Exercises for Week 3

---

1. Cache Effects

The goal of this exercise is to visualize how the caches affects performance. The exercise will expand on the example from section 6.1.1. in Fast Python. Consider the following code:

```python
import numpy as np

SIZE = 100

mat = np.random.rand(SIZE, SIZE)
double_column = 2 * mat[:, 0]
double_row = 2 * mat[0, :]
```

which is a slightly modified version of the code from the book - using `rand` instead of `randint` and floating point numbers instead of integers.

1. Make a Python program that measures the execution time of `2 * mat[:, 0]` and `2 * mat[0, :]`. Measure the time for at least 1000 repetitions. Hint: remember what you did in Week 2, Exercise 2.5.

2. **Autolab** Make a job script that runs your program. Submit it to the `hpc` queue, use a node with an Intel Xeon Gold 6126, Intel Xeon Gold 6142 or Intel Xeon Gold 6226R processor, and request a single core. Hint: remember the exercises from week 1.

3. Modify your Python program and/or your jobscript to measure the time for `SIZE` ranging from 10^1 to $10^{4.5}$. Do the measurement from a batch job - this is important, so you have control over the hardware!
   Hint: Use `np.logspace` to get logarithmitcally spaces numbers for `SIZE`.

4. Make a loglog plot of the performance of the row and column doubling as MFLOP/s over the size of the matrix in kilobytes. Do they perform the same? How does their performance align with the sizes of the CPU caches? Hint: you can read the cache sizes from the `lscpu` output.

   a) To make the performance difference clearer, plot the ratio of MFLOPS/s.

In the bash output I can see the following from using lscpu

```
L1d cache:     1 MiB (32 instances)
L1i cache:     1 MiB (32 instances)
L2 cache:     32 MiB (32 instances)
L3 cache:     44 MiB (2 instances)
where MiB = 1,048,576 bytes
```

5. Let us focus on the row scaling. Modify the code so `mat` is a row vector, i.e., `mat = np.random.rand(1, SIZE)`. Measure the time for `SIZE` ranging from 10^2 to 10^8 using at least 100 repetitions. Again, do the measurement from a batch job.
   Hint: Again, use `np.logspace` to generate values for `SIZE`.

6. **Autolab** Make a loglog plot of the performance of the row doubling as MFLOP/s over the size of the row vector in kilobytes. What do you see? Do the performance changes align with the cache sizes?

    a) (Optional:) Redo the experiment using `mat = mat.astype('float32')`. Do you observe the same pattern?

2. Efficient data storage with Blosc

The goal of this exercise is to analyze when the time required to read the compressed data and to uncompress it is smaller than the time to read the raw data. The exercise is based on the example from section 6.2.1 in Fast Python. We are going to use Blosc to perform efficient compression and decrompression algorithms. Use the following supporting functions:

```python
import os
import blosc
import numpy as np


def write_numpy(arr, file_name):
    np.save(f"{file_name}.npy", arr)
    os.sync()


def write_blosc(arr, file_name, cname="lz4"):
    b_arr = blosc.pack_array(arr, cname=cname)
    with open(f"{file_name}.bl", "wb") as w:
        w.write(b_arr)
    os.sync()


def read_numpy(file_name):
    return np.load(f"{file_name}.npy")


def read_blosc(file_name):
    with open(f"{file_name}.bl", "rb") as r:
        b_arr = r.read()
    return blosc.unpack_array(b_arr)
```

We add `os.sync()` to force the disk to flush, cleaning the operating system IO buffers as much as possible to ensure a fair comparison between the methods (see Section 6.2 in Fast Python).

1. **Autolab** Write a Python program that takes a number n as a command line argument. It must then generate a 3 dimensional NumPy array of zeros with size $n \times n \times n$ and `dtype='uint8'`.

The program must then measure and print the time it takes to perform each of the following operations:

- Save the array to a file using the provided `write_numpy`.
- Save the array to a file using the provided `write_blosc`.
- Read the array from the created file using the provided `read_numpy`.
- Read the array from the created file using the provided `read_blosc`.

Note, you must print the time for each operation seperately, i.e., four time values should be printed. You may print them on a single line seperated by a space or on 4 separate lines.

2. **Autolab** Make a job script that runs your program with n = 256, 512 and 1024 and submit it to the `hpc` queue. Request 1 core and remember to request enough memory to keep the arrays in memory.
3. Do the same, but now, instead of zero entries, the array will have tiled values of integers up to 256. Hint: You may use the following snippet to generate the array:

```
tiled_array = np.tile(
    np.arange(256, dtype='uint8'),
    (n // 256) * n * n,
).reshape(n, n, n)
```

4. Do it one last time, but now using random integer entries from 0 to 256. Hint: Use the `np.random.randint` function.
5. **Autolab** Compare the time it takes to write and read the files in the three items above for the different n. When should we use blosc? Why? Compare also the sizes the generated files.

   a) In the blosc implementation, change the compression algorithm from `cname="lz4"` to `cname="zstd"`. What improvements do you observe? What was the cost of these improvements?