0.000	0.000	0.000	0.000	{built-in method builtins.print}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Note: we're using a smaller case (3M instead of 30M points) for profiling.

Still, not much use here, because `auc.py` has no functions.

- Use `line_profiler` to know, line-by-line, where your script spends its time.
- You usually do this on a smaller but representative case.
- First thing to do is to have your code in a function.
- You also need to modify your script slightly:
 - ▶ Decorate your function with `@profile`
 - ▶ Run your script on the command line with

```
$ kernprof -l -v SCRIPTNAME
```

Script before:

```
import sys
ntot = int(sys.argv[1])
width = 3.0
dx = width/ntot
x = 0
a = 0.0
for i in range(ntot):
    y = 0.7*x**3 - 2*x**2 + 4
    a += y*dx
    x += dx
print(f"The area is {a}")
```

Script after:

```
import sys
@profile
def main():
    ntot = int(sys.argv[1])
    width = 3.0
    dx = width/ntot
    x = 0
    a = 0.0
    for i in range(ntot):
        y = 0.7*x**3 - 2*x**2 + 4
        a += y*dx
        x += dx
    print(f"The area is {a}")
main()
```

Run at the command line:

```
$ kernprof -l -v auc.py 3000000
```

Note: After profiling, remove the @profile line to be able to run normally.

```
$ kernprof -l -v auc.py 30000000
```

```
The area is 8.174999550082383  
Wrote profile results to auc.py.lprof  
Timer unit: 1e-06 s
```

```
Total time: 11.7401 s  
File: auc.py  
Function: main at line 2
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
2					@profile
3					def main():
4	1	9.0	9.0	0.0	ntot = int(sys.argv[1])
5	1	1.0	1.0	0.0	width = 3.0
6	1	2.0	2.0	0.0	dx = width/ntot
7	1	1.0	1.0	0.0	x = 0
8	1	1.0	1.0	0.0	a = 0.0
9	3000001	2537750.0	0.8	21.6	for i in range(ntot):
10	3000000	4031917.0	1.3	34.3	y = 0.7*x**3 - 2*x**2 + 4
11	3000000	2665478.0	0.9	22.7	a += y*dx
12	3000000	2504574.0	0.8	21.3	x += dx
13	1	400.0	400.0	0.0	print(f"The area is {a}")

Why worry about this?

Once your script runs out of memory, one of a number of things may happen:

- Computer may start using the harddrive as memory: **very slow**
- Your application crashes
- Your (compute) node crashes

How could you run out of memory?

- You're not quite sure how much memory your program takes.
- Python objects may take more memory than expected.
- Some functions may temporarily use extra memory.
- Python relies on a garbage collector to clean up unused variables.

- This module/utility monitors the Python memory usage and its changes throughout the run.
- Good for catching memory leaks and unexpectedly large memory usage.
- Needs same instrumentation as line profiler.

memory_profiler, details

Your decorated script is usable by memory profiler, but with the command

```
$ python -m memory_profiler auc.py 30000
```

```
The area is 8.174955005754168
```

```
Filename: auc.py
```

Line #	Mem usage	Increment	Line Contents
2	44.621 MiB	44.621 MiB	@profile
3			def main():
4	44.621 MiB	0.000 MiB	ntot = int(sys.argv[1])
5	44.621 MiB	0.000 MiB	width = 3.0
6	44.621 MiB	0.000 MiB	dx = width/ntot
7	44.621 MiB	0.000 MiB	x = 0
8	44.621 MiB	0.000 MiB	a = 0.0
9	44.703 MiB	0.000 MiB	for i in range(ntot):
10	44.703 MiB	0.000 MiB	y = 0.7*x**3 - 2*x**2 + 4
11	44.703 MiB	0.039 MiB	a += y*dx
12	44.703 MiB	0.035 MiB	x += dx
13	44.719 MiB	0.016 MiB	print(f"The area is {a}")

In this case, there is not much going on with memory usage.

Note: memory usage profiling tends to be very slow. Use it only on short test cases.

Ways to improve performance

We know that numpy can speed things up, as long as we use its element-wise 'vector' operations.

E.g., instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in range(100):
    c[i] = a[i] + b[i]
```

And to deal with shifts, instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in range(100):
    c[i] = a[i] + b[i+1]
```

You should write:

```
a = np.linspace(0.0,1.0,100)
b = np.linspace(1.0,2.0,100)
c = a + b
```

You should write:

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = a[0:100] + b[1:101]
```

First, we need to use numpy arrays, but there were no arrays?

Let's store all possible values into an array instead of creating them on the fly.
(sounds inefficient, doesn't it, but we'll try it anyway)

Script before:

```
import sys

def main():
    ntot = int(sys.argv[1])
    width = 3.0
    dx = width/ntot
    x = 0
    a = 0.0
    for i in range(ntot):
        y = 0.7*x**3 - 2*x**2 + 4
        a += y*dx
        x += dx
    print(f"The area is {a}")
main()
```

Script after adding numpy:

```
import sys, numpy

def main():
    ntot = int(sys.argv[1])
    width = 3.0
    dx = width/ntot
    a = 0.0
    x = numpy.linspace(0,width,ntot,
                       endpoint=False)
    for onex in x:
        y = 0.7*onex**3 - 2*onex**2 + 4
        a += y*dx
    print(f"The area is {a}")
main()
```

Script after adding vectorization:

```
import sys, numpy

def main():
    ntot = int(sys.argv[1])
    width = 3.0
    dx = width/ntot
    a = 0.0
    x = numpy.linspace(0,width,ntot,
                       endpoint=False)
    y = 0.7*x**3 - 2*x**2 + 4
    a = y.sum()*dx
    print(f"The area is {a}")
main()
```

Does changing to vectorized numpy really help?

Pure Python implementation:

```
$ time python auc.py 30000000  
The area is 8.17499995268473
```

```
real    0m21.000s  
user    0m20.983s  
sys     0m0.012s
```

Numpy vectorized implementation:

```
$ time python auc.py 30000000  
The area is 8.174999954999999
```

```
real    0m3.201s  
user    0m2.922s  
sys     0m0.266s
```

7× speed-up

Much better!

Note: We can call this vectorization because the code works on whole vectors. But this is different from 'vectorization' which uses the 'small vector units' or 'simd units' on the cpu. We're just minimizing the number of lines Python needs to interpret.

Once the performance of a script is reasonably fast (but not before that!), you can start thinking of using multiple cores in your script, ie., [parallelization](#).

There are many approaches to parallel programming with Python, all of which require external packages. (Parallelism in pure python is extremely limited)

The ones below are some of my favorites:

Package	Functionality
numexpr	threaded parallelization of certain numpy expressions
multiprocessing	create processes that behave more like threads
dask	expression-based parallelism
tensorflow	expression-based parallelism specialized for arrays
mpi4py	message passing between processes

The numexpr package is useful if you're doing matrix algebra:

- It is essentially a just-in-time compiler for NumPy.
- It takes matrix expressions, breaks things up into threads, and does the calculation in parallel.
- In some situations, numexpr can significantly speed up your calculations.

Note: While Python does have threads, there is no convenient OpenMP launching of threads. Even worse: threads running Python do not use multiple cpu cores because of the 'global interpreter lock'.

- Give it an array arithmetic expression, and it will compile and run it, and return or store the output.

- Supported operators:

`+, -, *, /, **, %, <<, >>, <, <=, ==, !=, >=, >, &, |, ~`

- Supported functions:

`where, sin, cos, tan, arcsin, arccos, arctan, arctan2, sinh, cosh, tanh, arcsinh, arccosh, arctanh, log, log10, log1p, exp, expm1, sqrt, abs, conj, real, imag, complex, contains.`

- Supported reductions:

`sum, product`

(don't use these - numpy's `sum` and `product` are faster)

Without numexpr:

```
>>> from time import time
>>> def etime(t):
...     print("Elapsed %f seconds" % (time()-t))
...
>>> import numpy as np
>>> a = np.random.rand(3000000)
>>> b = np.random.rand(3000000)
>>> c = np.zeros(3000000)
>>> t = time(); \
... c = a**2 + b**2 + 2*a*b; \
... etime(t)
Elapsed 0.110790 seconds
```

With numexpr:

```
>>> import numexpr as ne
>>> ne.set_num_threads(1)
>>> t = time(); \
... c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
... etime(t)
Elapsed 0.058771 seconds
>>> ne.set_num_threads(4)
>>> t = time(); \
... c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
... etime(t)
Elapsed 0.019996 seconds
>>> ne.set_num_threads(8)
>>> t = time(); \
... c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
... etime(t)
Elapsed 0.012083 seconds
```

Script before

```
import sys, numpy

def main():
    ntot = int(sys.argv[1])
    width = 3.0
    dx = width/ntot
    a = 0.0
    x = numpy.linspace(0,width,ntot,
                      endpoint=False)
    y = 0.7*x**3 - 2*x**2 + 4
    a = y.sum()*dx
    print(f"The area is {a}")
main()
```

```
$ time python auc.py 30000000
The area is 8.174999954999999
```

```
real    0m3.201s
user    0m2.922s
sys     0m0.266s
```

Script with numexpr

```
import sys, numpy, numexpr

def main():
    ntot = int(sys.argv[1])
    width = 3.0
    dx = width/ntot
    a = 0.0
    x = numpy.linspace(0,width,ntot,
                      endpoint=False)
    y = numexpr.evaluate("0.7*x**3 - 2*x**2 + 4")
    a = y.sum()*dx
    print(f"The area is {a}")
main()
```

```
$ time python auc.py 30000000
The area is 8.174999954999999
```

```
real    0m0.488s
user    0m0.462s
sys     0m0.160s
```

Other Techniques

- Numba allows compilation of selected portions of Python code to native code.
- Decorator based: compile a function.
- It can use multi-dimensional arrays and slices, like NumPy.
- Very convenient.
- Numba can use GPUs, but you're programming them like CUDA kernels, not like OpenACC.
- While it can also vectorize for multi-core and gpus with, it can only do so for specific, independent, non-sliced data.

- Cython is a compiler for Python code.
- Almost all Python is valid Cython.
- Typically used for packages, to be used in regular Python scripts.
- The compilation preserves the pythonic nature of the language, i.e, garbage collection, range checking, reference counting, etc, are still done: *no performance enhancement*.
- If you want to get around that, you need to use Cython specific extensions that use c types. That would be a whole lecture in and of itself.

There are several options to use GPUs in Python.

- PyCUDA: you're writing cuda kernels that are callable from CUDA
- Numba: you're writing more pythonic cuda kernels that are callable from CUDA
- Cupy: replace your numpy array with arrays that live on the GPU. If your code uses vectorized numpy expressions, using cupy can be an easy gain.
- Tensorflow and other AI packages can often use GPUs in the backend.