

threads

February 5, 2026

1 Multithreading with Numba

WARNING: Due to the CPU restrictions on notebook execution on Binder, the benefits of multithreading are going to be erratic. The code in this notebook will run on Binder, but for reasonable benchmarks, you should download and run this notebook on your own system.

Numba supports several approaches to multithreading:

- Automatic multithreading of array expressions and reductions
- Explicit multithreading of loops with `prange()`
- External multithreading with tools like `concurrent.futures` or Dask.

The first two options make use of the *ParallelAccelerator* optimization pass (contributed by Intel) in Numba. ParallelAccelerator is only supported on 64-bit platforms, and is not available for Python 2.7 on Windows. It is also only effective when compiling in `nopython` mode.

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

1.1 Automatic Multithreading

NumPy array expressions have a significant amount of implied parallelism, as operations are broadcast independently over the input elements. *ParallelAccelerator* can identify this parallelism and automatically distribute it over several threads. All we need to do is enable the parallelization pass with `parallel=True`:

```
[2]: SQRT_2PI = np.sqrt(2 * np.pi)

@jit(nopython=True, parallel=True)
def gaussians(x, means, widths):
    '''Return the value of gaussian kernels.

    x - location of evaluation
    means - array of kernel means
    widths - array of kernel widths
    '''

    n = means.shape[0]
    result = np.exp(-0.5 * ((x - means) / widths)**2) / widths
    return result / SQRT_2PI / n
```

```
[3]: means = np.random.uniform(-1, 1, size=1000000)
widths = np.random.uniform(0.1, 0.3, size=1000000)

gaussians(0.4, means, widths)
```

```
[3]: array([3.75444794e-08, 6.43581108e-09, 6.85210549e-11, ...,
           1.09848528e-06, 7.24932979e-07, 2.46927062e-08])
```

To see the effect of multiple CPUs, we can compare to the case where ParallelAccelerator disabled. Noting that decorators are functions that transform other functions, we can call `jit` as a function:

```
[4]: gaussians_nothread = jit(nopython=True)(gaussians.py_func)

%timeit gaussians_nothread(0.4, means, widths)
%timeit gaussians(0.4, means, widths)
```

```
2.87 ms ± 61.3 s per loop (mean ± std. dev. of 7 runs, 100 loops each)
759 s ± 1.58 s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

We can also compare the performance to the uncompiled NumPy array evaluation using the `.py_func` attribute to get the original Python function:

```
[5]: %timeit gaussians.py_func(0.4, means, widths) # compare to pure NumPy

4.06 ms ± 128 s per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The performance ratio depends on the number of CPUs in your system, but the multithreaded version is definitely faster than the single threaded version.

ParallelAccelerator can also handle reductions:

```
[6]: @jit(nopython=True, parallel=True)
def kde(x, means, widths):
    '''Return the value of gaussian kernels.

    x - location of evaluation
    means - array of kernel means
    widths - array of kernel widths
    '''
    n = means.shape[0]
    result = np.exp( -0.5 * ((x - means) / widths)**2 ) / widths
    return result.mean() / SQRT_2PI

kde_nothread = jit(nopython=True)(kde.py_func)
```

```
[7]: %timeit kde_nothread(0.4, means, widths)
%timeit kde(0.4, means, widths)
```

```
2.89 ms ± 28.4 s per loop (mean ± std. dev. of 7 runs, 100 loops each)
539 s ± 617 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

1.2 Multithreading with `prange()`

There are other situations where you would like multithreading, but do not have a straightforward array expression. In those cases, using `prange()` in a for loop indicates to ParallelAccelerator that this is a loop where each iteration is independent of the other and can be executed in parallel.

For example, we might want to run many Monte Carlo trials in a row:

```
[8]: import random

# Serial version
@jit(nopython=True)
def monte_carlo_pi_serial(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples

# Parallel version
@jit(nopython=True, parallel=True)
def monte_carlo_pi_parallel(nsamples):
    acc = 0
    # Only change is here
    for i in numba.prange(nsamples):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Note that `prange()` is automatically handling the reduction variable `acc` in a thread-safe way for you. We are also relying on Numba to automatically initialize the random number generator in each thread independently.

You can also have each thread in a `prange()` modify a separate element in an output array, but more general race conditions are not automatically resolved by ParallelAccelerator. Be careful!

Let's see how fast these two implementations are:

```
[9]: %time monte_carlo_pi_serial(int(4e8))
%time monte_carlo_pi_parallel(int(4e8))
```

```
CPU times: user 2.18 s, sys: 14.9 ms, total: 2.19 s
Wall time: 2.2 s
CPU times: user 1.77 s, sys: 11.3 ms, total: 1.78 s
Wall time: 330 ms
```

```
[9]: 3.14150915
```

The parallel version saturates all the CPUs once the initial compilation finishes.

1.2.1 External Multithreading

Sometimes your threading system is external to Numba entirely. You might be using `concurrent.futures` to run functions in multiple threads, or a parallel framework like `Dask`. For these situations, you do not want to use `ParallelAccelerator`, but do want to allow the Numba-compiled function to run concurrently in different threads.

To do this, you want the Numba function to release the Global Interpreter Lock (GIL) during execution. This can be done using the `nogil=True` option to `@jit`.

Let's do our Monte Carlo example again, but with Dask. Note that Numba will still handle initializing separate random number generator seeds on each thread, as it did with `ParallelAccelerator`.

```
[10]: import dask
import dask.delayed

@jit(nopython=True, nogil=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples

print(monte_carlo_pi(int(1e6)))

delayed_monte_carlo_pi = dask.delayed(monte_carlo_pi)
```

3.145824

Parallel execution:

```
[11]: %%time
futures = [delayed_monte_carlo_pi(int(4e8)) for i in range(4)]
results = dask.compute(futures)[0]
print(sum(results)/4) # average results
```

3.1416206675

CPU times: user 8.81 s, sys: 52.5 ms, total: 8.86 s
Wall time: 2.29 s

Serial execution

```
[12]: %%time
futures = [delayed_monte_carlo_pi(int(4e8)) for i in range(4)]
```

```
results = dask.compute(futures, num_workers=1)[0]
print(sum(results)/4) # average results
```

```
3.14157319
CPU times: user 8.41 s, sys: 36.4 ms, total: 8.45 s
Wall time: 8.45 s
```