Numba: An array-oriented Python compiler

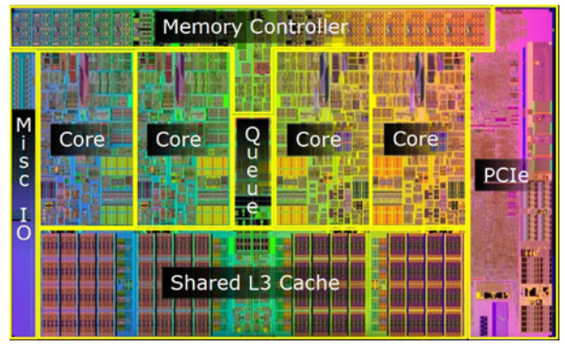
Python Beyond the CPU Tutorial

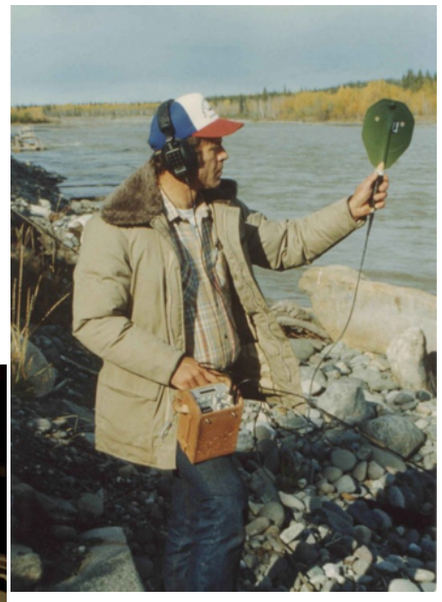
Travis E. Oliphant

March 13, 2013

Big Picture

Empower domain experts with high-level tools that exploit modern hard-ware

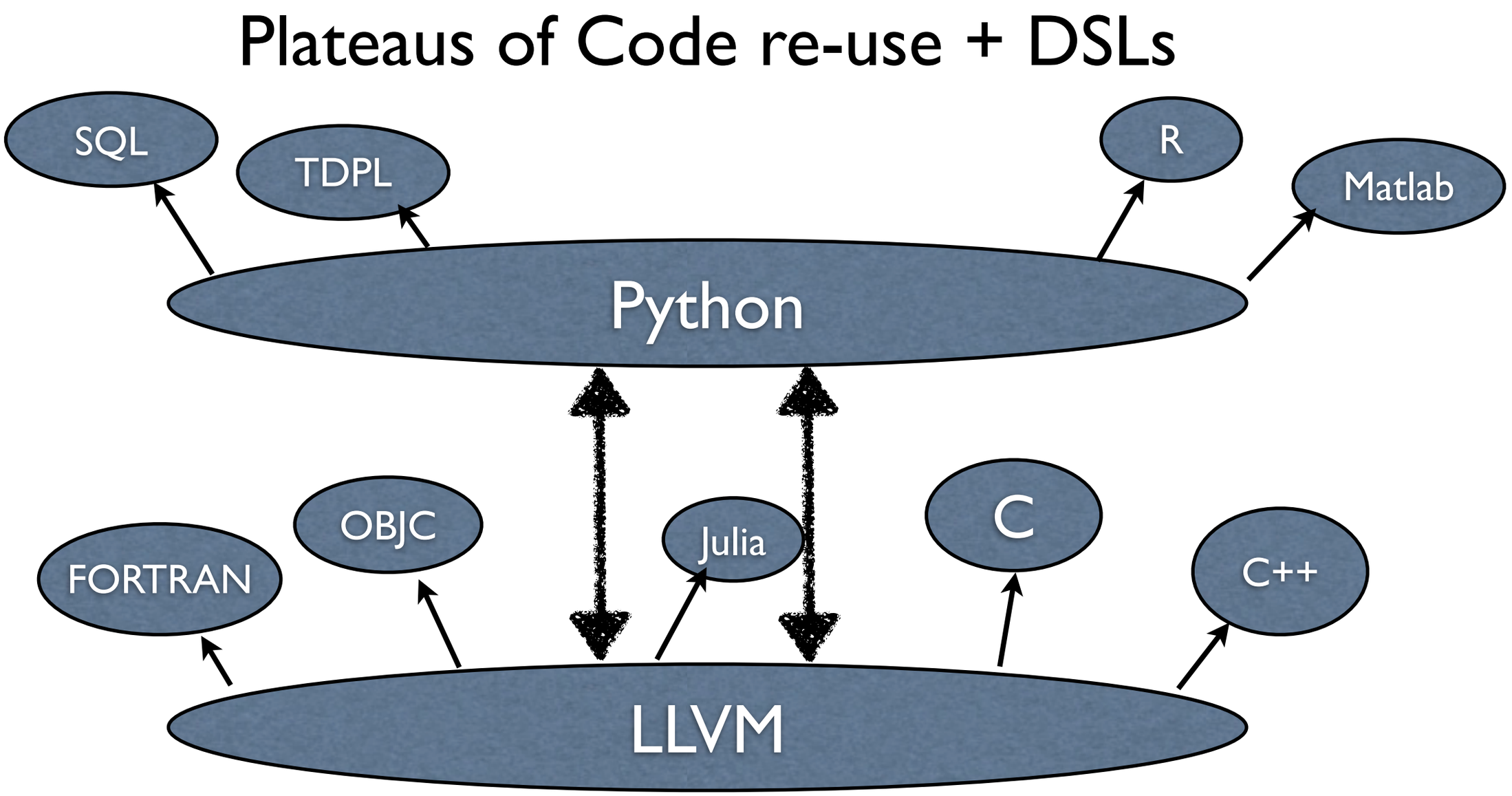




Array Oriented Computing

CONTINUUM ANALYTICS

Software Stack Future?



CONTINUUM ANALYTICS

Motivation

- Python is great for rapid development and high-level thinking-in-code
- It is slow for interior loops because lack of type information leads to a lot of indirection and "extra" code.

Motivation

- NumPy users have had a lot of type information for a long time --- but only currently have one-size fits all pre-compiled, vectorized loops.
- Idea is to use this type information to allow compilation of arbitrary expressions involving NumPy arrays

Current approaches

- Cython
- Weave
- Write fast-code in C/C++/Fortran and "wrap" with
  - SWIG or Cython
  - f2py or fwrap
  - hand-written

Numba philosophy : don't wrap or rewrite
--- just decorate

Simple API

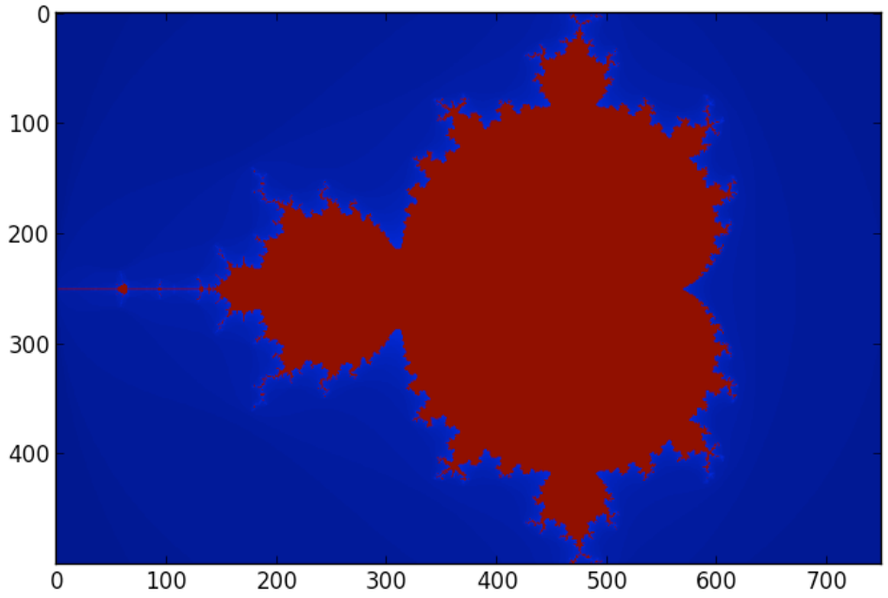
- jit --- provide type information (fastest to call)
- autojit --- detects input types, infers output, generates code if needed, and dispatches (a little more expensive to call)

```
@jit('void(double[:,:], double, double)')
#@autojit
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))
```

~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 in Python (50 fps Mandelbrot)




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

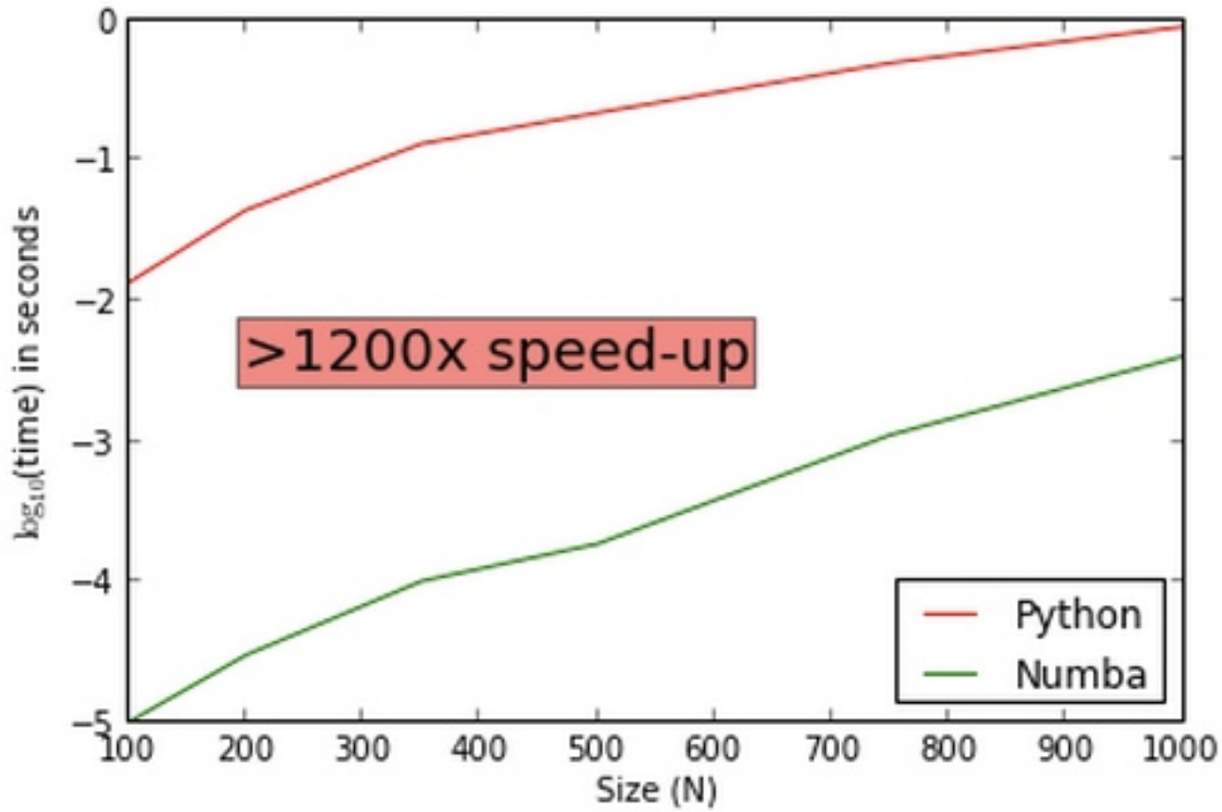
```
image = np.zeros((500, 750), dtype=np.uint8)
imshow(create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20))
```

CONTINUUM ANALYTICS

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
```

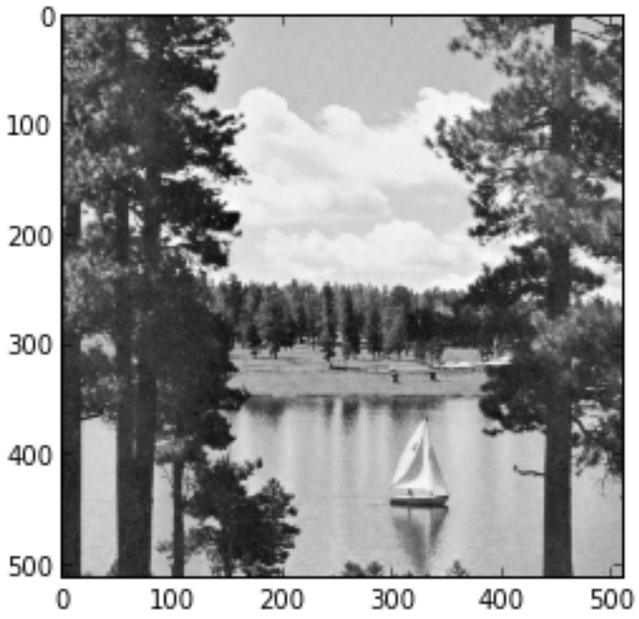
$$x_i = \sum_{j=0}^{i-1} k_{i-j} a_{i-j} a_j$$

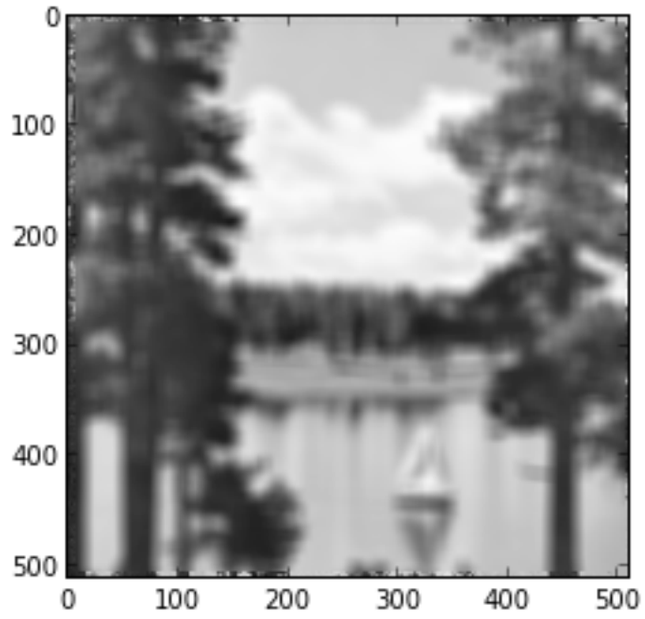


CONTINUUM ANALYTICS

Image Processing

~1500x speed-up

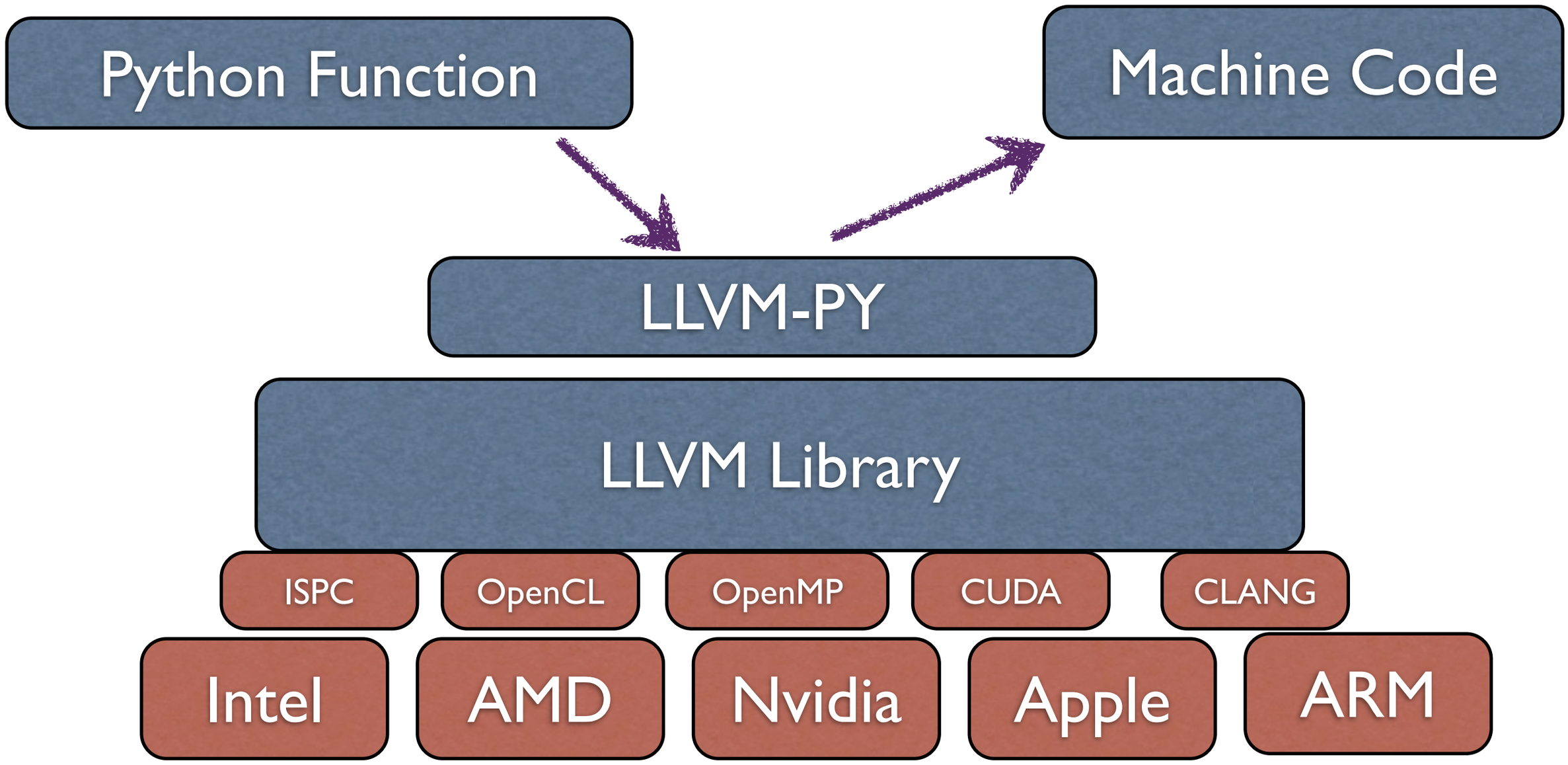





```
@jit('void(f8[:,:],f8[:,:],f8[:,:])')
def filter(image, filt, output):
    M, N = image.shape
    m, n = filt.shape
    for i in range(m//2, M-m//2):
        for j in range(n//2, N-n//2):
            result = 0.0
            for k in range(m):
                for l in range(n):
                 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

CONTINUUM ANALYTICS

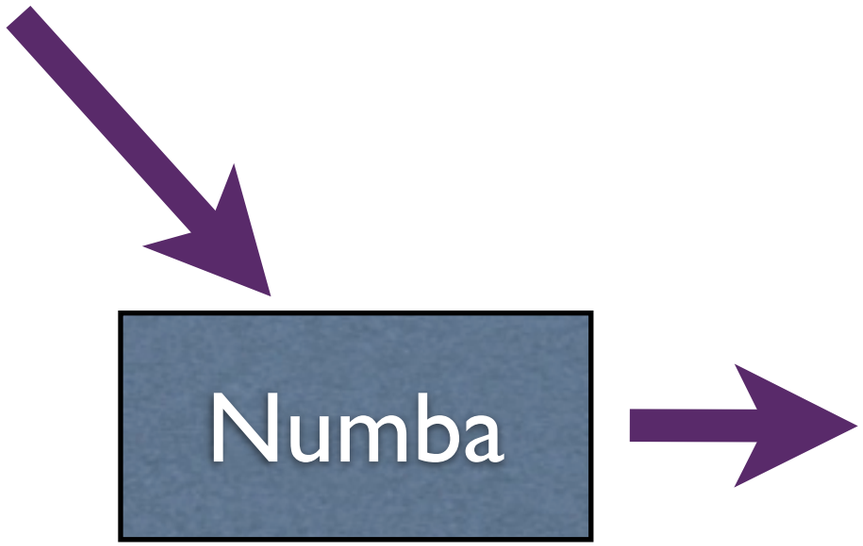
NumPy + Mamba = Numba



CONTINUUM ANALYTICS

Example

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



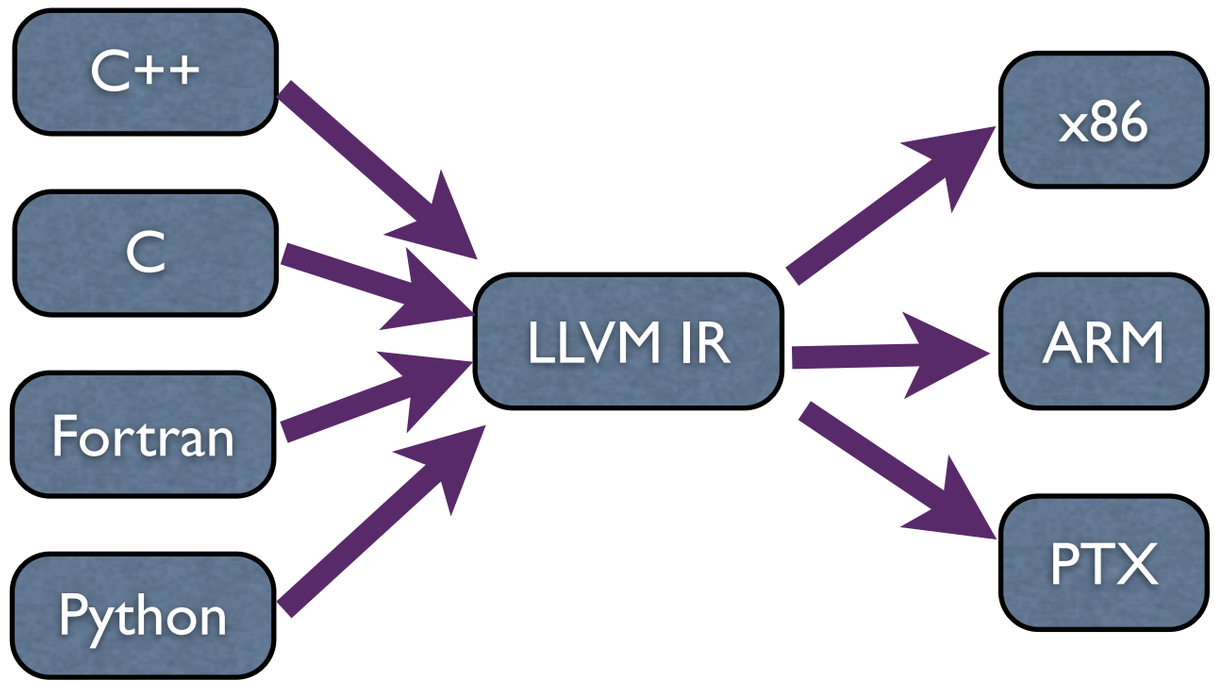

```
; ModuleID = 'sinc_mod_7b29370'

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
```

CONTINUUM ANALYTICS

Compiler Overview

```
// 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)

Compile NumPy array expressions

```
~~import numbapro~~
from numba import autojit

@autojit
def formula(a, b, c):
    a[1:,1:] = a[1:,1:] + b[1:,:-1] + c[1:,:-1]


@autojit
def express(m1, m2):
    m2[1:-1:2,0,...,::2] = (m1[1:-1:2,...,::2] *
                            m1[-2:1:-2,...,::2])
    return m2
```

Fast vectorize

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

```
from numbapro 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!

Updated Laplace Example

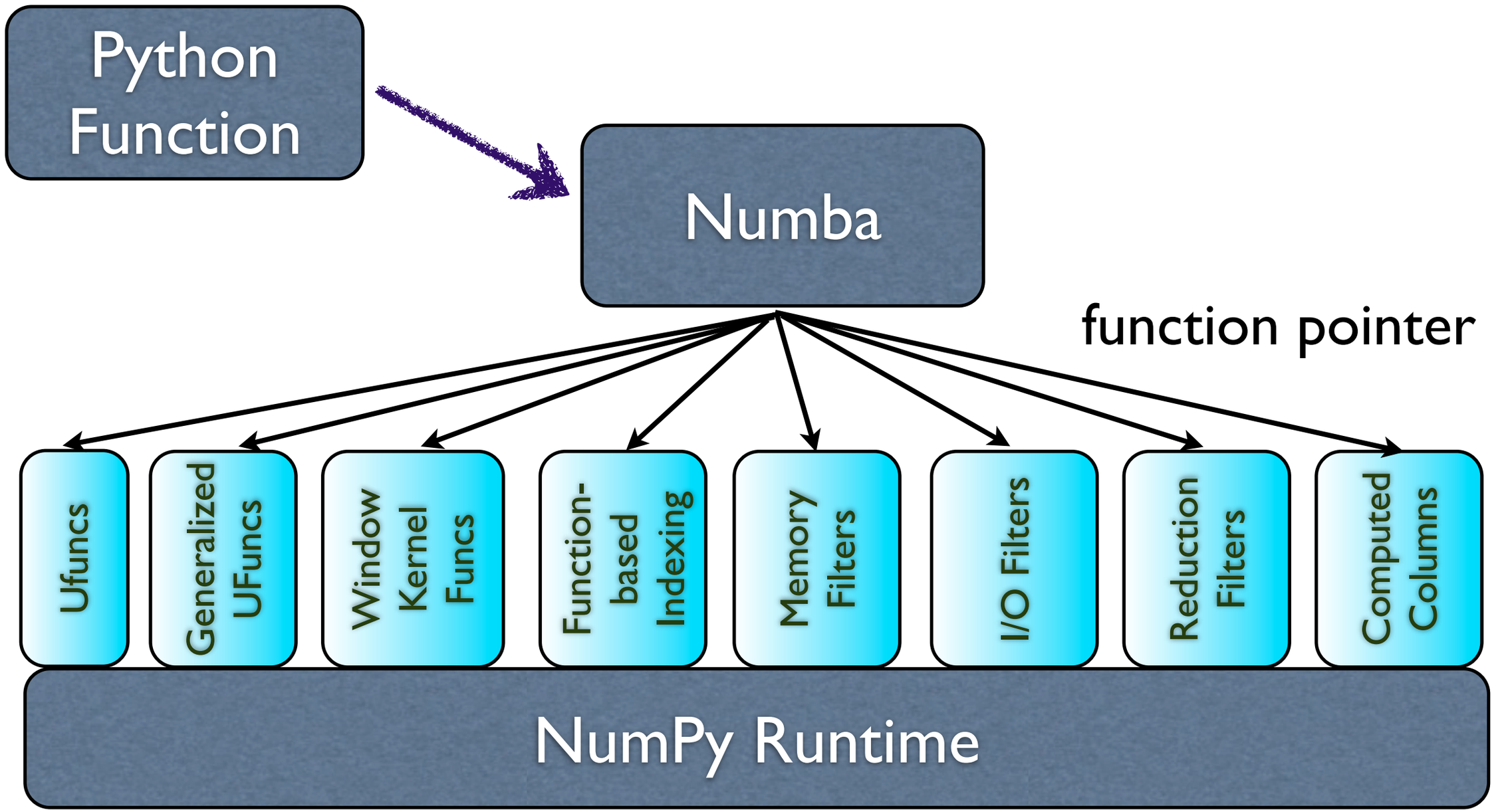
https://github.com/teoliphant/speed.git

Version	Time	Speed Up
NumPy	3.19	1.0
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	2.30	1.39
Vect. Fortran	1.50	2.13

Many Advanced Features

- Extension classes (jit a class)
- Struct support (NumPy arrays can be structs)
- SSA --- can refer to local variables as different types
- Typed lists and typed dictionaries coming
- pointer support
- calling ctypes and CFFI functions natively
- pycc (create stand-alone dynamic library and executable)
- pycc --python (create static extension module for Python)

Uses of Numba



CONTINUUM ANALYTICS

Status

- Main team sponsored by Continuum Analytics and composed of:
  - Travis Oliphant (NumPy, SciPy)
  - Jon Riehl (Mython, PyFront, Basil, ...)
  - Mark Florrison (minivect, Cython)
  - Siu Kwan Lam (pymothoa, llvmpy)
- Rapid progress this year
- Version 0.6 released first of February
- Version 0.7 next week
- Version 0.8 first of April
- Stable API (jit, autojit) easy to use
- Full Python support by 1.0 at end of summer
- Should be able to write equivalent of NumPy and SciPy with Numba

numba.pydata.org

CONTINUUM ANALYTICS

Introducing NumbaPro

fast development and fast execution!

Python and NumPy compiled to Parallel Architectures (GPUs and multi-core machines)

- Create parallel-for loops
- Parallel execution of ufuncs
- Run ufuncs on the GPU
- Write CUDA directly in Python!
- Free for Academics

CONTINUUM ANALYTICS

Create parallel-for loops

```
import numbapro # import first to make prange available
from numba import autojit, prange

@autojit
def parallel_sum2d(a):
    sum = 0.0
    for i in prange(a.shape[0]):
        for j in range(a.shape[1]):
            sum += a[i,j]
```

Ufuncs in parallel (multi-core or GPU)

```
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)
```

Introducing 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

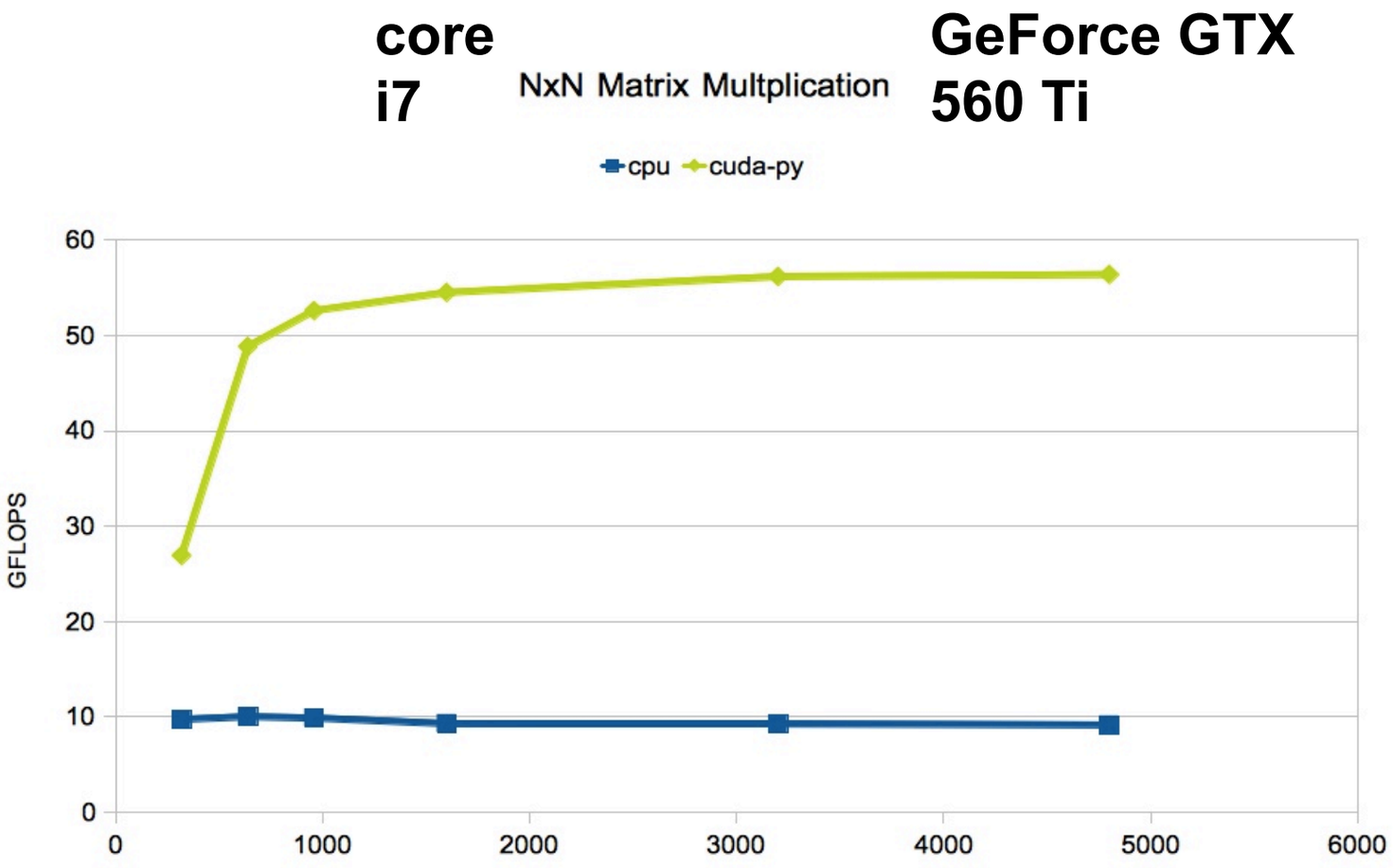
using Python syntax!

Example: 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)
```

Early Performance Results



Already about 6x faster on the GPU.

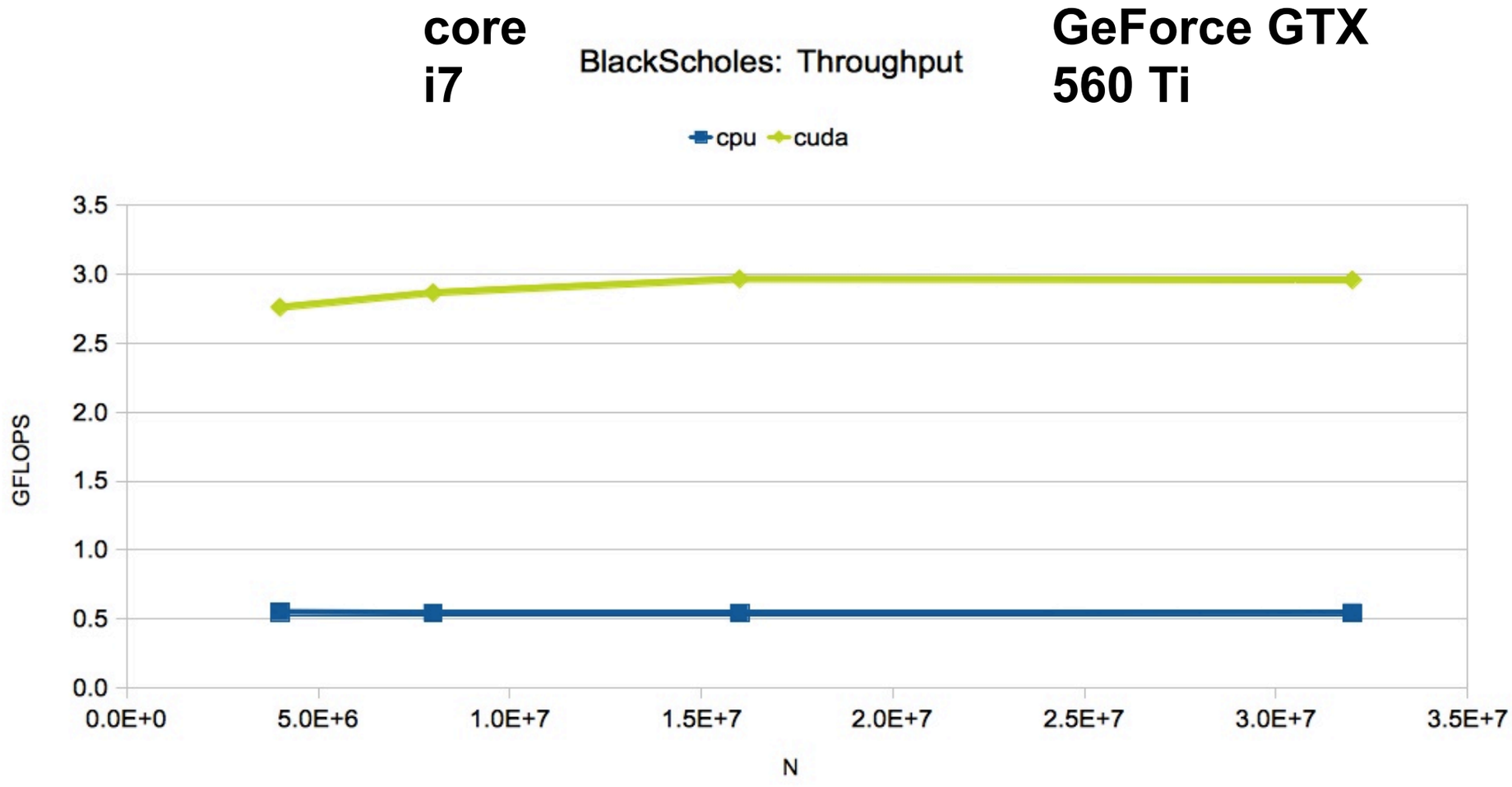
Example: Black-Scholes

```
@cuda.jit(argtypes=(double,), restype=double, device=True, inline=True)
def cnd_cuda(d):
    A1 = 0.31938153
    A2 = -0.356563782
    A3 = 1.781477937
    A4 = -1.821255978
    A5 = 1.330274429
    RSQRT2PI = 0.39894228040143267793994605993438
    K = 1.0 / (1.0 + 0.2316419 * math.fabs(d))
    ret_val = (RSQRT2PI * math.exp(-0.5 * d * d) *
               (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5))))))
    if d > 0:
        ret_val = 1.0 - ret_val
    return ret_val
```

```
blockdim = 1024, 1
griddim = int(math.ceil(float(OPT_N)/blockdim[0])), 1
stream = cuda.stream()
d_callResult = cuda.to_device(callResultNumbapro,
stream)
d_putResult = cuda.to_device(putResultNumbapro,
stream)
d_stockPrice = cuda.to_device(stockPrice, stream)
d_optionStrike = cuda.to_device(optionStrike, stream)
d_optionYears = cuda.to_device(optionYears, stream)
for i in range(iterations):
    black_scholes_cuda[griddim, blockdim, stream](
         d_callResult, d_putResult, d_stockPrice,
d_optionStrike,
         d_optionYears, RISKFREE, VOLATILITY)
    d_callResult.to_host(stream)
    d_putResult.to_host(stream)
    stream.synchronize()
```

```
@cuda.jit(argtypes=(double[:], double[:],
double[:],
                    double[:], double[:],
double, double))
def black_scholes_cuda(callResult, putResult,
S, X, T, R, V):
    # S = stockPrice
    # X = optionStrike
    # T = optionYears
    # R = Riskfree
    # V = Volatility
    i = cuda.threadIdx.x + cuda.blockIdx.x *
cuda.blockDim.x
    if i >= S.shape[0]:
        return
    sqrtT = math.sqrt(T[i])
    d1 = (math.log(S[i] / X[i]) +
          (R + 0.5 * V * V) * T[i]) / (V *
sqrtT)
    d2 = d1 - V * sqrtT
    cndd1 = cnd_cuda(d1)
    cndd2 = cnd_cuda(d2)

    expRT = math.exp((-1. * R) * T[i])
    callResult[i] = (S[i] * cndd1 - X[i] *
expRT * cndd2)
    putResult[i] = (X[i] * expRT * (1.0 -
cndd2) -
                    S[i] * (1.0 -
                   cndd1))
```

Black-Scholes: Results



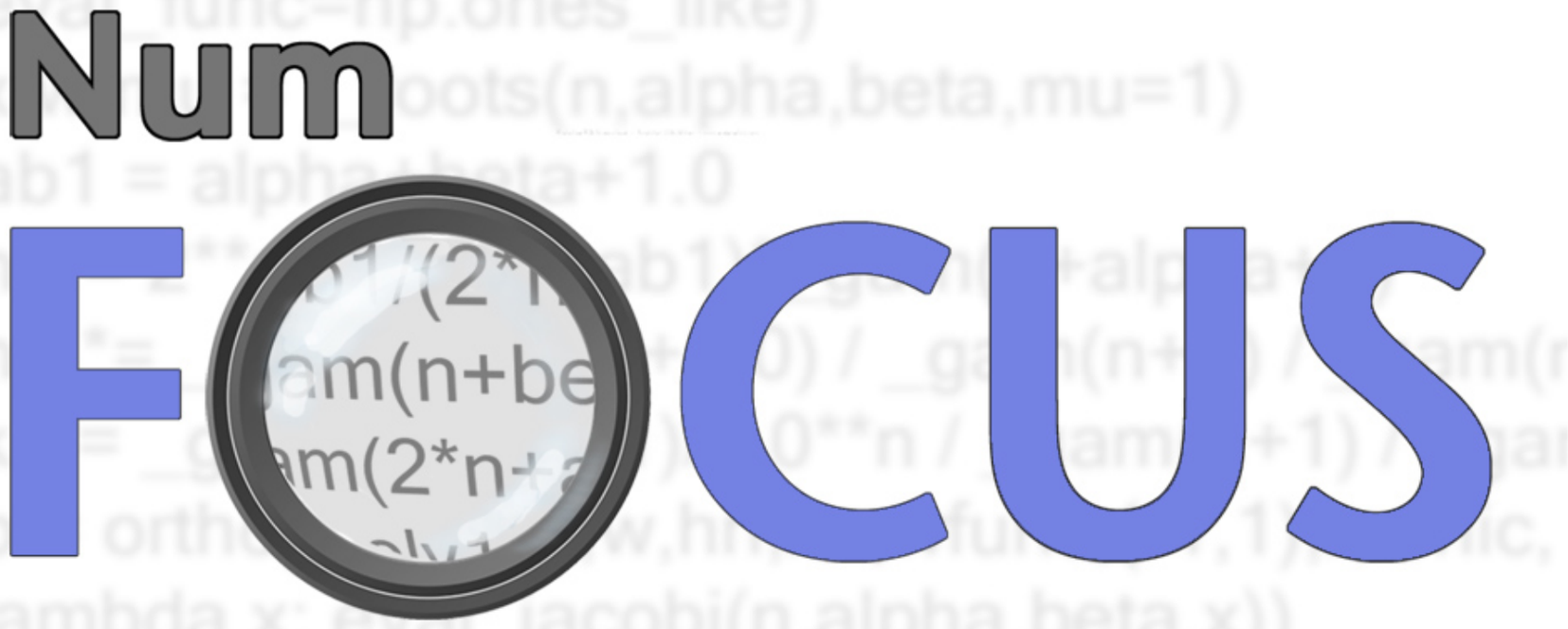
Already about 6x faster on the GPU

CONTINUUM ANALYTICS

NumFOCUS

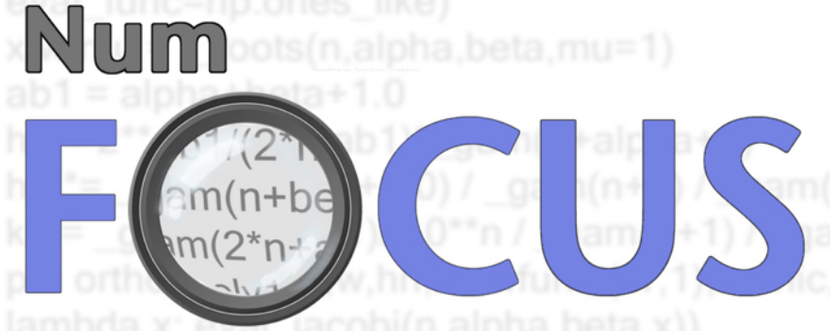
www.numfocus.org

501(c)3 Public Charity



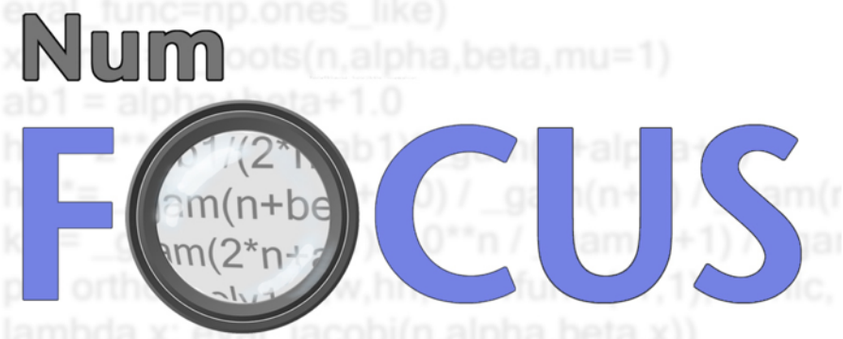
CONTINUUM ANALYTICS

NumFOCUS Mission



- Sponsor development of high-level languages and libraries for science
- Foster teaching of array-oriented and higher-order computational approaches and applied computational science
- Promote the use of open code in science and encourage reproducible and accessible research

NumFOCUS Activities



- Sponsor sprints and conferences
- Provide scholarships and grants
- Provide bounties and prizes for code development
- Pay for freely-available documentation and basic course development
- Equipment grants
- Sponsor BootCamps
- Raise funds from industries using open source high-level languages