CONTINUUM ANALYTICS

Numba: A JIT Compiler for Scientific Python

Stan Seibert
Continuum Analytics
January 22, 2015

Why Python?

- For many programs, the most important resource is *developer time*.
- The best code is:
  - Easy to read
  - Easy to understand
  - Easy to modify
- But sometimes execution speed does matter. Then what do you do?

Achieving Performance in Python

When you need speed in Python the most important things you can do are:

1. Use profiling to understand where your program spends time.
   (Most of your code is irrelevant. Only worry about the parts that matter.)
2. Leverage NumPy and SciPy when working with numerical code.
3. Take a look at Numba…

How can Numba help?

- Numba is an open source Just-In-Time compiler for Python functions.
- From the types of the function arguments, Numba can often generate a specialized, fast, machine code implementation at runtime.
- Designed to work best with numerical code and NumPy arrays.
- Uses the LLVM library as the compiler backend.

Numba Features

- Numba supports:
  - Windows (XP and later), OS X (10.7 and later), and Linux
  - 32 and 64-bit x86 CPUs and NVIDIA GPUs
  - Python 2 and 3
  - NumPy versions 1.6 through 1.9
- It does *not* require a C/C++ compiler on the user's system.
- Requires less than 70 MB to install.
- Does *not* replace the standard Python interpreter (all of your existing Python libraries are still available)

Creating a Ufunc

- Numba is the best way to make new ufuncs for working with NumPy arrays

In [1]:
```
import numpy as np
import numba
```

In [2]:
```
@numba.vectorize(['float64(float64, float64)'])
def fractional_difference(a, b):
    return 2 * (a - b) / (a + b)
```

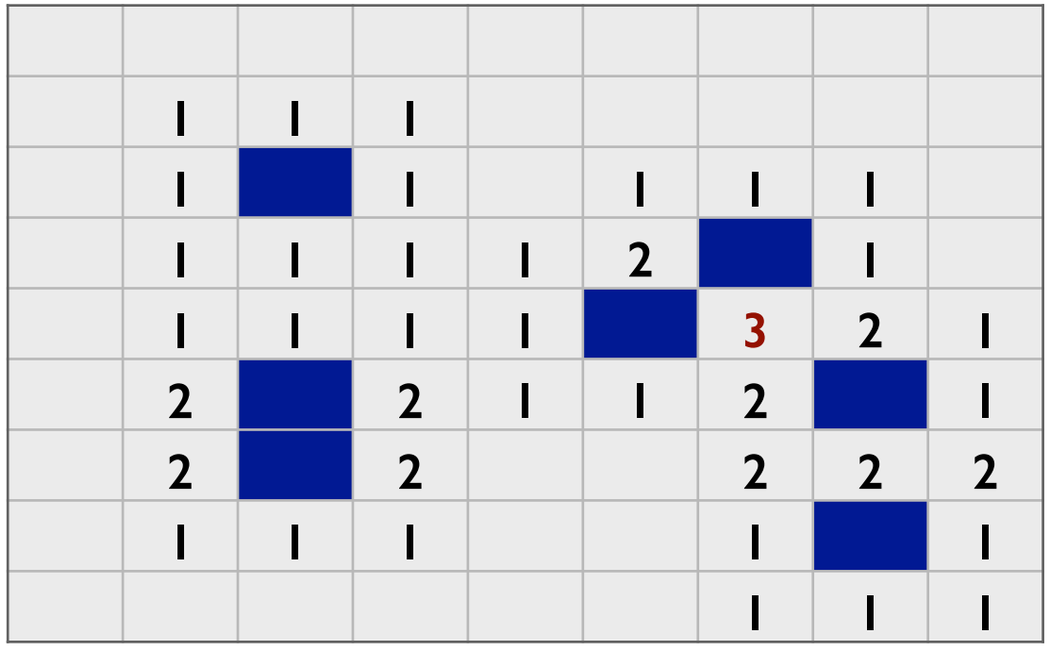
In [3]:
```
x = np.arange(10000, dtype=np.float64) + 1
y = np.arange(10000, dtype=np.float64) + 1.1
```

In [4]:
```
%timeit 2 * (x - y) / (x + y)          # Standard numpy
%timeit fractional_difference(x, y)    # Numba
```

```
10000 loops, best of 3: 48.5 µs per loop
10000 loops, best of 3: 23.6 µs per loop
```

Compiling a Function

- Sometimes you can't create a simple or efficient array expression or ufunc. Use Numba to work with array elements directly.
- Example: Suppose you have a boolean grid and you want to find the maximum number neighbors a cell has in the grid:



Compiling a Function

In [9]:
```
import numpy as np
import numba
```

In [10]:
```
@numba.jit
def check_neighbor(grid, i, j):
    if 0 <= i < grid.shape[0] and 0 <= j < grid.shape[1]:
        return grid[i, j]
    else:
        return False

@numba.jit
def find_max_neighbors(grid):
    max_neighbors = 0
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            neighbor_count = 0
            for i_offset in -1, 0, 1:
                for j_offset in -1, 0, 1:
                    if i_offset == 0 and j_offset == 0:
                        continue
                    elif check_neighbor(grid, i + i_offset, j + j_offset):
                        neighbor_count += 1
            max_neighbors = max(max_neighbors, neighbor_count)
    return max_neighbors
```

Compiling a Function

- Very hard to express this calculation purely as array operations (and even if you do, it is likely unreadable to non-NumPy experts).
- Numba let's you write out the loops, but avoid the penalty for having to loop over individual elements in the Python interpreter:

```
In [20]: grid = (np.random.uniform(size=100*100) > 0.95).reshape((100,100))
         %timeit py_find_max_neighbors(grid)
         %timeit find_max_neighbors(grid)

         10 loops, best of 3: 71.9 ms per loop
         1000 loops, best of 3: 424 µs per loop
```

169x faster!

Different Modes of Compilation

- Numba *automatically* selects between two different levels of optimization when compiling a function:
  - "object mode": supports nearly all of Python, but generally cannot speed up code by a large factor (exception: see next slide)
  - "nopython mode": supports a subset of Python, but runs at C/C++/FORTRAN speeds

Loop-Lifting

- In object mode, Numba will attempt to extract loops and compile them in nopython mode.
- Works great for functions that are bookended by uncompilable code, but have a compilable core loop.
- All happens automatically.

Loop-Lifting

In [1]:
```
import numpy as np
import numba
```

In [2]:
```
@numba.jit
def select_in_interval(a, lower, upper):
    output_buffer = np.empty_like(a)
    next_index = 0

    for element in a:
        if lower < element < upper:
            output_buffer[next_index] = element
            next_index += 1

    return output_buffer[:next_index]
```

object mode

nopython mode

object mode

In [3]:
```
x = np.random.uniform(size=100000)
%timeit x[(0.1 < x) & (x < 0.9)]
%timeit select_in_interval(x, 0.1, 0.9)
```

```
1000 loops, best of 3: 551 µs per loop
1000 loops, best of 3: 299 µs per loop
```

Nopython Mode Features

- Standard control and looping structures: if, else, while, for, range
- NumPy arrays, int, float, complex, booleans, and tuples
- Almost all arithmetic, logical, and bitwise operators as well as functions from the `math` and `numpy` modules
- Nearly all NumPy dtypes: int, float, complex, datetime64, timedelta64
- Array element access (read and write)
- Array reduction functions: sum, prod, max, min, etc
- Calling other nopython mode compiled functions
- Calling ctypes or cffi-wrapped external functions

Compiling for the GPU

GPU functions are called differently, but it is still Python!

In [1]:
```
import numpy as np
from numba import cuda
import math
```

In [2]:
```
@cuda.jit
def gpu_cos(a, out):
    i = cuda.grid(1)
    if i < a.shape[0]:
        out[i] = math.cos(a[i])
```

In [3]:
```
x = np.linspace(0, 2 * np.pi, 5000000, dtype=np.float32)
gpu_out = np.empty_like(x)
cpu_out = np.empty_like(x)

thread_config = (len(x)//128 + 1, 128)
```

In [5]:
```
%timeit np.cos(x, cpu_out)                          # Standard numpy
%timeit gpu_cos[thread_config](x, gpu_out)  # Numba using the GPU
```

```
10 loops, best of 3: 35.6 ms per loop
10 loops, best of 3: 18.3 ms per loop
```

MacBook Pro w/ GTX 650M GPU

Advanced Use Cases

- Compile new functions based on user input.
  - Great for inserting a user-provided math expression into a larger algorithm, while still achieving C speeds.
- Optimization (least squares, etc) libraries that can recompile themselves to inline a specific objective function right into the algorithm
- Multithreaded calculation without having to worry about the global interpreter lock (GIL).

NumbaPro

- NumbaPro adds higher level features on top of Numba:
  - Create ufuncs that run multithreaded on the CPU or on the GPU
  - GPU linear algebra
  - GPU FFTs

Conclusion

- Numba is a JIT compiler than understands Python!
- Achieve the same speeds as compiled languages for numerical and array-processing code.
- Can be used to create advanced workflows where user input drives compilation at runtime.
- Open source, available at:

  http://numba.pydata.org/

- Or:

```
conda install numba
```