Numba

Warwick RSE

27/3/2019

What is Numba?

- Numba is a “just-in-time” compiler that converts Python code to native machine code the first time that it is called
  - Uses the “LLVM” compiler backend which is also used by some C/C++/Fortran compilers
- It can only operate on a subset of Python and Numpy routines
  - Quite a large subset though

What is Numba?

- Numba claims “Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN”
- This isn’t really true in practice
- But you can get down to being only factors of 3-5 times slower than C or Fortran
- Using Numba is much, much easier than rewriting in C or Fortran
  - Doesn’t always help though

How to use numba?

- Pure Python
  - ```
    import numpy as np
    def distance(a,b):
        return np.sqrt(a**2+b**2)
    ```
- Numba
  - ```
    from numba import jit
    import numpy as np
    @jit
    def distance(a,b):
        return np.sqrt(a**2+b**2)
    ```

How to use Numba

- Numba is mostly implemented as function decorators
- The above is the simplest one
  - “lazy compilation”
  - When you call the function with any given types of parameters Numba will generate a compiled function matching the types and then use it

How to use Numba

- This means that the first time you call a function it will be much slower because it has to compile the function
- Subsequent calls with the same types of parameters will use already compiled functions and are rather faster
- Can enforce “eager” compilation by specifying types in the decorator

Types

- Python hides types from the developer as much as possible
- This is Python trying to be helpful
- Types are built right into the core design of the CPU
- Once you get outside normal Python you have to specify types in some way

How to use numba?

- ```
  import numpy as np
  from numba import jit, float32
  @jit(float32(float32, float32))
  def distance(a,b):
      return np.sqrt(a**2 + b**2)
  ```
  ```

How to use Numba

- `@jit(float32(float32, float32))`
  - Says “compile a function that returns a 32 bit float and takes two 32 bit floats as parameters
- Function compiles immediately so overhead when writing/loading function rather than when running it
- Turns off the lazy compilation
  - Will fail with TypeError if you call it with any types that don’t match the specification (even if the function is otherwise valid)

How to use Numba

- `@jit([float32(float32, float32), float64(float64, float64)])`
- Can specify multiple types for eager evaluation by supplying a list of types
- Important to specify the most restrictive types first since the first matching signature will be used
  - Since float32s can be exactly represented as float64s if these are the other way around the float32 version will never be called

How to use Numba

- Common types
  - void - returns nothing or takes no parameter
  - int8,int16,int32,int64 - Integers of specified bit length. Also “uint” (e.g. uint8) versions for unsigned integers
  - float32, float64 - Floating point at 32 and 64 bits length
  - Arrays are specified like numpy slices
    - float32[:,:] is a 2D 32 bit float array

Numba and profiling

- One thing that you might notice while profiling numba code is that some functions “disappear”
- This is because of a process called “inlining”
  - Compiler actually pastes code from small functions straight into the place where it is called from
  - Gives quite a big performance improvement but can make profiling a bit confusing

Numba compilation

- When Numba compiles Python code it follows a few rules
- By default it gives solid guarantees that everything will work the same as in normal Python
  - But it might have to be conservative to make everything work
    - Object mode - Everything will work the same as in core Python but performance gain will be much smaller
    - No Python mode - Loses most Python overhead and gives speed that's much closer to compiled code

Numba compilation

- Can be useful to prevent compiling to Object Mode if you want performance
  - `@jit(nopython=True)`
  - `@njit`
- Raises an error if Numba can’t compile in No Python mode

Numba compilation

- `@jit(nogil = True)`
  - Turns off the Python Global Interpreter Lock
    - Allows Python threads to actually work in parallel
  - Requires compilation in No Python Mode

Numba compilation

- `@jit(cache = True)`
  - Tells Numba to create a file based cache for the compiled function once it's compiled
  - Useful if you have a function that is slow to compile that you keep reusing without changing

Numba compilation

- `@jit(parallel = True)`
  - Tells Numba to try to create a function that runs in parallel on multiple processors
  - Quite restrictive in what will work
    - Must be successfully compiled in No Python mode
    - See http://numba.pydata.org/numba-doc/latest/user/parallel.html#numba-parallel for details
    - Can also write manual parallel loops (same link)
  - Can slow your code down (sometimes a lot!)

Numba compilation

- `@jit(fastmath = True)`
  - Turns off strict IEEE compliance of floating point maths
  - Reduces accuracy in some cases
  - Might be useful but is a bit risky

Numba performance

Numba performance

- So long as Numba can compile to No Python mode performance will be within a few times that of a C/Fortran version

- But the rules for how to maximise performance are different to normal Python

  - Array operations are no longer substantially faster than loops so long as you get the memory access right

  - Your main problem is keeping the CPU supplied with data

Numba performance

- Calling library code will generally prevent No Python compilation so library code is slow
- More or less have to write C/Fortran like Python code
  - Just follow rules for optimisation of compiled codes
  - That one about memory order really is the big one!

Numba performance

- Intel SVML is a library that Numba can use to do faster maths than it's built in routines
- It also has even faster versions that work with the "fastmath" option
- Can be 2x faster than Numba code without it
- Part of Intel Compiler Suite so available on SCRTP desktop
  - Can also install in various ways, see http://numba.pydata.org/numba-doc/latest/user/performance-tips.html
- No need to do anything, Numba will use if it can

Numba performance

- The numpy linear algebra routines are “mostly” compatible with No Python compilation
- Because these are backed by BLAS/LAPACK/Intel MKL etc. they are faster than “normal” compiled code
- Still definitely worth using these routines in Numba compiled code

Other Numba features

Numba @vectorize

- Use in the same way as @jit
- Produces a vector function to operate on an array same as numpy.vectorize
- But compiles the function before it returns it so it's much faster than numpy.vectorize
- Takes a "target" parameter (@vectorize(target="")) which can be
  - cpu - Run on a single core CPU
  - parallel - Run on multiple cores CPU
  - cuda - Run on a GPU

Numba @cuda.jit

- Numba can convert some types of Python function to CUDA code that will run on a GPU
- This is rather more involved and there isn't time to cover it today
- But it isn't much harder in principle
- GPU performance is often comparable to CPU performance for general numerical tasks nowadays
  - Used to often be faster but many core CPUs have caught up

Numba @stencil

- The @stencil decorator is a bit like the @vectorize in that it generates a function that applies to elements of arrays

- But in this case it allows you to specify a function that applies not only on the *current* element of an array but also the neighbouring elements

- Write a function that takes an array parameter and then access it using an “offset” from 0 in each index

Numba @stencil

- So to average nearby values

- ```
  @stencil
  def blur(A):
      return 0.25 * (A[-1,0] + A[1,0] +
          A[0,-1] + A[0,1])
  ```

- When this function is applied to an array the array indices are used as an offset and is moved across the array

- Have to specify what to do at the edges which by default are just clamped to 0
  ```