Pydata/Sparse: Merge lattices to enable C++ compile-time dispatch of sparse ndarrays

Sub-Organization Information:

Pydata/Sparse

Student Information:

Name: Adam Li
Time zone: EST
Source control username: http://github.com/adam2392/
GSoC Blog RSS feed (tentative): https://blogs.python-gsoc.org/en/adams-blog

Code Samples:

- https://github.com/hameerabbasi/xsparse/issues/15
- https://github.com/hameerabbasi/xsparse/pull/16#event-8585625793
- https://github.com/hameerabbasi/xsparse/issues/17

University Information:

University: Johns Hopkins University
Major: Biomedical Engineering
Graduation Date: December 2021
Degree: PhD

University: Columbia University
Department: Computer Science
Tentative Finish Date: January 2024
Current Academic Position*:* Postdoctoral Research Scientist

Link to academic CV: https://adam2392.github.io/pdfs/Personal_CV.pdf

Project Proposal Information:

Proposal Title:

Pydata/Sparse: Implementation, testing and documentation of merge lattices to enable C++ compile-time dispatch of sparse ndarrays

Proposal Abstract:

There is currently no robust and unified sparse ndarray library that is compliant with the numpy API. This prevents the substitution of sparse arrays for dense arrays when needed in many data science workflows. PyData/Sparse is a repository that presents an opportunity to address these issues. In order to do so, there needs to be a C++ implementation of a code-generation algorithm that can generate efficient code for any type of sparse tensor operation. This abstraction allows one to maintain a relatively small piece of code that enables functionality for an exponential number of different sparse array format operations. The main objective of the project is to implement, test and document the merge lattice and its related data structures and algorithms that are vital for code-generation. This will result in a generalizable code-generation algorithm that can be applied for implementing a variety of different sparse ndarray formats that work together. This will involve interfacing heavily with the existing C++ code and its implemented templates to add functionality that constructs and manipulates a merge lattice, which is a useful abstraction in code-generation for sparse tensor algebra operations. As a few stretch goals, I plan on: i) implementing abstractions of what a sparse array is and what operations can be done that leverage this underlying core API, ii) finishing the co-iteration algorithm to account for non-unique and non-ordered levels and iii) begin work on the actual code-generation algorithm that leverages the merge lattice.

Detailed Description:

Introduction:

Sparse array data structures are commonly used to save memory when storing massive high-dimensional multi-dimensional arrays. These take advantage of the fact that zeros will not affect computation in addition, subtraction, and multiplication and need not be stored explicitly in memory. Sparse arrays thus have wide utility in physics and simulations, electron microscopy research and even Covid-19 research.

The currently "most-used" sparse array implementation in Python is in scipy [1]. However, the current implementation of scipy is limited in many aspects. Firstly, the API is based off of a deprecated "numpy.matrix" API. Secondly, its current implementation only supports 2D arrays, rather than n-dimensional arrays (ndarrays), which is the currently supported API in "numpy.ndarray". Thirdly, it only supports a limited range of sparse array formats (e.g. COO, DOK, CSR, CSC), whereas there are a wide variety of other sparse array formats used, such as DIA, ELL, DCSR, CSB, and more [2]. Finally, there are numerous performance improvements that can be made when compared to other state-of-the-art

sparse array implementations [3]. A robust, high-performance library that is compliant with the "numpy.ndarray" API (thereby supporting ndarrays) and supports all widely used sparse array formats is needed in the Python community.

Pydata/sparse - https://github.com/pydata/sparse is a new emerging library that supports the "numpy.ndarray" API and implements a few of the common sparse array formats, such as DOK, COO, CSR and CSC. The library currently is primarily written in Python and leverages Numba for improved performance. However, even with Numba, the library is not as fast as scipy's 2D sparse array implementations, which are written in a compiled language. Therefore, there is a current push to implement the sparse arrays in C++ at xsparse (https://github.com/hameerabbasi/xsparse), which can then be exposed to Python via bindings.

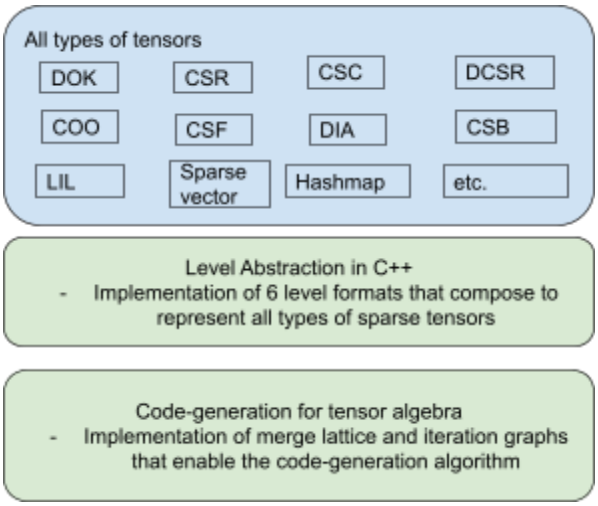

Figure 1: A diagram that demonstrates the different levels of abstraction, sparse will implement. Starting conceptually from all possible types of sparse tensors that are commonly used (top), sparse implements an abstraction of different level formats that can then be composed to represent all types of sparse tensors (middle). Finally, optimized code will be generated for tensor algebra operations, which will be implemented through the use of merge lattices and iterations graphs (bottom).

As specified earlier, there are many sparse array formats, which all have different advantages and disadvantages. Supporting computation with all and any combination of them is important for a generic sparse array library. This is difficult as highlighted in [4] because there is an exponential number of combinations in the number of formats, and thus writing compiled code for each format combination is infeasible, especially given different array functions one might want (e.g. exp, log, addition, subtraction, iteration, etc.). A scalable method for abstracting sparse ndarray functionality into six "level formats" was introduced in [2,4], which is the approach taken in xsparse. Each level format abstraction hides the details of how the level format encodes sparse array dimensions behind a common interface. By implementing this interface and the six level format abstractions, we can generate code during compilation that efficiently performs computation on sparse arrays stored in any combination of formats. Figure 1 gives a high-level schematic overview of the layers at which XSparse operates. There are many sparse tensor formats that one might want to support, but rather than explicitly implementing each one (blue box), XSparse plans to

implement abstractions and a code-generation algorithm (green boxes) that can generate optimized code for operations that sparse tensors would use.

Beyond the level format abstractions, there are additional algorithmic details that are important for writing the final code-generation algorithm. Co-iteration is a core algorithm that needs to be implemented for all combinations of level formats, where we are interested in "simultaneously" iterating through levels. This functionality is useful for "merging" different level formats together when creating the final output sparse tensor. For example, when taking the dot product of two sparse vectors, we are only interested in non-zero indices of the two vectors that are the same because otherwise, the contribution of that vector's index is 0 (multiplying by 0 equals 0). Generalizing this to sparse ndarrays (or tensors) leverages the merge lattice and iterator graph abstraction that was introduced in [3].

The goal of this part-time GSoC is to implement merge lattices and their related data structures and algorithms that integrate seamlessly with the existing level format C++ templates. Moreover, modification of the existing co-iteration algorithm to account for all level formats, specifically unordered and non-unique levels is necessary. We will conclude with an initial implementation of the code-generation algorithm if time permits.

Existing code

XSparse currently contains implementations that are mostly complete of the level format abstractions. Figure 2 shows a breakdown of the XSparse level format abstractions in C++, where representations, access capabilities, properties and construction are addressed. These six core level formats are composable to represent any sparse tensor. For example, a DOK sparse matrix is a composition of two hashed levels.

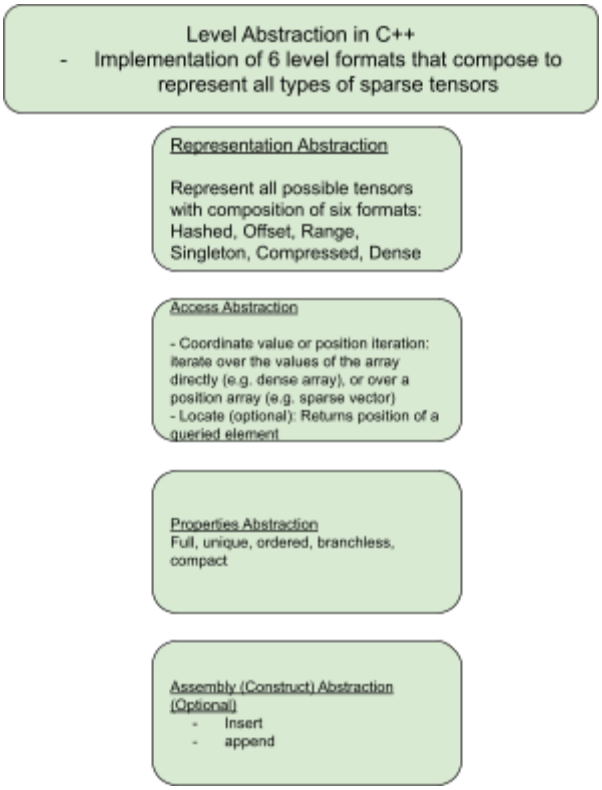

Figure 2: Details on the level format abstractions in C++ and their representation, properties and functional capabilities.

We have the following already implemented C++ level types, where Table 1 shows varying capabilities. Note, the current codebase requires C++20 and above since we use relatively new features of the C++ language.

Level Type	Level Function Definitions	

Dense	`insert_coord(`p_k`, `i_k`):`
	`  // do nothing`
	`size(`sz_{k-1}`):`
	`  return `sz_{k-1}` * `N_k
		`insert_init(`sz_{k-1}`, `sz_k`):`
		`  // do nothing`
		`insert_finalize(`sz_{k-1}`, `sz_k`):`
		`  // do nothing`

Compressed	`append_coord(`p_k`, `i_k`):`
	`  crd[`p_k`] = `i_k
	`append_edges(`p_{k-1}`, `$pbegin_k$`, `$pend_k$`):`
	`  pos[`p_{k-1}` + 1] = `$pend_k$` - `$pbegin_k$
		`append_init(`sz_{k-1}`, `sz_k`):`
		`  for (int `p_{k-1}` = 0; `p_{k-1}` <= `sz_{k-1}`; ++`p_{k-1}`) {`
		`  pos[`p_{k-1}`] = 0`
		`  }`
		`append_finalize(`sz_{k-1}`, `sz_k`):`
		`  int cumsum = pos[0]`
		`  for (int `p_{k-1}` = 1; `p_{k-1}` <= `sz_{k-1}`; ++`p_{k-1}`) {`
		`  cumsum += pos[`p_{k-1}`]`
		`  pos[`p_{k-1}`] = cumsum`
		`  }`

Singleton	`append_coord(`p_k`, `i_k`):`
	`  crd[`p_k`] = `i_k
	`append_edges(`p_{k-1}`, `$pbegin_k$`, `$pend_k$`):`
	`  // do nothing`
		`append_init(`sz_{k-1}`, `sz_k`):`
		`  // do nothing`
		`append_finalize(`sz_{k-1}`, `sz_k`):`
		`  // do nothing`

Hashed	`insert_coord(`p_k`, `i_k`):`
	`  crd[`p_k`] = `i_k
	`size(`sz_{k-1}`):`
	`  return `sz_{k-1}` * `W_k
		`insert_init(`sz_{k-1}`, `sz_k`):`
		`  for (int `p_k` = 0; `p_k` < `sz_k`; ++`p_k`) {`
		`  crd[`p_k`] = -1`
		`  }`
		`insert_finalize(`sz_{k-1}`, `sz_k`):`
		`  // do nothing`

Table 1: From [2] a definition of level functions that implement the assembly capabilities for various level types.

Proposed implementation of code-generation algorithm

The current proposal is to complete (or make significant progress on) the code-generation algorithm. The code-generation algorithm requires the implementation of core data structures: iteration graphs and the merge lattice. Once this is implemented, the code-generation algorithm proceeds by calling and manipulating the merge lattice to emit optimized C++ code (see Figure 3).

```
code-gen(index-expr, iv)  # iv is the index variable
  let L = merge-lattice(index-expr, iv)

  # initialize sparse pos variables
1 for Dj in sparse-dimensions(L)
    emit “int pDj = Dj_pos[pDj-1];”

  for Lp in L
2   # while all merged dimensions have more values
    emit “while(until-any-exhausted(merged-dimensions(Lp))) {”

    # initialize sparse idx variables
3   for Dj in sparse-dimensions(Lp)
      emit “int ivDj = Dj_idx[pDj];”

    # merge sparse idx variables
4   emit   “int iv = min([“ivDj,” Dj in sparse-dimensions(Lp)]);”

    # compute dense pos variables
5   for Dj in dense-dimensions(Lp)
      emit “int pDj = (pDj-1 * Dj_size) + iv;”

    # compute expressions available at this loop level
6   emit-available-expressions(index-expr, iv)    # Section 6.2

    # one case per lattice point below Lp
    for Lq in sub-lattice(Lp)
      emit “if (equals-iv([“ivDj” Dj in sparse-dimensions(Lq)])) {”
        for child-iv in children-in-iteraton-graph(iv)
          code-gen(expression(Lq), child-iv)
7       emit-reduction-compute()  # Section 6.2
        emit-index-assembly()     # Section 6.3
        emit-compute()            # Section 6.2
        if result dimension Dj is accessed with iv
          emit “pDj++;”
      emit “}”

    # conditionally increment the sparse pos variables
8   for Dj in sparse-dimensions(Lp)
      emit “if (ivDj == iv) pDj++;”
2   emit “}”
```

Figure 3: Code-generation recursive algorithm for tensor expressions.

Merge Lattices:

A merge lattice, *L*, is a lattice structure comprised of *n* lattice points, *L1,...,Ln,* and a merge operator. Each lattice point is associated with a set of tensor dimensions, $T_p = \{t_{p_1},..., t_{p_k}\}$, that are merged conjunctively (i.e. intersection; $\{t_{p_1} \wedge .. \wedge t_{p_k}\}$) and an expression that is to be evaluated. The meet operator takes two lattice points and their associated tensor dimensions and creates a new lattice point with the union of the two tensor dimensions, $T_1 \cup T_2$. Next, I describe a bit of details on how merge lattices are tied to tensor operations.

A conjunctive merge corresponds to a multiplication of tensors, whereas a disjunctive merge corresponds to an addition of tensors. When accessing dimensions of more than one tensor, one needs to iterate over the non-zero indices (i.e. the merged indices) of the resulting tensor's dimensions. The merge lattice is then an abstraction that represents how to combine indices that are coiterated over in tensors into the final resulting tensor.

Next, we also motivate why the merge lattice abstraction is useful. Consider the disjunctive merge operation, where every loop iteration must check that the merged index has more values left. The merge algorithm that is considered is a generalization of the two-way merge algorithm. For the disjunctive merge, this algorithm has three loops:

1. Iterate until either indices is exhausted (end is reached)
2. Iterate to process the rest of the unexhausted index i
3. Iterate to process the rest of the unexhausted index j

As an example of the disjunctive merge, consider Figure 4 taken from [3] where we want to add two sparse vectors (b + c). The disjunctive merge has three lattice points in the merge lattice data structure. Each lattice point corresponds to a loop. The first loop will coiterate over "b" and "c" simultaneously with index "i" and add merged dimensions to resulting sparse vector "a" if any of the following is true:

- "b" and "c" have non-zero value at "i"
- "b" has non-zero value at "i"
- "c" has non-zero value at "i"

Then the second and third loop will loop over the remainder of "b" and "c" respectively to add their values to "a".

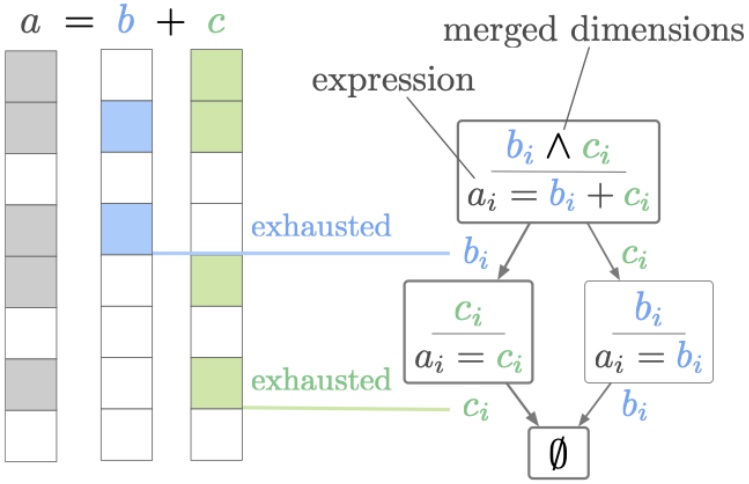

Figure 4: Example of merge lattice.

The implementation of the merge lattice will require constructing the data structures, which are just the level formats (i.e. lattice points are just abstractions for levels in this setting). Moreover, it will require integration of the existing coiteration code to support disjunctive and conjunctive merge operations for arbitrary combinations of level formats.

Proposed Abstractions of a sparse ndarray and what operations can be done

Following the numpy "ndarray" API, we can also implement a preliminary sketch of what the resulting Python API would be that performs operations on sparse ndarrays. Since we have numpy as a reference API, we would essentially replicate that API to have the same names, inputs and outputs, such that from a user-perspective, they can have all code of their Python packages remain the same whether or not a sparse ndarray, or a numpy ndarray is passed.

The numpy ndarray API is at https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html, where there are a numerous amount of functions with the corresponding C/C++ API at https://github.com/numpy/numpy/tree/main/numpy/core/src. Rather than be overly ambitious and claim we can replicate the full functionality suite, we will focus on some common tensor operations, such as "addition", "subtraction", "multiplication" and "division".

Sparse ndarray Python API

The API syntax will look something like the following:

```
sparse.ndarray.__add__
sparse.ndarray.__subtract__
sparse.ndarray.__mul__
sparse.ndarray.__truediv__
```

Which is similar to the numpy ndarray API. Other good references are XTensor: https://xtensor.readthedocs.io/en/latest/.

Implementation Details:

Some notes on potential implementation roadblocks and challenges along with potential solutions are outlined below:

Level formats: Dense, compressed, hashed, singleton, offset, range are the different types of level formats that are implemented, each with different properties and capabilities as described above.

Level format capabilities: co-iteration algorithm, iteration algorithm, merge algorithm. These are core algorithms described in the core references in this proposal that are utilized for iterating and merging different level formats together. A good understanding of the existing code and the algorithms in these capabilities are necessary in order to design, implement and test a robust merge lattice.

Writing functional tests: We will be using CMake to compile tests and run the test-suite. The full test suite was demonstrated to be runnable and I even proposed a minor fix to integrate better with macOS devices.

Writing documentation for the implemented code: The current documentation is in the code itself. So for the merge lattice, I will strive to write concise docstrings that document some of the intricacies of the implementation and usage. Moreover, I will add existing documentation to the code where necessary.

Proposal Timeline:

Community Bonding: I plan on performing work in two directions before the coding begins:

1. Implement an API for level formats to get their parent levels: This is important for iterator chaining, when performing merging of non-unique levels (i.e. singletons). I.e. we want to do coiteration, but running some deduplication procedures. This is illustrated in Figure 7 of [2].
2. Implement conjunctive merge logic when an input to co-iteration is a hash map (i.e. unordered, but unique): The API that calls coiteration should and will eventually perform the following logic: If it is sortable, sort the level. Otw if it is not sortable, then it must be `has_locate == True` property. If the unordered level is part of a disjunctive merge, then the current thought is that we will convert it to a compressed level that can be sortable.
    a. If the unordered level is part of a conjunctive merge, we can iterate through the level that is ordered and use `locate` to check if the index exists in the hash level. If it

returns NULL, then we don't append to the output. If it does not return NULL, then we append the output with this index.

b. We want logic that implements this depending on if one of the level formats passed to `Coiterate` return `has_locate == True` and is not ordered.

Weekly Timeline:

The goal is the development of the "XSparse" level format abstractions that enable a robust efficient generalizable sparse ndarray across many different formats.

Week 1: May 29 - June 2

- Analyze what is up-to-date in `XSparse` repository and general missing areas of interest, such as a generalized co-iteration algorithm
- Analyze the current unit-tests that are in XSparse and where unit-tests for a generalized co-iteration would go, as well as the setup and teardown design
- Draft a more in-depth strategy for implementation in a Github Issue that will outline exactly what will be implemented and how to handle the co-iteration

Week 2: June 5

- Draft implementation of generalized coiteration algorithm handling non-unique, but ordered levels
    - Also draft unit tests that will test the full range of functionality

Week 3: June 12

- Begin work on merge lattices and designing the C++ template class and the necessary API

Week 4: June 19

- Finish implementation of generalized co-iteration algorithm
- Begin drafting merge lattice unit tests and algorithms

Week 5: June 26

- Continue work on merge lattice and unit tests

Week 6: July 3

- Continue work on merge lattice and unit tests

Week 7: July 10

- Write additional tests that test all level formats functionality and co-iteration algorithm together along with merge lattices

Week 8: July 17

- Draft sparse ndarray abstractions that rely on the fully implemented levels and level capabilities
- Add what operations can be done by sparse ndarrays:

Week 9: July 24

- Continue work on sparse ndarray abstractions

Week 10: July 31

- Continue work on sparse ndarray abstractions and finish unit tests

Week 11: August 7

- Continue work on sparse ndarray abstractions and finish unit tests
- Review merge lattice implementation and sparse ndarray abstractions to determine if there are missing gaps in the design

Week 12: August 14

- Continue work on sparse ndarray abstractions and finish unit tests

Conclusion

Overall, this project is a research-oriented project that is aimed at implementation of a very technical research paper. The benefits of implementing the merge lattice will enable xsparse to then begin writing the code-generation algorithm, which will enable arbitrary sparse tensor algebraic operations. The overall plan is well-posed for the implementation of the merge lattice and is expected to be finished in the allotted time span. I have allotted plenty of time to further research, study, test and iterate on the proposed merge lattice algorithms and data structures, so that it fits within the rest of the existing codebase of xsparse.

I have included a stretch goal of implementing the sparse ndarray abstractions that rely on the level formats and their capabilities. Having a consistent sparse ndarray API that is compliant with Numpy ndarray will i) increase the user base of PyData/Sparse and ii) enable more streamlined substitution of sparse arrays in currently existing dense array workflows. Moreover, leveraging the level formats proposed, this project can create light-weight generic abstractions for all sparse array formats that allow combinations of computations done on any of them. By writing the entire codebase in C++, we can leverage compile-time optimizations and expose an API that is highly optimized and competitive with the current performance in scipy.sparse.

Other commitments:

I am applying for a part-time GSoC, where I will most likely work full-time, but I have prior commitments, like my postdoc. I typically work a lot on nights, weekends and during the day if I am ahead of my other commitments.

Currently, I will also have certain travel and weeks tentatively pre-planned, such as:

- the week of June 6th
- July 4th weekend

References

1. Virtanen P, Gommers R, Oliphant TE, et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nat Methods*. 2020;17(3):261-272. doi:10.1038/s41592-019-0686-2
2. Chou S, Kjolstad F, Amarasinghe S. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc ACM Program Lang*. 2018;2(OOPSLA). doi:10.1145/3276493
3. The tensor algebra compiler | Proceedings of the ACM on Programming Languages. Accessed February 18, 2023. https://dl.acm.org/doi/10.1145/3133901
4. Kjølstad FB. *Sparse Tensor Algebra Compilation*. PhD Thesis. Massachusetts Institute of Technology; 2020.