Tensor Library

A templated library handling tensors (multidimensional arrays)

Introduction

This library allow users to handling and manage tensors (multidimensional arrays) by applying some functions and computations.
A tensor is specified by:
- the **type** it holds
- the **rank** (the number of dimensions)
- the **size of each dimension**

Tensor class provides different data-access functions to be applyied to the tensors:
- **direct access**
- **slicing**
- **windowing**
- **flattening**
- **iterators**
- **force copying** (on construction and assignment)

There are two different levels of compile-time knowledge of the tensor's shape:
- **no knowledge**: there is no compile time information about rank or dimensions of the tensor, only type
- **rank only**: the tensor has rank information and its type, but no dimensional information

Data is maintained in a consecutive area.

Template specification maintains an array of the strides and width of each dimension:
- **stride** -> distance to the next element of this index
- **width** -> number of entries for this index
Representation of this values is in *right-major order*.

The various types of tensors share data if compatible.

Template design

Since there are two compile-time knowledge, it is necessary to distinguish if a tensor belongs to one or the other.

Type only compile-time knowledge
Tensor knows only its type, parameterized on `T`, so its template is `template <typename T>` and the class declaration is `class tensor_t`.
User can create a new tensor with `tensor_t<int> t` for example and passing as argument a vector containing the size of each dimension desired like `t({5,4,3})`. So, the complete declaration of a new tensor is `tensor_t<int> t({5,4,3});`. Furthermore, user can create a new tensor by giving the dimensions (as a

vector) and a vector containing data of type `T`, in this way the data structure contains the specified data at the beginning.

Type and Rank compile-time knowledge

Tensor knows its type, parameterized on `T`, and its rank, so its template is `template <typename T, size_t R>` and the declaration is `class tensor`.
User can create a new tensor with `tensor<int,3> t` for example and passing as argument a vector containing the size of each dimension desired like `t({5,4,3})`.
So, the complete declaration of a new tensor is `tensor<int,3> t({5,4,3});`. Like in the previous case, user can construct a new tensor by giving the dimensions and the data with which initialize the data structure.

This compile-time knowledge is specialized for tensors with rank 1 because they need different implementation in the functions provided by the library. The template is `template <typename T>` and the declaration is `class tensor<T,1>`.
User can create a new tensor with rank 1 with `tensor<double,1> t` for example and passing as argument a `size_t` variable containing the size of the single dimensione like `t(5)`.
So, the complete declaration of a new tensor is `tensor<double,1> t(5)`. Like in the previous cases, user can construct a new tensor by giving the dimension and the data with which initialize the data structure.

Implementation

"TYPE only at compile-time"

DATA

Class tensor has private data-members that cannot be reached from outside:
- `data` -> is the data stored in a consecutive area using the `std::vector` parameterized on type `T`. It is a `std::shared_ptr` parameterized on type `std::vector<T>` so data is always shared among two or more tensors (after a tensor operation is applied)
- `start_ptr` -> is a pointer to `T*` and it is used to optimize the access to the tensor (without dereferencing the pointer `data`)
- `width` -> is a `std::vector<size_t>` containing the size of each dimension
- `stride` -> is a `std::vector<size_t>` containing the stride of each dimension

CONSTRUCTORS

There are several constructors that allow to initialize a tensor (its data-members) by providing some parameters.
The copy constructor uses the default copy contructor of all members resulting in a copy of strides and width and sharing of the data vector.

DIRECT ACCESS

User can directly access any entry of the tensor by providing all the indexes as an array.
To do this, overload of `operator ()` is used. Then, to access to a single entry of the tensor, each stride is multiplied for each corresponding index in the array passed as argument.

SLICING

User can fix one (or more) index to produce a lower rank tensor sharing the same data. There are two

functions to slice a tensor, one that accepts in input an index and a starting point to slice the tensor along that index at the specified point. The other one that accepts in input a vector of tuples containing several indexes and starting points to slice tensor along different indexes at the specified points for each index.
Both return a tensor of type `tensor_t<T>`.

WINDOWING
User can create sub-windows of any given tensor, just by changing the starting-point and the end-point of several indexes.
Windowing function accepts as argument an array contains tuples (or, to create a sub-window with only one restricted dimension, a single tuple) with several attributes (index, start, end).
Windowing function returns a tensor of type `tensor_t<T>`.

FLATTENING
User can flattening two or more indices into one. Flattening function accepts as argument an array containing the indexes to be flattened and returns a tensor of type `tensor_t<T>`.
Flattening is restricted to consecutive ranges of indexes that also need to be consecutive in memory, meaning that sliced and windowed tensors cannot be flattened because their data is likely to not be consecutive anymore.

ITERATORS
Iterators are *forward iterators* to the full content of the tensor or to the content along any one index (providing all the other indexes).
Iterators can be used to access the sequence of elements in the tensor in the direction that goes from its beginning towards its end.
Iterators are inside a namespace called `iterator_namespace` because they are general and could be used by all the tensors specified in the templates.

The namespace contains the two types (classes) of iterators:
- **full content iterator** -> maintains a reference to stride and width of the tensor whose called the iterator, an own pointer, the `start_ptr` of the tensor and a support vector (useful for the operations on the iterator). The template parameters are `T` for the type of values in the tensor, `index` for the type of index used by the tensor (in this case `std::array` or `std::vector`) and also the rank in order for the `class tensor<T,R>` to be friend of iterator.
- **one index iterator** -> maintains an own pointer and a `size_t s` variable that represent that every `s` position there is a value belongs to the index specified by user (the stride). It has a constructor to inizitialize the data-members. In the template, parameter `R` is added for the same reason of the full content iterator.

Since they are forward iterators, operators `*`, `=`, `==`, `!=`, `++` (prefix) and `++` (postfix) were overloaded to access to the content of the tensor.
Iterators are called by tensor classes with the function `begin` that returns an iterator (or index iterator) pointing to the initial position of the tensor, and the function `end` that returns an iterator (or index iterator) pointing to the position that would follow the last element in the tensor (so it does not point to any element).

All the two types of iterators are `friend` of all the tensor classes (the two compile-time knowledge).
So, tensors can access to the members of the iterators and use them to access to their content.

FORCE COPYING

`shared_ptr` allows to share data in slicing, flattening, windowing operations and in construction and assignment (just by using the default copy and move constructors).
There is also the possibility to force copying on construction and assignment with the `copy` function implemented in all the tensor classes. This function returns a new tensor (depends by the template) with the same data (using the default copy constructor of `std::vector`) and the relative `start_ptr` of the new tensor whose called the force-copy function.

"TYPE and RANK at compile-time"

This type of tensor is `friend` of `class tensor<T,R+1>` and of `class tensor_t<T>`.

DATA

Class tensor has private data-members that cannot be reached from outside:
- `data` -> is the data stored in a consecutive area using the `std::vector` parameterized on type `T`. It is a `std::shared_ptr` parameterized on type `std::vector<T>` so data is always shared among two or more tensors (after a tensor operation is applied)
- `start_ptr` -> is a pointer to `T*` and it is used to optimize the access to the tensor (without dereferencing the pointer `data`)
- `width` -> is a `std::array<size_t,R>` containing the size of each dimension
- `stride` -> is a `std::array<size_t,R>` containing the stride of each dimension

CONSTRUCTORS

There are several constructors that allow to initialize a tensor (its data-members) by providing some parameters.
The copy constructor uses the default copy contructor of all members resulting in a copy of strides and width and sharing of the data vector.

DIRECT ACCESS

User can directly access any entry of the tensor by providing all the indexes as an array.
To do this, overload of `operator ()` is used. Then, to access to a single entry of the tensor, each stride is multiplied for each index in the array passed as argument.

SLICING

User can fix one index to produce a lower rank tensor sharing the same data. Slice function accepts in input an index and a starting position from which perform slicing.
This function returns a tensor of lower rank `tensor<T,R-1>`.
Slicing of more than one index has to be done by calling slice function repeatedly because slicing modifies the rank and the method cannot know the resulting tensor rank at compile time.

WINDOWING

User can create sub-windows of any given tensor, just by changing the starting-point and the end-point of each index.
Windowing function accepts as argument an array contains tuples with several attributes (index, start, end).
Also, windowing could be performed just in one dimension by calling the dedicated function.
Windowing does not modify the rank and function returns a tensor of type `tensor<T,R>`.

FLATTENING

User can flattening two or more indices into one. Flattening function accepts as argument two indexes to be flattened and returns a tensor of type `tensor<T,R-1>`.
For efficiency reason, flattening is restricted to consecutive ranges of indexes.

ITERATORS

See the section *ITERATORS* in the implementaion with *Type only at compile-time*.

FORCE COPYING

See the section *FORCE COPYING* in the implementaion with *Type only at compile-time*.

Specialization for tensor with rank 1

Tensor with rank 1 is `friend` of `class tensor<T,2>`.

DATA

Class tensor has private data-members that cannot be reached from outside:
- `data` -> is the data stored in a consecutive area using the `std::vector` parameterized on type `T`. It is a `std::shared_ptr` parameterized on type `std::vector<T>` so data is always shared among two or more tensors (after a tensor operation is applied)
- `start_ptr` -> is a pointer to `T*` and it is used to optimize the access to the tensor (without dereferencing the pointer `data`)
- `width` -> is a `size_t` containing the size of the dimension
- `stride` -> is a `size_t` containing the stride of the dimension

CONSTRUCTORS

There are several constructors that allow to initialize a tensor (its data-members) by providing some parameters.
There are also copy and move constructors.

DIRECT ACCESS

User can directly access any entry of the tensor by providing the only index.
To do this, overload of `operator ()` is used. Then, to access to a single entry of the tensor, stride is multiplied for the index passed as argument.

SLICING

Slicing on a tensor with rank 1 calls the overloaded `operator ()` passing as argument the fixed index given in input to the slice functions.
Returns a `T&` type where `T` is the parameterized type of elements stored in the tensor.

WINDOWING

User can create sub-windows of any given tensor, just by changing the starting-point and the end-point of the index provided.
Windowing function accepts as argument two values (begin and end) and return a tensor of rank 1 with modified range of values.

FLATTENING

Tensor with rank 1 cannot be flattened.

ITERATORS

See the section *ITERATORS* in the implementaion with *Type only at compile-time* with the exception that the iterator for rank 1 tensor is only an index iterator.

FORCE COPYING

See the section *FORCE COPYING* in the implementaion with *Type only at compile-time*.

Compiling

Compile with: `g++ tensor.cpp -o tensor -std=c++14`.
Then execute with `./tensor`.