Numba

A High Performance Python Compiler.

JIT-compiler based on **L**ow **L**evel **V**irtual **M**achine (LLVM) is the main engine behind Numba!

“`nopython`” mode

Decorators

@jit &@njit - to speed up almost any python function.

- @vectorize - to speed up numpy-like universal functions.

- @guvectorize - extended version of @vectorize decorator to gufuncs

- @stencil - to speed up function performing stencil kernel operations (e.g convolutions)

@jit accepts two modes of compilations:
A) nopython=True (nopython mode)
B) nopython=False (object mode)

The former produces much faster code, but has limitations that can force Numba to fall back to the latter. To prevent Numba from falling back, and instead raise an error, pass `nopython=True`.

The jit decorator

To avoid compilation times each time you invoke a Python program, you can instruct Numba to write the result of function compilation into a file-based cache. This is done by passing `cache=True`:

```
@jit(cache=True)
def f(x, y):
    return x + y
```

Numba-compiled functions can call other compiled functions. The function calls may even be inlined in the native code, depending on optimizer heuristics. For example:

```
@jit
def square(x):
    return x ** 2

@jit
def hypot(x, y):
    return math.sqrt(square(x) + square(y))
```

The `@jit` decorator *must* be added to any such library function, otherwise Numba may generate much slower code.

The threading layer in Numba

this is the library that is used internally to perform the parallel execution that occurs through the use of the `parallel` targets for CPUs, namely:

- The use of the `parallel=True` kwarg in `@jit` and `@njit`.
- The use of the `target='parallel'` kwarg in `@vectorize` and `@guvectorize`.

There are three threading layers available and they are named as follows:

- `tbb` - A threading layer backed by Intel TBB.
- `omp` - A threading layer backed by OpenMP.
- `workqueue` -A simple built-in work-sharing task scheduler.

In practice, the only threading layer guaranteed to be present is `workqueue`

The threading layer is set via the environment variable `NUMBA_THREADING_LAYER` or through assignment to `numba.config.THREADING_LAYER`

To discover the threading layer that was selected, the function `numba.threading_layer()` may be called after parallel execution.

Automatic Parallelization in Numba

Another feature of the code transformation pass (when `parallel=True`) is support for explicit parallel loops. One can use Numba's `prange` instead of `range` to specify that a loop can be parallelized. The user is required to make sure that the loop does not have cross iteration dependencies except for supported reductions.

A reduction is inferred automatically if a variable is updated by a binary function/operator using its previous value in the loop body. The initial value of the reduction is inferred automatically for the `+=`, `-=`, `*=`, and `/=` operators. For other functions/operators, the reduction variable should hold the identity value right before entering the `prange` loop. Reductions in this manner are supported for scalars and for arrays of arbitrary dimensions.

The example below demonstrates a parallel loop with a reduction (`A` is a one-dimensional Numpy array):

```
from numba import njit, prange

@njit(parallel=True)
def prange_test(A):
    s = 0
    # Without "parallel=True" in the jit-decorator
    # the prange statement is equivalent to range
    for i in prange(A.shape[0]):
        s += A[i]
    return s
```

The `@vectorize` decorator

Numba's vectorize allows Python functions taking scalar input arguments to be used as NumPy ufuncs. Creating a traditional NumPy ufunc is not the most straightforward process and involves writing some C code. Numba makes this easy. Using the `vectorize()` decorator, Numba can compile a pure Python function into a ufunc that operates over NumPy arrays as fast as traditional ufuncs written in C.

Using `vectorize()`, you write your function as operating over input scalars, rather than arrays. Numba will generate the surrounding loop (or *kernel*) allowing efficient iteration over the actual inputs.

The `vectorize()` decorator has two modes of operation:

- Eager, or decoration-time, compilation: If you pass one or more type signatures to the decorator, you will be building a Numpy universal function (ufunc). The rest of this subsection describes building ufuncs using decoration-time compilation.
- Lazy, or call-time, compilation: When not given any signatures, the decorator will give you a Numba dynamic universal function (`DUFunc`) that dynamically compiles a new kernel when called with a previously unsupported input type. A later subsection, "Dynamic universal functions", describes this mode in more depth.

```
1  from numba import vectorize, float64
2
3  @vectorize([float64(float64, float64)])
4  def f(x, y):
5      return x + y
```

The `@guvectorize` decorator

While `vectorize()` allows you to write ufuncs that work on one element at a time, the `guvectorize()` decorator takes the concept one step further and allows you to write ufuncs that will work on an arbitrary number of elements of input arrays, and take and return arrays of differing dimensions. The typical example is a running median or a convolution filter.

Contrary to `vectorize()` functions, `guvectorize()` functions don't return their result value: they take it as an array argument, which must be filled in by the function. This is because the array is actually allocated by NumPy's dispatch mechanism, which calls into the Numba-generated code.

Here is a very simple example:

```
@guvectorize([(int64[:], int64, int64[:])], '(n),()->(n)')
def g(x, y, res):
    for i in range(x.shape[0]):
        res[i] = x[i] + y
```

The underlying Python function simply adds a given scalar (`y`) to all elements of a 1-dimension array. What's more interesting is the declaration. There are two things there:

- the declaration of input and output *layouts*, in symbolic form: `(n),()->(n)` tells NumPy that the function takes a *n*-element one-dimension array, a scalar (symbolically denoted by the empty tuple `()`) and returns a *n*-element one-dimension array;
- the list of supported concrete *signatures* as per `@vectorize` ; here, as in the above example, we demonstrate `int64` arrays.