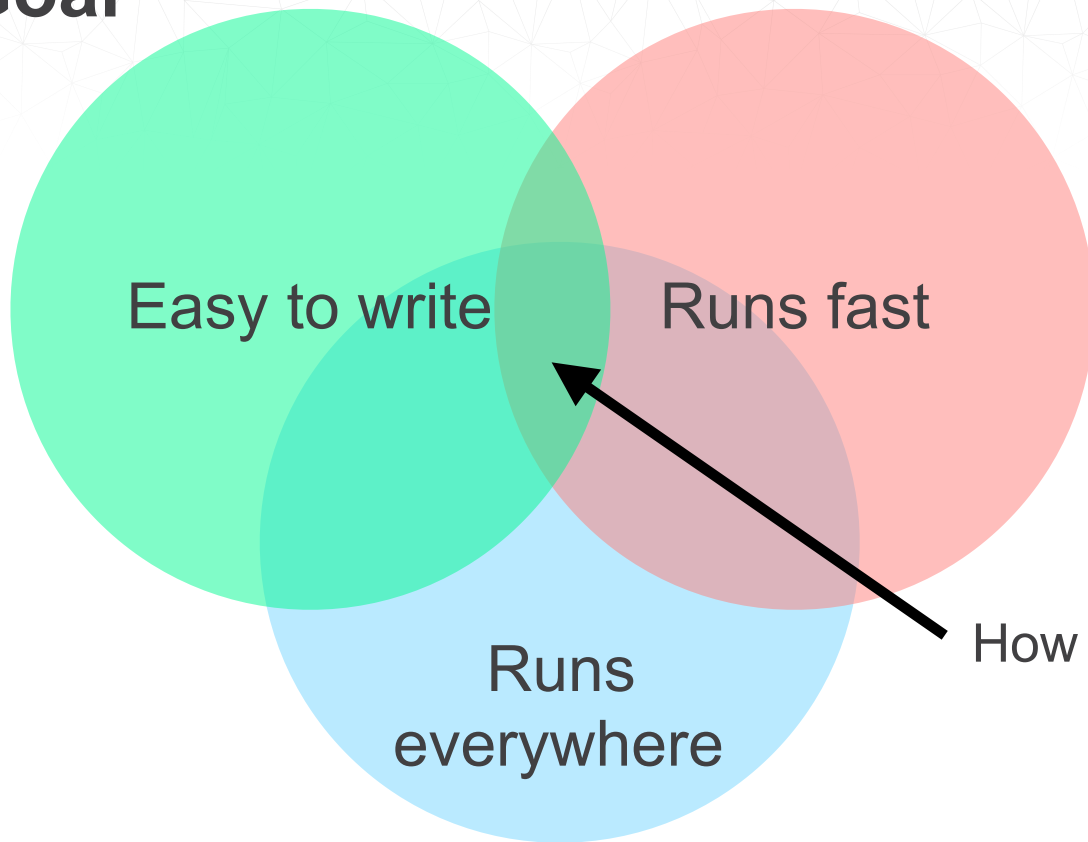
Accelerated Computing in Python (with Numba)

Stan Seibert

2019-10-29

© 2019 Anaconda

The Goal



How do we maximize this intersection?

© 2019 Anaconda

Big Picture: Why Is Python Great?

- (Mostly) straightforward language
  - Easy for users to learn
  - *Easy for domain experts to become library authors*
- Enormous community of software libraries for scientific computing, "data science", as well as sysadmin, web development, and basically everything else
- Python interpreter is **designed** to interface to compiled libraries:
  - Drive around your favorite C/C++/FORTRAN libraries from the comfort of an interpreter
- Extremely dynamic nature of the language makes almost everything possible

© 2019 Anaconda

Big Picture: Why Is Python Terrible?

- Extremely dynamic nature of the language makes it easy to write impossible to optimize code
  - *But, you don't have to use it that way*
- Interpreter is optimized for simplicity and single-threaded execution
  - *You can avoid the GIL in compiled extensions*
- Compiling all of the Python language is hard
  - *Do we actually need to compile all of it?*
- Built-in data structures are bad for HPC
  - *Fortunately, we have several good extensions already*

© 2019 Anaconda

Getting More Performance

Scale Up
(Bigger Nodes)

Scale Out
(More Nodes)

© 2019 Anaconda

Getting More Performance

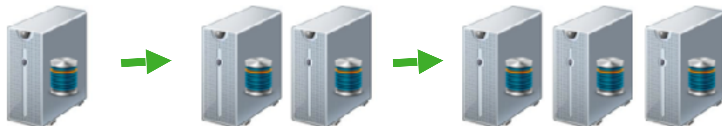
Scale Up
(Bigger Nodes)

Compilers

Distributed computing

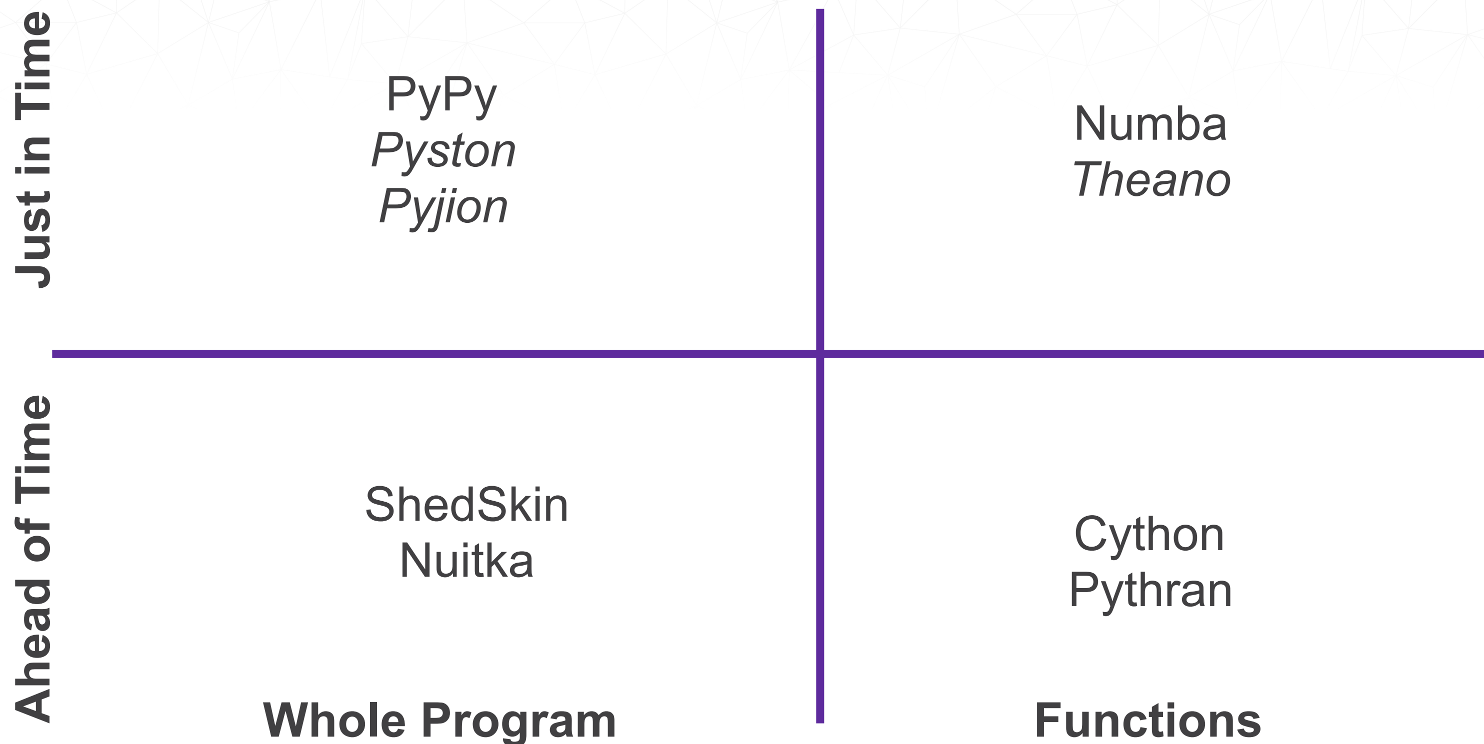
(not quite orthogonal approaches)

Scale Out
(More Nodes)



© 2019 Anaconda

The Python Compiler Quadrant



© 2019 Anaconda

A Compromise Compilation Strategy

Tracing JIT compilation of Python (PyPy)

Static translation of Python with type annotations to C (Cython)

© 2019 Anaconda

What is Numba?

- Numba is an:

  opt-in

  type-specializing

  just-in-time

  function compiler

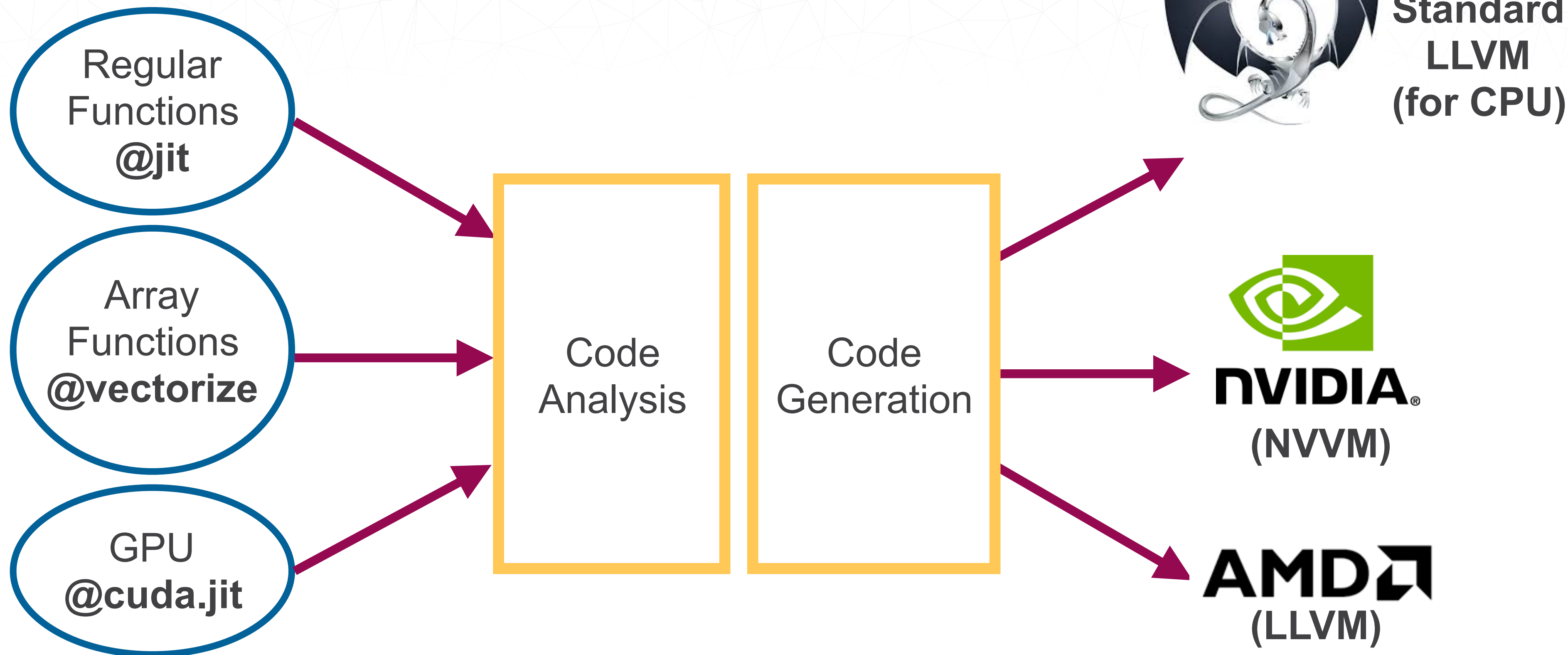
  for numerical Python

```
@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]
```

~100x faster than original Python

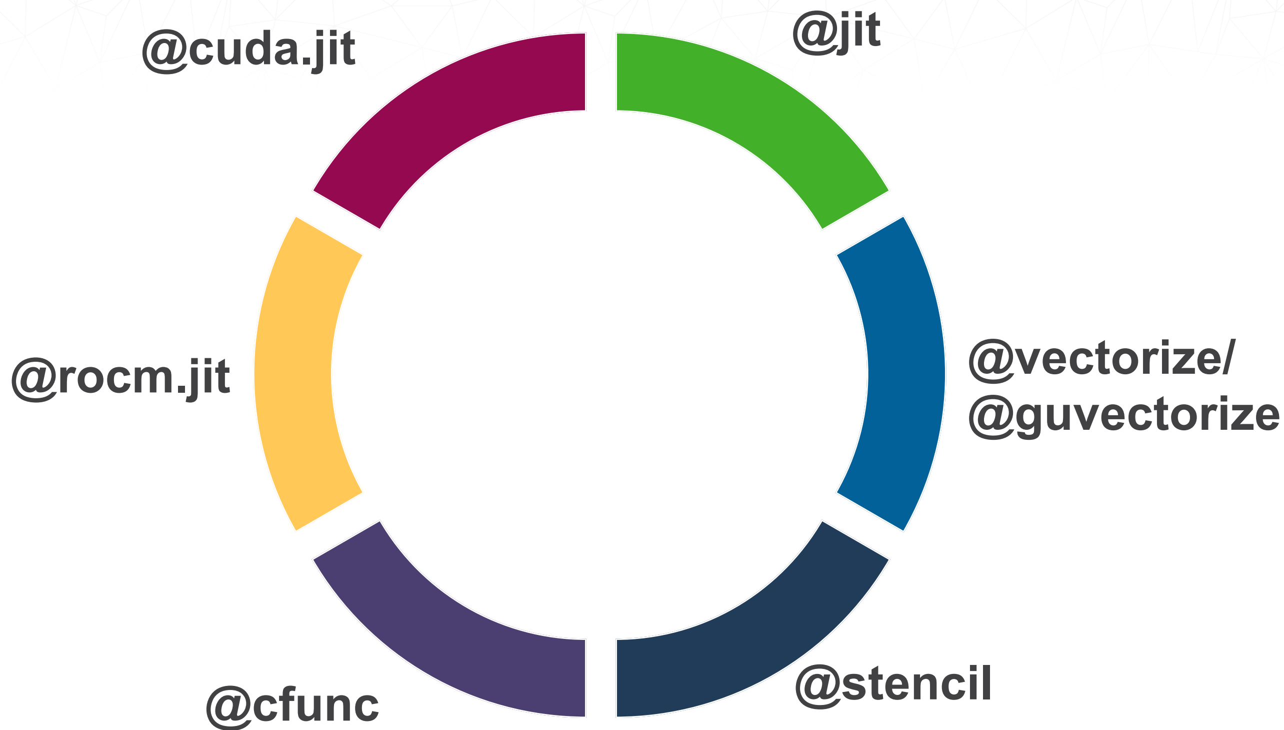
© 2019 Anaconda

How Numba Works



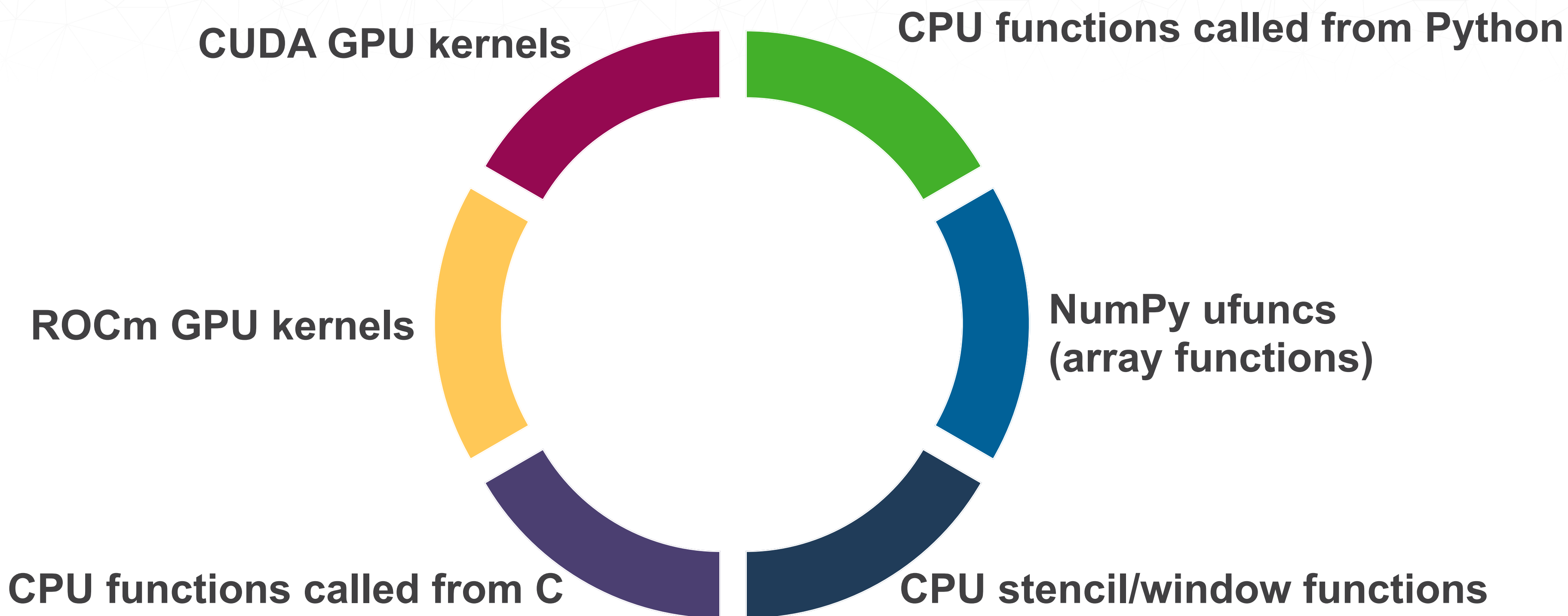
© 2019 Anaconda

A Family of Python Compilers



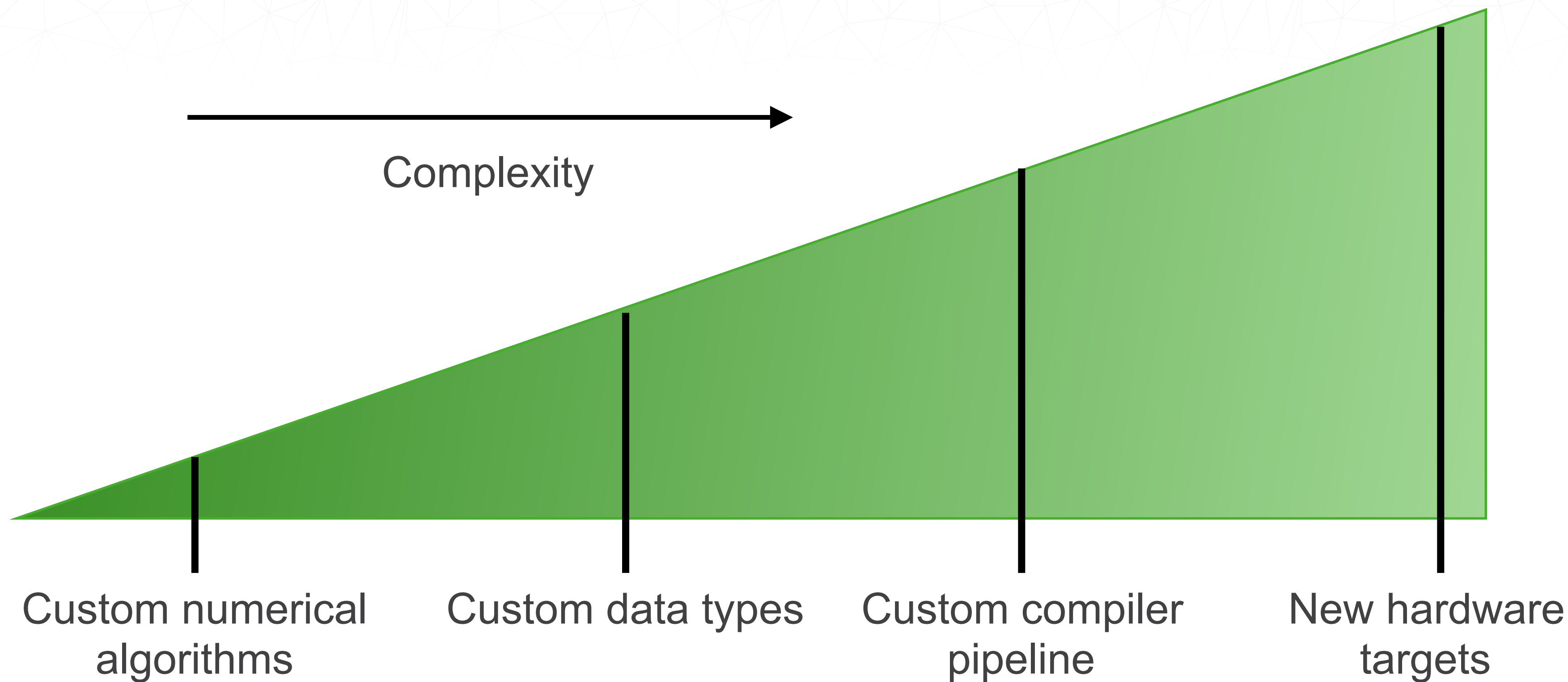
© 2019 Anaconda

A Family of Python Compilers



© 2019 Anaconda

Ways to Use Numba



© 2019 Anaconda

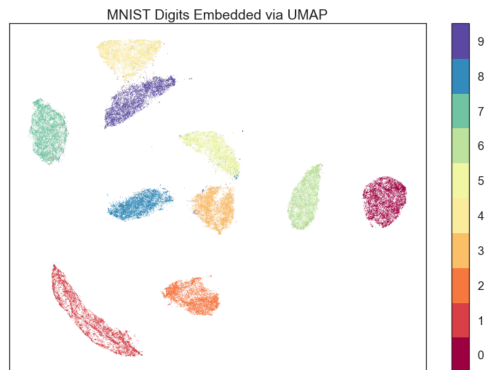
When is Numba unlikely to help?

- Whole program compilation
- Critical functions have already been converted to C or optimized Cython
- Need to interface directly to C++
- Need to generate C/C++ for separate compilation
- Algorithms are not primarily numerical
    - Exception: Numba can do pretty well at bit manipulation

© 2019 Anaconda

Custom Algorithms: UMAP

- Uniform Manifold Approximation and Projection
- Dimension reduction



Reference: McInnes, L, Healy, J, UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction, ArXiv e-prints 1802.03426, 2018

https://github.com/lmcinnes/umap

```
11  @numba.njit(fastmath=True)
12  def euclidean(x, y):
13      """Standard euclidean distance.
14
15      ..math::
16          D(x, y) = \sqrt{\sum_i (x_i - y_i)^2}
17      """
18      result = 0.0
19      for i in range(x.shape[0]):
20          result += (x[i] - y[i]) ** 2
21      return np.sqrt(result)
22
23
24  @numba.njit()
25  def standardised_euclidean(x, y, sigma=_mock_ones):
26      """Euclidean distance standardised against a vector of standard
27      deviations per coordinate.
28
29      ..math::
30          D(x, y) = \sqrt{\sum_i \frac{(x_i - y_i)**2}{v_i}}
31      """
32      result = 0.0
33      for i in range(x.shape[0]):
34          result += ((x[i] - y[i]) ** 2) / sigma[i]
35
36      return np.sqrt(result)
```


© 2019 Anaconda

Benefits of using Numba for Custom Algos

- Your library can be pure Python
- Approach FORTRAN speeds with key functions
- No need to create arch-dependent binary packages
- Reduce code-bloat by not having to pre-compile all possible type specializations (int16, int32, int64, float32, float64, etc)
- Take advantage of newer SIMD (like AVX-512) when available without sacrificing backward compatibility

© 2019 Anaconda

Another Benefit: Write it like FORTRAN

- Numba frees you from some of the constraints of Python, so make sure you take advantage of them:
  - Calling small functions is cheap / free (thanks to inlining)
    - Break up big chunky functions
  - Manual loops perform just as well as array functions.
    - Use them when you want to avoid making temporary arrays and to improve readability

© 2019 Anaconda - Confidential & Proprietary

Custom Data Types: OAMap

from DIANA-HEP

Object Array Mapping in Python

Perform high-speed calculations on columnar data without creating intermediate objects.

Access to LHC data (ROOT data)

Reference: Pivarski, Jim, et al. "Fast access to columnar, hierarchically nested data via code transformation." *Big Data (Big Data), 2017 IEEE International Conference on.* IEEE, 2017.

https://github.com/diana-hep/oamap

```
import numba
import oamap.compiler    # crucial! loads OAMap extensions!

@numba.njit
def period_ratio(stars):
    out = []
    for star in stars:
        best_ratio = None
        for one in star.planets:
            for two in star.planets:
                if (one.orbital_period is not None and one.orbital_period.val is
                    two.orbital_period is not None and two.orbital_period.val is
                    ratio = one.orbital_period.val / two.orbital_period.val
                    if best_ratio is None or ratio > best_ratio:
                        best_ratio = ratio
        if best_ratio is not None and best_ratio > 200:
            out.append(star)
    return out

# The benefit of compiling is lost on a small dataset like this (compilation tim
# but I'm sure you can find a much bigger one.  :)
>>> extremes = period_ratio(stars)
# Now that we've filtered with compiled code, we can examine the outliers in Pyt
>>> extremes
[<Record at index 284>, <Record at index 466>, <Record at index 469>, <Record at
 <Record at index 484>, <Record at index 502>, <Record at index 510>, <Record at
```

Transformed to access to data in ROOT, Parquet, etc..

© 2019 Anaconda

Benefits of using Numba for Custom Types

- Expand to more specialized use cases than arrays
- Custom types can create a "mini DSL" in Python
  - Mapping from Python syntax (attribute access, slicing, function calls, etc) to implementation is entirely overridable
- Numba implementation of type will be entirely independent of Python implementation of type

© 2019 Anaconda

Hardware Support

- Numba is continuously tested on:
    - x86 / x86_64
    - ARMv7 (Raspberry Pi)
    - ARMv8 (64-bit, everything else)
    - PPC64LE (POWER8 and POWER9)
    - NVIDIA GPUs (CUDA)
    - AMD GPUs (ROCm, not working currently)
- Adding accelerator hardware support to Numba is easier than other compilers because of our restricted compilation model

© 2019 Anaconda - Confidential & Proprietary

GPU example

```
@cuda.jit
def simulate(rng, n, prob, max_win, max_lose, out):
    tid = cuda.grid(1)
    step = cuda.gridsize(1)

    for i in range(tid, n, step):
        win = 0
        lose = 0
        while win < max_win \
                and lose < max_lose:
            if xoroshiro128p_uniform_float32(rng, tid) < prob:
                win += 1
            else:
                lose += 1
        cuda.atomic.add(out, 0, win)
```

(**7x** faster than Numba-compiled parallel code for CPU)

CUDA interop

- Numba has been pushing for other projects in the CUDA space to be able to share device arrays
- Can pass CuPy or PyTorch arrays to Numba-compiled GPU functions
- *If you like NumPy, look at CuPy, and if you need a GPU algorithm not in CuPy, look at Numba.*

© 2019 Anaconda - Confidential & Proprietary

Using Numba in a Project

- Options for introducing Numba into a code base:
  1. **Replace code with a Numba implementation**
     - *Numba is now a required dependency*
  2. **Compile functions only when Numba is present**
     - *Numba is optional dependency*
     - *Sometimes hard to write one function that maximizes performance both with and without Numba*
  3. **Pick between different implementations of same function at runtime**
     - *Numba is optional dependency*
     - *Can tailor each implementation to maximize performance*
     - *Also good strategy for exploring distributed or GPU-accelerated computing*

© 2019 Anaconda

Packaging Notes

- Packaging with Numba as a dependency:
    - Add it to your requirements.txt / conda recipe
    - Wheels for (Python 2.7, 3.5-3.7) * (win-32, win-64, osx, linux-32, linux-64) available
    - Conda packages for same combinations (some repos don't post Python 3.5 packages anymore)
- Numba **does not** require that end users have a compiler or LLVM present on their system if installed from binary packages.
- If all of your machine code comes via Numba, you can ship your package as generic for all platforms ("noarch" in conda, sdist for PyPI).

© 2019 Anaconda - Confidential & Proprietary

Conclusion

- Python is for driving around compiled functions
- Sometimes you want to create compiled functions with Python itself
  - Cython does ahead-of-time translation via C
  - Numba does just-in-time translation directly to machine code
- No silver bullet, so think about what your needs are, and who your user/developer audience is.

© 2019 Anaconda - Confidential & Proprietary

Resources

- Documentation: http://numba.pydata.org/numba-doc/latest/index.html
- Mailing list: http://numba.pydata.org/
- Github: https://github.com/numba/numba
- Gitter: https://gitter.im/numba/numba
- Feel free to ask general questions on mailing list or Gitter, and open Github issues on specific problems.

© 2019 Anaconda - Confidential & Proprietary

ANACONDA.
© 2019 Anaconda - Confidential & Proprietary