Usando numba onde python é lento

...

Edison Gustavo Muenz

 @edisongustavo

 edisongustavo@gmail.com

Quem sou eu

- Bacharel em Ciências da Computação pela UFSC
- Programador python há 4 anos
- Atualmente trabalho com python “numérico” / “científico” na ESSS

Agenda

- Mostrar que computação com python puro é lento
- Mostrar a biblioteca Numba para resolver este problema
- Features da biblioteca
  - Computação genérica
  - Numpy
  - GPU

Exemplo: bubblesort

```
def bubblesort(arr):
    for i in range(len(arr)):
        for j in range(i, len(arr)):
            if arr[i] > arr[j]:
                arr[i], arr[j] = arr[j], arr[i]
    return arr
```

Performance bubblesort

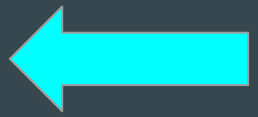
Medição (com jupyter notebook):

```
arr1 = np.random.rand(1000)

%time bubblesort(arr1)
```

- 1000 elementos: 200ms
- 10000 elementos: 20000ms

Aplicando numba

```
import numba
@numba.jit(nopython=True)
def bubblesort(arr):
    for i in range(len(arr)):
        for j in range(i, len(arr)):
            if arr[i] > arr[j]:
                arr[i], arr[j] = arr[j], arr[i]
```



Código novo

Performance bubblesort com numba

Medição (com jupyter notebook):

```
arr1 = np.random.rand(1000)

bubblesort_numba(arr1[0:2]) # numba compila o código para o tipo (np.float64):

%time bubblesort(arr1)
```

Com numba:

- 1000 elementos: 2.32ms
- 10000 elementos: 200ms

Com Python puro:

- 1000 elementos: 200ms
- 10000 elementos: 20000ms

Performance bubblesort com numba

- Numba acelerou o código em ~ 100x
- Utilizando apenas um @decorator

O que é numba

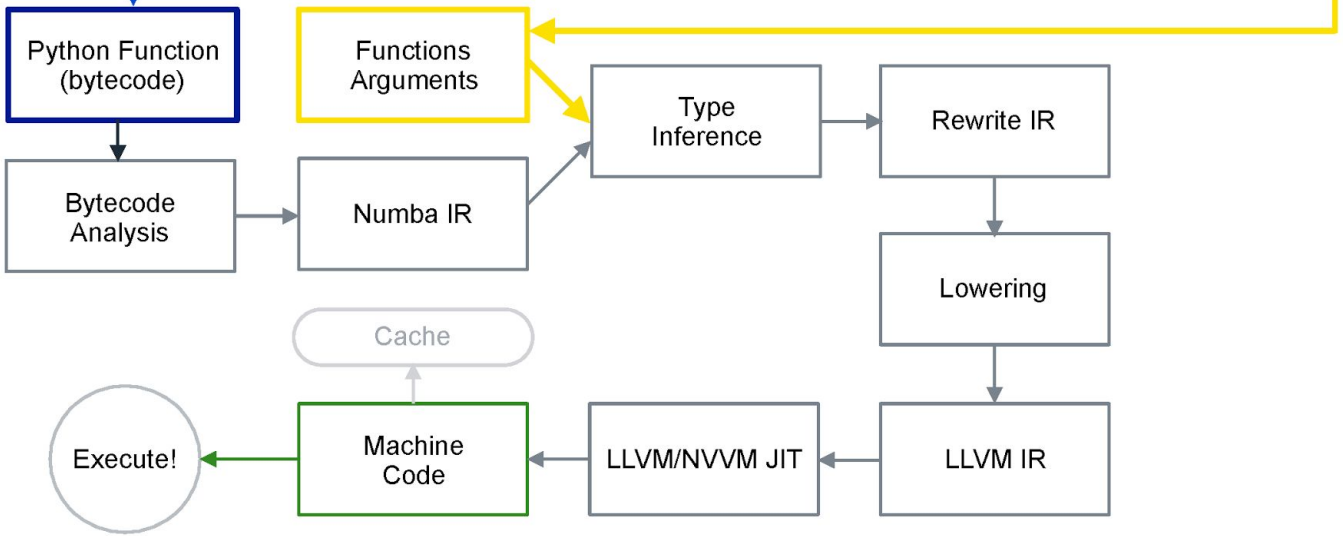
Numba é um compilador JIT (just-in-time) para funções (e classes) python

Simplificando, como funciona?

1. analisa o bytecode python e converte para LLVM
2. LLVM gera código nativo otimizado

How Does Numba Work?

```
@jit
def do_math(a, b):
    …
>>> do_math(x, y)
```



Vantagens e características

- Apenas uma biblioteca/módulo python
  - Simplifica o deploy
  - Sem passos intermediários no build (por exemplo: cython, c++, etc.)
- Muito simples para a maioria dos casos
  - apenas decorators: @jit e @vectorize
- Programar código rápido e eficiente em python “puro”
- Open source
  - Criada e mantida pela Continuum Analytics

Features python

- funções builtin: sorted(), len(), min(), max(), etc.
- tipos builtin: tuple, list
- classes (@jit_class)
- generators
- comprehensions
- context manager (palavra chave: with)
- exceções
  - try...except, try...finally
- print()

Integração com numpy

Por design, numba tem como objetivo uma boa integração com numpy.

Arrays da numpy possuem todas as informações necessárias para geração de código otimizado e especializado:

- Tipo dos dados do array
- Um array possui apenas dados com o mesmo tipo (homogêneo)

Então quando possível, utilizar arrays da numpy é o recomendado.

Integração com numpy

- Chamadas a funções da numpy:

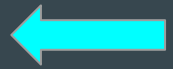
```
@numba.jit(nopython=True)
def call_numpy():
    a = np.array([1.0, 2.0, 3.0])
    b = np.array([1.0, 2.0, 3.0])
    return np.dot(a, b)

print(call_numpy())

$ 14.0
```

chamando numpy.dot()

Integração com numpy

- “slices” de arrays da numpy:

```
@numba.jit(nopython=True)
def slice_numpy(arr):
    half_index = int(len(arr) / 2)
    lower = arr[:half_index]
    upper = arr[half_index:]
    bubblesort(lower)
    bubblesort(upper)
```



slices

chamando outra função numba

Numpy universal functions (ufuncs)

São funções que atuam sobre um array ou escalares

Exemplo:

numpy.add()

- numpy.add(1, 2) -> 3
- numpy.add([1, 2], 3) -> [4, 5]
- numpy.add([1, 2], [3, 4]) -> [4, 6]

Criando ufuncs apenas com Numpy:

```
@np.vectorize
def my_sum(a, b):
   return a + b

print(my_sum(1, 2))
3
print(my_sum([1, 2], 3))
[4, 5]
print(my_sum([1, 2], [3, 4]))
[4, 6]
```

decorator da numpy

- Sem o decorator @np.vectorize ocorre:

  TypeError: can only concatenate list (not "int") to list

Ufuncs da numpy em python são lentas

Segundo a documentação:

"The vectorize function is provided primarily for convenience, not for performance. The implementation is essentially a for loop."

E é verdade:

```
a = np.arange(10000000)
%time my_sum(a, a)
CPU times: user 2.5 s, sys: 304 ms, total: 2.8 s Wall time: 2.8 s
```

@numba.vectorize

```
@np.vectorize
def my_sum(a, b):
   return a + b
```

decorator numba

a = np.arange(10000000)

%time my_sum(a, a)

CPU times: user 20 ms, sys: 4 ms, total: 24 ms Wall time: 22.4 ms

Numpy: 2800 ms

Numba: 22.4 ms

numba foi ~ 100x mais rápido

Multithreading

Multithreading

É possível liberar a GIL (Global Interpreter Lock) e obter paralelismo real:

```
@jit(nopython=True, nogil=True)
```

Exemplo:

```
@numba.jit(nopython=True, nogil=True)
def minha_funcao_liberando_gil():
    bubblesort(np.random.random(50000))
```

Multithreading - medindo

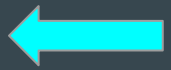
```
import time, threading
thread1 = threading.Thread(target=minha_funcao_liberando_gil)
thread2 = threading.Thread(target=minha_funcao_liberando_gil)
start = time.time()
thread1.start(); thread2.start()
thread1.join(); thread2.join()
end = time.time() - start
print(‘Tempo total: %s segundos' % (end * 1000))
```

Resultados:

nogil=False	8.2 segundos
nogil=True	4.2 segundos

Multithreading - @numba.vectorize

```
@numba.vectorize(["float32(float32, float32)"], target='cpu')
def slow_vectorize_serial(a, b):
    z = 0
    for i in range(200):
        z += np.exp(np.log(a)) - np.exp(np.log(b))
    return z + a - b
```



operações pesadas
muitas vezes

```
@numba.vectorize(["float32(float32, float32)"], target='parallel')
def slow_vectorize_serial(a, b):
    z = 0
    for i in range(200):
        z += np.exp(np.log(a)) - np.exp(np.log(b))
    return z + a - b
```

operações pesadas
muitas vezes

Multithreading - @numba.vectorize

a = np.arange(10000000, dtype=np.float32)

%time slow_vectorize_parallel(a, a)

CPU times: user 2.65 s, sys: 16 ms, total: 2.66 s Wall time: 0.411 s

%time slow_vectorize_serial(a, a)

CPU times: user 2.36 s, sys: 4 ms, total: 2.36 s Wall time: 2.36 s

CUDA

Numba também oferece suporte a GPUs:

Kernel:

```
from numba import cuda
@cuda.jit
def increment_by_one(an_array):
    tx = cuda.threadIdx.x
    ty = cuda.blockIdx.x
    bw = cuda.blockDim.x
    pos = tx + ty * bw
    if pos < an_array.size:
        an_array[pos] += 1
```

Rodar Kernel:

```
an_array = np.array([1, 2, 3, 4])
threadsperblock = 32
blockspergrid = (an_array.size + (threadsperblock - 1))
increment_by_one[blockspergrid, threadsperblock](an_array)
print(an_array)
# [2 3 4 5]
```

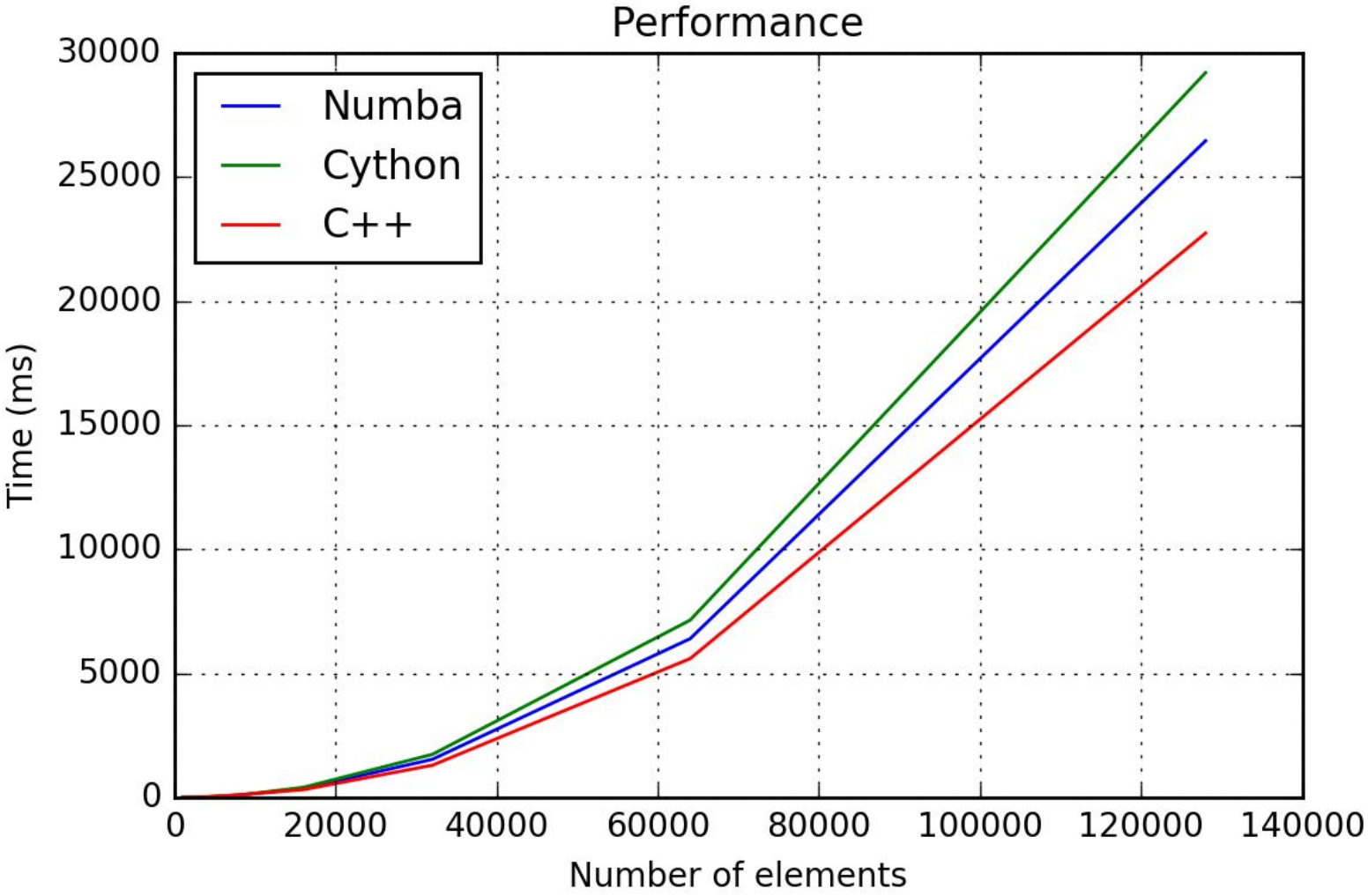
Outras alternativas

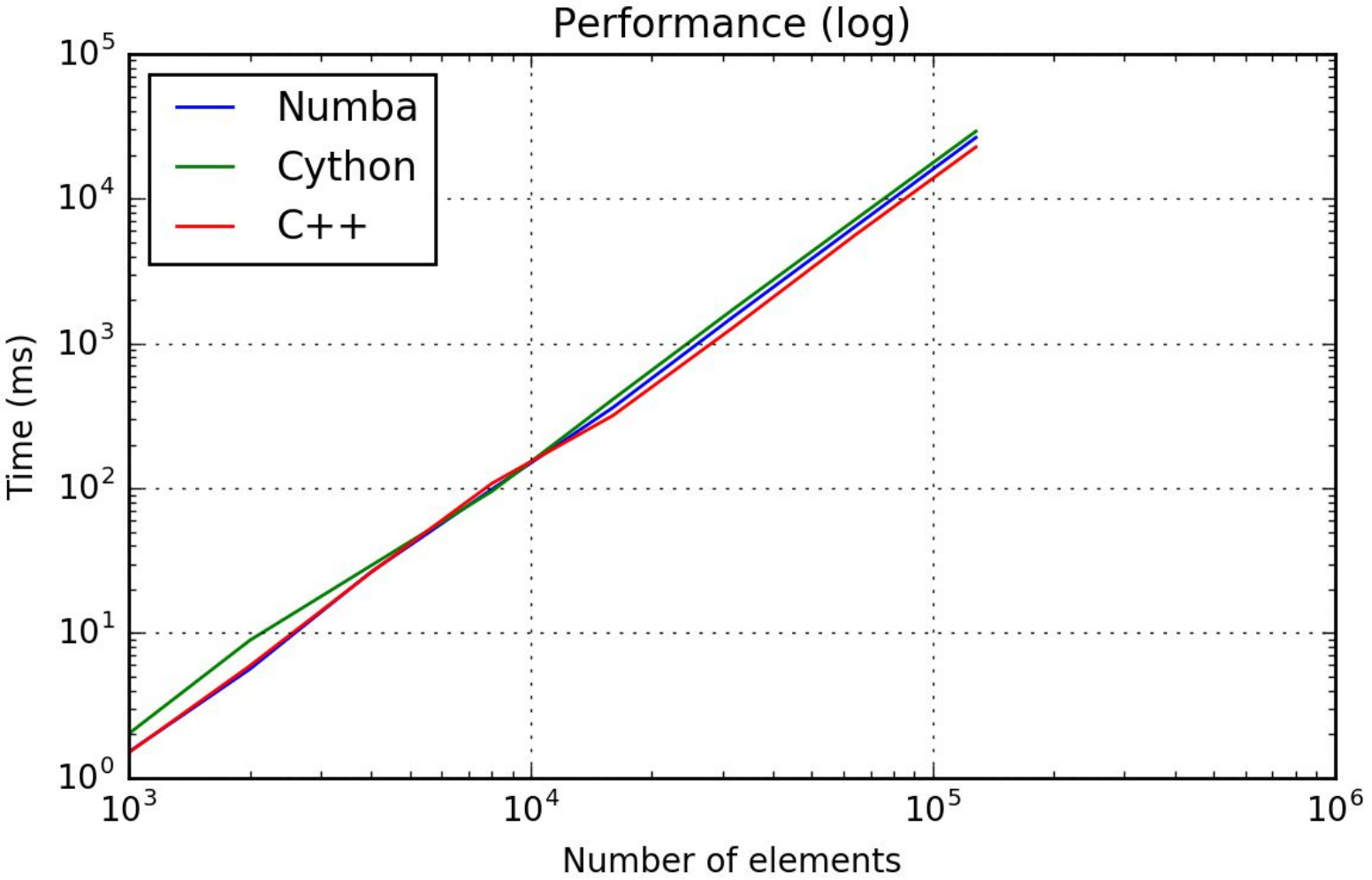
Cython

```
cimport numpy as np
cimport cython
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef bubblesort_cython(double[:] np_ary):
    cdef int count, i, j
    count = np_ary.shape[0]
    for i in range(count):
        for j in range(1, count):
            if np_ary[j] < np_ary[j-1]:
                np_ary[j-1], np_ary[j] = np_ary[j], np_ary[j-1]
```

C++

```
void bubblesort(std::vector<double>& v) {
  for (size_t i = 0; i < v.size(); ++i) {
    for (size_t j = i+1; j < v.size(); ++j) {
      if (v[i] > v[j]) {
        std::swap(v[i], v[j]);
      }
    }
  }
}
```


Performance
Numba
Cython
C++
Time (ms)
Number of elements
0
5000
10000
15000
20000
25000
30000
0
20000
40000
60000
80000
100000
120000
140000


Performance (log)
Numba
Cython
C++
Time (ms)
Number of elements
10^0
10^1
10^2
10^3
10^4
10^5
10^3
10^4
10^5
10^6

Performances

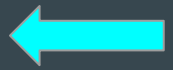
elementos	Numba	Cython	C++
1000	1.76	2.27	1.50
2000	6.79	8.35	5.95
4000	25.91	30.46	26.19
8000	102.34	97.88	108.07
16000	380.88	413.54	318.01
32000	1610.48	1750.63	1300.90
64000	6357.28	7175.45	5595.70
128000	26398.30	29709.92	22736.75

Problemas e “gotchas”

Sem “bounds checking”

- Arrays e listas são acessados sem “bounds-checking”, ou seja, “undefined behaviour”:

```
@numba.jit(nopython=True)
def out_of_bounds_access(arr):
    l = len(arr)
    return arr[l+1]

# Imprime um valor diferente a cada chamada
print(out_of_bounds_access([1, 2, 3]))
print(out_of_bounds_access(np.array([1, 2, 3])))
```



acesso fora do array

Limitações - Features

- Não permite recursão quando nopython=True
- Uma boa parte da biblioteca “standard” python não está disponível
  - Motivo: No código do numba é preciso mapear toda a standard library.
  - Boa notícia: os módulos numéricos estão mapeados (math/cmath, random, array, etc.)
  - Foco atual da biblioteca é com processamento de arrays (código numérico)

Erros crípticos

Algumas vezes podem acontecer erros “indecifráveis” na hora de **compilar um método**:

```
@numba.jit(nopython=True)
def fibonacci_rec(n):
    if n in (1, 2):
        return 1
    return fibonacci_rec(n-1) + fibonacci(n-2)
```

Exceção:

```
E           numba.errors.TypingError: Failed at nopython (nopython frontend)

E           Internal error at <numba.typeinfer.CallConstraint object at 0x7f69bcc78198>:
```

Problema: utilizei recursão

Erros crípticos

```python
@numba.jit(nopython=True)
def sum_list_of_lists(list_of_lists):
    ret = []
    for l in list_of_lists:
        s = 0
        for i in l:
            s += i
        ret.append(s)
    return ret
```

Exceção

```
NotImplementedError: reflected list(reflected list(int64)): unsupported nested memory-managed
object
```

Problema: listas de listas

Legal, quero usar!

conda install numba

Legal, quero usar!

- Disponível para Windows, Linux e OS X
- Versão mais atual em 17/05/2016: 0.25
- Desenvolvimento muito ativo
  - Lista de discussão ativa e amigável a novos desenvolvedores
  - Commits diários no repositório: https://github.com/numba/numba

Obrigado

http://numba.pydata.org/


@edisongustavo


edisongustavo@gmail.com