A Short History of Array Computing in Python

Wolf Vollprecht, PyParis 2018
- Array computing in general
- History up to NumPy
- Libraries “after” NumPy
  - Pure Python libraries
  - JIT / AOT compilers
  - Deep Learning
- NumPy extension proposal
Arrays

- Used practically in all scientific domains
- Physics, Controls, Biological System, Big Data, Deep Learning, Autonomous Cars ...
Array computing

Generalize operations on scalars to ... Arrays

\[ C \leftarrow A + B \]
## What is an n-dimensional Array?

- memory region (buffer)
- dimension
- shape
- Often strides

### Row Major (C) Layout

<table>
<thead>
<tr>
<th>Layout</th>
<th>Shape</th>
<th>Strides</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row Major</td>
<td>3, 4</td>
<td>4, 1</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11</td>
</tr>
</tbody>
</table>

### Col Major (F) Layout

<table>
<thead>
<tr>
<th>Layout</th>
<th>Shape</th>
<th>Strides</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col Major</td>
<td>3, 4</td>
<td>1, 3</td>
<td>0 1 2 3 0 4 8 1 5 9 2 6 10 3 7 11</td>
</tr>
</tbody>
</table>

- 1 el
- 4 el's
1957 / 1977 Fortran 77

- One of the oldest languages for scientific computing
- Still a reference in benchmarks
- Original implementation of BLAS & LAPACK in Fortran
- Maximum of 7 dimensions

```
integer, dimension (9, 0:99, -99:99) :: my_array
```
1966 APL: Honorable Mention

- Seriously dense language

```apl
life←{(↑1 ν.)^3 4=+/-,−1 0 1.0.ε−1 0 1.0.ε<ω}
A John Conway's "Game of Life". A Expression for next generation.
```

→ Try it online: https://tryapl.org/
1987 Matlab

- Proprietary software from Mathworks
- Dynamic interface to Fortran
- Pioneered interactive computing + visualization
1995 Numeric

- Python numerical computing package
- Inspired additions to Python (indexing syntax)
~2003 NumArray

- More flexible than Numeric
- Slower for small arrays, better for large arrays
- Split in the community:
  - SciPy remained on Numeric...
2006: NumPy

- “Merge” of Numeric and NumArray
- Fast & flexible array computing in Python
- Typed memory block
- Notion of broadcasting
- Vector Loops in C
NumPy Broadcasting

- Broadcasting: what to do when dimensions don’t match up?
NumPy ufunc

- Function that has specified input/output
- np.sin:
  - nin = 1, nout = 1
  - signature: f -> f, d -> d...
- np.add:
  - nin = 2, nout = 1
  - signature: ff -> f, dd -> d...
NumPy as a Standard

- Computing needs have shifted
- More specialized data containers needed
- Parallelization, speed, GPU, data size ...

NumPy interface de-facto standard!
Avoid temporaries

- \( R = A + B + C \)
  - \( \rightarrow T_1 = B + C \)
  - \( \rightarrow T_2 = A + T_1 \)
  - \( \rightarrow R = T_2 \)

Evaluate in chunks
import numpy as np
import numexpr as ne

a = np.arange(1e6)
b = np.arange(1e6)

ne.evaluate("a + 1")
2014 Dask

- Distributed array computing
- Can handle **large** data
- Execution of function distributed
2014 Dask
- Support for sparse ndarrays
- Advantages
  - Higher data compression
  - Faster computation
- Reuses scipy.sparse (but nD!)
- Store data in COO (coordinate list) model

```
>>> import sparse

>>> coords = [[0, 1, 2, 3, 4],
            [0, 1, 2, 3, 4]]
>>> data = [10, 20, 30, 40, 50]
>>> s = sparse.COO(coords, data, shape=(5, 5))

>>> s.todense()
array([[10, 0, 0, 0, 0],
       [0, 20, 0, 0, 0],
       [0, 0, 30, 0, 0],
       [0, 0, 0, 40, 0],
       [0, 0, 0, 0, 50]])
```
GPUs for computation

- Massively parallel
- Great for large data
- Cost of memory transfer from CPU → GPU
- Other programming model
2015 CuPy

- CUDA-aware NumPy implementation
- Part of the Chainer DL framework

```python
import cupy as cp
import numpy as np

a = np.arange(100)
gpu_a = cp.asarray(a)
gpu_a = gpu_a * 100

res_npy = cp.to_numpy(gpu_a)
```
3 libraries:
- ndtypes: shape, type & memory
- gumath: dispatch math functions on memory container
- xnd: python bridge for typed memory

```python
from xnd import xnd
from ndtypes import ndt

ndt("fixed(shape=10) * uint64")

xnd([[0, 1], [2, 3], [4, 5]], type='3 * 2 * int64')
```
JIT & AOT compilers

- Just in Time compilation for numeric code
- Can give incredible speed ups
2012 Pythran

- A Python/NumPy to C++ AOT compiler
- Supports high level optimizations in Python
- C++ implementation of NumPy with expression templates
- Cython integration

(Don’t miss the talk by Serge later today!)
#pythran export laplacien(float64[][][3])
import numpy as np

def laplacien(image):
    out_image = np.abs(4*image[1:-1,1:-1] -
                      image[0:-2,1:-1] - image[2:,1:-1] -
                      image[1:-1,0:-2] - image[1:-1,2:])

    valmax = np.max(out_image)
    valmax = max(1.,valmax) + 0.000001
    out_image /= valmax

    return out_image
2012 Numba

- A Python to LLVM JIT
- Takes Python and compiles it to Machine Code
- GPU support (Cuda + AMD)
- For High Performance: need to write explicit “for” loops
@jit('void(double[:], double[:], double[:])', nopython=True, nogil=True)
def inner_func_nb(result, a, b):
    for i in range(len(result)):
        result[i] = math.exp(2.1 * a[i] + 3.2 * b[i])
Numba + ufunc

```python
from numba import vectorize, float64

@vectorize([float64(float64, float64)])
def f(x, y):
    return x + y
```
Numba + GPU

@cuda.jit
def increment_a_2D_array(an_array):
    x, y = cuda.grid(2)
    if x < an_array.shape[0] and y < an_array.shape[1]:
        an_array[x, y] += 1
The AI winter is over ...

- Deep learning revolution
- Python ecosystem benefits **heavily**
- Lot’s of array computing
a = b = input

\[ c = a + b \]

\[ d = b + 1 \]

\[ e = c * d \]
Computation Graph

- Abstraction of computation
- Benefit: allows automatic differentiation
- Optimization opportunities
  - Common Subexpression Elimination
  - Algebraic simplifications: \((y \times x) / y \rightarrow (x)\)
  - Constant folding \((2 \times 3 + a) \rightarrow (6 + a)\)
  - Fuse ops
2007 Theano

- One of the first “Deep Learning” libraries
- Works on a computation graph
- Lazy evaluation
- Compiles kernels to C & CUDA
2015 TensorFlow

- Big library from Google
- Killed many others (including Theano)
- Same principle as Theano
- At the beginning: no compilation stage
eps = tf.placeholder(tf.float32, shape=())
damping = tf.placeholder(tf.float32, shape=())

U = tf.Variable(u_init)
Ut = tf.Variable(ut_init)

U_ = U + eps * Ut
Ut_ = Ut + eps * (laplace(U) - damping * Ut)

step = tf.group(U.assign(U_), Ut.assign(Ut_))

for i in range(1000):
    step.run({eps: 0.03, damping: 0.04})
2015 TensorFlow + XLA

- An experimental compiler for TensorFlow graphs
- JIT + AOT modes
- Uses LLVM under the hood
2016 PyTorch

- Deep Learning Framework from Facebook
- Computation Graph, but dynamic (no deferred graph model)
- Easier to have control flow
PyTorch JIT & TorchScript

- Subset of Python that can be compiled
- Generates CUDA & CPU code

```python
import torch
@torch.jit.script
def foo(x, y):
    if x.max() > y.max():
        r = x
    else:
        r = y
    return r
```
Conclusion

- NumPy is the best ... API
- Many NumPy implementations
- Many downstream projects
  - Pandas
  - xarray
  - scikit-..., scipy
The array extension proposal

- 6 months ago started by M. Rocklin
- Problem: it’s hard to write generic code
- Already extension points: __array__, __array_ufunc__

def f(x):
    y = np.tensordot(x, x.T)
    return np.mean(np.exp(y))
The array extension proposal

- E.g. CuPy input $\rightarrow$ CuPy output desired
- Arguments allowed to overload `__array_function__`

NEP 18
numpy.org/neps/nep-0018-array-function-protocol.html
Trends

- Ecosystem has become much richer in the past years
- More compilation
- More specialized NumPy implementations
- `__array_function__` will make it easy to write implementation independent code
Thanks

● Questions?

Check out xtensor & xtensor-python

NumPy for C++ ;)

Follow me on Twitter @wuoulf or GitHub @wolfv
NumPy ufunc

- Automatic broadcasting
- ufunc supports:
  - `__call__`
  - `reduce`
  - `reduceat`
  - `accumulate`
  - `outer`
  - `inner`