basics

February 5, 2026

1 Numba Basics

Numba is a just-in-time compiler of Python functions. It translates a Python function when it is called into a machine code equivalent that runs anywhere from 2x (simple NumPy operations) to 100x (complex Python loops) faster. In this notebook, we show some basic examples of using Numba.

[1]:
```
import numpy as np
import numba
from numba import jit
```

Let's check which version of Numba we have:

[2]:
```
print(numba.__version__)
```

```
0.62.1
```

Numba uses Python *decorators* to transform Python functions into functions that compile themselves. The most common Numba decorator is `@jit`, which creates a normal function for execution on the CPU.

Numba works best on numerical functions that make use of NumPy arrays. Here's an example:

[3]:
```
@jit(nopython=True)
def go_fast(a): # Function is compiled to machine code when called the first␣
 ↪time
    trace = 0.0
    # assuming square input matrix
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting
```

The `nopython=True` option requires that the function be fully compiled (so that the Python interpreter calls are completely removed), otherwise an exception is raised. These exceptions usually indicate places in the function that need to be modified in order to achieve better-than-Python performance. We strongly recommend always using `nopython=True`.

The function has not yet been compiled. To do that, we need to call the function:

[4]:
```
x = np.arange(100).reshape(10, 10)
go_fast(x)
```

[4]:
```
array([[  9.,  10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.],
       [ 19.,  20.,  21.,  22.,  23.,  24.,  25.,  26.,  27.,  28.],
       [ 29.,  30.,  31.,  32.,  33.,  34.,  35.,  36.,  37.,  38.],
       [ 39.,  40.,  41.,  42.,  43.,  44.,  45.,  46.,  47.,  48.],
       [ 49.,  50.,  51.,  52.,  53.,  54.,  55.,  56.,  57.,  58.],
       [ 59.,  60.,  61.,  62.,  63.,  64.,  65.,  66.,  67.,  68.],
       [ 69.,  70.,  71.,  72.,  73.,  74.,  75.,  76.,  77.,  78.],
       [ 79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,  88.],
       [ 89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.],
       [ 99., 100., 101., 102., 103., 104., 105., 106., 107., 108.]])
```

This first time the function was called, a new version of the function was compiled and executed. If we call it again, the previously generated function executions without another compilation step.

[5]:
```
go_fast(2*x)
```

[5]:
```
array([[  9.,  11.,  13.,  15.,  17.,  19.,  21.,  23.,  25.,  27.],
       [ 29.,  31.,  33.,  35.,  37.,  39.,  41.,  43.,  45.,  47.],
       [ 49.,  51.,  53.,  55.,  57.,  59.,  61.,  63.,  65.,  67.],
       [ 69.,  71.,  73.,  75.,  77.,  79.,  81.,  83.,  85.,  87.],
       [ 89.,  91.,  93.,  95.,  97.,  99., 101., 103., 105., 107.],
       [109., 111., 113., 115., 117., 119., 121., 123., 125., 127.],
       [129., 131., 133., 135., 137., 139., 141., 143., 145., 147.],
       [149., 151., 153., 155., 157., 159., 161., 163., 165., 167.],
       [169., 171., 173., 175., 177., 179., 181., 183., 185., 187.],
       [189., 191., 193., 195., 197., 199., 201., 203., 205., 207.]])
```

To benchmark Numba-compiled functions, it is important to time them without including the compilation step, since the compilation of a given function will only happen once for each set of input types, but the function will be called many times.

In a notebook, the `%timeit` magic function is the best to use because it runs the function many times in a loop to get a more accurate estimate of the execution time of short functions.

[6]:
```
%timeit go_fast(x)
```

```
294 ns ± 4.49 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Let's compare to the uncompiled function. Numba-compiled function have a special `.py_func` attribute which is the original uncompiled Python function. We should first verify we get the same results:

[7]:
```
np.testing.assert_array_equal(go_fast(x), go_fast.py_func(x))
```

And test the speed of the Python version:

[8]:
```
%timeit go_fast.py_func(x)
```

```
4.62  s ± 114 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

The original Python function is more than 20x slower than the Numba-compiled version. However, the Numba function used explicit loops, which are very fast in Numba and not very fast in Python.

Our example function is so simple, we can create an alternate version of `go_fast` using only NumPy array expressions:

[9]:
```
def go_numpy(a):
    return a + np.tanh(np.diagonal(a)).sum()
```

[10]:
```
np.testing.assert_array_equal(go_numpy(x), go_fast(x))
```

[11]:
```
%timeit go_numpy(x)
```

```
1.66  s ± 25.7 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

The NumPy version is more than 2x faster than Python, but still 10x slower than Numba.

1.0.1 Supported Python Features

Numba works best when used with NumPy arrays, but Numba also supports other data types out of the box:

- `int`, `float`
- `tuple`, `namedtuple`
- `list` (with some restrictions)
- … and others. See the Reference Manual for more details.

In particular, tuples are useful for returning multiple values from functions:

[12]:
```
import random

@jit(nopython=True)
def spherical_to_cartesian(r, theta, phi):
    '''Convert spherical coordinates (physics convention) to cartesian␣
 ↪coordinates'''
    sin_theta = np.sin(theta)
    x = r * sin_theta * np.cos(phi)
    y = r * sin_theta * np.sin(phi)
    z = r * np.cos(theta)

    return x, y, z # return a tuple

@jit(nopython=True)
def random_directions(n, r):
    '''Return ``n`` 3-vectors in random directions with radius ``r``'''
    out = np.empty(shape=(n,3), dtype=np.float64)

    for i in range(n):
        # Pick directions randomly in solid angle
        phi = random.uniform(0, 2*np.pi)
        theta = np.arccos(random.uniform(-1, 1))
        # unpack a tuple
        x, y, z = spherical_to_cartesian(r, theta, phi)
```

```
        out[i] = x, y, z

    return out
```

[13]:
```
random_directions(10, 1.0)
```

[13]:
```
array([[ 0.00174231, -0.92455074, -0.38105496],
       [ 0.0603081 ,  0.82998834,  0.55451086],
       [ 0.18122106, -0.51918238, -0.83522966],
       [ 0.9225733 , -0.35344512, -0.15470957],
       [-0.30071093,  0.95268541,  0.04431069],
       [ 0.48970106,  0.07449404,  0.8687022 ],
       [-0.23370618, -0.83207314,  0.50302656],
       [-0.54372407, -0.71933763, -0.43233958],
       [-0.00134377,  0.12594973,  0.99203571],
       [-0.70898654,  0.17427725,  0.68334876]])
```

When Numba is translating Python to machine code, it uses the LLVM library to do most of the optimization and final code generation. This automatically enables a wide range of optimizations that you don't even have to think about. If we were to inspect the output of the compiler for the previous random directions example, we would find that:

- The function body for `spherical_to_cartesian()` was inlined directly into the body of the for loop in `random_directions`, eliminating the overhead of making a function call.
- The separate calls to `sin()` and `cos()` were combined into a single, faster call to an internal `sincos()` function.

These kinds of cross-function optimizations are one of the reasons that Numba can sometimes outperform compiled NumPy code.