Python on the GPU with Parakeet
What’s a GPU?

- Originally for drawing pixels on your screen...
- Lots of simple processors (“massively parallel”)
- Can be dramatically faster than a CPU for numerically intensive programs (possibly orders of magnitude)
- Very annoying to program
  - *warp divergence, occupancy, coalescing, bank conflicts, &c!*
## CPU vs. GPU highlights

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Specs</th>
<th>Memory Bandwidth</th>
<th>Float Math</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intel Core i7 3960X</strong></td>
<td>$1000</td>
<td>6 cores @ 3.3ghz (8 float SIMD)</td>
<td>51 GB/s</td>
<td>140 billion FLOPS</td>
</tr>
<tr>
<td><strong>NVIDIA GTX 780</strong></td>
<td>$700</td>
<td>2880 “cores” @ 928mhz</td>
<td>336 GB/s</td>
<td>5 trillion FLOPS</td>
</tr>
</tbody>
</table>
Why is a GPU faster?

CPU spends transistors on cache, memory, branch prediction, &c

GPU is ruthlessly minimalist, mostly math & logic units
How do you program a GPU?

- Two competing languages: CUDA & OpenCL
- Extensions to C/C++
- Distinguish boundary between “host” (CPU) code and sections which run on the graphics card with attributes like __host__ and __device__.
- Program uses special “which thread is this?” variables to figure out what data elements to read and where to write results.
Example CUDA Program

### Matrix Transpose (Naive Version)

#### GPU (“device”) code

```c
int TILE_DIM = 32;
int BLOCK_ROWS = 8;
int NUM_REPS = 100;

// entry-point to GPU program
__global__
void transposeNaive(float *out, float *in)
{
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = blockDim.x * TILE_DIM;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        out[x*width + (y+j)] = in[(y+j)*width + x];
}
```

Performs actual transpose

#### CPU (“host”) code

```c
void transpose(float* src, float* dst, int nx, int ny)
{
    int n = nx*ny*sizeof(float);
    dim3 dimGrid(nx/TILE_DIM, ny/TILE_DIM, 1);
    dim3 dimBlock(TILE_DIM, BLOCK_ROWS, 1);
    float *d_src, *d_dst;
    checkCuda(cudaMalloc(&d_src, n));
    checkCuda(cudaMalloc(&d_dst, n));
    checkCuda(cudaMemcpy(d_src, src, n, cudaMemcpyHostToDevice));
    transposeNaive<<<dimGrid, dimBlock>>>(d_dst, d_src);
    checkCuda(cudaMemcpy(src, d_src, n, cudaMemcpyDeviceToHost));
    checkCuda(cudaFree(d_src));
    checkCuda(cudaFree(d_dst));
}
```

Moves data to/from GPU, sets up & launches computation
GPU Programming in Python

- PyCUDA / PyOpenCL
  - Low-level GPU programs as literal strings in Python
  - Library compiles kernels & moves data
  - GPUArray container implements small subset of NumPy’s array interface

- scikits.cuda
  - Wraps precompiled NVIDIA libraries (BLAS, FFT,
Anything higher-level?

- **Theano**
  - Expression trees compile into GPU kernels (loved by neural network folks, only supports float32)

- **Copperhead**
  - Purely functional data parallel DSL in Python
  - Reinterprets list comprehensions as parallel maps
  - Compiles to Thrust (C++ CUDA library)
Parakeet

A runtime compiler for numerical Python

@jit
def now_faster(x):
    return np.mean(x < 0)

When you call a @jit wrapped function, Parakeet compiles it to native code.

Only a numerical/array-oriented subset of Python is supported.

Array operations run in parallel using OpenMP or CUDA.
What subset of Python works?

- **Array + Scalar Expressions**: `a * x[2:-3:k] + y[2:, 4:] / b`

- **Tuples**: `a,b,c = (1,2) + (3,)`

- **Data Parallel Operators**
  - `parakeet.map(f_three_inputs, x, y, z)`
  - `parakeet.reduce(add, x, axis=1)`
  - `parakeet.imap(from_index, x.shape)`
  - `array([f(xi) for xi in x])`

- **(some) NumPy functions**: `np.arange(n) + np.linspace(a,b)`

- **Functions (with keywords)**
  ```python
def f(pred, y = 1, *zs):
    while pred(y):
      y += sum(zs)
    return y
  ```

- **Loops & Conditionals**
What *doesn’t* compile? *(Most things)*

- Types other than arrays, slices, scalars: dicts, sets, lists, &c
- User-defined objects
- Assertions and exceptions
- Modifying/mutating anything other than array data
- Most library functions

Parakeet *doesn’t* compete with PyPy, it’s a small DSL
How does Parakeet work?

1. **wrap**
   - Decorator parses function source, translates to untyped IR
   - `@jit
def f(x):
    return x + 1`

2. **specialize**
   - `f(673.6) ➟ f(x : float64): return x + float 1.0`
   - `f(np.arange(5)) ➟ f(x : array1<int>): return map(+int, x, 1)`

3. **optimize**
   - Compiler magic: Simplify, CSE, DCE, LICM, Fusion, LowerArrayOperators, Indexify, ...

4. **codegen**
   - Generate C (sequential), OpenMP (multi-core), or CUDA (GPU)
Data Parallel Operators

- **map**: Apply a function to the elements of some array(s), or to each slice along a specified array axis.

- **reduce**: Combine the elements of an array with a binary operator.

- **scan**: Cumulative sums, products, &c

- **outer_map**: Apply function to cartesian product of array elements.

- **imap**: Apply function to indices in cartesian product of ranges.

- **ireduce**: Apply a function to index ranges, collect/reduce results with a binary operator.
Data Parallelism Without Trying

You don’t need to always use data parallel operators explicitly.

Type Specializer expands array broadcasting into maps

\[
\text{matrix } + \text{ scalar} \rightarrow \text{map}(\lambda x, y: x + y, \text{matrix, scalar})
\]

Comprehensions become maps

\[
[\text{sqrt}(xi) \text{ for } xi \text{ in } x] \rightarrow \text{map}([\text{sqrt}, x])
\]

NumPy library functions reimplemented w/ data parallelism explicitly

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

```python
def matmult(X, Y):
    return array([[dot(row,col) for col in Y.T] for row in X])
```

Timings for 1200x1200 float32 arrays w/ 4-core Xeon 2.67ghz & GTX 780:

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parakeet (single core)</td>
<td>14.65s</td>
<td>0.336s</td>
</tr>
<tr>
<td>Parakeet (multi-core)</td>
<td>4.08s</td>
<td>0.280s</td>
</tr>
<tr>
<td>Parakeet (GPU)</td>
<td>0.11s</td>
<td>2.16s</td>
</tr>
<tr>
<td>Numba</td>
<td>fails</td>
<td>--</td>
</tr>
<tr>
<td>Numba w/ explicit loops</td>
<td>14.79s</td>
<td>0.146s</td>
</tr>
<tr>
<td>Python + NumPy dot</td>
<td>17.4s</td>
<td>--</td>
</tr>
<tr>
<td>Python ((dot = sum(row*col)))</td>
<td>~12 minutes</td>
<td>--</td>
</tr>
<tr>
<td>ATLAS (multi-core BLAS)</td>
<td>0.40s</td>
<td>--</td>
</tr>
<tr>
<td>cuBLAS (GPU)</td>
<td>0.008s</td>
<td>--</td>
</tr>
</tbody>
</table>
Example: Image Convolution

```python
def conv_3x3_trim(image, weights):
    return array([[[(image[i-1:i+2, j-1:j+2] * weights).sum() for i in xrange(1, image.shape[0]-2)] for j in xrange(1, image.shape[1]-2)]]
```

Timings for 1200x1200 float64 array:

<table>
<thead>
<tr>
<th></th>
<th>Execution</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parakeet (single core)</td>
<td>16.8ms</td>
<td>247ms</td>
</tr>
<tr>
<td>Parakeet (4 cores)</td>
<td>11.5ms</td>
<td>220ms</td>
</tr>
<tr>
<td>Parakeet (GPU)</td>
<td>3ms</td>
<td>~2.5 seconds</td>
</tr>
<tr>
<td>Numba w/ loops</td>
<td>17ms</td>
<td>317ms</td>
</tr>
<tr>
<td>Python</td>
<td>10,975ms</td>
<td>--</td>
</tr>
</tbody>
</table>
Example: Simple Regression

```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>
</tr>
</thead>
<tbody>
<tr>
<td>Parakeet (single core)</td>
<td>202ms</td>
</tr>
<tr>
<td>Parakeet (4 cores)</td>
<td>95ms</td>
</tr>
<tr>
<td>Parakeet (GPU)</td>
<td>(death by memory transfer!) 308ms</td>
</tr>
<tr>
<td>Numba</td>
<td>357ms</td>
</tr>
<tr>
<td>NumPy</td>
<td>362ms</td>
</tr>
</tbody>
</table>

(timings for x,y = 10 billion doubles)
What’s next?

- **Fleshing out library functions**  Tedious, but has to be done.

- **Improving the GPU backend**
  
  - *Data movement*: Currently, every parallel operator copies data to 
  & from the GPU. Possible to infer when this isn’t necessary.
  
  - *Data layout*: Currently, all new arrays are row-major. Can choose 
  layouts more intelligently based on access pattern.

  - *Compile Time*: NVIDIA’s compiler is slow, should cache compiled 
  modules from hash of generated source file.

  - *Loop Parallelizer*: Simple loops can be turned into parallel
Thanks!

Try out Parakeet: **pip install parakeet**

Website:  [www.parakeetpython.com](http://www.parakeetpython.com)
## Which NumPy functions work?

<table>
<thead>
<tr>
<th>Category</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>bool, uint8, int8, uint16, int16, uint32, int32, uint64, int64, float32, float64</td>
</tr>
<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 from NumPy?

• 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...