ADVANCED ALGORITHMS AND PROGRAMMING METHODS - 2 Project 2021

Daniele Barzazzi 863011, Andrea Cester 864008 and Enrico Siviero 865112

January 2021

1 Tensors

1.1 Definition

Tensors are geometric objects that describe linear relations between geometric vectors, scalars, and other tensors.
With respect to our field of interest, the multidimensional arrays, a tensor is just as a vector in an n-dimensional space. It is represented by a one-dimensional array of length n with respect to a given basis; any tensor with respect to a basis is represented by a multidimensional array. Tensors are specified by:

- the type they are holding;
- their rank, i.e., the number of dimensions (or indexes);
- the size of each dimension

1.2 Ranks

The term rank of a tensor extends the notion of the rank of a matrix in linear algebra, although the term is also often used to mean the order (or degree) of a tensor. The rank of a matrix is the minimum number of column vectors needed to span the range of the matrix. A matrix thus has rank one if it can be written as an outer product of two nonzero vectors:

	Notation	**Rank**
scalar	a	0
vector	a_i	1
tensor	A_{ij}	2
tensor	A_{ijk}	3

The above table gives the most common nomenclature associated to tensors of various rank.

1.3 Implementation

Implement a templated library handling tensors, i.e., multidimensional arrays. The tensor class must provide various data-access functions:

- **direct access**:the user must be able to directly access any entry of the tensor by providing all the indexes. I suggest the use of an operator ()
- **slicing**: the user can fix one (or more) index producing a lower rank tensor sharing the same data.

The tensors, once created, should be unmodifiable in their dimensions (while-modifiable in their content), but the library we provide two type of tensor, one ranked and one unranked:

- **Ranked**: the tensor has rank information in its type, but no dimensional information;
- **Unranked**: the tensor that knows only the type, we use this type of tensor to simplify the operation between tensor with different ranks;

1.4 Implementation notes

The data should be maintained in a consecutive area. The template specification must maintain an array of the strides and width of each dimension.

- **stride**: distance in the next element of this index;
- **width**: number of entries for this index.

for example a rank 3 tensor of size (3,4,5) represented in right-major order will have strides (20,5,1) and width (3,4,5).
Entry (i, j, k) will be at index $(20 \cdot i + 5 \cdot j + k \cdot 1)$ in the flat storage.

2 Tensor base structure

A tensor M is a multi-dimensional array composed of elements of the same type **T**. A tensor has a certain number of dimensions called **rank** r = n and each one has a specific size.
We defined how users will read and write data within the tensor. The technique adopted is right measure. We decided to choose this method only for practicality reasons. Indeed, in our opinion, users will better understand how to access data.

2.1 Mathematical definition

The tensor dimensions are defined as follow:

$$d = (d_1, d_2, ..., d_n)$$

Where d_i represents the size of i-th dimension. Although M is a multi-dimensional array, it is stored linearly, as a simple array A. So to access consecutive elements on an index that is not the least significant it is not necessary to consider consecutive elements on the array, but to skip a certain number of elements observing the so called strides (Ref.1.2).
Strides are calculated in this way:

$$s = (d_2 \cdot d_3 \cdot ... \cdot d_n, d_3 \cdot ... \cdot d_n, dn_{n-1} \cdot ... \cdot d_n, d_n, 1)$$

In a more synthetic way:

$$\begin{cases} 1 & \text{if } i = n \\ s_{i+1} \cdot d_{i+1} & \text{otherwise} \end{cases}$$

So given an element $x = (x_1, x_2, ..., x_n)$, it is stored on the linear array on the position:

$$P(x) = \sum_{i=1}^{n} x_i \cdot s_i$$

To access its content:

$$M(x) \; = \; A(P(x))$$

Then to iterate elements it is necessary to define **start** and **end pointer**. The start pointer:

$$\text{start ptr} \; = \; (0, 0, ..., 0)$$

is obviously the minimum value of each index, instead the end pointer:

$$\text{end ptr} \; = \; (d1, 0, ..., 0)$$

To observe, is that if r=0 the tensor is like a simple variable so that $d = \varnothing$, $s = \varnothing$ and **start ptr**=**end ptr**=0.
Consider also that start and end pointers are calculated in this way, on a new tensor. If users apply some operations on it, as we are going to say, it will be changed. For this reason all following operations will be referred to **start ptr** and **end ptr** and not to (0, 0, ..., 0) and (d1 , 0, ..., 0) because they may be different.

2.2 Slicing

The slicing operation consists in set one of the indexes as a constant. We do not want to create a new tensor duplicating the memory. Instead, we want to share the same linear array A of the original tensor. In this way we generate a tensor which has the rank decreased by one. This new tensor has to be read in a different way, starting from a different point and with a different ending point. Consider a tensor M with r = n, d = $(d_1, ..., d_n)$, s = $(s_1, ..., s_n)$ and as start and end pointers respectively, **start ptr** and **end ptr** . Suppose the operation of slicing consists on imposing $x_i = c$, the new tensor M' has:

- $r' = n - 1$
- $d' = (d_1, ..., d_{i-1}, d_{i+1}, ..., d_n)$
- $s' = (s_1, ..., s_{i-1}, s_{i+1}, ..., s_n)$
- $sp' = sp + c \cdot s_i$
- $ep' = sp' + s'_1 \cdot d'_1$

Slicing operation cannot be applied on tensors with $r = 0$.

2.3 Tensor Implementation

2.4 Fields

The Ranked tensor class, contains the following fields, which describe the object characteristics:

- **rank**: it is an int value which stores the tensor's rank;
- **stride**: it is a pointers vector of type T, which is used to share tensors memory;
- **indices**: it is an ints vector, which contains tensor's strides;
- **start ptr**: it is an int value which specify tensor's start pointer;

3 Einstein's Notation

Einstein's notation was introduced in 1916 to simplify the notation used for general relativity, expresses tensor operations by rendering implicit the summation operations. According to this notation, repeated indices in a tensorial expression are implicitly summed over, so the expression:

$$a_{ijk}b_j$$

Represents a rank 2 tensor c indexed by i and k such that:

$$c_{ik} = \sum_j a_{ijk} \cdot b_j$$

This specific notation allows also for simple contractions, like the following:

$$Tr(a) = a_{ii}$$

As well as additions (subtractions) and multiplications.

3.1 Implementation

The program can execute the following operation between two tensor:

- **Product**: product between two or more tensor
- **Trace**: on one tensor;

We create a specific tensor **labeled tensor** to compute the operations.

3.2 Test

In the main function "main.ccp" we have included several simple example.
In this section we will explain our test example. We have implemented a function called "setData" that fill all the Tensor with value. Direct access is implemented using the operator "()".

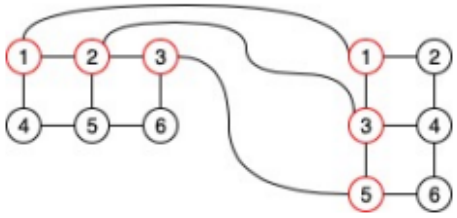
3.2.1 Product with one common label

To keep the calculations easy two bi-dimensional tensors, i.e. with dimensions *(n, m)*, have been multiplied.

	Size	**Label**
Tensor1	$(2, 3)$	(n,m)
Tensor2	$(3, 2)$	(m,o)

The first element is computed using the formula:

$$\text{res}_{0,0} = \sum_{m=0}^{2} a_{0,m} \cdot b_{m,0}$$



We create the Tensor using this code:

```
tensor::tensor<int,2> t(2,3);
tensor::tensor<int,2> t2(3,2);
t.setData();
t2.setData();
```

We create the index of the tensor and we set the right index for the two Tensor:

```
tensor::index n('n');
tensor::index m('m');
tensor::index o('o');

tensor::labeled_tensor<int> lt(t,n,m);
tensor::labeled_tensor<int> lt2(t2,m,o);
```

And perform the operation Product using the code:

```
tensor::labeled_tensor<int> ris=lt*lt2;
```

3.2.2 Product with more than one common label

The product test with multiple labels is carried out on two tensors.

	Size	**Label**
Tensor1	$(3, 2)$	(n,o)
Tensor2	$(2, 3, 2)$	(a,n,o)

Every element will be computed using:

$$res_o = \sum_{n=0}^{3} \sum_{o=0}^{2} a_{n,m,o} \cdot b_{m,n}$$

We create the Tensor using this code:

```
tensor::tensor<int,2> t2(3,2);
tensor::tensor<int,3> t3(2,3,2);
t2.setData();
t3.setData();
```

We create the index of the tensor and we set the right index for the two Tensor:

```
tensor::index n('n');
tensor::index o('o');
tensor::index a('a');

tensor::labeled_tensor<int> lt2(t2,n,o);
tensor::labeled_tensor<int> lt3(t3,a,n,o);
```

And perform the operation Product using the code:

```
tensor::labeled_tensor<int> ris=lt2*lt3;
```

3.2.3 Product with more tensor

It is possible to perform multiple product between Tensor, we perform a test using three Tensor:

	Size	**Label**
Tensor1	$(2, 3)$	(m,n)
Tensor2	$(3, 2)$	(n,o)
Tensor3	$(2, 3, 2)$	(n,a,o)

We create the three Tensor using this code:

```
tensor::tensor<int,2> t(2,3);
tensor::tensor<int,2> t2(3,2);
tensor::tensor<int,3> t3(2,3,2);
t.setData();
t2.setData();
t3.setData();
```

We create the index of the tensor and we set the right index for the three Tensor:

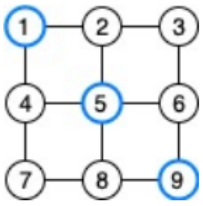
```
tensor::index n('n');
tensor::index m('m');
tensor::index o('o');
tensor::index a('a');

tensor::labeled_tensor<int> lt(t,m,n);
tensor::labeled_tensor<int> lt2(t2,n,o);
tensor::labeled_tensor<int> lt3(t3,n,a,o);
```

And perform the operation Product using the code:

```
tensor::labeled_tensor<int> ris=lt*lt2*lt3;
```

3.2.4 Trace

The last function is trace, so if we want to apply the trace to a tensor ($n = 2, n = 2$). The result will be the sum of the diagonal as reported in the following image:



We create the Tensor using this code:

```
tensor::tensor<int,2> t_dir(2,2);
t_dir.setData();
```

We create the index of the tensor and we set the right index for the three Tensor:

```
tensor::index n('n');
tensor::labeled_tensor<int> ltrace(t_dir,n,n);
```

And we call the function Trace to calculate the Trace:

```
int val=ltrace.trace();
```