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(np.arange(5))`
   - `f(x : int) { return x +float 1.0 }`
   - `f(x : array1<int>) { return map(+int, x, 1) }`

3. **High level optimizations, lowering, compile to LLVM & execute**
“Numerical Python”?

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

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

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

- Structured Control Flow
  
  ```
  parakeet.map(f, array)
  ```

- Functions (with keywords)
  
- Data Parallel Operators
  
- Some NumPy library functions
@jit
def count_thresh(values, thresh):
    return sum(values < thresh)

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

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

count_thresh(np.array([1,2]))
Example: Image Convolution

```python
@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
```

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parakeet</td>
<td>293ms</td>
<td>338ms</td>
</tr>
<tr>
<td>Numba</td>
<td>2,386ms</td>
<td>714ms</td>
</tr>
<tr>
<td>Python</td>
<td>4,173ms</td>
<td>~0ms</td>
</tr>
</tbody>
</table>

*(timings for 500x500 float64 array)*
Numba $\textit{loves}$ 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

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parakeet</td>
<td>118ms</td>
<td>206ms</td>
</tr>
<tr>
<td>Numba</td>
<td>5ms</td>
<td>516ms</td>
</tr>
<tr>
<td>Python</td>
<td>5,271ms</td>
<td>~0ms</td>
</tr>
</tbody>
</table>
Salvaging My Ego: Growcut

```python
@jit
def growcut(image, state, state_next, window_radius):
    changes = 0
    height = image.shape[0]
    width = image.shape[1]
    for j in range(width):
        for i in range(height):
            winning_colony = state[i, j, 0]
            defense_strength = state[i, j, 1]
            for jj in range(max(j-window_radius, 0), min(j+window_radius+1, width)):
                for ii in range(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
```

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parakeet</td>
<td>197ms</td>
<td>377ms</td>
</tr>
<tr>
<td>Numba</td>
<td>10,960ms</td>
<td>1,133ms</td>
</tr>
<tr>
<td>Python</td>
<td>27,065ms</td>
<td>~0ms</td>
</tr>
</tbody>
</table>
Example: NumPy-heavy code

```python
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
```

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parakeet</td>
<td>226ms</td>
<td>120ms</td>
</tr>
<tr>
<td>Numba</td>
<td>357ms</td>
<td>480ms</td>
</tr>
<tr>
<td>NumPy</td>
<td>362ms</td>
<td>~0ms</td>
</tr>
</tbody>
</table>

(timings for \(x, y = 10\) billion doubles)
Pretending to be NumPy

Broadcasting = explicit maps inserted by type specializer

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

Library functions have to be implemented explicitly
### Which NumPy functions work?

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