EE2703: Optimizing Python Performance

Annangi Shashank Babu, EE21B021

April 19, 2023

Contents

1 How does cython work

Cython is a programming language that is a superset of Python, which means that it adds additional syntax and functionality to Python. Cython optimizes the python code by adding static typing to Python, which allows the compiler to optimize the code for speed. In addition, Cython allows Python code to call C and C++ functions directly, which can further improve performance.

1.1 Optimising Fib.py

we optimize a script containing a function that computes fibonacci numbers, the .pyx file is the cython code that is the optimized version of the python code and setup-fib.py file helps us to create a extention module so that this optimized function can be imported to python scripts for use.

Listing 1: fib.pyx

```
#!python
#cython: language_level=3
# distutils: language = c++
# cython: boundscheck = False
# cython: wraparound = False
# cython: nonecheck = False


cpdef long long fib(int x):
    """returns the xth fibonnaci number

    :x: TODO
    :returns: TODO

    """
    if x < 2:
        return 1
    cdef long long curr = 1   #chance of overflow so defined as long long
    cdef long long prev = 1
    cdef int i

    for i in range(x - 1):
        curr,prev = curr + prev,curr

    return curr
#for i in range(7):
    #print(fib(i))
```

Listing 2: setup-fib.py

```
from distutils.core import setup
from Cython.Build import cythonize

setup(name = "fib",ext_modules=cythonize("fib.pyx"))
```

following command is used (can be used in jup.dev.iitm.ac.in) for generating the c/cpp and the extention module

```
$ python3 setup_fib.py build_ext -if
```

comparing the time of the optimized and nonoptimzed version of the fibonacci function we see nearly 3000 times speed up (of-course this depends on the input given while timing the function).

Listing 3: ipython testing session

```
# coding: utf-8
from fib import fib
get_ipython().run_line_magic('timeit', 'fib(100000)')
get_ipython().run_line_magic('run', 'fib.py')
get_ipython().run_line_magic('timeit', 'fib_nopt(100000)')
```

```
$ ipython ipython_testfib_session.py
70.2 ţs ś 142 ns per loop (mean ś std. dev. of 7 runs, 10,000 loops each)
197 ms ś 462 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

1.2 Motivation to use cython

if we get similar speedups then, a function that takes 8 hrs to complete some task can only take 9.6 seconds by using cython, this is a huge gain for the little changes we do in our code.

2 Speeding up the equation solver

We should endsure that the nested loops run in the most optimised way as these are the bottelnecks for our programs.

the first place where there are nested loops is where we perform partial pivoting so here we optimise by introducing a function that calculates the norm to compare norm of 2 complex numbers. So now the underlying arithmatic takes place in cpp so it is faster.

We can use Cython's typed memory views to access the NumPy arrays more efficiently. Memory views provide direct access to the underlying memory of the arrays, which can eliminate the need for extra array indexing and copying.

We store the a and b in separate memory views, which can be accessed more efficiently than the original NumPy arrays. We also use local variables to store the real and imaginary parts of temporary values, such as temp, to avoid repeated indexing.

On top of the above optimisations we use some compiler flags to optimise the binaries produces such as '-O3' flag The -O3 flag enables a range of optimizations including inlining, loop unrolling, function outlining, and more. However, it can also make the compilation process slower and may result in larger binaries.

2.1 Code

here is the setup file which will generate the extention module(.so file on mac and linux)

Listing 4: setup-linal-solver.py

```
from distutils.core import setup, Extension
from Cython.Build import cythonize
import numpy

ld = Extension(name="linal_solver",
               sources=["linal_solver.pyx", "helper.cpp"])

setup(ext_modules=cythonize([ld]),include_dirs=[numpy.get_include()])
```

cimport numpy gives access to C functions or attributes or sub-modules under numpy
import numpy gives access to Python functions or attributes or sub-modules under numpy.

- boundscheck = False: This flag tells the Cython compiler to disable bounds checking on array and indexing operations.
- wraparound = False: This flag tells the Cython compiler to assume that array indexing will not wrap around if it exceeds the array bounds.
- nonecheck = False: This flag tells the Cython compiler to disable checks for None values
- cdivision compiler directive in Cython can improve the performance of code that involves division operations by replacing them with faster C division operations.

We are also using the language = c++ directive to tell the Cython compiler to generate C++ code instead of C code.

here is the optimisation of the matrix solver code

Listing 5: linal-solver.pyx

```
#!python
#cython: language_level=3
#cython:profile=True


# distutils: language = c++
# distutils: extra_compile_args = -Wall -O3 -ffast-math -ffinite-math-only -funsafe-math-opti




cimport numpy as np
import numpy as np
cimport cython

cdef extern from "helper.h":
    float norm2(float a,float b)

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
@cython.cdivision(True)
cpdef np.ndarray[np.complex128_t, ndim=1] my_linal_solver(np.ndarray[np.complex128_t, ndim=2]
    # Define typed memoryviews for a and b
    cdef np.complex128_t[:, :] virtual_a = a
    cdef np.complex128_t[:] virtual_b = b

    # Define loop variables
    cdef int i, j, k, N = len(a)
    cdef int index
    cdef float temp1_real, temp2_real, temp2_imag, temp1_imag
    cdef np.complex128_t temp

    for i in range(N):
        index = i  # max_value index
        for j in range(i, N):
            temp1_real = float(virtual_a[j, i].real)
            temp2_real = float(virtual_a[index, i].real)
            temp2_imag = float(virtual_a[index, i].imag)
            temp1_imag = float(virtual_a[j, i].imag)
            if norm2(temp1_real, temp1_imag) > norm2(temp2_real, temp2_imag):
                index = j

        # swapping
        virtual_a[i, index] = virtual_a[index, i]
        virtual_b[i],virtual_b[index] = virtual_b[index],virtual_b[i]

        # if the maximum element is also zero then the matrix is singular
        temp1_real = float(virtual_a[i, i].real)
        temp1_imag = float(virtual_a[i, i].imag)
        if norm2(temp1_real, temp1_imag)== 0:
            print("Exception occured : Matrix is singular")
            return np.array([-1])

        # taking norm now
        for j in range(i + 1, N):
            virtual_a[i, j] /= virtual_a[i, i]
        virtual_b[i] /= virtual_a[i, i]
        virtual_a[i, i] = 1

        # subtracting a[i] row from a[j] row with suitable multiplying factor
        for j in range(i + 1, N):
            temp = (virtual_a[j, i] / virtual_a[i, i])
            for k in range(i + 1, N):
                virtual_a[j, k] -= (temp) * virtual_a[i, k]
            virtual_b[j] -= (temp) * virtual_b[i]
            virtual_a[j, i] = 0

    # now we have the upper triangular matrix so we reverse our process from bottom to top, m
    for i in reversed(range(0, N)):
        for j in range(0, i):
            virtual_b[j] -= virtual_a[j, i] * virtual_b[i]
            virtual_a[j, i] = 0
    return np.array(virtual_b)
```

2.2 Testing

Listing 6: ipython testing session

```
# coding: utf-8
get_ipython().system('python3 setup_linal_solver.py build_ext -if')
from linal_solver import my_linal_solver
import numpy as np
A = np.random.randint(1,100,size=(100,100))
B = np.random.randint(1,100,size=(100))
A = np.array(A,dtype = complex)
B = np.array(B,dtype = complex)
get_ipython().run_line_magic('timeit', 'my_linal_solver(A,B)')
get_ipython().run_line_magic('run','linal_solver.py')
get_ipython().run_line_magic('timeit', 'my_linal_solver_nopt(A,B)')
```

the result for a 100 x 100 matrix :

```
$ python3 setup_linal_solver.py build_ext -if
$ ipython ipython_test_linal_session.py
  667 ţs ś 965 ns per loop (mean ś std. dev. of 7 runs, 1,000 loops each)
  150 ms ś 201 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

we see roughly a speed up of **200** times.

3 Learning outcomes

- Acquired knowledge of the standalone Cython package and its capabilities
- Learned how to write and use setup files for creating extension modules
- Gained experience in running and testing scripts in iPython
- Learned about the different compiler optimizers available for Cythonu and their effects on performance
- Became familiar with the Just-In-Time compilation provided by Numba
- Recognized the importance of performance optimization in scientific computations, and the impact it can have on the efficiency and accuracy of computational methods.
- I got a slight touch on SWIG, which provides an interface for creating bindings between C/C++ code and other programming languages, making it possible to use existing C/C++ code in Python.

4 Conclusion

this assignment taught how to use Cython to improve the performance of Python code, By leveraging static typing and generating C/CPP code, Cython can significantly speed up the execution of computationally intensive Python programs. However, it is important to keep in mind that Cython does have some limitations, such as requiring additional effort to write and maintain code

SWIG can be used to integrate C/C++ code with Python, while Numba provides just-in-time (JIT) compilation of Python code. These tools may be useful in certain scenarios, but they each have their own strengths and limitations.

Finally, it is worth emphasizing the importance of performance optimization in Python for scientific computations. As datasets grow the runtime starts to increase, the ability to efficiently process and analyze data becomes increasingly crucial. By understanding and utilizing tools like Cython, Python developers can achieve the performance gains necessary for scientific computing.