Beyond Numpy Arrays in Python Preparing the ecosystem for GPU, distributed, and sparse arrays

Executive Summary

In recent years Python's array computing ecosystem has grown organically to support GPUs, sparse, and distributed arrays. This is wonderful and a great example of the growth that can occur in decentralized open source development.

However to solidify this growth and apply it across the ecosystem we now need to do some central planning to move from a pair-wise model where packages need to know about each other to an ecosystem model where packages can negotiate by developing and adhering to community-standard protocols.

With moderate effort we can define a subset of the Numpy API that works well across all of them, allowing the ecosystem to more smoothly transition between hardware. This post describes the opportunities and challenges to accomplish this.

We start by discussing two kinds of libraries:

1. Libraries that *implement* the Numpy API
2. Libraries that *consume* the Numpy API and build new functionality on top of it

Libraries that Implement the Numpy API

The Numpy array is one of the foundations of the numeric Python ecosystem, and serves as the standard model for similar libraries in other languages. Today it is used to analyze satellite and biomedical imagery, financial models, genomes, oceans and the atmosphere, super-computer simulations, and data from thousands of other domains.

However, Numpy was designed several years ago, and its implementation is no longer optimal for some modern hardware, particularly multi-core workstations, many-core GPUs, and distributed clusters.

Fortunately other libraries implement the Numpy array API on these other architectures:

- CuPy (https://cupy.chainer.org/): implements the Numpy API on GPUs with CUDA
- Sparse (https://sparse.pydata.org/): implements the Numpy API for sparse arrays that are mostly zeros
- Dask array (https://dask.pydata.org/): implements the Numpy API in parallel for multi-core workstations or distributed clusters

So even when the Numpy implementation is no longer ideal, the *Numpy API* lives on in successor projects.

Note: the Numpy implementation remains ideal most of the time. Dense in-memory arrays are still the common case. This blogpost is about the minority of cases where Numpy is not ideal

So today we can write code similar code between all of Numpy, GPU, sparse, and parallel arrays:

```
import numpy as np
x = np.random.random(...)  # Runs on a single CPU
y = x.T.dot(np.log(x) + 1)
z = y - y.mean(axis=0)
print(z[:5])

import cupy as cp
x = cp.random.random(...)  # Runs on a GPU
y = x.T.dot(cp.log(x) + 1)
z = y - y.mean(axis=0)
print(z[:5].get())

import dask.array as da
x = da.random.random(...)  # Runs on many CPUs
y = x.T.dot(da.log(x) + 1)
z = y - y.mean(axis=0)
print(z[:5].compute())

...
```

Additionally, each of the deep learning frameworks (TensorFlow, PyTorch, MXNet) has a Numpy-like thing that is *similar-ish* to Numpy's API, but definitely not trying to be an exact match.

Libraries that consume and extend the Numpy API

At the same time as the development of Numpy APIs for different hardware, many libraries today build algorithmic functionality on top of the Numpy API:

1. XArray (http://xarray.pydata.org/en/stable/) for labeled and indexed collections of arrays
2. Autograd (https://github.com/hips/autograd) and Tangent (https://github.com/google/tangent/): for automatic differentiation
3. TensorLy (http://tensorly.org/stable/index.html) for higher order array factorizations
4. Dask array (https://dask.pydata.org) which coordinates many Numpy-like arrays into a logical parallel array

   (dask array both *consumes* and *implements* the Numpy API)
5. Opt Einsum (http://optimized-einsum.readthedocs.io/en/latest/) for more efficient einstein summation operations
6. …

These projects and more enhance array computing in Python, building on new features beyond what Numpy itself provides.

There are also projects like Pandas, Scikit-Learn, and SciPy, that use Numpy's in-memory internal representation. We're going to ignore these libraries for this blogpost and focus on those libraries that only use the high-level Numpy API and not the low-level representation.

Opportunities and Challenges

Given the two groups of projects:

1. New libraries that *implement* the Numpy API (CuPy, Sparse, Dask array)
2. New libraries that *consume* and *extend* the Numpy API (XArray, Autograd/tangent, TensorLy, Einsum)

We want to use them together, applying Autograd to CuPy, TensorLy to Sparse, and so on, including all future implementations that might follow. This is challenging.

Unfortunately, while all of the array implementations APIs are *very similar* to Numpy's API, they use different functions.

```
>>> numpy.sin is cupy.sin
False
```

This creates problems for the consumer libraries, because now they need to switch out which functions they use depending on which array-like objects they've been given.

```python
def f(x):
    if isinstance(x, numpy.ndarray):
        return np.sin(x)
    elif isinstance(x, cupy.ndarray):
        return cupy.sin(x)
    elif ...
```

Today each array project implements a custom plugin system that they use to switch between some of the array options. Links to these plugin mechanisms are below if you're interested:

- xarray/core/duck_array_ops.py (https://github.com/pydata/xarray/blob/c346d3b7bcdbd6073cf96fdeb0710467a284a611/xarray/core/duck_array_ops.py)
- tensorly/backend (https://github.com/tensorly/tensorly/tree/af0700af61ca2cd104e90755d5e5033e23fd4ec4/tensorly/backend)
- autograd/numpy/numpy_vspaces.py (https://github.com/HIPS/autograd/blob/bd3f92fcd4d66424be5fb6b6d3a7f9195c98eebf/autograd/numpy/numpy_vspaces.py)
- tangent/template.py (https://github.com/google/tangent/blob/bc64848bba964c632a6da4965fb91f2f61a3cdd4/tangent/template.py)
- dask/array/core.py#L59-L62 (https://github.com/dask/dask/blob/8f164773cb3717b3c5ad856341205f605b8404cf/dask/array/core.py#L59-L62)
- opt_einsum/backends.py (https://github.com/dgasmith/opt_einsum/blob/32c1b0adb50511da1b86dc98bcf169d79b44efce/opt_einsum/backends.py)

For example XArray can use either Numpy arrays or Dask arrays. This has been hugely beneficial to users of that project, which today seamlessly transition from small in-memory datasets on their laptops to 100TB datasets on clusters, all using the same programming model. However when considering adding sparse or GPU arrays to XArray's plugin system, it quickly became clear that this would be expensive today.

Building, maintaining, and extending these plugin mechanisms is *costly*. The plugin systems in each project are not alike, so any new array implementation has to go to each library and build the same code several times. Similarly, any new algorithmic library must build plugins to every ndarray implementation. Each library has to explicitly import and understand each other library, and has to adapt as those libraries change over time. This coverage is not complete, and so users lack confidence that their applications are portable between hardware.

Pair-wise plugin mechanisms make sense for a single project, but are not an efficient choice for the full ecosystem.

Solutions

I see two solutions today:

1. Build a new library that holds dispatch-able versions of all of the relevant Numpy functions and convince everyone to use it instead of Numpy internally
2. Build this dispatch mechanism into Numpy itself

Each has challenges.

Build a new centralized plugin library

We can build a new library, here called `arrayish`, that holds dispatch-able versions of all of the relevant Numpy functions. We then convince everyone to use it instead of Numpy internally.

So in each array-like library's codebase we write code like the following:

```
# inside numpy's codebase
import arrayish
import numpy
@arrayish.sin.register(numpy.ndarray, numpy.sin)
@arrayish.cos.register(numpy.ndarray, numpy.cos)
@arrayish.dot.register(numpy.ndarray, numpy.ndarray, numpy.dot)
...
```

```
# inside cupy's codebase
import arrayish
import cupy
@arrayish.sin.register(cupy.ndarray, cupy.sin)
@arrayish.cos.register(cupy.ndarray, cupy.cos)
@arrayish.dot.register(cupy.ndarray, cupy.ndarray, cupy.dot)
...
```

and so on for Dask, Sparse, and any other Numpy-like libraries.

In all of the algorithm libraries (like XArray, autograd, TensorLy, …) we use arrayish instead of Numpy

```
# inside XArray's codebase
# import numpy
import arrayish as numpy
```

This is the same plugin solution as before, but now we build a community standard plugin system that hopefully all of the projects can agree to use.

This reduces the big `n by m` cost of maintaining several plugin systems, to a more manageable `n plus m` cost of using a single plugin system in each library. This centralized project would also benefit, perhaps, from being better maintained than any individual project is likely to do on its own.

However this has costs:

1. Getting many different projects to agree on a new standard is hard
2. Algorithmic projects will need to start using arrayish internally, adding new imports like the following:

   ```
   import arrayish as numpy
   ```

   And this wll certainly cause some complications interally

3. Someone needs to build an maintain the central infrastructure

Hameer Abbasi (https://github.com/hameerabbasi) put together a rudimentary prototype for arrayish here: github.com/hameerabbasi/arrayish (https://github.com/hameerabbasi/arrayish). There has been some discussion about this topic, using XArray+Sparse as an example, in pydata/sparse #1 (https://github.com/pydata/sparse/issues/1)

Dispatch from within Numpy

Alternatively, the central dispatching mechanism could live within Numpy itself.

Numpy functions could learn to hand control over to their arguments, allowing the array implementations to take over when possible. This would allow existing Numpy code to work on externally developed array implementations.

There is precedent for this. The **array_ufunc** (https://docs.scipy.org/doc/numpy/reference/arrays.classes.html#numpy.class.__array_ufunc__) protocol allows any class that defines the `__array_ufunc__` method to take control of any Numpy ufunc like `np.sin` or `np.exp`. Numpy reductions like `np.sum` already look for `.sum` methods on their arguments and defer to them if possible.

Some array projects, like Dask and Sparse, already implement the `__array_ufunc__` protocol. There is also an open PR for CuPy (https://github.com/cupy/cupy/pull/1247). Here is an example showing Numpy functions on Dask arrays cleanly.

```
>>> import numpy as np
>>> import dask.array as da

>>> x = da.ones(10, chunks=(5,))  # A Dask array

>>> np.sum(np.exp(x))             # Apply Numpy function to a Dask array
dask.array<sum-aggregate, shape=(), dtype=float64, chunksize=()>  # get a Dask array
```

I recommend that all Numpy-API compatible array projects implement the `__array_ufunc__` protocol.

This works for many functions, but not all. Other operations like `tensordot`, `concatenate`, and `stack` occur frequently in algorithmic code but are not covered here.

This solution avoids the community challenges of the `arrayish` solution above. Everyone is accustomed to aligning themselves to Numpy's decisions, and relatively little code would need to be rewritten.

The challenge with this approach is that historically Numpy has moved more slowly than the rest of the ecosystem. For example the `__array_ufunc__` protocol mentioned above was discussed for several years before it was merged. Fortunately Numpy has recently received (https://www.numfocus.org/blog/numpy-receives-first-ever-funding-thanks-to-moore-foundation) funding (https://bids.berkeley.edu/news/bids-receives-sloan-foundation-grant-contribute-numpy-development) to help it make changes like this more rapidly. The full time developers hired under this funding have just started though, and it's not clear how much of a priority this work is for them at first.

For what it's worth I'd prefer to see this Numpy protocol solution take hold.

Final Thoughts

In recent years Python's array computing ecosystem has grown organically to support GPUs, sparse, and distributed arrays. This is wonderful and a great example of the growth that can occur in decentralized open source development.

However to solidify this growth and apply it across the ecosystem we now need to do some central planning to move from a pair-wise model where packages need to know about each other to an ecosystem model where packages can negotiate by developing and adhering to community-standard protocols.

The community has done this transition before (Numeric + Numarray -> Numpy, the Scikit-Learn fit/predict API, etc..) usually with surprisingly positive results.

The open questions I have today are the following:

1. How quickly can Numpy adapt to this demand for protocols while still remaining stable for its existing role as foundation of the ecosystem
2. What algorithmic domains can be written in a cross-hardware way that depends only on the high-level Numpy API, and doesn't require specialization at the data structure level. Clearly some domains exist (XArray, automatic differentiation), but how common are these?
3. Once a standard protocol is in place, what other array-like implementations might arise? In-memory compression? Probabilistic? Symbolic?

Update

After discussing this topic at the May NumPy Developer Sprint (https://scisprints.github.io/#may-numpy-developer-sprint) at BIDS (https://bids.berkeley.edu/) a few of us have drafted a Numpy Enhancement Proposal (NEP) available here (https://github.com/numpy/numpy/pull/11189).

0 Comments **Matthew Rocklin's Blog**

1 Login

Recommend Tweet Share

Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy