A Tensor Algebra Compiler Library Interface and Runtime

by

Patricio Noyola

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author . . . .
Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by . . . .
Saman Amarasinghe
Professor and Associate Department Head
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

A Tensor Algebra Compiler Library Interface and Runtime

by

Patricio Noyola

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

Tensor algebra is a powerful tool for computing on multidimensional data and has applications in many fields. The number of possible tensor operations is infinite, so it is impossible to manually implement all of them to operate on different tensor dimensions. The Tensor Algebra Compiler (taco) introduced a compiler approach to automatically generate kernels for any compound tensor algebra operation on any input tensor formats.

In this thesis, we present a new API for the taco library. The API removes the need to call compiler methods with the introduction of a delayed execution framework. Additionally, the API introduces multiple important tensor algebra features previously unavailable in taco. Finally, we propose extensions to taco's code generation algorithm to automatically generate tensor API methods for any tensor format.

The proposed API makes taco code cleaner, shorter and more elegant. Furthermore, the extensions to its code generation algorithm make the API scalable to new formats and operations.

Thesis Supervisor: Saman Amarasinghe
Title: Professor and Associate Department Head

Acknowledgments

I would first like to thank my research advisors Prof. Saman Amarasinghe, who is also my thesis supervisor, and Fredrik Kjolstad. They provided a lot of insightful feedback and ideas that greatly improved the design of the API described in this thesis, as well as significant help with the presentation of the thesis. Their guidance and advice during my time as a master's student at MIT was invaluable.

Additionally, I would like to thank Stephen Chou for the feedback he provided in the design of the API, as well as for his help reviewing and improving my code.

I would also like to thank the friends I made during my time at MIT, especially my partner Juju. They have been with me throughout one of the most important and exciting experiences in my life, and have always been there to support me and keep me going.

Last but not least, I would like to thank my parents, Daniel Noyola and Pilar de la Rosa, as well as my brother, Juan Pablo Noyola, for their unconditional love and support. I would not be where I am and who I am without them.

Contents

List of Figures

Chapter 1

Introduction

Tensor algebra is a powerful tool for computing on multidimensional data and has practical applications in fields ranging from data analytics and machine learning to the physical sciences and engineering [1, 2, 3, 8, 16, 12]. Tensors generalize matrices to any number of dimensions and represent multilinear relationships. Real-world applications often operate on tensors that are both very large and extremely sparse, meaning most components are zeros. For example, a tensor encoding product reviews on Amazon [17] contains 1.5×10^{19} components, but only 1.7×10^{9} of them are non-zero.

Routines to compute tensor operations (tensor kernels) can be complex for a number of reasons. Sparse input tensors are compressed into indexed data structures that the kernels must operate on. Furthermore, many operations require only non-zero output values should be produced, yet the memory layout of the resulting tensor is dependent on which input tensors contribute what non-zero values. Finally, sparse tensor data structures typically do not support constant-time random access, so kernels must carefully orchestrate co-iteration over multiple data structures to achieve good performance.

Standard algebra libraries provide a limited set of hand-optimized operations which allow programmers to compute compound operations through a sequence of supported operations using temporary tensors. This reduces locality and efficiency, and for some compound operations the temporaries are much larger than the kernel's inputs and output. Moreover, the implementation of kernels to compute compound tensor expressions can vary largely depending on the dimensionality of the tensors, the operations between tensors, and the formats of each one of the tensors. A tensor expression can involve any number of tensors of any possible order, and any combination of algebraic operations between them. Additionally, the data of each tensor can be stored in one of many possible tensor formats. All of these factors lead to a combinatorial explosion of all possible compound operations, which makes it infeasible to write optimized code for each one of them by hand.

Recently, Kjolstad et al. proposed the first compiler technique to generate kernels for all tensor algebra expressions [15]. The Tensor Algebra Compiler (taco) is a C++ library which generates efficient imperative code to compute tensor expressions entirely from tensor index notation and simple format descriptors that fully specify the desired computation and the tensor data structures. This compiler approach turns any complex compond tensor expressions into single kernel, which addresses locality issues observed in other approaches, and allows for expression-wide optimizations and reductions to achieve increased performance. Furthermore, taco presents a level-based abstraction that captures how to efficiently access different data structures that encode individual tensor dimensions. These per-dimension formats can be composed in arbitrary order to create a large variety of tensor formats, all of which are supported by taco.

With its code generating capabilities on arbitrary tensor expressions and a huge variety of supported tensor formats, taco provides an outstanding amount of functionality. Due to the compiler nature of the library however, taco's interface is hard to use, and leads to code that is hard to read. Consider, for instance, the syntax example for computing sparse matrix-vector multiplication shown in figure 1-1. The figure compares the old syntax of taco to Eigen [10], a popular linear algebra library. taco performs at-run-time compilation to generate the code used to compute user-defined tensor expressions. users must then specify when when each operation should be compiled, assembled and computed (lines 27-29), as well as when tensor values are packed into their respective data structure (lines 20-21).

```
// taco SpMV

// Create formats
Format csr({Dense,Compressed});
Format  dv({Dense});

// Create operands
Tensor<double> a({3},   dv);
Tensor<double> B({3,4}, csr);
Tensor<double> c({4},   dv);

// Insert data into B and c
B.insert({0,0}, 1.0);
B.insert({1,2}, 2.0);
B.insert({2,1}, 3.0);
c.insert({0}, 2.0);
c.insert({1}, 6.0);

// Pack inserted data
B.pack();
c.pack();

// SpMV index notation
IndexVar i, j;
a(i) = B(i,j) * c(j);

a.compile();
a.assemble();
a.compute();

// Print the output
std::cout << a << std::endl;
```

```
// Eigen SpMV

// Create operands
VectorXd a(3);
SparseMatrix<double> B(3,4)
VectorXd c(4);

// Insert data into B and c
B(0,0) = 1.0;
B(1,2) = 2.0;
B(2,1) = 3.0;
c(0)   = 2.0;
c(1)   = 6.0;

// SpMV index notation
a = B * c;

// Print the output
std::cout << a << std::endl;
```

Figure 1-1: SpMV syntax comparison between taco(left) and Eigen(right).

In addition, the taco tensor interface lacks important tensor manipulation methods available in most traditional libraries, and therefore expected by users. This motivates the need for a simple tensor interface that hides the taco compiler. In Chapter 4, we describe a new tensor interface which simplifies taco syntax and extends the set of tensor methods available; figure 1-2 shows an example that uses the new API. Then we then introduce extensions to taco which enhance the range of operations that can be expressed through taco's format abstraction and index notation.

The new tensor interface removes the need to call any of the tensor compiler methods (pack, compile, assemble, compute). This is achieved via a new lazy execution framework, in which compiler method calls occur automatically when needed. The use of compiler methods therefore becomes optional for users with the new API. The lazy execution framework's design is described in Chapter 5.

Given the number of formats supported by taco, it is unfeasible to hand-implement

```
// Create formats
Format csr({Dense,Compressed});
Format  dv({Dense});

// Create operands
Tensor<double> a({3},   dv);
Tensor<double> B({3,4}, csr);
Tensor<double> c({4},   dv);

// Insert data into B and c
B(0,0) = 1.0;
B(1,2) = 2.0;
B(2,1) = 3.0;
c(0) = 2.0;
c(1) = 6.0;

// Form a tensor-vector multiplication expression
IndexVar i, j;
a(i) = B(i,j) * c(j);

// Print the output
std::cout << a << std::endl;
```

Figure 1-2: New taco API SpMV syntax.

tensor interface methods to access each possible format. Similarly, attempting to implement methods that address all formats simultaneously leads to complicated and slow code that does not scale with the addition of new formats (such as the tensor iterating routines implemented for dense and compressed mode formats in taco). In Chapter 6 we describe how to leverage and extend taco's code generation capability to generate code specific to each supported tensor format that executes different tensor methods. We introduce a set of new operations to be supported by taco's code generation algorithm, along with details about how these operations can be used to generate code to execute additional interface methods. The algorithm works by emitting code that efficiently iterates and merges coordinate hierarchy representations of the tensor operands based on high-level characteristics of each format's access patterns. A simple inlining pass then specializes the emitted code to work with specific tensor formats and target runtime environments. The proposed operations introduce modifications to existing inlining specializations, without changing any of the algorithms logic.

To summarize, we make the following contributions:

Tensor API We describe a new tensor interface which encompasses an extended set of features, and traditional library syntax (Chapter 4), along with extensions to the format and index notation abstractions which allow expressing new tensor operations.

Lazy Execution Framework We introduce a lazy execution framework (Chapter 5), which automatically executes compiler methods as needed.

Code Generation We present new operations to be supported by the taco code generation algorithm, and describe how code generation can be used to generate format specific implementations of tensor methods (Chapter 6).

We implement our tensor interface and the lazy execution framework as part of the open-source tensor algebra compiler taco. The focus on tensor algebra syntax of our interface makes taco code simpler and easier to read, and provides increased code portability between taco and other tensor and linear algebra libraries. We discuss different design decisions of our interface in Chapter 8 providing insight regarding design goals and language restrictions which influenced our design.

Chapter 2

Motivating Example

Tensor methods are an core element of taco's syntax and functionality. While tensor reads and writes, and operation declaration dictate what should happen during program execution, compiler methods determine how and when. In contrast, traditional libraries do not perform at-run-time compilation, so compiler methods are not required; all tensor operations occur at the moment the operation is declared.

Compiler methods provide users with fine-grained control over when each step of the tensor computation occurs, which can be leveraged to avoid redundant computations and to measure performance. However, the constant necessity of compiler methods also leads to increased code size and syntax complexity. As an example, consider taco syntax of a simple routine for exponentiating a square matrix A to the kth power by creating a Result matrix which aggregates the result of multiplying A k times. To perform this routine, a temporary tensor is required to store an intermediate result, since taco currently does not allow any given tensor to appear on both sides of a tensor assignment expression (lines 15-16,21-22.)

Figure 2-1 exemplifies the simplicity and clarity introduced by the new API. Moreover, correct implementation of many routines require the in-code location where an operation in defined and where it is executed to be distinct, which leads to pieces of code that are conceptually unintuitive and hard to read. For example, the exponentiation routine declares two tensor expressions once in its main body (lines 15-19), which are later executed multiple times within the loop with calls to assemble

```
1  // Return a tensor representing A**k
2  Tensor<double> exponentiate(
3      Tensor<double>& A, double k) {
4    Tensor<double> R = Tensor<double>(
5        A.getDimensions(), A.getFormat());
6    Tensor<double> Temp = Tensor<double>(
7        A.getDimensions(), A.getFormat());
8
9    IndexVar i,j,k;
10   R(i,j) = A(i,j);
11   R.compile();
12   R.assemble();
13   R.compute();
14
15   Temp(i,j) = R(i,k) * A(k,j);
16   Temp.compile();
17
18   R(i,j) = Temp(i,j);
19   R.compile();
20   for (int i = k; i > 1; i--) {
21     Temp.assemble();
22     Temp.compute();
23     R.assemble();
24     R.compute();
25   }
26   return R;
27 }
```

```
1  // Return a tensor representing A**k
2  Tensor<double> exponentiate(
3      Tensor<double>& A, double k) {
4    Tensor<double> R = Tensor<double>(
5        A.getDimensions(), A.getFormat());
6    Tensor<double> Temp = Tensor<double>(
7        A.getDimensions(), A.getFormat());
8
9    IndexVar i,j,k;
10   R(i,j) = A(i,j);
11
12
13
14
15
16
17
18
19
20   for (int i = k; i > 1; i--) {
21     Temp(i,j) = R(i,k) * A(k,j);
22
23     R(i,j) = Temp(i,j);
24
25   }
26   return R;
27 }
```

Figure 2-1: Exponentiation routine syntax comparison between the current(left) and new(right) APIs.

and compute (lines 21-24.) By making the use of compiler methods optional, the new API simplifies taco syntax, and completely removes the conceptual separation between expression definition and execution that compiler methods introduce.

The example presented shows the advantages of the new tensor interface for taco in terms of syntax size and readability. In Chapter 3 we describe the new API, highlighting the differences between it and the old API.

Chapter 3

Tensor Algebra Compiler Background

The Tensor Algebra Compiler[15, 14] (taco) is a C++ library that allows users to define and execute code to compute any tensor algebra operations. The library accomplishes this with a compiler that generates code specific to each tensor operation requested by the user, depending on the type of operation and the formats of the input and output tensors. Taco takes in user-defined tensor index notation, and performs online compilation to provide kernels to compute the specified operation.

Traditional tensor and linear algebra libraries such as the Eigen linear algebra library [10], or the MATLAB Tensor Toolbox [4] support a fixed set of hand implemented operations for specific input and output operand formats. This approach allows developers to focus on developing a small set of highly optimized operations which can be composed to compute more complex expressions. While the set of hand optimized operations can indeed achieve outstanding performance results, performance suffers when computing compound expressions due to reduced locality and the need for intermediate temporary tensors. Yet there exists an infinite number of possible tensor algebra expressions, so it is impossible to hand implement and optimize all of them. The compiler approach introduced by taco addresses this problem by automatically generating the code to compute compound expressions.

The taco library provides a set of C++ abstractions to represent different tensor algebra concepts, such as tensors, tensor formats, tensor operations, etc. This section introduces some of the most relevant abstractions, which will be referenced throughout the rest of this thesis.

Figure 3-1 demonstrates how to use taco to compute a tensor-times-vector multiplication. `Tensor` objects can be created by specifying the dimensions of the tensor, the type of its entries, and its storage format. The storage format of a tensor is in turn declared by creating a `Format` object which specifies the data structure which will hold the tensor values, describing the storage kind of each tensor level and the order in which levels are stored. On lines 1-8, `A` is declared as a CSR matrix, `B` as a CSF tensor, and `c` as a compressed vector. Tensors that are inputs to computations can be initialized with user-defined data by specifying the coordinates and values of all non-zero components with the INSERT method, and then invoking the PACK method to store the tensor in the desired storage format, as seen in lines 10-17.

A variant of the tensor index notation developed by Ricci-Curbastro and Levi-Civita [18], from now on referred as Index notation, is used to describe tensor computation as a scalar expression where tensors are indexed by index variables (`IndexVar`) in lines lines 19-20. Index variables range over the tensor dimensions they index and determine how the different components in the operands are combined during ten-

```
1  Format csr({Dense,Compressed});
2  Tensor<double> A({16,32}, csr);
3
4  Format csf({Compressed,Compressed,Compressed});
5  Tensor<double> B({16,32,100}, csf);
6
7  Format sv({Compressed});
8  Tensor<double> c({100}, sv);
9
10 B.insert({0,0,0}, 1.0);
11 B.insert({1,2,0}, 2.0);
12 B.insert({1,2,1}, 3.0);
13 B.pack();
14
15 c.insert({0}, 3.0);
16 c.insert({1}, 6.0);
17 c.pack();
18
19 IndexVar i, j, k;
20 A(i,j) = B(i,j,k) * c(k);
21
22 A.compile();
23 A.assemble();
24 A.compute();
```

Figure 3-1: Old taco API tensor-times-vector multiplication syntax.

sor operation. Operator overloading is used to make tensor operations mirror their mathematical definition. Summation reductions are implied over index variables that only appear on the right-hand side of an expression.

Calling `compile` on the target of a tensor algebra computation (`A` in the example) triggers the generation of kernels to assemble and compute the operation declared by the index notation. Based on user instructions, taco generates either `C` or `CUDA` code, and then calls the system compiler to compile it to a dynamic library, which is dynamically linked via `dlopen`. This makes the `assemble` and `compute` functions available for later calls.

The `assemble` method assembles the sparse index structure of the output tensor and allocates memory for it. Finally, the actual computation is performed via the `compute` method, which execute the code generated by `compile`. By separating the output assembly from the actual computation, taco enables users to assemble output tensors once and then recompute the defined operation multiple times.

There exist many formats for storing sparse tensors, each of which is ideal under specific circumstances depending on the structure and sparsity of the data, the computation, and the hardware. Computing operations with different existing formats however requires specialized code for every combination of operand tensor formats, which makes it challenging to support many formats simultaneously.

In order to support different tensor formats, taco defines six per-dimension formats that can be composed in a hierarchical fashion. By composing these per-dimension formats, it is possible to recreate variants of many tensor formats, as well as many generalizations of these formats for higher order tensors [15, 7].

The idea of per-dimension formats can best be understood by viewing tensor storage as a composition of levels that each encode coordinates along one tensor dimension. The different levels implicitly follow the order imposed by the tensor coordinates, generating a coordinate hierarchy. Entries at each level of the hierarchy then refer to elements in the following level, but the levels are otherwise independent. The tensor values are stored as the bottom level of the hierarchy.

Each per-dimension format supports a set of capabilities, or functions that the format is capable of performing. In addition, each format possesses a set of properties which reflect invariants that are explicitly enforced or implicitly assumed by the underlying physical format index. The code generation algorithm implemented by taco uses the properties and capabilities of each coordinate hierarchy level to determine the best way to simultaneously iterate over the tensors involved in each tensor expression at a high level, as well as how to access and store values in each of them. Taco is then capable of generating code to operate on any format constructed by composing taco's per-dimension formats.

Chapter 4

The Lazy Execution API

APIs play an important role in developing useful software tools. Clear, simple interfaces promote usability and lower the barrier of entry for new users to adopt the tool. Furthermore, users expect consistency throughout tools that address similar problems, which in turn promotes code portability.

The new taco API extends the set of operations available in taco and simplifies its syntax. It is now closer to traditional tensor and linear algebra interfaces. Figure 4-1 presents the same tensor-times-vector multiplication shown previously in Figure 3-1, but implemented in the new interface.

```
Format csr({Dense,Compressed});
Tensor<double> A({16,32}, csr);

Format csf({Compressed,Compressed,Compressed});
Tensor<double> B({16,32,100}, csf);

Format sv({Compressed});
Tensor<double> c({100}, sv);

B(0,0,0) = 1.0;
B(1,2,0) = 2.0;
B(1,2,1) = 3.0;

c(0) = 3.0;
c(1) = 6.0;

IndexVar i, j, k;
A(i,j) = B(i,j,k) * c(k);
```

Figure 4-1: New taco API sparse tensor-times-vector multiplication syntax.

As shown in the figure in lines 10-15, the new API introduces tensor component access via operator overloading, indexing tensors with an integer coordinate location instead of the index variable indexing used for index notation. Additionally, syntax is more compact. Namely, calls to the compiler methods `pack`, `compile`, `assemble` and `compute` are no longer required.

4.1 The Tensor Class

The `Tensor` class is the main interface between users and taco. It allows users to reason about tensors and tensor operations at a high level without worrying about the different arrays and data structures storing tensor data, or the specifics of how tensor operations are implemented. `Tensor` objects let users interact with tensor data to read, write and print values stored. Tensor operations are also defined through `Tensor` objects via index notation.

The `Tensor` class does work behind the scenes to provide a simple API to the user. It creates the data structures used to store tensor data, handles memory allocation, and packs inserted values into the right tensor format for the user. When a tensor operation is declared by the user the `Tensor` class automatically performs code compilation for the operation, and ensures that all tensor computations are executed in the appropriate order before the user accesses the resulting data.

In taco there are two C++ classes that represent tensors: `Tensor` and `TensorBase`. Both classes provide largely the same functionality, and the main difference between them is that `TensorBase` is a regular C++ class and `Tensor` is a class template. `Tensor` takes in as template arguments the order of the tensor and the data type of the elements stored in the tensor. This prevents dynamic memory allocation, and improves the performance of the API overall. On the other hand, `TensorBase` takes in its tensor order and data type through its constructor. (Since taco performs online compilation, tensor order and data type are not strictly necessary at compile time.) The `TensorBase` class can therefore be used to write fully functional taco code with no prior knowledge about the input tensors.

The new interface classifies methods in the `Tensor` class into four major categories: constructor, metadata, read/write, and compiler methods. The distinction between read, write, and compiler methods comes into play in the lazy execution framework, and will be elaborated in Chapter 5.

4.1.1 Constructor Methods

Constructor methods are used to instantiate new `Tensor` objects. This category consists mainly of `Tensor` class constructors, but other routines to create new tensors (such as the `transposition` operation, and some file IO functions.)

The `Tensor` class has many constructors, which allow varying degrees of metadata specification. Namely choosing the tensor's *name*, the *data type* of the stored values, tensor *dimensions*, and the storage data structure to be used via the tensor's *format*.

4.1.2 Metadata Methods

Metadata methods refer to any methods that are used to extract tensor metadata, such as tensor *name*, *order*, *dimensions*, *format*, or *component type*.

Additionally, metadata methods let users set and get the tensor `TensorStorage` object, which stores the tensor values in whichever format was specified for the tensor during creation. Direct interaction with the storage object can break the tensor abstraction imposed by the tensor interface, but it is necessary to allow users to efficiently perform a number of operations not currently supported by taco.

4.1.3 Read/Write Methods

This category encompasses all methods which allow the user to access and modify values stored in the tensor without making any modifications to it. This includes reading and writing individual values stored in a tensor via the `at` and `insert` methods, iterating over tensor contents using the standard C++ `begin` and `end` methods, different routines and tensor copying.

The new API introduces the `at` method, which allows users to access values stored at specific tensor coordinates. This functionality is unavailable in the old taco interface, which requires users to iterate over all tensor values to find the component of interest. Similarly, the `insert` method inserts a tensor value to any location specified with tensor coordinates. Figure 4-2 exemplifies the use of `insert` and `at` (left).

```
Format csr({Dense,Compressed});
Tensor<double> A({16,32}, csr);

Format csf(
    {Compressed,Compressed,Compressed});
Tensor<double> B({16,32,100}, csf);

Format sv({Compressed});
Tensor<double> c({100}, sv);

// Insert Values using insert method
B.insert({0,0,0}, 1.0};
B.insert({1,2,0}, 2.0};
B.insert({1,2,1}, 3.0};

c.insert({0}, 3.0);
c.insert({0}, 6.0);

IndexVar i, j, k;
A(i,j) = B(i,j,k) * c(k);

// Read Values using at method
double x = A.at({0,0});
double y = x * A.at({12,10});
```

```
Format csr({Dense,Compressed});
Tensor<double> A({16,32}, csr);

Format csf(
    {Compressed,Compressed,Compressed});
Tensor<double> B({16,32,100}, csf);

Format sv({Compressed});
Tensor<double> c({100}, sv);

// Insert Values using ScalarAccess
B(0,0,0) = 1.0;
B(1,2,0) = 2.0;
B(1,2,1) = 3.0;

c(0) = 3.0;
c(0) = 6.0;

IndexVar i, j, k;
A(i,j) = B(i,j,k) * c(k);

// Read Values using ScalarAccess
double x = A(0,0);
double y = x * A(12,10);
```

Figure 4-2: Comparison of using insert and at methods (left) and access methods (right) for interacting with tensor values.

The API also provides a new method for inserting multiple components at a time called `setFromComponents`. This method provides similar functionality to the `setFromTriplets` method available for sparse matrices in Eigen [10]. The method takes in any collection with `begin` and `end` methods that store tensor `Component`s, and bulk inserts them to the tensor in a single action. The user can optionally specify a duplicate policy, which determines how insertion of multiple components to the same coordinate is be resolved.

Access methods are a special kind of read/write methods that return Access objects. Access objects wrap their respective tensor object, and enable additional tensor functionality, such as read/write access to tensor values, and writing tensor index notation. Access methods do not interact with the tensor directly (performing reads or

```
vector<Component<double>> componentList;
componentList.push_back({0,0,0}, 1.0);
componentList.push_back({1,2,0}, 2.0);
componentList.push_back({1,2,1}, 3.0);

Format csf(
    {Compressed,Compressed,Compressed});
Tensor<double> B({16,32,100}, csf);

B.setFromComponents(componentList);

...
```

```
template<int Order, typename CType>
class Component {
public:
  Component();
  Component(Coordinate<Order> coordinate,
            CType value);

  int coordinate(int mode) const;
  Coordinate<Order> coordinate() const;

  CType& value() const;

private:
  Coordinate<Order> coord;
  CType val;
};
```

Figure 4-3: New taco API component access functionality (left) and the `Component` interface (right).

writes for example), but rather enable more elegant syntax for the users to interact with tensors. The main purpose of access methods and objects is to enable syntax that resembles mathematical tensor indexing.

Access objects are created with the overloaded `operator()` method. Each access object stores a reference to the tensor which created it. It also contains metadata that is passed in as argument to the `operator()` call which determines how the access interacts with the tensor.

> `Access` objects are created by calling `operator()` with index variables as input, and are used to write tensor index notation. The stored index variables indicate how the tensor operation should iterate over the tensor dimensions. Index expression is the main mean for users to express tensor operations and to assign results of an operation to a tensor.

> `ScalarAccess` objects are created by calling `operator()` with integer coordinates as input. They can perform reads and writes to that specific coordinate of the tensor, as an alternative to using the `insert` and `at` methods.

Access objects are not intended as an object which users consciously create and assign to a variable, but rather as temporary containers to make tensor indexing syntax possible. As such, their functionality largely relies on method overloading, so the user should never require to explicitly call any methods on access objects. The right side of figure 4-2 shows how `ScalarAccess` objects can be used to interact with tensor values in lines 12-17 and 23-24. Line 20 provides an example of index notation constructed using `Access` objects.

4.1.4 Compiler Methods

Compiler methods play a central role in the taco library, as they create code to execute operations previously stated by the user, such as value insertions, and most notably tensor expression computation.

Pack consolidates all values inserted to a tensor into the specified storage data structure. Operands must be packed before they can be used for operation execution.

Compile Executes taco's code generation algorithm to generate code to assemble the result of, and compute any tensor expression declared via index notation.

Assemble assembles the index structure of the output tensor and allocates memory for it.

Compute Executes the code to compute the declared tensor expression, which was previously generated during the compilation step.

The proposed API does not require users to explicitly call compiler methods, and instead makes them optional. Compiler methods can, however, be called to assert control over the flow of program computations, for example to time the compilation, computation, or both. If they are not explicitly called by the user, they are instead called behind the scenes by the lazy execution framework described in chapter 5.

4.2 Block Tensor Formats

Tensor formats describe the data structure used to store the components of a tensor. In taco the Format class serves as an abstraction to conceptually separate the layout used to store tensor data from the operations performed on the tensor.

The result of any tensor expression is fully determined by its index notation and the values stored in the input tensors. The formats of the input and output tensors on the other hand change the specifics about how the generated code accesses each tensor. Using different tensor formats will therefore have a big role in the performance of the operation, but does not change high-level behavior nor correctness of the generated code.

Block matrices and tensors can be expressed in taco with the use of higher order tensors. Consider the block compressed sparse row format (BCSR), which can be represented as a 4th-order tensor [15] (a blocked matrix with two inner dense dimensions storing small dense blocks of non-zeros,) as shown in line 5 of figure 4-4. The new API makes this approach for defining blocked tensors explicit by introducing *block tensor formats*. Blocks allow formats to represent each tensor dimension as multiple tensor modes, creating tensors with multiple block levels (each mode of a dimension belongs to a different block level.) For each dimension, all but one of the modes must have a fixed size, where the size of the last mode is determined during tensor creation by the overall size of the dimension. Blocked formats support block tensors of any number of dimensions and any number of block levels.

Block formats let users reason about tensors as if they had lower dimensionality. For instance, with block formats users can write taco code to perform linear algebra and execute it with block matrices as input to their operations. Furthermore, block formats let users change between tensor and block tensor formats without having to add indices to index expressions throughout their code, which would be otherwise necessary to operate on tensors with higher dimensionality.

For example, a BCSR matrix is composed of two block levels: an outer dense sparse layer of free size modes, and an dense fixed size inner block. A BCSR format defined in taco as a block format behaves for all intents and purposes as a 2-dimensional tensor, even when its underlying data structure is that of a 4-tensor. In figure 4-4 we compare the definition of a 4th-order tensor format representing BCSR (line 5) to a blocked BCSR matrix format with blocks of fixed size 16×16 (line 8).

```
 1  // 2-tensor block vector format
 2  Format bsv({Dense,Compressed});
 3  Tensor<double> a({8,4}, bsv);
 4  Tensor<double> c({8,4}, bsv);
 5
 6  // 4-tensor BCSR format
 7  Format bcsr(
 8      {Dense,Compressed,
 9       Dense,Dense});
10  Tensor<double> B({8,8,4,4}, bcsr);
11
12  // Insert Values using 4 indices
13  B(0,0,0,0) = 1.0;
14  B(1,2,0,1) = 2.0;
15  B(1,2,1,3) = 3.0;
16
17  c(0,2) = 3.0;
18  c(2,3) = 6.0;
19
20  // Define operation using 4 indexVars
21  IndexVar i1, i2, j1, j2;
22  A(i1,i2) = B(i1,j1,i2,j2) * c(j1,j2);
```

```
 1  // block vector format
 2  Format bsv({{Dense},{Compressed(4)}});
 3  Tensor<double> a({32}, bsv);
 4  Tensor<double> c({32}, bsv);
 5
 6  // block BCSR format with fixed size blocks
 7  Format bcsr(
 8      {{Dense,Compressed},
 9       {Dense(4),Dense(4)}});
10  Tensor<double> B({32,32}, bcsr);
11
12  // Insert Values using 4 indices
13  B(0,0)  = 1.0;
14  B(8,17) = 2.0;
15  B(9,19) = 3.0;
16
17  c(2)  = 3.0;
18  c(19) = 6.0;
19
20  // Define operation using 4 indexVars
21  IndexVar i, j;
22  A(i) = B(i, j) * c(j);
```

Figure 4-4: Comparison of using a 4-tensor format (left) and a block format (right) to represent BCSR. Both yield identical functionality.

4.3 Tensor Index Notation

The new API extends taco's index notation language to support tensor transpose operations and tensor slices.

4.3.1 Tensor Transpose Notation

The code generated by taco requires tensor expressions to iterate over tensor levels in the same order in which the tensor format presents them. Therefore, taco cannot currently generate code to perform tensor transposition. It does however provide a tensor transposition routine. The new API leverages the transposition routine to enable transpose notation at the index notation level.

```
Format csr({Dense,Compressed});
Tensor<double> A({10,16}, csr);
Tensor<double> B({16,10}, csr);
Tensor<double> C({10,16}, csr);

B(0,0)  = 7.0;
B(12,4) = 3.0;
B(2,9)  = 13.0;
C(4,12) = 5.0;

// transpose index notation
IndexVar i, j;
A(i,j) = B(j,i)

A(i,j) = B(j,i) + C(i,j)
```

Figure 4-5: Tensor transposition index notation.

Tensor transposition is performed as a preprocessing step to tensor code generation. Before compiling tensor expression, the result `Tensor` object checks if the indices of all the tensors involved in the operation have the proper ordering to perform the operation. If this is not the case, the transposes of all tensors which have an incorrect ordering are computed and stored in temporary tensor objects. These temporary tensor objects are used for code generation and computation instead of the original ones, and discarded once the operation is complete.

4.3.2 Tensor Slice Notation

The new API also introduces tensor slicing notation as part of taco's index notation. Tensor slices restrict the iteration range of a tensor during a computation, so they can be used to reduce the range of values accessed for a given dimension, or reduce the dimensionality of a tensor altogether. Slicing performs two transformations to dimension iteration. First, it restricts the tensor data being accessed during the iteration by effectively dropping all data outside the slice range from the computation. Second, it shifts the index space of the tensors in the operation to align new iteration ranges.

For example, tensor slices allow the user to perform vector addition of a size 16 vector with the first 16 coordinates of a size 100 vector (figure 4-6). As another example, tensor slices allow us to slice out a matrix from within a 3-tensor, and store it as a new tensor (figure 4-7).

```
a(i) = b(i) + c(i(0,16));
```

Figure 4-6: Vector addition slice index notation.

```
A(i,j) = B(k(5), i, j);
```

Figure 4-7: Matrix slice of 3-tensor slice index notation.

Tensor slice notation extends index notation by letting the user specify a restricted iteration range to each dimension of each tensor. For example, in figure 4-6, we restrict dimension `i` of `c` to range from indices 0 to 16. In figure 4-7 on the other hand, we restrict dimension `k` of `B` to the single index 5.

Tensor slices are declared per dimension using *range index variables*, a variation of index variables which contains metadata specifying the index iteration range. When indexing a tensor with index variables, the user can use the variable's `operator()` to specify the *beginning* and *end* bounds of the desired range. Users can also specify a range *stride*, so for example, it is possible to iterate over every other value in a given dimension range. Finally, it is possible to specify a range *shift*, which lets users take the values in a index range, and offset their location within the tensor. Iteration ranges for each tensor can be defined independently.

If a *shift* is not specified, the corresponding tensor dimension size is reduced to match the size of the range. In this case, range index variables for the same dimension in an expression are required to have the same iteration size. This allows their iteration indices to be aligned one to one, pairing the first index of each iteration range, then the second, and the third, and so forth. Therefore, pairing iteration ranges to each other is never ambiguous. On the other hand, when a *shift* is defined, then the size of the corresponding dimension is not modified. Instead, the operation drops all tensor values outside of the specified range, but the tensor shape does not change. The shift modifies the location of the index range within the tensor dimension. In figure 4-8, we can see an example of how shift can be used to selectively remove values in a tensor.

```
Tensor<double> a({6}, {Compressed});
Tensor<double> b({6}, {Dense});

// b = [1, 7, 4, 3, 1, 8]
b.setFromComponents(b_vals);

a(i) = b(i(\*start*\ 2, \*end*\ 5, \*stride*\ 1, \*shift*\ -1));

// a = [0, 4, 3, 1, 0, 0]
a.compute();

// Shift of 0 is also allowed to perform value masking.
a(i) = b(i(\*start*\ 2, \*end*\ 5, \*stride*\ 1, \*shift*\ 0));

// a = [0, 0, 4, 3, 1, 0]
a.compute();
```

Figure 4-8: Vector masking using a tensor slice shift.

It is important to notice that slice notation is independent of tensor formats. The specified ranges refer to the theoretical values stored within the specified ranges of each tensor, regardless of how many values are actually stored within the ranges.

Figure 4-9 shows more examples of slice index notation. As illustrated in the figure, the size of the range assigned to each index variable is constant throughout each tensor operation. For example, in the expression from line 14, dimension `i` in matrix `A` only has 6 elements, and the corresponding dimension in `B` is restricted to a value interval of size 6. Similarly, dimension `i` in matrix `A` only has 5 elements, and the corresponding dimension in `B` is also restricted to a size 5 range, but this time the range starts at index 3 (so index 0 of `A` corresponds to 3 of `B`, 1 to 4, and so forth.) In line 16 we can see examples of ranges with stride 2 and 3, so in the case of the dimension with index variable `i`, index 0 of `A` corresponds to 0 of `B`, and then index 1 `A` corresponds to 2 of `B` instead of 1, and so forth.

The taco code generation machinery currently does not support the required functionality to implement index range iteration. Therefore, tensor operations expressed with tensor slice notation cannot yet be computed. In Chapter 6 we introduce a code generation extension to taco that will enable this behavior.

```
Format csr({Dense,Compressed});
Format dv ({Dense,Compressed});
Tensor<double> A({6,5},   csr);
Tensor<double> B({15,10}, csr);
Tensor<double> C({16,16}, csr);

B(0,0)  = 7.0;
B(12,4) = 3.0;
B(2,9)  = 13.0;
c(12,4) = 5.0;

// slice index notation
IndexVar i, j, k;
A(i,j) = B(i(0,6),j(3,8))

A(i,j) = B(i(0,12,2),k) * C(k(0,10),j(1,16,3))
```

Figure 4-9: Tensor slice index notation examples.

4.4 Linear Algebra API

In the new API we also introduce a taco linear algebra API that enables linear algebra notation in taco. The API allows users to write tensor expressions involving matrices and vectors using linear algebra conventions instead of tensor index notation. For example, instead of writing matrix-vector multiplication in index notation as `a(i) = B(i,j) * c(j)` users can write `a = B * c`.

The linear algebra API defines a `Matrix` class and the `Vector` class, both of which inherit from the `Tensor` class with their order set to 2 and 1 respectively. The `Matrix` and `Vector` classes overload their algebraic operators (addition, subtraction, multiplication, etc.) to enable linear algebra notation. The operations supported by the API are however limited to the subset of linear algebra supported by taco, namely any operations between matrices and vectors. Other linear algebra computations such as eigenvalues, linear solvers, determinants, and matrix factorizations are not supported. Figure 4-10 shows some examples of linear algebra expressions, and their equivalent tensor algebra expressions

```
// Instantiate CSR matrices
Format csr({Dense,Compressed});
Tensor<double> A({10,10}, csr);
Tensor<double> B({10,10}, csr);
Tensor<double> C({10,10}, csr);

// Instantiate dense vectors
Format dv({Dense});
Tensor<double> a({10}, dv);
Tensor<double> b({10}, dv);
Tensor<double> c({10}, dv);

IndexVar i, j, k;

// Vector subtraction
a(i) = b(i) - c(i);

// Element-wise vector product
a(i) = b(i) * c(i);

// Vector dot product
double x = b(i) * c(i);

// Matrix addition
A(i,j) = B(i,j) + C(i,j);

// Matrix multiplication
A(i,j) = B(i,k) * C(k,j);

// Element-wise matrix multiplication
A(i,j) = B(i,j) * C(i,j);

// Matrix vector multiplication
a(i) = B(i,j) * c(j);
```

```
// Instantiate CSR matrices
Format csr({Dense,Compressed});
Matrix<double> A({10,10}, csr);
Matrix<double> B({10,10}, csr);
Matrix<double> C({10,10}, csr);

// Instantiate dense vectors
Format dv({Dense});
Vector<double> a({10}, dv);
Vector<double> b({10}, dv);
Vector<double> c({10}, dv);

IndexVar i, j, k;

// Vector subtraction
a = b - c;

// element-wise vector product
a = b % c;

// vector dot product
double x = b * c;

// Matrix addition
A = B + C;

// Matrix multiplication
A = B * C;

// Element-wise matrix multiplication
A = B % C;

// Matrix vector multiplication
a = B * c;
```

Figure 4-10: Example linear algebra API operations (right) and their equivalent index expressions (right).

4.5 Compatibility with Eigen

Taco supports a large variety of tensor algebra operations between tensors of any order, and a large number of tensor formats. However, taco does not support other kinds of tensor and linear operations, including linear and eigenvalue solvers, geometric transformations, and LU and Cholesky decompositions, to name a few. Nevertheless, there exist many other tensor and linear algebra libraries that do implement the functionality that taco lacks. It is therefore important to let users make use of the combined functionality provided by the different libraries available.

The new API has a module that allows compability with Eigen [10], a popular C++ linear algebra library that supports a large variety of performant linear algebra operations. This Eigen module lets users convert back and forth between taco tensor objects and Eigen matrix and vector objects for all formats supported by Eigen. With this package users can perform major tensor algebra operations in taco, while delegating other linear algebra routines to Eigen.

Chapter 5

Lazy Execution Framework

In the new API it is not necessary for users to call tensor compiler methods. Instead, compiler methods are automatically called when required. These method calls are hidden behind the tensor abstraction, lazily executed by the tensor lazy execution framework described in this section. Users can, however, still explicitly call these methods to force a compilation or execution step to happen, but are not required to do so.

In the old taco API compiler methods must be called explicitly to perform most tensor operations. When values are inserted to a tensor, the tensor must be packed after the last value insertion and before the tensor is used as input to compute a tensor expression. Similarly, after a tensor expression is defined, it must be compiled, assembled, and computed before accessing the values of the resulting tensor. Compiler methods can be called at any point within these time frames, so users have fine-grained control over when tensor operation execution happens. For example, users can use this framework to to avoid unnecessary computations or to time the execution of an operation. However, the necessity of compiler methods also leads to increased code size and code complexity. Furthermore, in many scenarios compiler methods cannot be leveraged to improve performance, so they become boilerplate code accompanying every declared operation.

The lazy execution framework instead executes compiler methods automatically when needed (At the last possible moment that preserves correctness.) For example, when an operation is declared on some tensor A the operation is compiled and executed when the user attempts to access values stored in A, either through a read method, or during the computation of a different expression. Compiler method execution is then indirectly determined by the operations the user decides to perform.

There are two situations where a tensor requires compiler method execution. The first is when values are inserted into a tensor and have not been packed. In this scenario, `pack` must be called before using the tensor to perform other actions. The second situation is when a tensor index expression is assigned to a tensor. Then the expression must be compiled, assembled and computed before the user can access any resulting tensor values. A tensor can therefore conceptually be in one of the following three states:

Do pack: values have been inserted into the tensor, but they have not been compiled into their target data structure.

Do compute: the tensor has been assigned a tensor operation, but it has not been computed.

Ready for access: the tensor does not require the execution of any other compiler methods.

The *do compute* state is further subdivided to keep track of independent executions of compile, assemble and compute. This subdivision allows users to explicitly execute any compiler method if needed.

Transition between tensor states is determined through method calls issued by the user such as read/write methods, tensor expression assignment, etc. We can group user actions into the following events:

Tensor Creation: When a tensor is created, it does not have data or an allocated data structure. New tensors start in the *do pack* state, indicating that they need a data structure to be part of an operation (even if they have no values.)

Tensor Modification: Users can perform tensor modifications via write methods (such as `insert`.) Writing to a Tensor will transition the tensor into the *do pack*. If the modified tensor T was previously in the *do compute* state, the modification triggers T to compute its expression, since modifications cannot be performed on tensors that have not yet been computed. On the other hand, if T is needed as an input for the computation of a different tensor T', the modification to T triggers T' to compute itself, since the computation must happen before its inputs are modified to preserve correctness. Finally, if T was already in the *do pack* state, the new modification is simply added to the set of modifications that will occur during the next call to `pack`.

Tensor Read: When the user calls a read method to access the values in a tensor T, the values in T must be updated before they are returned to the user. If T is in the *do pack* state, a call to `pack` is triggered before the read method is executed. Similarly, of T is in *do compute*, T's tensor expression is compiled, assembled and computed before data access.

Expression Assignment: When a index expression (such as `T(i,j) = B(i,j) + C(i,j)`) is declared, the expression is assigned to the *result* tensor (`T`), transitioning `T` to *do compute*. If `T` was previously in the *do pack* or *do compute* states then the unfinished the modifications/operations are discarded, since the new assignment replaces them. However, if the values of `T` are needed for the computation of a different tensor T' at the time of the assignment, T' must be computed before `T` replaces its previous values. In this case, the assignment triggers T' to compute itself, which in turn triggers T to update its current values. If T is in *do pack* or *do compute* when this happens, T first updates itself (via `pack` or `compute`), then T' computes itself, and finally the new expression is assigned to T. At this point, the tensor values in the *operands* are not yet required, so *operands* remain in their current state. However, the new dependency between all tensors in the expression must be recorded, so tensors can later comunicate with each other to coordinate computations.

Compiler Method Call: Users can also explicitly call compiler methods.

Calling `compile`, `assemble`, and `compute` will transition a tensor from *do compute* to `ready for access`. The user can perform just `compile`, or `compile` and `assemble`, and the lazy execution framework will execute the rest of the methods automatically when needed. However, the methods must be called in order, so users cannot call only `compute` or `assemble`.

Calling `pack` will transition a tensor from *do pack* to `ready for access`.

Tensor Deletion: When an expression is declared, the *result* tensor stores a smart pointer reference to the contents of all the expression *operands*. Therefore, if an *operand* T is deleted (either by the user, or when it goes out of scope), *result* tensors that depend on T will still have access to its contents for as long as they need it. Therefore, deleting a tensor does not affect other tensor computations.

Figure 5-1 illustrates tensor state transitions via a state diagram.

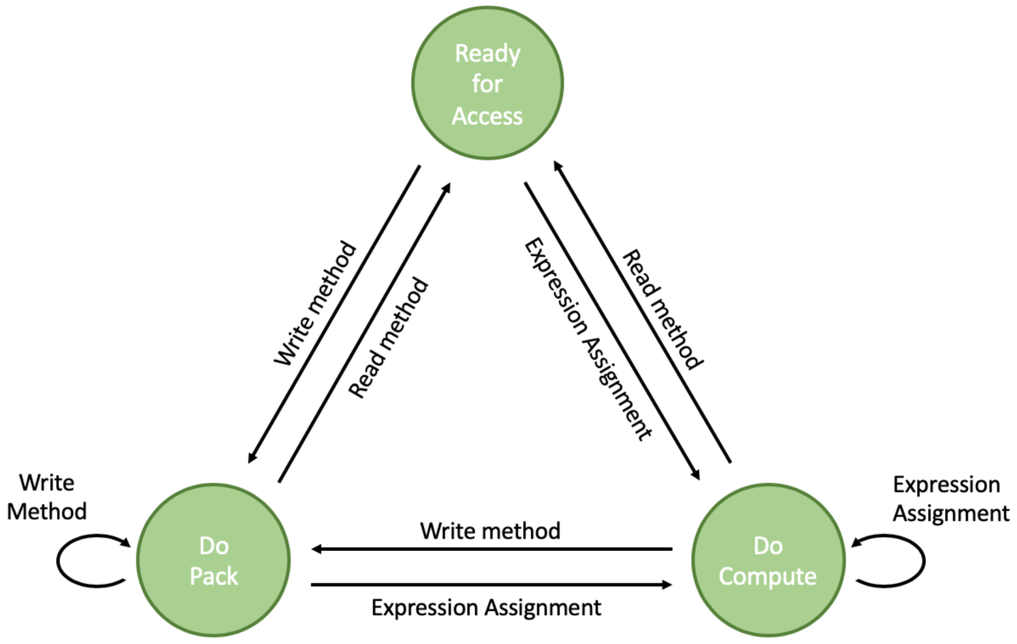

Figure 5-1: Tensor State Diagram.

When a tensor expression is defined, it introduces a two way dependency between the *result* tensor and each *operand* tensor. When the *result* tensor computes itself, it first must communicate to its operands that they are about to be accessed, and trigger them to update their values On the other hand, when an operand is about to be modified, it must trigger any result tensors that depend in it to compute themselves before the update proceeds. Tensors keep track of this dependencies with a double observer pattern:

> A *Result* tensor observes its operands, waiting for a notification indicating it should be computed (which happens when an operand is modified.)
>
> An *Operand* tensor observes the result tensors for all the operations it is an operand of, waiting for a notification indicating it should be computed (which happens when the result wants to compute its values.)

Notice that when a tensor expression is declared, its declaration does not trigger any computations. Therefore, an operand in one expression can also be the result in another that has not been computed, so the tensor is simultaneously observing a result tensor and its operands. Nevertheless, it is impossible for a tensor to depend on itself via a chain of computations: if a tensor `A` depends on tensor `B` (directly or indirectly), and then `A` is assigned an expression which operates on `B`, the assignment would trigger `A` to notify to all its dependents that it is about to be modified by the assignment. So before the assignment happens `A` will remove any dependencies on itself, including its previous dependency with `B`.

The double observer pattern is used to coordinate the correct resolution of computation dependencies. Tensors can receive different messages from other tensors:

> **Add to dependents**: When a tensor expression is declared, the *result* tensor sends a message to its operands to add it to their list of dependent tensors. Each *operand* tensor keeps track of all the tensors which depend on it as its observers.
>
> Notice that when a tensor expression expression is declared, the *result* tensor is informed of all its operands, so the operands do not send any *add to dependents* messages to the result. The Assignment itself is used by the result to keep track of, and message the result's operands.

Remove from dependents: When a tensor expression has been completed, the *result* tensor no longer depends on its operands, so it sends out a *remove from dependents* message. The operands then remove the result tensor from their list of dependents. This is done to prevent the operands from triggering future unrelated operations in the result.

Update: Both result and operand objects can send a *update* message to each other to trigger computation or packing. When an *operand* is about to be modified, it triggers all of its dependents to update in order to remove the existing dependencies and proceed with the modification. On the other hand, when a *result* tensor needs to be updated, it first triggers all its operands to compute so the *result* can correctly complete its computation.

When a tensor expression is assigned, the *result* tensor creates strong references to its operands to keep them alive, since they are required for the computation. As soon as the computation occurs, the references are removed. Operands instead hold weak references to their dependents, to avoid circular references and memory leaks.

We use the computation from figure 5-2 to run through an example of how the tensor state machine and the double observer pattern are used to achieve lazy execution. First, tensors `A`, `B`, and `C` are created, starting in the *do pack* state (1). Then, values are inserted to tensors `B` and `C`, so they stay in the *do pack* state (2). At (3), a tensor expression is defined. Through the expression, `B` and `C` are subscribed to `A` as operand observers. Then, `A` sends an *add to dependents* message to both `B` and `C`, becoming their observer as well. The assignment concludes and `A` transitions to the *do compute* state. At (4), `A` is accessed via a print statement. To compute itself, `A` first sends a *compute* message to both `B` and `C`. The message triggers `B` and `C` to call `pack`. Then, `A` compiles, assembles, and computes its expression, and sends a *remove from dependents* message to `B` and `C`. All tensors are now in the *ready for access* state. Finally, `A` performs the print operation and the program terminates. We provide a dependency diagram (figure 5-3) and a sequence diagram (figure 5-4) to further illustrate the example. Appendix B contains a more involved example of the lazy execution framework.

```
// (1)
Format csr({Dense,Compressed});
Tensor<double> A({10,10}, csr);
Tensor<double> B({10,10}, csr);
Tensor<double> C({10,10}, csr);

// (2)
B.setFromComponents(b_vals);
C.setFromComponents(c_vals);

// (3)
IndexVar i, j, k;
A(i,j) = B(i,k) * C(k,j);

// (4)
std::cout << A << std::endl;
```

Figure 5-2: Example matrix multiplication computation.

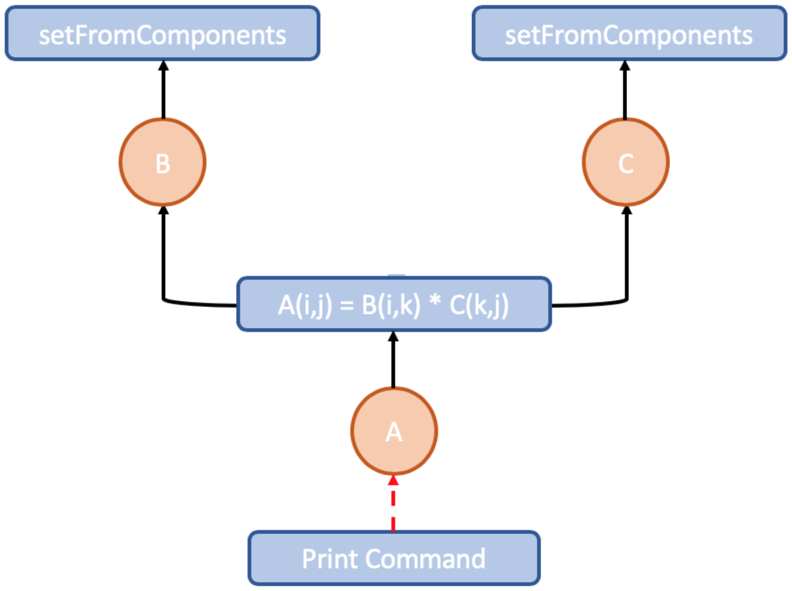

Figure 5-3: Tensor dependency diagram for the computation in figure 5-2. Black arrows indicate tensor and operation dependencies. Red dashed arrows indicate computation triggers.

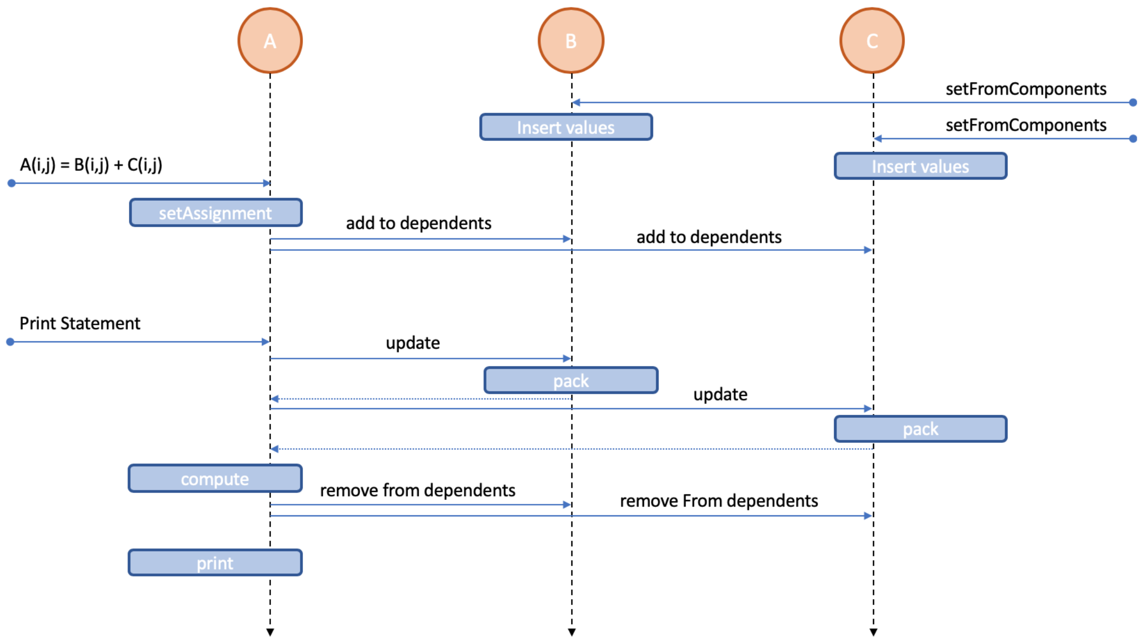

Figure 5-4: Tensor sequence diagram for the computation in figure 5-2.

Chapter 6

Code Generation

Each tensor format in taco defines a unique data structure to store the values it contains. Therefore, the implementation of methods to access and interact with tensors and tensor data is directly related to the format of the tensors. For example, a routine to access the value stored at a particular coordinate will be faster for a dense tensor compared to that of a tensor stored in the coordinate format (COO). Given the number of possible tensor formats, it is infeasible to write code to access each individual format to perform a tensor operation. A code generation approach is therefore needed to access different format data structures.

In this chapter we describe how taco's code generation capabilities can be used to generate code that executes tensor methods such as component access and insertion for any of the formats supported by the library. We then propose two new features to extend taco's code generation capabilities, and describe how they can be used to implement different interface methods and functionality.

6.1 Framing Methods as Tensor Operations

It is possible to frame many tensor read and write methods as tensor operations. For example, tensor packing from COO to compressed sparse fiber format (CSF) can be expressed in index notation with a simple tensor assignment operation `A(i,j) = B(i,j)`. To further illustrate this concept, we consider performing value insertion to a tensor coordinate using the `+=` operator.

```
...
Tensor <double> A({3,3}, format);

A.setFromComponents(a_vals);
std::cout  << A << std::endl;

A(0,2) += 5.0;
A(1,0) += 3.0;
A(1,2) += 2.0;
std::cout  << A << std::endl;
```

```
Output log:

>  1.0, 3.0, 0.0
   0.0, 2.0, 7.0
   0.0, 0.0, 1.0

>  1.0, 3.0, 5.0
   3.0, 2.0, 9,0
   0.0, 0.0, 1.0
```

Figure 6-1: taco routine for value insertion (left) and its output log (right).

Figure 6-1 shows taco syntax for inserting `5.0` to coordinate location `(0,2)` of tensor `A`. A routine to perform the presented component insertion typically involves first finding the in-memory location of coordinate `(0,2)` of tensor `A`, and then replacing the `0.0` currently stored at that location with `5.0`. The implementation of such routine requires knowledge of the format used to store tensor `A` in memory, since the location of `A(0,2)` is not the same if `A` is column major matrix or if it is a row major one.

The value insertion problem can instead be framed as a tensor addition operation. Inserting `5.0` to location `(0,2)` of `A` is equivalent to performing tensor addition of `A` and `T`, where `T` is a temporary tensor which contains only the value `5.0` at position `(0,2)`. Multiple value insertion assignments can be performed simultaneously by compiling the inserted values into `T` before performing the tensor addition. Figure 6-2 illustrates how the new values from figure 6-1 are inserted into `A` in a single operation

$$A = A' + T = \begin{pmatrix} 1 & 3 & 0 \\ 0 & 2 & 7 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 5 \\ 3 & 0 & 2 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 5 \\ 3 & 2 & 9 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 6-2: tensor component insertion framed as tensor addition.

Since taco is capable of generating code to perform tensor addition between tensors of arbitrary format, tensor `T` can be expressed in any format independently of the format `A` is stored in. For example, using the COO format to store `T` ensures that `T` has always constant size, regardless of its dimensionality.

In the old API, if a value is inserted into `A`, and then `A` is used in a tensor expression, `A` must be materialized into a single data structure before it can be used for the computation. If `A` is sparse, packing new values into `A` requires creating and allocating memory for a whole new data structure to hold all of `A`'s values. Inserting a few values into `A` before a computation can therefore introduce large overhead: it takes just as much time to iterate over the values of `A` to perform a single value insertion as it does to iterate over `A` to perform any other operation. Using `A' + T` instead of `A` also lets us remove the materialization overhead altogether. If only a small number of values are inserted into `A`, when `A` is used in a tensor expression, we can replace `A` with `A' + T` behind the scenes for code compilation and computation. Figure 6-3 shows how such a substitution would look like in index notation.

$$B(i,j) = A(i,k) * C(k,j)$$
$$\rightarrow$$
$$B(i,j) = (A'(i,k) + T(i,k)) * C(k,j)$$

Figure 6-3: Substituting `A` for `A' + T` to perform matrix multiplication.

This approach can be generalized to implement different tensor interface methods. For instance, general value insertion can be performed using the same approach with a *replace* routine, where values in tensor `A` are replaced by non-zeros in tensor `T` (instead of added together.) We describe how to extend the taco code generation algorithm to support such *replace* operation in section 6.3.

6.2 Code Generation Background

In this section, we give a brief description of the code generation technique introduced by Kjolstad et al. [15] to emit efficient code to compute any compound tensor algebra expression.

The code generation technique takes as input a tensor algebra expression in tensor index notation, which describes how each component in the output of a tensor computation can be computed in terms of components in the operands. Computing a tensor algebra expression in this form requires iterating over the merged iteration space of the operands dimension by dimension. For instance, efficient code that adds two sparse matrices must logically iterate only rows that have non-zeros in either matrix and, for each row, iterate only the columns that are non-zero in either matrix. How exactly tensor dimensions should be merged depends on the computation.

First, the algorithm determines the proper order in which to iterate over dimensions based on the order of index variables in the index expression as well as the order in which dimensions of the accessed tensor are stored. The algorithm then recursively generates code for each index variable in the input index expression. For each dimension i the algorithm determines the loops needed to simultaneously iterate over all tensors that contain dimension i. Multiplication requires iterating over the intersection of the operands as the result is non-zero only if both operands are non-zero. On the other hand, addition requires iterating over the union of the operands as the result is non-zero if at least one of the operands is non-zero. The algorithm generates loops for any arbitrary combination of intersection and union merges to effectively perform the specified tensor operation. Finally, at each loop level the algorithm emits code to compute values as necessary to calculate the tensor operation. Figure 6-4 shows examples of code generated by taco to perform a tensor multiplication.

```
// A(i,j) = B(i,k) * C(k,j)
// B and C have a csr format
for (int32_t iB = 0; iB < B1_dimension; iB++) {
  for (int32_t pB2 = B2_pos[iB]; pB2 < B2_pos[(iB + 1)]; pB2++) {
    int32_t kB = B2_crd[pB2];
    double tk = B_vals[pB2];
    for (int32_t pC2 = C2_pos[kB]; pC2 < C2_pos[(kB + 1)]; pC2++) {
      int32_t jC = C2_crd[pC2];
      int32_t pA2 = iB * A2_dimension + jC;
      A_vals[pA2] = A_vals[pA2] + tk * C_vals[pC2];
    }
  }
}
```

Figure 6-4: Code generated by taco to perform CSR matrix multiplication.

6.3 The *Replace* Tensor Operation

In section 6.1 we describe how to perform value insertions declared with the `+=` operator using tensor addition. In order to generate code to perform general value insertion, we propose to extend taco with a new tensor operation: *replace*.

Replace is a binary tensor operation which takes as input two tensors `A` and `B`. `A (replace) B` is computed by taking `A`, and replacing values in `A` with values in `B` for each non-zero component in `B`. In figure 6-5 we show an example of how *replace* is computed.

$$A(replace)B = \begin{pmatrix} 1 & 3 & 0 \\ 0 & 2 & 7 \\ 0 & 0 & 1 \end{pmatrix} (replace) \begin{pmatrix} 0 & 0 & 5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 5 \\ 0 & 2 & 7 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 6-5: Tensor replace operation example.

Computing `A (replace) B` requires iterating over the disjunction of the dimensions of `A` and `B`, since for each coordinate, only one term needs to be non-zero for the result to be non-zero. This corresponds to the union of the iteration spaces of each merged dimension, so the iteration loops needed to compute *replace* are then the same as those emitted to compute *addition*. Therefore, extending taco to support *replace* requires minimal changes to the code generation algorithm. Figures 6-6 and 6-7 compare taco code generated for tensor addition and for the new tensor replace operation.

```
// A(i,j) = B(i,j) + C(i,j)
for (int32_t iB = 0;
     iB < B1_dimension; iB++) {
  for (int32_t jB = 0;
       jB < B2_dimension; jB++) {
    int32_t pB2 = iB * B2_dimension + jB;
    int32_t pC2 = iB * C2_dimension + jB;
    int32_t pA2 = iB * A2_dimension + jB;
    A_vals[pA2] = B_vals[pB2] + C_vals[pC2];
  }
}
```

```
// A(i,j) = B(i,j) (replace) C(i,j)
for (int32_t iB = 0;
     iB < B1_dimension; iB++) {
  for (int32_t jB = 0;
       jB < B2_dimension; jB++) {
    int32_t pB2 = iB * B2_dimension + jB;
    int32_t pC2 = iB * C2_dimension + jB;
    int32_t pA2 = iB * A2_dimension + jB;
    A_vals[pA2] = B_vals[pB2];
    if (C_vals[pC2] != 0) {
      A_vals[pA2] = C_vals[pC2];
    }
  }
}
```

Figure 6-6: Comparison between code to compute tensor addition and tensor replacement with dense matrices.

```
// A(i,j) = B(i,j) + C(i,j)
for (int32_t iB = 0;
     iB < B1_dimension; iB++) {
  int32_t pC2 = C2_pos[iB];
  int32_t C2_end = C2_pos[(iB + 1)];
  int32_t jB = 0;
  int32_t B2_end = B2_dimension;
  while (pC2 < C2_end) {
    int32_t jC = C2_crd[pC2];
    int32_t pB2 = iB * B2_dimension + jB;
    int32_t pA2 = iB * A2_dimension + jB;
    if (jC == jB) {
      // Value addition
      A_vals[pA2] = B_vals[pB2] + C_vals[pC2];
    }
    else {
      A_vals[pA2] = B_vals[pB2];
    }
    pC2 += (int32_t)(jC == jB);
    jB++;
  }
  while (jB < B2_end) {
    int32_t pB2 = iB * B2_dimension + jB;
    int32_t pA2 = iB * A2_dimension + jB;
    A_vals[pA2] = B_vals[pB2];
    jB++;
  }
}
```

```
// A(i,j) = B(i,j) (replace) C(i,j)
for (int32_t iB = 0;
     iB < B1_dimension; iB++) {
  int32_t pC2 = C2_pos[iB];
  int32_t C2_end = C2_pos[(iB + 1)];
  int32_t jB = 0;
  int32_t B2_end = B2_dimension;
  while (pC2 < C2_end) {
    int32_t jC = C2_crd[pC2];
    int32_t pB2 = iB * B2_dimension + jB;
    int32_t pA2 = iB * A2_dimension + jB;
    if (jC == jB) {
      // Value replacement
      A_vals[pA2] = C_vals[pC2];
    }
    else {
      A_vals[pA2] = B_vals[pB2];
    }
    pC2 += (int32_t)(jC == jB);
    jB++;
  }
  while (jB < B2_end) {
    int32_t pB2 = iB * B2_dimension + jB;
    int32_t pA2 = iB * A2_dimension + jB;
    A_vals[pA2] = B_vals[pB2];
    jB++;
  }
}
```

Figure 6-7: Comparison between code to compute tensor addition and tensor replacement with CSR matrix C.

Tensor Component Insertion. We introduce *replace* into taco to support code generation for general value insertion without packing. We extend the approach shown in section 6.1 to value insertion by replacement, in addition to the shown value insertion by addition.

To implement tensor insertion, the `Tensor` object first compiles inserted values to a temporary buffer in COO format. To materialize `A` we then compute `A = A'(replace)T`, where `A'` contains `A`'s original stored values. This process replaces the old API's hand-implemented `pack` routine. When the old `pack` is called, it replaces the current values of `A` with the new inserted values. On the other hand, the new approach retains both the current values of `A` and the new inserted values. In addition, the new approach automatically supports new tensor formats as soon as they are supported by the code generation algorithm.

As described in section 6.1, we can also remove the tensor materialization overhead by using `A'+T` instead of `A` during tensor expression computation, as shown in figure 6-8. With this approach it is no longer required to call `pack` before tensor computation (implicitly or explicitly.)

```
Format csr({Dense,Compressed});
Tensor<double> A({10,10}, csr);
Tensor<double> B({10,10}, csr);
Tensor<double> C({10,10}, csr);

...

// Values inserted to temporary buffer T
A(0,0) = 4.0;
A(2,4) = 1.0;
A(7,6) = 4.0;
A(9,9) = 4.0;

// At this point, A is represented as
// A(i,j) = A'(i,j) (replace) T(i,j)

...

IndexVar i, j;
B(i,j) = A(i,j) + C(i,j);

// We internally substitute A for A' + T during computation:
// B(i,j) = (A'(i,j) (replace) T(i,j)) + C(i,j)
std::cout << B << std::newl;
```

Figure 6-8: Example of substituting pack with replace.

6.4 Ranged dimension iteration

The goal of ranged dimension iteration is to allow iteration over a subset of each dimension's iteration space. Ranged dimension iteration lets the user iterate over a specific numerical range of a dimension, defined by its start and end bounds, as well as an optional positive stride. We extend taco with ranged dimension iteration to support operations with the slice index notation introduced in section 4.3.2.

To generate code for ranged dimension iteration, we introduce a transformation over the iteration loops emitted by the code generation algorithm. For each dimension, we introduce an abstract iteration index which represents the index of iteration over the specified range (in addition to the concrete index which keeps track of the iteration over the complete tensor dimension.) The abstract index is used to adjust all concrete indices to their respective iteration range. The adjustments to index variables differ depending on the format of each dimension. To illustrate this, we demonstrate generated code for iterating over a CSR matrix in the right side of figure 6-9. Dimension i is dense, so we iterate directly using the abstract index `range_i`.

We then compute the real indices `iA` and `iB` corresponding to `range_i` for matrices `A` and `B`, respectively. Dimension j is compressed, so we cannot directly iterate over the abstract index `range_j`. Instead, we iterate over the compressed dimension of `B`, and infer `range_j` from `jB`. Generating code for ranged iteration affects only the syntax of each loop, while the overall structure of the iteration, as well as the code emitted to perform the requested operation remain unchanged.

```
// A(i,j) = B(i,j)
for (int32_t iB = 0;
     iB < B1_dimension;
     iB++) {


  for (int32_t pB2 = B2_pos[iB];
       pB2 < B2_pos[(iB + 1)];
       pB2++) {
    int32_t jB = B2_crd[pB2];








    int32_t pA2 =
        iB * A2_dimension + jB;
    A_vals[pA2] = B_vals[pB2];
  }
}
```

```
// A(i,j) = B(i(2,12,2),j(10,20))
for (int32_t range_i = 0;
     range_i < 5;
     range_i++) {
  int32_t iA = ir;
  int32_t iB = 2 * range_i + 2;
  for (int32_t pB2 = B2_pos[iB];
       pB2 < B2_pos[(iB + 1)];
       pB2++) {
    int32_t jB = B2_crd[pB2];
    int32_t range_j = jB - 10;
    int32_t jA = range_j;
    if (range_j >= 10) {
      break;
    }
    if (range_j < 0) {
      continue;
    }
    int32_t pA2 =
        iA * A2_dimension + jA;
    A_vals[pA2] = B_vals[pB2];
  }
}
```

Figure 6-9: Code generated for regular iteration (left) and range iteration (right) of a CSR matrix.

Each tensor in an expression can specify its own range for the iteration over a dimension. However, ranges over the same dimension are required to have the same size. This ensures proper coiteration across tensors which define different ranges for the same dimension. Figure 6-10 shows an example of coiteration over different index ranges for a matrix addition operation.

Slice Index Notation. We can support *slice index notation* as described in section 4.3.2 as a direct application of ranged dimension iteration. This allows taco to restrict the range of a tensor dimension to operate on tensors with different dimension sizes, as well as to reduce the dimensionality of a tensor altogether (figure 6-11). It also lets us implement sub-tensor value insertion(figure 6-12). Finally, slice index notation also allows us to extract tensor slices that have a completely different format

```
// A(i,j) = B(i,j) + C(i,j)

for (int32_t iB = 0;
     iB < B1_dimension; iB++) {



  for (int32_t jB = 0;
       jB < B2_dimension; jB++) {



    int32_t pB2 = iB * B2_dimension + jB;
    int32_t pC2 = iB * C2_dimension + jB;
    int32_t pA2 = iB * A2_dimension + jB;
    A_vals[pA2] =
        B_vals[pB2] + C_vals[pC2];
  }
}
```

```
// A(i,j) = B(i(0,10),j(5,15,3))
//        + C(i(10,30,2),j(7,12))
for (int32_t range_i = 0;
     range_i < 10; range_i++) {
  int32_t iA = range_i;
  int32_t iB = range_i;
  int32_t iC = range_i * 3 + 5;
  for (int32_t range_j = 0;
     range_j < 10; range_j++) {
    int32_t jA = range_j;
    int32_t jB = range_j * 2 + 10;
    int32_t jC = range_j + 7;
    int32_t pB2 = iB * B2_dimension + jB;
    int32_t pC2 = iC * C2_dimension + jC;
    int32_t pA2 = iA * A2_dimension + jA;
    A_vals[pA2] =
        B_vals[pB2] + C_vals[pC2];
  }
}
```

Figure 6-10: Matrix addition with iteration over distinct dimension ranges.

than the format of the tensor they originated from (fig:slice-format-change).

```
Format dm({Dense,Dense});
Tensor<double> B({10,10}, dm);

Format dv({Dense});
Tensor<double> a({10}, dv);
Tensor<double> a({10}, dv);
B.setFromComponents(b_vals);
c.setFromComponents(c_vals);

// Reduce dimensionality of B for vector addition.
IndexVar i, j;
a(j) = B(i(3),j) + c(j);
```

Figure 6-11: Tensor dimension reduction to perform vector addition over a matrix slice.

```
Format csr({Dense,Compressed});
Tensor<double> A({100,100}, csr);
Tensor<double> B({10,10},   csr);
A.setFromComponents(b_vals);
B.setFromComponents(c_vals);

// Insert B as a subtensor to A
IndexVar i, j;
A(i(0,10),j(50,60)) = B(i,j);
```

Figure 6-12: Tensor sub-tensor insertion.

Tensor Component Access. Ranged dimension iteration also lets us generate code to implement the tensor component access behavior provided by the `at` method or

```
Format csf({Dense,Compressed,Compressed});
Tensor<double> A({20,100,20}, csf);

Format dm({Dense,Dense});
Tensor<double> B({20,20},   dm);
A.setFromComponents(b_vals);

// Extract slice of A with different format
IndexVar i, j, k;
B(i,j) = A(i, k(10), j);
```

Figure 6-13: Accessing a tensor slice and changing its format.

via `ScalarAccess` objects, as described in section 4.1.3. Access to the value stored at a specific tensor coordinate location can be implemented by restricting the iteration range of all dimensions on the target tensor to a specific tensor coordinate (figure 6-14).

```
// Value access expressed as index notation:
// x = A(i(3), j(5))
double x = A.at(3,5);
```

Figure 6-14: Implementing tensor value access with ranged dimension iteration.

Chapter 7

Related Work

Related work in this area consists of the exploration of different existing systems for sparse and dense linear and tensor algebra, with a focus on their tensor interfaces and supported operations.

Dense and sparse linear and tensor algebra systems There exists a number of linear and tensor algebra libraries, each of which provides a different set of methods and operations which can be used to interact with tensors. Some of the most popular libraries for computing on sparse matrices and tensors are Eigen [10], MATLAB [4], Julia [6], TensorFlow [1], MKL [11] and PETSc [5]. MATLAB, Eigen, and Julia are general systems that support all basic linear algebra operations; however, their sparse matrix formats are limited. PETSc targets supercomputers and supports distributed and blocked matrix formats. TensorFlow is a recent machine learning framework developed by Google where dense (and some sparse [9]) tensors are passed between tensor computation kernels in a dataflow computation. Intel MKL is a math processing library for C and Fortran that is heavily optimized for Intel processors. The proposed API draws in particular from the set of features available in Eigen's and MATLAB's Tensor Toolbox interfaces.

Eigen Eigen is a C++ linear algebra library which has become popular due to its high performance and relative ease of use, and it is used in many large-scale projects such as Google's TensorFlow [1]. It supports a robust set of dense linear algebra operations including linear solvers, eigenvalue finders, a variety of factorizations and decompositions, among other. Eigen also supports many sparse linear algebra operations with a custom column sparse column (or Row) (CSC/CSR) format which allows for faster amortized random value insertion.

Eigen provides multiple ways to initialize matrix values. Users can specify values one at a time by inserting the values one after the other. They can also declare all the matrix values at matrix initialization via the so-called comma-initializer syntax, in which the whole list of coma-separated values is presented upon matrix creation. The most relevant initialization form provided by Eigen is their `setFromTriplets` method, which allows users to provide the values of a matrix in the form of a list of components (or Triplets in Eigen). Having access to the full list of non-zeros allows Eigen to efficiently construct CSR/CSC data structures, and users can additionally provide handy duplicate-element policies for any repeated input value coordinates.

Other useful interface characteristics include:

- component access and modification via overloaded parenthesis operators.

- Matrix resizing and reshaping.

- Data mapping into a Matrix. Eigen provides a `Map` class which allows users to conceptually map a Matrix (or Vector) to an existing array of data. This allows users to use Eigen's interface to access non-Eigen data structures with little to no overhead. `Map` can also be used to implement views to a matrix (e.g. to implement matrix reshaping or slicing.)

- Access to matrix subsections via blocks and slices.

MATLAB Tensor Toolbox The MATLAB Tensor Toolbox is an early system for sparse tensors that supports many tensor operations with the coordinate format and other formats for factorized tensors [4]. It implements a number of sparse tensor factorization algorithms in addition to supporting primitive operations on general (unfactorized) sparse and dense higher-order tensors.

The MATLAB Tensor Toolbox provides a similar set of methods to interact with tensors as Eigen does with Matrices. It implements multiple value initialization routines,component access via overloaded parenthesis operators, matrix reshaping, slicing and blocking, among others. In particular, the Tensor Toolbox introduces the notion of performing block and slice accesses via operator parenthesis overloading, enables very simple and concise notation for expressing blocks and slices (This differs from Eigen, where slice functionality is implemented with `Map`, and blocks have their own `block` method.)

Chapter 8

Discussion

In this section we discuss the implications of our work in this thesis, and we give insight into some of the challenges motivation behind the design of the new taco API.

Work on a new API was motivated by user feedback regarding the difficulty of using the first release of taco as a tensor library. Compiler methods often make code complicated and too verbose. Additionally, the first taco API lacked important tensor functionality. The new API was desiged with the following goals:

- Simplify taco syntax by making compiler method calls optional
- Provide common tensor algebra methods and notation, such as value access via tensor indexing.
- Make taco syntax similar to other libraries to increase code portability between them.
- Hide the taco compiler from the user to make the library easier to use, and reduce the barrier of entry for new users.

Tensor Value Insertion. The new API introduces `setFromComponents` as a new method to insert values to a tensor. The inclusion of this method was a source of debate, as `insert` can be used to achieve equivalent behavior. Ultimately, we believe `setFromComponents` and `insert` provide functionality that will be convenient in different scenarions. `insert` is meant to ease the scripting process in taco; insertion via `ScalarAccess` objects provides convenient syntax for users to test and try out different functionality, inserting tensor values by hand. On the other hand, `setFromComponents` provides a mechanism to divide the process of fetching tensor data and inserting it to a tensor. Users can write routines to extract tensor data from any backend, and then insert it all at once. Furthermore, specifying a duplicate value policy can make the overall data fetching process simpler.

`ScalarAccess` and operator overloading. As part of the new API, we wanted to allow indexing notation for tensor value access in addition to tensor index notation. However, the C++ limitations on operator overloading and template metaprogramming quickly became a roadblock for this feature.

We first considered introducing overloading `operator[]` to achieve indexing syntax similar to python's numpy [], but C++ limits the number of arguments of `operator[]` to 1, as the `C` and C++ standard indexing conventions assume a nested style of indexing appropriate for multidimensional array access. Our next option was to use `operator()` instead, as it allows for multiple inputs. Template variadic arguments are required to overload `operator()` to take in a variable number of arguments. However, the template variadic argument interface does not provide any sort of built-in type-checking mechanism, which makes it impossible to distinguish between variadic arguments of different types for template specialization.

We eventually solved this limitation by abusing C++ SFINAE techniques to force `operator()` specialization. Nevertheless, this problem illustrates how domain specific language development pushes the limit of what existing languages are capable of expressing.

`ScalarAccess` vs read/write methods. In the new API `ScalarAccess` objects enable component insertion and access with coordinate indexing notation, which enable clean and elegant syntax for performing these actions. However, the API still provides regular `insert` and `at` methods. The reason why `insert` and `at` are important is because `ScalarAccess` objects are not available for the `TensorBase` class. Since `ScalarAccess` allow direct insertion and access with minimal syntax, they must be templatized on the tensor type. Enabling `ScalarAccess` objects for `TensorBase` objects would require providing input template parameters to the overloaded `operator()`. This is possible, but the notation for calling `operator()` with template parameters is complex and ugly, so we decided to keep `insert` and `at` as the main methods of interaction with `TensorBase` objects.

Block Formats. Taco is capable of representing block formats using higher order tensors. The new API formalizes this notion with the introduction of block formats. Block formats are therefore not technically required, as they do not extend the set of functionality provided by taco. After all, block formats are just a layer of abstraction on top of tensors of higher dimensionality. The main motivation for introducing block formats is to make blocking more intuitive from a user stand point.

Aside from user convenience, block formats extend the notion of tensor formats in taco. Before, tensors were grouped by the number of modes in their formats, but now even tensors with the same number of dimensions can be stored as formats with a completely different number of modes and mode formats. By continuing to extend the notion of formats, it might be possible to achieve a completely new set of tensor optimizations.

The Lazy Execution Framework. We introduce the lazy execution framework and the tensor state machine as a mechanism to hide compiler methods from the user. However, there exists easier ways to achieve this. For example, it would be really simple to execute tensor expressions the moment they are defined, just like in traditional libraries. One reason to not do this is that delaying execution can lead to performance savings if the result of an operation is never used. The real motivation for our decision however, is that taco is at its core designed to have a scheduling language, allowing users to change the way in which a computation is performed without affecting the resulting tensor. In future taco releases, after a tensor expression is defined the user will be able to command modifications to the tensor computation which should be taken into consideration during expression compilation. However, it is impossible to predict when a user will specify additional scheduling commands, so delaying execution until the result is requested is the only way to ensure there will be no more scheduling commands added to the operation. Some scheduling constructs are already available in taco, such as adding workspaces [13] to speed up tensor operations, and the taco development team intends to add more.

Tensor Data Representation. In Chapter 6 we describe how databuffers in the `Tensor` class can be though of as COO tensors that can be used as input for code generated by taco. This turned out to be a powerful idea which enables us to automatically generate the implementation of different tensor API methods for any format supported by taco. We also explored the idea of using multiple storage objects to represent a single tensor during larger tensor expressions to save up the costs of materializing the operand tensors.

We use a data buffer to amortize the cost of inserting values via `operator+=`, as well as inserting values via `insert`. By using not one, but two COO data buffers, we can combine this techniques to amortize both kind of insertions simultaneously, using multiple data structures to represent a single tensor. Using different combinations of data structures can be explored in the future to achieve improved performance for a number of different tensor operations, as well as to enable functional programming in taco, in which expressions can be nested into each other to generate single kernels for large compound tensor expressions.

Chapter 9

Conclusion

We have described and implemented a new API for the taco library that improves and extends the functionality available to taco. We presented a lazy execution framework implemented as a tensor state machine that hides compiler methods from the user behind the tensor abstraction. We then introduce extensions to the taco code generation algorithm that let us automatically generate the implementation of different API methods and functionality such as tensor packing, value insertion, value access, and slice index notation. Our API provides a more complete set of tools that enables users to use the taco library with ease. One major implication of the API is that it makes taco syntax similar to traditional compiler-free libraries, by hiding all compiler functionality behind a user friendly API. This means that users have access to a familiar-looking library with extensive compiler-enabled functionality. The introduction of block formats makes it easier for users to understand how block operations are achieved in taco. In addition, the linear algebra API makes porting code from other libraries easier, and will hopefully help draw in users who do not need higher dimensionality tensors, as linear algebra notation is more common than index expressions. Finally, the introduction of compatibility with Eigen will dramatically increase the number of tools readily available to users when using taco.

Future work includes extending taco's API to allow users to exploit taco's code generation capabilities with custom user-defined behavior. For example, the taco API could provide a `apply` method, which takes as input a function pointer, and generates code which iterates over the tensor and applies the specified function to each tensor component. This `apply` method could be used to implement a large number of new operations such as:

- Element-wise operations: applying a mathematical operation (such as sin, cos, modulo, floor, ceiling, etc) to each tensor value. This is functionality that is notably currently missing from taco.
- Aggregation operations: implementing aggregation operators directly within the iteration loops generated by taco to find average, count, max, min, sum, etc.
- Other: users could bake application specific code into taco generated code, implement custom iterators, etc.

This could be further extended to allow users to specify custom operations between multiple tensors. Another direction of future work is to extend taco compatibility to other C++ libraries such as MKL to extend the range of features easily accessible from taco. The code generation algorithm may also be extended to implement more API methods and functionality such as tensor printing, and most notably file IO operations to read and write tensors in files.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[2] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *J. Mach. Learn. Res.*, 15:2773–2832, January 2014.

[3] Brett W. Bader, Michael W. Berry, and Murray Browne. *Discussion Tracking in Enron Email Using PARAFAC*, pages 147–163. Springer London, 2008.

[4] Brett W Bader and Tamara G Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[5] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, pages 163–202. Birkhäuser Boston, Boston, MA, 1997.

[6] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing, 2012.

[7] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA):123:1–123:30, October 2018.

[8] Albert. Einstein. The Foundation of the General Theory of Relativity. *Annalen der Physik*, 354:769–822, 1916.

[9] Google. Tensorflow sparse tensors. http://www.tensorflow.org/api_guides/python/sparse_ops, 2017.

[10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[11] Intel. Intel math kernel library reference manual. Technical report, 630813-051US, 2012. `http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf`, 2012.

[12] Venera Khoromskaia and Boris N. Khoromskij. Tensor numerical methods in quantum chemistry: from hartree-fock to excitation energies. *Phys. Chem. Chem. Phys.*, 17:31491–31509, 2015.

[13] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Sparse tensor algebra optimizations with workspaces. *ArXiv e-prints*, April 2018.

[14] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 943–948, Oct 2017.

[15] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[16] Joseph C Kolecki. An Introduction to Tensors for Students of Physics and Engineering. *Unixenguaedu*, 7(September):29, 2002.

[17] Julian McAuley and Jure Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 165–172. ACM, 2013.

[18] Gregorio Ricci-Curbastro and Tullio Levi-Civita. Méthodes de calcul différentiel absolu et leurs applications. *Mathematische Annalen*, 54, 1901.

Appendix A

Tensor API Documentation

In this appendix we provide an API instruction manual for the different taco C++ objects relevant for this thesis.

A.1 Coordinate

`Coordinate` is a class representing a tensor multidimensional coordinate.

Template Parameters

`size_t` Order. The order, or number of dimensions in the coordinate.

`typename` Type. The integral Type of the coordinates.

Member Functions

`(Constructor)`	Construct Coordinate
`operator[]`	Access coordinate
`operator std::vector<int>`	Cast Coordinate to coordinate vector
`order`	Return Coordinate order

A.1.1 Member Functions

(Constructor)

Constructs a Coordinate, initializing its contents depending on the constructor version used:

empty constructor (default constructor) Constructs an empty Coordinate.

coordinate constructor Constructs a coordinate with its `Order` dimensions populated.

Parameters

coordinates. A variadic argument of type `Type`. The constructor receives a variable number of dimensions which must match the order of the coordinate.

operator[]

Returns a reference to the coordinate at position `n` in the vector container.

Parameters

n. Integral position of a coordinate element in the container. Notice that the first coordinate element has a position of 0.

Return Value

The `Type` coordinate value at the specified position in the Coordinate.

operator std::vector<int>

Casting operator to a vector object. Returns a vector representation of the coordinate.

Return Value

An `std::vector<int>` of size `Order` containing the coordinate elements in the container.

order

Return the order of the coordinate.

Return Value

The `size_t` order of the Coordinate object.

A.2 Component

the `Component` class allows the user to represent tensor components consisting on a tensor coordinate and its associated value.

Template Parameters

`size_t` Order. The order, or number of dimensions in the coordinate.

`typename` CType. The Type of the component value.

Member Functions

`(Constructor)`	Construct Component
`coordinate`	Return component coordinate
`value`	Return component value

A.2.1 Member Functions

(Constructor)

Constructs a Component, initializing its contents depending on the constructor version used:

empty constructor (default constructor) Constructs an order 0 component with value 0.

coordinate/value constructor Constructs a coordinate with its coordinate and value specified.

Parameters

coordinate. `Coordinate<Order>` the coordinate of the component

value. `CType` value stored at the component.

coordinate

Returns the `Coordinate` object corresponding to the Component.

Return Value

The `Coordinate<Order>` coordinate of the component.

value

Returns the `Ctype` value stored at the Component.

Return Value

The `CType` value of the component.

A.3 IndexVar

Index variables are used to index into tensors in index expressions, and they represent iteration over the tensor modes they index into.

Member Functions

`(Constructor)`	Construct IndexVar
`getName`	Return indexVar name
`operator()`	Return RangeIndexVar

A.3.1 Member Functions

(Constructor)

Constructs an IndexVar, initializing its contents depending on the constructor version used:

empty constructor (default constructor) Constructs an index variable with automatically assigned name.

name constructor Constructs an index variable with specified name.

Parameters

name. `string` name of the index, which is used to represent the indexVar during code generation.

getName

Returns the name of the IndexVar.

Return Value

The `string` name of the IndexVar.

operator()

Creates a `RangeIndexVar` associated with the creator index variable. The generated `RangeIndexVar` iterates over the same dimension as this `IndexVar`, but additionally specifies a tensor range to which to limit the iteration.

Parameters

start_bound. `int` index at which to begin the iteration range.

end_bound. `int` index at which to end the iteration range.

stride. `int` the iteration stride. Increment by which the iteration index is incremented at every iteration step.

shift. `int` the iteration shift. Increment by which the iteration range should be shifted within the corresponding dimension.

Example Usage

```
IndexVar i;

// index var iterating over the indices 2-4 of dimension i
RangeIndexVar i1 = i(2,5);

// index var iterating over every other index
// bewteen 10 and 20 of dimension i
RangeIndexVar i2 = i(10,20,2);

// index var masking dimension indices
// outside of the 2-4 index range.
RangeIndexVar i3 = i(2,5,1,0);
```

A.4 RangeIndexVar

Index variable used to iterate over a specified range of a tensor dimension in an index expression

Inherits from `IndexVar`.

Member Functions

`(Constructor)`	Construct RangeIndexVar
`startBound`	Return the index range start bound.
`endBound`	Return the index range stride.
`stride`	Return the index range start bound.

A.4.1 Member Functions

(Constructor)

Constructs a RangeIndexVar, initializing its contents. RangeIndexVar objects require specifying an iteration range and stride.

Parameters

start_bound. `int` index at which to begin the iteration range.

end_bound. `int` index at which to end the iteration range.

stride. `int` the iteration stride. Increment by which the iteration index is incremented at every iteration step.

startBound

Returns the start bound of the iteration range.

Return Value

The `int` start bound of the RangeIndexVar.

endBound

Returns the end bound of the iteration range.

Return Value

The `int` end bound of the RangeIndexVar.

stride

Returns the stride of the iteration.

Return Value

The `int` stride of the RangeIndexVar.

A.5 ModeFormat

The type of a mode defines how it is stored. For example, a mode may be stored as a dense array, a compressed sparse representation, or a hash map. taco defines ModeFormats for the user, but new formats can be defined by extending ModeTypeImpl.

Member Functions

`(Constructor)`	Construct ModeFormat
`operator()`	Instantiates a variant of the mode format with differently configured properties.
`getName`	Return the mode format name.
`hasProperties`	Return true if the mode format has the given properties.
`isFull`	Returns true if the mode format has the full property.
`isOrdered`	Returns true if the mode format has the ordered property.
`isUnique`	Returns true if the mode format has the unique property.
`isBranchless`	Returns true if the mode format has the branchless property.
`isCompact`	Returns true if the mode format has the compact property.
`hasCoordValIter`	Returns true if a mode format has the coordinate value iteration capability.
`hasCoordPosIter`	Returns true if a mode format has the coordinate position iteration capability.
`hasLocate`	Returns true if a mode format has the locate capability.
`hasInsert`	Returns true if a mode format has the insert capability.
`hasAppend`	Returns true if a mode format has the append capability.
`hasFixedSize`	Returns true if a mode format is a fixed size format.
`size`	Returns the fixed size of the format.
`defined`	Return true if the mode format is defined.

A.5.1 Member Functions

(Constructor)

Constructs a ModeFormat, initializing its contents depending on the constructor version used:

empty constructor (default constructor) Constructs an undefined mode format.

name constructor Constructs a new mode format.

Parameters

implementation. `ModeTypeImpl`, a predefined mode format implementation.

operator()

Instantiates a variant of the mode format with differently configured properties, depending on the constructor version used:

property list operator() Constructs a mode format with the specified properties.

format size operator() Constructs a new mode format with a fixed mode size.

Parameters

property list. `vector<Property>` list of properties to instantiate a mode format with.

size. `int` mode format size. Fixed size mode formats are used to define block tensor formats.

Example Usage

```
ModeFormat FixedDense = ModeFormat::Dense(5);
```

getName

Returns the name of the ModeFormat.

Return Value

The `string` name of the ModeFormat.

hasProperties

Returns true if the mode format has the given properties.

Parameters

property list. `vector<Property>` list of properties to check for.

Return Value

The `bool` true if the ModeFormat contains all the specified properties.

hasFixedSize

Returns True if the ModeFormat has a fixed size.

Return Value

The `bool` true if the ModeFormat has a defined fixed size.

size

Returns the ModeFormat's fixed size. Raises an error if the modeformat does not have a fixed size.

Return Value

The `int` size of the mode format.

defined

Returns True if the ModeFormat has been defined. An undefined mode type can be used to indicate a mode whose format is not (yet) known.

Return Value

The `bool` true if the ModeFormat is defined.

A.6 Format

A Format describes the data layout of a tensor, and the sparse index data structures that describe locations of non-zero tensor components.

Member Functions

`(Constructor)`	Construct Format.
`getOrder`	Return the number of modes in the format.
`getModeFormats`	Return the the storage types of the modes.
`getModeFormatPacks`	Return the storage formats of the modes, with modes that share the same physical storage grouped together
`getModeOrdering`	Return the ordering in which the modes are stored.
`isBlocked`	Return true if the Format is a Block format
`numBlockLevels`	Return the number of nested block levels in the format
`getBlockSizes`	Return the fixed-sizes of ModeFormats in the format.
`getDimensionFree SizeBlock`	returns the block which contains each dimension's free size

A.6.1 Member Functions

(Constructor)

Constructs a Format, initializing its contents depending on the constructor version used:

empty constructor (default constructor) Constructs a format for a 0-order tensor (a scalar).

standard constructor Constructs a tensor format whose modes have the given mode storage formats.

ordered format constructor Constructs a tensor format where the modes have the given mode storage formats and modes are stored in the given sequence.

block format constructor Constructs a tensor format whose modes have the given mode storage formats, and the mode formats have been grouped into blocks.

Parameters

mode formats. `vector<ModeFormat>`, a list of mode formats. The number of mode formats in the input determines the dimensionality of the format.

blocked mode formats. `vector<vector<ModeFormat»`, a list of mode format groupings. Each grouping represents a block or block level. All groupings are required

to be the same size. The number of blocks in the input determines the dimensionality of the format.

Blocked formats create a level of abstraction in which each dimension of a tensor is divided into multiple modes. For each dimension, there must be exactly one block with non-fixed size.

mode ordering. `vector<int>`, a mode format ordering. The mode stored in position i of the input mode formats is specified by modeOrdering[i].

Example Usage

```
// 4-tensor format
Format({Dense,Dense,Compressed,Compressed});

// 4-tensor format with custom mode ordering
Format({Dense,Dense,Compressed,Compressed}, {1,0,3,2});

// block 2-tensor
Format({{Compressed,Compressed},{Dense(8),Dense(8)}});

// block 2-tensor with custom mode ordering
Format({{Compressed,Compressed},{Dense(8),Dense(8)}}{1,0,3,2});
```

getOrder

Returns the number of modes in the format.

Return Value

n. `int` number of modes in the format.

getModeFormats

Returns the storage types of the modes. The type of the mode stored in position i is specifed by element i of the returned vector.

Return Value

mode formats. `vector<ModeFormat>` the format's mode formats.

getModeFormatPacks

Get the storage formats of the modes, with modes that share the same physical storage grouped together.

Return Value

mode format packs. `vector<ModeFormatPack>` the format's mode format packs.

getModeOrdering

Return the ordering in which the modes are stored. The mode stored in position i is specifed by element i of the returned vector.

Return Value

mode ordering. `vector<int>` the format's mode ordering.

isBlocked

Return true if the format is a block format.

Return Value

block format. `bool`.

numBlockLevels

Return the number of nested block levels in the format.

Return Value

block levels. `int` number of blocks in the format.

getBlockSizes

Gets all the block fixed-sizes.

A zero size is given for the free block dimension. Each tensor dimension contains exactly one free block size, which is determined at tensor instantiation.

Return Value

block sizes. `vector<vector<int»` a list of dimension fixed sizes for each block in the format.

getDimensionFreeSizeBlock

For each dimension in the format, returns the block which contains the dimension's free size.

free block dimensions. `vector<int>`. Entry i corresponds to the block that contains a free size for dimension i. Each dimension has exactly one free size.

A.7 ScalarAccess

Template Parameters

`typename` CType. The Type of the component value.

Member Functions

`(Constructor)`	Construct ScalarAccess
`operator=`	Insert value to the tensor coordinate.
`operator CType`	Return the value stored at the tensor coordinate.

A.8 TensorBase

`TensorBase` is the super-class for all tensors. You can use it directly to avoid templates, or you can use the templated `Tensor<T>` that inherits from `TensorBase`.

Member Types

`const_iterator`	a random access iterator to const tensor values.
`Content`	stores tensor data and metadata.

Member Functions

`(Constructor)`	Construct TensorBase

Metadata Methods

`setName`	Set the name of the tensor
`getName`	Return the name of the tensor
`getOrder`	Return the order of the tensor (the number of modes)
`getDimension`	Return the dimension of a tensor mode
`getDimensions`	Return a vector with the dimension of each tensor mode
`getComponentType`	Return the type of the tensor components
`getFormat`	Return the format the tensor is packed into
`setStorage`	Set the tensor's storage
`getStorage`	Return the tensor's storage

Write Methods

`insert`	Insert a value into a tensor coordinate.
`setFromComponents`	Insert the Components stored in a container to the tensor.

Read Methods

`at`	Return the value stored at the specified coordinate.
`iterator`	Return an iterator object to iterate over the values of the tensor.
`iteratorTyped`	Return an iterator with typed coordinates.

Access Methods

`operator()`	Return an `Access` object to write index notation.

Compiler Methods

`pack`	Pack tensor into the given format.
`compile`	Compile the tensor expression.
`assemble`	Assemble the tensor storage, including index and value arrays.
`compute`	Compute the given expression and put the values in the tensor storage.
`evaluate`	Compile, assemble and compute as needed.
`needsPack`	Returns true if the Tensor needs to be packed.
`needsCompile`	Returns true if the Tensor needs to be compiled.
`needsAssemble`	Returns true if the Tensor needs to be assembled.
`needsCompute`	Returns true if the Tensor needs to be computed.

A.8.1 Member Types

Content

const_iterator

A.8.2 Member Functions

(Constructor)

Constructs a TensorBase, initializing its contents depending on the constructor version used:

empty constructor (default constructor) Constructs a scalar.

datatype constructor Constructs a scalar with specified data type.

format constructor Constructs a tensor with the given dimensions and format.

Parameters

name. `string` tensor name.

data type. `DataType` the type of the tensor values.

format. `Format` tensor format.

dimensions. `vector<int>` tensor dimensions.

setName

Set the name of the tensor. Tensor name is used during code generation, and during tensor print routines.

Parameters

name. `string` tensor name.

getName

Return the name of the tensor.

Return Value

name. `string` tensor name.

getOrder

Return tensor order.

Return Value

order. `int` tensor order.

getDimension

Return the dimension of a tensor mode.

Parameters

mode. `int` tensor mode to get the dimension from.

Return Value

dimension size. `int` tensor dimension size.

getDimensions

Return a vector with the dimension of each tensor mode.

Return Value

dimensions. `vector<int>` tensor dimensions.

getComponentType

Return the type of the tensor components.

Return Value

type. `DataType` the type of the values stored in the tensor.

getFormat

Return the tensor format describing the data structure of the tensor values.

Return Value

tensor format. `Format`

setStorage

Set the storage of the tensor. The `TensorStorage` object stores the tensor's data structure and values.

Parameters

storage. `TensorStorage` a storage object.

getStorage

Returns the storage for this tensor. Tensor values are stored according to the format of the tensor.

Return Value

storage. `TensorStorage` containing tensor values.

insert

Insert a value into the tensor.

Template Parameters

tensor type. `CType` the type of the tensor.

Parameters

coordinate. `Coordinate<Order>` the tensor coordinate to which to insert the value. The coordinate's order must match the tensor order.

value. `CType` the value to be inserted.

setFromComponents

Fill the tensor with the list of components defined by the iterator range (begin, end).

Template Parameters

iterator type. `Iterator` a `Component` iterator type.

Parameters

iterator. `Iterator` an iterator containing the input tensor components. The iterator must iterate over `Component` objects.

at

Return the value stored at a tensor coordinate.

Template Parameters

tensor type. `CType` the type of the tensor.

Parameters

coordinate. `Coordinate<Order>` the coordinate from which to read a tensor value. The coordinate Order must match the tensor order.

iterator

Returns an iterator object which can be used to iterate over the tensor values. The iterator is a typed wrapper for `TensorBase` that implements the `begin` and `end` iteration methods.

Template Parameters

tensor type. `CType` the type of the tensor.

Return Value

iterator. `iterator_wrapper<int,CType>`, an iterator over the values of `TensorBase`.

iteratorTyped

Returns an iterator object which can be used to iterate over the tensor values. The iterator is a typed wrapper for `TensorBase` that implements the `begin` and `end` iteration methods.

Template Parameters

tensor type. `CType` the type of the tensor.

coordinate type `T` the type used to store coordinate values.

Return Value

iterator. `iterator_wrapper<T,CType>`, an iterator over the values of `TensorBase`.

operator()

Create index expression that represents a tensor access, such as `A(i,j))`. Access expressions can also be assigned an expression, which happens when they occur on the left-hand-side of an assignment.

Template Parameters

index variables. `IndexVars...` a variadic template argument to have a variable number of index variables as input.

Parameters

index variables. `IndexVars...` a sequence of index variables to denote iteration over the tensor. The number of input index variables must be equal to the order of the tensor.

Return Value

access. `Access`, an access object which can be used to declare index expressions involving the tensor.

pack

Pack tensor into the given format. Compiles the values inserted to the tensor, and packs them into a data structure as described by the tensor's format.

compile

Compile the tensor expression. Takes an index expression assigned to the tensor, and generates code to assemble the result data structure of the operation, and to compute the operation.

assemble

Assemble the tensor storage, including index and value arrays. Generate the data structure in which the result of the tensor computation will be stored.

compute

ompute the given expression and put the values in the tensor storage.

A.9 Tensor

The templetized version of `TensorBase`, that knows its type and order at compile time. Type and order are used to improve the performance of different tensor operations.

Template Parameters

`size_t` Order. The order, or number of dimensions in the tensor.

`typename` CType. The Type of tensor values.

Member Functions

`(Constructor)`	Construct Tensor
`transpose`	Creates a tensor with the values of this tensor transposed
`begin`	Return the start pointer of an iterator through tensor
`beginTyped`	Return the typed start pointer of an iterator through tensor
`end`	Return the end pointer of an iterator through tensor
`endTyped`	Return the typed end pointer of an iterator through tensor
`operator()`	Return a `ScalarAccess` object to perform tensor value insertion and access.

A.9.1 Member Functions

(Constructor)

Constructs a Tensor, initializing its contents depending on the constructor version used:

empty constructor (default constructor) Constructs a scalar.

value constructor Constructs a scalar with specified value.

format constructor Constructs a tensor with the given dimensions and format.

TensorBase constructor Constructs a Tensor using the contents of a TensorBase.

Parameters

name. `string` tensor name.

format `Format` tensor format.

dimensions `vector<int>` tensor dimensions.

transpose

Creates a tensor corresponding to the transpose of this tensor.

begin

Returns an iterator pointing to the first element in the tensor.

Return Value

iterator. `const_iterator<int,Ctype>`

beginTyped

Returns an iterator pointing to the first element in the tensor.

Template Parameters

coordinate type. `T` The integral type used to store coordinate dimensions.

Return Value

iterator. `const_iterator<T,Ctype>`

end

Returns an iterator referring to the past-the-end element in the tensor container.

Return Value

iterator. `const_iterator<int,Ctype>`

endTyped

Returns a typed iterator referring to the past-the-end element in the tensor container.

coordinate type. `T` The integral type used to store coordinate dimensions.

Return Value

iterator. `const_iterator<T,Ctype>`

operator()

Create a `ScalarAccess` object to perform tensor value insertion or access. Scalar access objects provide a window to a tensor coordinate that enables tensor interaction with indexing syntax.

Template Parameters

coordinate integers. `Ints...` a variadic template argument to have a variable number of integers as input.

Parameters

indices. `Ints...` a sequence of integer indices representing the coordinate being accessed. The number of input integers must be equal to the order of the tensor.

Return Value

scalar access. `ScalarAccess`, an access object which can be used to interact with a tensor value at a specified coordinate.

Example Usage

```
Tensor<2, double> A({5,5}, Format::csr);

// insert a value with ScalarAccess
A(0,0) = 2.0;

// read a value with ScalarAccess
double x = A(0,0);
```

Appendix B

Lazy Execution Example

In this appendix we provide a lazy execution sample piece of code (figure B-1), along with its corresponding dependency diagram (figure B-2) and sequence diagram (fig B-3). This example complements the reading in chapter 5.

```
Format csr({Dense,Compressed});
Tensor<double> A({10,10}, csr);
Tensor<double> B({10,10}, csr);
Tensor<double> C({10,10}, csr);
Tensor<double> D({10,10}, csr);
Tensor<double> E({10,10}, csr);

B.setFromComponents(b_vals);
D.setFromComponents(d_vals);
E.setFromComponents(e_vals);

IndexVar i, j, k;
C(i,j) = D(i,j) + E(i,j);
A(i,j) = B(i,k) * C(k,j);

// Value insertion triggers
// D.pack();
// E.pack();
// C.compute();
E(0,0) = 2.0;

// Print statement triggers
// B.pack();
// A.compute();
std::cout << A << std::endl;
```

Figure B-1: Code sample commented with lazy execution compiler method logical placement.

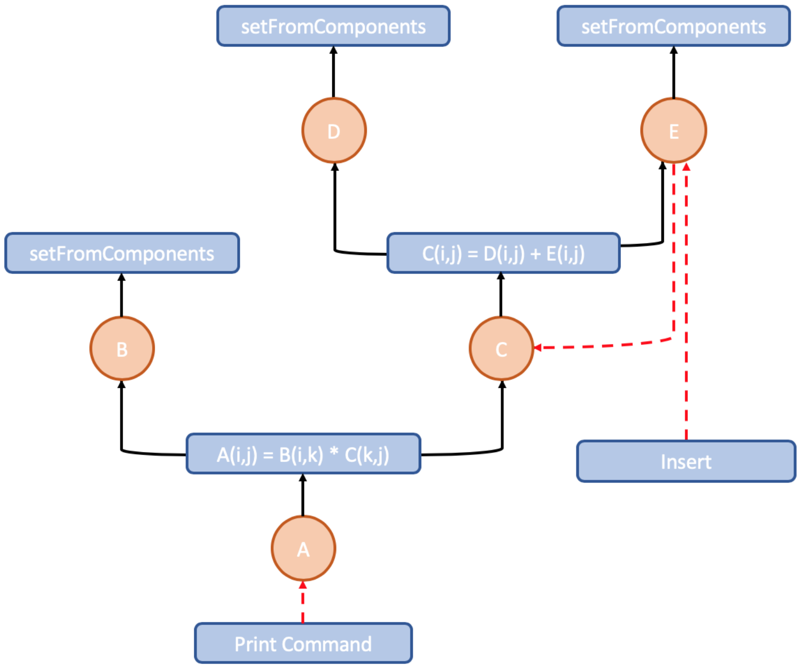

Figure B-2: Tensor dependency diagram for the computation in figure B-1. Black arrows indicate tensor and operation dependencies. Red dashed arrows indicate computation triggers.

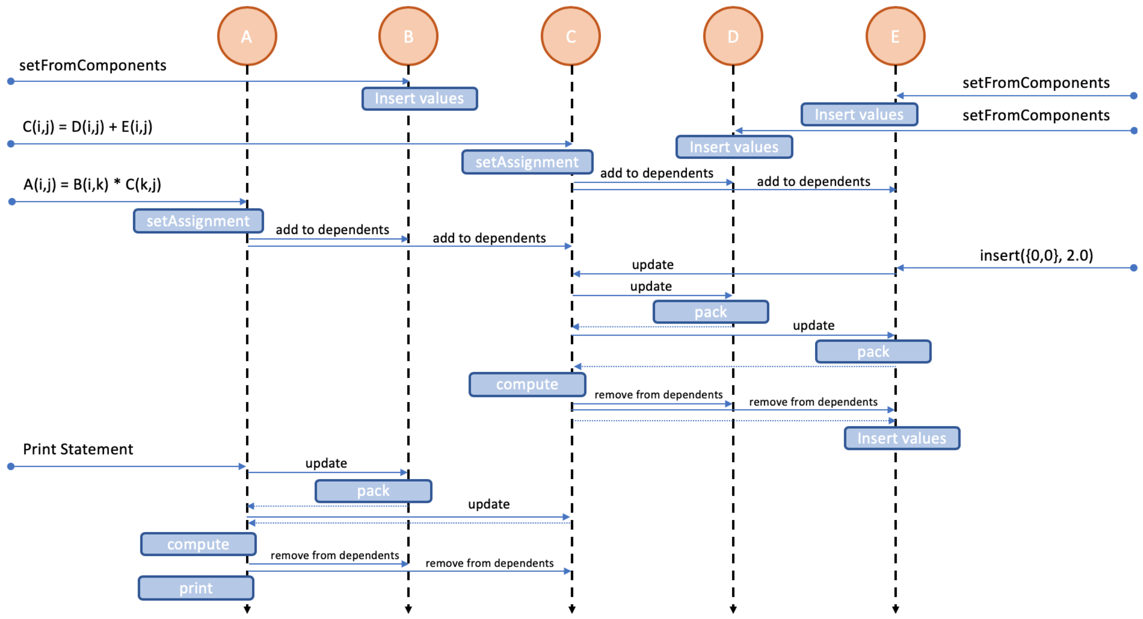

Figure B-3: Tensor sequence diagram for the computation in figure B-1.