University of Zurich[UZH]

**Department of Physics** **Scientific Programming with Python**

# Hardware Speedup Exercises

**June 26, 2025**

## General comments

- In case you work with a virtual environment, do not forget to activate it in your terminal, e.g.:
  `source path/to/your/venv/bin/activate`
- download the exercise source code from the exercise page.
- The first four exercises are more about playing with your hardware. The results might differ on different computers. There are no solutions provided to these exercises. Discuss the results together.
- Only in the last two exercises you have to implement new code. For these you find an example solution. You might find even better solutions! If you don't want to solve them by yourself, just look at the solution and try to understand it.
- There are more exercises that you have time for. Do the exercises that interest you.

## 1 Optimizing arithmetic expressions

- Use the script `poly.py` to check how much time it takes to evaluate this polynomial:
  `y = .25*x**3 + .75*x**2 - 1.5*x - 2`
  with x in the range [-1, 1], and with 10 millions points.
  - you can execute the script with different arguments. For example:
    `./poly.py --library numpy --expression-index 0`.
  - Set the `--library` argument to `numexpr` and take note of the speed-up versus the `numpy` case. Why do you think the speed-up is so large?

  If you get a "Permission denied" error you need to set execution permission to the file with the following command: `chmod +x poly.py`
  If your `python` command points to Python2, you need to change the first line (also for the other scripts) to `#!/usr/bin/env python3`.

- The expression
  `y = ((.25*x + .75)*x - 1.5)*x - 2`
  represents the same polynomial as the original one, but with some interesting side-effects in efficiency. Repeat this computation (`--expression-index 1`) for `numpy` and `numexpr` and draw your own conclusions.
  - Why do you think `numpy` is performing much more efficiently with this new expression?
  - Why is the speed-up in `numexpr` not so high in comparison?
  - Why does `numexpr` continues to be faster than `numpy`?
- The C program `poly.c` does the same computation as above, but in pure C. Compile it like this:
  `gcc -O3 -o poly poly.c -lm`
  and execute it with `./poly`
  - Why do you think it is more efficient than the above approaches?

## 2 Evaluating transcendental functions

- Evaluate the expression `sin(x)**2+cos(x)**2` in `poly.py`, a function that includes transcendental functions (`--expression-index 3`).
  - Why is the difference in time between `numpy` and `numexpr` so small?
- In `poly.c`, comment out expression 1) (around line 56) and uncomment expression 3) – the transcendental function). Don't forget to compile again.
  - Do these pure C approaches go faster than the Python-based ones?
  - What would be needed to accelerate the computations?

## 3 Using Numba

The goal of Numba is to compile complex Python code on-the-fly and executing it for you. It is fast, although one should take in account compiling times.

- Open `poly-numba.py` and look at how numba works.
  - Run several expressions and determine which method is faster. What is the compilation time for numba and how does it compare with the execution time?
  - Raise the amount of data points to 100 millions. What happens?

# 4 Parallelism

- Be sure that you are on a multi-processor machine. Use the
  `y = ((.25*x + .75)*x - 1.5)*x - 2`
  expression in `poly-mp.py` by using the argument `--expression-index 1`. Repeat the computation for both `numpy` and `numexpr` for a different number of processes (`numpy`) or threads (`numexpr`). Pass the desired number with `--threads` to the script.
  - How does the efficiency scale?
  - Why do you think it scales that way?
  - How does the performance compare with the pure C computation?
- With the previous examples, compute the expression:
  `y = x`
  That is, do a simple copy of the 'x' vector. What is the performance?
  - How does the performance evolve when using different threads? Why does it scale very similarly than the polynomial evaluation?
  - Could you have a guess at the memory bandwidth of this machine?

# 5 Calculating the Mandelbrot set

The Mandelbrot set $M$[1] is a set of complex numbers $c$ where the sequence $z_{n+1} = x_n^2 + c$ with $z_0 = 0$ does not diverge. In other words, the absolute value $|z_n|$ must remain at or below 2 for $c$ to be in the Mandelbrot set. As soon as the value exceeds 2, it is clear that the sequence will escape to infinity. The elements that diverge are often visualized in the complex plane by assigning a color given by the number of iterations $n$ that are made until $|z_n|$ is for the first time bigger than 2.

In `mandelbrot.py` you will find an inefficient example of how to calculate the Mandelbrot set. You can already execute the script (`./mandelbrot.py`) to see what it does. Try to find a more efficient way. There is already a placeholder in this script where you can insert your code. A comparison of the result and the speed is done when you run the script. The result of your implementation will be shown in the right figure. Do not focus on multi-threading here. Try to improve the code.

Hints:

- What library helps you to avoid `for`-loops (vectorization)?
- No need to avoid all `for`-loops. Think about which loops are the critical ones.

[1] `https://en.wikipedia.org/wiki/Mandelbrot_set`

# 6 Dithering images

*This is a tough one. Only do it if you have some time left and think it is fun to do! I spent half a day on finding a solution.*

In printing or old-school computers there is no grayscale available. There is only 1 bit per pixel (i.e. black or white). Different shades of grey are simulated by using the dithering approach, meaning that the density of black dots in the image approximates the average gray-level in the original. You know this typically from news-papers. There are different algorithms available. Each has pro and cons. In this exercise we look at an algorithm where each pixel can be calculated independently. In `ditherperfomance.py` you find an inefficient implementation of the "ordered dither" algorithm. It already loads an example image. You can also load another image if you like. If you load a too big image it will be downsampled in matplotlib to fit the whole image in the window. That might lead to weird repetitive patterns. If you see this, try to zoom in or load a smaller image.

Try to find a more efficient way to dither the image. Again we do not focus on multi-threading here. Try to improve the code. There is already a placeholder where you can insert your code.

Hints (in increasing level of helpfulness):

- Carefully read the Wikipedia article[2] about ordered dithering to understand what is going on in the example.
- We have two `for`-loops. Try to vectorize!
- Look at the modulo-operator when accessing the threshold matrix. The same elements are accessed again and again, depending on the image pixel position. Maybe you could blow up the threshold matrix in a way that the element indices of the image and the thresholdmatrix fit together? That might help to get rid of the `for`-loops.
- Think in `numpy`! E.g. being `a = np.array([[1,2], [3,4]])`
  and `b = np.array([[1,2], [1,2]])`
  and `c = np.array([[0,0],[0,0]])`,
  you can change certain elemtens in `c` with `c[a>b] = 1`.

[2] `https://en.wikipedia.org/wiki/Ordered_dithering`