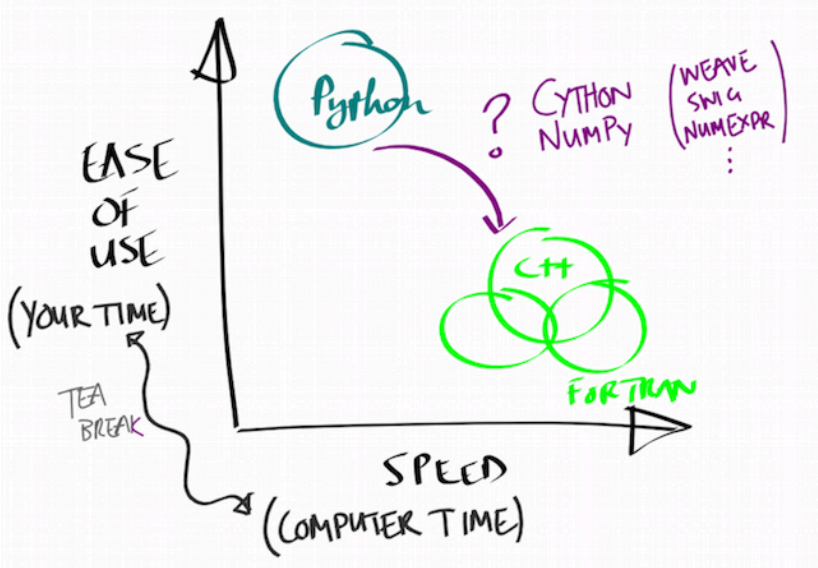
Python Numba for scientific code

Zbigniew Jędrzejewski-Szmek

zbyszek@in.waw.pl

Instytut im. M. Nęckiego, 25.11.2018

Programmer time vs. computer time



Previous approaches

a.k.a. the graveyard of technologies

1. Use compiled C or C++ or Fortran code with cpython
    - C extension
    - .so library + ctypes
    - Fortran + f2py
    - boost-python
    - SWIG
    - (Cython)

Previous approaches

a.k.a. the graveyard of technologies

1. Use compiled C or C++ or Fortran code with cpython
   - C extension
   - .so library + ctypes
   - Fortran + f2py
   - boost-python
   - SWIG
   - (Cython)
2. A different Python interpreter
   - jython
   - psyco
   - pypy

Previous approaches

a.k.a. the graveyard of technologies

1. Use compiled C or C++ or Fortran code with cpython
    - C extension
    - .so library + ctypes
    - Fortran + f2py
    - boost-python
    - SWIG
    - (Cython)
2. A different Python interpreter
    - jython
    - psyco
    - pypy
3. Numpy

Previous approaches

a.k.a. the graveyard of technologies

1. Use compiled C or C++ or Fortran code with cpython
   - C extension
   - .so library + ctypes
   - Fortran + f2py
   - boost-python
   - SWIG
   - (Cython)
2. A different Python interpreter
   - jython
   - psyco
   - pypy
3. Numpy
4. “jit” just the inner loop
   - weave (Python + annotations to C++, inline)
   - pyrex (Python + annotations to C, external)
   - cython
   - numexpr (“jitting” of basic numpy array operations)
   - numba

Special syntax

numexpr

```python
# numexpr_evaluate.py
import numpy as np, numexpr as ne
a = np.arange(1e6)
b = np.random.randint(10, size=(1_000_000,))
print(ne.evaluate('a*b-4.1*a > 2.5*b'))
```

Special syntax and type annotations

cython

```
# cython_integrate.pyx
def f(double x):
    y = (x*x*x - 3)*x
    return y
def integrate_f(double a, double b, int n):
    cdef:
        double dx = (b - a) / n
        double dx2 = dx / 2
        double s = f(a) * dx2
        int i = 0
    for i in range(1, n):
        s += f(a + i * dx) * dx
    s += f(b) * dx2
    return s
```

“jit”?

```
>>> import numba

>>> @numba.jit
... def f(x):
...     y = x*5 + x
...     return y
```

“jit”?

```
>>> import numba

>>> @numba.jit
... def f(x):
...     y = x*5 + x
...     return y

>>> f(1)
6
```

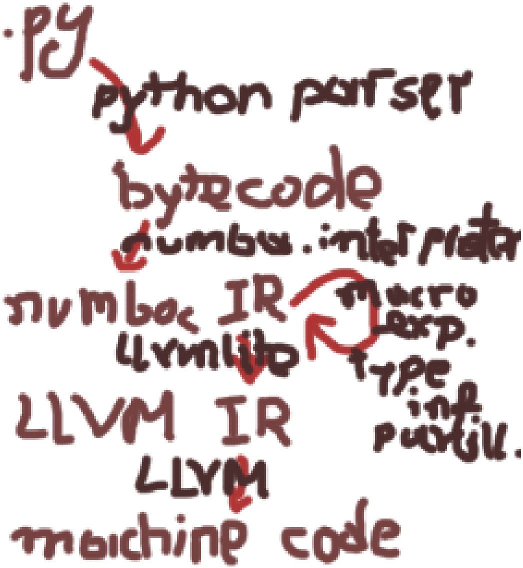
```
>>> import numpy as np
>>> x = np.eye(3)
>>> print('x:', x)
x: [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
>>> print('f(x):', f(x))
f(x): [[6. 0. 0.]
 [0. 6. 0.]
 [0. 0. 6.]]
```

```
>>> import numpy as np
>>> x = np.eye(3)
>>> print('x:', x)
x: [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
>>> print('f(x):', f(x))
f(x): [[6. 0. 0.]
 [0. 6. 0.]
 [0. 0. 6.]]

>>> f('abc')
'abcabcabcabcabcabc'
```

```
>>> f
CPUDispatcher(<function f at 0x...>)
>>> f.signatures
[(int64,),
 (array(float64, 2d, C),),
 (str,)]
>>> f.nopython_signatures
[(int64,) -> int64,
 (array(float64, 2d, C),) -> array(float64, 2d, C)]
```

Numba architecture



Some timings

```
def runningsum_loop(N):
    s = 0
    for i in range(N + 1):
        s += i
    return s
```

```
% timeit runningsum_loop(1_000_000)
54 ms ± 973 µs per loop (mean ± std. dev. of 7 runs,
                                              10 loops each)
```

```python
def runningsum_list(N):
    return sum([i for i in range(N + 1)])
```

```
%timeit runningsum_list(1_000_000)
52.7 ms ± 543 µs per loop (mean ± std. dev. of 7 runs,
                                        10 loops each)
```

```
def runningsum_list(N):
    return sum([i for i in range(N + 1)])
```

```
%timeit runningsum_list(1_000_000)
52.7 ms ± 543 µs per loop (mean ± std. dev. of 7 runs,
                                        10 loops each)
```

```
def runningsum_generator(N):
    return sum(i for i in range(N + 1))
```

```
%timeit runningsum_generator(1_000_000)
50.4 ms ± 1.32 ms per loop (mean ± std. dev. of 7 runs,
                                         10 loops each)
```

```python
def runningsum_numpy(N):
    return np.arange(N+1).sum()
```

```
%timeit runningsum_numpy(1_000_000)
1.41 ms ± 76.8 µs per loop (mean ± std. dev. of 7 runs,
                                        1000 loops each)
```

```python
import numba

@numba.jit
def runningsum_numba_loop(N):
    s = 0
    for i in range(N + 1):
        s += i
    return s
```

```python
import numba

@numba.jit
def runningsum_numba_loop(N):
    s = 0
    for i in range(N + 1):
        s += i
    return s
```

```
%timeit -r 1 -n 1 runningsum_numba_loop(1_000_000)
173 ms ± 0 ns per loop (mean ± std. dev. of 1 run,
                                          1 loop each)
```

```python
import numba

@numba.jit
def runningsum_numba_loop(N):
    s = 0
    for i in range(N + 1):
        s += i
    return s
```

```
%timeit -r 1 -n 1 runningsum_numba_loop(1_000_000)
173 ms ± 0 ns per loop (mean ± std. dev. of 1 run,
                                       1 loop each)

%timeit runningsum_numba_loop(1_000_000)
195 ns ± 5.48 ns per loop (mean ± std. dev. of 7 runs,
                                 10_000_000 loops each)
```

```c
long int sum(int N) {
        long int s = 0;
        for (int i=0; i <= N; i++)
                s += i;
        return s;
}
int main(int argc, char **argv) {
        long int s;
        for (int i = 0; i < 10000; i++)
                s = sum(1000000);
        printf("%ld\n", s);
        return 0;
}
```

`gcc -g -Wall` → 2.8 ms

`gcc -g -Wall -O3` → <1 µs (naive)

`gcc -g -Wall -O3` → 237 µs (external compilation unit)

Some timings — summary

	time / ms	
`runningsum_loop`	54	
`runningsum_list`	53	
`runningsum_generator`	50	
`runningsum_numpy`	1.41	
`runningsum_numba_loop`	173	(single iteration)
`runningsum_numba_loop`	0.000195	(repeated)
`runningsum_c`	2.8	`-O0`
`runningsum_c`	<0.001	`-O3`
`runningsum_c`	0.237	`-O3`, seperate compilation units

Other numba features

Automatic parallelization

```
def trig_ident_np(x):
    return (np.sin(x)**2 + np.cos(x)**2 +
            np.sin(x)**2 + np.cos(x)**2 +
            np.sin(x)**2 + np.cos(x)**2 +
            np.sin(x)**2 + np.cos(x)**2).sum()/4
```

Automatic parallelization

```
def trig_ident_np(x):
    return (np.sin(x)**2 + np.cos(x)**2 +
            np.sin(x)**2 + np.cos(x)**2 +
            np.sin(x)**2 + np.cos(x)**2 +
            np.sin(x)**2 + np.cos(x)**2).sum()/4

trig_ident_jit = numba.jit(trig_ident_np)
trig_ident_jitp = numba.jit(parallel=True)(trig_ident_np)
```

Automatic parallelization

```
def trig_ident_np(x):
    return (np.sin(x)**2 + np.cos(x)**2 +
            np.sin(x)**2 + np.cos(x)**2 +
            np.sin(x)**2 + np.cos(x)**2 +
            np.sin(x)**2 + np.cos(x)**2).sum()/4
```

```
trig_ident_jit = numba.jit(trig_ident_np)
trig_ident_jitp = numba.jit(parallel=True)(trig_ident_np)

x = np.random.randn(500, 50_000)
```

```
%timeit trig_ident_np(x)
4.52 s ± 160 ms per loop

%timeit trig_ident_jit(x)
788 ms ± 24 ms per loop

%timeit trig_ident_jitp(x)
290 ms ± 7.38 ms per loop
```

Hardware support

- vector instructions (when CPU supports SSE, AVX, or AVX-512)
- Nvidia CUDA backend

Evaluation of numba

- good: native syntax and seamless integration
- good: excellent speed (when it works)
- bad: requires the whole LLVM backend to be present
- bad: hard to debug
- bad: not "reproducible"

Where is this all going?

Is Python a statically typed language?

Is Python a statically typed language?

```python
# adder.py
def add(a:int, b:int = 1) -> int:
    return a + b

add(1, 2)
add(1.2, 2.2)
```

Is Python a statically typed language?

```python
# adder.py
def add(a:int, b:int = 1) -> int:
    return a + b

add(1, 2)
add(1.2, 2.2)
```

```
$ mypy adder.py
adder.py:6: error: Argument 1 to "add" has incompatible
                   type "float"; expected "int"
adder.py:6: error: Argument 2 to "add" has incompatible
                   type "float"; expected "int"
```

The future?

- Python continues to be used a glue language
- Python code is seamlessly compiled with various backends
- Type hints are used where automatic type inference is insufficient

The End

Inspecting numba outputs

```
print(runningsum_numba_loop.inspect_llvm()[(numba.int64,)])
```

```
; ModuleID = 'runningsum_numba_loop'
source_filename = "<string>"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@"_ZN08NumbaEnv8__main__25runningsum_numba_loop$242Ex" = common local_unnamed_addr global i8* null
@.const.runningsum_numba_loop = internal constant [22 x i8] c"runningsum_numba_loop\00"
@PyExc_RuntimeError = external global i8
@".const.missing Environment" = internal constant [20 x i8] c"missing Environment\00"

; Function Attrs: norecurse nounwind
define i32 @"_ZN8__main__25runningsum_numba_loop$242Ex"(i64* noalias nocapture %retptr, { i8*, i32 }** noa
entry:
  %.27 = add nsw i64 %arg.N, 1
  %.76 = icmp slt i64 %arg.N, 0
  %spec.select = select i1 %.76, i64 0, i64 %.27
  %.1214 = icmp sgt i64 %spec.select, 0
  br i1 %.1214, label %B20.lr.ph, label %B32

B20.lr.ph:                                        ; preds = %entry
  %0 = xor i64 %spec.select, -1
  %1 = icmp sgt i64 %0, -2
  %smax = select i1 %1, i64 %0, i64 -2
  %2 = add i64 %spec.select, %smax
  %3 = add i64 %2, 1
  %4 = zext i64 %3 to i65
  %5 = zext i64 %2 to i65
  %6 = mul i65 %4, %5
  %7 = lshr i65 %6, 1
  %8 = trunc i65 %7 to i64
  %9 = add i64 %2, %8
  %10 = add i64 %9, 1
  br label %B32
```

Inspecting numba outputs

```
print(runningsum_numba_loop.inspect_asm()[(numba.int64,)])
```

```
        .text
        .file           "<string>"
        .globl          _ZN8__main__25runningsum_numba_loop$242Ex
        .p2align        4, 0x90
        .type           _ZN8__main__25runningsum_numba_loop$242Ex,@function
_ZN8__main__25runningsum_numba_loop$242Ex:
        xorl            %ecx, %ecx
        testq           %rdx, %rdx
        leaq            1(%rdx), %rax
        cmovsq          %rcx, %rax
        testq           %rax, %rax
        jle             .LBB0_2
        movq            %rax, %rcx
        notq            %rcx
        cmpq            $-3, %rcx
        movq            $-2, %rdx
        cmovgq          %rcx, %rdx
        leaq            (%rax,%rdx), %rcx
        addq            %rax, %rdx
        addq            $1, %rdx
        mulxq           %rcx, %rax, %rdx
        shldq           $63, %rax, %rdx
        addq            %rdx, %rcx
        addq            $1, %rcx
.LBB0_2:
        movq            %rcx, (%rdi)
        xorl            %eax, %eax
        retq
.Lfunc_end0:
        .size           _ZN8__main__25runningsum_numba_loop$242Ex, .Lfunc_end0-_ZN8__main__25runningsum_numba

        .globl          _ZN7cpython8__main__25runningsum_numba_loop$242Ex
        .p2align        4, 0x90
```