SciPy

XSparse - A C++ sparse array library with optimized runtime performance

Details and Introduction

Name: Bharath Kanchi Kandadai
GitHub: https://github.com/bharath2438
Institution: Dayanand Sagar College of Engineering
Origin: Bangalore, Karnataka, India

I'm currently a 3rd year CSE undergrad at Dayanand Sagar College of Engineering. I was introduced to open-source libraries last year when I used SciPy clustering as a part of a course in college. I then began tinkering with Scipy where I happened to discover a bug in `test_propack.py` for which I drafted this PR. Around February this year, I came across the xsparse project in the ideas list and found it to be a challenging one, so I read the TACO paper that was recommended and once I felt confident about implementing it, I decided to apply for GSoC 2022 to work on this project.

Abstract

The main purpose of this project is to build a C++ library that allows efficient computation of sparse matrices stored in different formats such as COO, CSR, DIA etc.

The library should be able to generate code that unifies different formats and generate optimized loops to aid in computation.

Detailed Project Description

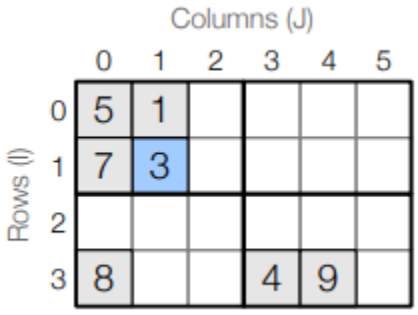
The two main issues with building an efficient computation technique are

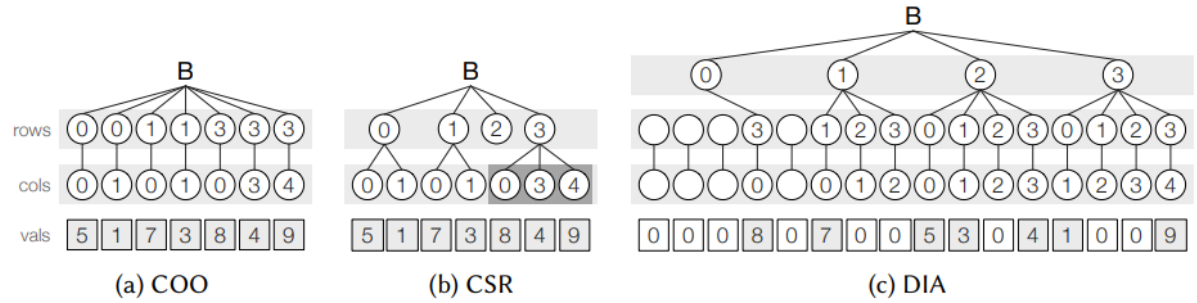
- The existence of different tensor formats that use their own data structures, and
- The existence of many combinations and it is impractical to write code for every possible combination

This issue can be solved with the introduction of a common method of abstraction - **coordinate hierarchies**.

Coordinate Hierarchies

Coordinate hierarchies help us differentiate the dimensions of a tensor into discrete **levels**. These levels are sufficient to represent most of the tensor formats that are used today.





Consider the simple 4x6 sparse matrix like the one above represented in 3 common level storage formats (as a coordinate hierarchy).

- The rows dimension of the CSR representation is a **dense** level as it encodes all possible coordinates in the dimension.

- The rows dimension of the COO representation is a **compressed** level as it encodes the bounds in an array called pos and the actual coordinates are encoded in an array called crd.

- The cols dimension of the COO representation is a **singleton** level as it has a 1:1 parent-child structure with the previous dimension.

- The row dimension of the DIA representation is a **range** level as it can be constructed using an offset array and the matrix dimensions.

- The column dimension of the DIA representation is an **offset** level as it is just shifted from its parent by values from an offset array.

- **Hashed** levels store a dimension as a hashmap with empty buckets marked by -1.

As discussed above, our main aim is to write a format-agnostic **code generation** procedure. For a start, we will have to establish some ground

rules regarding how we plan to index into levels and how to iterate over each level so as to increase computational performance.

To attack this problem, we introduce two parameters: **level capabilities** and **level properties**.

Level Type	Capabilities			Properties				
	Iteration	Locate	Assembly	Full	Ordered	Unique	Branchless	Compact
Dense	V	✓	I	✓	(✓)	(✓)		✓
Range	V				(✓)	(✓)		
Compressed	P		A	(✓)	(✓)	(✓)		✓
Singleton	P		A	(✓)	(✓)	(✓)	✓	✓
Offset	P				(✓)	(✓)	✓	
Hashed	P	✓	I	(✓)		(✓)		

Level Access Capabilities

These capabilities allow levels to access or modify coordinates. Each level type has its own set of capabilities as highlighted in the table above.
The abstract interface to a coordinate hierarchy level exposes three different access capabilities: **value iteration**, **position iteration** and **locate**.

Coordinate Value Iteration directly iterates over coordinates. It is implemented using two level functions: coord_bounds and coord_access. `coord_bounds` returns the coordinate bounds over which we need to iterate and `coord_access` returns the position of each coordinate in the given range.

Coordinate Position Iteration iterates over coordinate positions.
It is implemented using two level functions: pos_bounds and pos_access.

`pos_bounds` returns the position bounds over which we need to iterate and `pos_access` returns the coordinate encoded by each position in the given range.

`locate` capability allows random access of a coordinate by returning the position of a coordinate. It has similar semantics to that of `coord_access`.

Level Access Properties

These capabilities instruct the code generator to generate optimized loops to increase performance. Each level type has its own set of properties as highlighted in the table above.

Five properties are defined on coordinate hierarchies.

- A level is **full** if it encodes all the valid coordinates in a dimension. A dense level is a typical example of the full property.

- A **unique** level is a level that does not encode duplicate coordinates. A CSR matrix's row dimension is a unique level.

- A level is said to be **ordered** if all the coordinates are ordered in increasing order, lexicographically and by their parent positions. A CSR matrix's column dimension can be made to be ordered. However, a hashed level cannot be ordered.

- A level is **branchless** if no coordinate has a sibling and each coordinate in the previous level has a child. A COO matrix's column dimension does not have siblings and is branchless.

- If a level contains unlabeled coordinates, it is not considered to be **compact**, else it is said to be compact. A hashed level cannot be compact, as it encodes unlabeled coordinates as empty buckets.

Level Assembly Capabilities

Common level capabilities described above aid in indexing into levels and iterating over them. The level assembly capabilities focus on insertion of coordinates into a level.

Two types of insertion can be performed: **insert** and **append.**

The **insert** capability is exposed as four level functions - `insert_coord`, `insert_init`, `insert_finalize` and `size`.
`insert_coord` is used to insert a coordinate into a particular position provided by the `locate` function; `insert_init` is used to initialize data structures before insertion and `insert_finalize` is used to perform any post-processing operations; `size` returns the number of child coordinates, given a parent coordinate.

The **append** capability is again exposed as four level functions - `append_coord`, `append_init`, `append_finalize` and `append_edges`.
`append_coord` appends a coordinate to the end of a level; `append_init` and `append_finalize` perform the same function as insert_init and insert_finalize; `append_edges` is used to connect all children of a particular parent to their parent.

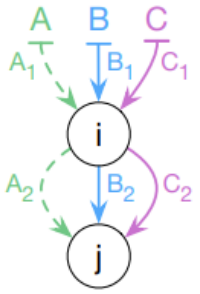
Code Generation

One of the core objectives of this project is the generation of code that would be able to handle computations between disparate formats. The code generation algorithm should ideally be able to accommodate even newer tensor storage formats.

Iteration Graphs and Merge Lattices

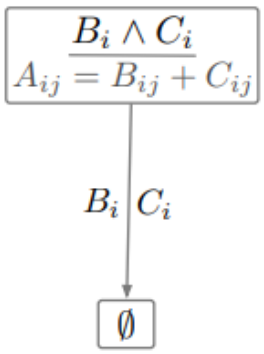
Consider computations like addition and multiplication, addition can be represented as Aij = Bij + Cij and multiplication can be represented as Aij = Σk Bik*Ckj.

From these tensor notations, we observe that there are multiple dimensions over which we can iterate together which will improve the looping time. The proper order in which we need to iterate over this joint iteration space is given by an **iteration graph**.
The figure below is an iteration graph for the notation Aij = Bij + Cij, where B is a CSR matrix and C is a COO matrix (no empty rows in C).

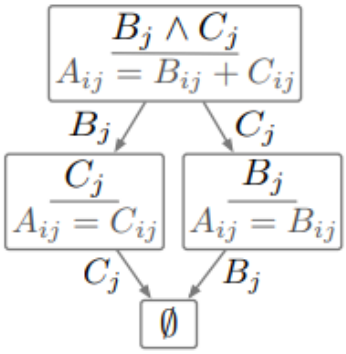


(a) Iteration graph

A **merge lattice** for a given dimension tells us about the loops which are required to efficiently merge that dimension from both matrices. The figure below gives us the merge lattice for the notation Aij = Bij + Cij where B is a CSR matrix and C is a COO matrix (no empty rows in C).



(b) Merge lattice for i



(c) Merge lattice for j

Level Iterator Conversion

Level iterator conversion is used to convert level iterators over unordered and non-unique levels to iterators with desired properties for which efficient algorithms exist.
Two methods can be used for this purpose - **deduplication** and **reordering**.

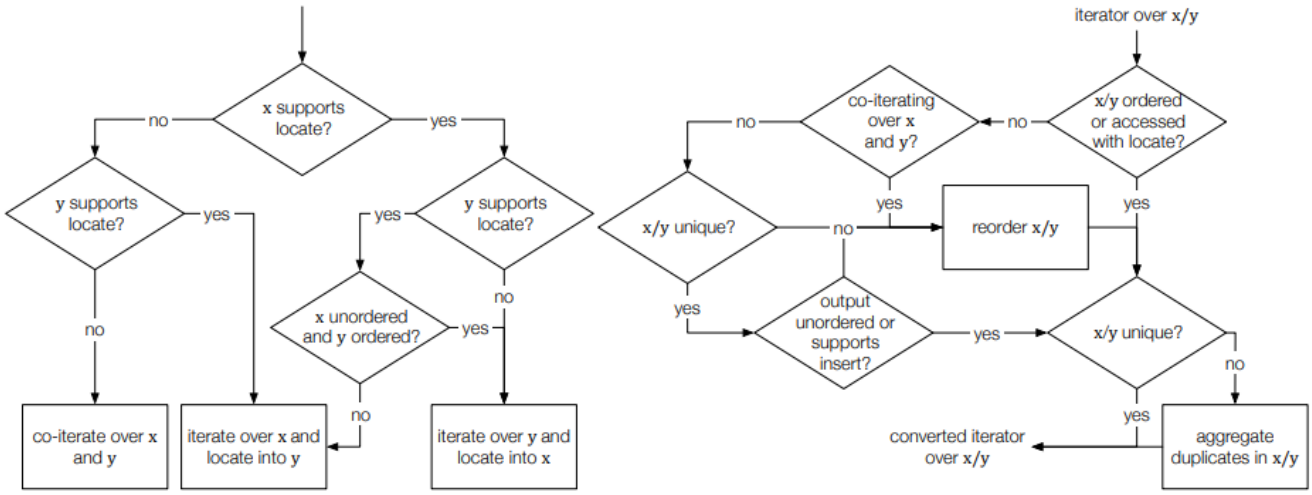
Deduplication: The main aim of this method is to remove duplicates from iterators using a deduplication loop. This helps us iterate over all children of a coordinate at once.

Reordering: Reordering uses a scratch array to store the ordered copy of an unordered level, and while co-iterating, the iterator can use the scratch array.

Level Merge Code

As we have established so far, carrying out computation between different level formats boils down to how we intend to merge the individual levels of these formats.
For this, we have to develop strategies for the code generation algorithm. The best factors to base these strategies on would be level capabilities and level properties.

For example, the figure below shows the strategy that our code generation algorithm would choose based on whether the levels are ordered, unique and they support locate capability.



Code Generation Algorithm

Finally, having understood the working of all the moving parts of the algorithm, we can integrate them into our code generation algorithm. Dividing the algorithm into discrete sections can help ease our understanding.

The algorithm works by generating code for each index level that is passed to it as input. (Refer diagram below)
Section 1 generates code to initialize iterators over coordinate positions or coordinate values.
Section 2 generates code for level iterator conversion, where a scratch array is used to store an ordered copy.
Next, the algorithm forms a merge lattice for the input index variable.
Section 4 starts a loop to iterate over every lattice point in the merge lattice

and merge the coordinate hierarchy levels corresponding to that lattice point.
Section 5, 7, 10 focus on dereferencing iterators that co-iterate over levels. **10** particularly helps generate code for chaining iterators over duplicate coordinates as discussed previously.
Section 8 helps calculating the next coordinate to be visited and **Section 9** exploits the locate capability provided by levels.
Section 11 increments every coordinate that has been visited so that they are not visited again by other iterators.

The algorithm generates loops to iterate over a single coordinate hierarchy level. However, this can be optimized by fusing iterators i.e allowing iterating over multiple coordinate hierarchy levels.

```
code-gen(index-expr, index-var):
  let L := merge-lattice(index-expr, index-var)

  for Dj in coord-value-iteration-dims(L):
    emit   "int ivDj, int Dj_end = coord_iter_Dj(ivD1,...,ivDj-1);"
  for Dj in coord-pos-iteration-dims(L):
    if Dj-1 is unique or iterator for Dj is fused:
1     emit "int pDj, int Dj_end = pos_iter_Dj(pDj-1);"
    else:
      emit "int pDj, _ = pos_iter_Dj(pDj-1);"
      emit "_, int Dj_end = pos_iter_Dj(Dj-1_segend - 1);"
  for Dj in noncanonical-dims(L):
    emit-scratch-array-assembly(Dj)
2   if Dj is unordered:
      emit "sort(Dj_scratch, 0, Dj_end);"
    emit   "int itDj = 0;"

3 if result dimension Dj indexed by iv, supports append, is branching:
    emit   “int pbeginDj = pDj;"
  for Lp in L:
    if iterator for each Dj in coiter-dims(Lp) is unfused:
4     let cdims := coiter-dims(Lp)  # co-iterated dimensions
      emit “while(all(["{p|it|iv}Dj < Dj_end" for Dj in cdims])) {”

    for Dj in coord-value-iteration-dims(Lp):
      emit   “int pDj, bool fDj = coord_access_Dj(pDj-1,...,ivDj);"
      emit   "while (!fDj && ivDj < Dj_end)"
      emit     "pDj, fDj = coord_access_Dj(pDj-1,...,++ivDj);"
5   for Dj in coord-pos-iteration-dims(Lp):
      emit   “int ivDj, bool fDj = pos_access_Dj(pDj,...,ivDj-1);"
      emit   "while (!fDj && pDj < Dj_end)"
      emit     "ivDj, fDj = pos_access_Dj(++pDj,...,ivDj-1);"
6   emit     "if(all([“fDj" for Dj in canonical-coiter-dims(Lp)])) {"
    for Dj in noncanonical-dims(Lp):
7     emit     "int ivDj = Dj_scratch[itDj].i;"
      emit     "int pDj = Dj_scratch[itDj].p;"

8   emit       “int iv = min([“ivDj" for Dj in coiter-dims(Lp)]);"
9   for Dj in locate-dims(Lp):  # dimensions accessed with locate
      emit     "int pDj, bool fDj = locate_Dj(pDj-1,...,iv);"

    for Dj in noncanonical-dims(Lp) U coord-pos-iteration-dims(Lp):
10    emit     "int Dj_segend = {p|it}Dj + 1;"
      if Dj is not unique and iterator for Dj is unfused:
        emit-deduplication-loop(Dj)

    emit-available-expressions(index-expr, iv)
    if result dimension Dj indexed by iv, supports insert:
      emit     "int pDj, _ = locate_Dj(pDj-1,...,iv);"
    for Lq in sub-lattice(Lp):  # a case per lattice point below Lp
      let cdims := coiter-dims(Lq) \ full-dims(Lq)
      let ldims := locate-dims(Lq) \ full-dims(Lq)
      emit     “if (all([“ivDj == iv" for Dj in cdims]) &&
                    all(["fDj" for Dj in ldims])) {”
      for child-index-var in children-in-iteraton-graph(index-var):
        code-gen(expression(Lq), child-index-var)
      emit-compute-statements()
3     if result dimension Dj indexed by iv and Dj+1 not branchless:
        emit     “{insert|append}_coord_Dj(pDj,iv);"
        if Dj supports append:
          emit   "pDj++;"
        while Dj is branchless:
          if Dj supports append:
            emit “append_edges_Dj(pDj-1,pDj - 1,pDj);
          Dj := Dj-1  # parent dimension in output hierarchy
          emit   “append_coord_Dj(pDj,iv);"
          emit   "pDj++;"
      emit     “}"

    for Dj in coiter-dims(Lp):
      if Dj is not full:
        emit   "if (ivDj == iv) "
11    if Dj in coord-value-iteration-dims(Lp):
        emit     "ivDj++;"
      else:
        emit     "{p|it}Dj = Dj_segend;"
6   emit     "}"
4   if iterator for each Dj in coiter-dims(Lp) is unfused:
      emit "}"
3 if result dimension Dj indexed by iv, supports append, is branching:
    emit   “append_edges_Dj(pDj-1,pbeginDj,pDj);"
```

(a) Algorithm to generate tensor algebra code.

Plan of Action: C++ implementation

The main aim of the xsparse library is to implement the level formats, their capabilities and properties and code generation.
For this, the idea is to use compile time features of C++ like templates and design patterns.

Levels

`dense.hpp` implements the dense level format.

The class `dense` defines the level capabilities provided by dense and also the functions that are required for the same.

```
class dense<std::tuple<LowerLevels...>, IK,
PK>::public
level_capabilities::coordinate_value_iterate
<dense, std::tuple<LowerLevels...>, IK, PK>
```

We can iterate over a dense level using coordinate value iteration, this level capability is defined in `coordinate_iterate.hpp`.

```
inline std::pair<IK, IK> coord_bounds([[maybe_unused]]
typename BaseTraits::I i)

inline std::optional<PK> coord_access(typename
BaseTraits::PKM1 pkm1, [[maybe_unused]] typename
BaseTraits::I i, IK ik)
```

The functions `coord_bounds` and `coord_access` are implemented as inline functions.

In this way, just like the dense level format, compressed, range, singleton, offset and hashed need to be implemented.

Level Capabilities

Coordinate position iteration and value iteration are implemented in `coordinate_iterate.hpp`. Operators for each type are inline functions.

```
template <template <class...> class T, class...
LowerLevels, class... Opts>
class coordinate_value_iterate<T,
std::tuple<LowerLevels...>, Opts...>

inline iterator& operator++() noexcept
{
    ++m_ik;
    return *this;
}
```

`Assembly.hpp` must implement the **insert** and **append** capabilities. **Insert** needs the following functions to be implemented - `insert_coord(), insert_init(), insert_finalize()` and `size()` and **Append** requires `append_coord(), append_init(), append_finalize()` and `append_edges()`.

Locate needs to be implemented in `locate.hpp`.

Level Properties

`Level_properties.hpp` defines the 5 level properties discussed above. They are crucial for developing strategies to aid in generation of optimal code.

Code Generation

The chunk of the stipulated time needs to be spent on code generation and optimization. The idea is to use **templates, template metaprogramming, constant expressions, type traits and the Curiously Recurring Template Pattern (CRTP design pattern)**.

The most important aim of our code generation implementation is to achieve genericity and maintainability. **Template metaprogramming** and **constant expressions** can help in achieving these objectives.

Instead of computing functions at runtime, metaprograms generate source code at compile time which facilitates functions to be executed at compile-time. This gives us **compile-time polymorphism** with the only limitation being that all data required by the function is available at compile-time.

CRTP is a design pattern where the derived class is a template parameter to the base class.

```
template<class T>
class BaseClass{
    //Call functions of derived class through T
};

class DerivedClass : public BaseClass<DerivedClass>
{
    void doSomething();
};
```

This kind of design helps us in cutting down on the costs incurred during runtime when choosing which function to call by matching parameters.

Pull request

I submitted the implementation of the range and offset level formats at `range.hpp` and `offset.hpp` and a test-case which checks if the implementation works for the DIA tensor storage format.

Pull Request Link

Timeline

Community Bonding Period - May 20, 2022 to June 12, 2022

- Communicate with mentors on the design of the code generation algorithm.
- Explore xsparse, write additional tests, fix existing bugs, improve documentation if needed.
- Read and understand the code generation algorithm thoroughly from this paper and discuss its conversion to compile time C++ with mentors.
- Set up a blog to track each week's progress.

Working period - June 13, 2022 to September 4, 2022

Week 1 - June 13, 2022 – June 19, 2022

- Implement two level formats.
- Write test-cases for the implemented levels and add additional test-cases for already existing levels.

Week 2 - June 20, 2022 – June 26, 2022

- Implement remaining level formats.

- Add test-cases for commonly used tensor storage formats like CSR, DIA et cetera for all levels.

Week 3 - June 27, 2022 – July 3, 2022

- Shift focus to level capabilities.
- Implement the insert capability, along with its four functions in `assembly.hpp`.
- Write tests for coordinate insertion and fix any implementation issues.

Week 4 - July 4, 2022 – July 10, 2022

- Implement the append capability, along with its four functions in `assembly.hpp`.
- Implement the locate capability at `locate.hpp`.
- Write test-cases with focus on coordinate edge assembly and locate.

Week 5 - July 11, 2022 – July 17, 2022

- Shift focus to level properties.
- Implement at least two properties, preferably full and unique.
- Write test-cases for the implemented properties.

Week 6 - July 18, 2022 – July 24, 2022

- Implement remaining properties.
- Write test-cases for the implemented properties.

Week 7 - July 25, 2022 – July 31, 2022

- Dedicated to evaluation of the first half of the project.

- Make any proposed design changes regarding level formats, capabilities and properties.
- Update documentation and write base classes for code generation.

Week 8 - Aug 1, 2022 – Aug 7, 2022

- Implement level iterator conversion practices, deduplication and reordering approaches.
- Write relevant test-cases for the same.

Week 9 and Week 10 - Aug 8, 2022 – Aug 21, 2022

- Dedicated to work on the code generation algorithm and implement the approaches discussed during the community bonding period using template metaprogramming and CRTP.
- Update documentation and finalize tests for release.

Week 11 - Aug 22, 2022 – Aug 28, 2022

- Buffer week. Discuss and fix design issues and utilize this time to handle unexpected delays.

Week 12 - Aug 29, 2022 – Sept 4, 2022

- Create PRs and make changes to code, based on feedback given by the mentors.
- Submit the final evaluation.

Post GSoC, I plan to be actively involved with xsparse and fix bugs and implement new features outside those mentioned in the TACO paper, such as reshaping, transposing et cetera.

I also intend to write a cppyy NumPy API wrapper to generate Python-C++ bindings.

Work Schedule

The plan is to start coding at 4:00 PM and try to complete the goal that is set for a particular day. I intend to communicate educational commitments like college examinations with mentors and plans to make up for the lost time.
At the end of every week, I plan to write a blog post talking about what I implemented that week and what I learnt while doing the same.

Acknowledgements

- I would like to thank my mentor, Hameer Abbasi, for helping me understand the codebase of xsparse and clearing my doubts regarding compile-time C++ features.

References

- This paper, authored by Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe.
- This video for understanding template metaprogramming and type traits.