Python is slow: Myth or Curse?

Numerical Processing Tasks

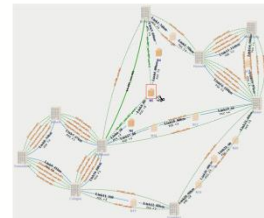
Nikolay Karelin

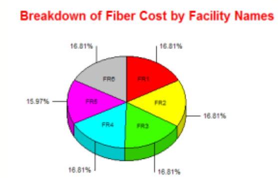
PiterPy
Saint Petersburg, April 22-23, 2016

Andre.Richter@VPIphotonics.com

Company Overview

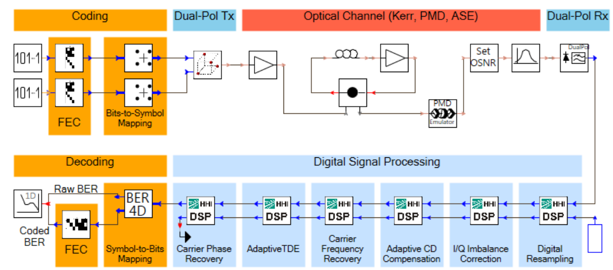
Leading provider of flexible simulation software and design services for 18+ years



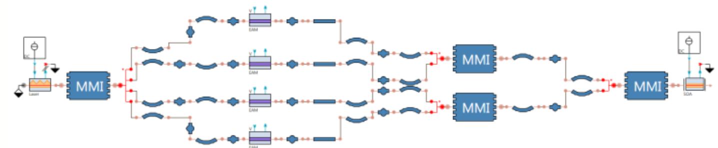


Supporting requirements of

- ✓ waveguides and fibers
- ✓ active/passive integrated photonics
- ✓ fiber optics
- ✓ optical transmission systems and networks
- ✓ link engineering and equipment configuration



Locations in Berlin, Boston, **Minsk**;
global network of regional representatives



The Standard for industry & academia

- ✓ 140+ public R&D institutions & universities
- ✓ 100+ private companies
- ✓ 1100+ citations in scientific publications

Value proposition

- ✓ Virtual prototyping for faster product development and reduced R&D efforts
- ✓ Research on cutting-edge technologies
- ✓ Teaching optical communications topics

Outline

- The right question
  - When and why Python is slow?
  - Interpreted vs. Dynamic
- Python practices
  - Pure python
  - NumPy
- Compilation
  - Numba
  - Cython

~~Is Python slow?~~

When Python can be slow?

Why?

How to make it fast?

Simplest Example

PyObject

```
In [2]: import dis

In [3]: def fmadd(x, y, z):
   ...:     return x*y+z
   ...:

In [4]: dis.dis(fmadd)
  2           0 LOAD_FAST                0 (x)
              3 LOAD_FAST                1 (y)
              6 BINARY_MULTIPLY
              7 LOAD_FAST                2 (z)
             10 BINARY_ADD
             11 RETURN_VALUE
```

`__add__()` [native] or custom code

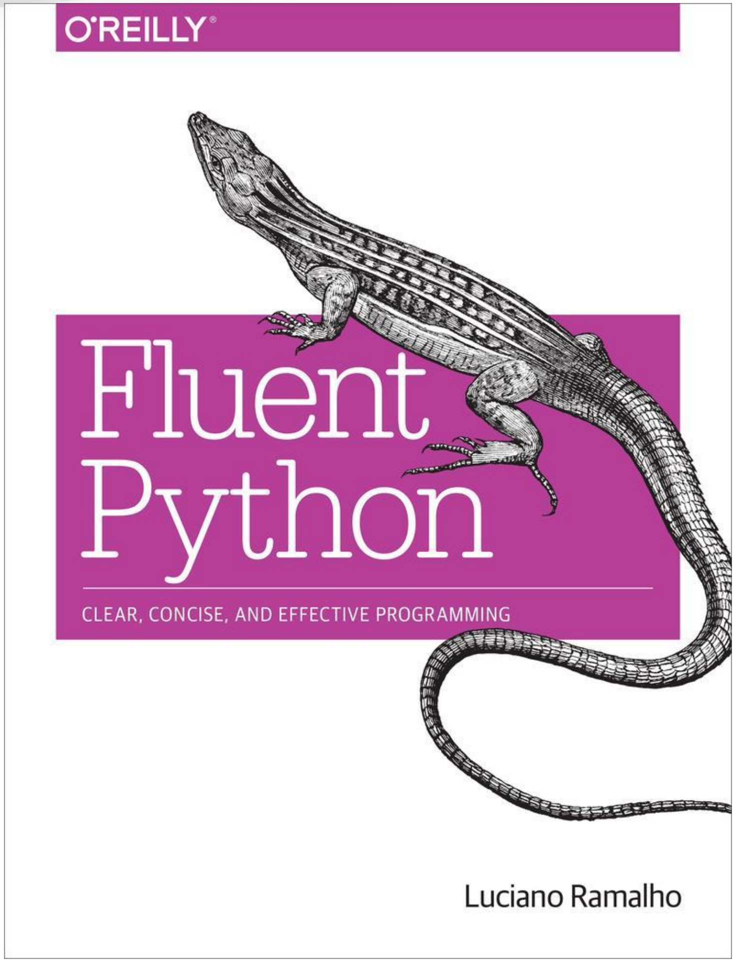
Full Polymorphism

```
In [9]: import numpy as np

In [5]: fmadd(1, 2, 3)
Out[5]: 5

In [6]: fmadd(1.0, 2.0, 3.0)
Out[6]: 5.0

In [7]: fmadd(2, 'x', 'yz')
Out[7]: 'xxyz'

In [8]: fmadd(2, [3, 4], [5, 6])
Out[8]: [3, 4, 3, 4, 5, 6]

In [10]: x = np.array([1, 2, 3])

In [11]: y = np.array([4, 5, 6])

In [12]: z = np.array([7, 8, 9])

In [13]: fmadd(z, y, z)
Out[13]: array([35, 48, 63])
```

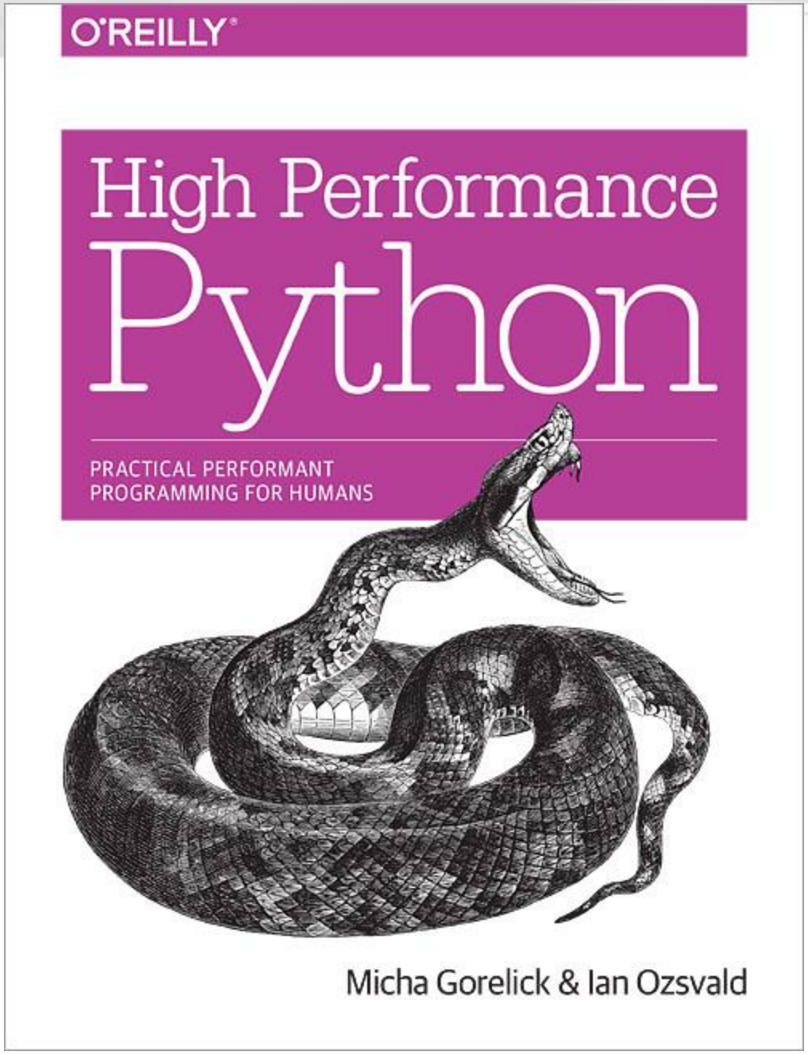
Python motto: *Everything is Object!*

Online Resources

- https://wiki.python.org/moin/PythonSpeed/PerformanceTips
  - Old
- http://scipy.github.io/old-wiki/pages/PerformancePython
  - Old… quite old
- https://docs.python.org/devguide/
- cPython’s source code
- https://wiki.python.org/moin/NumericAndScientific
- https://wiki.python.org/moin/TimeComplexity



http://shop.oreilly.com/product/0636920032519.do



http://shop.oreilly.com/product/0636920028963.do

General Approach

Make it work

Make work correct

Tests / profiling

Make it fast

http://c2.com/cgi/wiki?MakeItWorkMakeItRightMakeItFast

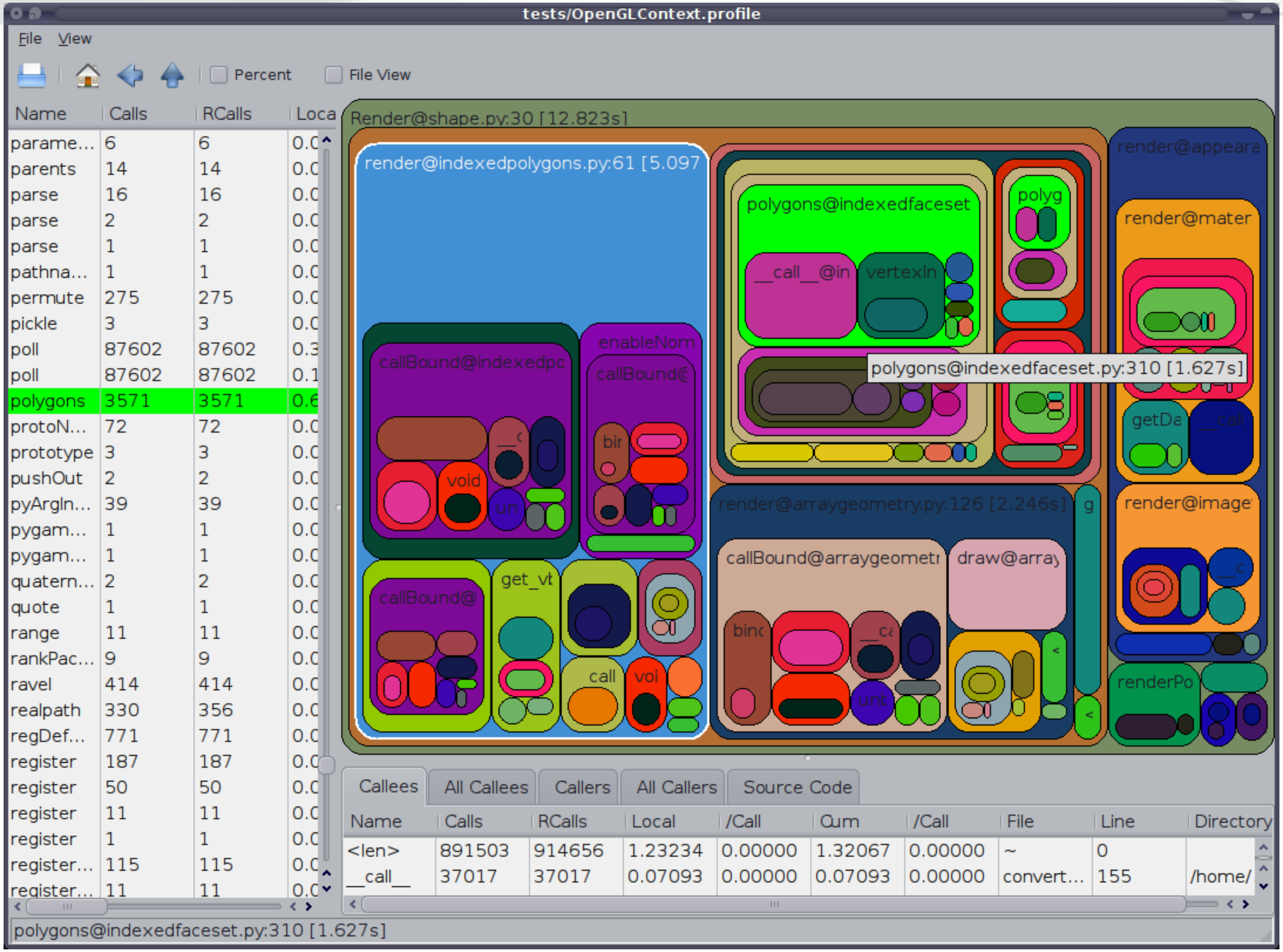
Main Idea

- Find *performance-critical* places in code
  - Normally only small parts
  - May depend on input data
    (many iterations with small data set vs. large data)
- Optimize hotspots
  - Remove polymorphism (correct data structures is the **must**)
  - Optionally compile the code
- Optimize to hardware
  - Release GIL
  - asyncio
  - … (know the hardware)

Tools of the Trade

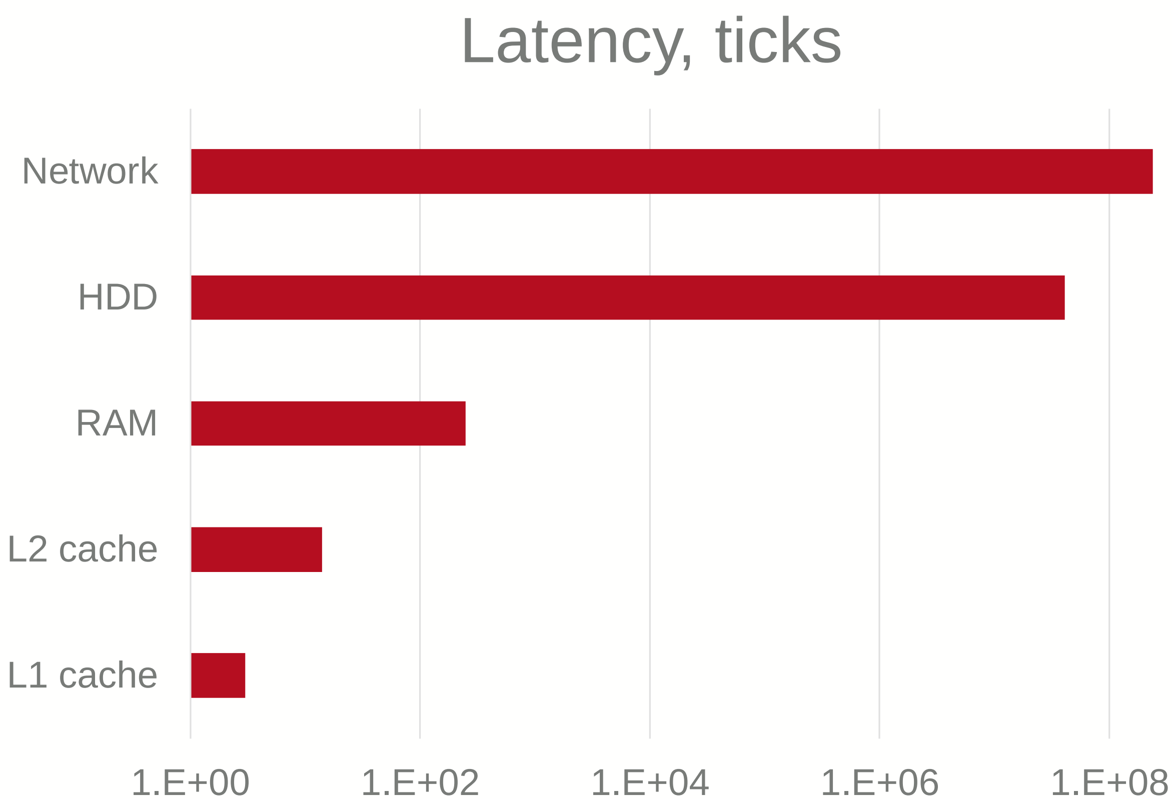
- Function level
  - cProfile (standard lib)
  - runsnakerun (http://www.vrplumber.com/programming/runsnakerun/)
- Line level
  - line_profiler (http://pypi.python.org/pypi/line_profiler/)
- Memory profiling
  - memory_profiler (https://pypi.python.org/pypi/memory_profiler)
  - runsnakerun
  - heapy (https://pypi.python.org/pypi/guppy/)
- dis (standard lib)

runsnakerun: Example



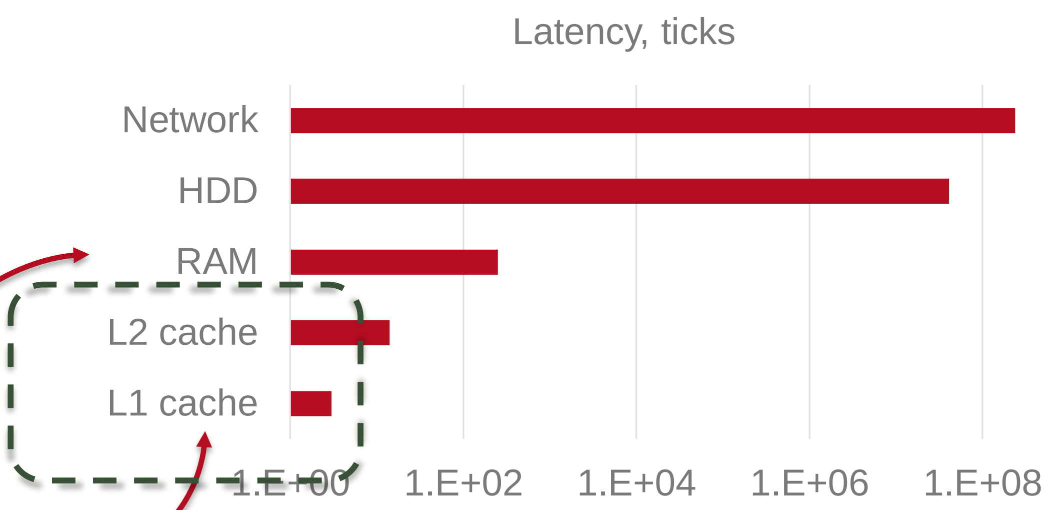
http://www.vrplumber.com/programming/runsnakerun/

Latency Scales



Fluent Python / https://www.youtube.com/watch?v=M-sc73Y-zQA

Presentation Focus



- **CPU-bound**
- Memory bound
- IO (network/GUI) bound
  - … not here

Python Performance

- Dynamic nature
  - Multiple lookups for functions and methods
  - Checks for types, etc.
- Memory management
  - Automatic allocation
  - GC
- Interpreted
  - Least important
- Other side: developer performance

How to Make Better

(Pure Python)

- Local variables (function refs)
- Less function calls
- Avoid string concatenation
- `while` loop -> `for` loop -> list comprehensions
- Less dynamic
- Use built-ins
- …

Simple Example

Log table - 100 000 000 numbers ;)

54 s

```
dat = []
for x in arg:
    dat.append(math.log10(x))
dat = np.array(dat)
t0 = time.clock() - t0
```

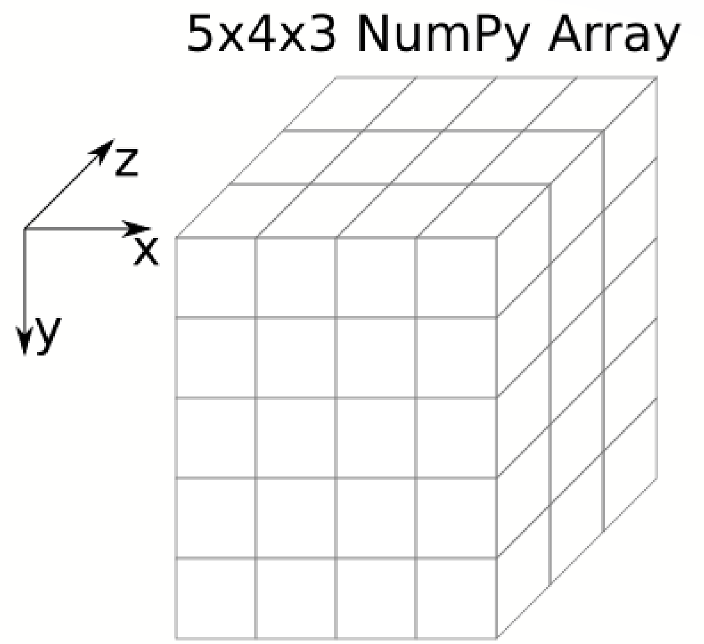
```
  2           0 BUILD_LIST               0
              3 STORE_FAST               1 (dat)

  3           6 SETUP_LOOP              36 (to 45)
              9 LOAD_FAST                0 (arg)
             12 GET_ITER
        >>   13 FOR_ITER                28 (to 44)
             16 STORE_FAST               2 (x)

  4          19 LOAD_FAST                1 (dat)
             22 LOAD_ATTR                0 (append)
             25 LOAD_GLOBAL              1 (math)
             28 LOAD_ATTR                2 (log10)
             31 LOAD_FAST                2 (x)
             34 CALL_FUNCTION            1
             37 CALL_FUNCTION            1
             40 POP_TOP
             41 JUMP_ABSOLUTE           13
        >>   44 POP_BLOCK

  5     >>   45 LOAD_GLOBAL              3 (np)
             48 LOAD_ATTR                4 (array)
             51 LOAD_FAST                1 (dat)
```

37 s (~ 1.46 x)

```
t0 = time.clock()
dat = np.array([math.log10(x) for x in arg])
t0 = time.clock() - t0
flg = np.allclose(ref, dat)
```

```
  2           0 LOAD_GLOBAL              0 (np)
              3 LOAD_ATTR                1 (array)
              6 BUILD_LIST               0
              9 LOAD_FAST                0 (arg)
             12 GET_ITER
        >>   13 FOR_ITER                21 (to 37)
             16 STORE_FAST               1 (x)
             19 LOAD_GLOBAL              2 (math)
             22 LOAD_ATTR                3 (log10)
             25 LOAD_FAST                1 (x)
             28 CALL_FUNCTION            1
             31 LIST_APPEND              2
             34 JUMP_ABSOLUTE           13
        >>   37 CALL_FUNCTION            1
             40 RETURN_VALUE
```

- nD array
  - Primitive types
  - Structs
  - PyObject (inefficient)
  - Buffer protocol
- **ufunc's**
  - Hide loops
  - Release GIL
- Extension API
  - Custom functions (C, …)
  - See below
- MKL bindings



http://brosnotes.com/python-series-on-number-crunching-data-visualization-getting-started-using-numpy-2/

http://www.slideshare.net/shoheihido/sci-pyhistory

Example

1.55 s (35x)

```
t0 = time.clock()
ref = np.log10(arg)
t0 = time.clock() - t0
```

- Approach:
- Python as a glue language
- Calling / orchestrating of external libraries

Trades Memory for Speed

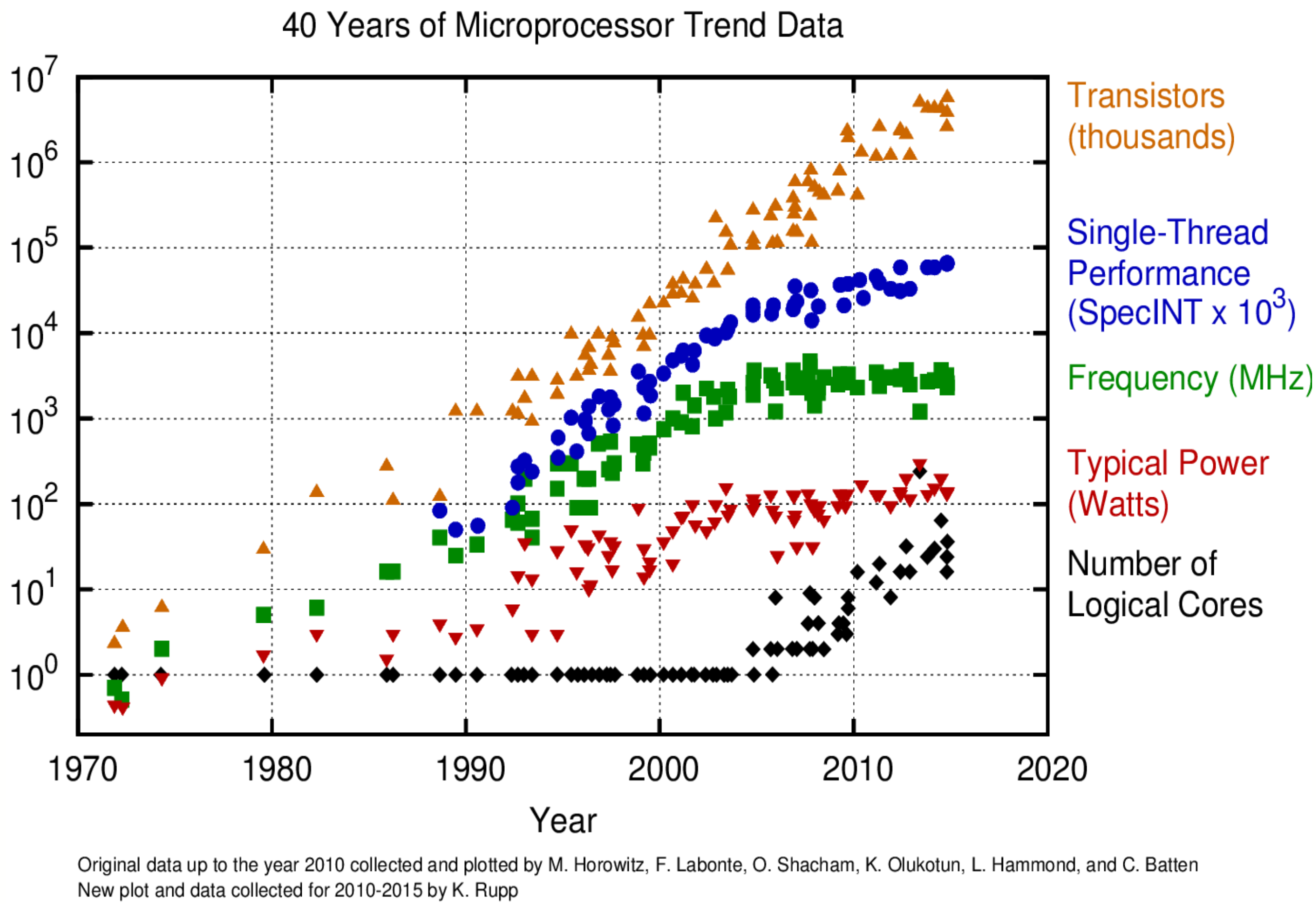
```
def py_update(u):
    nx, ny = u.shape
    for i in xrange(1,nx-1):
        for j in xrange(1, ny-1):
            u[i,j] = ((u[i+1, j] + u[i-1, j]) * dy2 +
                      (u[i, j+1] + u[i, j-1]) * dx2) / (2*(dx2+dy2))

def calc(N, Niter=100, func=py_update, args=()):
    u = zeros([N, N])
    u[0] = 1
    for i in range(Niter):
        func(u,*args)
    return u
```

```
def num_update(u):
    u[1:-1,1:-1] = ((u[2:,1:-1]+u[:-2,1:-1])*dy2 +
                    (u[1:-1,2:] + u[1:-1,:-2])*dx2) / (2*(dx2+dy2))
```

… and can be quite non-trivial!

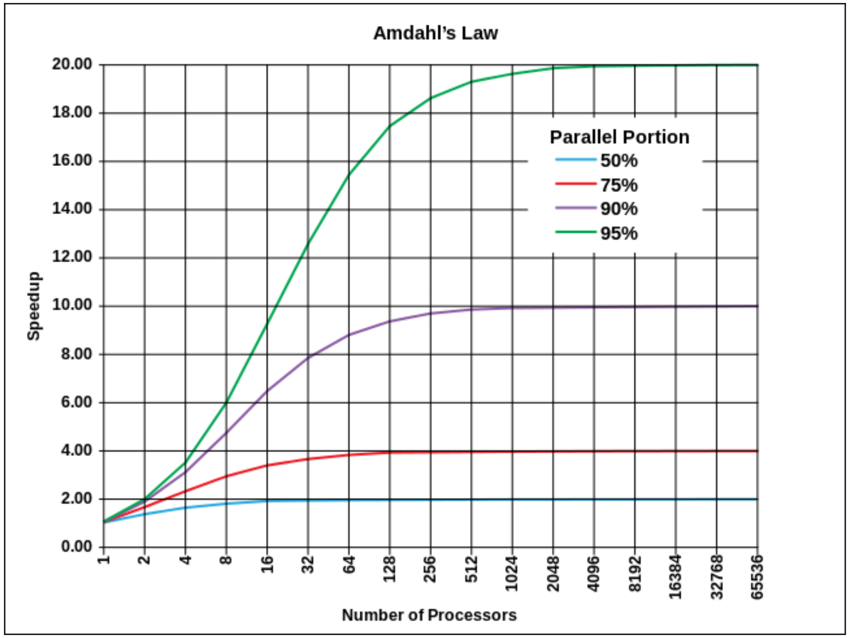
http://technicaldiscovery.blogspot.com.by/2011/06/speeding-up-python-numpy-cython-and.html

Why Parallel?



Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Amdahl's Law



$$S = \frac{1}{1 - p + p/s}$$

- S – speedup
- p – parallelizable part of algorithm
- s – number of processors

https://en.wikipedia.org/wiki/Amdahl's_law

Embarrassingly Parallel

```
t0 = time.clock()
dat = np.hstack(parallel_map(log_table_np, [arg], threads=4))
t0 = time.clock() - t0
```

2: 0.81 s

4: 0.5 s

- Variants:
  - “Perfectly parallel”
  - “Pleasingly parallel”
- Examples:
  - Brute-force (crypto)
  - Climate models
  - Computer graphics
  - …
- Map pattern

http://scipy.github.io/old-wiki/pages/ParallelProgramming

Python Compilation Space

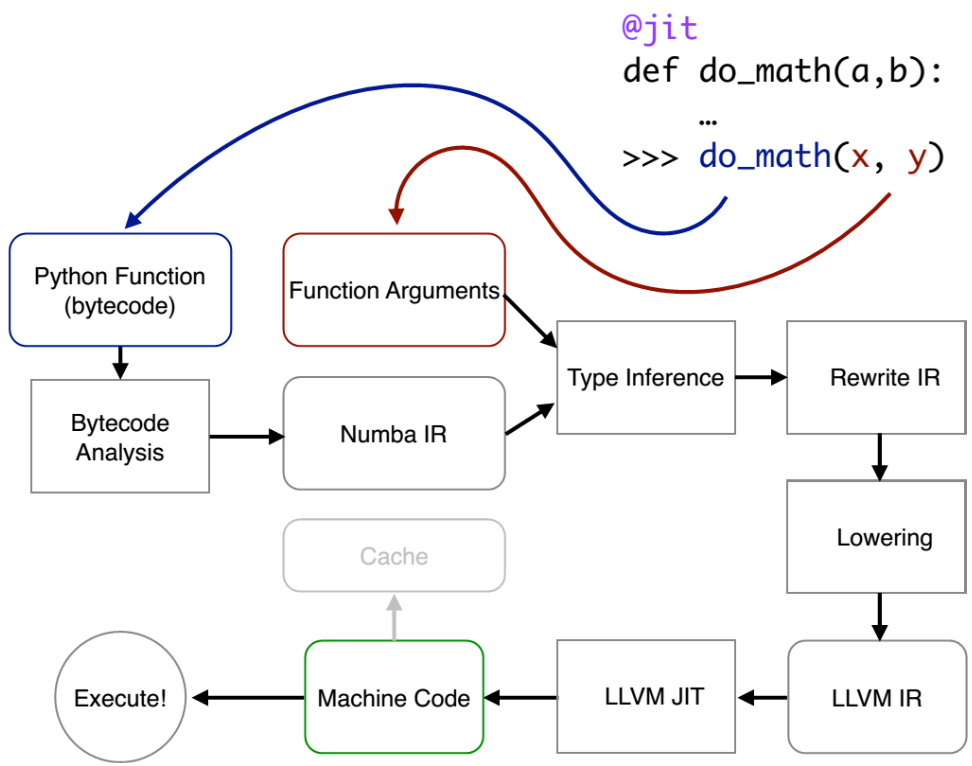
	Relies on CPython / libpython	Replaces CPython / libpython
Ahead Of Time	**Cython** Shedskin Nuitka (today) Pythran	Nuitka (future)
Just In Time	**Numba** HOPE Theano Psyco Unladen Swallow Pyjion	Pyston PyPy
Install-time	Numpy	

https://www.youtube.com/watch?v=mNvPiV37F7Q
http://www.slideshare.net/teoliphant/python-as-the-zen-of-data-science
https://github.com/Microsoft/Pyjion

Numba

- Dynamic Python compiler (JIT)
  - Continuum Analytics
  - FLOSS
  - CUDA support since v. 0.13 (Apr. 2014)
  - … still buggy
- Bytecode -> PyLLVM -> Native code (caching)
- Numpy support
- Data analysis / simulation / …
- Anaconda distribution

How Numba Works



http://www.slideshare.net/teoliphant/python-as-the-zen-of-data-science

Example

Type annotations

```
@numba.jit(numba.float64[:](numba.float64[:]), nopython=True)
def log_numba(arg):
    data = np.zeros_like(arg)
    for i in xrange(arg.shape[0]):
        data[i] = np.log10(arg[i])
    return data
```

Error if not compiled to native types

~ 1.68s (32x)

Cython

- Python -> C -> .pyd (.so/.dll) translator
- HTML for profiling
- Binding of C extensions
- Optional type annotations, NumPy support
- **nogil** context manager
- openMP library (parallel range)
- (Almost) full Python support
  – http://docs.cython.org/src/userguide/limitations.html (4 cases!)
- Base for Nuitka

Ipython Magic

1.77 s (~30x)

```
%%cython -a
import numpy as np
import cython
cimport numpy as np
DTYPE = np.float64
ctypedef np.float64_t DTYPE_t

from libc.math cimport log10

@cython.boundscheck(False) # turn of bounds-checking
def cython_log(np.ndarray[DTYPE_t, ndim=1] arg):
    cdef int i
    cdef np.ndarray[DTYPE_t, ndim=1] h = np.zeros_like(arg)
    for i in range(arg.shape[0]):
        h[i] = log10(arg[i])
    return h
```

Typing

```
@cython.boundscheck(False) # turn of bounds-checking
def cython_log(np.ndarray[DTYPE_t, ndim=1] arg):
    cdef int i
    cdef np.ndarray[DTYPE_t, ndim=1] h = np.zeros_like(arg)
    for i in range(arg.shape[0]):
        h[i] = log10(arg[i])
    return h
```

Annotated Python (.pyx) + setup.py file
Variant: decorators (.py) + .pxd file + setup.py

Conversion & Annotation

Generated by Cython 0.23.4

%%cython -a

Yellow lines hint at Python interaction.

Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
+01: import numpy as np
 02: import cython
 03: cimport numpy as np
+04: DTYPE = np.float64
 05: ctypedef np.float64_t DTYPE_t

 09: @cython.boundscheck(False) # turn of bounds-checking
+10: def cython_log(np.ndarray[DTYPE_t, ndim=1] arg):
 11:     cdef int i
+12:     cdef np.ndarray[DTYPE_t, ndim=1] h = np.zeros_like(arg)
+13:     for i in range(arg.shape[0]):
+14:         h[i] = log10(arg[i])
+15:     return h
```

VPIphotonics DESIGN AUTOMATION

Conversion & Annotation

```
 09: @cython.boundscheck(False) # turn of bounds-checking
+10: def cython_log(np.ndarray[DTYPE_t, ndim=1] arg):
 /* Python wrapper */
 static PyObject *__pyx_pw_46_cython_magic_d60373eecefd175d926493f7af8fafae_1cython_log(PyObject *__pyx_self, PyObjec
 t *__pyx_v_arg); /*proto*/
 static PyMethodDef __pyx_mdef_46_cython_magic_d60373eecefd175d926493f7af8fafae_1cython_log = {"cython_log", (PyCFunc
 tion)__pyx_pw_46_cython_magic_d60373eecefd175d926493f7af8fafae_1cython_log, METH_O, 0};
 static PyObject *__pyx_pw_46_cython_magic_d60373eecefd175d926493f7af8fafae_1cython_log(PyObject *__pyx_self, PyObjec
 t *__pyx_v_arg) {
   PyObject *__pyx_r = 0;
   __Pyx_RefNannyDeclarations
   __Pyx_RefNannySetupContext("cython_log (wrapper)", 0);
   if (unlikely(!__Pyx_ArgTypeTest(((PyObject *)__pyx_v_arg), __pyx_ptype_5numpy_ndarray, 1, "arg", 0))) {__pyx_filen
 ame = __pyx_f[0]; __pyx_lineno = 10; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
   __pyx_r = __pyx_pf_46_cython_magic_d60373eecefd175d926493f7af8fafae_cython_log(__pyx_self, ((PyArrayObject *)__pyx
 _v_arg));
   CYTHON_UNUSED int __pyx_lineno = 0;
   CYTHON_UNUSED const char *__pyx_filename = NULL;
   CYTHON_UNUSED int __pyx_clineno = 0;

   /* function exit code */
   goto __pyx_L0;
   __pyx_L1_error:;
   __pyx_r = NULL;
   __pyx_L0:;
   __Pyx_RefNannyFinishContext();
   return __pyx_r;
```

Conversion & Annotation

```
@cython.boundscheck(False) # turn of bounds-checking
def cython_log(np.ndarray[DTYPE_t, ndim=1] arg):
    cdef int i
    cdef np.ndarray[DTYPE_t, ndim=1] h = np.zeros_like(arg)
    for i in range(arg.shape[0]):
  __pyx_t_6 = (__pyx_v_arg->dimensions[0]);
  for (__pyx_t_7 = 0; __pyx_t_7 < __pyx_t_6; __pyx_t_7+=1) {
    __pyx_v_i = __pyx_t_7;
        h[i] = log10(arg[i])
    return h
```

Important: the body of loop is highly optimized C code!

Speedup Summary

Approach	Time, s	Speed-up
Pure Python	54	
List comprehension	37	1.46
Numpy	1.55	35
Numpy, 2 threads	0.81	67 (1.9 vs. 1 thread)
Numpy, 4 threads	0.5	108 (3.1 vs. 1 thread)
Numba	1.68	32
Cython	1.77	30

IPython Notebook (DRAFT, to be updated):
http://nbviewer.jupyter.org/github/karelin/PiterPy2016/blob/master/log_table.ipynb

Conclusion

- Python can be slow or not
  - Slowness of cPython VM is dark side of flexibility
  - Depends on the task
  - Optimization of hotspots possible
- Numpy
  - Speed-up of many important algorithms
  - Multithreading (MKL or native), vectorized operations
- Compilation of Python:
  - Possible
  - Can be non-trivial
  - Rewarding

Thank you for attention!