Numba: An Array-Oriented Just-in-Time Specializing Compiler for Python

Siu Kwan Lam & Mark Florisson

January 9, 2013

Why Python?

- Rapid development cycle
- Powerful libraries
- Allows interfacing with native code
  - Excellent for glue
- ... but, slow!
  - especially for computation-heavy code: numerical algorithms

Why Numba?

Breaking the speed barrier

- Provides **JIT** for **array-oriented programming** in CPython
- Numerical loops
- Low-level C-like code in pure Python
  - pointers, structs, callbacks

Why Numba?

Work with existing tools

- Works with existing CPython extensions
- Goal: Integration with scientific software stack
  - NumPy/SciPy/Blaze
    - indexing and slicing
    - array expressions
    - math
  - C, C++, Fortran, Cython, CFFI, Julia?

Why Numba?

Minimum effort for Maximum hardware utilization

- High level tools for domains experts to exploit modern hardware
  - multicore CPU
  - manycore GPU
- Easily take advantage of parallelism and accelerators

Software Stack

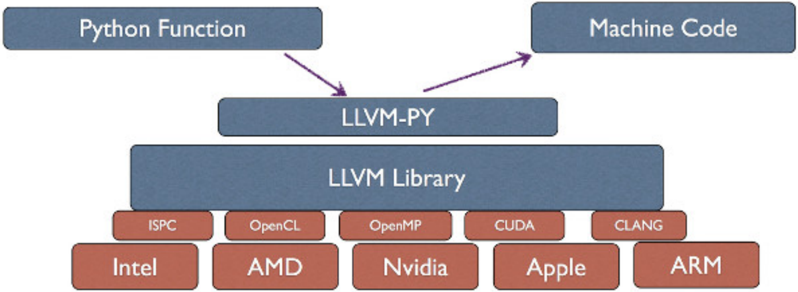

Figure : Software Stack

@jit, @autojit

- Instead of JIT-ing all Python code, we target the hotspot
- Use decorators to mark functions or classes for *just-in-time* compilation

Static runtime compilation

```
@jit(double(double[:, :]))
def func(array):
    ...
```

Dynamic just-in-time specialization

```
@autojit
def func(array):
    ...
```

JIT a Class

```
@jit
class Shrubbery(object):
    @void(int_, int_)
    def __init__(self, w, h):
        # All instance attributes must be defined in the i
        self.width = w
        self.height = h
        # Types can be explicitly specified through casts
        self.some_attr = double(1.0)

    @int_()
    def area(self):
        return self.width * self.height

    @void()
    def describe(self):
        print("This shrubbery is ", self.width,
              "by", self.height, "cubits.")
```

Compiling Strategy: Numba vs Cython vs PyPy

Numba	Cython	PyPy

▶ Runtime
  ▶ Static or dynamic
▶ Ahead of time
	▶ Ahead of time
	  ▶ build step
		▶ Runtime tracing JIT

Compiler IR: Numba vs Cython vs PyPy

Numba	Cython	PyPy
▶ LLVM	▶ C/C++	▶ PyPy JIT

Typing: Numba vs Cython vs PyPy

Numba	Cython	PyPy
▶ Type inferred ▶ Single type at each control flow point (like RPython) ▶ Variable reuse ▶ Python semantics for objects	▶ Explicit types & type inference ▶ Quick fallback to objects ▶ Python semantics for objects	▶ Full Python compatability

Example: Sum

Python

```python
@jit(f8(f8[:]))
def sum1d(A):
    n = A.shape[0]
    s = 0.0
    for i in range(n):
        s += A[i]
    return s
```

Example: Sum

LLVM IR

```
"loop_body_6:8":
    ...
    %24 = getelementptr i64* %23, i32 0
    %25 = load i64* %24, !invariant.load !0
    %26 = mul i64 %19, %25
    %27 = add i64 0, %26
    %28 = getelementptr i8* %21, i64 %27
    %29 = bitcast i8* %28 to double*
    %30 = load double* %29
    %31 = fadd double %s_2, %30
    br label %"for_increment_5:4"
```

Example: Sum

x86 Assembly

```
...
LBB0_5:
    movq    16(%rbx), %rcx
    movq    40(%rbx), %rdx
    movq    24(%rsp), %rax
    movq    (%rdx), %rdx
    imulq   %rax, %rdx
    vaddsd  (%rdx,%rcx), %xmm0, %xmm0
    incl    %eax
    movslq  %eax, %rax
    movq    %rax, 24(%rsp)
```

Example: Mandelbrot

```
@autojit
def mandel(x, y, max_iters):
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z ** 2 + c
        if (z.real ** 2 + z.imag ** 2) >= 4:
            return i

    return 255
```

Example Mandelbrot

```
@autojit
def create_fractal(min_x, max_x, min_y, max_y, image, i
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image
```

Example Mandelbrot

1000x speedup !!!

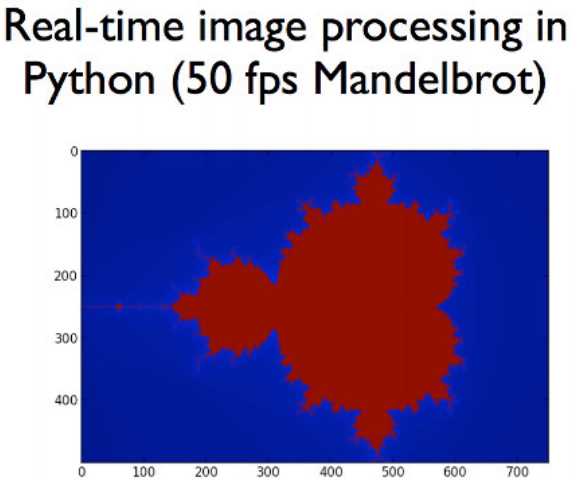

Figure : Mandelbrot

Demo

Linear Regression with Gradient Descent

Pure Python + Numpy

```
def gradient_descent(X, Y, theta, alpha, num_iters):
    m = Y.shape[0]
    theta_x = 0.0
    theta_y = 0.0
    for i in range(num_iters):
        predict = theta_x + theta_y * X
        err_x = (predict - Y)
        err_y = (predict - Y) * X
        theta_x = theta_x - alpha * (1.0 / m) * err_x.sum(
        theta_y = theta_y - alpha * (1.0 / m) * err_y.sum(
    theta[0] = theta_x
    theta[1] = theta_y
```

Numba

```
from numba import jit, f8, int32, void
@jit(void(f8[:], f8[:], f8[:], f8, int32))
def gradient_descent(X, Y, theta, alpha, num_iters):
    m = Y.shape[0]
    theta_x = 0.0
    theta_y = 0.0
    for i in range(num_iters):
        err_acc_x = 0.0
        err_acc_y = 0.0
        for j in range(X.shape[0]):
            predict = theta_x + theta_y * X[j]
            err_acc_x += predict - Y[j]
            err_acc_y += (predict - Y[j]) * X[j]
        theta_x = theta_x - alpha * (1.0 / m) * err_acc_x
        theta_y = theta_y - alpha * (1.0 / m) * err_acc_y
    theta[0] = theta_x
    theta[1] = theta_y
```

NumbaPro

```
import numbapro
from numba import jit, f8, int32, void
@jit(void(f8[:], f8[:], f8[:], f8, int32))
def gradient_descent(X, Y, theta, alpha, num_iters):
    m = Y.shape[0]
    theta_x = 0.0
    theta_y = 0.0
    for i in range(num_iters):
        predict = theta_x + theta_y * X
        err_x = (predict - Y)
        err_y = (predict - Y) * X
        theta_x = theta_x - alpha * (1.0 / m) * err_x.sum()
        theta_y = theta_y - alpha * (1.0 / m) * err_y.sum()
    theta[0] = theta_x
    theta[1] = theta_y
```

Future

- Integration with (and extension of) C++, Cython
- Task parallelism
- OpenCL
- Just-in-time specializing extension types
  - Data-Polymorphic attributes
  - Specialize methods on attribute and parameter types

Thanks

Questions?

Get Anaconda for Numba and NumbaPro

Go to https://store.continuum.io/cshop/anaconda

- Full license for Anaconda, or
- 30 days trial, or
- Anaconda CE for opensource software only (no NumbaPro)