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.

2. 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.

3. 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.

4. What is a page fault? What is the difference between a minor and a major page fault?

1) A **page-fault** is part of the modern memory allocation scheme.
2) 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.

5. 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 and we interrupt the flow of our execution pipeline

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

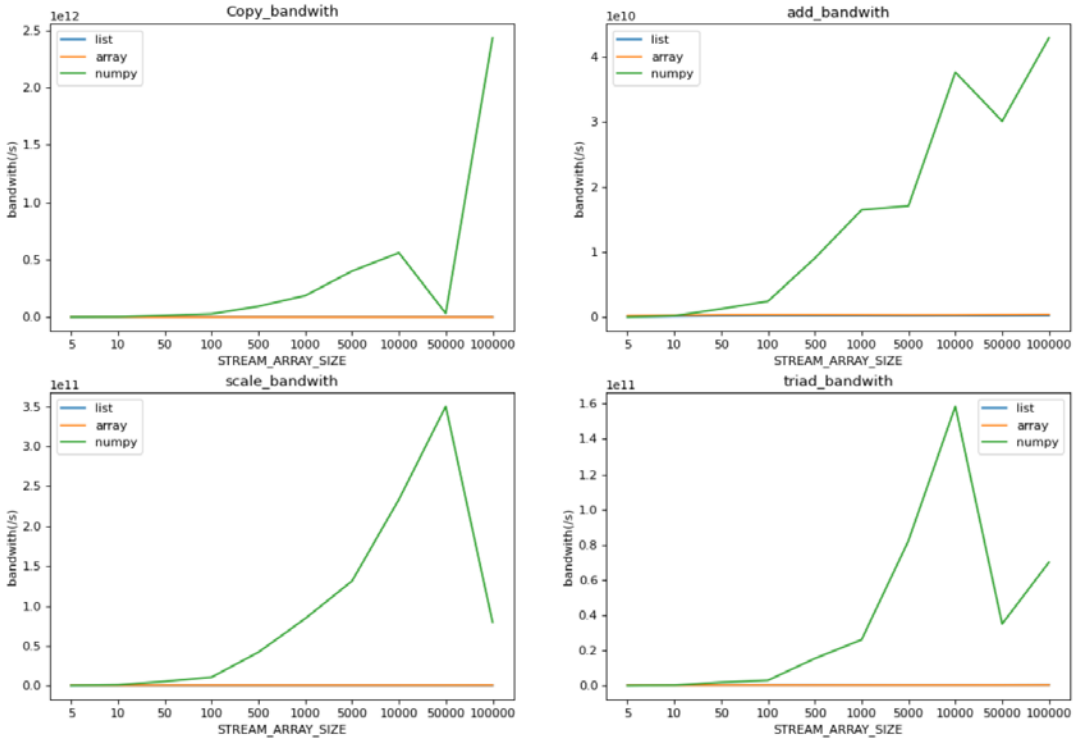
```
(base) PS C:\Users\daybeha> python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> np.show_config()
openblas64__info:
    library_dirs = ['D:\\a\\numpy\\numpy\\build\\openblas64__info']
    libraries = ['openblas64__info']
    language = f77
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'), ('HAVE_BLAS_ILP64', None)]
blas_ilp64_opt_info:
    library_dirs = ['D:\\a\\numpy\\numpy\\build\\openblas64__info']
    libraries = ['openblas64__info']
    language = f77
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'), ('HAVE_BLAS_ILP64', None)]
openblas64__lapack_info:
    library_dirs = ['D:\\a\\numpy\\numpy\\build\\openblas64__lapack_info']
    libraries = ['openblas64__lapack_info']
    language = f77
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'), ('HAVE_BLAS_ILP64', None), ('HAVE_LAPACKE', None)]
lapack_ilp64_opt_info:
    library_dirs = ['D:\\a\\numpy\\numpy\\build\\openblas64__lapack_info']
    libraries = ['openblas64__lapack_info']
    language = f77
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'), ('HAVE_BLAS_ILP64', None), ('HAVE_LAPACKE', None)]
Supported SIMD extensions in this NumPy install:
    baseline = SSE,SSE2,SSE3
    found = SSSE3,SSE41,POPCNT,SSE42,AVX,F16C,FMA3,AVX2
    not found = AVX512F,AVX512CD,AVX512_SKX,AVX512_CLX,AVX512_CNL,AVX512_ICL
```

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:

```
============================= test session starts ==============================
collecting ... collected 5 items

JuliaSet.py::test_calc_pure_python[1000-300-33219980]
JuliaSet.py::test_calc_pure_python[100-300-334236]
JuliaSet.py::test_calc_pure_python[100-100-131532]
JuliaSet.py::test_calc_pure_python[180-300-1076586]
JuliaSet.py::test_calc_pure_python[180-200-1076586] PASSED             [ 20%]calculate_z_serial_purepython took 5.472217798233032 seconds
PASSED                [ 40%]calculate_z_serial_purepython took 0.05482816696166992 seconds
PASSED                [ 60%]calculate_z_serial_purepython took 0.023000240325927734 seconds
PASSED               [ 80%]calculate_z_serial_purepython took 0.18157362937927246 seconds
FAILED               [100%]calculate_z_serial_purepython took 0.12715601921081543 seconds

JuliaSet.py:70 (test_calc_pure_python[180-200-1076586])
751254 != 1076586

Expected :1076586
Actual   :751254
<Click to see difference>

num1 = 180, num2 = 200, expected = 1076586

    @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
E       assert 751254 == 1076586
E        +  where 751254 = sum([0, 0, 0, 0, 0, 0, ...])
E        +    where [0, 0, 0, 0, 0, 0, ...] = calc_pure_python(desired_width=180, max_iterations=200)

JuliaSet.py:74: AssertionError



======================== 1 failed, 4 passed in 6.28s =========================
```

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:

```
A:
[[0.45427996 0.93410411 0.43667526 0.51488453 0.4109359 ]
 [0.7041321  0.78952869 0.06591291 0.93581901 0.70911714]
 [0.60446704 0.94156487 0.92020224 0.0183316  0.60946608]
 [0.17245395 0.7982487  0.33645573 0.65606593 0.68943622]
 [0.07190902 0.57348201 0.18412812 0.77623531 0.06461922]]
B[[0.43728348 0.09022857 0.37619573 0.01525552 0.38051541]
 [0.99077245 0.0313196  0.7873943  0.957936   0.52984774]
 [0.72182833 0.19059772 0.62299358 0.08354924 0.93917315]
 [0.27000756 0.8186325  0.50687042 0.84327751 0.00387851]
 [0.72117571 0.48371393 0.11740592 0.10319876 0.73022721]]

Result:
[[1.87471803 0.77375075 1.48767837 1.41482474 1.37998072]
 [1.90180268 1.2099251  1.48521873 1.63490197 1.26961295]
 [2.30590983 0.56923216 1.62290749 1.06641731 2.03824409]
 [1.78350479 0.9752556  1.31650699 1.41980727 1.31055149]
 [0.98873492 0.72625254 0.99435637 1.22709024 0.55434617]]
```

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

Code:

```python
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:

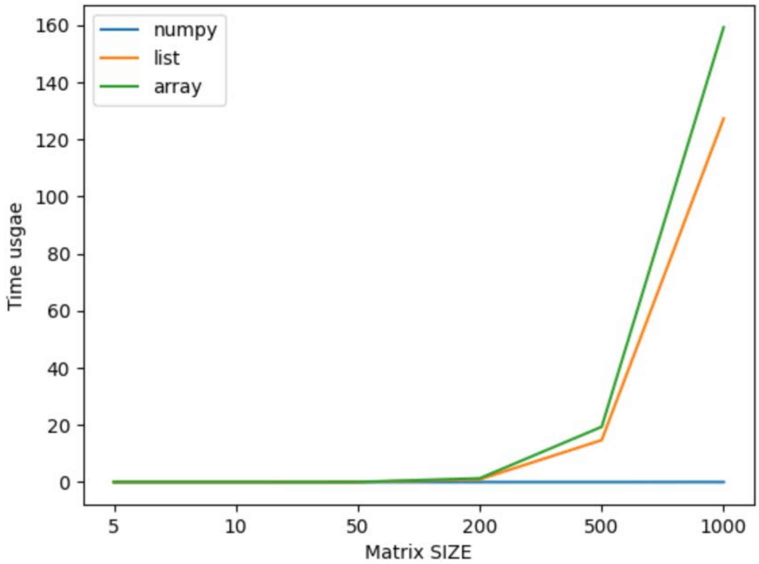
```
============================= test session starts ==============================
collecting ... collected 1 item

e_4_dgemm.py::test_dgemm PASSED                                          [100%]

============================== 1 passed in 0.02s ===============================
```

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]

```
Numpy time   average:22.636333333328853 ms   min:92.00000000042508 us    max:0.010604299999997124 s
 list time   average:235937.7938333333 ms    min:220.99999999913854 us   max:126.4852885 s
 array time  average:302671.2825 ms  min:261.9999999997624 us    max:161.37949329999998 s
```



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

$$N^2 * 2N = 2N^3$$

------------FLOPS/s-------------

	numpy	list	array
5	347801.8920422833	11210762.331823166	9363295.880133634
10	114942528.73526055	14760147.601480905	12070006.035007726
50	6925207756.235878	16134236.850596929	11936535.826318618
200	19316672703.126793	16668699.900789984	12035215.944494786
500	110894251242.01611	17001932.650489017	12903995.250875525
1000	192294749391.77063	15701986.705879984	12553818.25022325

The clock frequency of my CPU is 3.2GHz= $3.2 * 2^{30} = 3{,}435{,}973{,}836.8$

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()
```

```
============================== test session starts ==============================
collecting ... collected 1 item

e5_DFT.py::test_DFT PASSED                                                  [100%]

=============================== 1 passed in 4.71s ===============================
```

```python
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")
```

```
DFT.py ×   e5_dft.log ×
supporting *.log files found.
INFO:root:Sucess!
```

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

Assginment2

Navigation

Quick search

Go

Hide Search Matches

Welcome to Assginment2's documentation!

Indices and tables

- Index
- Module Index
- Search Page

©2023, TengfeiLu. | Powered by Sphinx 6.1.3 & Alabaster 0.7.13 | Page source

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.

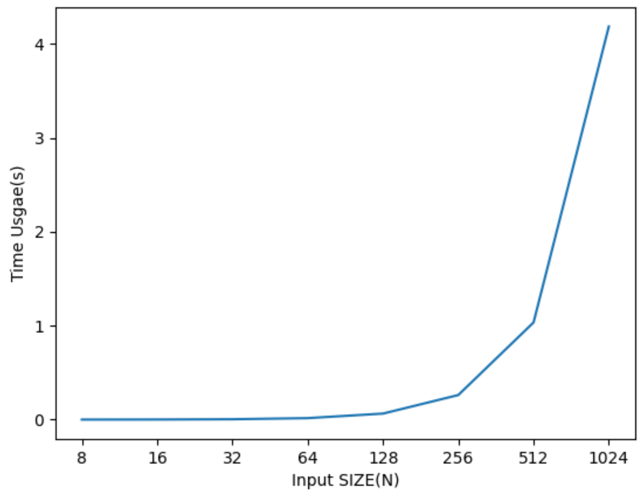
```
(evaluator) PS D:\win\桌面\KTH\courses\DD2358 Introduction to High Performance Computing\Assignments\Assignment 2> pylint .\e5_DFT.py
************* Module e5_DFT
e5_DFT.py:1:0: C0103: Module name "e5_DFT" doesn't conform to snake_case naming style (invalid-name)
e5_DFT.py:44:0: C0103: Function name "DFT" doesn't conform to snake_case naming style (invalid-name)
e5_DFT.py:60:4: C0103: Variable name "X" doesn't conform to snake_case naming style (invalid-name)
e5_DFT.py:61:4: C0103: Variable name "X_r" doesn't conform to snake_case naming style (invalid-name)
e5_DFT.py:62:4: C0103: Variable name "X_i" doesn't conform to snake_case naming style (invalid-name)
e5_DFT.py:64:12: C0103: Variable name "n" doesn't conform to snake_case naming style (invalid-name)

------------------------------------------------------------------
Your code has been rated at 8.12/10 (previous run: 8.12/10, +0.00)
```

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. <u>Motivate the choice of profiling tools and report the profiling results</u>

Considering there are only DFT() funcions used, just line_profiler instead of cProfile.

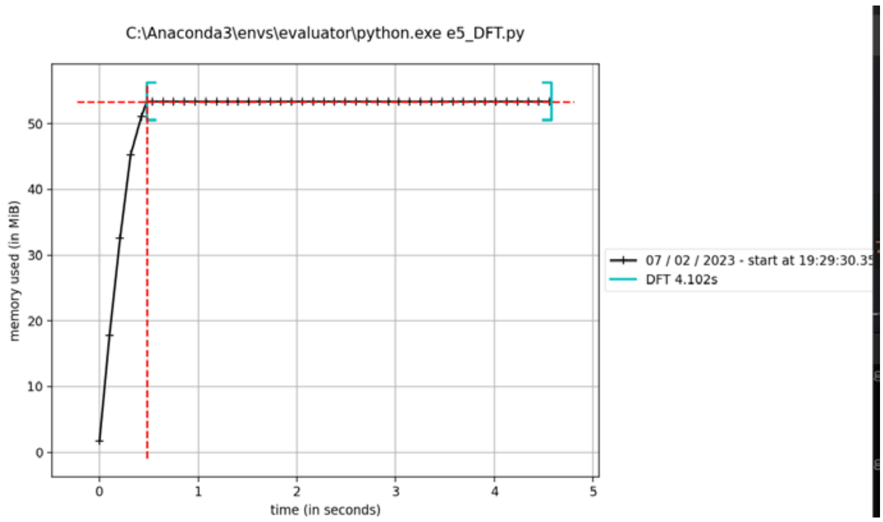
line_profiler:

```
Total time: 5.63189 s
File: .\e5_DFT.py
Function: DFT at line 41

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    41                                           @profile
    42                                           def DFT(x_r, x_i):
    43                                               """
    44                                               DFT calculator
    45
    46                                               an implementation of DFT calculator
    47
    48                                               Parameters:
    49
    50                                               xr (np.array): real part of signal
    51
    52                                               xi (np.array): image part of signal
    53
    54                                               Returns:
    55
    56                                               X(np.array): DFT result
    57                                               """
    58         1          8.1      8.1      0.0      X = np.zeros(N, np.complex_)
    59         1          1.7      1.7      0.0      X_r = X.real
    60         1          1.4      1.4      0.0      X_i = X.imag
    61      1024        222.4      0.2      0.0      for k in range(N):
    62   1048576     220338.1      0.2      3.9          for n in range(N):
    63                                                       # Real part of X[k]
    64   1048576    1389639.9      1.3     24.7              X_r[k] += x_r[n] * np.cos(n * k * 2 * np.pi / N) \
    65   1048576    1314279.8      1.3     23.3                  + x_i[n] * np.sin(n * k * 2 * np.pi / N)
    66                                                       # Imaginary part of X[k]
    67   1048576    1411542.7      1.3     25.1              X_i[k] += -x_r[n] * np.sin(n * k * 2 * np.pi / N) \
    68   1048576    1295853.3      1.2     23.0                  + x_i[n] * np.cos(n * k * 2 * np.pi / N)
    69         1          0.6      0.6      0.0      return X
```

Memory_profiler

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    41   53.844 MiB   53.844 MiB           1   @profile
    42                                         def DFT(x_r, x_i):
    43                                             """
    44                                             DFT calculator
    45
    46                                             an implementation of DFT calculator
    47
    48                                             Parameters:
    49
    50                                             xr (np.array): real part of signal
    51
    52                                             xi (np.array): image part of signal
    53
    54                                             Returns:
    55
    56                                             X(np.array): DFT result
    57                                             """
    58   53.848 MiB    0.004 MiB           1       X = np.zeros(N, np.complex_)
    59   53.848 MiB    0.000 MiB           1       X_r = X.real
    60   53.848 MiB    0.000 MiB           1       X_i = X.imag
    61   53.898 MiB  -26.086 MiB        1025       for k in range(N):
    62   53.898 MiB -26698.652 MiB     1049600           for n in range(N):
    63                                                     # Real part of X[k]
    64   53.898 MiB -53345.207 MiB     2097152               X_r[k] += x_r[n] * np.cos(n * k * 2 * np.pi / N) \
    65   53.898 MiB -26672.625 MiB     1048576                   + x_i[n] * np.sin(n * k * 2 * np.pi / N)
    66                                                     # Imaginary part of X[k]
    67   53.898 MiB -53345.223 MiB     2097152               X_i[k] += -x_r[n] * np.sin(n * k * 2 * np.pi / N) \
    68   53.898 MiB -26672.625 MiB     1048576                   + x_i[n] * np.cos(n * k * 2 * np.pi / N)
    69   53.844 MiB   -0.055 MiB           1       return X
```

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.

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:

```
pylint .\conway.py
************* Module conway
conway.py:84:0: R1707: Disallow trailing comma tuple (trailing-comma-tuple)
conway.py:142:0: C0304: Final newline missing (missing-final-newline)
conway.py:9:0: C0410: Multiple imports on one line (sys, argparse) (multiple-imports)
conway.py:12:0: R0402: Use 'from matplotlib import animation' instead (consider-using-from-import)
conway.py:19:0: C0103: Function name "randomGrid" doesn't conform to snake_case naming style (invalid-name)
conway.py:19:15: C0103: Argument name "N" doesn't conform to snake_case naming style (invalid-name)
conway.py:24:0: C0103: Function name "addGlider" doesn't conform to snake_case naming style (invalid-name)
conway.py:32:0: C0103: Function name "addGosperGliderGun" doesn't conform to snake_case naming style (invalid-name)

conway.py:61:0: C0116: Missing function or method docstring (missing-function-docstring)
conway.py:61:11: C0103: Argument name "frameNum" doesn't conform to snake_case naming style (invalid-name)
conway.py:61:32: C0103: Argument name "N" doesn't conform to snake_case naming style (invalid-name)
conway.py:64:4: C0103: Variable name "newGrid" doesn't conform to snake_case naming style (invalid-name)
conway.py:61:11: W0613: Unused argument 'frameNum' (unused-argument)
conway.py:88:0: C0116: Missing function or method docstring (missing-function-docstring)
conway.py:102:4: C0103: Variable name "N" doesn't conform to snake_case naming style (invalid-name)
conway.py:104:8: C0103: Variable name "N" doesn't conform to snake_case naming style (invalid-name)
conway.py:107:4: C0103: Variable name "updateInterval" doesn't conform to snake_case naming style (invalid-name)
conway.py:109:8: C0103: Variable name "updateInterval" doesn't conform to snake_case naming style (invalid-name)
conway.py:125:9: C0103: Variable name "ax" doesn't conform to snake_case naming style (invalid-name)
conway.py:9:0: W0611: Unused import sys (unused-import)

------------------------------------------------------------------
Your code has been rated at 7.37/10
```

After auto-formatter:

```
black .\conway.py
reformatted conway.py

All done! ✨ 🍰 ✨
1 file reformatted.
(evaluator) PS D:\win\桌面\KTH\courses\DD2358 Introduction to High Performance Computing\Assignments\Assignment 2>
pylint .\conway.py
************* Module conway
conway.py:9:0: C0410: Multiple imports on one line (sys, argparse) (multiple-imports)
conway.py:12:0: R0402: Use 'from matplotlib import animation' instead (consider-using-from-import)
conway.py:19:0: C0103: Function name "randomGrid" doesn't conform to snake_case naming style (invalid-name)
conway.py:19:15: C0103: Argument name "N" doesn't conform to snake_case naming style (invalid-name)
conway.py:24:0: C0103: Function name "addGlider" doesn't conform to snake_case naming style (invalid-name)
conway.py:30:0: C0103: Function name "addGosperGliderGun" doesn't conform to snake_case naming style (invalid-name
)
conway.py:59:0: C0116: Missing function or method docstring (missing-function-docstring)
conway.py:59:11: C0103: Argument name "frameNum" doesn't conform to snake_case naming style (invalid-name)
conway.py:59:32: C0103: Argument name "N" doesn't conform to snake_case naming style (invalid-name)
conway.py:62:4: C0103: Variable name "newGrid" doesn't conform to snake_case naming style (invalid-name)
conway.py:59:11: W0613: Unused argument 'frameNum' (unused-argument)
conway.py:95:0: C0116: Missing function or method docstring (missing-function-docstring)
conway.py:111:4: C0103: Variable name "N" doesn't conform to snake_case naming style (invalid-name)
conway.py:113:8: C0103: Variable name "N" doesn't conform to snake_case naming style (invalid-name)
conway.py:116:4: C0103: Variable name "updateInterval" doesn't conform to snake_case naming style (invalid-name)
conway.py:118:8: C0103: Variable name "updateInterval" doesn't conform to snake_case naming style (invalid-name)
conway.py:134:9: C0103: Variable name "ax" doesn't conform to snake_case naming style (invalid-name)
conway.py:9:0: W0611: Unused import sys (unused-import)

------------------------------------------------------------------
Your code has been rated at 7.63/10 (previous run: 7.37/10, +0.26)
```

It seems the auto-formatter can just deal with parts of format problem. Especially, can not solve problems of 'invalid-name'.

Sphinx document

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

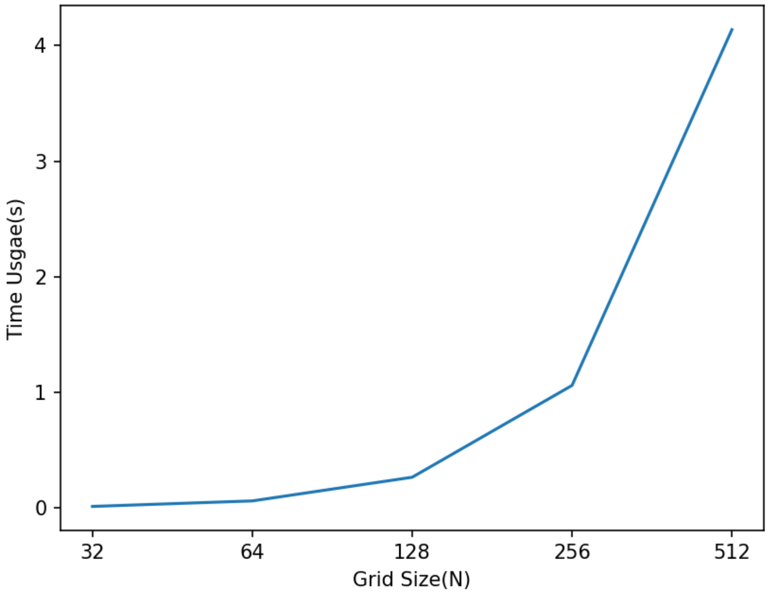
To evaluate the execution time, slightly modify codes first:
(remove params of "frameNum" and "img" which used to animation visualization)

```python
def update_(grid, N):
    # copy grid since we require 8 neighbors for calculation
    # and we go line by line
    newGrid = grid.copy()
    for i in range(N):
        for j in range(N):
            # compute 8-neghbor sum
            # using toroidal boundary conditions - x and y wrap around
            # so that the simulaton takes place on a toroidal surface.
            total = 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
            )
            # apply Conway's rules
            if grid[i, j] == ON:
                if (total < 2) or (total > 3):
                    newGrid[i, j] = OFF
            else:
                if total == 3:
                    newGrid[i, j] = ON
    return newGrid
```

and *for* loop instand of *animation.FuncAnimation*

```python
for i in range(10):
    grid = update_(grid, N)
```

The results as follows:



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:

```
Total time: 17.7194 s
File: .\conway.py
Function: update_ at line 95

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    95                                           @profile
    96                                           def update_(grid, N):
    97                                               # copy grid since we require 8 neighbors for calculation
    98                                               # and we go line by line
    99        50       2810.4     56.2      0.0      newGrid = grid.copy()
   100      9920       2019.1      0.2      0.0      for i in range(N):
   101   3491840     908259.9      0.3      5.1          for j in range(N):
   102                                                       # compute 8-neghbor sum
   103                                                       # using toroidal boundary conditions - x and y wrap around
   104                                                       # so that the simulaton takes place on a toroidal surface.
   105   3491840    1065499.5      0.3      6.0              total = int(
   106   3491840    1119548.5      0.3      6.3                  (
   107   3491840    1754562.0      0.5      9.9                      grid[i, (j - 1) % N]
   108   3491840    1334396.6      0.4      7.5                      + grid[i, (j + 1) % N]
   109   3491840    1358046.3      0.4      7.7                      + grid[(i - 1) % N, j]
   110   3491840    1285906.6      0.4      7.3                      + grid[(i + 1) % N, j]
   111   3491840    1529260.2      0.4      8.6                      + grid[(i - 1) % N, (j - 1) % N]
   112   3491840    1473136.2      0.4      8.3                      + grid[(i - 1) % N, (j + 1) % N]
   113   3491840    1523843.2      0.4      8.6                      + grid[(i + 1) % N, (j - 1) % N]
   114   3491840    1448156.6      0.4      8.2                      + grid[(i + 1) % N, (j + 1) % N]
   115                                                           )
   116   3491840     647310.4      0.2      3.7                  / 255
   117                                                       )
   118                                                       # apply Conway's rules
   119   2870196    1412281.5      0.5      8.0              if grid[i, j] == ON:
   120    331862     101111.4      0.3      0.6                  if (total < 2) or (total > 3):
   121    289782     101067.5      0.3      0.6                      newGrid[i, j] = OFF
   122                                                       else:
   123   2594374     555058.6      0.2      3.1                  if total == 3:
   124    275822      97144.3      0.4      0.5                      newGrid[i, j] = ON
   125        50         18.2      0.4      0.0      return newGrid
```

As we can see, there are two time expensive operation: ‘*8-neghbor sum*’ and assignment of *newGrid*

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

For the '*8-neghbor sum*', the first shoot to optimize is compute it as matrixes. But the computation of first and last row/col is different, we need to take it into consideration:

```
# consider first and last row/col
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)

# compute as matrix
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)
```

For assignment of *newGrid*, its also better to assignment as matrix:

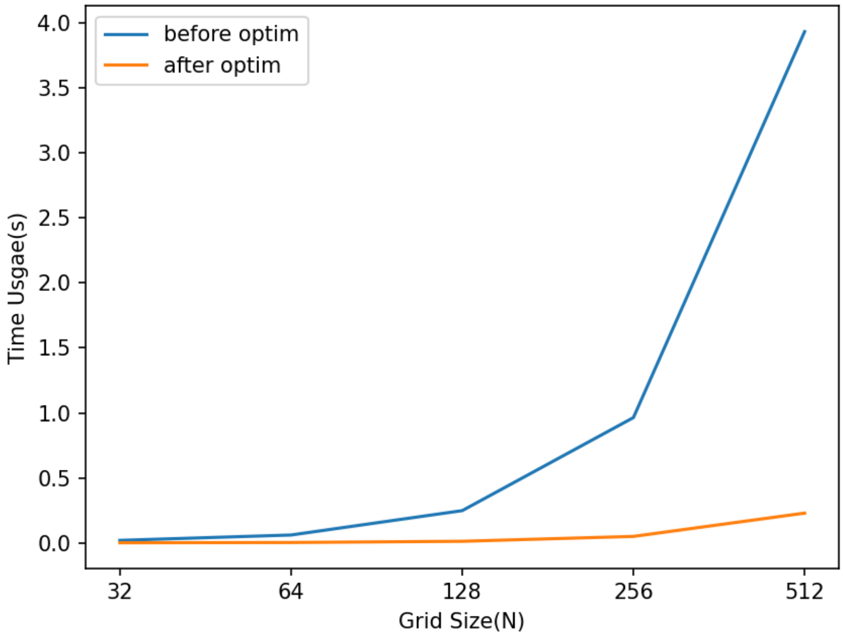
```
newGrid[np.logical_and(grid == ON, np.logical_or((totals < 2),
(totals > 3)))] = OFF
newGrid[np.logical_and(grid != ON, totals == 3)] = ON
```

so the **Updated Code** as follows**:**

```
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
```

The time usage after update as follows: (conway.py)



line_profiler (greatly optimized)

```
Timer unit: 1e-06 s

Total time: 1.62476 s
File: .\conway.py
Function: update_ at line 98

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    98                                           @profile
    99                                           def update_(grid, N):
   100                                               # copy grid since we require 8 neighbors for calculation
   101                                               # and we go line by line
   102        50       2332.6     46.7      0.1      newGrid = grid.copy()
   103        50       2669.9     53.4      0.2      totals = np.zeros_like(grid)
   104
   105      9920       1926.5      0.2      0.1      for i in range(N):
   106   3491840     678873.3      0.2     41.8          for j in range(N):
   107   3491640     883443.6      0.3     54.4              if i in [0, N - 1] and j in [0, N - 1]:
   108       600        165.0      0.3      0.0                  totals[i, j] = int((grid[i, (j - 1) % N] + grid[i, (j + 1) % N] +
   109       400        183.9      0.5      0.0                                      grid[(i - 1) % N, j] + grid[(i + 1) % N, j] +
   110       400        183.2      0.5      0.0                                      grid[(i - 1) % N, (j - 1) % N] + grid[(i - 1) % N, (j + 1) % N] +
   111       600        214.0      0.4      0.0                                      grid[(i + 1) % N, (j - 1) % N] + grid[(i + 1) % N, (j + 1) % N]) / 255)
   112       250      28043.5    112.2      1.7      totals[1:-1, 1:-1] += ((grid[:-2, :-2] + grid[:-2, 1:-1] + grid[:-2, 2:] +
   113       100         84.2      0.8      0.0                              grid[1:-1, :-2]              + grid[1:-1, 2:] +
   114       250        196.7      0.8      0.0                              grid[2:, :-2] + grid[2:, 1:-1] + grid[2:, 2:])/255).astype(np.int32)
   115
   116        50      14880.5    297.6      0.9      newGrid[np.logical_and(grid == ON, np.logical_or((totals < 2), (totals > 3)))] = OFF
   117        50      11550.8    231.0      0.7      newGrid[np.logical_and(grid != ON, totals == 3)] = ON
   118
   119        50         17.2      0.3      0.0      return newGrid
```

Visualize (conway_vis.py):