Parakeet

A Runtime Compiler for Numerical Python

Alex Rubinsteyn @ PyData Boston 2013

What's Parakeet?

A runtime compiler for numerical Python

- ***Runtime Compiler***

  When you call a function, Parakeet bakes some native code specialized for the input types.

- ***Numerical Python***

  core syntax + array expressions + NumPy library functions

The Parakeet Pipeline

1. *wrap*

```
@jit
def f(x):
  return x + 1
```

Decorator parses function source, translates to untyped intermediate language

2. *specialize*

```
f(673.6)
```

f(x : int) { return x $+_{float}$ 1.0 }

```
f(np.arange(5))
```

f(x : array1<int>) { return map($+_{int}$, x, 1) }

3. *high level optimizations, lowering, compile to LLVM & execute*

“Numerical Python”?

- Scalar Operations
- Array Expressions

```
alpha * x[2:-3:k] + y[2:, 4:]  / beta
```

- Tuples

```
a,b,c = (1,2) + (3,)
```

- Structured Control Flow
- Functions (with keywords)

```
def f(x, y = 1, *zs):
  return x / y + np.sum(zs)

parakeet.map(f, array)
```

- Data Parallel Operators
- Some NumPy library functions

Show me the assembly

```
@jit
def count_thresh(values, thresh):
  return sum(values < thresh)
```

When you call `count_thresh` with an array of integers...

```
count_thresh(np.array([1,2]))
```

```
count_thresh:
;; %entry
    movq    8(%rdi), %rax
    movq    (%rax), %r8
    xorl    %eax, %eax
    testq   %r8, %r8
    jle .LBB0_3
;; %loop_body.preheader
    movq    24(%rdi), %rax
    movq    (%rdi), %rdx
    leaq    (%rdx,%rax,8), %rdx
    xorl    %eax, %eax
    .align  16, 0x90
;; %loop_body
.LBB0_2:
    ucomisd (%rdx), %xmm0
    seta    %cl
    movzbl  %cl, %esi
    addq    %rsi, %rax
    addq    $8, %rdx
    decq    %r8
    jne .LBB0_2
.LBB0_3:
    ret
```

Example: Image Convolution

```
@jit
def conv_3x3_trim(image, weights):
  result = np.zeros_like(image)
  for i in xrange(1, image.shape[0]-2):
    for j in xrange(1, image.shape[1]-2):
      window = image[i-1:i+2, j-1:j+2]
      result[i, j] = np.sum(window*weights)
  return result
```

	Execution Time	Compile Time
Parakeet	**293ms**	338ms
Numba	2,386ms	714ms
Python	4,173ms	~0ms

(timings for 500x500 float64 array)

Numba *loves* loops

```
@jit
def conv_3x3_trim(image, weights):
  result = np.zeros_like(image)
  for i in xrange(1, image.shape[0]-2):
    for j in xrange(1, image.shape[1]-2):
      for ii in xrange(-1,2,1):
        for jj in xrange(-1,2,1):
          coef = weights[ii+1, jj+1]
          result[i,j] += image[ii-i, jj-j] * coef
  return result
```

	Execution Time	Compile Time
Parakeet	118ms	206ms
Numba	**5ms**	516ms
Python	5,271ms	~0ms

Salvaging My Ego: Growcut

```
@jit
def growcut(image, state, state_next, window_radius):
    changes = 0
    height = image.shape[0]
    width = image.shape[1]
    for j in xrange(width):
        for i in xrange(height):
            winning_colony = state[i, j, 0]
            defense_strength = state[i, j, 1]
            for jj in xrange(max(j-window_radius, 0), min(j+window_radius+1, width)):
                for ii in xrange(max(i-window_radius, 0), min(i+window_radius+1, height)):
                    if ii != i or jj != j:
                        s = np.sum( (image[i, j, :] - image[ii, jj, :]) ** 2)
                        gval = 1.0 - np.sqrt(s) / np.sqrt(3)
                        attack_strength = gval * state[ii, jj, 1]
                        if attack_strength > defense_strength:
                            defense_strength = attack_strength
                            winning_colony = state[ii, jj, 0]
                            changes += 1
            state_next[i, j, 0] = winning_colony
            state_next[i, j, 1] = defense_strength
    return changes
```

	Execution Time	Compile Time
Parakeet	**197ms**	377ms
Numba	10,960ms	1,133ms
Python	27,065ms	~0ms

Example: NumPy-heavy code

```
def covariance(x,y):
  return ((x-x.mean()) * (y-y.mean())).mean()

@jit
def fit_simple_regression(x,y):
  slope = covariance(x,y) / covariance(x,x)
  offset = y.mean() - slope * x.mean()
  return slope, offset
```

	Execution Time	Compile Time
Parakeet	**226ms**	120ms
Numba	357ms	480ms
NumPy	362ms	~0ms

(timings for x,y = 10 billion doubles)

Pretending to be NumPy

Broadcasting = explicit maps inserted by type specializer

```
matrix + vector
```

⤳

```
map(lambda v1, v2:
      map(lambda x,y: x+y, v1, vector),
    matrix)
```

Library functions have to be implemented explicitly

```
@jit
def prod(x, axis = None):
  return reduce(prims.multiply, x, init = 1, axis = axis)
```

Which NumPy functions work?

Types	bool, uint8, int8, uint16, int16, uint32, int32, uint64, int64, float32, float64
Constructors and Views	empty_like, empty, zeros_like, zeros, ones_like, ones, arange, transpose, ravel, copy
Array Properties	alen, size, rank, dtype
Reductions	min, max, argmin, argmax, all, any, sum, mean
Scans	cumsum, cumprod
Basic Math & Logic	minimum, maximum, abs, add, subtract, multiply, divide, true_divide, mod, remainder, sign, reciprocal, logical_and, logical_or, logical_not
Comparisons	less, less_equal, equal, not_equal, greater, greater_equal
Logs and Exponents	sqrt, square, power, exp, exp2, expm1, log, log10, log2, log1p, logaddexp, logaddexp2
Rounding	trunc, rint, floor, ceil, round
Trig	cos, arccos, cosh, arccosh, sin, arcsin, sinh, cosh, tan, arctan, arctan2, tanh, arctanh

What's missing? *Lots!*

- Assertions and exceptions
- Complex numbers & structured dtypes
- Iterators (`flatiter`, `nditer`, `ndindex`, etc..)
- Random numbers (`numpy.random`)
- Linear Algebra (`numpy.linalg`)
- ...and lots more!

Math & Logic ufuncs were the easy part...

What's next?

- **Fleshing out the library functions.** *Long and tedious, but has to be done.*
- **GPU backend.** Went to the trouble of filling your code with data parallel operators, now I'd like to use them.
  - *Nested parallelism:* Kepler GPUs support nested kernel launches.
  - *Shared memory:* "Locality optimization for data parallel operators"

Thanks!

Try out Parakeet @ *https://github.com/iskandr/parakeet*