PARAKEET

A Just-in-Time Parallel Accelerator for Numerical Python

Alex Rubinsteyn
Eric Hielscher
Nathaniel Weinman
Dennis Shasha
New York University
**Naive Python Code (is slow)**

Count the number of times a value occurs within an array:

```python
def count(big_array, target):
    c = 0
    for x in big_array:
        if x == target: c += 1
    return c
```

Takes ~10 minutes on a billion integers
NumPy exists for a reason

```python
def count(big_array, target):
    return np.sum(big_array == target)
```

Runs in 6.62 seconds, an 88X improvement!

However:

- Creates large temporary array
- Only uses single core

Can we do better without leaving Python?
Parakeet to the Rescue
(Sequential version)

```python
from parakeet import PAR
@PAR
def count(big_array, target):
    c = 0
    for x in big_array:
        if x == target: c += 1
    return c
```

- `@PAR` decorator marks boundary between Parakeet and Python
- Dynamically compiled to (sequential) LLVM

Runs in 1.4 seconds!
Let's Get Parallel

```python
@PAR
def count(big_array, t):
    return parakeet.sum(big_array == t)
```

Runs in 0.2 seconds across 8 cores!
~3000X faster than naive Python
~33X faster than NumPy

...but where did the parallelism come from?
MEET THE ADVERBS

Adverbs are *higher order array operators*

- **map**: transform each element or subarray
- **reduce**: sum, min, etc...
- **scan**: reduction which keeps intermediate values (e.g. prefix sum)
- **allpairs**: transform all pairs of elements or subarrays (e.g. matrix multiply)

Adverbs abstract enough for many implementations: sequential, multicore, GPU kernel, loop within kernel
Adverbs in disguise

No parallelism without adverbs
...but don’t always have to be explicit

parakeet.sum(big_array == t)

Library function, defined in Python as:

```python
def sum(x):
    return reduce(add, x)
```

Array broadcasting will get rewritten as:

```python
map(eq, big_array, t)
```
Python Subset

Most Python won’t run in Parakeet:

• Need source (nothing pre-compiled)
• No non-uniform data structures: lists, sets, dictionaries, etc...
• No support for user-defined objects, exceptions, generators, etc...
• Restrictions recursively apply to every called function
Is anything left?

Scalars + control flow + arrays + adverbs

- numbers, booleans, tuples, None
- math & logic operators, NumPy ufuncs
- loops, if statements
- array literals & functions like `arange`
- array attributes (e.g. `shape`, `T`)
- Parakeet’s adverbs (e.g. `map`, `reduce`, ...)

*If it’s not supported, leave it in Python*
How does it work?

1. **wrap**
   - `@PAR
   - def f(x):
   - return x + 1

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. **schedule & compile**
   - Decide where should each adverb run, synthesize native code

4. **execute**
   - add tasks to work queue (multi-core), transfer data & launch kernel (GPU)

Decorator parses function source, translates to untyped intermediate language.
Details: Typed IL

ScalarType = i8 | ... | i64 | f32 | f64

Type = scalar | tuple | array {ScalarType, rank}

- Every value annotated with type
- Rewrite polymorphism into coercions (e.g. addition becomes \(+_{\text{int32}}, +_{\text{float64}}, \ldots\) )
- Array broadcasting & indexing ⇒ maps
- Optimized aggressively (adverb fusion)
Parallelizing Adverbs is (conceptually) easy

\[
\text{map}(f, \text{concat}(x, y)) = \\
\text{concat}(\text{map}(f, x), \text{map}(f, y))
\]

\[
\text{reduce}(f, \text{concat}(x, y)) = \\
f(\text{reduce}(f, x), \text{reduce}(f, y))
\]

In practice, the split/recombine logic is more complicated and the implementations are messy.
Adverb Parallelization

**GPU**
- Kernel templates for each adverb (splice in user-defined function)
- Adverb-specific launching logic

**CPU**
- Threaded work queue
- Adverbs implemented as loops (same as single-core)
- Adverb-specific logic for combining output of each worker
Scheduling

Different locations where an adverb can run:

* Multicore backend: interpreter, multicore, sequential
* GPU backend: interpreter, kernel, thread

Choose locations which minimize (very naive) cost:

- Scalar operations all have same constant cost
- Loops will execute only once
- Sequential adverbs: cost(nested fn) \* number of elements
- Parallel adverbs: divide by number of processors

Special considerations for GPU:

- memory transfer cost
- tree-structured scans and reductions
Lots of plumbing!

- Shape inference
- Keep track of multiple function specializations
- Code caches for CPU & GPU implementations of adverb instances
- What data is already on the GPU?
- What data is no longer used?
It’s Not Magic

Matrix multiplication, Parakeet style:

```python
parakeet.allpairs(parakeet.dot, X, Y.T)
```

With 1000x1000 inputs:

- Parakeet: 310 ms (8 CPU cores)
- NumPy: 90 ms (single core BLAS)

We’re ignoring data layout and cache locality
What’s Next?

- Dynamically choose better data layout, transposed copy to local buffer (huge performance gains on both CPU and GPU)
- Fix our busted GPU backend (moving to LLVM for saner PTX generation)
- Heterogeneity! (if we have multiple backends, why can’t they split the work?)
- A less naive cost model (need to know how much work to give each backend)
**Summary**

- Restricting the programmer liberates the compiler
- Higher order array operators ("adverbs") admit diverse (parallel) implementations
- Many adverbs hiding in array-oriented code
- Python *can* be as "fast as C", for a sufficiently small definition of Python
Thanks For Listening!

Friday, June 8, 12