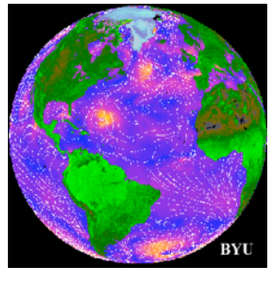
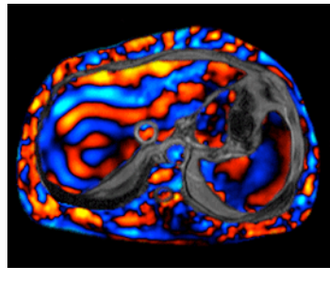
Performance Python
Introduction to Numba

Travis E. Oliphant
Continuum Analytics
February 18, 2015

Who am I?

- PhD 2001 from Mayo Clinic in Biomedical Engineering
- MS/BS degrees in Elec. Comp. Engineering
- Creator of **SciPy** (1998-2001)
- Professor at BYU (2001-2007)
- Author of **NumPy** (2005-2012)
- Started **Numba** (2012)
- Founding Chair of **Numfocus** / **PyData**
- Current Python Software Foundation Director





NumPy SciPy

Num FOCUS
Open Code, Better Science

python

PyData

CONTINUUM ANALYTICS

Why Performance Matters

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

- Donald E. Knuth

Why Python

License

Free and Open Source, Permissive License

Community

- Broad and friendly community
- Over 36,000 packages on PyPI
- Commercial Support
- Many conferences (PyData, SciPy, PyCon...)

Readable Syntax

- Executable pseudo-code
- Can understand and edit code a year later
- Fun to develop
- Use of Indentation

IPython

- Interactive prompt on steroids (Notebook)
- Allows less working memory
- Allows failing quickly for exploration

Modern Constructs

- List comprehensions
- Iterator protocol and generators
- Meta-programming
- Introspection
- (JIT Compiler and Concurrency)

Batteries Included

- Internet (FTP, HTTP, SMTP, XMLRPC)
- Compression and Databases
- Logging, unit-tests
- Glue for other languages
- Distribution has much, much more....

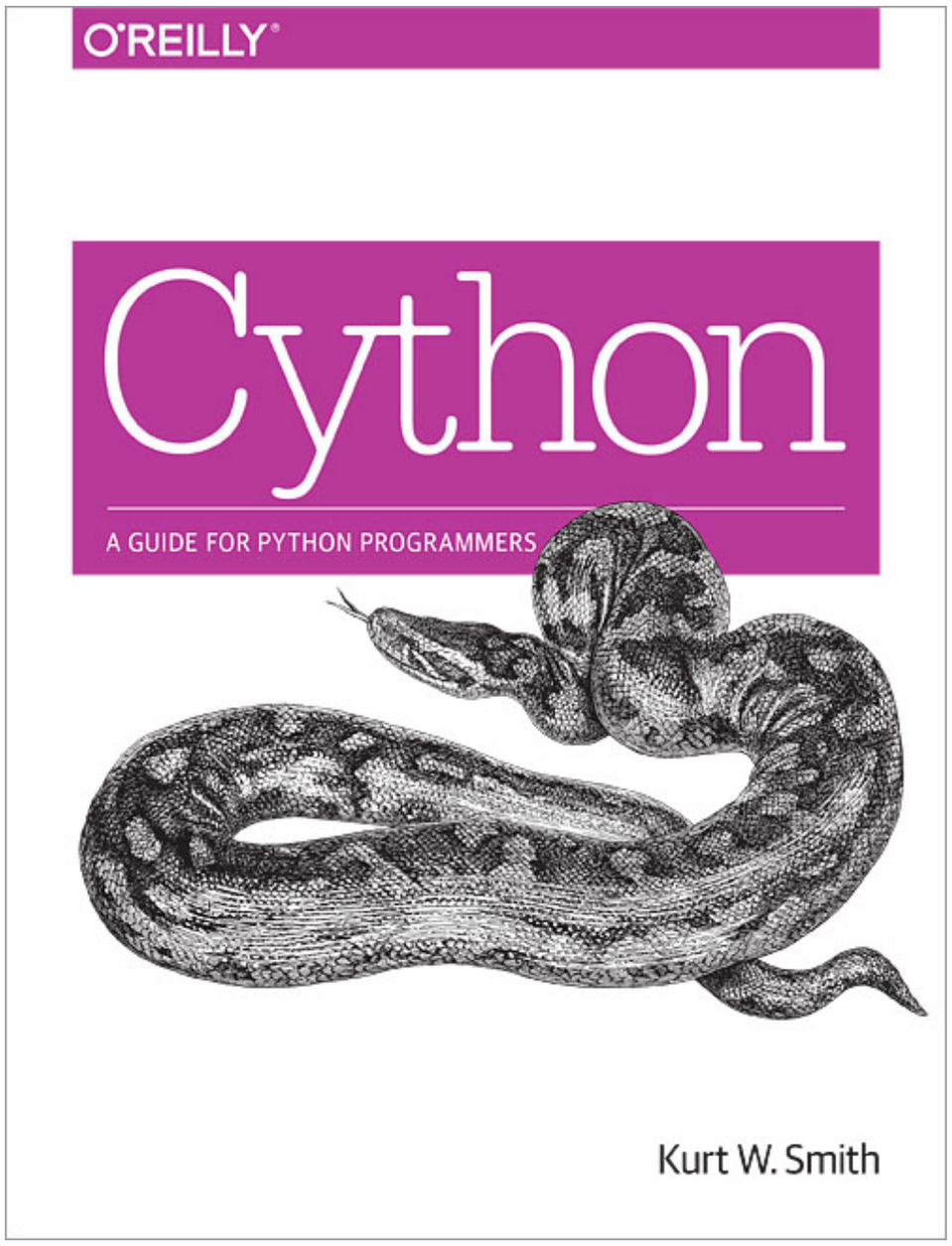
Why Python?

- For many programs, the most important resource is *developer time* (both to create and to maintain)
- Code that respects developer time is:
  - Easy to read
  - Easy to understand
  - Easy to modify
- But execution speed does matter at times: Then what do you do?

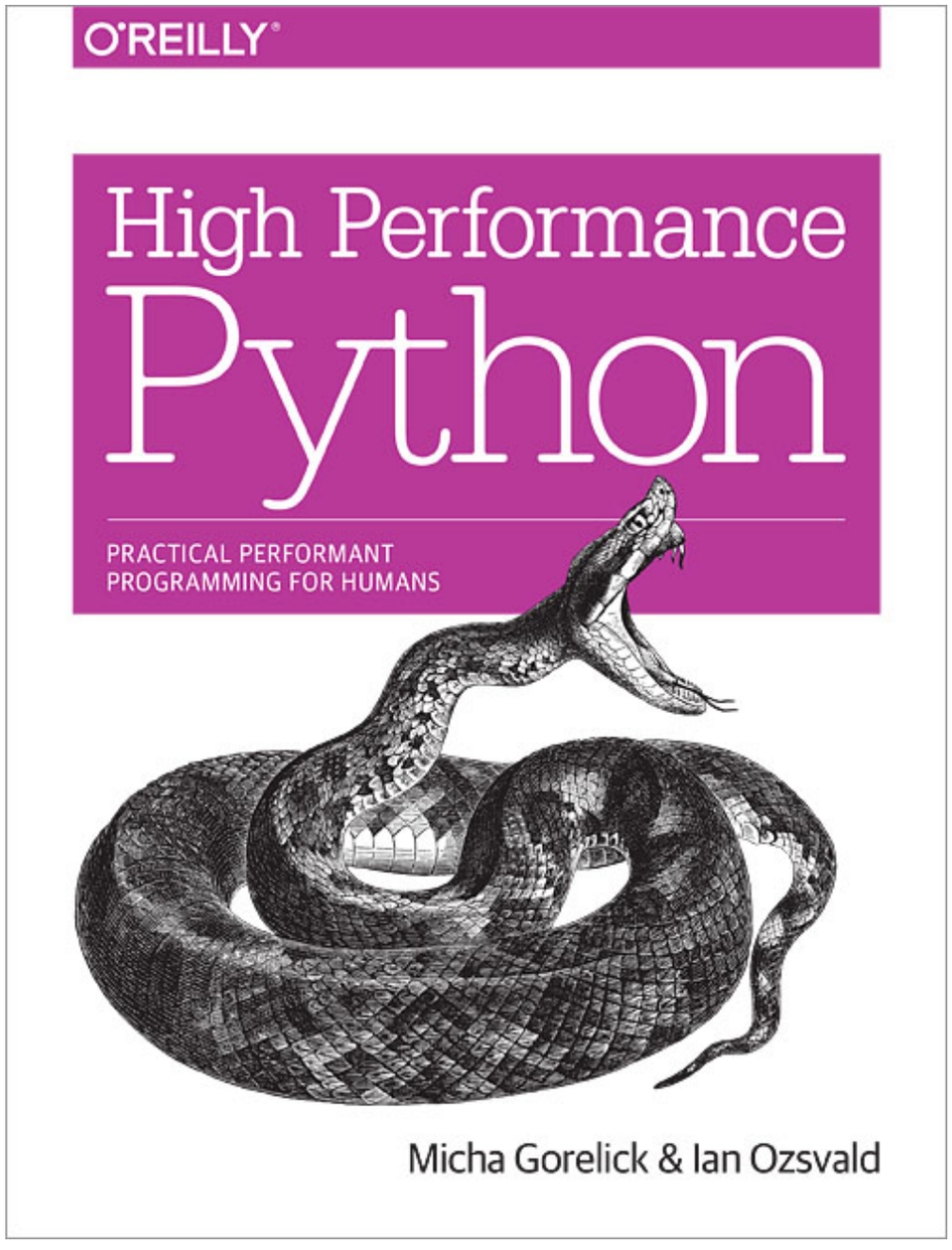
Some good books (pre-numba)

Just released book by
Kurt Smith

Excellent tutorial and
reference



Some good books (pre-numba)



Broad overview of Python tricks + Cython

References old version of Numba

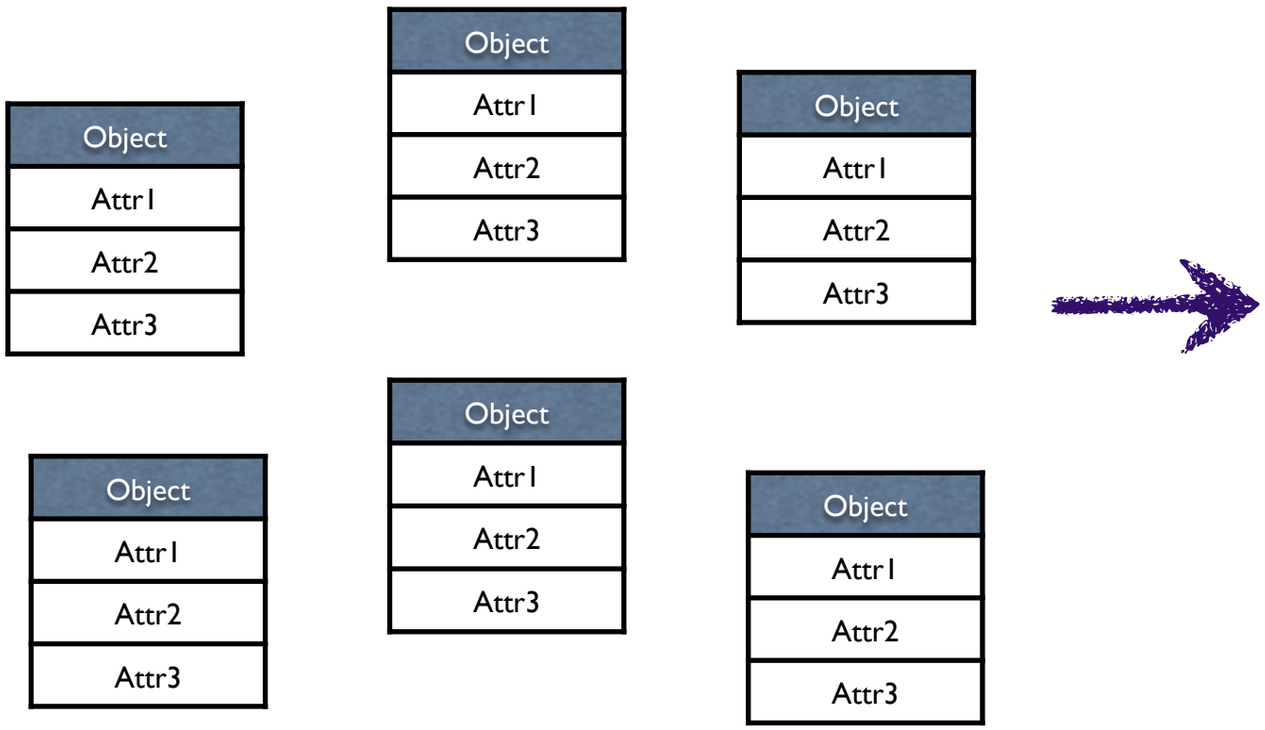
Achieving Performance in Python

When you need speed in Python the most important things you can do are:

1. **Use profiling** to understand where your program spends time. (Most of your code is irrelevant with respect to time spent. Only worry about the parts that matter.) I like the line-profiler **kernprof.py** *binstar search -t conda line_profiler*
2. Leverage NumPy-stack when working with data.
3. Use Numba to optimize hot-spots
4. Occasionally use Cython (especially for libraries).

Data Structures Matter

- "Organize data-together" and operate on it together with array-level operations (e.g. NumPy or Pandas) (column-oriented is a subset of array-oriented)
- **Don't use a lot of little small objects**



	Attr1	Attr2	Attr3
Object1			
Object2			
Object3			
Object4			
Object5			
Object6			

How can Numba help?

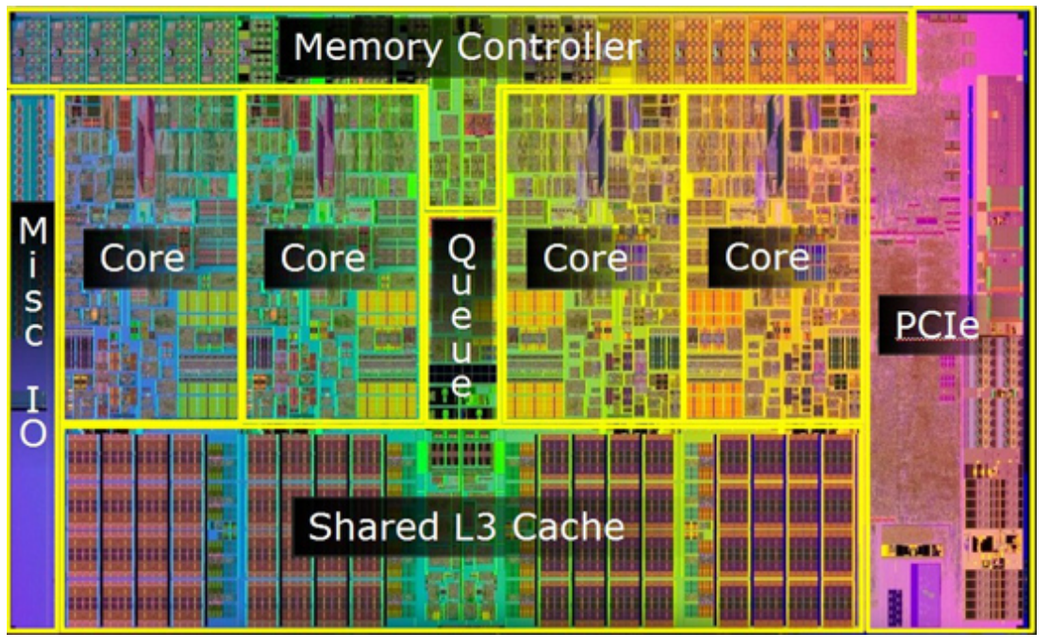
- If you have big NumPy arrays with your data (remember Pandas uses NumPy under-the-covers, so this applies to Pandas). Numba makes it easy to write simple functions that are **fast** that work with that data.
- Numba is an open source Just-In-Time compiler for Python functions.
- From the types of the function arguments, Numba can often generate a specialized, fast, machine code implementation at runtime.
- Designed to work best with numerical code and NumPy arrays.
- Uses the LLVM library as the compiler backend.

Numba Features

- Numba supports:
  - Windows (XP and later), OS X (10.7 and later), and Linux
  - 32 and 64-bit x86 CPUs and NVIDIA GPUs
  - Python 2 and 3
  - NumPy versions 1.6 through 1.9
- It does *not* require a C/C++ compiler on the user's system.
- Requires less than 70 MB to install.
- Does *not* replace the standard Python interpreter (it's just another module — all of your existing Python libraries are still available)

More Ranting

- Today's vector machines (and vector co-processors, or GPUS) were **made** for array-oriented computing (and Numba and Fortran).
- There is a reason Fortran remains popular.



Goal:

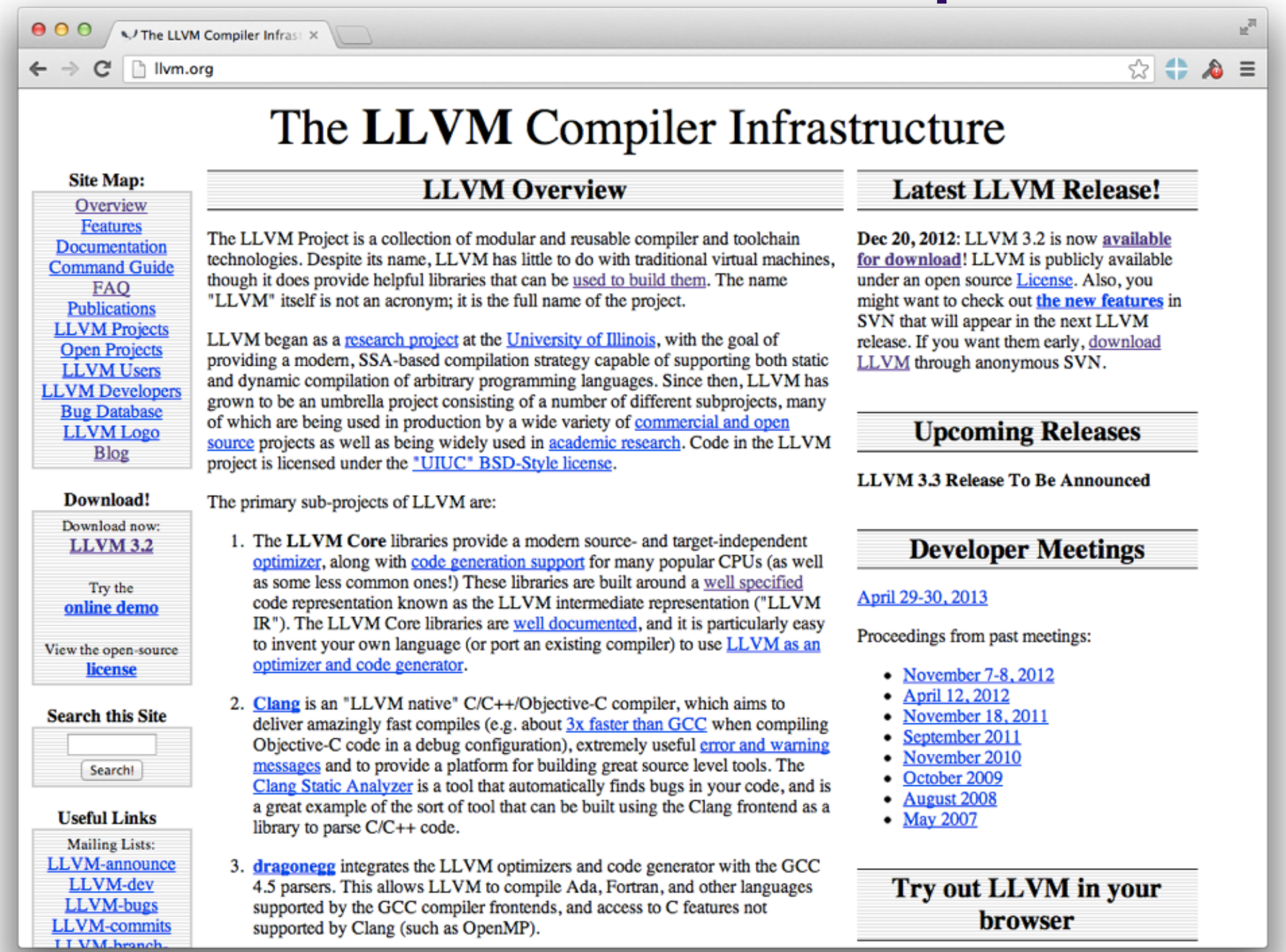
Numba should be the world's best
array-oriented compiler.

Do we have to write the full compiler??

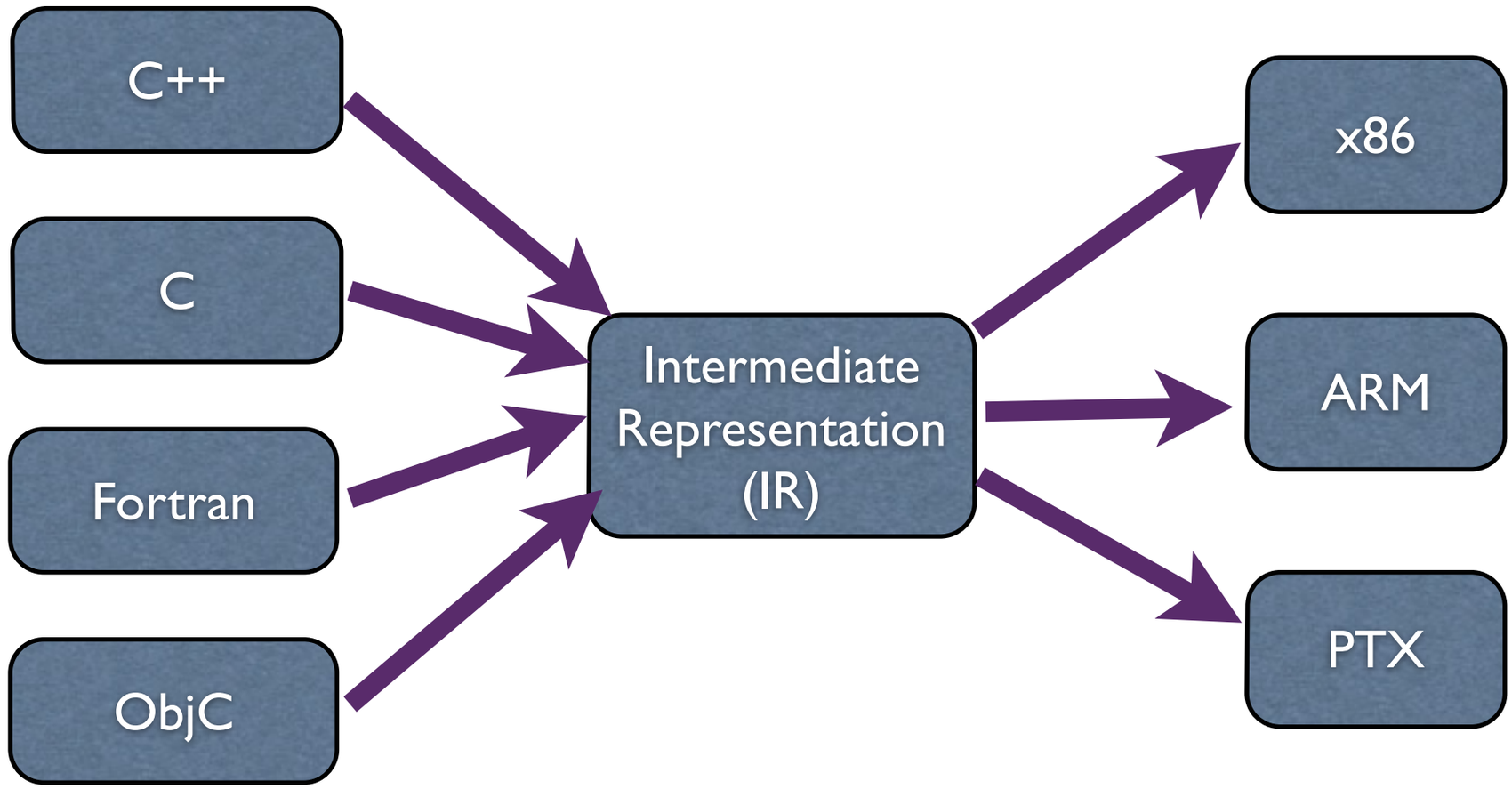
No!

LLVM has done much heavy lifting

LLVM = Compilers for everybody



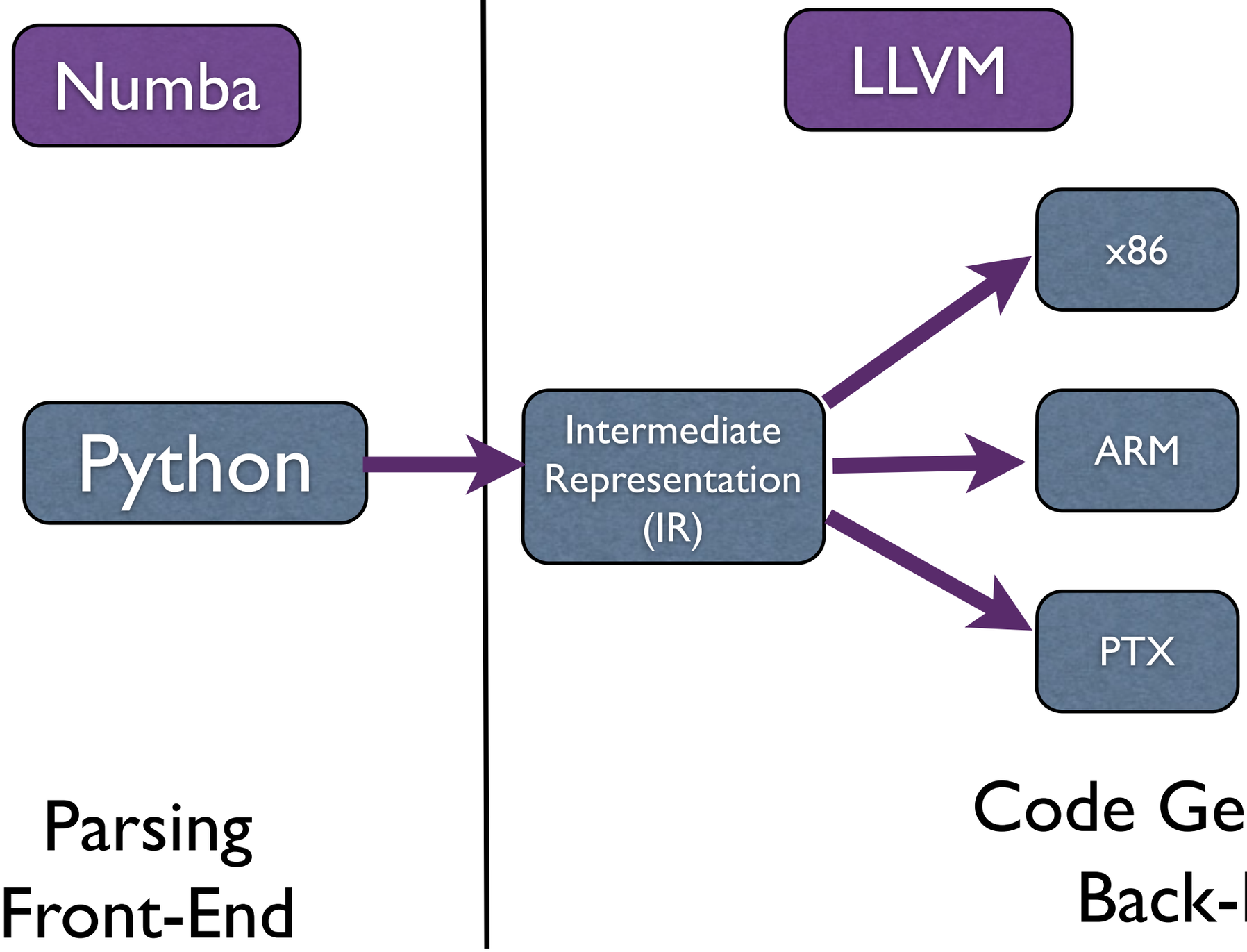
Face of a modern compiler



Parsing
Front-End

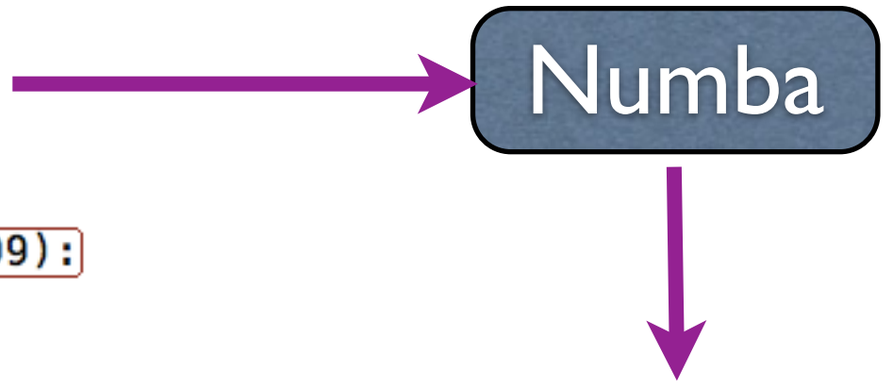
Code Generation
Back-End

Face of a modern compiler



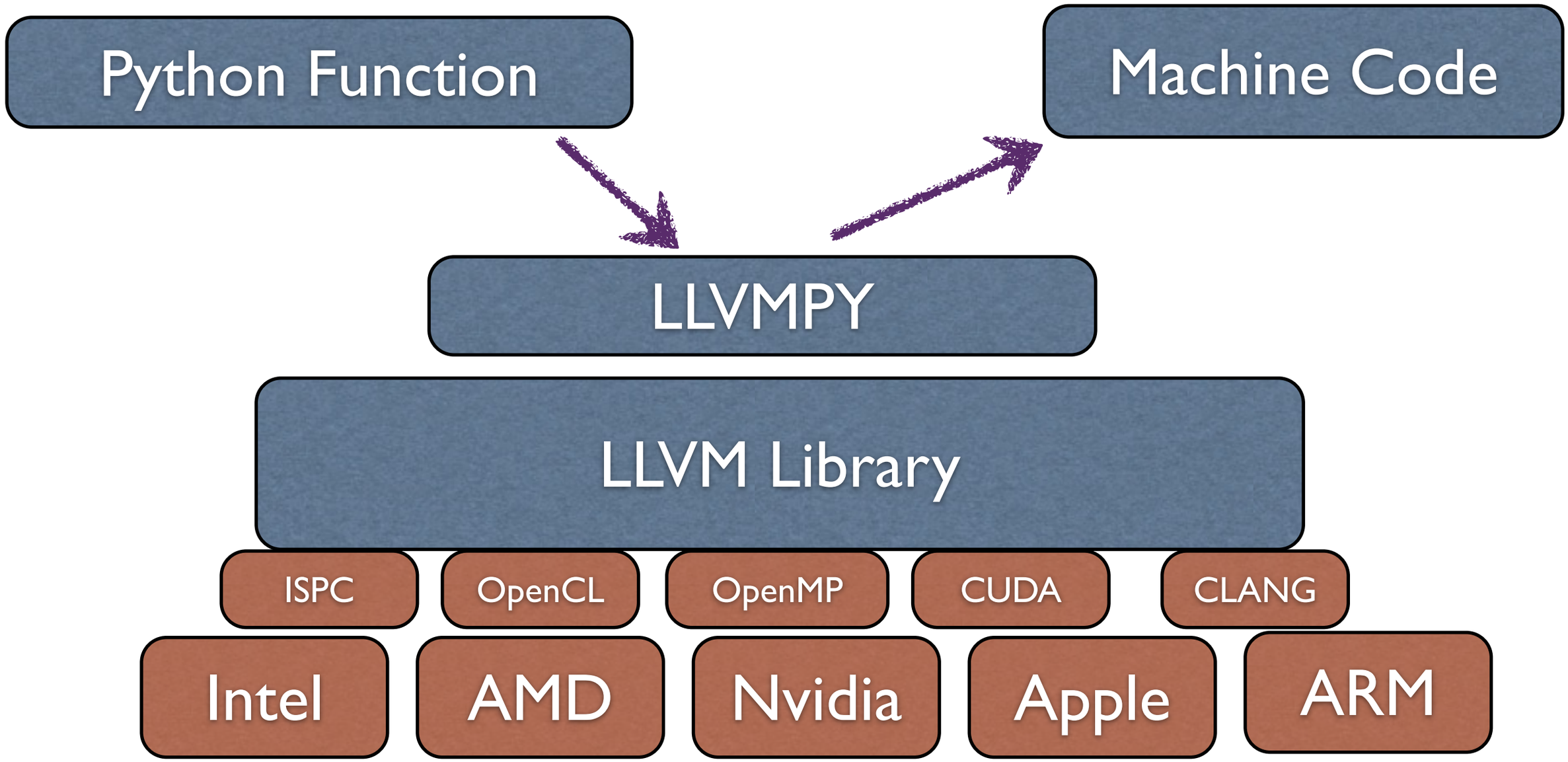
Example

```
1  from numba import njit
2
3  @njit
4  def simple():
5      total = 0.0
6      for i in range(9999):
7          for j in range(1,9999):
8              total += (i / j)
9      return total
```

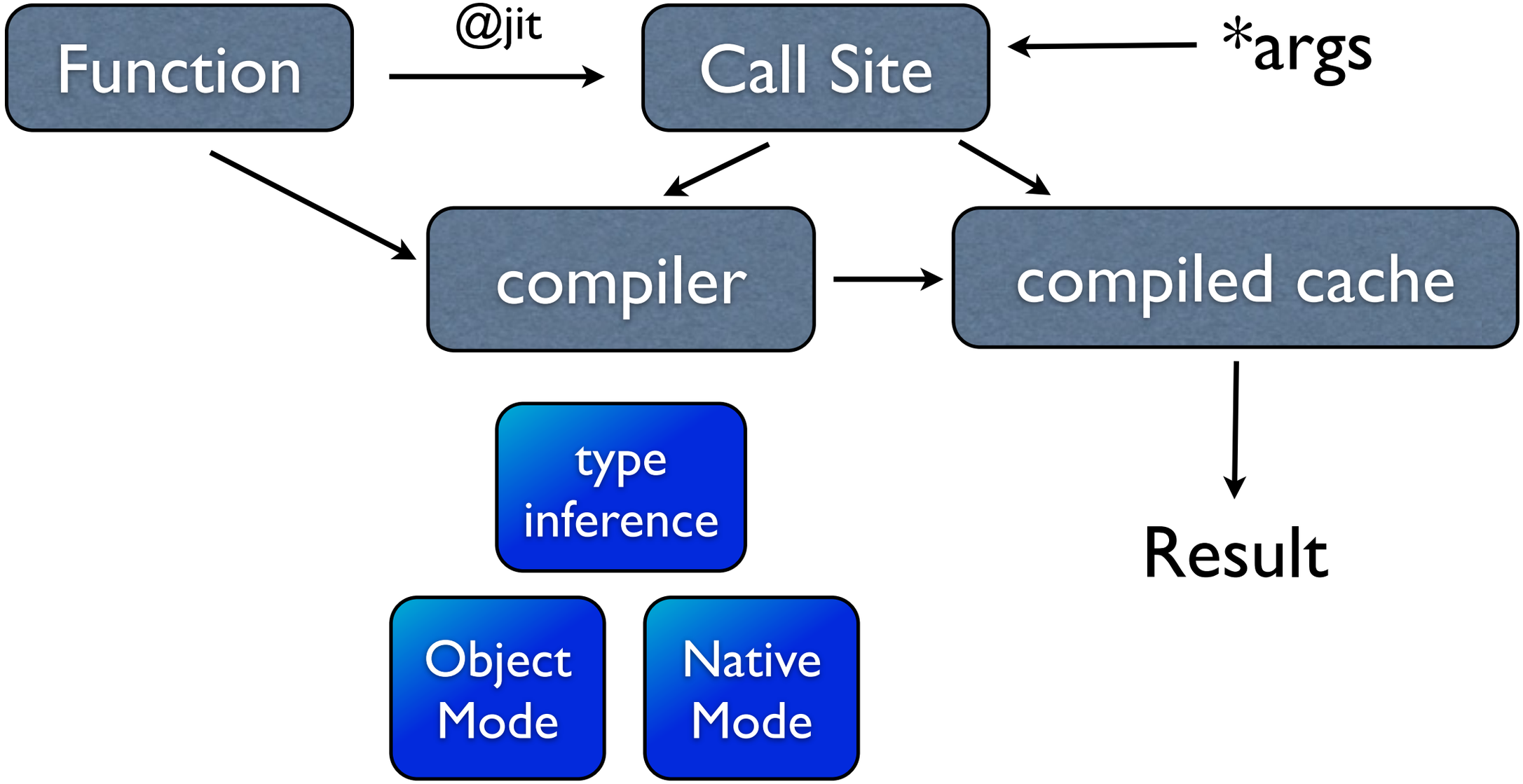



```
define double @__numba_specialized_0___main___2E_simple() nounwind readnone {
entry:
  br label %"for_condition_7:17.preheader"

"for_condition_7:17.preheader":                   ; preds = %"exit_for_7:8", %entry
  %total_28 = phi double [ 0.000000e+00, %entry ], [ %3, %"exit_for_7:8" ]
  %storemerge7 = phi i64 [ 0, %entry ], [ %0, %"exit_for_7:8" ]
  br label %"loop_body_8:12"

"exit_for_6:4":                                   ; preds = %"exit_for_7:8"
  ret double %3

"exit_for_7:8":                                   ; preds = %"loop_body_8:12"
  %0 = add i64 %storemerge7, 1
  %exitcond9 = icmp eq i64 %0, 9999
  br i1 %exitcond9, label %"exit_for_6:4", label %"for_condition_7:17.preheader"

"loop_body_8:12":                                 ; preds = %"loop_body_8:12", %"for_condition_7:17.prel
  %lsr.iv = phi i64 [ %lsr.iv.next, %"loop_body_8:12" ], [ 1, %"for_condition_7:17.preheader" ]
  %total_36 = phi double [ %total_28, %"for_condition_7:17.preheader" ], [ %3, %"loop_body_8:12" ]
  %1 = sdiv i64 %storemerge7, %lsr.iv
  %2 = sitofp i64 %1 to double
  %3 = fadd double %total_36, %2
  %lsr.iv.next = add i64 %lsr.iv, 1
  %exitcond = icmp eq i64 %lsr.iv.next, 9999
  br i1 %exitcond, label %"exit_for_7:8", label %"loop_body_8:12"
}
```

Numba (comes from NumPy + Mamba)



How it works


Function
@jit
Call Site
*args
compiler
compiled cache
type inference
Object Mode
Native Mode
Result

Simple API --- jit

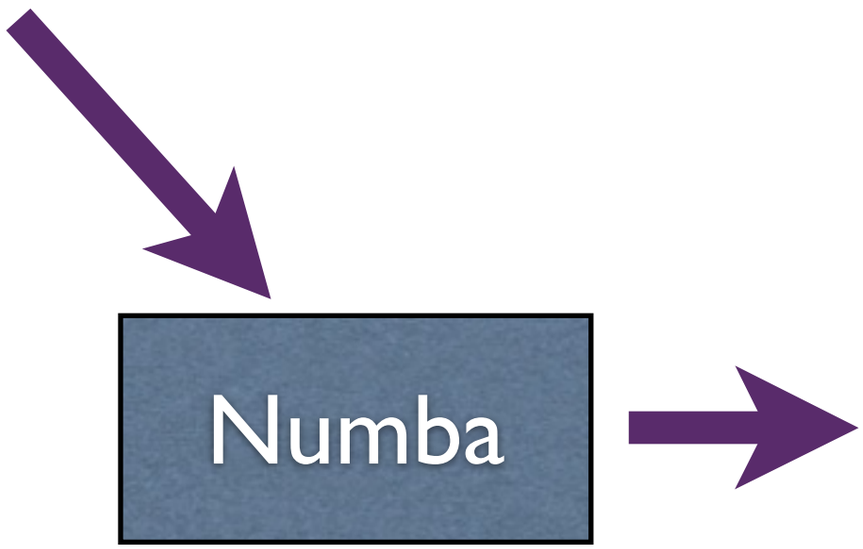
- with arguments --- provide type information (fastest to call at run-time)
- without arguments --- detects input types, infers output, generates code if needed, and dispatches (a little more run-time call overhead)

un-comment one of the 'jit' lines

```
#@jit('void(double[:,:], double, double)')
#@jit
def numba_update(u, dx2, dy2):
    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))
```

Example

```
@numba.jit('f8(f8)')
def sinc(x):
    if x==0.0:
        return 1.0
    else:
        return sin(x*pi)/(pi*x)
```

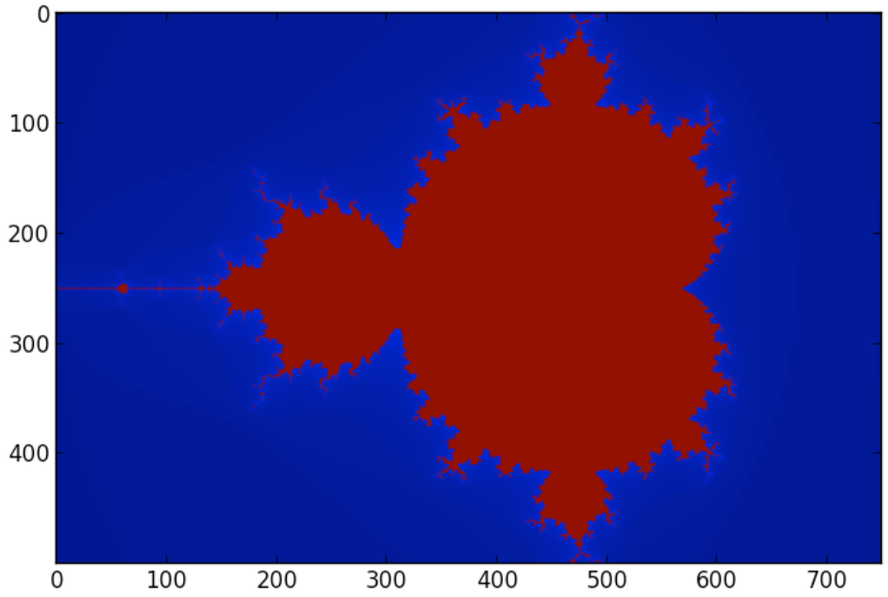



```
; ModuleID = 'sinc_mod_7b29370'

define double @sinc(double %x) {
Entry:
  %0 = fcmp oeq double %x, 0.000000e+00
  br i1 %0, label %BLOCK_12, label %BLOCK_16

BLOCK_12:                                         ; preds = %Entry
  ret double 1.000000e+00

BLOCK_16:                                         ; preds = %Entry
  %1 = fmul double %x, 0x400921FB54442D18
  %2 = call double @llvm.sin.f64(double %1)
  %3 = fmul double %x, 0x400921FB54442D18
  %4 = fdiv double %2, %3
  ret double %4

BLOCK_47:                                         ; No predecessors!
  ret double 0.000000e+00
}

declare double @llvm.sin.f64(double) nounwind readonly
```

~150x speed-up

```
@autojit
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255

@autojit
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image
```

Real-time image processing (50 fps Mandelbrot)




```
from numba import autojit
import numpy as np
from pylab import imshow, jet, show, ion
```

```
image = np.zeros((500, 750), dtype=np.uint8)
imshow(create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20))
```

Speeding up Math Expressions

```
@numba.autojit
def looped_ver(k, a):
    x = np.empty_like(a)
    x[0] = 0.0
    for i in range(1, x.size):
        sm = 0.0
        for j in range(0, i):
            sm += k[i-j,j] * a[i-j] * a[j]
        x[i] = sm
    return x
```

$$x_i = \sum_{j=0}^{i-1} k_{i-j,j} a_{i-j} a_j$$

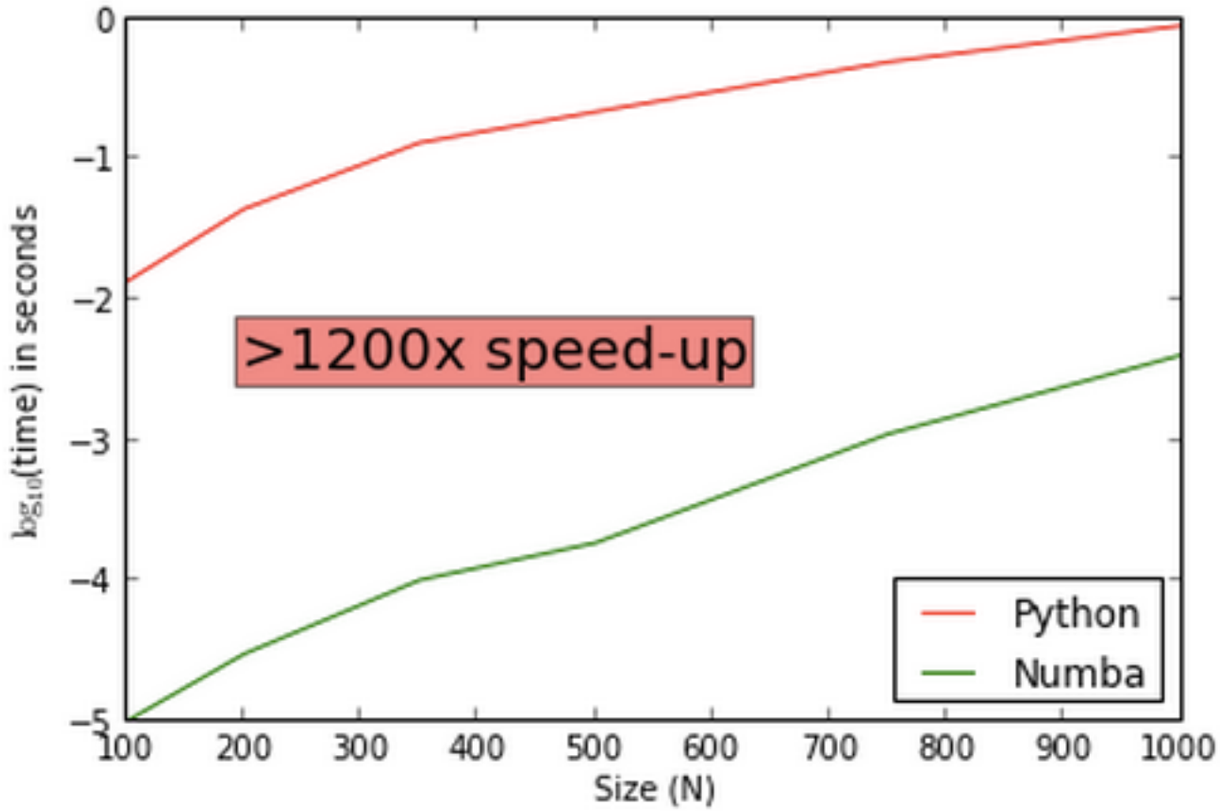
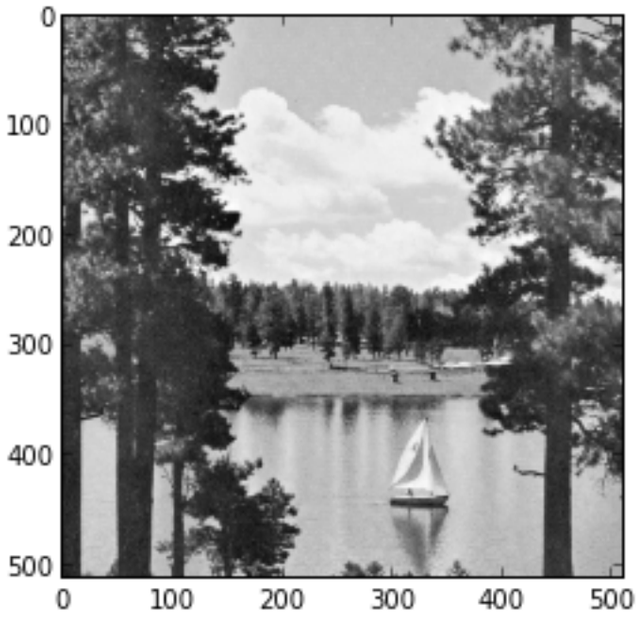
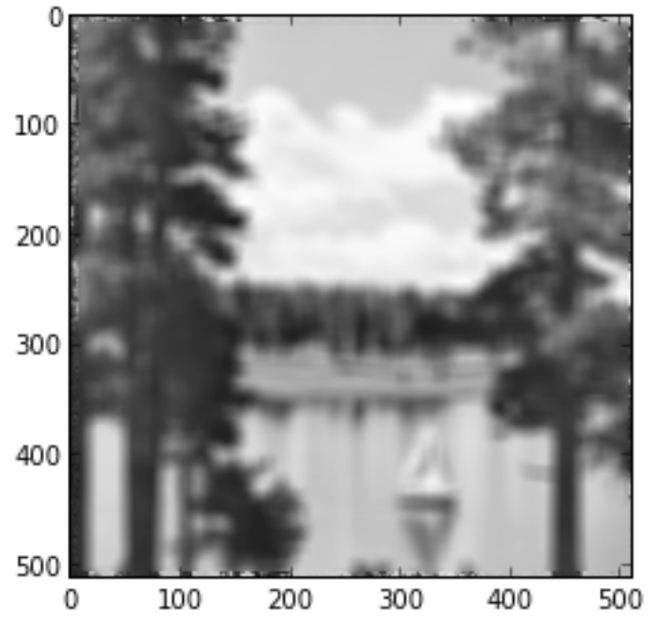


Image Processing

~800x speed-up






```
@jit('void(f8[:,:],f8[:,:],f8[:,:])')
def filter(image, filt, output):
    M, N = image.shape
    m, n = filt.shape
    for i in range(m//2, M-m//2):
        for j in range(n//2, N-n//2):
            result = 0.0
            for k in range(m):
                for l in range(n):
                    result += image[i+k-m//2,j+l-n//2]*filt[k, l]
            output[i,j] = result
```

Fast vectorize

NumPy's ufuncs take "kernels" and apply the kernel element-by-element over entire arrays

```
from numba import vectorize
from math import sin

@vectorize(['f8(f8)', 'f4(f4)'])
def sinc(x):
    if x==0.0:
        return 1.0
    else:
        return sin(x*pi)/(pi*x)
```

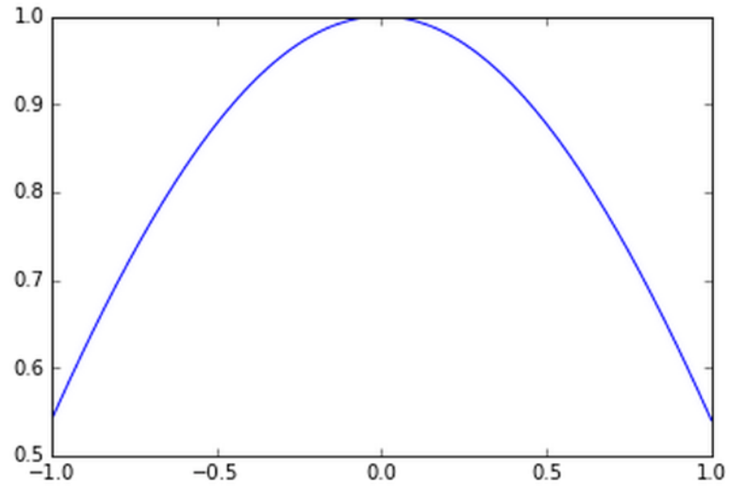
Write kernels in Python!

What is a Universal Function (ufunc)

- Ufuncs are the core of NumPy's calculation element-by-element infrastructure
- It's how +,-,*,/, **, sin, cos, etc. work
- It is not how linear-algebra calcs work (typically)

  — though see generalized ufuncs for a way.

```
In [9]: x = np.linspace(-1,1,100)
        y = np.cos(x)
        plot(x,y)

Out[9]: [<matplotlib.lines.Line2D at 0x10ee6e650>]
```



Creating a "Ufunc"

- Numba is the best way to make new ufuncs for working with NumPy arrays

In [1]:
```
import numpy as np
import numba
```

In [2]:
```
@numba.vectorize(['float64(float64, float64)'])
def fractional_difference(a, b):
    return 2 * (a - b) / (a + b)
```

In [3]:
```
x = np.arange(10000, dtype=np.float64) + 1
y = np.arange(10000, dtype=np.float64) + 1.1
```

In [4]:
```
%timeit 2 * (x - y) / (x + y)          # Standard numpy
%timeit fractional_difference(x, y)    # Numba
```

```
10000 loops, best of 3: 48.5 µs per loop
10000 loops, best of 3: 23.6 µs per loop
```

Case-study -- j0 from scipy.special

- scipy.special was one of the first libraries I wrote
- extended "umath" module by adding new "universal functions" to compute many scientific functions by wrapping C and Fortran libs.
- Bessel functions are solutions to a differential equation:

$$x^2 \frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \alpha^2)y = 0$$

$$y = J_\alpha(x)$$

$$J_n(x) = \frac{1}{\pi}\int_0^\pi \cos(n\tau - x\sin(\tau))\, d\tau$$

scipy.special.j0 wraps cephes algorithm

```
@jit('f8(f8)')
def j0(x):
    if (x < 0):
        x = -x

    if (x <= 5.0):
        z = x * x
        if (x < 1.0e-5):
            return (1.0 - z / 4.0)
        p = (z-DR1) * (z-DR2)
        p = p * polevl(z, RP) / polevl(z, RQ)
        return p

    w = 5.0 / x
    q = 25.0 / (x*x)
    p = polevl(q, PP) / polevl(q, PQ)
    q = polevl(q, QP) / p1evl(q, QQ)
    xn = x - NPY_PI_4
    p = p*math.cos(xn) - w * q * math.sin(xn)
    return p * SQ2OPI / math.sqrt(x)

vj0 = vectorize(['f8(f8)'])(j0)
```

```
@jit('f8(f8,f8[:])')
def polevl(x, coef):
    N = len(coef)
    ans = coef[0]
    i = 1
    while i < N:
        ans = ans * x + coef[i]
        i += 1
    return ans

@jit('f8(f8,f8[:])')
def p1evl(x, coef):
    N = len(coef)
    ans = x + coef[0]
    i = 1
    while i < N:
        ans = ans * x + coef[i]
        i += 1
    return ans
```

```
DR1 = 5.783185962946784521175995758455807035071
DR2 = 30.471262343662086399078163175022755848 42

NPY_PI_4 = .7853981633974483096
SQ2OPI   = .7978845608028653558798
```

```
QP = np.array([
-1.1366383889846914 9931E-2,
-1.2825271867050931 8512E0,
-1.9553954425773597 2385E1,
-9.3206015212376823 1369E1,
-1.7768116798048805 0595E2,
-1.4707750515495117 0175E2,
-5.1410532676659933 0220E1,
-6.0501435060072848 1186E0], 'd')

QQ = np.array([
   # 1.0000000000000000000E0,
  6.4317825611817802 3184E1,
  8.5643002597698058 7198E2,
  3.8824018360540160 9683E3,
  7.2404677419565247 8189E3,
  5.9307270118731698 4827E3,
  2.0620933166032784 7417E3,
  2.4200574024029139 3179E2], 'd')

PP = np.array([
  7.9693672929734705 1624E-4,
  8.2835239210744079 9803E-2,
  1.2395337164641429 9388E0,
  5.4472500305876877 5090E0,
  8.7471650019981701 1941E0,
  5.3032403823539489 2183E0,
  9.9999999999999997 821E-1], 'd')

PQ = np.array([
  9.2440881055886363 7013E-4,
  8.5628847435447443 1428E-2,
  1.2535274390105895 3537E0,
  5.4709774033041710 5182E0,
  8.7619088323706959 4232E0,
  5.3060528823539461 7618E0,
  1.0000000000000000 218E0], 'd')

RP = np.array([
-4.7944322097820177 3821E9,
 1.9561749194655657 7543E12,
-2.4924834436096771 6204E14,
 9.7086225104730632 3952E15], 'd')

RQ = np.array([
   # 1.0000000000000000000E0,
 4.9956314715265101 7219E2,
 1.7378540167637468 3123E5,
 4.8440965833996204 5305E7,
 1.1185553704535683 4862E10,
 2.1127752011548921 7587E12,
 3.1051822985742258 3814E14,
 3.1812195594320494 3306E16,
 1.7108629408104313 6091E18], 'd')
```

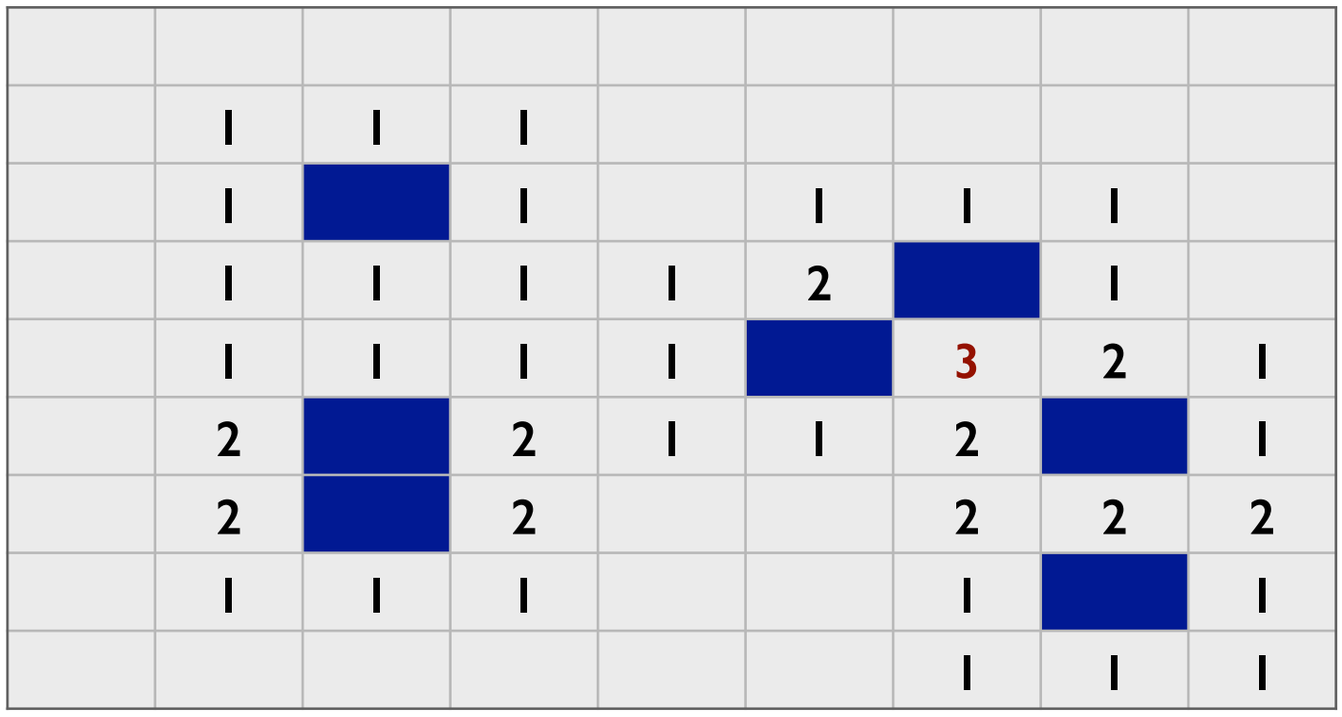
Result --- equivalent to compiled code

```
In [6]: %timeit vj0(x)
10000 loops, best of 3: 75 us per loop

In [7]: from scipy.special import j0

In [8]: %timeit j0(x)
10000 loops, best of 3: 75.3 us per loop
```

But! Now code is in Python and can be experimented with more easily (and moved to the GPU / accelerator more easily)!

Compiling a Function

- Sometimes you can't create a simple or efficient array expression or ufunc. Use Numba to work with array elements directly.
- Example: Suppose you have a boolean grid and you want to find the maximum number neighbors a cell has in the grid:



Compiling a Function

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

In [10]:
```
@numba.jit
def check_neighbor(grid, i, j):
    if 0 <= i < grid.shape[0] and 0 <= j < grid.shape[1]:
        return grid[i, j]
    else:
        return False

@numba.jit
def find_max_neighbors(grid):
    max_neighbors = 0
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            neighbor_count = 0
            for i_offset in -1, 0, 1:
                for j_offset in -1, 0, 1:
                    if i_offset == 0 and j_offset == 0:
                        continue
                    elif check_neighbor(grid, i + i_offset, j + j_offset):
                        neighbor_count += 1
            max_neighbors = max(max_neighbors, neighbor_count)
    return max_neighbors
```

Compiling a Function

- Very hard to express this calculation purely as array operations (and even if you do, it is likely unreadable to non-NumPy experts).
- Numba let's you write out the loops, but avoid the penalty for having to loop over individual elements in

```
In [20]: grid = (np.random.uniform(size=100*100) > 0.95).reshape((100,100))
         %timeit py_find_max_neighbors(grid)
         %timeit find_max_neighbors(grid)

         10 loops, best of 3: 71.9 ms per loop
         1000 loops, best of 3: 424 µs per loop
```

169x faster!

Laplace Example

Adapted from http://www.scipy.org/PerformancePython originally by Prabhu Ramachandran

```
@jit('void(double[:,:], double, double)')
def numba_update(u, dx2, dy2):
    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))
```

```
@jit('void(double[:,:], double, double)')
def numbavec_update(u, dx2, dy2):
    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))
```

Results of Laplace example

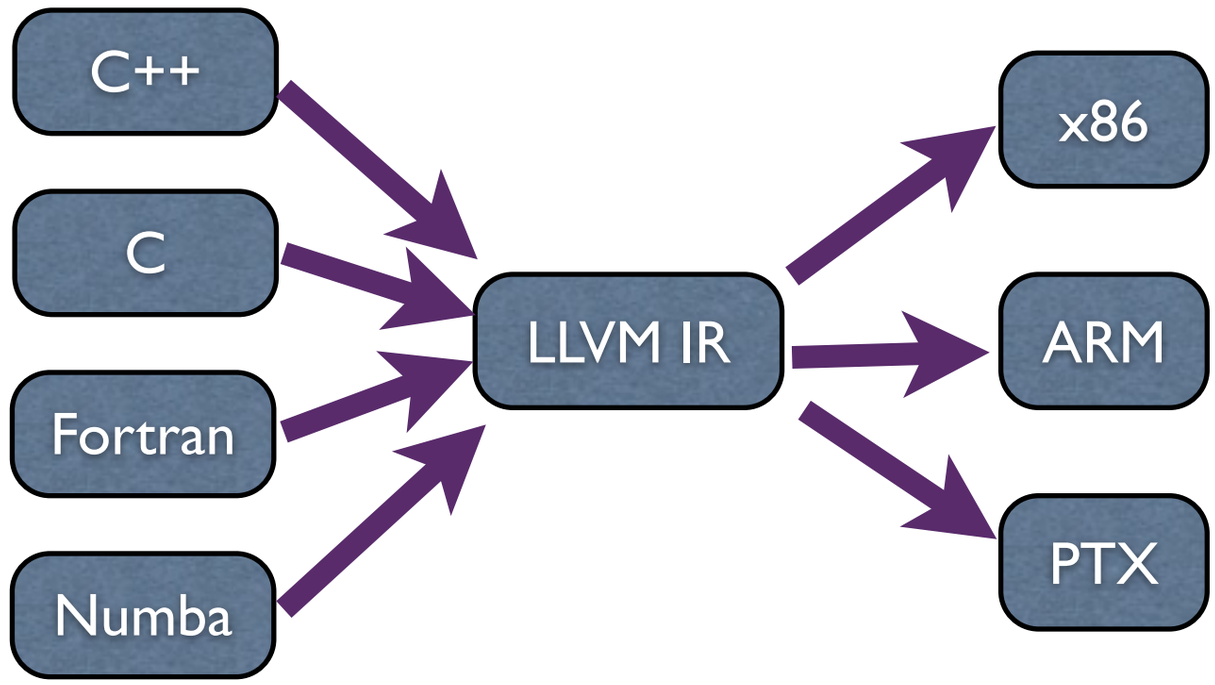
https://github.com/teoliphant/speed.git

Version	Time	Speed Up
NumPy	3.19	1
Numba	2.32	1.38
Vect. Numba	2.33	1.37
Cython	2.38	1.34
Weave	2.47	1.29
Numexpr	2.62	1.22
Fortran Loops	2.3	1.39
Vect. Fortran	1.5	2.13

Numba can change the game!

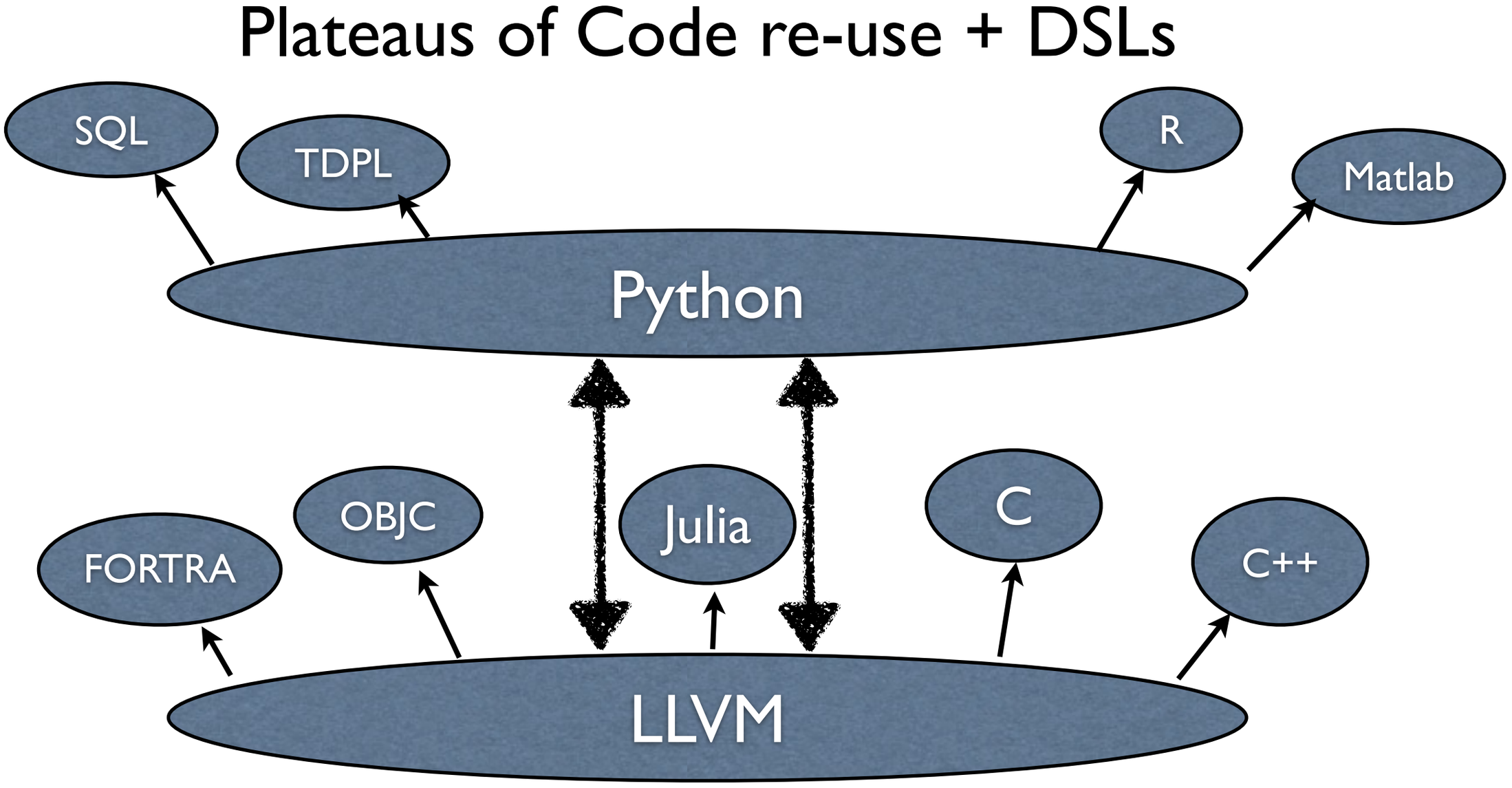
```
// mean(vector)
template<typename P_numtype>
inline
BZ_FLOATTYPE(BZ_SUMTYPE(P_numtype)) mean(const Vector<P_numtype>& x)
{
    BZPRECONDITION(x.length() > 0);

    typedef BZ_FLOATTYPE(BZ_SUMTYPE(P_numtype)) T_floattype;
    return _bz_vec_sum(x._bz_asVecExpr()) / (T_floattype) x.length();
}
```

```
@autojit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result
```



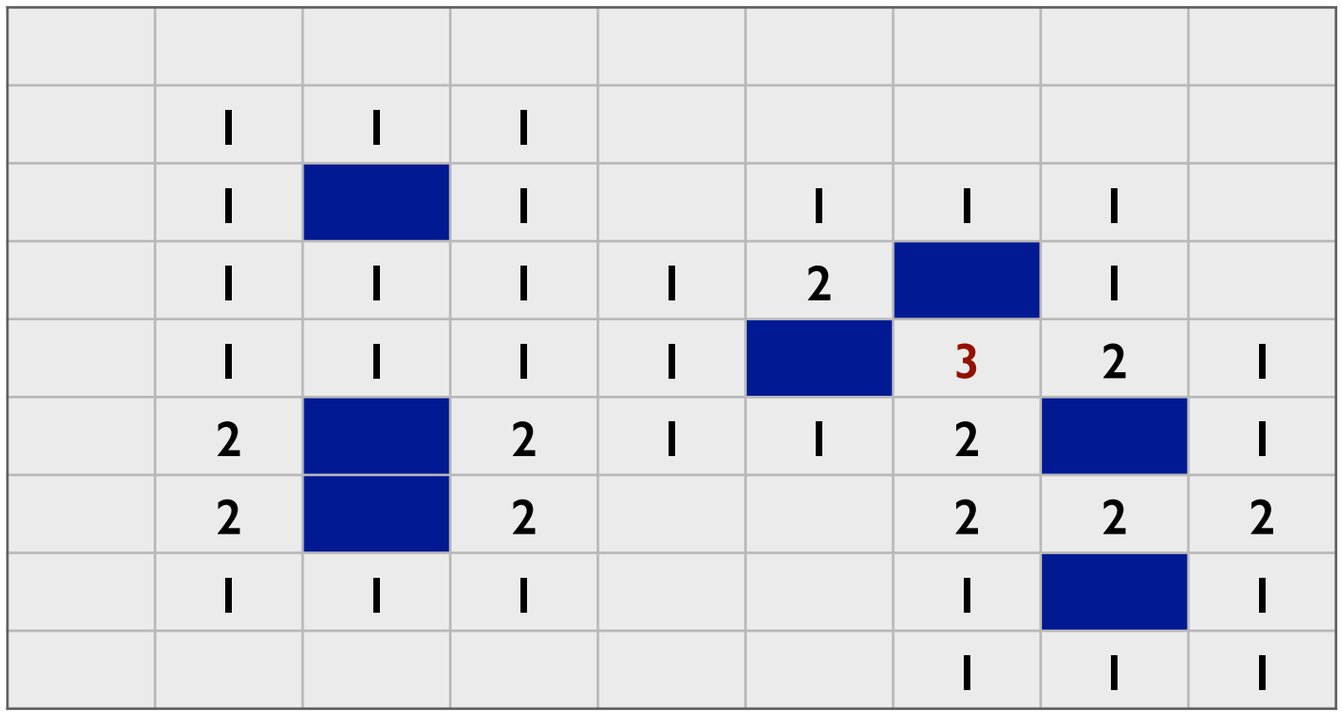
Numba turns Python into a “compiled language” (but much more flexible). You don’t have to reach for C/C++

Software Stack Future?



Compiling a Function

- Sometimes you can't create a simple or efficient array expression or ufunc. Use Numba to work with array elements directly.
- Example: Suppose you have a boolean grid and you want to find the maximum number neighbors a cell has in the grid:



Compiling a Function

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

In [10]:
```
@numba.jit
def check_neighbor(grid, i, j):
    if 0 <= i < grid.shape[0] and 0 <= j < grid.shape[1]:
        return grid[i, j]
    else:
        return False

@numba.jit
def find_max_neighbors(grid):
    max_neighbors = 0
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            neighbor_count = 0
            for i_offset in -1, 0, 1:
                for j_offset in -1, 0, 1:
                    if i_offset == 0 and j_offset == 0:
                        continue
                    elif check_neighbor(grid, i + i_offset, j + j_offset):
                        neighbor_count += 1
            max_neighbors = max(max_neighbors, neighbor_count)
    return max_neighbors
```

Compiling a Function

- Very hard to express this calculation purely as array operations (and even if you do, it is likely unreadable to non-NumPy experts).
- Numba let's you write out the loops, but avoid the penalty for having to loop over individual elements in

```
In [20]: grid = (np.random.uniform(size=100*100) > 0.95).reshape((100,100))
         %timeit py_find_max_neighbors(grid)
         %timeit find_max_neighbors(grid)

         10 loops, best of 3: 71.9 ms per loop
         1000 loops, best of 3: 424 µs per loop
```

169x faster!

Different Modes of Compilation

- Numba *automatically* selects between two different levels of optimization when compiling a function:
  - "object mode": supports nearly all of Python, but generally cannot speed up code by a large factor (exception: see next slide)
  - "nopython mode": supports a subset of Python, but runs at C/C++/FORTRAN speeds

Loop-Lifting

- In object mode, Numba will attempt to extract loops and compile them in nopython mode.
- Works great for functions that are bookended by uncompilable code, but have a compilable core loop.
- All happens automatically.

Loop-Lifting

In [1]:
```
import numpy as np
import numba
```

In [2]:
```
@numba.jit
def select_in_interval(a, lower, upper):
    output_buffer = np.empty_like(a)
    next_index = 0

    for element in a:
        if lower < element < upper:
            output_buffer[next_index] = element
            next_index += 1

    return output_buffer[:next_index]
```

object mode

nopython mode

object mode

In [3]:
```
x = np.random.uniform(size=100000)
%timeit x[(0.1 < x) & (x < 0.9)]
%timeit select_in_interval(x, 0.1, 0.9)
```

```
1000 loops, best of 3: 551 µs per loop
1000 loops, best of 3: 299 µs per loop
```

nopython Mode Features

- Standard control and looping structures: if, else, while, for, range
- NumPy arrays, int, float, complex, booleans, and tuples
- Almost all arithmetic, logical, and bitwise operators as well as functions from the `math` and `numpy` modules
- Nearly all NumPy dtypes: int, float, complex, datetime64, timedelta64
- Array element access (read and write)
- Array reduction functions: sum, prod, max, min, etc
- Calling other nopython mode compiled functions
- Calling ctypes or cffi-wrapped external functions

Compiling for the GPU

GPU functions are called differently, but it is still Python!

In [1]:
```
import numpy as np
from numba import cuda
import math
```

In [2]:
```
@cuda.jit
def gpu_cos(a, out):
    i = cuda.grid(1)
    if i < a.shape[0]:
        out[i] = math.cos(a[i])
```

In [3]:
```
x = np.linspace(0, 2 * np.pi, 5000000, dtype=np.float32)
gpu_out = np.empty_like(x)
cpu_out = np.empty_like(x)

thread_config = (len(x)//128 + 1, 128)
```

In [5]:
```
%timeit np.cos(x, cpu_out)                          # Standard numpy
%timeit gpu_cos[thread_config](x, gpu_out)  # Numba using the GPU
```

```
10 loops, best of 3: 35.6 ms per loop
10 loops, best of 3: 18.3 ms per loop
```

MacBook Pro w/ GTX 650M GPU

Example of CUDA Python

```
from numbapro import cuda
from numba import autojit

@autojit(target='gpu')
def array_scale(src, dst, scale):
    tid = cuda.threadIdx.x
    blkid = cuda.blockIdx.x
    blkdim = cuda.blockDim.x

    i = tid + blkid * blkdim

    if i >= n:
        return

    dst[i] = src[i] * scale

src = np.arange(N, dtype=np.float)
dst = np.empty_like(src)

array_scale[grid, block](src, dst, 5.0)
```

CUDA Development directly in Python

Matrix Multiply

```
@cuda.jit(argtypes=[f4[:,:], f4[:,:], f4[:,:]])
def cu_square_matrix_mul(A, B, C):
    sA = cuda.shared.array(shape=(tpb, tpb), dtype=f4)
    sB = cuda.shared.array(shape=(tpb, tpb), dtype=f4)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    bw = cuda.blockDim.x
    bh = cuda.blockDim.y

    x = tx + bx * bw
    y = ty + by * bh

    acc = 0.
    for i in range(bpg):
        if x < n and y < n:
            sA[ty, tx] = A[y, tx + i * tpb]
            sB[ty, tx] = B[ty + i * tpb, x]

        cuda.syncthreads()

        if x < n and y < n:
            for j in range(tpb):
                acc += sA[ty, j] * sB[j, tx]

        cuda.syncthreads()

    if x < n and y < n:
        C[y, x] = acc
```

```
bpg = 50
tpb = 32
n = bpg * tpb

A = np.array(np.random.random((n, n)),
             dtype=np.float32)
B = np.array(np.random.random((n, n)),
             dtype=np.float32)
C = np.empty_like(A)

stream = cuda.stream()
with stream.auto_synchronize():
    dA = cuda.to_device(A, stream)
    dB = cuda.to_device(B, stream)
    dC = cuda.to_device(C, stream)
    cu_square_matrix_mul[(bpg, bpg),
                         (tpb, tpb),
                         stream](dA, dB, dC)
    dC.to_host(stream)
```

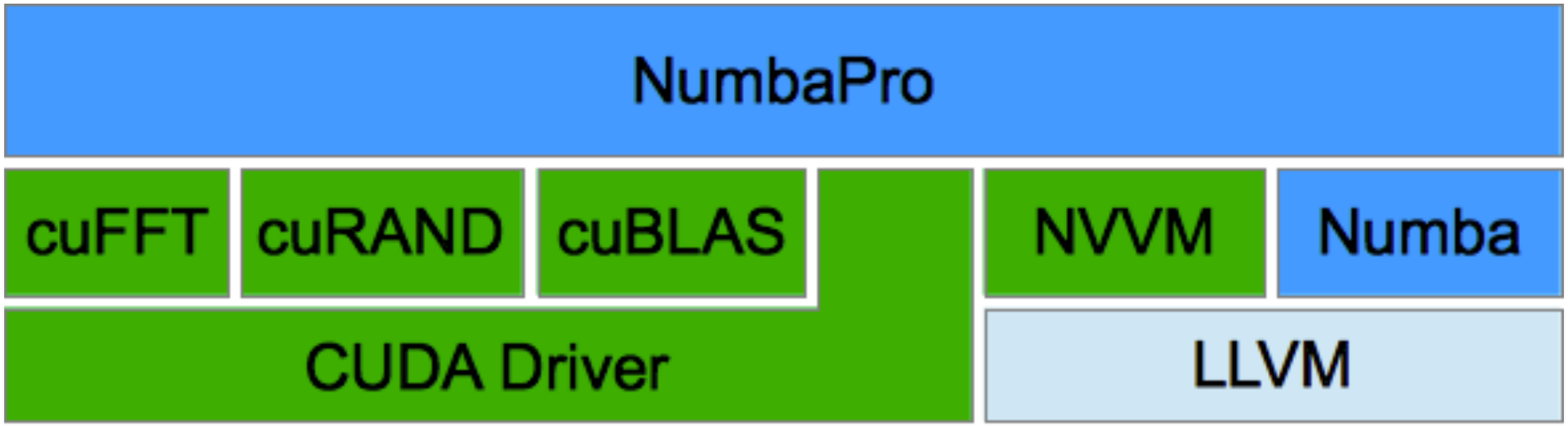
Advanced Use Cases

- Compile new functions based on user input.
  - Great for inserting a user-provided math expression into a larger algorithm, while still achieving C speeds.
- Optimization (least squares, etc) libraries that can recompile themselves to inline a specific objective function right into the algorithm
- Multithreaded calculation without having to worry about the global interpreter lock (GIL).

NumbaPro (part of Anaconda Accelerate)

- NumbaPro adds higher level features on top of Numba:
  - Create ufuncs that run multithreaded on the CPU or on the GPU
  - GPU linear algebra
  - GPU FFTs

Examples




```
from numbapro import vectorize
from math import sin

@vectorize(['f8(f8)', 'f4(f4)'], target='gpu')
def sinc(x):
   if x==0.0:
        return 1.0
    else:
        return sin(x*pi)/(pi*x)

@vectorize(['f8(f8)', 'f4(f4)'], target='parallel')
def sinc2(x):
   if x==0.0:
        return 1.0
    else:
        return sin(x*pi)/(pi*x)
```

Example

```
from numbapro import vectorize

sig = 'uint8(uint32, f4, f4, f4, f4, uint32, uint32, uint32)'

@vectorize([sig], target='gpu')
def mandel(tid, min_x, max_x, min_y, max_y, width, height, iters):
    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height

    x = tid % width
    y = tid / width

    real = min_x + x * pixel_size_x
    imag = min_y + y * pixel_size_y

    c = complex(real, imag)
    z = 0.0j

    for i in range(iters):
        z = z * z + c
        if (z.real * z.real + z.imag * z.imag) >= 4:
            return i
    return 255
```

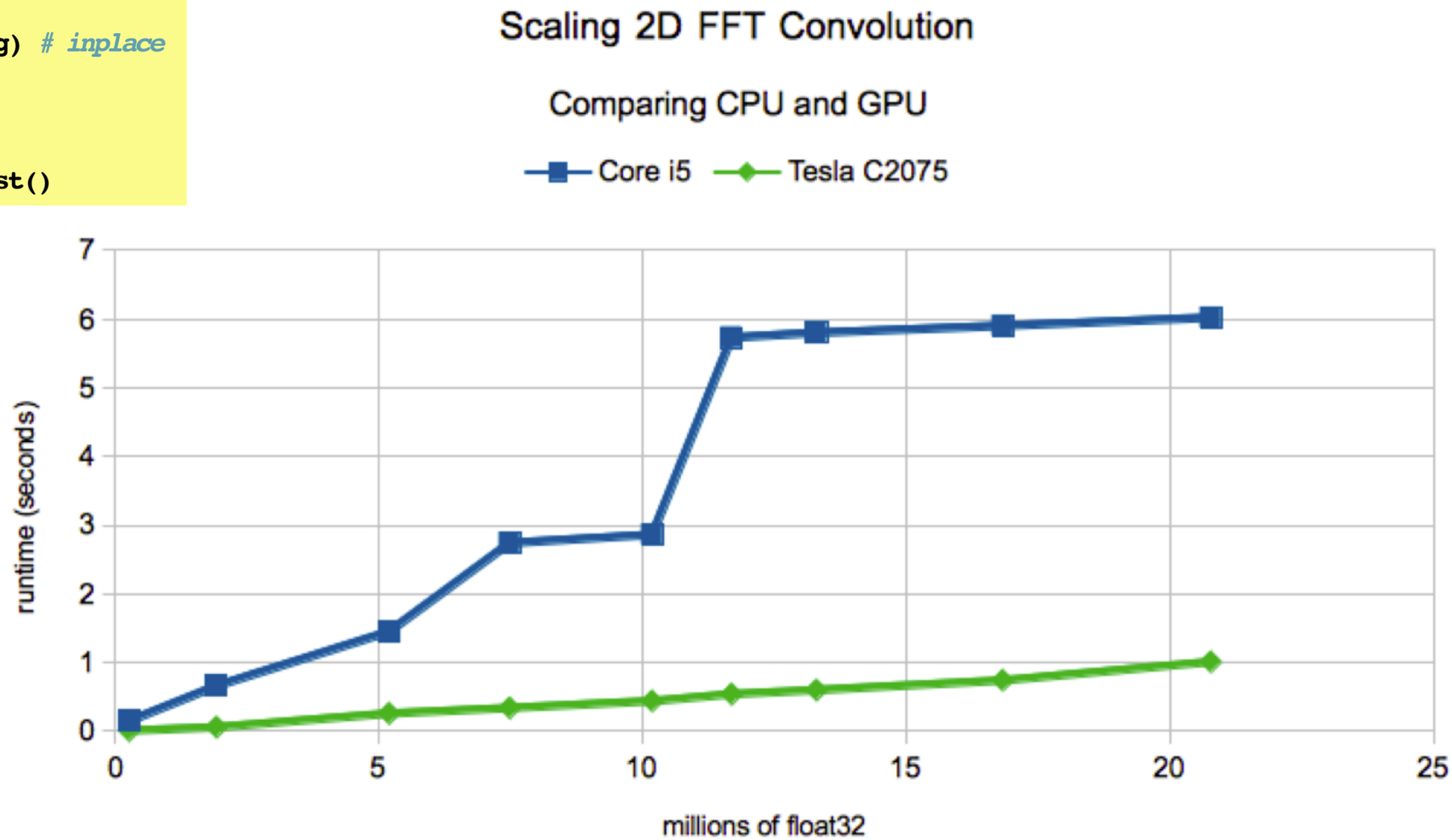
Tesla S2050

Kind	Time	Speed-up
Python	263.6	1.0x
CPU	2.639	100x
GPU	0.1676	1573x

Example with cuFFT

```
# host -> device
d_img = cuda.to_device(img)      # image
d_fltr = cuda.to_device(fltr)    # filter
# FFT forward
cufft.fft_inplace(d_img)
cufft.fft_inplace(d_fltr)
# multply
vmult(d_img, d_fltr, out=d_img) # inplace
# FFT inverse
cufft.ifft_inplace(d_img)
# device -> host
filted_img = d_img.copy_to_host()
```

```
@vectorize(['complex64(complex64, complex64)'],
           target='gpu')
def vmult(a, b):
    return a * b
```



Conclusion

- Numba is a JIT compiler than understands Python!
- Achieve the same speeds as compiled languages for numerical and array-processing code.
- Can be used to create advanced workflows where user input drives compilation at runtime.
- Open source, available at:
  http://numba.pydata.org/
- Or:
  `conda install numba`