Raster processing

Some core computer science knowledge for non-computer science GIS people working and coding with large raster datasets

1. Background - the theory section

Introduction

We're going to talk a little bit of theory here.

Why? Well I think it's helpful to think of scientific programming as fitting into one of two categories:

- Scripting, or "driving" something - tying together a bunch of existing tools

or

- Lower level programming; literally telling the computer what to do (e.g. add this number to this one) or, more likely, somewhere in between(*)

* *(Then totally separately we've got application-development type coding, web development, etc where we don't really care so much about what the computer is REALLY doing, it's more about what the user sees.).*

Motivation

When it comes to writing code for doing stuff with big rasters, we're probably very much at the second end of that scale.

Stuff that's quite simple in theory - most of the time we're doing relatively simple calculations.

- But the trick is in doing it well, or fast, or within the memory we've got, etc

We're not computer scientists so we probably don't want to spend a lot of time thinking about the nuts and bolts.

We don't need to know *everything* but it's worth knowing a bit.

- Why are some things fast and some things slow?
- Where should we look to try speeding things up?
- How much does "slow" matter?

Rasters

Fundamentally a CPU - the brain of the computer - just does arithmetic with numbers - that's all. Any time a computer is doing anything more than this, there's just layers upon layers of abstraction turning what you see / do into basic sums.

Now, rasters are basically nothing but a bunch of numbers - also known as "arrays", and all we're doing with them is some (probably fairly simple) arithmetic operations.

So doing raster processing efficiently is actually quite a "pure" type of programming. We're doing simple things with numbers, but we're doing it with LOTS of numbers - a single global "1k" grid is nearly a billion cells.

So it's really beneficial to know a bit about how computers work, because it's helpful to get a feeling for:

- whether something is as fast as it could be: is this piece of work always going to be slow, or is my code just not very fast?
- if not then where the easy wins might be to make it better

This is a pretty unusual thing to worry about in modern programming. Computers are so quick now that most times we just don't need to worry about this.

Performance

Step back. If CPUs really just do sums, how does a computer happen? All those games and windows and things?

- Because they do sums REALLY REALLY FAST
  - (and some smart code turns everything else we want done into lots and lots of sums)

How fast?

- A 3GHz CPU means there are 3 **billion** clock cycles a second
- Your laptop CPU probably has 4 cores -> that's 12 billion total cycles a second. Every. Second.
- On *each cycle* a modern CPU can theoretically do several basic "FLOPS" (Floating Point Operations) or, "sums"

Who really thinks about what numbers like that mean? These are massive numbers. It's worth trying to visualise them. Put it this way - **in theory** the CPU in your desktop could power a calculator for every person in the world, each doing a sum every second, without even breaking a sweat. I think that's pretty amazing.

Oh right. Well why's my computer slow and annoying then?

It's REALLY REALLY hard to keep a modern CPU fully occupied. It's the fastest part of the system (excluding the graphics card, maybe), and so the chances are that it will spend most of its time waiting for other stuff to catch up.

Just getting all the instructions and their associated data in the right place at the right time so that the CPU can do something useful every cycle is a big problem. So the CPU ends up "wasting" a lot of its time.

I'm going to miss out a lot of detail - but coming up are some basics about how data and instructions get to the CPU and where the bottlenecks are that stop it doing useful sums every cycle.

We're going to look at two fundamental areas that we need to understand just a little bit about to get into doing a good job of processing large arrays quickly.

1. How memory and storage works
2. How languages work

Memory and storage

begin indepth nerd lesson #1

Equivalent speeds of different levels of data storage

Assuming a 3.3GHz CPU - 1 cycle = 0.3 nanoseconds (ns).

*< side note - light travels ~ 10cm in this time. So in a modern PC the **SPEED OF LIGHT** is actually relevant to the physical layout of where things like the RAM are placed >*

EVERYTHING in the computer is much much slower than the CPU.

You probably already know that if you're loading from disk that's going to be slow, so you want to work in memory (RAM) when possible.

Yeah, true, but that too is incredibly slow, for the CPU. To keep things moving along we need to know about cache, too.

How slow? Let's scale it up so that the CPU ticks once per second. Here's how long everything else might take on that scale... (scroll down)

Event	Latency (real time)	Scaled to 1 cycle = 1 second	Typical capacity
1 CPU Cycle	0.3 ns	1 s	
CPU register access	0.3 ns	1 s	8 - 256 registers
Level 1 cache access	0.9 ns	3 s	32KB - 256KB
Level 2 cache access	2.8 ns	9 s	128KB - 24MB
Level 3 cache access	12.9 ns	43 s	2 MB - 32MB
Main memory access (DRAM, from CPU)	120 ns	**6 min**	4GB - 32GB
SSD storage access	50 - 150 us	**2-6 days**	64GB - 1024GB
Normal HDD access	1-10 ms	**1-12 months**	256GB - 4 TB
Internet: San Francisco to UK	81 ms	8 years	
TCP packet retransmit	1-3 s	105-317 years	
Physical system reboot	5 min	32 millennia	

If the CPU has to read something from RAM (this is what we'd think of as fast, right) then on this scale that's the equivalent of a 6 minute delay, doing nothing. In which time your 4-core laptop CPU could have done several THOUSAND operations.

If it has to go to disk - then that's an equivalent delay of up to a **YEAR**

So - if we're coding something where we expect there to be so many calculations that the CPU will be the limiting factor, then we REALLY REALLY need to make sure that nothing else holds it up!

- And - this is the kicker - it's not really enough just to say "I've loaded it into RAM" and assume that the code will therefore be fast
- NOTHING is fast compared to the CPU itself.

OK what do we need to know to fix this?

- Thankfully, not too much.

I mean there's loads to be known but if we just get to grips with these basics we'll be 90% of the way there. The rest would be much harder: there's a real case of diminishing returns with optimising computer programs for speed.

- From RAM upwards, this "hierarchy" of memory is pretty much dealt with automatically (unless we're really serious about things). We don't need to load things into cache or whatever. That just happens.
- We just need to vaguely know HOW it is done automatically - what assumptions are made by the hardware and computer designers
- Then design our code so that we don't screw up those assumptions more than we can help.

Things to know about memory

There are several levels of memory: disk storage, RAM, then (usually) 3 levels of CPU cache, and finally the actual registers which the CPU works with directly.

- Each level is smaller, but faster, than the last.
- Memory is laid out, and read, in blocks. So every byte in memory (RAM) has an address - like a linear number starting from zero - but we don't read just one byte at a time
- If we ask to read a value from RAM, even just a single number, we actually read a whole block of maybe a few thousand values, which then get stored further up the hierarchy
- Inevitably the levels above will be full (being smaller) so something else will get shoved out
- When we ask for memory at a particular location then we'll get it from the L1 cache if it's there. If not, then load it from the L2 cache into the L1. And so on.

Exactly how this happens is a dark art and depends on the particular computer we're running on. But there's one fundamental principle to know:

- Once we read something, it's much better if the next thing we ask to read is the next number along, as that will already be in cache.

If we use the number that's already in cache then we don't have to wait. And if we keep doing this in order then the system will be smart enough to notice what we're doing and pre-load things to cache for us.

If we don't do this then maybe the CPU will do 1 second of work, then want something different, and have to wait for 6 minutes for the next bit of data to be loaded.

See Locality of reference (https://en.wikipedia.org/wiki/Locality_of_reference#Spatial_and_temporal_locality_usage)

Let's take a look

We can show this in practice. We'll make two arrays and multiply them together element-wise.

Once by going across the rows first, and once by going down the columns first. We'll time how long each takes.

In [3]:
```
import numpy as np

# create two arrays (rasters) of 2000 (h) * 4000 (w) elements (pixels) with random value
s
h = 2000
w = 4000
testarr = np.random.random((h, w))
testarr2 = np.random.random((h, w))
# and an empty one for the results
outarr = np.empty((h, w))
```

In [8]:
```
def multiply_Silly():
    for x in range(w):
        for y in range(h):
            outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

In [9]:
```
# Note this timeit thing: it's a feature of IPython, most handy
%timeit multiply_Silly()
```

```
1 loop, best of 3: 5.02 s per loop
```

In [6]:
```
def multiply_LessSilly():
    for y in range(h):
        for x in range(w):
            outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

In [7]:
```
%timeit multiply_LessSilly()
```

```
1 loop, best of 3: 4.05 s per loop
```

What happened?

Literally all we did is changed the order of the loops, but one version ran noticeably faster than the other.

Arrays in python, or rather numpy, (and other C-based languages) are laid out in memory in "row-major" order - i.e. the values of one row in turn followed by the values of the next row etc. (Some other languages are the other way round!)

In the first test we iterated over it the other way round: we went down a whole column first then onto the next column. This meant there was more time waiting on memory access and less time spent doing anything useful

Convinced? You shouldn't be, as actually that was "only" a 20% speedup. In fact this is a bad demo because:

- the array was small enough that some of each row still probably survived in the cache until we next needed it
- there's so much else going on in a language like Python to slow things down that this effect is only a small part of the problem (more on this later)

Let's try again to really destroy any caching, we'll multiply the array elements in a totally random order:

In [10]:
```
# create a list of values to index the height and width of the raster in a totally rando
m order
random_x_indices = np.arange(w)
random_y_indices = np.arange(h)
np.random.shuffle(random_x_indices)
np.random.shuffle(random_y_indices)
# multiply the elements of the arrays in this random order
def multiply_ReallyDumb():
    for y_i in range(h):
        y_index = random_y_indices[y_i]
        for x_i in range(w):
            x_index = random_x_indices[x_i]
            outarr[y_index, x_index] = testarr[y_index, x_index] * testarr2[y_index, x_i
ndex]
%timeit multiply_ReallyDumb()
```

```
1 loop, best of 3: 5.96 s per loop
```

So even in this terrible example we've got a 50% difference in how quickly an algorithm runs, simply by running it in a different order.

With a more complex example, loading more data and doing more things, the difference would be even bigger.

Remember: ensure that you iterate through the data in the order it's stored in memory. And also try not to do lots of other stuff in between iterations that might cause it to be shoved out of the caches.

congratulations you have completed nerd lesson #1

Languages

begin indepth nerd lesson #2

Remember our best effort above:

In [11]:
```
%timeit multiply_LessSilly()
```

```
1 loop, best of 3: 3.95 s per loop
```

That's 4 seconds to do 8 million multiplications of random floating point numbers. Is that good?

4 seconds is about 12 billion CPU cycles on my machine. So that's 1500 CPU cycles to multiply each number.

So... no. that is NOT good. It's terrible.

*Think about that - we just did 8 million multiplications in a couple of seconds and yet that counts as **catastrophically** slow! Be prepared to work out roughly how fast something "ought" to be*

Anyway, why? What else is going on?

In python and using the numpy library we don't actually need to use a loop to multiply one array by the other. We can just tell it to do it directly.

Would that be better?

In [12]:
```
%timeit outarr = testarr * testarr2
```

```
10 loops, best of 3: 27.8 ms per loop
```

Well hello - that's more than 100 times faster. ONE HUNDRED TIMES. What's going on?

In the first example we were literally spelling out what should happen - multiply the number at this position in the first array, by the number at the same position in the second array, and put the answer in the output array.

Computers are dumb and need this kind of instruction so it's not a surprise that we had to do that. But in that case, how did the new version work? Who tells it what we actually mean there?

Actually someone else has already written those instructions and they're in the "numpy" python library. You call that code by just saying `testarr * testarr2` and then it knows to go off and run that loop. The loop's still happening, you just don't need to bother writing it.

(note - you'll hear the term vectorised code. That just means this kind of thing: writing something that should be applied to individual numbers and applying them to a whole array at once)

The code inside numpy probably looks the same as what we wrote before, except it isn't written in Python, it's written in C. And that's the main difference.

Python is a really SLOW language* and C is a really FAST one. Why? And if so why don't we just write in C?

What's a programming language anyway?

You can think of the instructions the CPU actually runs as just a few, simple ones like

```
`add the contents of location 1 to the contents of location 5`
```

or

```
`take the square root of location 12`
```

Machine code, as these CPU instructions are called, is characterised by being really really boring and long and impossible to read or write by humans. Everything has to be spelled out - and this doesn't only mean what to do, but exactly what to do it with - where in memory to find things. (i.e. what number, what bit of memory, does "X" mean? gotta work that out first)

So even for work that actually consists of simple tasks like the one we're doing, nobody really wants to write instructions like that (how are you going to remember what memory location everything is?).

And most work consists of way more abstracted things than this, like "resize this window" so there's a lot to be done to turn that into the simple instructions, and there will be a LOT of the simple instructions.

So that's why we have programming languages. To fill the gap between what you can find a human being nerdy enough to want to write, and what the CPU actually needs. The human writes in a programming language and then "something" turns that into the instructions for the CPU. That "something" is called a **compiler**.

What's the difference between languages

1 - Interpreted vs compiled

So whatever language we write in, at some point it has to be turned into machine code instructions by a compiler.

One big difference between languages is **when** this happens. Does it happen once, in advance, leaving a batch of machine-code instructions ready to go? Or does it happen as-and-when it's needed?

This is the difference between a compiled language and an interpreted language. C is a compiled language, and Python is an interpreted language.

In either case the language still has to be translated (compiled) into machine code sometime - it's just a question of when.

What's the difference?

Different machines might have different sets of CPU instructions or memory layouts.

So a bunch of machine code has to be compiled for a given machine - we can't just run a Windows application written for x86 processors on an iphone.

So being compiled in advance means that things are less flexible - it can only run on a machine like the one it was compiled for.

Whereas a language like Python we literally can run anywhere (at least, anywhere that has python). The python interpreter turns it into machine code instructions only when the code actually runs. This is much more flexible, but loads slower.

(Note - actually, many languages, including Python, are actually somewhere between the two)

What's the difference between languages

2 - Dynamic vs static

Say we have a basic loop to add up an array of values, in python

```
def sumValues(arr):
    n = len(arr)
    total = 0
    for i in range(n):
        total = total + arr[n]
```

Now say that the values in arr are all 32-bit floating point numbers. (This means that each number takes up precisely 4 bytes / 32 bits of memory, no matter what the number is).

In a language like C we would write something like this

```
float sum_array(float a[], int num_elements)
{
   int i, sum=0;
   for (i=0; i<num_elements; i++)
   {
     sum = sum + a[i];
   }
   return(sum);
}
```

The fact that the array contains only floating point numbers is known in advance, and so is the length of the array. So we can figure out where everything sits relative to everything else. Each number in the array will be exactly 4 bytes after the previous one - and so, almost certainly already in the cache.

So the compiler can go ahead and generate some instructions like this

```
set the address pointer T to point to memory location 0
set the memory at address T to value 0
set address pointer P to the place where the a[] starts
add the value at location P to the value at location T
move pointer P forward by exactly 4 bytes
add the value at location P to the value at location T
<repeat num_elements times...>
```

In other words:

- We know exactly where everything is and can easily just say what should be added to what just in terms of memory locations, without looking up addresses.
- When we load one number from RAM we will get the next ones too, so we won't need to go back to RAM next time

This is one reason why a so-called statically typed language can be faster.

- We can write instructions that just "know" where to look for things.

And we can check **when the program is compiled** that the bits of data we will give to those instructions are indeed what we say they are

- i.e. that arr[] really is an array of floating point numbers and not some boolean values or whatever. We don't have to take time to do this when the program is running.

But in the python version we didn't say what arr was. It could be an array of float numbers. But it could equally be a list of random crap like this:

```
arr = [1, 3.0000, -20e4, "cat", 4, "bruce"]
```

In other words the contents of `arr` are all different. So we have no idea whether they are all 4 bytes apart in memory or even if it's appropriate to add them together.

In fact it's even worse as arr isn't actually a continuous region of memory containing those things. That's not how a Python list works.

Instead `arr` contains a load of variables. Each variable is basically just a signpost pointing to other bits of memory plus instructions how to read each of those bits of memory (should the data at this address be read as a number or as the string "bruce"?).

So there's a **lot** of jumping around and checks before we even get our numbers ready for adding.

Worse: what if arr is some input that the user has provided? it could be different every time. So we can't even work this stuff out in advance. So we can't create the machine code instructions until it's actually time to run them.

Because of all this, when we do run the code there is way, way more work to do to make sure everything happens like it should:

```
    - find a bit of free memory big enough to hold a floating point number
    - create a label for this bit of memory, call it total and point it to the m
emory
    - set the target memory that "total" points to, to 0
    - find the place in memory that arr points to
    - find the properties of this "arr" thing and see how many items are in it
    - find the place in memory that the label of the first thing in arr points t
o
    - see whether the label for that says it is meant to be a number rather than
text
    - if it is then see how many bytes long the label that number is
    - read the bit of memory pointed to by label, for the appropriate number of
 bytes
    - add the result to the bit of memory pointed to by the label called "total"
    - repeat the appropriate number of times
```

the point is this

This is all a big simplification, but actually it covers most of what you need to realise. To summarise what we've learned:

- the less work we do in writing code, the chances are, the more work the computer has to do in order to run that code and ensure it does what we intended
- conversely, the more information we can give the computer about exactly what we're asking it to do, the quicker it's probably going to be
- languages like C are really low level and you can literally move bits of memory around and get the computer to do exactly what you're telling it do
- lanaguages like Python are the opposite end of the spectrum and make it really easy for us humans but the cost is that they're pretty slow, because they have to figure stuff out on our behalf
- Because modern machines are so fast this slowness doesn't matter most of the time - it's still pretty quick on a human scale.
- **BUT** doing complicated stuff with enormous rasters is one area where it really DOES matter. Which is why we just did it.

congratulations you have completed nerd lesson #2

congratulations you have completed theory section

2. Practically Useful Section

What does this mean for us?

Most people who really write code where performance matters are massive geeks like computer games programmers, who are happy to get down to the nuts and bolts of writing something like C

As GIS people who are dealing with raster datasets we - along with many data scientists - are probably on a bit of a frontier of skills and requirements, between wanting the performance you can get from C without wanting the hassle of writing it. Writing good C is really really hard.

So what to do? How can we get fast code whilst also being lazy?

Easy wins

- Numpy is a good first step.

In [13]:
```
%timeit outarr = testarr * testarr2
```

```
10 loops, best of 3: 25.7 ms per loop
```

It takes away some of the problems of python (lack of typing, etc) and thus lets some repetitive tasks happen much more quickly for all the reasons we've discussed.

- But it can be hard or impossible to program some things - can we express what we want to do in that "vectorised" form?
- Also it's still only single-threaded: it will only run on one CPU core. Much of what we do could be split across cores, so this is a waste

- numexpr is another relevant python package for some relatively simple array operations and can handle running things across multiple threads:

In [14]:
```
import numexpr as ne
%timeit outarr = ne.evaluate("testarr * testarr2")
```

```
10 loops, best of 3: 20.4 ms per loop
```

numexpr builds on numpy and uses numpy arrays. You specify the operation you want to do as a string then it goes off and does it in C, using multiple threads if available. For big but simple array operations this is a great choice.

Cython

Spoiler: this is my go-to solution for most raster processing algorithms.

You write "normal" python, using all the standard loops and stuff, give it a few extra clues, then it writes C for you! The python you write will be a straightforward, easy-to-understand, loop rather than head-exploding vectorised stuff.

Because it's C, we then need to have a compile step. But it's not that much hassle really and even easier in a notebook like this. We're not going to cover how to set it up in this notebook, we'll do that later. However once it is properly installed on your machine, you can just do this:

In [ ]:
```
%load_ext cython
```

Then you can write and compile C code just like this. I'm not going to explain it now, we'll do that in a later notebook. Just note that it looks fairly similar to the python version with a couple of slight additions.

In [16]:
```
%%cython
cpdef multiply_NotSilly(double[:,::1] testarr, double[:,::1] testarr2,
                        double[:,::1] outarr):
    cdef int x, y, h, w
    h = testarr.shape[0]
    w = testarr.shape[1]
    for y in range(h):
        for x in range(w):
            outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

In [15]:
```
%timeit multiply_NotSilly(testarr, testarr2, outarr)
```

```
10 loops, best of 3: 35.9 ms per loop
```

OK so that's not faster than numpy, yet. Add `-a` to see what's going on - you can see the C that gets written. The more yellow highlighting there is the less efficient it is (click the + to expand)

In [19]:
```
%%cython -a
cpdef multiply_NotSilly(double[:,::1] testarr, double[:,::1] testarr2, double[:,::1] out
arr):
    cdef int x, y, h, w
    h = testarr.shape[0]
    w = testarr.shape[1]
    for y in range(h):
        for x in range(w):
            outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

Out[19]:

Generated by Cython 0.26.1

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
+1: cpdef multiply_NotSilly(double[:,::1] testarr, double[:,::1] testarr2, double
[:,::1] outarr):
 2:     cdef int x, y, h, w
+3:     h = testarr.shape[0]
+4:     w = testarr.shape[1]
+5:     for y in range(h):
+6:         for x in range(w):
+7:             outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

So there's a lot of stuff there. Our few lines of python got turned into pages and pages of C. What's happening is called "bounds checking".

What happens here?

```
arr = [1,2,3,4,5]
x = arr[8]
```

We get an error of course. But only because some work goes on behind the scenes to figure out this is an error.

In C, this doesn't happen automatically and you can access a bit of memory you didn't mean to, leading to death and destruction.

Cython will generate code that does these checks for us to keep things safe, but they take time. That one line of code above got turned into like a hundred lines of C, which would have been no fun at all to write.

If we're sure we know what we're doing then we can turn that off, with two lines that tell it not to do the checks and to promise that we won't try to use negative indices:

In [20]:
```
%%cython -a
import cython
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef multiply_NotSilly(double[:,::1] testarr, double[:,::1] testarr2, double[:,::1] out
arr):
    cdef Py_ssize_t x, y, h, w
    h = testarr.shape[0]
    w = testarr.shape[1]
    for y in range(h):
        for x in range(w):
            outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

Out[20]:

Generated by Cython 0.26.1

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
+01: import cython
 02: @cython.boundscheck(False)
 03: @cython.wraparound(False)
+04: cpdef multiply_NotSilly(double[:,::1] testarr, double[:,::1] testarr2, double
[:,::1] outarr):
 05:     cdef Py_ssize_t x, y, h, w
+06:     h = testarr.shape[0]
+07:     w = testarr.shape[1]
+08:     for y in range(h):
+09:         for x in range(w):
+10:             outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

In [21]:
```
%timeit multiply_NotSilly(testarr, testarr2, outarr)
```

```
100 loops, best of 3: 13.9 ms per loop
```

13.9 milliseconds! And this is still single-threaded.

That's 8 million double-precision multiplications in ~14 milliseconds.

Now we're properly getting somewhere. Work it out and we're looking at around 5 CPU cycles per calculation now. In theory we might be able to do better but at this point we're looking at diminishing returns.

Now that we're generating C code we can get a better idea of the implications of going through the arrays in the wrong order. Rewrite the back-to-front version in cython:

In [25]:
```
%%cython -a
import cython
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef multiply_Cy_Reverse(double[:,::1] testarr, double[:,::1] testarr2, double[:,::1] o
utarr):
    cdef Py_ssize_t x, y, h, w
    h = testarr.shape[0]
    w = testarr.shape[1]
    for x in range(w):
        for y in range(h):
            outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

Out[25]:

Generated by Cython 0.26.1

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
+01: import cython
 02: @cython.boundscheck(False)
 03: @cython.wraparound(False)
+04: cpdef multiply_Cy_Reverse(double[:,::1] testarr, double[:,::1] testarr2, double
[:,::1] outarr):
 05:     cdef Py_ssize_t x, y, h, w
+06:     h = testarr.shape[0]
+07:     w = testarr.shape[1]
+08:     for x in range(w):
+09:         for y in range(h):
+10:             outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

In [24]:
```
%timeit multiply_Cy_Reverse(testarr, testarr2, outarr)
```

```
1 loop, best of 3: 191 ms per loop
```

191ms, or THIRTEEN times slower than the version which does the exact same thing the other way round.

Hopefully a more convincing demo of the need to understand the order we go through things from memory.

Enough of that silliness, back to the right way round.

Next - we have lots of CPU cores. Why not use them?

In [11]:
```
%%cython -a --compile-args=/openmp --link-args=/openmp --force
from cython.parallel cimport parallel, prange
import cython
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef multiply_NotSilly_P(double[:,::1] testarr, double[:,::1] testarr2, double[:,::1] o
utarr):
    cdef Py_ssize_t x, y, h, w
    h = testarr.shape[0]
    w = testarr.shape[1]
    with nogil, parallel():
        for y in prange(h):
            for x in range(h):
                outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

Out[11]:

Generated by Cython 0.26.1

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
+01: from cython.parallel cimport parallel, prange
 02: import cython
 03: @cython.boundscheck(False)
 04: @cython.wraparound(False)
+05: cpdef multiply_NotSilly_P(double[:,::1] testarr, double[:,::1] testarr2, double
[:,::1] outarr):
 06:     cdef Py_ssize_t x, y, h, w
+07:     h = testarr.shape[0]
+08:     w = testarr.shape[1]
+09:     with nogil, parallel():
+10:         for y in prange(h):
+11:             for x in range(h):
+12:                 outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

In [12]:
```
%timeit multiply_NotSilly_P(testarr, testarr2, outarr)
```

```
100 loops, best of 3: 5.77 ms per loop
```

BOOM

That's more like it!

So we started with this

```
def multiply_LessSilly():
    for y in range(h):
        for x in range(h):
            outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

and it ran in ~4 seconds

We've ended with this

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef multiply_NotSilly_P(double[:,::1] testarr, double[:,::1] testarr2, double[:,::1]
outarr):
    cdef Py_ssize_t x, y, h, w
    h = testarr.shape[0]
    w = testarr.shape[1]
    with nogil, parallel():
        for y in prange(h):
            for x in range(h):
                outarr[y,x] = testarr[y,x] * testarr2[y,x]
```

It doesn't look much different, it isn't much harder to write, and yet it runs in 5.7 MILLIseconds

If we scale that up to processing something the size of a global 1k raster, then we're looking at the difference between around an hour to do one raster, vs doing three of them every second.

That's definitely worthwhile.

Take-home message

The key message in all of this is to have some idea of how much you're asking the CPU to do, vs how much it theoretically might be able to do. This will enable you to think about whether your slow code is slow because there's just so much to be done, or whether it's because of some features of your code that you could change.

Here we were doing 8 million multiplications, plus the associated work of getting the data out of memory etc. It ran in around 40 million CPU cycles so about 5 cycles per operation. That's a little way off the theoretical best, then, but it's in the ballpark. So we can probably be happy with that.

Congratulations you have completed nerd level: 1