Sparse: A more modern sparse array library

Hameer Abbasi[‡*]

https://youtu.be/xH5eVcb1S1A

Abstract—This paper is about sparse multi-dimensional arrays in Python. We discuss their applications, layouts, and current implementations in the SciPy ecosystem along with strengths and weaknesses. We then introduce a new package for sparse arrays that builds on the legacy of the scipy.sparse implementation, but supports more modern interfaces, dimensions greater than two, and improved integration with newer array packages, like XArray and Dask. We end with performance benchmarks and notes on future work. Additionally, this work provides a concrete implementation of the recent NumPy array protocols to build generic array interfaces for improved interoperability, and so may be useful for broader community discussion.

Index Terms—sparse, sparse arrays, sparse matrices, scipy.sparse, ndarray, ndarray interface

Introduction

Sparse arrays are important in many situations and offer both speed and memory benefits over regular arrays when solving a broad spectrum of problems. For example, they can be used in solving systems of equations [LN89], solving partial differential equations [MR91], machine learning problems involving Bayesian models [Tip01] and natural language processing [NTK11].

As a motivating example, consider two NumPy arrays with a shape of `(10 ** 5, 10 ** 5)` and only five nonzero elements per row. Computations on such arrays, such as addition, multiplication and so on would perform the operation on each of the 10^{10} elements individually, taking up a large amount of time and memory.

If we instead focused on just the nonzero elements in each array and worked with those, we would be down to at most 10^6 elements to work with, a *huge* improvement. If we were smart about how the array would be stored, we could also bring down memory usage as well. This is, in essence, what sparse arrays do and what they're used for.

Traditionally, within the SciPy ecosystem, sparse arrays have been provided within SciPy [Sci18] in the submodule `scipy.sparse`, which is arguably the most feature-complete implementation of sparse matrices within the ecosystem, providing support for basic arithmetic, linear algebra and graph theoretic algorithms.

However, it lacks certain features which prevent it from working nicely with other packages in the ecosystem which consume NumPy's [Num18] `ndarray` interface:

- It doesn't follow the `ndarray` interface (rather, it follows NumPy's deprecated `matrix` interface)
- It is limited to two dimensions only (even one-dimensional structures aren't supported)

In addition, `scipy.sparse` is depended on by many downstream projects, which makes removing NumPy's `matrix` interface that much more difficult, and limits usage of both `ndarray` style duck arrays and `scipy.sparse` arrays within the same codebase.

This is important for a number of other packages that are quite innovative, but cannot take advantage of `scipy.sparse` for these reasons, because they expect objects following the `ndarray` interface. These include packages like Dask [Das18] (which is useful for parallel computing, even across clusters, for both NumPy arrays and Pandas dataframes) and XArray [xar18] (which extends Pandas dataframes to multiple dimensions).

Both of these frameworks could benefit tremendously from sparse structures. In the case of Dask, it could be used in combination with sparse structures to scale up computational tasks that need sparse structures. In the case of XArray, datasets with large amounts of missing data could be represented efficiently, as well as other benefits such as broadcasting by axis name rather than by rather opaque axis positions.

In this paper, we present Sparse [Spa18], a sparse array library that supports arbitrary dimension sparse arrays and supports most common parts of the `ndarray` interface. It supports basic arithmetic, application of `ufunc` s directly to sparse arrays (including with broadcasting), most common reductions, indexing, concatenation, stacking, transpose, reshape and a number of other features. The primary format in this library is based on the coordinate format, which stores indices where the array is nonzero, and the corresponding data.

Since a full explanation of usage would be a repeat of the NumPy user manual and the package documentation, we move on to some of the design decisions that went into making this package, including some challenges we had to face some optimizations, applications and possible future work.

Algorithms and Challenges

Choice of storage format

We chose the COO format for its simplicity while storing and accessing elements, even though it isn't the most efficient storage format. In this format, two dense arrays are required to store the sparse array's data. The first is a coordinates array, which stores the coordinates where the array is nonzero. This array has a shape `(ndim, nnz)`. The second is a data array, which stores the

* *Corresponding author: hameerabbasi@yahoo.com*
‡ *TU Darmstadt*

dim1	dim2	dim3	...	data
0	0	0	...	10
0	0	3	...	13
0	2	2	...	9
...	...	...	...	...
3	1	4	...	21

TABLE 1
A visual representation of the COO format.

data corresponding to each coordinate, and thus it has the shape `(nnz,)`. Here, `ndim` represents the number of dimensions of the array and `nnz` represents the number of nonzero entries in the array.

For simplicity of operations in many cases, the coordinates are always stored in C-contiguous order. Table 1 shows a visual representation of how data is stored in the COO format.

We use whatever data-type the source array has for the data array and `np.int64` for the coordinates array. This means that, assuming `ndim = 3` and `dtype.itemsize = 8` (as is the case for a data type of `np.int64`, `np.uint64` and `np.float64`), the tipping point versus dense arrays for memory usage will be a density of `0.25`, with the benefit increasing with the inverse of the density.

Element-wise operations

Element-wise operations are an important and common part of any array interface. For example, arithmetic, casting an array, and all NumPy `ufunc` s are common examples of element-wise operations.

These turn out to be simple for NumPy arrays, but are surprisingly complex for sparse arrays. The first problem to overcome was that there was no dependency on Numba [Ana18]/Cython [Cyt18]/C++ at the time that this algorithm was to be implemented, and a discussion was ongoing about which algorithm to use. [Spae] I, therefore wished to solve the problem in pure NumPy, therefore looping over all possible nonzero coordinates was not an option, and we had to process the coordinates and data in batches. The batches that made sense at the time were something like the following:

1) Coordinates in the first array but not in the second.
2) Coordinates in the second array but not in the first.
3) Coordinates in both arrays simultaneously.

This algorithm (when applied to multiple inputs instead of just two) looks like the following:

```
all_coords = []
all_data = []

for each combination of inputs where some are zero
    and some nonzero:
    if all inputs are zero:
        continue

    coords = find coordinates common to
             nonzero inputs
    coords = filter out coordinates that are
             in zero inputs
    data = apply function to data corresponding
           to these coordinates

    all_coords.append(coords)
    all_data.append(data)

concatenate all_coords and all_data
```

The addition of broadcasting makes this problem even more complex to solve, as it turns out that for sparse arrays, simply broadcasting all arrays to a common shape and then performing element-wise operations is not the most efficient way to perform such an operation.

Consider two arrays, one shaped `(n,)` and another shaped `(m, n)`, both with only one nonzero entry. If all we wanted to do was multiply them, the result would have just one nonzero entry, yet broadcasting the first array would result in an array with `m` nonzero entries (which clearly isn't the most optimal way to do things). For this reason, we chose to handle broadcasting within the algorithm itself, instead of broadcasting all inputs upfront.

Effectively, this resulted in the following algorithm, which doesn't have the limitation mentioned above. This is because any zeros are filtered out before any broadcasting is done:

```
all_coords = []
all_data = []

for each combination of inputs where some are zero
    and some nonzero:
    if all inputs are zero:
        continue

    coords = find coordinates common to
             nonzero inputs
             (for dimensions that are not being
             broadcast in both, with repetition
             similar to an SQL outer join)
    data = apply function to data corresponding
           to these coordinates

    coords, data = filter out zeros from coords/data

    coords, data = filter out coordinates/data that
                   are in zero inputs
                   (again, for non-broadcast dimensions)

    broadcast coordinates and data to output shape

    all_coords.append(coords)
    all_data.append(data)

concatenate all_coords and all_data
```

The full implementation can be found in [Spaa]. While this algorithm is effective at applying all sorts of element-wise operations for any amount of inputs, it does have a few drawbacks:

- It's slower than `scipy.sparse`, because
  - It loops over all possible combinations of zero/nonzero coordinates, which makes it $O\left(\left(2^{\text{nin}} - 1\right) \times \text{nnz}\right)$ in the worst case, where `nin` is the number of inputs to the operation and `nnz` are the number of nonzero elements.
  - It's in COO format rather than CSR/CSC.
  - `scipy.sparse` uses specialized code paths for each operation that greatly reduce the strain on the CPU whereas we keep everything generic.
- In the current implementation, sorting of coordinates is sometimes done unnecessarily.

This can be improved in the future in the following ways:

- Looping over possibly nonzero coordinates with something like Numba or Cython.

  - This approach will solve most of the speed issues.
  - Sorting will be rendered unnecessary.
  - Specialized code paths introduce a large maintenance burden, but can be implemented.

- Introducing multidimensional CSR/CSC.

You can see the current performance of the code in Table 2.

Currently, the implementation raises a `ValueError` if `ndarray` s are mixed with sparse arrays, or if the operation produces a dense array, such as operations like `y = x + 5` where `x` is sparse. This is an intentional design choice: We raise an error to show that the result is likely dense, and that if the user wishes to perform a dense operation, they should convert all arrays involved to dense ones and repeat the operation. This is better than an undesired performance degradation, which can be hard to detect.

However, work is being done to reduce the amount of such errors. For example, there is a feature planned to allow mixed `ndarray` -sparse operations if such operations do not produce dense results e.g. multiplication. [Spac]. Also, we are planning to allow arbitrary fill values in arrays, which will allow for operations such as `y = x + 5` (if `x.fill_value` was zero, `y.fill_value` will be five). [Spad]

Reductions

We implemented reductions by the elegant concept of a "grouped reduce". The idea is to first group the coordinates by the non-selected axes, and then reduce along the selected axes. This is simple to implement in practice, and also works quite well. Here is some psuedocode that we use for reductions:

```
x = x.transpose((selected_axes, non_selected_axes))
x = x.reshape((selected_axes_size,
               non_selected_axes_size))

y, counts = perform a reduce on x
            grouped by the first coordinate
            using ufunc.reduceat
where counts < non_selected_axes_size, reduce
    an extra time by zero

y = y.reshape(non_selected_axes_shape)
```

The full implementation can be found at [Spab]. Only some reductions are possible with this algorithm at the moment, but most common ones are supported. Supported reductions must have a few properties:

- They must be implemented in the form of `ufunc.reduce`
- The `ufunc` must be reorderable
- Reducing by multiple zeros shouldn't change the result
- An all-zero reduction must produce a zero.

Although these criteria seem restricting, in practice most reductions such as `sum`, `prod`, `min`, `max`, `any` and `all` actually fall within the class of supported reductions. We used `__array_ufunc__` protocol to allow application of `ufunc` reductions to COO arrays. Notable unsupported reductions are `argmin` and `argmax`, because they cannot be implemented in the form `ufunc.reduce`.

This is nearly as fast as the reductions in `scipy.sparse` when reducing along C-contiguous axes, but is slow otherwise. Performance results can be seen in Table 2. Profiling reveals that most of the time in the slow case is taken up by sorting, as `ufunc.reduceat` expects all "groups" to be right next to each other. This can be improved in the following ways:

- Implement a radix argsort, which will significantly speed up the sorting.
- Perform a "grouped reduce" by other methods, such as how Pandas does it, perhaps by using a `dict` to maintain the results.

Indexing

For indexing, we realize that to construct the new coordinates and data, we can perform two kinds of filtering as to which coordinates will be in the new array and which ones won't.

The first is where we look at the coordinates directly, and then filter them out successively for each given index. For integers, we check for coordinates that are exactly equal to that index. For slices, we similarly check for matching coordinates. We do this for each index. This turns out to be $O(\text{ndim} \times \text{nnz})$ in total. where `ndim` is the number of dimensions of the array to the operation and `nnz` are the number of nonzero elements.

This has a few benefits: it is simple to do and the performance only depends on the size of the input array.

The second is where we look at each integer index in series, and then look at *sub-arrays* for each integer index. Since the coordinates are sorted in lexographical order, we will have to do a binary search for the start and end of each sub array, and repeat this for each integer index within the previous sub-array. Getting a single item or an integer slice in this case is $O(\text{nidx} \times \log \text{nnz})$. Here, `nidx` is the number of provided integer indices. For slices, we will loop over each possible integer in the slice and repeat the above procedure.

For integer indexing, the second method is almost always faster. For slices, the situation becomes more complicated. Even for slices, in some cases, it is faster to use the second procedure. This happens for small slices, e.g. `x[:10]`.

For other cases, it's wise to initially use the second procedure (to filter out some sub-arrays), and then switch to the first. For example, for `x[:500, :500, :500]`, as using just the second procedure will require a large amount of binary searches (500^3 in this case).

So we used a hybrid approach where the second method is used until there are a sufficiently low number of coordinates left for filtering, then we fall back to simple filtering. Where we do the switch is determined by a heuristic: will the expected number of binary searches be faster in a specific case, or directly filtering the number of left-over coordinates? The overall algorithm is implemented in Numba, because when this algorithm was implemented, the discussion in [Spae] had been resolved. However, it has since been reopened due to further missing features in Numba.

After getting the required coordinates and corresponding data, we apply some simple transformations to it to get the output coordinates and data.

However, one thing is important to realize: indexing sparse arrays is more expensive than indexing dense arrays. Indexes of dense arrays produce a view for any combination of slices and integers, and take $O(\text{nidx})$ time in every case. Sparse arrays take more time, and it's usually not possible to produce a view of the original array.

Benchmark	Sparse	SciPy Sparse	NumPy
Addition	50.8 ms ± 3.45 ms	2.49 ms ± 211 µs	507 ms ± 6.43 ms
Multiplication	10.7 ms ± 526 µs	14.9 ms ± 1.68 ms	529 ms ± 13.5 ms
Sum, Axis=0	12 ms ± 116 µs	545 µs ± 49.8 µs	97.8 ms ± 4.19 ms
Sum, Axis=1	959 µs ± 23.7 µs	641 µs ± 83.9 µs	62.7 ms ± 4.86 ms

TABLE 2
Performance benchmarks comparing Sparse to SciPy and dense NumPy code

Transposing and Reshaping

Transposing corresponds to a simple reordering of the dimensions in the coordinates, along with a re-sorting of the coordinates and data to make the coordinates C-contiguous again.

Reshaping corresponds to linearizing the coordinates and then doing the reverse for the new shape, similar to `np.ravel_multi_index` and `np.unravel_index`. However, we write our own custom implementation for this.

Matrix and tensor multiplication

For `tensordot`, we currently just use the NumPy implementation, replacing `np.dot` with `scipy.sparse.csr_matrix.dot`. This is mainly just transposing and reshaping the matrix into 2-D, using `np.dot` (or `scipy.sparse.csr_matrix.dot` in our case), and performing the reshape and transpose operations in reverse.

For `sparse.dot`, we simply dispatch to `tensordot`, providing the appropriate axes.

This may not always produce a sparse array as output. If we think of each element of the output matrix as a dot product of the appropriate row of the first matrix and the appropriate column of the second matrix, we realize that it may be difficult to guarantee that this will be zero. Indeed, in general, $\text{nnz}_{\text{out}} \leq \text{nnz}_{\text{in1}} \times \text{nnz}_{\text{in2}}$, without knowing much about the structure of the matrix. For some inputs however, the outputs will be relatively sparse (for example for identity matrices and diagonal matrices).

Benchmarks

Because of our desire for clean and generic code as well as using mainly pure Python as opposed to Cython/C/C++ in most places, our code is not as fast as `scipy.sparse.csr_matrix`. It, however, does beat `numpy.ndarray`, provided the sparsity of the array is small enough. The benchmarks were performed on a laptop with a Core i7-3537U processor and 16 GB of memory. Any arrays used had a shape of `(10000, 10000)` with a density of `0.001`. The results are tabulated in Table 2.

The NumPy results are given only for comparison, and for the purposes of illustrating that using sparse arrays does, indeed, have benefits over using dense arrays when the density of the sparse array is sufficiently low.

Outlook and Future Work

We discussed the current leading solution for sparse arrays in the ecosystem, `scipy.sparse`, along with its shortcomings and limitations. We then introduced a new package for N-dimensional sparse arrays, and how it has the potential to address these shortcomings. We discuss its current implementation, including the algorithms used in some of the different operations and the limitations and drawbacks of each algorithm. We also discuss future improvements that could be made to improve these algorithms.

There are a number of areas we would like to focus on in the future. These include, in very broad terms:

- Better performance
- Better integration with community packages, such as scikit-learn, Dask and XArray
- Support for more of the `ndarray` interface (particularly through protocols)
- Implementation of more linear algebra routines, such as `eig`, `svd`, and `solve`
- Implementation of more sparse storage formats, such as a generalization of CSR/CSC

References

[Ana18] Anaconda, Inc. Numba, 2018. URL: https://numba.pydata.org/.

[Cyt18] Cython developers. Cython, 2018. URL: http://cython.org/.

[Das18] Dask core developers. Dask, 2018. URL: https://dask.pydata.org/en/latest/.

[LN89] Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[MR91] Mo Mu and John R Rice. An organization of sparse Gauss elimination for solving PDEs on distributed memory machines. 1991.

[NTK11] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *ICML*, volume 11, pages 809–816, 2011.

[Num18] NumPy developers. Numpy, 2018. URL: https://www.numpy.org/.

[Sci18] SciPy developers. Scipy, 2018. URL: https://www.scipy.org/.

[Spaa] Sparse developers. Sparse Implementation: Elemwise. URL: https://github.com/pydata/sparse/blob/b51d74924d62ff6537b15ce4e1dd4e56080a3b6f/sparse/coo/umath.py#L12.

[Spab] Sparse developers. Sparse Implementation: Reductions. URL: https://github.com/pydata/sparse/blob/b51d74924d62ff6537b15ce4e1dd4e56080a3b6f/sparse/coo/core.py#L564.

[Spac] Sparse developers. Sparse Issue: Allow ndarray in elemwise again. URL: https://github.com/pydata/sparse/issues/124.

[Spad] Sparse developers. Sparse Issue: Arbitrary fill value. URL: https://github.com/pydata/sparse/issues/143.

[Spae] Sparse developers. Sparse Issue: Use Cython, Numba, or C/C++ for algorithmic code. URL: https://github.com/pydata/sparse/issues/143.

[Spa18] Sparse developers. Sparse, 2018. URL: https://sparse.pydata.org/en/latest/.

[Tip01] Michael E Tipping. Sparse Bayesian learning and the relevance vector machine. *Journal of machine learning research*, 1(Jun):211–244, 2001.

[xar18] xarray Developers. xarray, 2018. URL: https://xarray.pydata.org/en/stable/.