Numba:
A Compiler for Python Functions

Stan Seibert

Director of Community Innovation @ Anaconda

My Background

- 2008: Ph.D. on the Sudbury Neutrino Observatory
- 2008-2013: Postdoc working on SNO, SNO+, LBNE (before it was DUNE), MiniCLEAN dark matter experiment
- 2013-2014: Worked at vehicle tracking and routing company (multi-traveling salesman problem)
- 2014-present: Employed at Anaconda (formerly Continuum Analytics), the creators of the Anaconda Python/R Distribution, as well many open source projects. Currently, I:
    - Manage the Numba project
    - Coordinate all of our open source projects
    - Bunch of other things (it's a startup!)

A Compiler for Python?

Striking a Balance Between Productivity and Performance

- Python has become a very popular language for scientific computing
- Python integrates well with compiled, accelerated libraries (MKL, TensorFlow, ROOT, etc)
- But what about custom algorithms and data processing tasks?
- Our goal was to make a compiler that:
  - **Works within the standard Python interpreter, and does not replace it**
  - **Integrates tightly with NumPy**
  - **Compatible with both multithreaded and distributed computing paradigms**
  - **Can be targeted at non-CPU hardware**

Numba: A JIT Compiler for Python

- An open-source, **function-at-a-time** compiler library for Python
- Compiler toolbox for different targets and execution models:
  - single-threaded CPU, multi-threaded CPU, GPU
  - regular functions, “universal functions” (array functions), etc
- Speedup: 2x (compared to basic NumPy code) to 200x (compared to pure Python)
- Combine ease of writing Python with speeds approaching FORTRAN
- BSD licensed (including GPU compiler)
- Goal is to empower scientists who make tools for themselves and other scientists

A Twisty Maze of Trade-offs

- Numba may be best understood by what it is **not**:
  - Replacement Python interpreter: PyPy, Pyston, Pyjion
    - *Hard to implement*
    - *Difficult (but not impossible) to maintain compatibility with existing Python extensions*
    - *Does not address non-CPU targets*
  - Translator of Python to C/C++: Cython, Pythran, Theano, ShedSkin, Nuitka
    - *Static analysis of dynamic languages is limiting*
    - *Ahead-of-time generated code is either underspecialized (both in data types and CPU capabilities) or bloated to cover all variants*
    - *JIT compilation requires C/C++ compiler on end user system*
- **Note**: These tradeoffs do not make these approaches bad!

Basic Example

In [87]:
```
@jit(nopython=True)
def nan_compact(x):
    out = np.empty_like(x)
    out_index = 0
    for element in x:
        if not np.isnan(element):
            out[out_index] = element
            out_index += 1
    return out[:out_index]
```

In [88]:
```
a = np.random.uniform(size=10000)
a[a < 0.2] = np.nan
np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])
```

In [89]:
```
%timeit a[~np.isnan(a)]
%timeit nan_compact(a)
```

```
10000 loops, best of 3: 52 µs per loop
100000 loops, best of 3: 19.6 µs per loop
```

Basic Example

Numba decorator (nopython=True not required)

In [87]:
```
@jit(nopython=True)
def nan_compact(x):
    out = np.empty_like(x)
    out_index = 0
    for element in x:
        if not np.isnan(element):
            out[out_index] = element
            out_index += 1
    return out[:out_index]
```

Array Allocation

Looping over ndarray x as an iterator

Using numpy math functions

Returning a slice of the array

In [88]:
```
a = np.random.uniform(size=10000)
a[a < 0.2] = np.nan
np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])
```

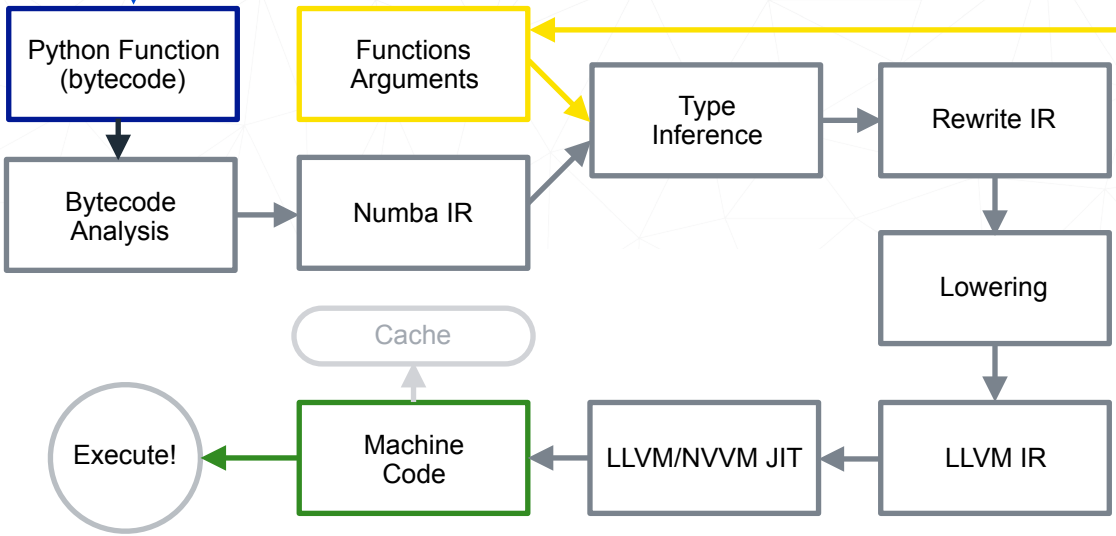
In [89]:
```
%timeit a[~np.isnan(a)]
%timeit nan_compact(a)
```

```
10000 loops, best of 3: 52 µs per loop
100000 loops, best of 3: 19.6 µs per loop
```

2.7x speedup!

How does Numba work?

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



Supported Platforms and Hardware

OS	HW	SW
Windows (7 and later)	32 and 64-bit CPUs (Incl Xeon Phi)	Python 2.7, 3.4-3.6
OS X (10.9 and later)	NVIDIA GPUs	NumPy 1.10 and later
Linux (RHEL 6 and later)	Experimental support for ARM, POWER8/9 and AMD GPUs (ROCm)	

Numba Features

- Detects CPU model during compilation and optimizes for that target
- Automatic type inference: No need to give type signatures for functions
- Dispatches to multiple type-specializations for the same function
- Call out to C libraries with CFFI and types
- Special "callback" mode for creating C callbacks to use with external libraries
- Optional caching to disk, and ahead-of-time creation of shared libraries
- Compiler is extensible with new data types and functions

SIMD: Single Instruction Multiple Data

- Numba's CPU detection will enable LLVM to autovectorize for appropriate SIMD instruction set:
  - SSE, AVX, AVX2, AVX-512
- Will become even more important as AVX-512 is now available on both Xeon Phi and Skylake Xeon processors

```python
@numba.jit(nopython=True, error_model='numpy')
def frac_diff(a, b):
    ret = np.empty_like(a)

    for i in range(a.shape[0]):
        ret[i] = np.float32(2) * (a[i] - b[i]) / (a[i] + b[i])
    return ret
```

```
LBB1_14:
        vmovups (%rdi,%rbx), %ymm0
        vmovups (%rdi,%r14), %ymm1
        vsubps  %ymm1, %ymm0, %ymm2
        vaddps  %ymm2, %ymm2, %ymm2
        vaddps  %ymm1, %ymm0, %ymm0
        vdivps  %ymm0, %ymm2, %ymm0
        vmovups %ymm0, (%r12,%rdi)
        addq    $8, %rdx
        addq    $32, %rdi
        addq    $-1, %rsi
        jne     LBB1_14
        jmp     LBB1_22
```

Manual Multithreading: Release the GIL

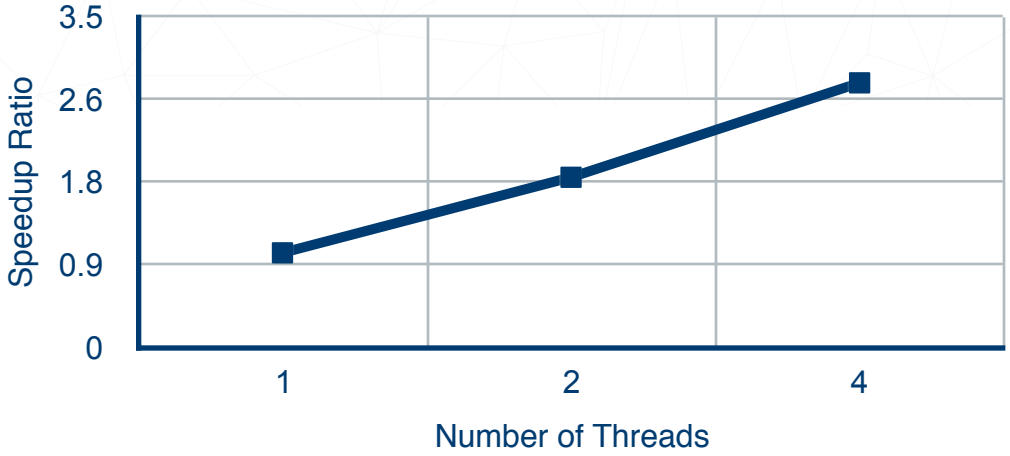
```
In [14]: SQRT_2PI = np.sqrt(2 * np.pi)
         @numba.jit(nogil=True)
         def gaussian(x, mu, sigma):
             return np.exp(0.5 * ((x - mu)**2) / sigma) / (sigma * SQRT_2PI)

         print(gaussian(0.5, 1.5, 1.0))

         0.657744623479
```

Option to release the GIL

Using Python concurrent.futures



Universal Functions (Ufuncs)

Ufuncs are a core concept in NumPy for array-oriented computing.

- A function with scalar inputs is broadcast across the elements of the input arrays:
  - `np.add([1,2,3], 3) == [4, 5, 6]`
  - `np.add([1,2,3], [10, 20, 30]) == [11, 22, 33]`
- Parallelism is present, by construction. Numba will generate loops and can automatically multi-thread if requested.
- Before Numba, creating fast ufuncs required writing C. No longer!

Universal Functions (Ufuncs)

In [13]:
```
@numba.vectorize
def response(v, gamma):
    if v < 0:
        return 0.0
    elif v < 1:
        return v ** gamma
    else:
        return v
```

Different decorator!

In [14]:
```
x = np.linspace(-1, 2, 10000)
gamma = 1.7
%timeit np.piecewise(x, [x < 0, x >= 1],[0.0, lambda x: x, lambda x: x**gamma])
%timeit response(x, gamma)
```

```
1000 loops, best of 3: 244 µs per loop
The slowest run took 411.51 times longer than the fastest. This could mean that an intermedia
te result is being cached.
10000 loops, best of 3: 136 µs per loop
```

1.8x speedup!

Multi-threaded Ufuncs

Specify type signature

```
In [12]: SQRT_2 = np.sqrt(2)

         @numba.vectorize('float64(float64, float64, float64)', target='parallel')
         def gaussian_cdf_parallel(x, mu, sigma):
             return 0.5 * (1 + math.erf( (x - mu) / (sigma * SQRT_2)))
```

Select parallel target

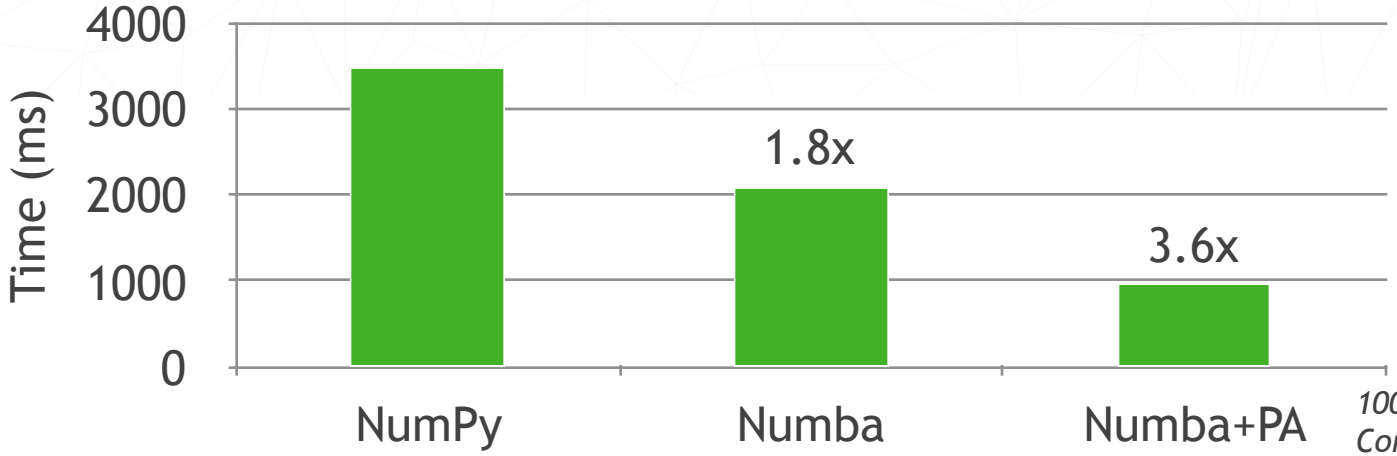
Automatically uses all CPU cores!

ParallelAccelerator

- ParallelAccelerator is a special compiler pass contributed by Intel Labs
  - Todd A. Anderson, Ehsan Totoni, Paul Liu
  - Based on similar contribution to Julia
- Automatically generates mulithreaded code in a Numba compiled-function:
  - Array expressions and reductions
  - Random functions
  - Dot products
  - Reductions
  - Explicit loops indicated with prange() call

ParallelAccelerator

[3]:
```
@numba.jit(nopython=True, parallel=True)
def logistic_regression(Y, X, w, iterations):
    for i in range(iterations):
        w -= np.dot(((1.0 / (1.0 + np.exp(-Y * np.dot(X, w))) - 1.0) * Y), X)
    return w
```



1000000x10 input, Core i7 Quad Core CPU

"OpenMP for Python": prange()

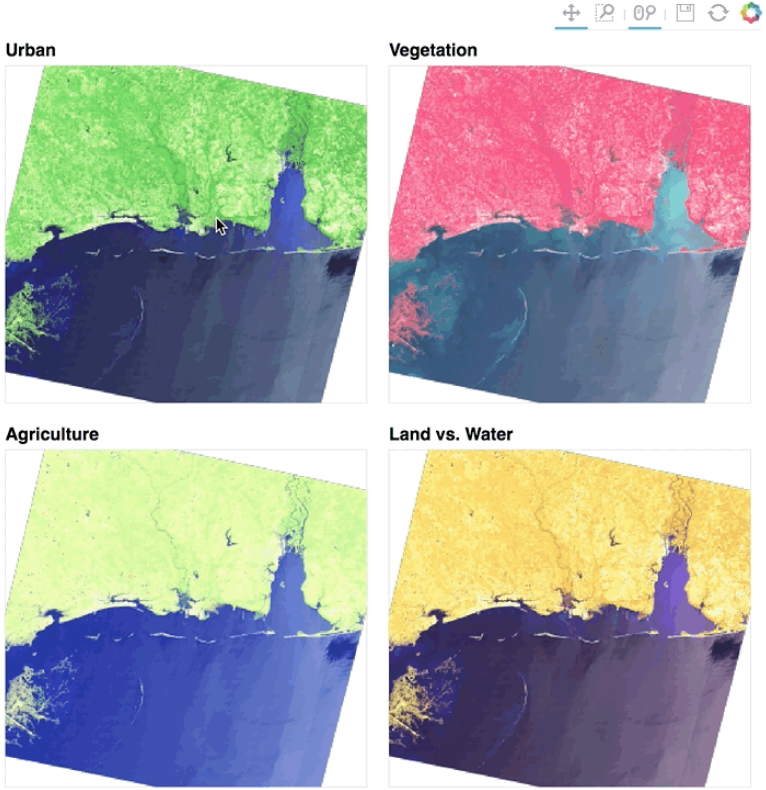
```
In [44]: @numba.jit(nopython=True, parallel=True)
         def normalize(x):
             ret = np.empty_like(x)

             for i in numba.prange(x.shape[0]):
                 acc = 0.0
                 for j in range(x.shape[1]):
                     acc += x[i,j]**2

                 norm = np.sqrt(acc)
                 for j in range(x.shape[1]):
                     ret[i,j] = x[i,j] / norm

             return ret
```

ParallelAccelerator: Image Resampling

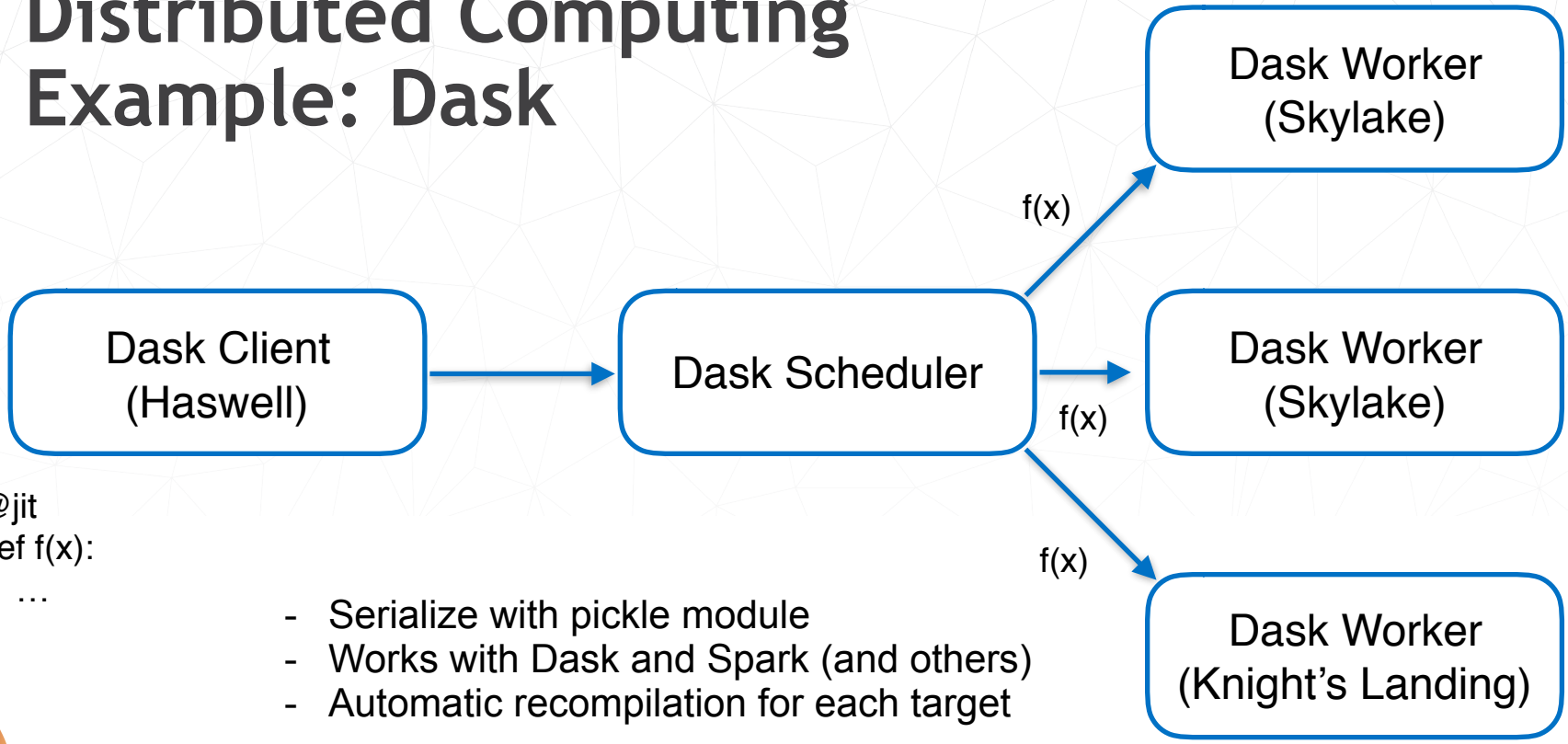


Interactive image resampling with Holoviews + Datashader

Datashader resampling implemented with Numba + prange()

https://twitter.com/bokehplots/status/915611351933947904

Distributed Computing Example: Dask




```
@jit
def f(x):
    …
```

- Serialize with pickle module
- Works with Dask and Spark (and others)
- Automatic recompilation for each target

DASK

GPU Computing

- Primary support is for NVIDIA GPUs (via CUDA)
- Approaches:
  - Ufunc compiler targeting GPU
  - Implement and call CUDA kernels in Python syntax
- Working toward interop with other CUDA-related Python projects, like CuPy, PyCUDA, etc

CUDA Python

Denotes CUDA kernel function
(launched from CPU, runs on GPU)

```
In [16]: @cuda.jit
         def cuda_histogram(x, xmin, xmax, histogram_out):
             '''Increment bin counts in histogram_out, given histogram range [xmin, xmax).'''
             nbins = histogram_out.shape[0]
             bin_width = (xmax - xmin) / nbins

             start = cuda.grid(1)
             stride = cuda.gridsize(1)

             for i in range(start, x.shape[0], stride):
                 bin_number = np.int32((x[i] - xmin)/bin_width)

                 if bin_number >= 0 or bin_number < histogram_out.shape[0]:
                     cuda.atomic.add(histogram_out, bin_number, 1)
```

Work with NumPy array elements
and attributes on the GPU

Special CUDA function
for atomic addition

```
In [19]: histogram_out.fill(0)
         cuda_histogram[32, 32](x, xmin, xmax, histogram_out)
         histogram_out

Out[19]: array([   6,   85,  446, 1601, 2925, 2849, 1547,  464,   66,   11],
               dtype=int32)
```

Launch CUDA kernel with 32
blocks of 32 threads each

Future of Numba

- Aiming for Numba 1.0 in 2018:
  - Better diagnostics to understand performance and compiler failures
  - Revamped ufunc compiler with TBB support and fewer limitations
  - Ability to have a statement block inside a compiled function jump back into the interpreter
  - Support on all major architectures: x86, ARM, PPC64, NVIDIA & AMD GPUs
  - Clearly documented compiler extension points (custom types, structs, compiler pipelines)

- Post-1.0 Numba:
  - Rewrite type inference to allow broader range of Python idioms
  - Broaden data types: strings, Arrow data frames, etc
  - C++ interop (via cling?)
  - Improve Numba architecture to enable other compiler projects to build on it
  - Integration with other Python interpreters? (Numba + PyPy anyone?)

When is Numba a Good Idea?

- Numerical algorithms
- Data is in the form of NumPy arrays, or (more broadly) flat data buffers
- Performance bottleneck is a handful of well encapsulated functions
- Example use cases:
  - Compiling user-defined functions to call from another algorithm (like an optimizer)
  - Creating "missing" NumPy/SciPy functions (librosa)
  - Rapidly prototyping GPU algorithms (FBPIC)
  - Constructing specialized Python compilers (HPAT, OAMap)

Conclusion

- Numba lets you JIT compile high performance numerical Python on-the-fly
- To learn more about Numba:
  - Easiest way to install it:

```
conda install numba
conda install numba cudatoolkit  # enable GPU support
```

  - Homepage: http://numba.pydata.org/
  - General docs: http://numba.pydata.org/numba-doc/dev/index.html
  - CPU tutorial: https://github.com/barbagroup/numba_tutorial_scipy2016
  - GPU tutorial: https://github.com/ContinuumIO/gtc2018-numba
  - Numba use cases: https://www.youtube.com/watch?v=6oXedk2tGfk

- *Numba development has been supported by Anaconda, DARPA, Gordon and Betty Moore Foundation, Intel, AMD, NVIDIA, and others*