Python III: High performance computing

Raul P. Pelaez

April 15, 2026

Contents

1 Introduction

Warning

The conda environment for today
We will be exploring many libraries today, so it is a good idea to create a new conda environment for this session. You may use the following environment file:

```yaml
name: hpc
channels:
  - conda-forge
dependencies:
  - python
  - numpy
  - pandas
  - matplotlib
  - numba
  - pyinstrument
  - pytorch
  - pip
```

In this session we will learn how to use python for high performance computing. For the first time, we are going to focus on how to make our code efficient. Mind you, this is not something that you should

worry about 100% of the time.

We will start by learning about numba, a just-in-time compiler that can speed up your code by orders of magnitude. We will then learn about pytorch, a popular library for deep learning that can also be used for general purpose numerical computing. Finally, we will learn how to profile our code to find bottlenecks and optimize it.

Advice

Early optimization is the root of all evil
Back in the 1960s, Donal Knuth wrote the following in his book "The Art of Computer Programming".

> The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming

This is still relevant today, one of those ancient pieces of wisdom in software development that we seem to constantly forget and rediscover. The idea is that we should focus on writing clear, maintainable code first, and only optimize when (and where) necessary.
This is the reason why this is one of the last lessons in the course.
While it is a good idea to design our software around the idea of it being performant, we should not spend time optimizing code that is not a bottleneck. If we make smart choices during our implementation the code will be performant enough for most cases. For instance, if you are trying to multiply two matrices, you should use `numpy` instead of writing your own matrix multiplication code. On the other hand, modern tools and the Python interpreter are insanely good at optimizing code for us, so instead of trying to outsmart the computer, we should strive to make things as clear as possible for it.

Advice

Testing and optimization
When focusing our developing efforts in performance, it is important to count on a good test suite to make sure that the optimizations do not break any functionality. Whenever we make a change to our code, we should run the tests to make sure that everything is still working as expected. Fast code is useless if it is not correct!

Advice

Optimizing should be the last step
While we should keep performance in mind when designing our software, we should first and foremost focus on writing code that is simple (a.i. the opposite of complex), clear, and maintainable. After everything else is done, if we find a bottleneck, we can optimize it. So before optimizing, ask yourself:

- Does the project have a thorough test suite?
- Is the code clear and maintainable?

Joel Armstrong, one of the creators of Erlang, used to say:

> Make it work, then make it beautiful, then if you really, really have to, make it fast. 90 percent of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful!

2 Basics of profiling

Info

Profiling
Profiling is the process of measuring various aspects of a program's execution to identify areas where performance can be improved. It involves collecting data on function call frequencies, execution times, memory usage, and more.

Advanced

We can distinguish between two types of profiling:

- **Deterministic profiling**: Measures the time it takes to run a piece of code.
- **Statistical profiling**: Samples the program's state at regular intervals to determine where time is being spent.

Statistical sampling is achieved with tools like `cProfile` or `pyinstrument`.

2.1 Deterministic profiling: Measuring time

`time` is a library that can be used to measure the time it takes to run a piece of code. It is useful for measuring the performance of small pieces of code, but it is not suitable for measuring the performance of entire programs. For example

```python
import time

start = time.time()
# do some work
time.sleep(0.5)
end = time.time()
print(f"Time: {end - start:.2f} s")
```

Output

```
Time: 0.50 s
```

`timeit` is another library that can be used to measure the time it takes to run a piece of code. It is similar to `time`, but it runs the code multiple times and returns the average time it took to run it. For example:

```python
from timeit import timeit
import time

def some_foo():
      time.sleep(0.05)
iterations = 10
t = timeit(some_foo, number=iterations)
print(f"Time: {t/iterations:.2f} s")
```

Output

```
Time: 0.05 s
```

`timeit` will run the function `some_foo` 10 times and return the time it took to run them. Running the snippet several times is important because the time it takes to run a piece of code can vary depending on the state of the system. For instance, if the system is under heavy load, the code will take longer to run.

Advice

Warming up
Another good practice when measuring the time it takes for a piece of code to run is ”warming up”, which consists of running the code to profile several times before actually measuring the time it takes to run it. The first time a piece of code is run, the Python interpreter will do some things, like inspecting it and storing some information about it in a cache. This can take some time, and it can skew the results of the measurement.

Advice

Magic commands in Jupyter notebooks

- `%timeit` is a magic command that can be used to measure the time it takes to run a cell. It runs the cell multiple times and returns the average time it took to run it.
- `%time` is another magic command that can be used to measure the time it takes to run a cell. It only runs the cell once and returns the total time it took to run it.

Try placing `%timeit` or `%time` at the beginning of a cell in a Jupyter notebook.

2.2 Statistical profiling: pyinstruments

Statistical profiling is a technique that samples the program's state at regular intervals to determine where time is being spent. Think about it this way, if we take a snapshot of the program's state every 10ms, we can see where the program was at that time. If we do this many times, we can get a good idea of where the program spends most of its time. This is done by using a profiler, which is a tool that collects data about the program's execution. There are many profilers available for Python, but one of the most popular ones is `pyinstrument`.

Example

```python
from pyinstrument import Profiler
import time

profiler = Profiler()
profiler.start()

# do some work
time.sleep(0.5)

profiler.stop()
print(profiler.output_text())
```

Output

```
  _     ._   __/__   _ _  _  _ _/_   Recorded: 11:48:07  Samples:  1
 /_//_/// /_\ / //_// / //_'/ //     Duration: 0.501     CPU time: 0.000
/   _/                      v5.1.2

Profile at <stdin>:5

0.500 <module>  <stdin>:1
`- 0.500 sleep  <built-in>
```

The output shows the time spent in each function, as well as the number of samples taken. In this case, we can see that most of the time was spent in the `sleep` function, which is expected. The profiler also

shows the call stack, which can be useful for identifying where the program spends most of its time. Lets see a more interesting example with a snippet that process a dataset with pandas:

```
import pandas as pd
from pyinstrument import Profiler

movies = pd.read_csv('https://raw.githubusercontent.com/WPNderi/IMDBData/refs/heads/master/movies
movies['movie_id'] = movies['id']
genres = pd.read_csv('https://raw.githubusercontent.com/WPNderi/IMDBData/refs/heads/master/movies
df = pd.merge(movies, genres, on='movie_id')
print(df.head())
with Profiler() as profiler:
    df.dropna(subset=['genre', 'rank'], inplace=True)
    df.drop_duplicates(inplace=True)
    df['genre'] = df['genre'].str.split(', ')
    df_exploded = df.explode('genre')
    average_ratings = df_exploded.groupby('genre')['rank'].mean()
profiler.print()
```

Output

```
   id                                    name  year  rank  movie_id        genre
0   1  #7 Train: An Immigrant Journey, The  2000   NaN         1  Documentary
1   1  #7 Train: An Immigrant Journey, The  2000   NaN         1        Short
2   2                                     $  1971   6.4         2       Comedy
3   2                                     $  1971   6.4         2        Crime
4   5                         $1,000 Reward  1923   NaN         5      Western

  _      ._   __/__   _ _  _  _ _/_   Recorded: 11:48:17  Samples:  46
 /_//_/// /_\ / //_// / //_'/ //     Duration: 0.070     CPU time: 0.069
/   _/                      v5.1.2

Profile at <stdin>:9

0.069 <module>  <stdin>:1
├─ 0.037 StringMethods.wrapper  pandas/core/strings/accessor.py:134
│     [22 frames hidden]  pandas, <built-in>, re
│        0.028 <lambda>  pandas/core/strings/object_array.py:384
│        └─ 0.018 [self]  pandas/core/strings/object_array.py
├─ 0.013 DataFrame.drop_duplicates  pandas/core/frame.py:7847
│     [8 frames hidden]  pandas
├─ 0.008 DataFrame.dropna  pandas/core/frame.py:7654
│     [32 frames hidden]  pandas
├─ 0.005 DataFrame.explode  pandas/core/frame.py:11364
│     [17 frames hidden]  pandas
├─ 0.004 SeriesGroupBy.mean  pandas/core/groupby/groupby.py:2192
│     [12 frames hidden]  pandas
├─ 0.001 [self]  <stdin>
└─ 0.001 DataFrameGroupBy.__getitem__  pandas/core/groupby/generic.py:2929
      [10 frames hidden]  pandas, <built-in>
```

Prior to the profile we might not have a clear idea of which of the operations is taking the most time.

The profiler shows us that splitting the genre column is taking the most time, followed by dropping duplicates and exploding the dataframe. This is useful information that can help us optimize our code. Perhaps in this case we can drop duplicates before exploding the dataframe, or we can use a different method to split the genre column.

Advice

Inspecting the profile data
Besides printing a summary, `pyinstruments` can also generate reports in several other ways. For instance, if we call `profiler.open_in_browser()`, it will open an interactive report in the browser.

Besides helping with snippets, we can use `pyinstrument` to profile entire scripts. For instance, we can run the following command in the terminal:

```
python -m pyinstrument my_script.py
```

This will run the script and generate a report in the terminal for the entire application.

3 Performance considerations

Advice

”””””Python is slow”””””
Python is an interpreted language, which means that it is not compiled to machine code before it is run. Instead, each instruction is *interpreted* by another program (the Python interpreter) at runtime. In essence, there is a middleman between our code and the CPU, which makes Python slower than compiled languages like C or Fortran.
While this is true, the Python interpreter is incredibly good at optimizing code and is most probably better than you at it for most cases.

While it is true that Python ”is slow”, that does not mean that it is not suitable for high performance computing. In fact, Python is used in many high performance computing applications, including machine learning, scientific computing, and more. For instance, many world-leading LLMs are implemented in Python, using libraries like `PyTorch` and `jax`. Python is also the language of choice for many Data Scientists, who use it to analyze large datasets (e.g. using `pandas`).

So how do we write highly performant code in a ”slow” language? The main gist of it is to use Python as a driver for highly optimized libraries written in other languages. For instance, we can use `numpy` for numerical computing, `scipy` for scientific computing, `pandas` for data analysis, and `PyTorch` for deep learning. These libraries are written in languages like C, C++, or even CUDA. For instance, this is quite slow:

```python
import numpy as np
from timeit import timeit

def matmul(a, b):
    """Multiply two matrices, return the result"""
    assert a.shape[1] == b.shape[0]
    n = a.shape[0]
    m = b.shape[1]
    c = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            for k in range(len(b)):
                c[i][j] += a[i][k] * b[k][j]
    return c

a = np.random.rand(100, 100)
b = np.random.rand(100, 100)
c = matmul(a, b)
t = timeit(lambda: matmul(a, b), number=10)
print(f"Time Naive: {t/10:.2f} s")
```

Output

```
Time Naive: 0.28 s
```

In this function, the interpreter must process each instruction in the inner loop sequentially, which is slow. Instead, we can use `numpy` to speed things up:

```python
import numpy as np
from timeit import timeit

a = np.random.rand(100, 100)
b = np.random.rand(100, 100)
t = timeit(lambda: np.matmul(a, b), number=10)
print(f"Time Numpy: {t/10:.6f} s")
```

Output

```
Time Numpy: 0.000108 s
```

This is less code, its clearer and many orders of magnitude faster. To me, the true power of Python is not in its speed, but in its ability to glue together highly optimized libraries written in other languages. This is why Python is so popular in scientific computing and machine learning. These Python wrappers are also much easier to use. Compare `np.matmul(a,b)` with the C function that `numpy` is using to multiply two matrices, which comes from a library called `BLAS`, this is its signature:

```c
int dgemm(char *transa, char *transb, int *m, int *n, int *k, double *alpha,
      double *a, int *lda, double *b, int *ldb, double *beta, double *c, int *ldc);
```

I am not joking, calling this function (which is named "Double precision GEneral Matrix Matrix multiplication"), which accomplishes the same thing as `np.matmul` (documentation here) , truly requires 13 arguments with cryptic names and purposes. See its documentation. The cognitive load is so much higher!

Advice

Focus on algorithmic complexity
Your first priority when trying to make code run faster should be to focus on algorithms. For instance, if you are trying to sort a vector, use `np.sort` instead of writing a naive $O(N^2)$ version yourself. Try to identify the places in your logic where $O(N^2)$ or $O(N^3)$ algorithms can be replaced with $O(N \log N)$ or $O(N)$ algorithms.

Advice

The interpreter and tooling know better than you
Modern compilers and tools (like `Numba` or the Python interpreter) are incredibly good at optimizing code. They can do things like function inlining, loop unrolling, and vectorization automatically. When allowed, compilers will reorganize your code in insane ways to make it faster. Never ever try to outsmart the tools. Instead, write clear, maintainable code and let the tools do their job.

4 Numba

Info

Numba
A just-in-time compiler for Python that can speed up your code by orders of magnitude. It works by compiling Python code to machine code at runtime, which can be much faster than the Python interpreter. Numba can be used to speed up numerical code, but it can also be used to speed up other types of code as well.

Info

Just-in-time (JIT) compilation
JIT consists of compiling code at runtime, rather than ahead of time. This allows the compiler to optimize the code based on the current state of the program, which can result in faster code. In practical terms, this means that the first time a function is called, it will be compiled to machine code. The next time the function is called, the compiled code will be used instead of the Python interpreter. This can result in significant speedups, especially for numerical code.

4.1 Basic usage

To use Numba, you need to import the `jit`, or `njit`, function from the `numba` module and decorate the functions you want to speed up with it. Numba will then compile the function to machine code and cache the result so that it can be reused in future calls.

```python
from numba import njit
import numpy as np

@njit
def process_array(a, b):
    c = np.zeros_like(a)
    for i in range(len(a)):
        a[i] = b[i] + c[i]
    return c

n = 100000
a = np.arange(n)
b = np.random.rand(n)
process_array(a, b)
```

Note tat the only thing we had to do was to add the `njit` decorator to the function. Numba will take care of the rest.

Lets try the matmul example from before with Numba:

```python
import numpy as np
from timeit import timeit
from numba import njit

@njit
def matmul(a, b):
    """Multiply two matrices, return the result"""
    assert a.shape[1] == b.shape[0]
    n = len(a)
    m = len(b[0])
    c = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            for k in range(len(b)):
                c[i][j] += a[i][k] * b[k][j]
    return c

a = np.random.rand(100, 100)
b = np.random.rand(100, 100)
c = matmul(a, b)
t = timeit(lambda: matmul(a, b), number=10)
print(f"Time Numba: {t/10:.6f} s")
```

Output

```
Time Numba: 0.000405 s
```

Still substantially slower than `np.matmul`, but much faster than the naive implementation. Sometimes your logic will be hard to port into your known libraries, in these cases we can always write our functions in pure, simple, Python and then use Numba to speed them up.

Advanced

`jit` vs `njit`
`njit` is equivalent to `jit(nopython=True)`. It tells Numba to process the function in `nopython` mode, meaning that it is required to compile the function without using the Python interpreter. This is more restrictive, if Numba cannot manage to understand the code it will throw an error. On the contrary, `jit` will fall back to using the Python interpreter if it cannot compile the function.

Advanced

Numba goes much further away than this "automagical" `jit` compilation. We can use it to write CUDA code (which will run in a Graphical Processor Unit), or to write code that runs in parallel in multiple cores.

5 Pytorch

Info

PyTorch is a popular library for deep learning that can also be used for general purpose numerical computing. It is similar to numpy, but it is optimized for deep learning and can run on GPUs. Pytorch is the library of choice for companies like OpenAI and Meta. In other words, ChatGPT is powered by this library.

5.1 Basic usage

PyTorch is similar to numpy, but it has some key differences. For instance, PyTorch uses tensors instead of arrays, and it has a different API. Here is an example of how to use PyTorch to multiply two matrices:

```python
import torch
from timeit import timeit

a = torch.rand(100, 100)
b = torch.rand(100, 100)
c = torch.matmul(a, b)
t = timeit(lambda: torch.matmul(a, b), number=10)
print(type(c))
print(f"Time PyTorch: {t/10:.6f} s")
```

Output

```
<class 'torch.Tensor'>
Time PyTorch: 0.000052 s
```

Notice how the syntax is identical to the one in `numpy`. Pytorch tries to be a drop-in replacement for `numpy`. On the other hand, Pytorch is optimized for deep learning, so it has many features that are not present in `numpy`. For instance, it has automatic differentiation, which is essential for training neural networks. Additionally, in this particular case, it turns out to be much faster than `numpy`.

Another killer feature of Pytorch is that it can run on GPUs. This is a game changer for deep learning, as GPUs are much faster than CPUs for certain types of computations. For instance, lets make the previous example run on a GPU:

```python
import torch

a = torch.rand(100, 100, device="cuda")
b = torch.rand(100, 100, device="cuda")
c = torch.matmul(a, b)
```

That is all it takes! I do not dare show you the CUDA/C++ code that is being run by Pytorch here.

5.2 Deep learning in Pytorch

Let me show you now how to use Pytorch to implement a simple neural network. We will implement a simple multi-layer perceptron (MLP), which models a sequence of connected artificial neurons.

Advanced

Multi layer perceptron
A multi-layer perceptron (MLP) is one of the simplest types of neural networks. It consists of an input layer, one or more hidden layers, and an output layer. Each layer is made up of neurons, which are connected to the neurons in the previous and next layers. The connections between the neurons have weights, which are adjusted during training to minimize the error of the network. Mathematically, the output of a layer is given by the following equation:

$$y = f(w^T x + b)$$

Where x is the input tensor, w is the weight tensor, b is the bias, and f is the activation function. The activation function is a non-linear function that determines the output of the neuron. Common activation functions are sigmoid, tanh, and ReLU. By stacking multiple layers together, we can create a deep neural network that can learn complex functions. The MLP is a universal approximator, meaning that it can approximate any continuous function given enough neurons and layers. For instance, with two layers the expression for the output would be:

$$y = f(w_2^T f(w_1^T x + b_1) + b_2)$$

```python
import torch
import torch.nn as nn

class SigmoidActivation(nn.Module):
    def __init__(self):
        super(SigmoidActivation, self).__init__()

    def forward(self, x):
        return 1 / (1 + torch.exp(-x))

class Linear(nn.Module):
    def __init__(self, input_size, output_size):
        super(Linear, self).__init__()
        self.weight = nn.Parameter(torch.randn(output_size, input_size))
        self.bias = nn.Parameter(torch.randn(output_size))

    def forward(self, x):
        return torch.matmul(self.weight, x) + self.bias

mlp = nn.Sequential(
    Linear(1, 10),
    SigmoidActivation(),
    nn.Linear(10, 10),
    SigmoidActivation(),
    nn.Linear(10, 1),
    SigmoidActivation(),
)
sample_data = torch.tensor([-1.0])
output = mlp(sample_data)

print(f"Input: {sample_data}")
print(f"MLP output: {output}")
```

Output

```
Input: tensor([-1.])
MLP output: tensor([0.4896], grad_fn=<MulBackward0>)
```

Note how each of out custom layers is a subclass of `nn.Module`. This is the base class for all neural network modules in Pytorch. It provides a lot of functionality out of the box, like automatic differentiation for training the models. This is an example of inheritance in real-life code. Note how we implement the `forward` method, the base class will take care of translating that (via inheritance) into a call to the `__call__` method, which is the one that will be called when we feed data to the model. On the other hand, by making our custom layers subclasses of `nn.Module`, we can make use of all the goodies that pytorch provides for Modules, like in this case `nn.Sequential`, which allows us to stack multiple layers together in a single module. This is a common pattern in Pytorch, and it allows us to create complex models by composing simple modules together.

Our network is not very interesting at the moment. Given that the weights are initialized at random, the output will also be random. In order to tailor or adapt our model to a specific task we would need to train it. Although beyond the scope of this course, Pytorch provides a lot of functionality for training models, including optimizers, loss functions, and data loaders. Possible tasks for this MLP could be to learn a function that maps a number to its square, or classify numbers as even or odd.

6 Exercises

6.1 The timing decorator

Goal

Write a decorator that can be used to log the time it takes to run a function. The decorator should make it so that after a function is called:

- The time of the call is printed to the console
- The time it took to run the function is printed to the console
- The name of the function and the arguments passed to it are printed to the console

Once your decorator is implemented, it should work like this:

```python
import time
@timing
def foo(x,  y):
    time.sleep(0.5)
    return x + y

foo(1, 2)
```

Output

```
12:19:31 - Function foo called with args: (1, 2), kwargs: {} -- took 0.51 s
```

6.2 Scheduler generator

Goal

Write a generator that can be used to schedule tasks. The generator should take a time in seconds and yield the time elapsed since its creation. The generator should be used like this:

```python
for t in scheduler(seconds=1):
    print(f"Elapsed time: {t}")
    # do some work
    if t>5:
        break
```

Output

```
Elapsed time: 0.0
Elapsed time: 1.0017039775848389
Elapsed time: 2.006760835647583
Elapsed time: 3.011340856552124
Elapsed time: 4.016228914260864
Elapsed time: 5.02060079574585
```

We could use this generator to schedule tasks in a timely manner. For instance, to gather data from a sensor or an API every X seconds, or to log some event.

6.3 What is taking so long?

Goal

Write a function that takes a list of numbers and returns the sum of the squares of the numbers. The function should be implemented in several different ways:

- Using a naive for loop
- Using a list comprehension
- Using a generator expression
- Using a numpy array
- Using a numba function (using `@njit` in a copy of the for-loop version)

Profile these functions in two ways:

- Using `timeit`
- Using `pyinstrument`

Are these profiling methods having any effect on the performance of the code? Is the fastest implementation the same for both profiling methods and the one you expected?