Numba - speeding up your Python

Note: these examples are illustrating the power of Numba to speed up your code. While all of our code examples work, they are not trying to be impeccable code, and while all of the suggestions are useful, they are not all sensible things to do in your "real" codes

01 Basic use of Jitting

- Here we will be using a super simple "prime finding" algorithm which works by trial division (see e.g. https://en.wikipedia.org/wiki/Trial_division) with some minor speed ups (ignore even numbers, stop at largest possible single factor). We are going to say that 1 is NOT prime.
- We are going to run this code with and without using the JIT to compile it, and compare the timings

Info:

Recall that the JIT takes time! That means the first run after importing will usually be much slower than subsequent runs. Take care to separate the "JIT time" from the true "run time"

Try:

We have included some code called **jit_example.py**. Start up python3 and import this (`import jit_example`). Try running the main function, for example `jit_example.main(1, 10000)`.

You should have seen an output of "Found 1229 primes in ..... seconds". Run the same line again. What happens to the time taken? Why?

We can run a version without using the jit by passing a third parameter to the main function, called do_jit. This defaults to True, but if we pass False, it will run using non-jitted functions. Try this. What happens to the time taken?

- 02 Keeping jitted functions around

- Recall that numba can cache compiled functions on disk. Here we'll look into this a little

Info:

While the file based cache will be created inside an interpreter session, it's easier to see it working by invoking python afresh each time. We made sure the timings included only the core function, but in practice you do care about the time to start up the interpreter - don't work this way unless you need to

Try:

Edit the **jit_example.py** code to use the Numba cache. Exit python and run the code a few times directly using `python3 jit_example.py`. Does it get faster after the first run?

You might notice this is still slower than staying within the interpreter. Why might this be? Try using a larger range so that the code takes longer. Is this slowdown a fixed amount or a fractional one?

- 03 Using Eager compilation

- Numba can also be told to compile a function at import time, instead of when it is run
- Recall that this means it can only compile for the types you request - you can't mix this with "on-demand" compilation

Info:

If you're not familiar with how to choose the correct data type, a useful rules-of-thumb are - for numbers up to about a billion, int32 is enough. This is enough for all lengths of array you are likely to use, or any number you might type in. For floating point (decimal) numbers, a float32 can hold numbers up to 1038 and has about 7 decimal digits of accuracy. A float64 holds numbers over 10300 with about 16 decimal digits of accuracy. For most purposes, it is the decimal digits which inform your decision.

Try:

The code **jit_eager.py** uses eager compilation. We have put timing calls around the imports of the functions, as well as inside the run. Try running this with and without the eager compilation (comment or remove the jit lines). What happens?

How does the compilation time compare to the run time? Do the various times add up to (roughly) what you saw from the original jitted code?

What are the downsides of this? In what circumstances might a user want to call your functions with types you didn't think of?

The code **jit_eager_broken.py** is a bit broken. What is wrong with it? Why? Can you fix it?

- 04 Numba in Parallel

- The numba jit can also try to paralellise code. Here we give an example of a parallel loop
- This is a tricky subject - usually it is up to you to make sure that iterations of the loop are independent (can be freely reordered) and strange things will happen if you are wrong or this changes

Info:

You may recognise part of this code from the Clusters, Queues and Modules Core module. Here we go into a little more detail about it

Try:

The code **numba_parallel.py** uses numba to run a loop in parallel across multiple processors. You may want or need to use this

`export NUMBA_NUM_THREADS=4 ` to set the number of parallel threads to run. Try running the code.

The function `broken_prange_test` has a problem - the result will depend on the way the work is divided. In serial it is OK, because we only change values we have already used. Change the code to use this. Run it a few times. What happens? Why?

Can this code be fixed? Suppose rather than totalling items, we were doing something that was a lot of work and worth parallelising. What could we do?

- 05 Other bits of Numba - Stencil

- Numba has a lot of helpful capabilities - too many to cover them all! Here we are going to focus on the Stencil decorator
- We saw an example of image blurring in the Core module on Notebooks, and again in some of the Skills modules. Here we show another variant of this

Info:

Convolving with a Gaussian, like we did before, is a much better image blur, but this simple algorithm works OK. Note that stencil does not require you use any particular points, and doesn't have to have any symmetries.

Try:

The code **numba_stencil_blur.py** uses numba.stencil to apply a blurring kernel to the same image we saw earlier. Can you see how this works?

Edit the code to use a wider-range blur, say including the nearest 2 cells. You might want to work-in a Gaussian or other shape function to make this better.

We excluded diagonals completely, which leaves us with some artefacts compared to the better Gaussian blur. Can you see how you would fix this using stencil?

Try coding the original 4-point blur as a direct loop. How does this perform? Does Stencil improve the speed?