**GitHub：**https://github.com/DayBeha/HPC\_assignment

Exercise 1 - List, Tuples, array, and NumPy

Answer the following questions:

1. What is the advantage of using Lists vs. Tuples

**Lists** are dynamic arrays that let us modify and resize the data we are storing, while **Tuples** are static arrays whose contents are fixed and immutable. This means that once a tuple is created, unlike a list, it cannot be modified or resized.

1. What is the advantage of using the array module vs. Python lists?

**array modules** have a static type and can store only that type of data, While type of elements in **Python lists** can be different, because lists only store 8-byte pointer to the actual object. But **array** modules is just a thin wrapper on C arrays. when storing same amount of data, **array** modules will use less space than **Python lists**.

**array modules** store data sequentially in memory, so that a slice of the array actually represents a continuous range in memory.

1. What are the memory fragmentation problem and Von-Neumann bottleneck? How do they affect the performance of a code? How can we try to address it?

**Memory fragmentation problem**: when our data is fragmented, we must move each piece over individually instead of moving the entire block over. This means we are invoking more memory transfer overhead, and we are forcing the CPU to wait while data is being transferred.

We can alleviate **Memory fragmentation problem** by **using the array module instead of lists**.

And for any loop that does arithmetic on our array one element at a time to work on chunks of data.

**Von-Neumann bottleneck:** This refers to the limited bandwidth that exists between the memory and the CPU as a result of the tiered memory architecture that modern computers use.

To address **Von-Neumann bottleneck,** CPU try to predict the next instruction and load the relevant portions of memory into the cache while still working on the current instruction. And the best way to minimize the effects of the bottleneck is to be smart about how we allocate our memory and how we do our calculations over our data.

1. What is a page fault? What is the difference between a minor and a major page fault?
2. A **page-fault** is part of the modern memory allocation scheme.
3. When memory is first used, the OS throws a **minor page fault** interrupt, which pauses

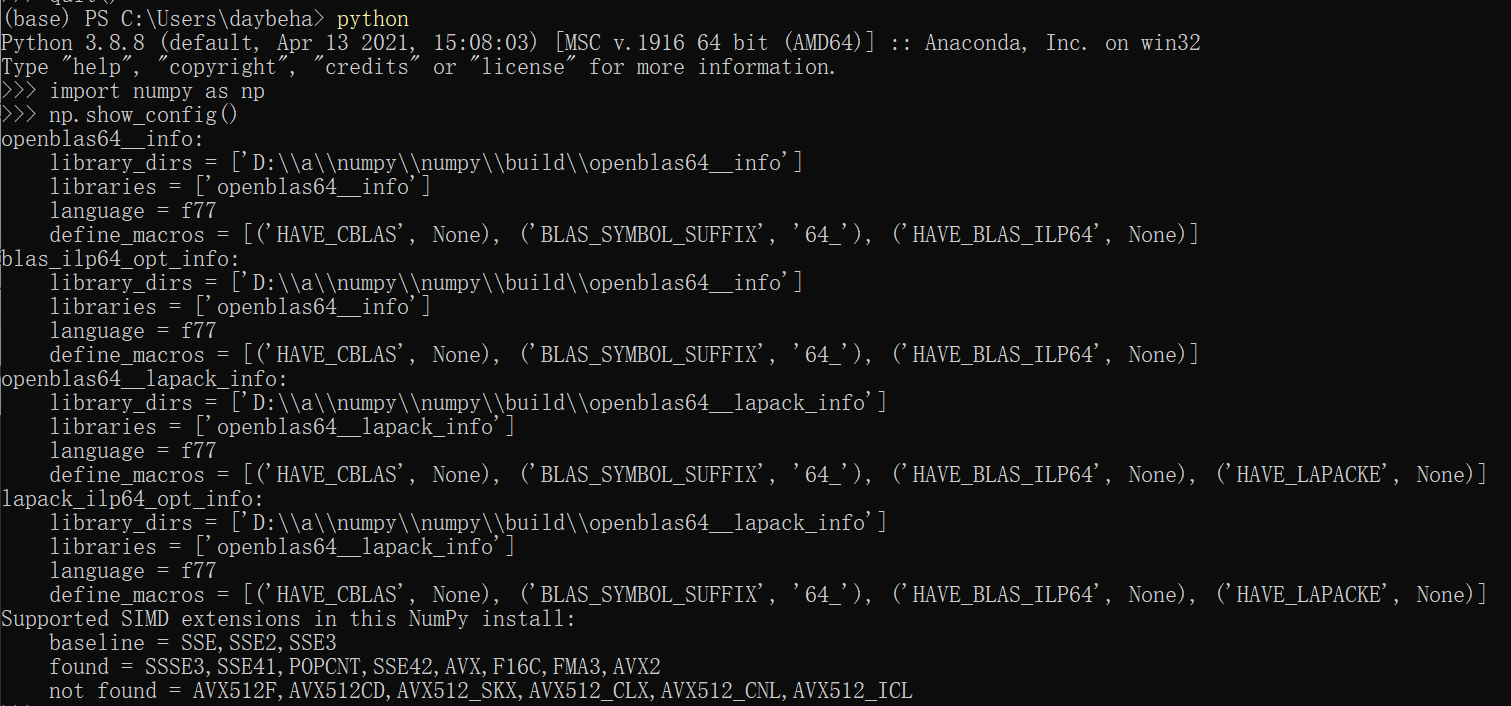
the program that is being run and properly allocates the memory; **Major page fault** happens when the program requests data from a device(disk, network, etc.) that hasn’t been read yet. These are even more expensive operations: not only do they interrupt your program, but they also involve reading from whichever device the data lives on.

1. What is the impact of a cache miss on the performance?

Cache misses can be a source of slowdowns, since we need to wait to fetch the data from

RAM andwe interrupt the flow of our execution pipeline

1. Which HPC libraries does your NumPy installation use? *Hint:*you can check by writing a simple code.

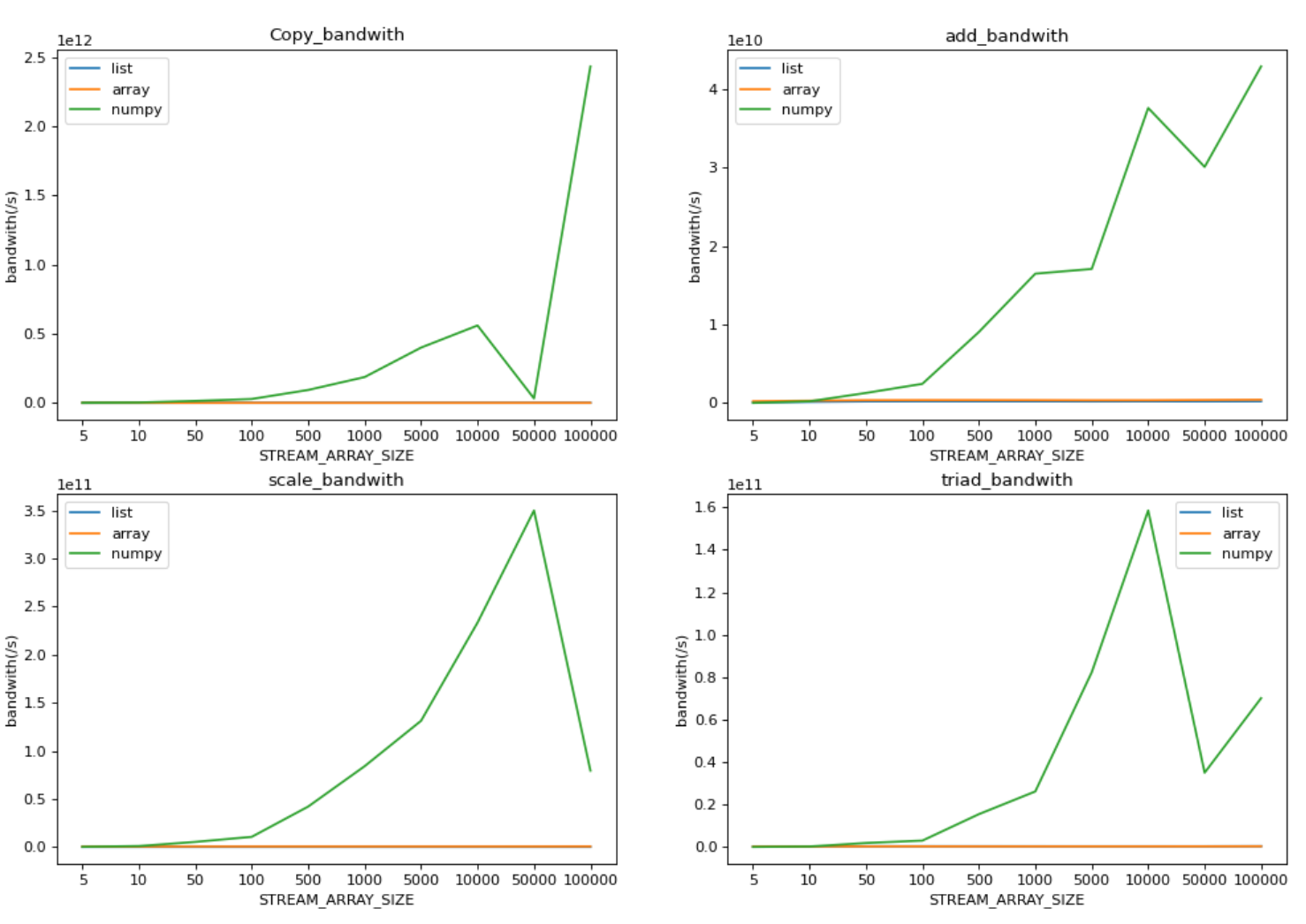


OpenBLAS64

Exercise 2 - STREAM Benchmark in Python to Measure the Memory Bandwidth

**Task 2.1** Implement in Python the STREAM benchmark using Python lists, arrays from the array module, and NumPy arrays.

**Task 2.2**Measure the bandwidth for the three Python array implementations (lists, array and numpy) varying the STREAM\_ARRAY\_SIZE and plot the results. Answer the questions: How does the bandwidth vary when increasing the STREAM\_ARRAY\_SIZE and why? How do the different implementation bandwidths compare to each other?



As shown above, bandwith of list and array are very close, and bandwith of numpy is lowest when STREAM\_ARRAY\_SIZE is low while far beyond list and array as STREAM\_ARRAY\_SIZE grows. The beginning is lowest is because numpy need to transfer values to object which increases the expense. But as STREAM\_ARRAY\_SIZE grows, the advantage of contiguous memory in numpy show up.

Exercise 3 - PyTest with the Julia Set Code

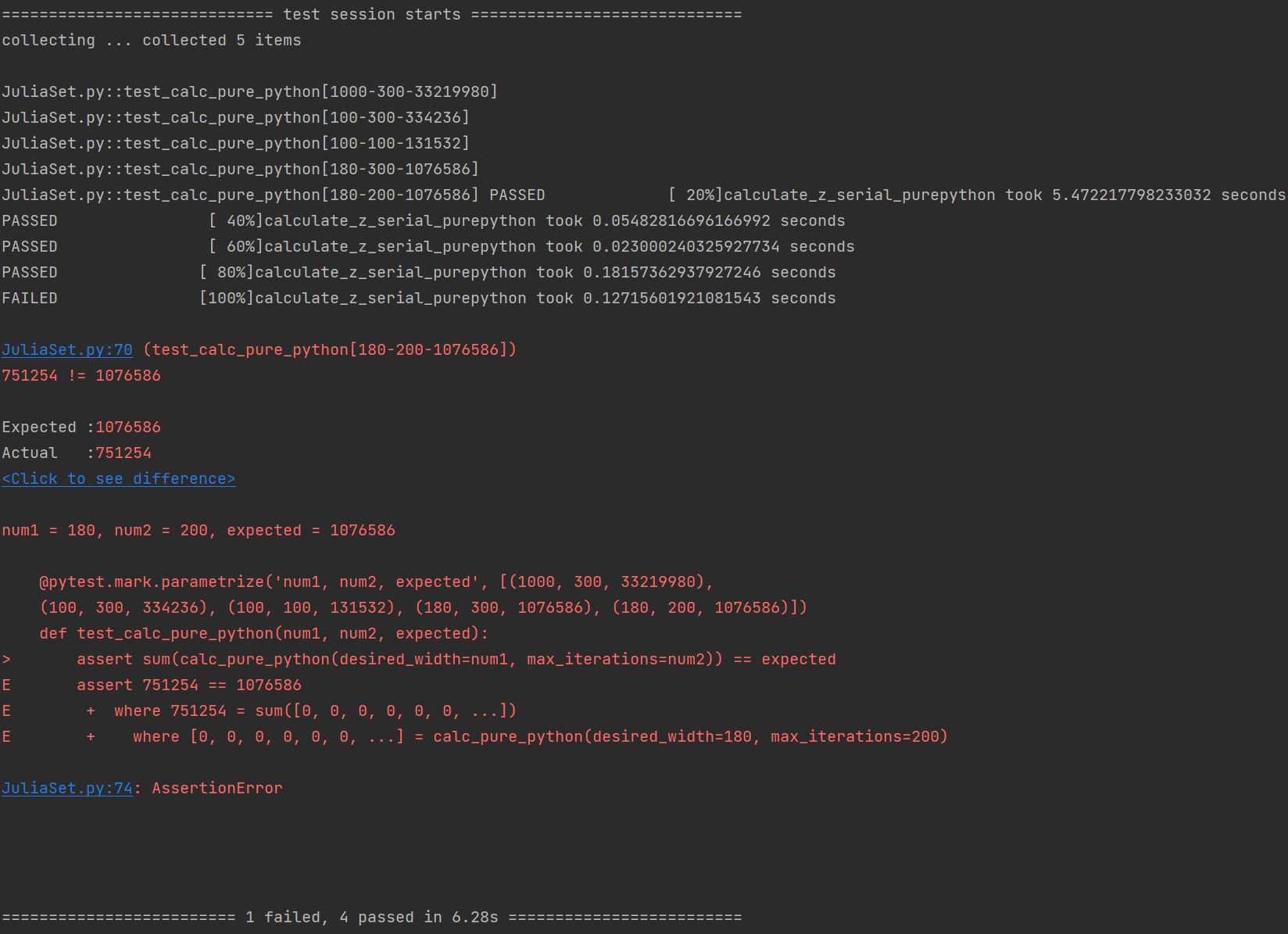
**Task 3.1** Implement a separate code to test the assertion above using the pytest framework.

**Code:**

import pytest  
@pytest.mark.parametrize('num1, num2, expected', [(1000, 300, 33219980),  
(100, 300, 334236), (100, 100, 131532), (180, 300, 1076586), (180, 200, 1076586)])  
def test\_calc\_pure\_python(num1, num2, expected):  
 assert sum(calc\_pure\_python(desired\_width=num1, max\_iterations=num2)) == expected

**Run:** pytest .\JuliaSet.py

**Output:**



**Task 3.2** How would you implement the unit test with the possibility of having a different number of iterations and grid points? Implementation is optional.

If we know the result, we can just test by it. If we don’t, we can caculate the shape of output as usual. So we can test by the output shape.

Exercise 4 - Python DGEMM Benchmark Operation

Answer the following questions:

* For which kind of problems do you use the BLAS libraries ?

When we need to calculate a large amount of number or matrix, and have requirements of time consuming.

* What is the difference between BLAS level-1, level-2 and level-3?

BLAS level-1 were limited to vector operations;

Level-2 provide routines for matrix-vector;

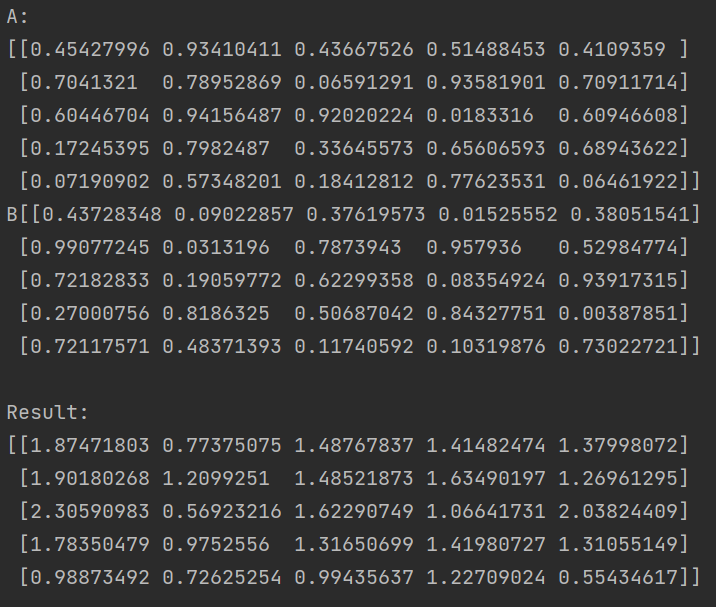
Level-3 provide routines for matrix-matrix;

**Task 4.1** Implement the DGEMM with matrices as NumPy array

**Code**:

import numpy as np  
  
N = 5  
  
A = np.random.random((N, N)).astype(np.float64)  
B = np.random.random((N, N)).astype(np.float64)  
print(f"A:\n{A}\nB:\b{B}\n")  
  
C = np.zeros\_like(A)  
# Multiplying first and second matrices and storing it in result  
for i in range(N):  
 for j in range(N):  
 for k in range(N):  
 C[i][j] += A[i][k] \* B[k][j]  
  
print(f"Result:\n{C}")

**Output:**

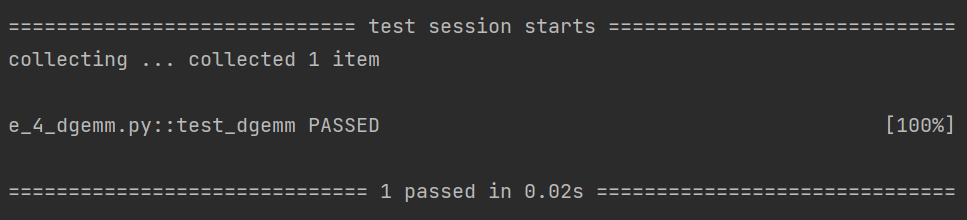


**Task 4.2**Using pytest develop a unit test for checking the correctness of your implementations.

**Code:**

import numpy as np  
  
N = 5  
A = np.ones((N,N)).astype(np.float64)  
B = np.ones((N,N)).astype(np.float64)  
print(f"A:\n{A}\nB:\b{B}\n")  
  
def dgemm(A, B):  
 C = np.zeros\_like(A)  
 # Multiplying first and second matrices and storing it in result  
 for i in range(N):  
 for j in range(N):  
 for k in range(N):  
 C[i][j] += A[i][k] \* B[k][j]  
  
 print(f"Result:\n{C}")  
 return C  
C = dgemm(A,B)  
  
import pytest  
def test\_dgemm():  
 assert (C==np.ones((N,N))\*5).all()

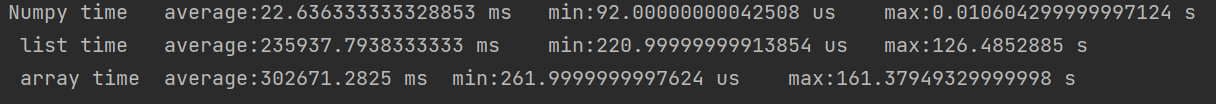
**Output:**

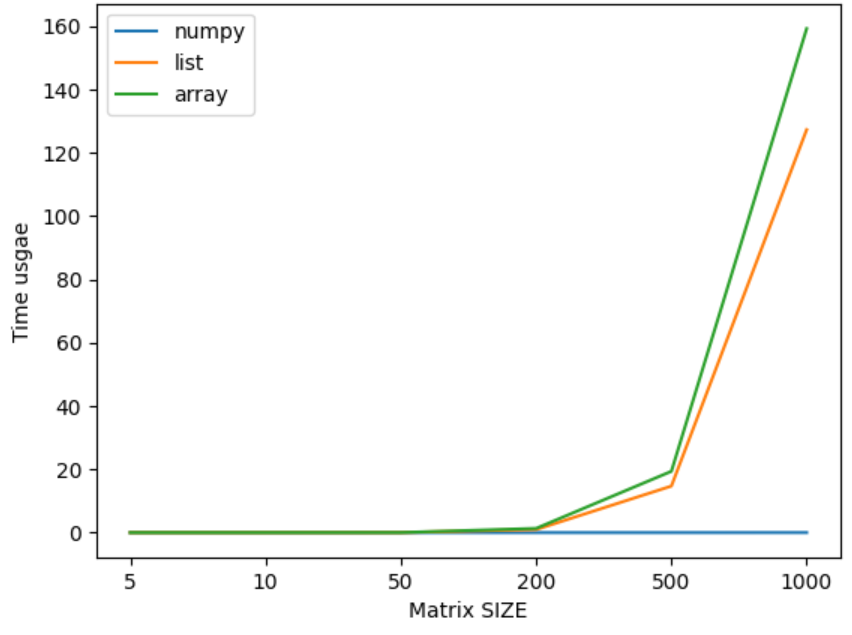


**Task 4.2** Measure the execution time for each approach varying the size of the matrix. Report the average and error (std. deviation or min/max or interval of confidence). Answer the question: how the computational performance varies with increasing the size of the matrices and why so?

**Task 4.3** Using the timing information and the number of operations for the DGEMM, calculate the FLOPS/s. How many operations are carried out in DGEMM with N as the matrices dimension? *Hint:* Think about the number of iterations completed in the loops and the number of flops per iteration. How do the FLOPS/s you measured compares to the theoretical peak of your processor (if we assume that we do one operation per cycle, then the peak is the clock frequency value)

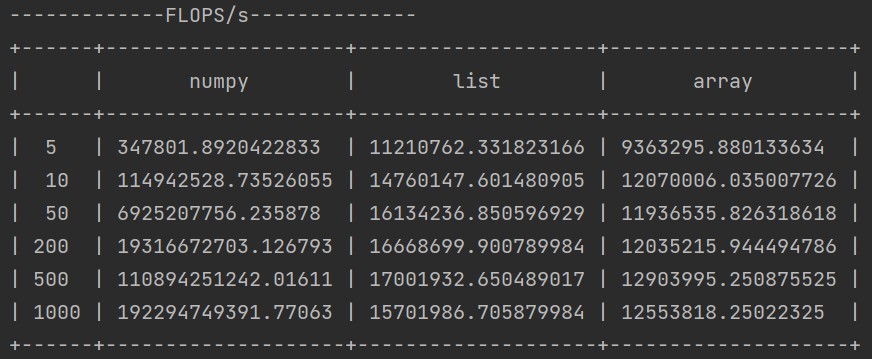
Set N as [5,10,50,200,500,1000]





Answer4.2: As the size of matrices increase, the time usage non-linear increases in list and array. That is because the memory can no longer hold the values contiguous, which increases cache-miss.

Answer4.3: The number of operations in DGEMM with N as the matrices dimension is



The clock frequency of my CPU is 3.2GHz=

It can be seen that as N grows, the FLOPS/s of numpy grows quickly and far beyond the theoretical peak of my processor if we assume one operation per cycle.

Exercise 5 - A Python Discrete Fourier Transform

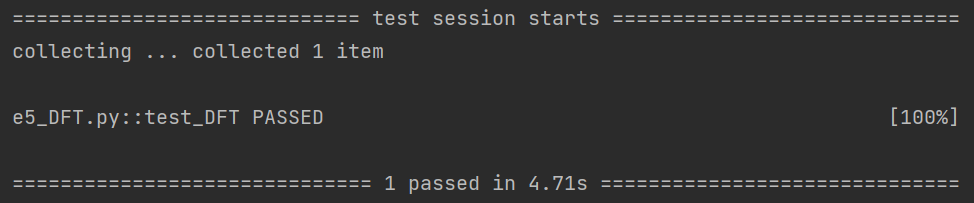
**Task 5.1** Develop a DFT in Python and a unit test with pytest to check the calculation's correctness. Also, use the Python logging module to log the results. The data structures (lists, array, or NumPy) are of your choice.

Code:

N = 1024 # 采样点数  
sample\_freq = 120 # 采样频率 120 Hz, 大于两倍的最高频率  
# sample\_interval = 1 / sample\_freq # 采样间隔  
signal\_len = N / sample\_freq # 信号长度  
t = np.arange(0, signal\_len, 1 / sample\_freq)  
  
signal = 3 \* np.sin(2 \* np.pi \* 20 \* t) # 采集的信号

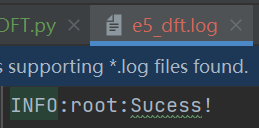
def DFT(xr, xi):  
 N = len(X)

X = np.zeros(N, np.complex\_)  
 Xr\_o = X.real  
 Xi\_o = X.imag  
 for k in range(N):  
 for n in range(N):  
 # Real part of X[k]  
 Xr\_o[k] += xr[n] \* np.cos(n \* k \* 2 \* np.pi / N) + xi[n] \* np.sin(n \* k \* 2 \* np.pi / N)  
 # Imaginary part of X[k]  
 Xi\_o[k] += -xr[n] \* np.sin(n \* k \* 2 \* np.pi / N) + xi[n] \* np.cos(n \* k \* 2 \* np.pi / N)  
  
 return X  
  
import pytest  
def test\_DFT():  
 Freq = np.zeros(N, np.complex\_)  
 Freq = DFT(signal, np.zeros\_like(signal))  
  
 fft\_data = fft(signal)  
 assert ((Freq-fft\_data)<10e-6).all()



import logging  
logging.basicConfig(filename='e5\_dft.log', level=logging.INFO)

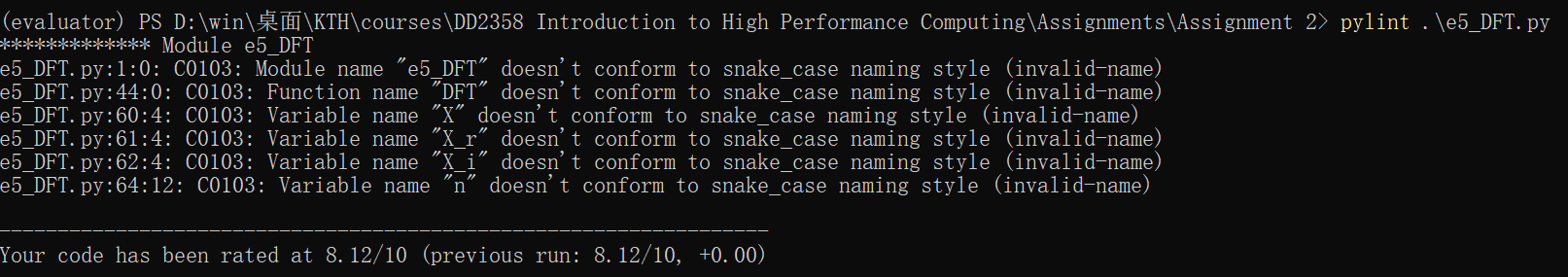
Freq = np.zeros(N, np.complex\_)  
Freq = DFT(signal, np.zeros\_like(signal), Freq)  
fft\_data = fft(signal)  
logging.info("Sucess!")  
  
if(((Freq-fft\_data)<10e-6).all()):  
 logging.info("Sucess!")  
else:  
 logging.error("Something went wrong")



**Task 5.2** Document the code using docstrings and generate automatic HTML documentation.



**Task 5.3** Check your code using a Python linter. Address the issues raised by the linter, including the style issues. To address some of the issues, you can use a Python auto-formatter.



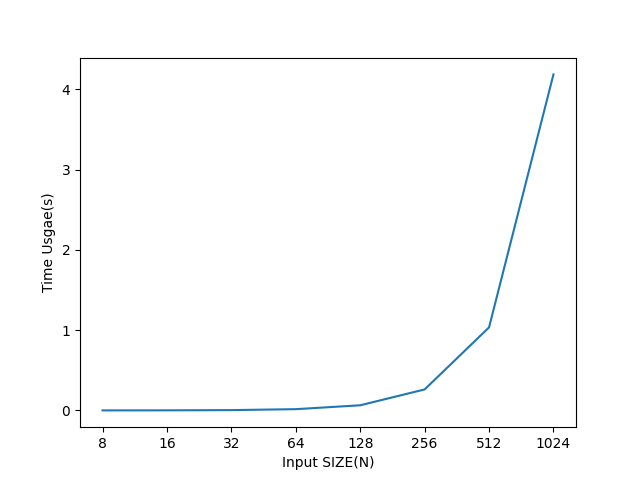
**Task 5.4** Measure the execution time, varying the input size from 8 to 1024 elements, and plot it.

**Code:**

times = []

Ns = [8,16,32,64,128,256,512,1024]  
for N in Ns:  
 print(f"N:{N}")  
 t = timer()  
 Freq = np.zeros(N, np.complex\_)  
 Freq = DFT(signal, np.zeros\_like(signal), Freq)  
 times.append(timer()-t)  
plt.plot(range(len(Ns)), times)  
plt.xticks(range(len(Ns)), labels=Ns)  
plt.xlabel("Input SIZE(N)")  
plt.ylabel("Time Usgae(s)")  
plt.show()

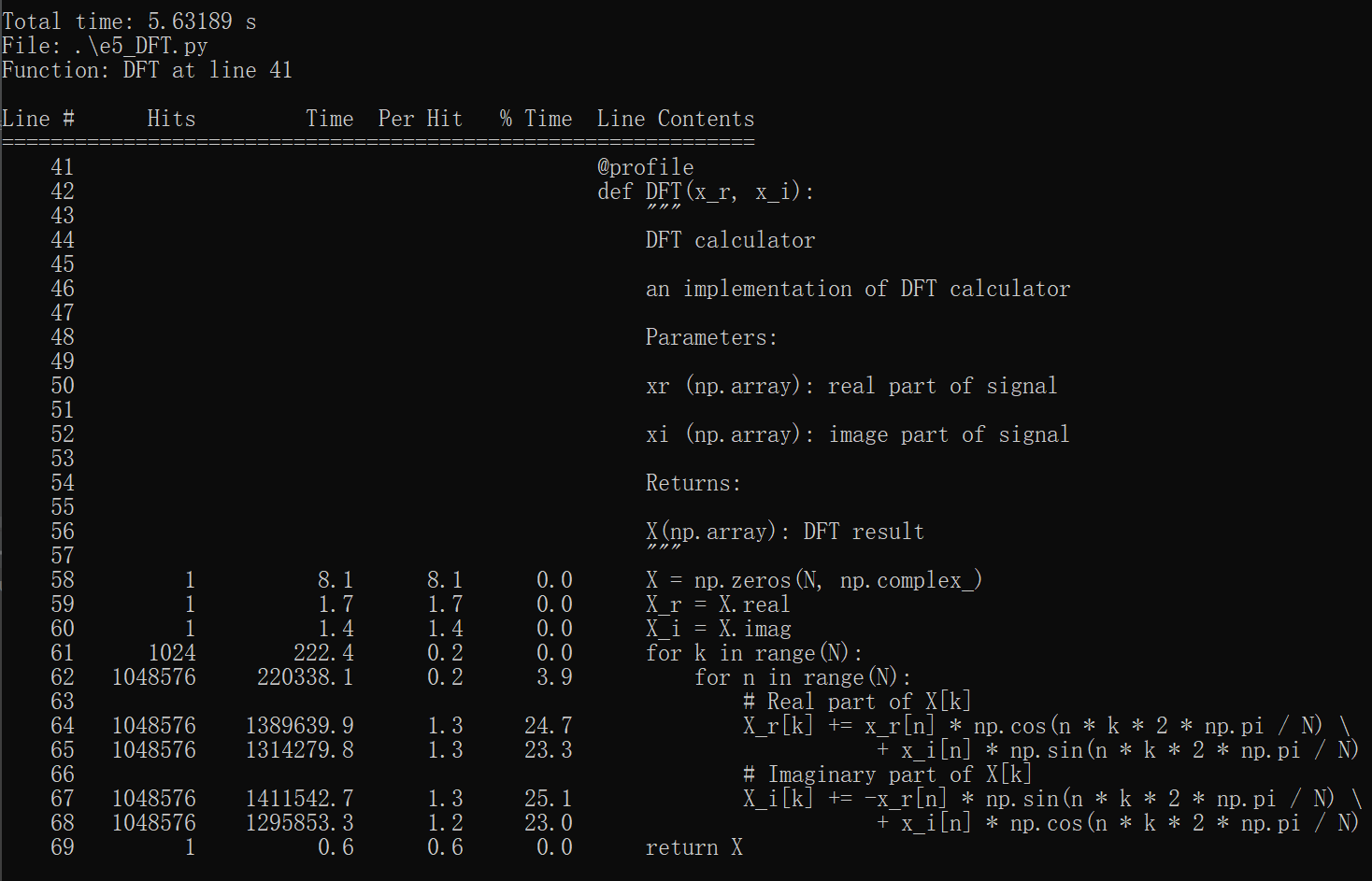
**Output:**



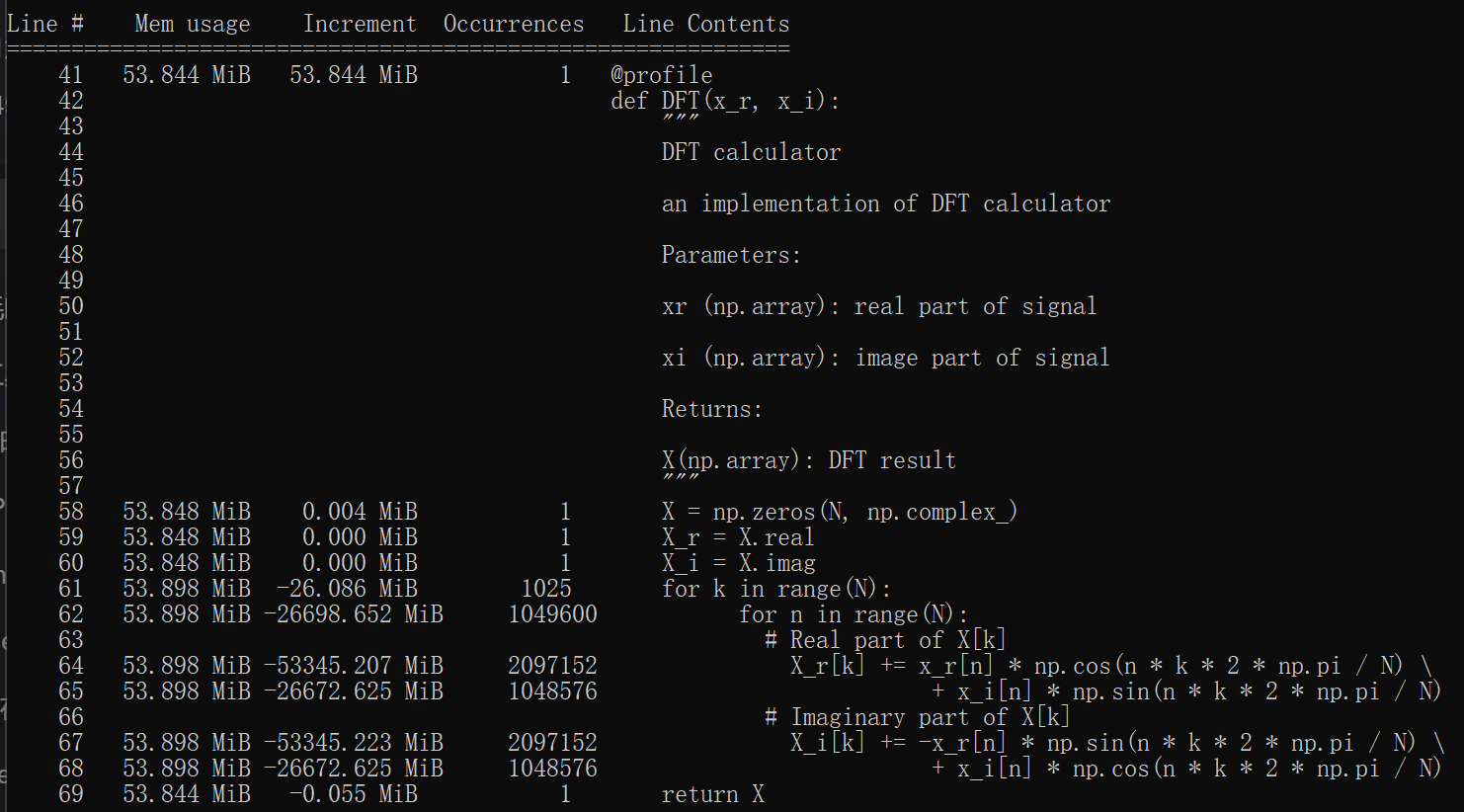
**Task 5.5** Profile the code with all the profiling tools that can be useful for performance analysis (from coarse-grained to fine-grained), fixing the input size, e.g., 1024. Motivate the choice of profiling tools and report the profiling results

Considering there are only DFT() funcions used, just line\_profiler instead of cProfile.

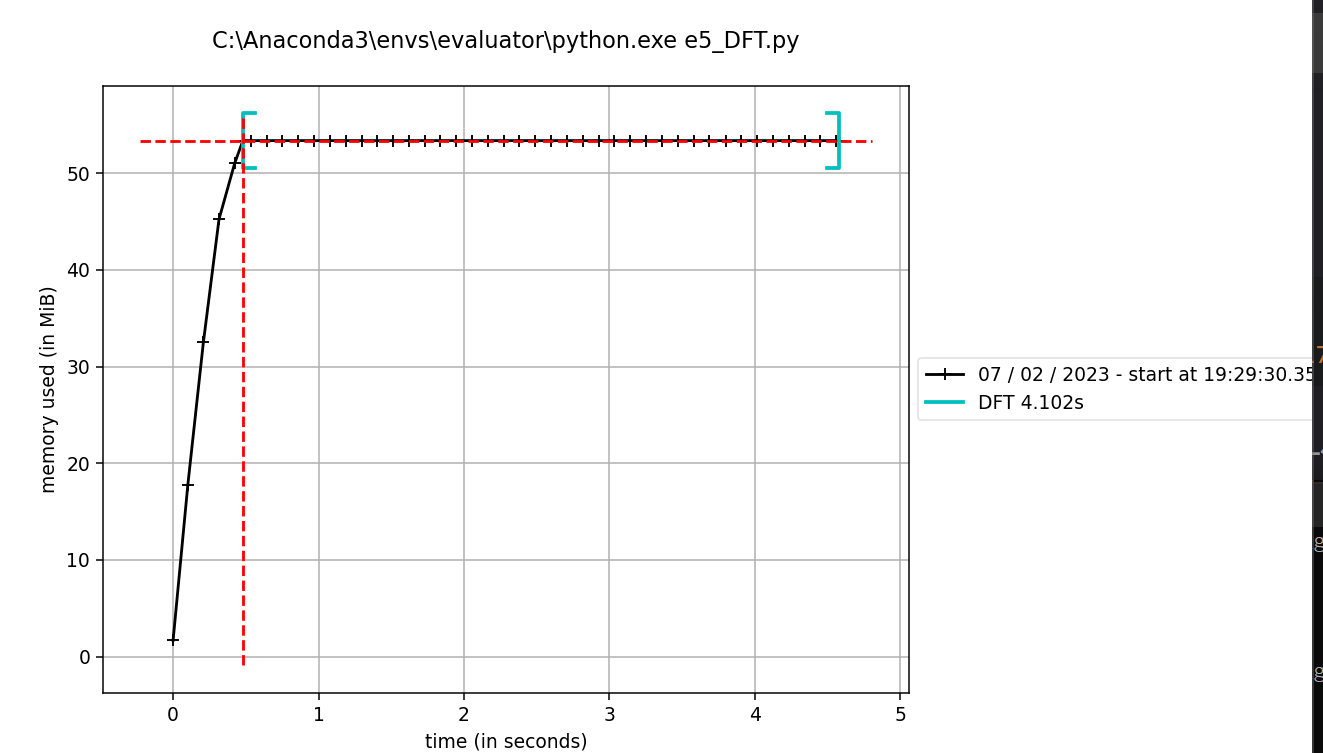
**line\_profiler:**



**Memory\_profiler**



**mprof**



Exercise 6 - Experiment with the Python Debugger

As part of this exercise, we ask you to complete an online tutorial on the Python pdb debugger. Follow the instructions at [https://github.com/spiside/pdb-tutorial.Links to an external site.](https://github.com/spiside/pdb-tutorial)

**Task 6.1 Reflection:**answer the questions: What are the advantages of using a debugger? What challenges did you find in using the pdb debugger, if any?

1 ) With a debugger, we can:

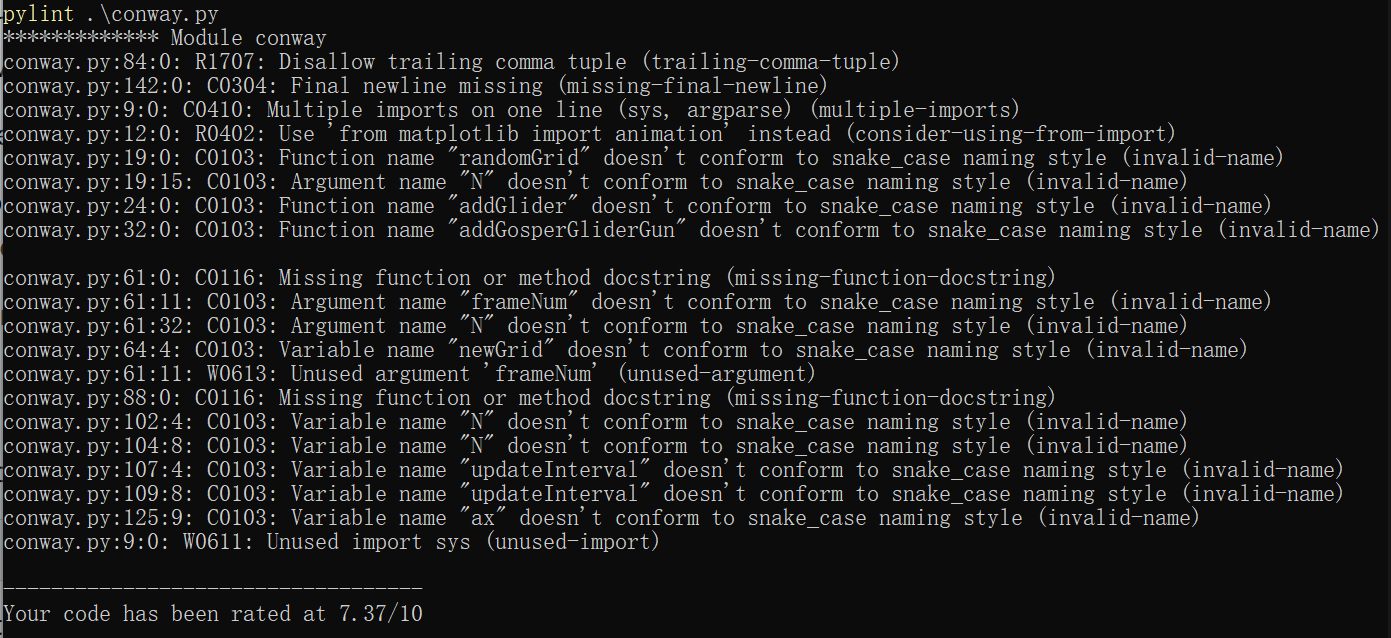
* + Explore the state of a running program
  + Test implementation code before applying it
  + Follow the program's execution logic

2 ) pdb debugger will make our code bloat. Every IDEA of Python , like Pycharm, VSCode etc. , have the interface for debug and its easier to use. Except that, IDEAs have many other features like autocomplete which greatly improve programming process.

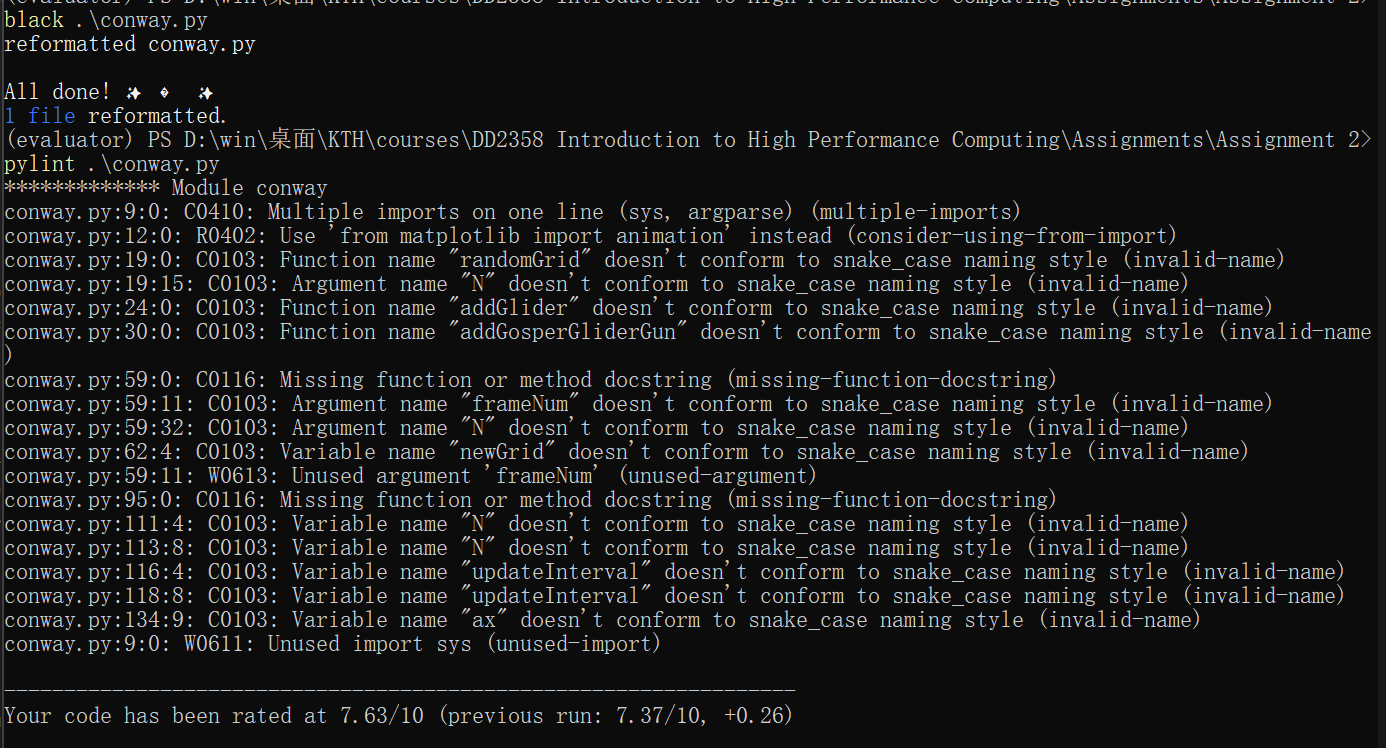
Bonus Exercise - Performance Analysis and Optimization of the Game of Life Code

**Task B.1**Check the code with a linter, and in case, run an auto-formatter. Produce HTML documentation running sphinx.

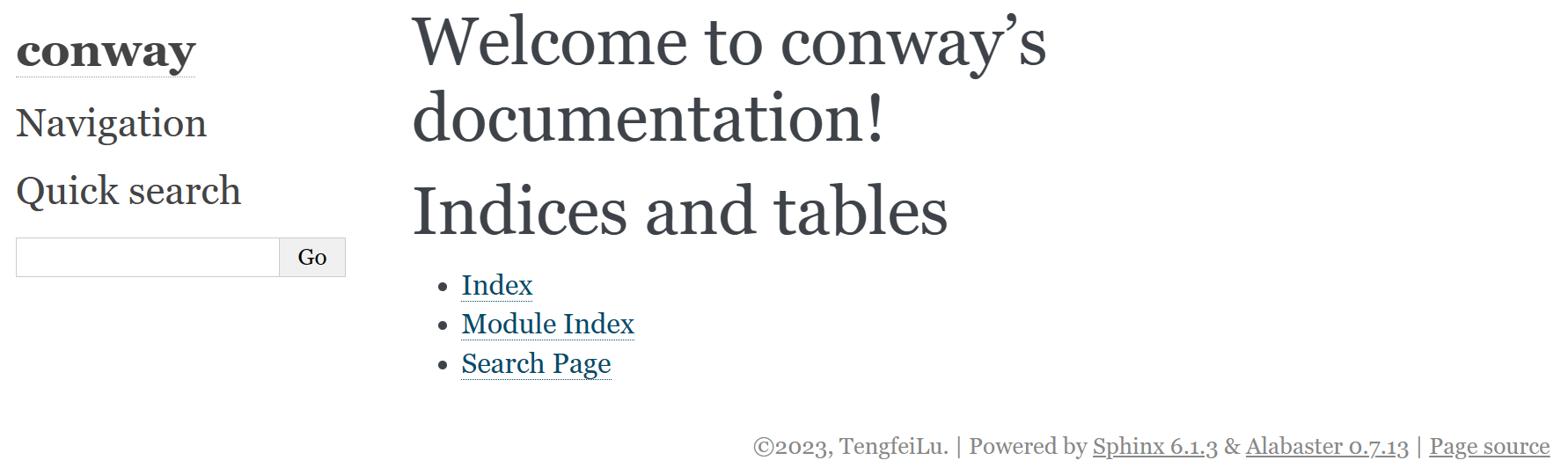
**Before** auto-formatter:



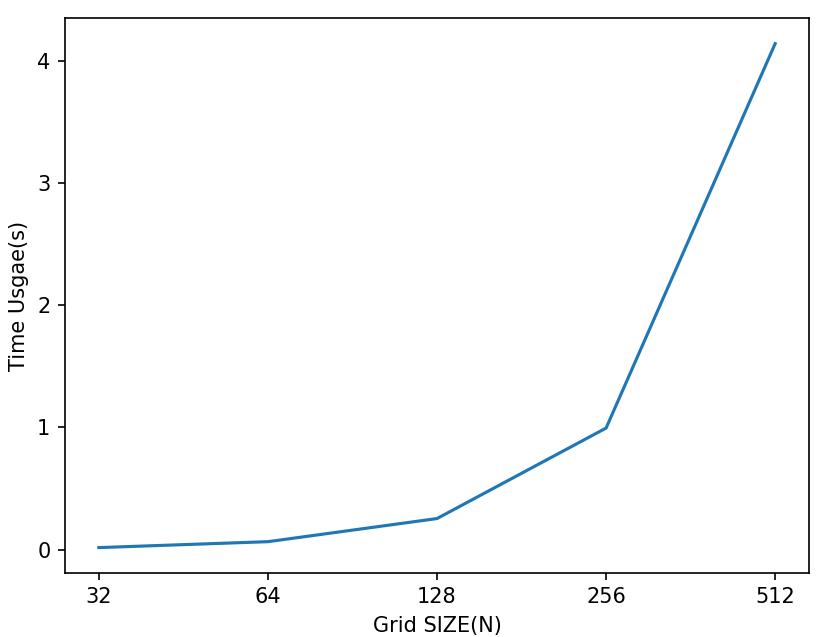
After auto-formatter:



Sphinx document

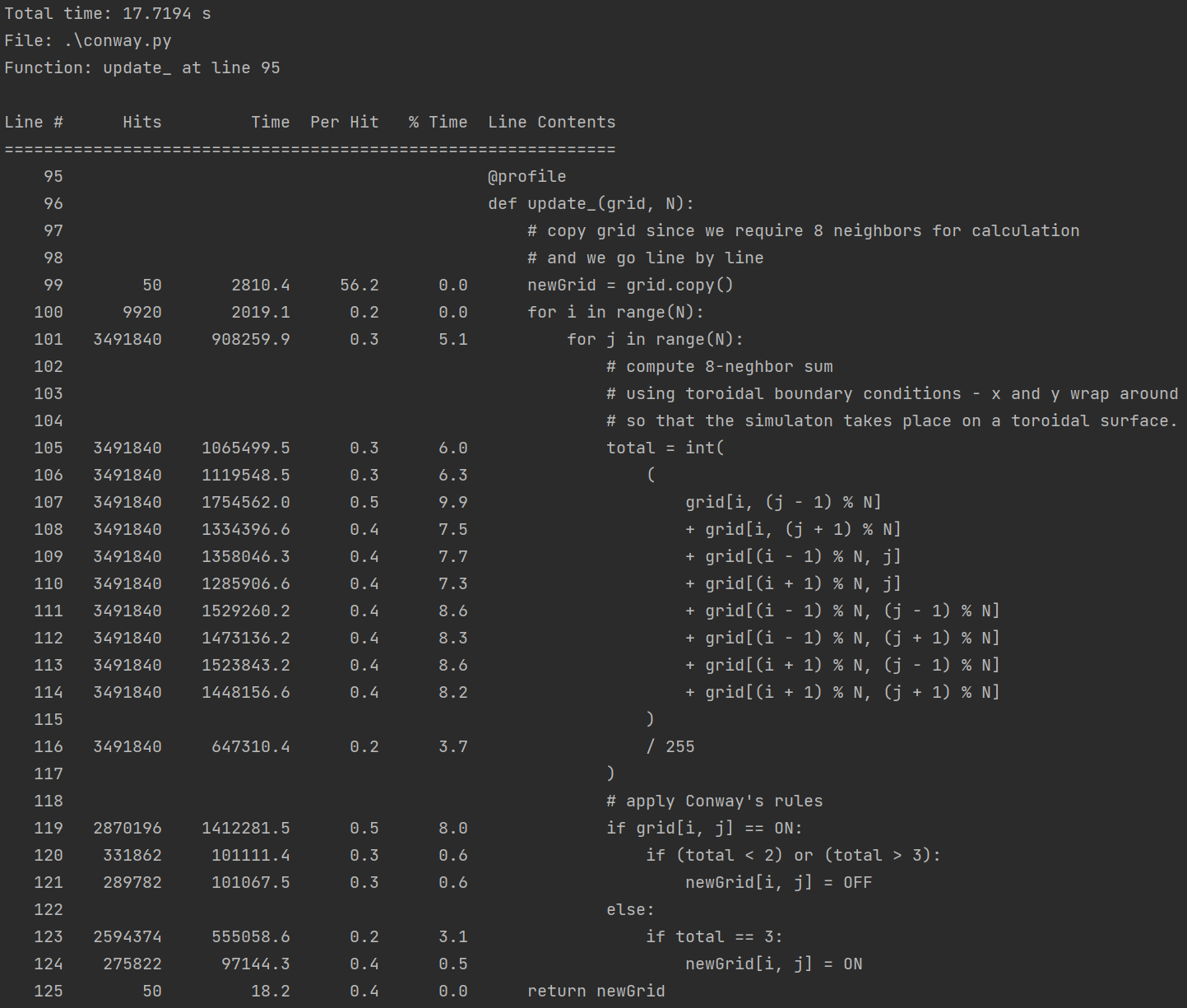


**Task B.2** Measure the execution time, varying the grid size (and fixed number of iterations). Make a plot with this information.

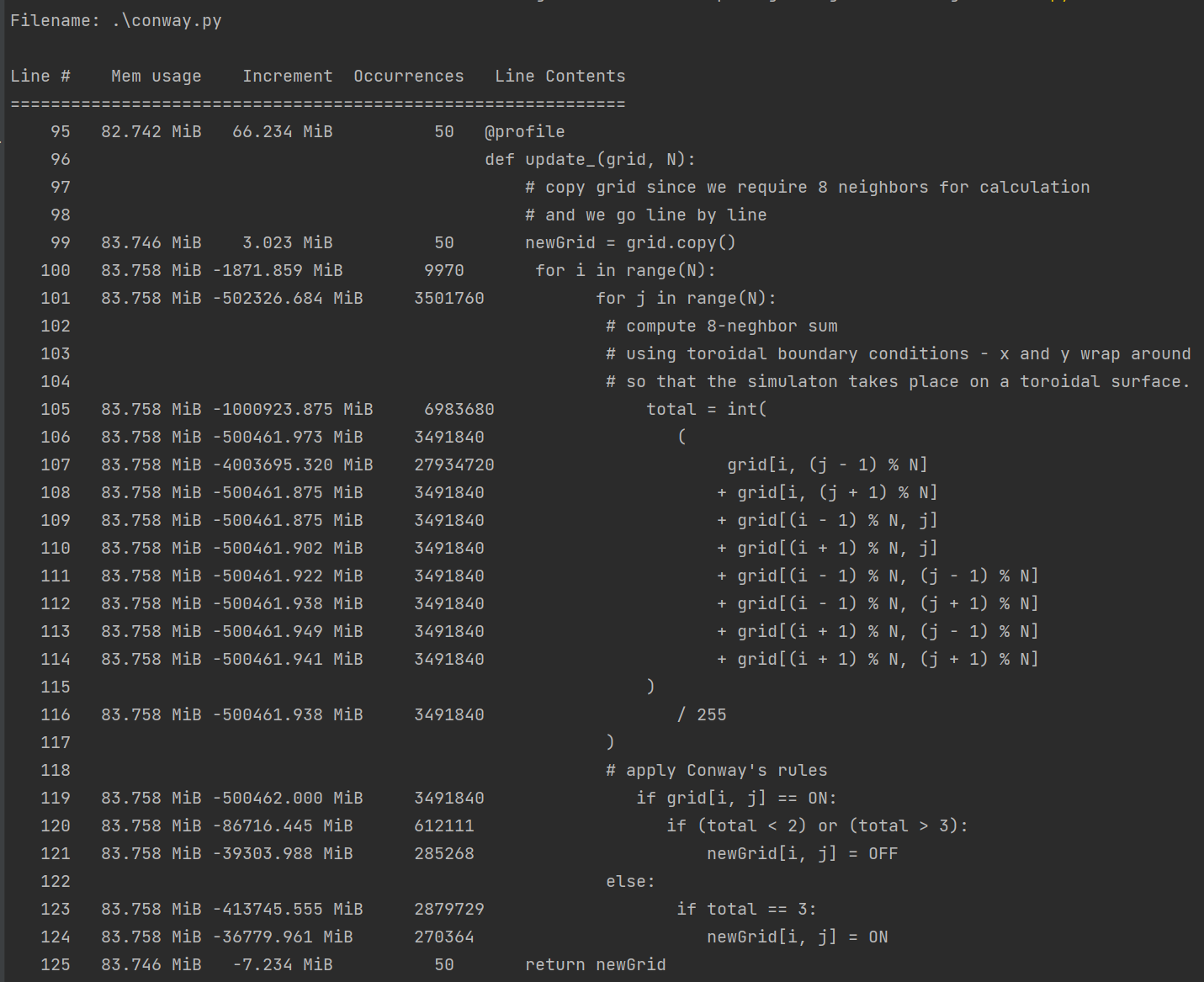


**Task B.3** Use different profilers (from coarse- to fine-grained) to identify performance bottlenecks and potential improvement. Report the results of the profilers. The choice of profilers is up to you.

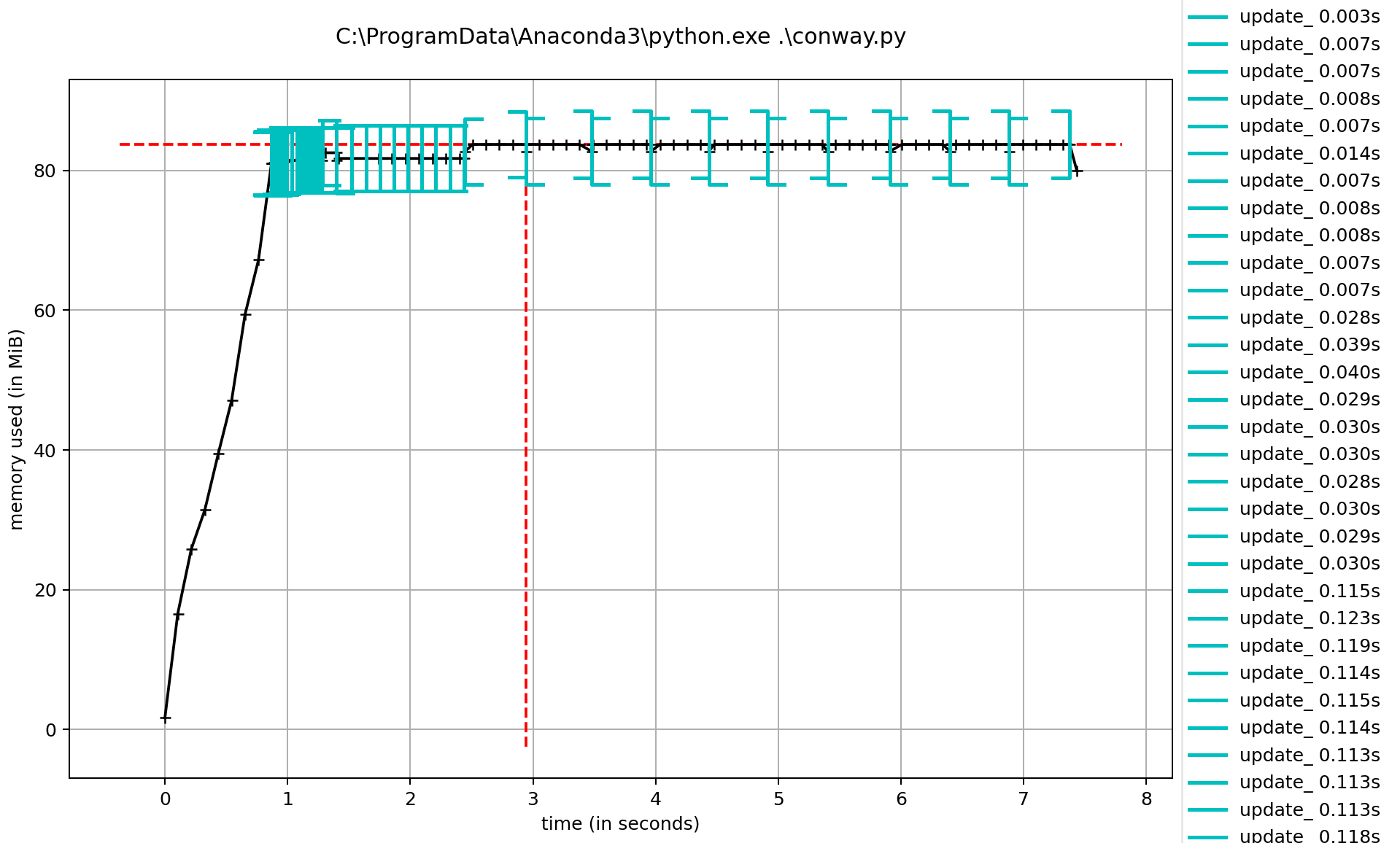
**line\_profiler:**



**Memory\_profiler**



**mprof**



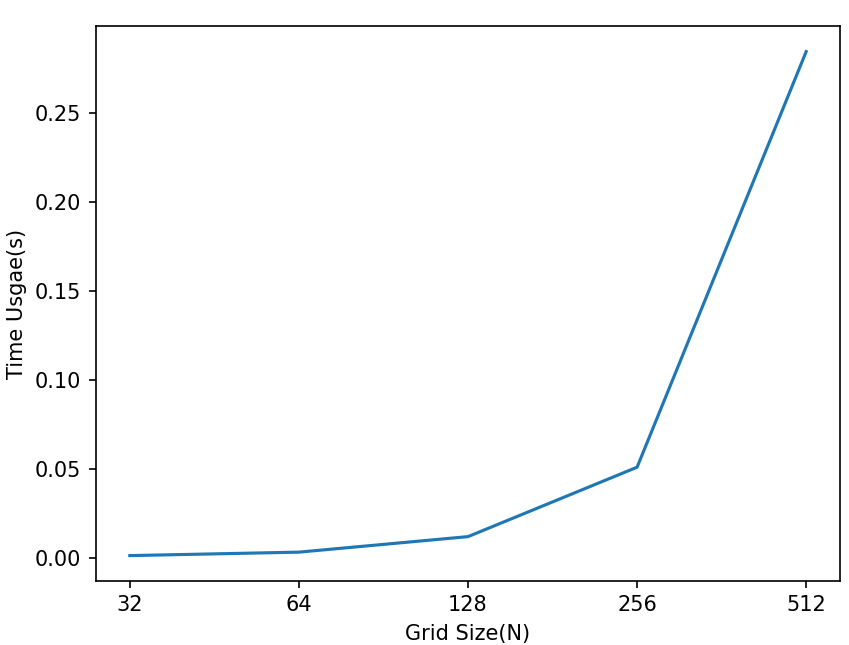
**Task B.4** Implement an optimization, report the new profiling results and show the performance improvement.

**Updated Code:**

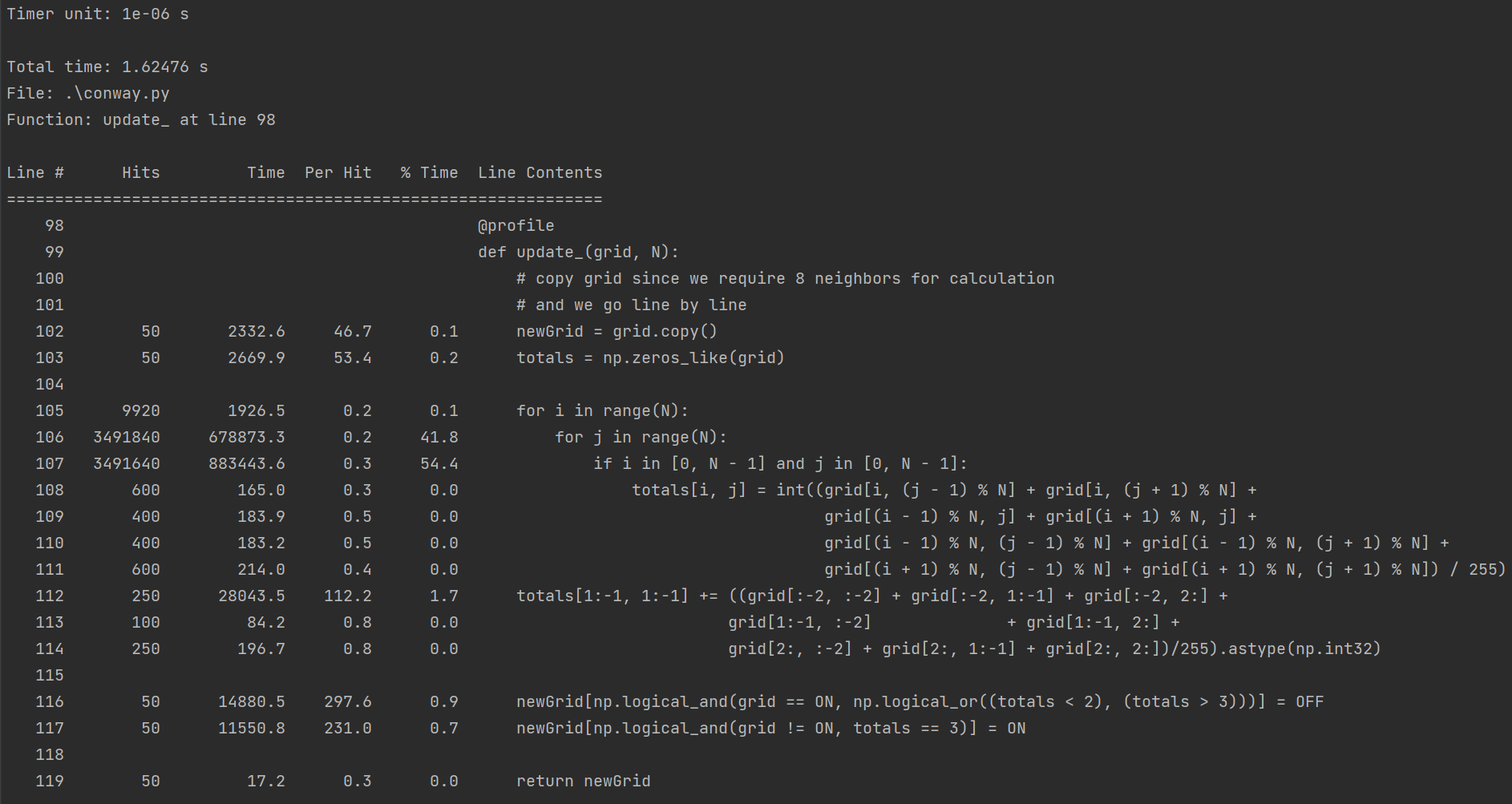
def update\_(grid, N):  
 # copy grid since we require 8 neighbors for calculation  
 # and we go line by line  
 newGrid = grid.copy()  
 totals = np.zeros\_like(grid)

for i in range(N):  
 for j in range(N):  
 if i in [0, N - 1] and j in [0, N - 1]:  
 totals[i, j] = int((grid[i, (j - 1) % N] + grid[i, (j + 1) % N] +  
 grid[(i - 1) % N, j] + grid[(i + 1) % N, j] +  
 grid[(i - 1) % N, (j - 1) % N] + grid[(i - 1) % N, (j + 1) % N] +  
 grid[(i + 1) % N, (j - 1) % N] + grid[(i + 1) % N, (j + 1) % N]) / 255)  
 totals[1:-1, 1:-1] += ((grid[:-2, :-2] + grid[:-2, 1:-1] + grid[:-2, 2:] +  
 grid[1:-1, :-2] + grid[1:-1, 2:] +  
 grid[2:, :-2] + grid[2:, 1:-1] + grid[2:, 2:])/255).astype(np.int32)  
  
 newGrid[np.logical\_and(grid == ON, np.logical\_or((totals < 2), (totals > 3)))] = OFF  
 newGrid[np.logical\_and(grid != ON, totals == 3)] = ON  
  
 return newGrid

**After update:**



line\_profiler (greatly optimized)



memory\_profiler(on optimization)

