Lecture 4

Profiling and optimizing

Development Tools for Scientific Computing - SISSA, 2024-2025

Pasquale Claudio Africa, Dario Coscia

20 Feb 2025

Outline

1. Profiling and optimizing
2. Performance boosting

Part of these notes is re-adapted from this lecture and this lecture (license).

Profiling and optimizing

Why can Python be slow?

When developing computer programs today, high-level, human-readable programming languages are typically used, which are then translated into the actual machine instructions that processors execute. This translation can occur in two primary ways:

- **Compiled languages**: The code is translated into machine language prior to execution using a compiler. This method generally results in more efficient execution but limits the flexibility of the program during runtime. The compilation process itself can be time-consuming, which may slow down the rapid testing and development cycle.
- **Interpreted languages**: The code is translated on-the-fly during execution by an interpreter. While this allows for more dynamic and flexible program behavior during runtime, it typically sacrifices performance.

Python falls into the category of interpreted languages, which facilitates rapid development due to its flexibility. However, this very flexibility often comes with the trade-off of decreased performance.

Dynamic typing

Python is dynamically typed, meaning variables are only assigned a type at runtime when they are first assigned a value. This dynamic nature makes it challenging for the Python interpreter to optimize code execution as efficiently as a compiler, which can perform extensive analyses and optimizations beforehand. Although advancements in just-in-time (JIT) compilation techniques have improved runtime optimizations, Python's inherent dynamism still significantly impacts its performance.

Flexible data structures

Python’s built-in data structures like lists and dictionaries are highly flexible, accommodating a wide variety of data types. However, this flexibility makes them less efficient for certain tasks, such as extensive numerical computations. The generic nature of these structures means there is considerable overhead involved when they are used for processing uniform data types.

Python lists vs. NumPy arrays

Profiling

Once your code is working reliably, you can start thinking about optimizing it.

> **Warning:** Always measure the code before you start optimization. Don't base your optimization on theoretical considerations, otherwise you might be surprised.

Profiling is a key technique in software development used to analyze a program's execution to identify resource-intensive parts. It aids in pinpointing performance bottlenecks and provides insights into the runtime behavior of code components. Profiling involves various tools that measure aspects such as execution time, function call frequency, and resource usage. This process helps developers focus their optimization efforts effectively, enhancing the performance and efficiency of applications, especially in complex systems where issues may not be immediately obvious.

time

An easy way to profile a program is to use the `time` function:

```
import time

start_time = time.time()
# Code to profile.
a = np.arange(1000)
a = a ** 2
end_time = time.time()

print(f"Runtime: {end_time - start_time:.4f} seconds")

# Runtime: 0.0001 seconds
```

Timeit

If you're using a Jupyter notebook, using `%timeit` to time a small piece of code is advisable:

```
import numpy as np

a = np.arange(1000)

%timeit a ** 2
# 1.4 µs ± 25.1 ns per loop
```

For long-running calls, consider using `%time` instead of `%timeit`; it's less precise but faster.

cProfile (1/2)

For more complex code, use the built-in Python profilers `cProfile` or `profile`:

```
# walk.py
import numpy as np

def step():
    import random
    return 1. if random.random() > .5 else -1.

def walk(n):
    x = np.zeros(n)
    dx = 1. / n
    for i in range(n - 1):
        x_new = x[i] + dx * step()
        if x_new > 5e-3:
            x[i + 1] = 0.
        else:
            x[i + 1] = x_new
    return x

if __name__ == "__main__":
    n = 100000
    x = walk(n)
```

cProfile (2/2)

```
python -m cProfile -s time walk.py
```

The `-s` switch sorts the results by time. Other options include e.g. function name, cumulative time, etc. However, this will print a lot of output which is difficult to read.

To save the profile to a file, use:

```
python -m cProfile -o walk.prof walk.py
```

The output file can be inspected with `profile pstats module` or profile visualisation tools like Snakeviz or profile-viewer.

Similar functionality is available in interactive IPython or Jupyter sessions with the magic command `%%prun`.

Line-profiler

While `cProfile` indicates which function takes the most time, it does not provide a line-by-line breakdown. For that, you can use `line_profiler`. For line-profiling source files from the command line, we can add a decorator `@profile` to the functions of interests.

```
kernprof -l -v walk.py
```

In Jupyter:

```
%load_ext line_profiler
%lprun -f walk -f step walk(10000)
```

Line-profiler: sample output

```
Wrote profile results to walk.py.lprof
Timer unit: 1e-06 s

Total time: 0.113249 s
File: walk.py
Function: step at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   4                                           @profile
   5                                           def step():
   6     99999      57528.0      0.6     50.8      import random
   7     99999      55721.0      0.6     49.2      return 1. if random.random() > .5 else -1.
```

Based on this output, can you spot a mistake which is affecting performance?

Performance optimization

In software performance optimization, strategies can be broadly categorized into:

- **Algorithm optimization**: Focuses on improving the efficiency of the algorithms used, often by selecting more suitable data structures or algorithms that reduce computational complexity. This typically involves refining the logical structure of the code to perform fewer operations or more efficient ones.
- **CPU optimization**: Aims to enhance the way a program uses the processor's resources to increase execution speed. Techniques might include parallel processing, vectorization, or tuning the code to better fit the CPU's cache and pipelining features.
- **Memory optimization**: Deals with reducing the program's footprint in RAM to prevent memory leaks, reduce paging, and improve cache utilization of data. This can speed up the program by minimizing memory access times.

Algorithm optimization (1/2)

The first step is to review the underlying algorithm. Consider if there are more efficient mathematical approaches or operations that could improve performance.

Example: Singular Value Decomposition (SVD)

```
import numpy as np
from scipy import linalg
data = np.random.random((4000,100))

%timeit np.linalg.svd(data)
# 1.09 s ± 19.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

%timeit linalg.svd(data)
# 1.03 s ± 24.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

%timeit np.linalg.svd(data, full_matrices=False)
# 23.8 ms ± 3.06 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

%timeit linalg.svd(data, full_matrices=False)
# 21.2 ms ± 716 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Algorithm optimization (2/2)

Example: Fibonacci sequence

```
# Recursion.
def fib_rec(n):
    if n < 2:
        return n
    return fib_rec(n-2) + fib_rec(n-1)

# Iteration.
def fib_iter(n):
    a, b = 0, 1
    for i in range(n):
        a, b = a + b, a
    return a

# Caching.
def fib_cached(n, cache={}):
    if n < 2:
        return n
    try:
        val = cache[n]
    except KeyError:
        val = fib_cached(n-2) + fib_cached(n-1)
        cache[n] = val
    return val
```

CPU usage optimization (1/2)

Vectorization

Vectorization implies multiple operations being performed per clock cycle.

```
import numpy as np
a = np.arange(1000)
a_dif = np.zeros(999, np.int64)
for i in range(1, len(a)):
    a_dif[i-1] = a[i] - a[i-1]

# 564 µs ± 25.2 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
import numpy as np
a = np.arange(1000)
a_dif = a[1:] - a[:-1]

# 2.12 µs ± 25.8 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Vectorizing more complex functions

```
import numpy as np
import math

def f(x, y):
    return math.pow(x, 3.0) + 4 * math.sin(y)

f_numpy = np.vectorize(f)

x = np.ones(10000, dtype=np.int8)
f_numpy(x, x)
```

```
import numba # More about Numba later!

@numba.vectorize
def f(x, y):
    return x ** 3 + 4 * np.sin(y)

x = np.ones(10000, dtype=np.int8)
f(x, x)
```

Memory usage optimization

Broadcasting

NumPy operations can be optimized using broadcasting, which allows operations between arrays of different sizes but compatible shapes.

Example (1/3):

```
import numpy as np
a = np.array([1, 2, 3])
b = 4
result = a + b
```

Memory usage optimization

Broadcasting

NumPy operations can be optimized using broadcasting, which allows operations between arrays of different sizes but compatible shapes.

Example (2/3):

```
import numpy as np
a = np.array([[0, 0, 0],[10, 10, 10],[20, 20, 20],[30, 30, 30]])
b = np.array([1, 2, 3])
a + b
```

Memory usage optimization

Broadcasting

NumPy operations can be optimized using broadcasting, which allows operations between arrays of different sizes but compatible shapes.

Example (3/3):

```
import numpy as np
a = np.array([0, 10, 20, 30])
b = np.array([1, 2, 3])
a + b # This does not work!
a[:, None] + b # Or: a[:, np.newaxis] + b
```

Memory usage optimization

Cache effects (1/2)

Memory usage optimization

Cache effects (2/2)

```
# A matrix stored row-wise.
A = np.zeros((10000, 10000), order='C')

%timeit A.sum(axis=0)
# 1 loops, best of 3: 3.89 s per loop

%timeit A.sum(axis=1)
# 1 loops, best of 3: 188 ms per loop

A.strides
# (80000, 8)
```

Memory usage optimization

Temporary arrays

Optimize usage of temporary arrays in NumPy to avoid unnecessary memory consumption.

```
a = np.random.random((1024, 1024, 50))
b = np.random.random((1024, 1024, 50))

# Two temporary arrays will be created.
c = 2.0 * a - 4.5 * b

# Four temporary arrays will be created, and from which two are due to unnecessary parenthesis.
c = (2.0 * a - 4.5 * b) + (np.sin(a) + np.cos(b))

# Solution: apply the operation one by one for really large arrays.
c = 2.0 * a
c = c - 4.5 * b
c = c + np.sin(a)
c = c + np.cos(b)
```

Memory usage optimization

Numexpr

- Evaluation of complex expressions with one operation at a time can lead also into suboptimal performance.
- Numexpr package provides fast evaluation of array expressions.

```
import numexpr as ne
x = np.random.random((10000000, 1))
y = np.random.random((10000000, 1))
%timeit y = ((0.25 * x + 0.75) * x - 1.5) * x - 2
%timeit y = ne.evaluate("((0.25 * x + 0.75) * x - 1.5) * x - 2")
```

- Numexpr tries to use multiple threads (`numexpr.set_num_threads(nthreads)`).
- Supported operations: `+` , `-` , `*` , `/` , `**` , `sin` , `cos` , `tan` , `exp` , `log` , `sqrt` .
- Speedups in comparison to NumPy are typically between 0.95 and 4.
- Works best on arrays that do not fit in CPU cache.

Performance boosting

Performance boosting

After benchmarking and optimizing your code, the next step involves leveraging libraries such as Cython and Numba for pre-compiling functions that are crucial for performance.

Pre-compiling Python

For a majority of applications, employing libraries like NumPy or Pandas suffices. Yet, for certain high-load tasks, it becomes beneficial to pre-compile resource-intensive functions. Cython and Numba are widely favored for these tasks, particularly for their effective handling of NumPy arrays.

Pre-compiling Python

Cython

Cython is a Python dialect that allows C function calls and type declarations on variables and class attributes. Under Cython, your code is translated into optimized C/C++ code and compiled into Python extension modules. This conversion process is facilitated by the `cython` command-line utility that produces a `.c` file from a `.py` file, which then needs to be compiled with a C compiler into an `.so` library. This library can subsequently be imported directly into a Python program. Moreover, Cython can be used directly from Jupyter notebooks via the `%%cython` magic command. For a comprehensive understanding of its capabilities, refer to the official Cython documentation.

Demo: integrating a function in Cython

Consider a basic Python function designed to compute an integral $\int_a^b f(x)\mathrm{d}\,x$, such as the one implemented here.

This function, when applied to data columns within a DataFrame, can be timed for execution performance:

```
import numpy as np
import pandas as pd

df = pd.DataFrame({"a": np.random.randn(1000),
                   "b": np.random.randn(1000),
                   "N": np.random.randint(100, 1000, (1000))})

%timeit apply_integrate_f(df['a'], df['b'], df['N'])
# 321 ms ± 10.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Cython

By translating the function into Cython, and applying data type annotations, you can markedly enhance its execution speed.

In order to use Cython, we need to import the Cython extension:

```
%load_ext cython
```

As a first cythonization step we add the cython magic command with the `-a`, `--annotate` flag, i.e. `%%cython -a`, to the top of the Jupyter code cell.

Cython (1/3)

The yellow coloring in the output shows us the amount of pure Python:

Our task is to remove as much yellow as possible by *static typing*, i.e. explicitly declaring arguments, parameters, variables and functions.

Cython (2/3)

We can start by simply compiling the code using Cython without any changes:

```
%%cython
# Python code for numerical integration.

%timeit apply_integrate_f_cython(df['a'], df['b'], df['N'])
# 276 ms ± 20.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Simply by using Cython and a copy-and-paste gives us about 10% increase in performance.

Cython (3/3)

We can add data and function type annotation:

Now it is over 400 times faster than the original Python implementation, and all we have done is to add type declarations! If we add the `-a` annotation flag we indeed see much less Python interaction in the code.

Numba (1/3)

An alternative to Cython is Numba, a Just-In-Time compiler that translates a subset of Python and NumPy code into fast machine code. Numba is particularly adept at speeding up functions that utilize NumPy arrays, and works by decorating such functions with the `@jit` decorator to indicate they should be JIT compiled. This approach is especially beneficial for functions requiring high performance on large data sets or intensive computations.

Example: using Numba for function optimization

A simple example involves a function designed to perform a computational task on array data:

```
from numba import jit
import numpy as np

@jit
def compute_array(data):
    # Code for computation on array data.
```

Here, the use of Numba's `@jit` decorator helps in compiling this function into optimized machine

Numba (2/3)

```
import numpy as np
import numba

@numba.jit
def f_numba(x):
    return x ** 2 - x

@numba.jit
def integrate_f_numba(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_numba(a + i * dx)
    return s * dx

@numba.jit
def apply_integrate_f_numba(col_a, col_b, col_N):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_numba(col_a[i], col_b[i], col_N[i])
    return res

# Try passing Pandas Series.
%timeit apply_integrate_f_numba(df['a'],df['b'],df['N'])
# 6.02 ms ± 56.5 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

# Try passing NumPy array.
%timeit apply_integrate_f_numba(df['a'].to_numpy(),df['b'].to_numpy(),df['N'].to_numpy())
# 625 µs ± 697 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Numba (3/3)

Numba supports compilation of Python to run on either CPU or GPU hardware and is designed to integrate with the Python scientific software stack. The optimized machine code is generated by the LLVM compiler infrastructure.

Numba is best at accelerating functions that apply numerical functions to NumPy arrays. When used with Pandas, pass the underlying NumPy array of Series or DataFrame (using `to_numpy()`) into the function. If you try to `@jit` a function that contains unsupported Python or NumPy code, compilation will fall back to the object mode which will mostly likely be very slow. If you would prefer that Numba throw an error for such a case, you can do e.g. `@numba.jit(nopython=True)` or `@numba.njit`.

Cython vs. Numba

- The performance between the two is often comparable, though it can vary based on the versions of Python, Cython, Numba, and NumPy being used.
- Numba tends to be simpler to implement, requiring just the addition of the `@jit` decorator.
- Cython offers more stability and has been around longer, whereas Numba is evolving more rapidly.
- Numba supports GPU acceleration, which can be a deciding factor for specific applications.
- Cython allows the compilation of any Python code and can directly interact with C libraries, unlike Numba, which has some limitations.
- Using Numba necessitates having the LLVM toolchain, whereas Cython needs only a C compiler.

Conclusions

NumPy excels within its scope. For straightforward tasks or smaller datasets, neither Numba nor Cython typically offers a significant performance boost over NumPy. However, for more intricate operations, these tools can be incredibly effective.

Binding C++ and Python: pybind11

pybind11 :

- Modern, relevant, and practical for industry demands.
- Header-only library, which simplifies the build process.
- Lightweight, and easy to use.
- Balances ease of use with powerful features.
- Generates more pythonic bindings compared to alternatives.
- Suitable for a range of projects, enhancing problem-solving skills.

⚠️ **Note**: pybind11 may require more manual work for complex bindings.

Creating bindings for a custom type (1/2)

Let's now look at a more complex example where we'll create bindings for a custom C++ data structure named `Pet`.

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    void set_name(const std::string &name_) { name = name_; }
    const std::string &get_name() const { return name; }

    std::string name;
};

PYBIND11_MODULE(example, m) {
    py::class_<Pet>(m, "Pet")
        .def(py::init<const std::string &>())
        .def("set_name", &Pet::set_name)
        .def("get_name", &Pet::get_name);
}
```

Creating bindings for a custom type (2/2)

```
import example

p = example.Pet("Molly")
print(p)
# <example.Pet object at 0x10cd98060>

p.get_name()
# 'Molly'

p.set_name("Charly")
p.get_name()
# 'Charly'
```