Distributed Computing and Introduction to High Performance Computing

Imad Kissami[1], Nouredine Ouhaddou[1]
[1]Mohammed VI Polytechnic University, Benguerir, Morocco

Outline of this lecture

- Accelerate a Python code
  - Using Numpy
  - Using Cython
  - Using Numba
  - Using Pyccel
- Some Benchmarks

Accelerate a Python code

What is Numpy

- The NumPy library is the core library for scientific computing in Python.
- It provides a high-performance multidimensional array object, and tools for working with these arrays.
- The NumPy package integrates C, C++, and Fortran codes in Python. These programming languages have very little execution time compared to Python.
- The NumPy package breaks down a task into multiple fragments and then processes all the fragments parallelly.

Accelerate a Python code

Numpy vs python benchmarks

```python
import numpy, time

size = 1000000

print("Concatenation: ")
list1 = [i for i in range(size)]
list2 = [i for i in range(size)]

array1 = numpy.arange(size)
array2 = numpy.arange(size)

# List
initialTime = time.time()
list1 = list1 + list2
# calculating execution time
print("Time taken by Lists :", (time.time() - initialTime), "seconds")

# Numpy array
initialTime = time.time()
array = numpy.concatenate((array1, array2), axis = 0)
# calculating execution time
print("Time taken by NumPy Arrays :", (time.time() - initialTime), "seconds")
```

```
Concatenation:
Time taken by Lists : 0.021048307418823242 seconds
Time taken by NumPy Arrays : 0.009451150894165039 seconds
```

Accelerate a Python code

Numpy vs python benchmarks

```python
import numpy, time

...

dot = 0
print("\nDot Product:")

# List
initialTime = time.time()
for a, b in zip(list1, list2):
    dot = dot + (a * b)
print("Time taken by Lists :", (time.time() - initialTime), "seconds")

# Numpy array
initialTime = time.time()
array = numpy.dot(array1, array2)
print("Time taken by NumPy Arrays :", (time.time() - initialTime), "seconds")
```

```
Dot Product:
Time taken by Lists : 0.13322114944458008 seconds
Time taken by NumPy Arrays : 0.0025365352630615234 seconds
```

Accelerate a Python code

Numpy vs python benchmarks

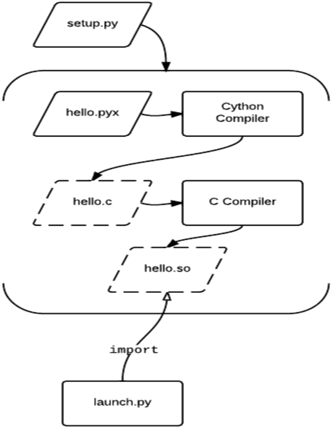
```
import numpy, time

...

print("\nDeletion: ")

# List
initialTime = time.time()
del(list1)
print("Time taken by Lists :", (time.time() - initialTime), "seconds")

# NumPy array
initialTime = time.time()
del(array1)
print("Time taken by NumPy Arrays :", (time.time() - initialTime), "seconds")
```

```
Deletion:
Time taken by Lists : 0.016112804412841797 seconds
Time taken by NumPy Arrays : 9.512901306152344e-05 seconds
```

Accelerate a Python code

What is Cython

- Cython is an optimizing static compiler for both the Python programming language and the extended Cython programming language (based on Pyrex).
- Cython gives you the combined power of Python



Accelerate a Python code

Cython example

- Python

```
def mandelbrot(m, size, iterations):
    for i in range(size):
        for j in range(size):
            c = -2 + 3./size*j + 1j*(1.5-3./size*i)
            z = 0
            for n in range(iterations):
                if np.abs(z) ¡= 10:
                    z = z*z + c; m[i, j] = n
                else:
                    break
```

- Cython

```
def mandelbrot_cython(int[:,::1] m, int size, int iterations):
    cdef int i, j, n
    cdef complex z, c
    for i in range(size):
        for j in range(size):
            c = -2 + 3./size*j + 1j*(1.5-3./size*i)
            z = 0
            for n in range(iterations):
                if z.real**2 + z.imag**2 ¡= 100:
                    z = z*z + c; m[i, j] = n
                else:
                    break
```

Accelerate a Python code

Cython example

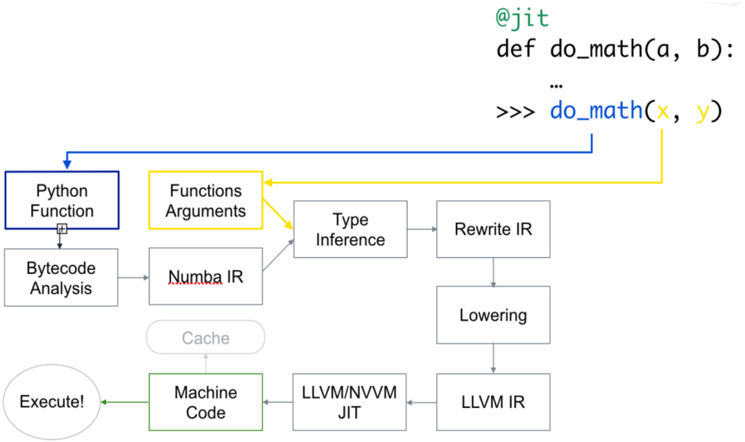
- Execution time

```
%%timeit -n1 -r1
m = np.zeros(s, dtype=np.int32)
mandelbrot(m, size, iterations)
>> 12.2 s +/- 0 ns per loop (mean +/- std. dev. of 1 run, 1 loop each)


%%timeit -n1 -r1
m = np.zeros(s, dtype=np.int32)
mandelbrot_cython(m, size, iterations)
>> 29.8 ms +/- 0 ns per loop (mean +/- std. dev. of 1 run, 1 loop each)
```

Accelerate a Python code

Numba

- Open source Just-In-Time compiler for python functions.
- Uses the LLVM library as the compiler backend.



Accelerate a Python code

Numba example

- Python

```python
import numpy as np

def do_sum():
    acc = 0.
    for i in range(10000000) :
        acc += np.sqrt(i)
    return acc
```

- Numba

```python
from numba import njit

@njit
def do_sum_numba():
    acc = 0.
    for i in range(10000000) :
        acc += np.sqrt(i)
    return acc
```

- Execution time:

```
Time for Pure Python Function: 7.724030017852783
Time for Numba Function: 0.015453100204467773
```

Accelerate a Python code

Pyccel

- Pyccel is a static compiler for Python 3, using Fortran or C as a backend language, with a focus on high-performance computing (HPC) applications.
- Public repository is now hosted on GitHub, freely available for download.
- Python function:

```
import numpy as np

    def do_sum_pyccel():
    acc = 0.
    for i in range(10000000) :
        acc += np.sqrt(i)
    return acc
```

- Compilation using fortran:

```
pyccel --language=fortran pyccel_example.py
```

Accelerate a Python code

Pyccel: Generated fortran function

```fortran
module pyccel_example

use, intrinsic :: ISO_C_Binding, only : i64 => C_INT64_T , f64 => C_DOUBLE
    implicit none

    contains
    !........................................
    function do_sum_pyccel() result(acc)

        implicit none
        real(f64) :: acc
        integer(i64) :: i

        acc = 0.0_f64
        do i = 0_i64, 9999999_i64, 1_i64
            acc = acc + sqrt(Real(i, f64))
        end do
        return

    end function do_sum_pyccel
!........................................

end module pyccel_example
```

- Execution time:

```
Time for Pure Python Function: 7.400242328643799
Time for Pyccel Function: 0.01545262336730957
```

Accelerate a Python code

Pyccel: Generated c function

```c
#ifndef PYCCEL_EXAMPLE_H
#define PYCCEL_EXAMPLE_H

#include <stdlib.h>

double do_sum_pyccel(void);
#endif // PYCCEL_EXAMPLE_H
```

```c
#include "pyccel_example.h"
#include <stdlib.h>
#include <math.h>
#include <stdint.h>

/*........................................*/
double do_sum_pyccel(void)
{
    int64_t i;
    double acc;
    acc = 0.0;
    for (i = 0; i < 10000000; i += 1)
    {
        acc += sqrt((double)(i));
    }
    return acc;
}
/*........................................*/
```

Some Benchmarks

Rosen-Der

Tool	Python	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs) Speedup	229.85 –	2.06 $\times$ 111.43	4.73 $\times$ 48.57	2.07 $\times$ 110.98	**0.98** $\times$ **232.94**	**0.64** $\times$ **353.94**

Black-Scholes

Tool	Python	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs) Speedup	180.44 –	309.67 $\times$ 0.58	3.0 $\times$ 60.06	1.1 $\times$ 163.8	**1.04** $\times$ **172.35**	$6.56\ 10^{-2}$ $\times$ **2748.71**

Laplace

Tool	Python	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs) Speedup	57.71 –	7.98 $\times$ 7.22	$6.46\ 10^{-2}$ $\times$ 892.02	$6.28\ 10^{-2}$ $\times$ 918.56	$8.02\ 10^{-2}$ $\times$ **719.32**	$2.81\ 10^{-2}$ $\times$ **2048.65**

Growcut

Tool	Python	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (s) Speedup	54.39 –	$1.02\ 10^{-1}$ $\times$ 532.37	$4.67\ 10^{-1}$ $\times$ 116.45	$8.57\ 10^{-2}$ $\times$ 634.32	$6.27\ 10^{-2}$ $\times$ **866.49**	$6.54\ 10^{-2}$ $\times$ **831.7**