PHYSM0032 Advanced Computational Physics
Lecture 11

Dr. Simon Hanna

November 16, 2021

14 Accelerating Python with Numba

14.1 Introduction

- We have seen that Cython can achieve huge speed-ups when applied to Python scripts, on par with the speeds available from native C/C++ code.
- The main disadvantage of Cython is that, to get the most from it requires a good understanding of C/C++. Having that level of knowledge begs the question "why not simply write in C/C++?"
- ***Numba*** provides an alternative compilation option that requires very little intervention or adaptation of the Python script.
- Further details are available from the Numba website:

https://numba.pydata.org/numba-doc/dev/user/5minguide.html

14.2 Modus Operandi

- Numba is a system for "just in time" compilation of Python script into a binary executable.
- Numba operates via simple code "decorators" which are used to flag which functions to compile.
- As the name suggests, compilation occurs when needed – ***just in time*** – which implies a small compilation overhead, which can dominate short programs although some cacheing is used.
- Compatible with Windows, MacOS, Linux, Intel x86 and ARM (Apple M1); also works with nVidia CUDA and AMD ROCm (AMD's open software platform for HPC).
- Easily installed:
  - conda install numba
  - pip install numba

14.3 Example of use – PiCalc program

```
import sys

import time
from numpy import sqrt

def main( nmax ):
    pibyfour = 0.0
    dx = 1.0 / nmax
    for i in range(nmax):
        pibyfour += sqrt(1-pow((i*dx),2))
    pi = 4.0 * pibyfour * dx
    return pi

if __name__ == '__main__':
    if int(len(sys.argv)) == 2:
        initial = time.time()
        pi = main(int(sys.argv[1]))
        final = time.time()
        print("Pi = {:18.16f}".format(pi))
        print("Elapsed time: {:8.6f} s".format(
            final-initial))
    else:
        print("Usage: python {} <ITERATIONS>".
            format(sys.argv[0]))
```

```
import sys
from numba import jit
import time
from numpy import sqrt
@jit(nopython=True)
def main( nmax ):
    pibyfour = 0.0
    dx = 1.0 / nmax
    for i in range(nmax):
        pibyfour += sqrt(1-pow((i*dx),2))
    pi = 4.0 * pibyfour * dx
    return pi

if __name__ == '__main__':
    if int(len(sys.argv)) == 2:
        initial = time.time()
        pi = main(int(sys.argv[1]))
        final = time.time()
        print("Pi = {:18.16f}".format(pi))
        print("Elapsed time: {:8.6f} s".format(
            final-initial))
    else:
        print("Usage: python {} <ITERATIONS>".
            format(sys.argv[0]))
```

- `@jit` is the just-in-time decorator.
- `nopython=True` requires no reliance on Python constructions i.e. fully compiled code – this is the fastest method.
- If compilation fails with `nopython=True`, retry with it set false. This will be slower but might work.
- `@jit(nopython=True)` can be abbreviated to `@njit`
- Typical timings (10^9 terms):
  - Original python version: 145 s
  - Numba version: 1.38 s
  - Speed up x 105
  - Cythonized version: 1.21 s
- Very effective speed-ups possible with minimal effort.

14.4 Example of use – PiCalc program – parallel version

```
import sys

import time
from numpy import sqrt

def main( nmax ):
    pibyfour = 0.0
    dx = 1.0 / nmax
    for i in range(nmax):
        pibyfour += sqrt(1-pow((i*dx),2))
    pi = 4.0 * pibyfour * dx
    return pi

if __name__ == '__main__':
    if int(len(sys.argv)) == 2:
        initial = time.time()
        pi = main(int(sys.argv[1]))
        final = time.time()
        print("Pi = {:18.16f}".format(pi))
        print("Elapsed time: {:8.6f} s".format(
            final-initial))
    else:
        print("Usage: python {} <ITERATIONS>".
            format(sys.argv[0]))
```

```
import sys
from numba import jit,prange
import time
from numpy import sqrt
@jit(nopython=True,parallel=True)
def main( nmax ):
    pibyfour = 0.0
    dx = 1.0 / nmax
    for i in prange(nmax):
        pibyfour += sqrt(1-pow((i*dx),2))
    pi = 4.0 * pibyfour * dx
    return pi

if __name__ == '__main__':
    if int(len(sys.argv)) == 2:
        initial = time.time()
        pi = main(int(sys.argv[1]))
        final = time.time()
        print("Pi = {:18.16f}".format(pi))
        print("Elapsed time: {:8.6f} s".format(
            final-initial))
    else:
        print("Usage: python {} <ITERATIONS>".
            format(sys.argv[0]))
```

- `parallel=True` invokes parallel execution.
- Note use of `numba.prange` for the parallel loop. No other code change required.
- Typical timings (10^9 terms):
  - Original python version: 145 s
  - Numba version non-parallel: 1.38 s
  - Numba version parallel: 0.46 s (on 10 core machine, 1st run was 0.60 s, before cacheing)
  - Speed up x 315
  - Cythonized version: 0.13 s (10 cores)
- Parallel Numba gives an easy win over non-parallel code – Cython generally performs better overall, but for greater effort from the programmer.

14.5 Caveats

Although Numba sometimes appears miraculous, there are a number of points to bear in mind:

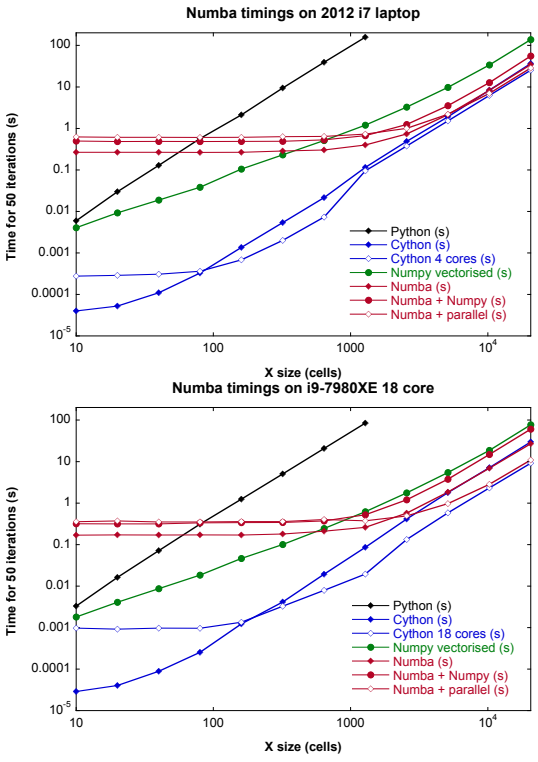
- Generally only individual functions are compiled;
- Timings should be performed after compilation has occurred to avoid overestimating the time needed;
- Not all python will be compilable – as with Cython, there can be unexpected issues, so it is best to limit compilation to individual functions;
- Numpy arrays work well with Numba, but vectorisation does not work particularly effectively (see next example);
- It can be hard to predict which features will lead to good speed-ups;
- Additional control is possible e.g. with typing of functions and arguments – see documentation.

14.6 Example of use – heat equation – various versions compared

```
import numba
@numba.jit(nopython=True)
def update(nx, ny, u1, u2):
    temp = 1-2*(Cx+Cy)
    for ix in range(1,nx-1):
        for iy in range(1,ny-1):
            u2[ix,iy] = u1[ix,iy]*temp + Cx * (u1
                [ix+1,iy] + u1[ix-1,iy]) + Cy * (
                u1[ix,iy+1] + u1[ix,iy-1])
```

```
import numba
@numba.njit(parallel=True)
def update(nx, ny, u1, u2):
    temp = 1-2*(Cx+Cy)
    for ix in numba.prange(1,nx-1):
        for iy in range(1,ny-1):
            u2[ix,iy] = u1[ix,iy]*temp + Cx * (u1
                [ix+1,iy] + u1[ix-1,iy]) + Cy * (
                u1[ix,iy+1] + u1[ix,iy-1])
```

- Only the main cell update function needs to be compiled;
- Benefits of cacheing not seen for smaller models;
- Cython generally outperforms Numba, but not by much;
- Numba performs poorly on small models (see next slide);
- Timings vary wildly with operating system and compiler, for a given architecture.



14.7 Numba with CUDA

- Numba has taken over from PyCuda as the preferred way to access GPUs from Python (support for PyCuda has waned in recent years);
- Requirements:
  - CUDA graphics card programming model 3 or later (e.g. less than about 10 years old)
  - nVidia CUDA toolkit and drivers for your chosen system (i.e. not a Mac)
  - Python cudatoolkit package (install with conda)
  - The above all need to be compatible versions.
- Python needs to be written expressly for CUDA, using the same kernel approach – this is not a transparent accelerator.

14.8 CUDA example - matrix multiplication

```
from numba import cuda
import numpy as np
import math

@cuda.jit
def matmul(A,B,C):
    i,j = cuda.grid(2)
    if i < C.shape[0] and j<C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i,k]*B[k,j]
        C[i,j] = tmp
N=16
A = np.random.random((N,N))
B = np.random.random((N,N))
C = np.zeros((N,N))

threadsperblock = (N,N)
blockspergrid_x = math.ceil(C.shape[0] /
    threadsperblock[0])
blockspergrid_y = math.ceil(C.shape[1] /
    threadsperblock[1])
blockspergrid = (blockspergrid_x,blockspergrid_y)
matmul[blockspergrid,threadsperblock](A,B,C)
```

- Note use of `@cuda.jit` decorator
- `matmul` is the CUDA kernel; it takes values of `i,j` from the grid defined by `blockspergrid` and `threadsperblock`
- Note how `matmul` is called: the square brackets take the place of the `<<>>` used when calling CUDA from C;
- Running this script through Python will invoke the nVidia nvcc compiler as required for compiling CUDA code.