# PyData/Sparse

## Completion of Python bindings for the TACO compiler

## Personal details

Name: Sayandip Halder
Institute: Jadavpur University
Country of residence: India
Email: sayandiph4@gmail.com
Github: sayandip18

## About me

I am pursuing a Bachelor of Engineering degree at Jadavpur University, Kolkata. I have been coding in Python for a couple of years now. Sublime Text 3 as my primary editor on my Ubuntu 20.04.1 system. In GSoC 2020, I had thought of applying for the organisation Uarray, but it proved to be a daunting task to me who was a beginner in the world of open-source development then. However, I have made some contributions to Sympy and LiberTEM throughout the next one year and I am confident about taking on large codebases now. I will be free for the entire time period of GSoC 2021 and I can easily afford 40 hours a week.

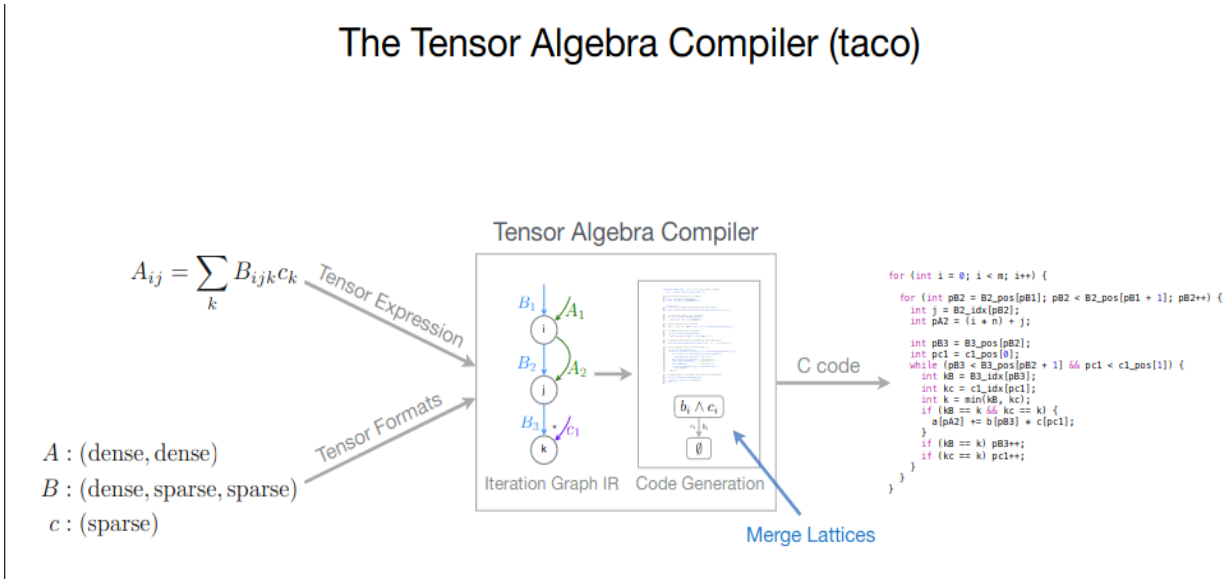
## The Project: an overview

TACO is a library for generating tensor algebra kernels. It generates efficient code to handle both sparse and dense linear algebra and tensor algebra computations.

The traditional approach to handle these cases is to implement handwritten code for kernels for every operation on sparse and dense tensors of different formats and an infinite number of possible binary operations. Traditional libraries, therefore, hand-code a few expressions, choose a few formats and then optimize on those.

However, this approach is now unsuitable due to an abrupt increase in the number of compound operations that must be developed.

TACO generates code entirely from tensor index notation and simple format descriptors that fully specify the compressed data structures.



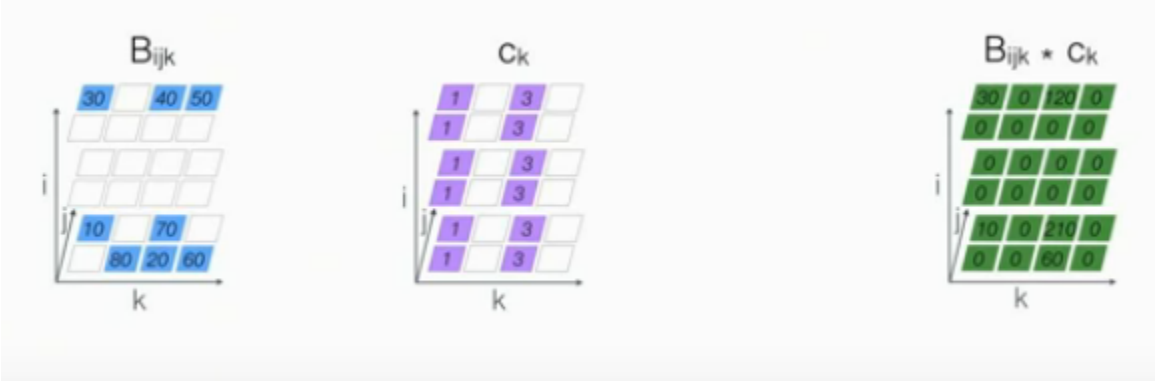

The image above describes how TACO works. The tensor expression and the format is taken as input and a C code kernel that computes the expression is received as output. Inside the compiler, the expression is represented as an iteration graph from which the code is produced. Merge lattices are used to deal with merging of operands.

Let us consider an example to understand how TACO works and explore the key concepts in more detail. Consider the tensor kernel below.

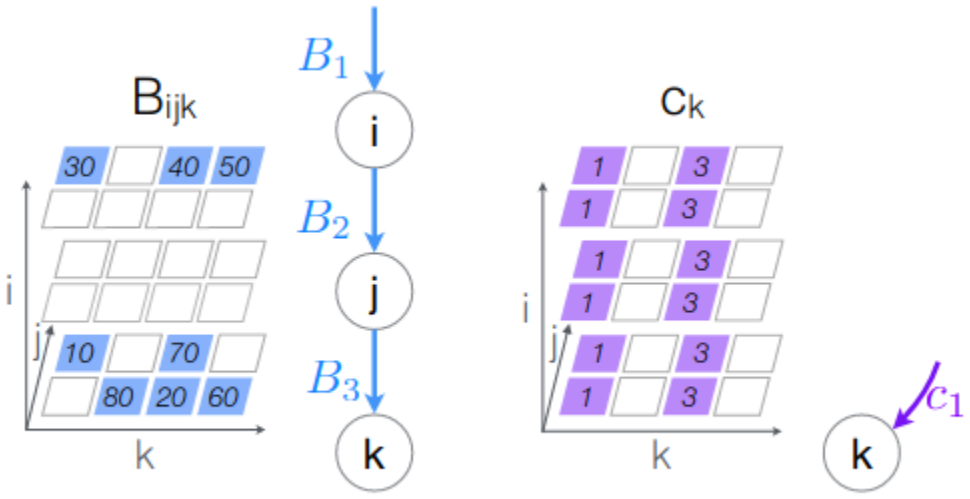
$$A_{ij} = \sum_k B_{ijk} * c_k$$

## Iteration graphs

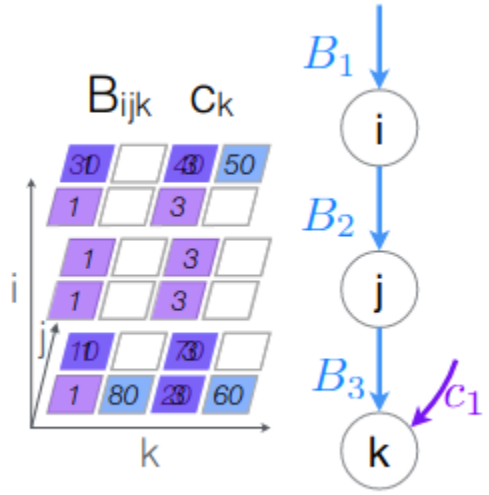
If we want to compute this expression, we will need 3 loops for i,j and k. It can be thought of as 3 dimensions in a polyhedron. Each expression can be thought of as a sparse polyhedron in the following way



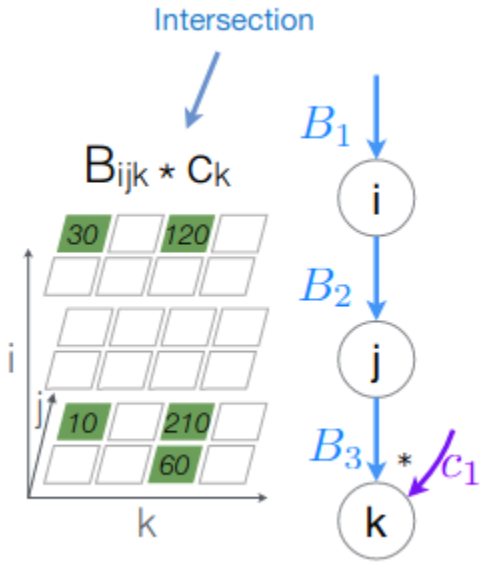

The generated code should skip over the holes, i.e. cells with the value zero. In order to generate efficient code that does that, TACO generates a dependency chain for each expression.



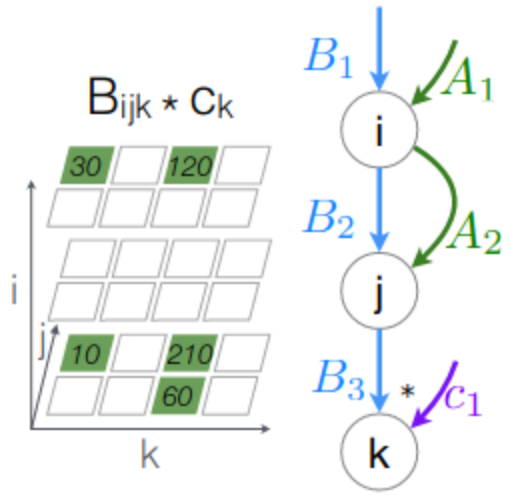

To perform the multiplication, we aim to merge the two polyhedrons and the dependency chains, resulting in a clash as seen in the figure below.




So, in order to get the product, we do an AND operation. This gives us the structure shown below.




Finally, the dependency on the result is added too.




This final iteration graph is used for code generation.

An iteration graph is therefore, a structure generated by TACO that describes how to iterate over non-null values in the tensor expression. More formally, an iteration graph is a directed graph G = (V, P) with a set of index variables V = {$v_1$, $v_2$, …, $v_n$} and a set of tensor paths P = {$p_1$, ...,$p_m$}. [Defined in https://dl.acm.org/doi/10.1145/3133901]

Some examples of iteration graphs are shown below.

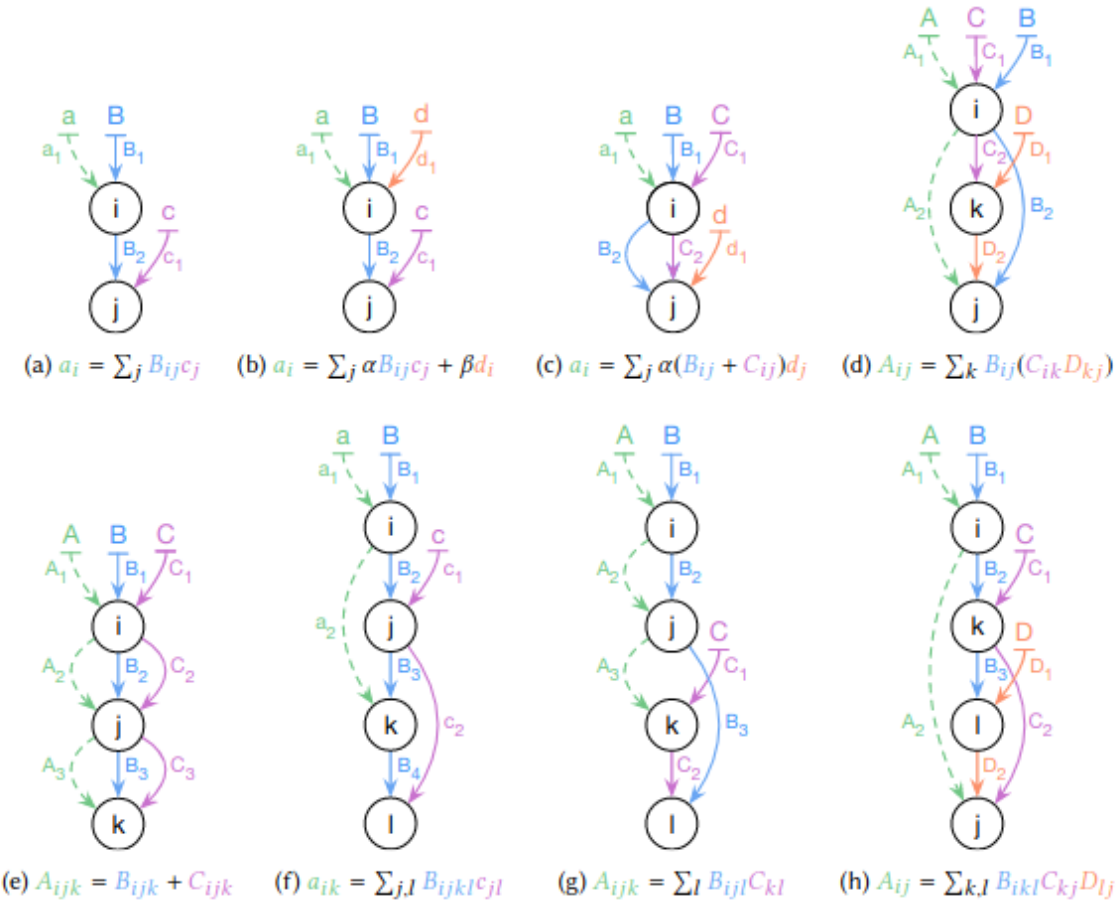
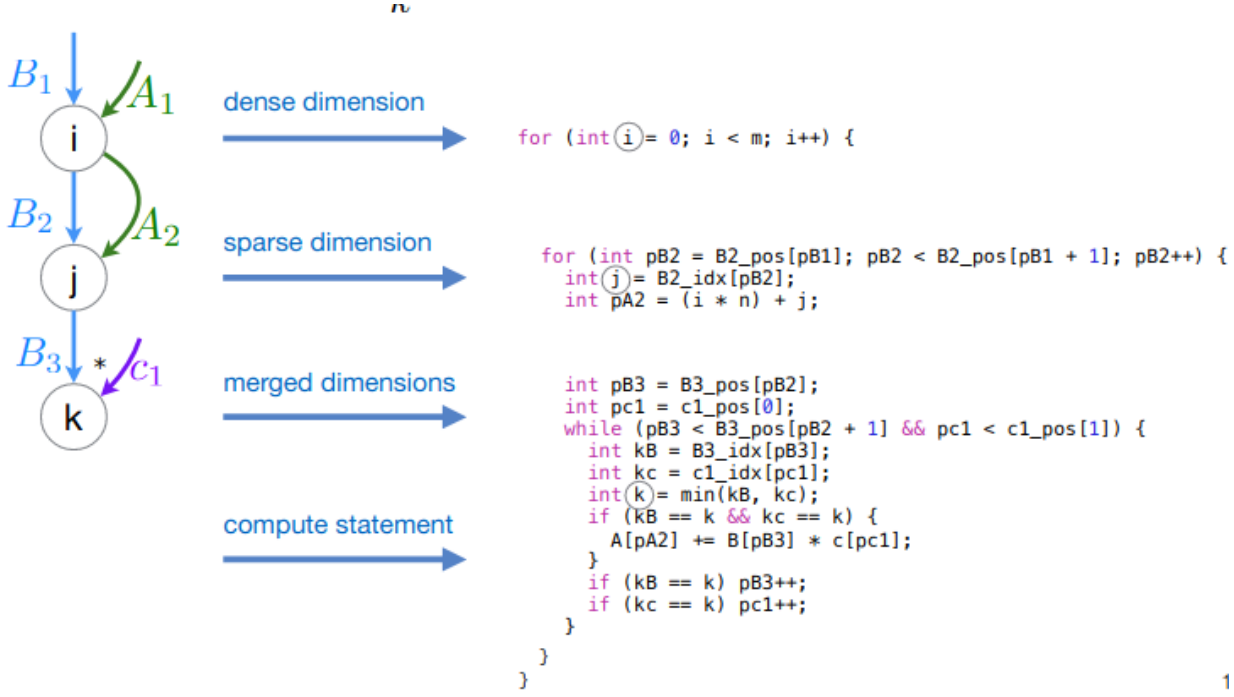



Fig. 8. Iteration graphs for (a) matrix-vector multiplication, (b) scaled matrix-vector multiplication plus a scaled vector, (c) scaled sum of two matrices times vector, (d) sampled dense-dense matrix product, (e) tensor addition, (f) *blocked* matrix-vector multiplication, (g) tensor-times-matrix multiplication (TTM), and (h) matricized tensor times Khatri-Rao product.

Code is generated from each level of the iteration graph. We can ignore all other levels and concentrate on the level of our interest to generate code for that level.



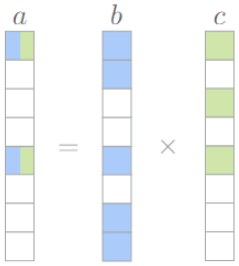

The merge operation here is what is known as conjunctive merge. Conjunctive merge is applied for multiplication operations whereas for addition, disjunctive merge is applied.

## Merge lattices

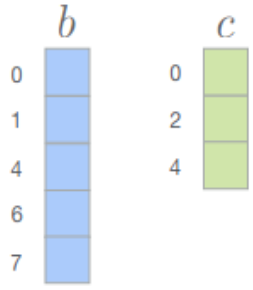
Let's discuss conjunctive and disjunctive merges in more detail.

1. **Conjunctive merge**

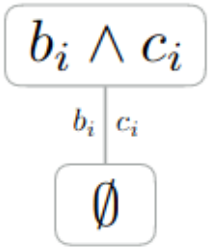
Let's take the expression $a_i = b_i c_i$ denoting an element-wise multiplication of two sparse vectors.



Since they are sparse, the non zeros are removed.




Then, we take an intersection of $b$and $c$. This operation suggests that we iterate over both $b$and $c$until one of them runs out of value. The merge lattice is represented as

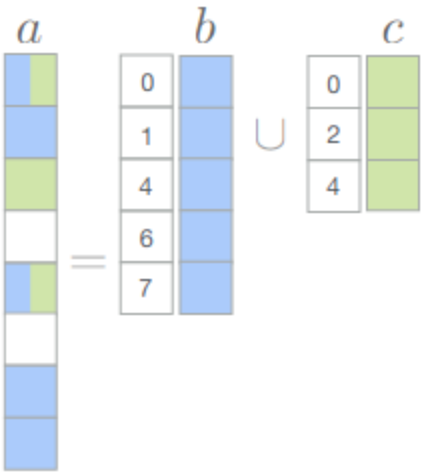



When we run out of value, we drop down to the bottom and our operation is done.

2. **Disjunctive merge**

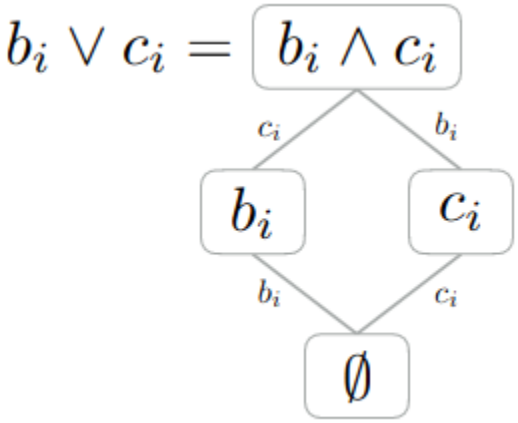
Let's take the expression $a_i = b_i + c_i$ denoting element-wise addition.

We take the union of $b$and $c$ here.




$b_i \vee c_i$ is too expensive because of the additional loops, unlike the previous operation $b_i \wedge c_i$ where we are done iterating once we run out of values in one of the vectors. So, the expression is rewritten as $b_i \wedge c_i \vee b_i \vee c_i$

We get a lattice like this

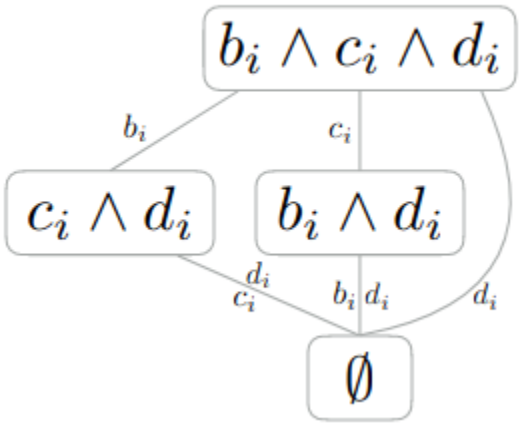



We iterate till one of the two vectors has a value left. Once one runs out of value, we run through the rest of the other one.

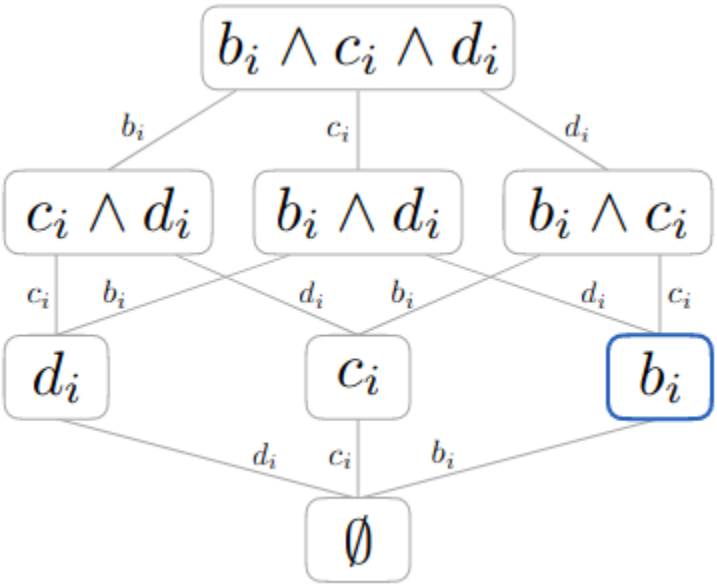
3. **Compound merge**

Now that we have seen how the two basic operations work, let's consider a few Compound expressions.

The merge lattice for $a_i = (b_i + c_i)\, d_i$ will be




For $a_i = b_i + c_i + d_i$ the lattice will be




More formally, a merge lattice L is a lattice comprising n lattice points{L1, ...,Ln}and a meet operator. Each lattice point Lp has associated with it a set of tensor dimensions Tp={tp1, ...,tpk} to be merged conjunctively (i.e.tp1∧...∧tpk)

and an expression $expr_p$ to be evaluated. The meet of two lattice points L1 and L2 with associated tensor dimensions T1 and T2 respectively is a lattice point with tensor dimensions $T1 \cup T2$. We say $L1 \leq L2$ if and only if $T1 \subseteq T2$, in other words if L2 has tensor dimensions that are exhausted in L1 but not vice versa. [Defined in https://dl.acm.org/doi/10.1145/3133901]

## PYTACO

TACO has a Python library, called Pytaco, with the help of which the user can access TACO objects with Python. Pytaco is essentially a Python wrapper around TACO using pybind11. Pybind11 is a header-only library that provides python bindings of C++ code. It exposes C++ types in Python and vice versa. Pytaco is hosted here.

The following code is an example of how pybind11 can be used after installation

```
#include <pybind11/pybind11.h>
#include<iostream>

namespace py = pybind11;

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.def("add", &add, "A function which adds two numbers");
}
```

The PYBIND11_MODULE() macro creates a function that will be called when an import statement is issued from within Python. The module name (here, “example”) is given as the first macro argument. The second argument (m) defines a variable of type py::module_ which is the main interface for creating

bindings. The method module_::def() generates binding code that exposes the add() function to Python.

The tensor() class defined in this file can be used to initialise tensors.

For example, the code below initialises a tensor of dimension 512 x 64 x 2048.

```
import pytaco as pt
from pytaco import dense, compressed

# Declare a new tensor of double-precision floats with
dimensions
# 512 x 64 x 2048, stored as a dense-sparse-sparse tensor
A = pt.tensor([512, 64, 2048], pt.format([dense, compressed,
compressed]), pt.float64)
```

Tensor formats can be initialised in the following way

```
dm   = pt.format([dense, dense])
# (Row-major) dense matrix format
csr  = pt.format([dense, compressed])
# Compressed sparse row matrix format
csc  = pt.format([dense, compressed], [1, 0])
# Compressed sparse column matrix format
dcsr = pt.format([compressed, compressed], [1, 0])
# Doubly compressed sparse column matrix format
csf  = pt.format([compressed, compressed, compressed])
# Compressed sparse fiber tensor format
```

Tensors can also be initialized from either NumPy arrays or SciPy sparse (CSR or CSC) matrices too.

```
sparse_matrix = scipy.sparse.load_npz('sparse_matrix.npz')
taco_tensor = pt.from_sp_csr(sparse_matrix)
np_array = np.load('arr.npy')
dense_tensor = pt.from_array(np_array)
```

The Python wrapper is not fully complete yet and needs to have some additional features. Pytaco needs to have a numpy ndarray compatible API.
In the code below, a dense tensor is initialised from a numpy ndarray. We can apply ufuncs to it.

```
>>> import numpy as np; import pytaco as pt
>>> t=np.array([1,2,3])
>>> type(t)
<class 'numpy.ndarray'>
>>> A = pt.from_array(t)
>>> A
A152 (3) ({dense}; 0):
dense (0):
  [3]
[1, 2, 3]
>>> np.exp(A)
array([ 2.71828183,  7.3890561 , 20.08553692])
```

However, for a sparse tensor, ufunc cannot be applied.

```
>>> A = pt.tensor([2,2,2], compressed)
>>> A
A5 (2x2x2) ({compressed},{compressed},{compressed}; 0,1,2):
compressed (0):
  [0, 0]
  []
```

```
compressed (1):
  [0]
  []
compressed (2):
  [0]
  []
[]
>>> A.insert([0, 1, 0], 42.0)
>>> A.pack()
>>> np.exp(A)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"/home/sayandip/taco/build/lib/pytaco/pytensor/taco_tensor.py",
line 384, in __array__
     raise ValueError("Cannot export a compressed tensor. Make
sure all dimensions are dense "
ValueError: Cannot export a compressed tensor. Make sure all
dimensions are dense using to_dense() before attempting this
conversion.
```

The plan is to apply ufuncs directly to pytaco tensors without densifying, since densifying messes up the memory and time benefits of sparse arrays. The ufuncs we intend to make applicable are np.sum, np.add, np.subtract, np.multiply and np.exp.

# The Plan

- Part 1: Modify `pytaco.tensor` so that universal functions can be applied to it directly.

- Part 2: Seeing which tests fail from the test suite of the tensora repository. Making sure that most tests pass. Adding more tests to pytaco. There should be a separate tests folder.

- Part 3: Implementation of advanced indexing of `pytaco.tensor` objects.

# The timeline

- Community bonding (May 17, 2021 - June 7, 2021): Interacting with team TACO and exploring the codebase more. Planning out the details of the project in a more specific way and possibly restructuring the plan if thought necessary.

**Phase 1**:

- Week 1: Implement np.add to pytaco.tensor.
- Week 2: Implement np.subtract tp pytaco.tensor.
- Week 3: Implement np.sum and np.exp
- Week 4: Implement np.multiply.

Documentation and tests will be added too.

**Phase 2:**

- Week 5 and 6 will consist of seeing which tests pass and which ones fail from the test suite of tensora repository. The test suite of pytaco should improve. Instead of a single file, there should be a directory having numerous files with each file having tests relevant to a certain feature.

**Phase 3:**

- Week 7 and 8 will see implementation one or both of the following features depending on time constraints

    1. TACO does not support computations that have tensors as operand and result, e.g. a[i] = a[i] * b[i]. A computation like this has to be implemented using temporary variables. I intend to implement this feature in pytaco.
    2. TACO does not support using the same index variable to index into multiple dimensions of the same tensor operand (e.g., A[i,i]). This should be implemented too.

- Week 9-10: Buffer weeks.

# Link to PRs merged:

- PR 448
- PR 450

# Bibliography

This talk at Microsoft Research by Fredrik Kjolstad.

This presentation, which was used in the talk.

This and this paper authored by Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato and Saman Amarasinghe.

The TACO documentation and the tutorial notebooks.