Indexed Tensor Template Library

Indexed Tensor Template Library is driven by indices and based on BLAS. It works at compile time.

Examples:

M(I,I)=1; // Set diagonal elements of matrix M to 1

A(I,J)=B(I,K)\*C(K,J); // Matrix multiplication. A=BC

a(I)=b(I)\*c(I); // Element-wise vector multiplication

a(I).asum(M(I,J)); // Store sum of absolute values of each row of M in a

tr=M(I,I).sum(); // Trace of matrix

Indexed Tensor Template Library ( iTTL ) is designed for C++ developers who implements heavy optimizations including statistical models, data mining, big data analysis. Power of BLAS libraries gives a possibility to process high-dimension data. Power of C++ gives a possibility to create fast non-trivial algorithms. Power of iTTL makes code readable and decreases development time.

iTTL works at compile time. It chooses a proper BLAS subroutine and joins loops if possible. Suppose you have a matrix M of 1000000 rows and 2 columns. Assignment M=0 is translated by iTTL into two steps. In the first step the total size of M is calculated. The second step is a call to dcopy subroutine of BLAS.

Index concept

Purpose of Index

* linking dimensions of tensors in expressions
* choosing elements of dimension for processing
* optionally defining the order of element processing

|  |  |  |  |
| --- | --- | --- | --- |
| Index types | Affected elements | Optimization | Processing order |
| defaultIndex | All | Maximum | Undefined |
| segmentIndex | Specified number of elements successively from the specified position | Partial | Undefined |
| simpleIndex | All | No | Ascending |
| forwardIndex | Specified number of elements successively from the specified position | No | Ascending |
| reverseIndex | Specified number of elements successively from the specified position | No | Descending |
| container-based indices | Elements of container | No | According to iterator |

Valence

Valence is an integer property of the index. If valences of indices are equal then the corresponding dimensions are linked. They are iterated together during processing.

Normally each index has its own unique valence. It is assigned automatically when you declare an index. You can create a new index using the valence of the existing index using tpp::index\_creator<desired\_index\_type>::*create*(base\_index...). You can use operators +/- to create a new index with the valence of thebase index.

|  |  |  |
| --- | --- | --- |
| Index types | Result of operator-() | operator+(size\_t)  operator-(size\_t) |
| defaultIndex |  |  |
| segmentIndex | reverseIndex | Shifted elements |
| simpleIndex |  |  |
| forwardIndex | reverseIndex | Shifted elements |
| reverseIndex | forwardIndex | Shifted elements |
| container-based indices |  | Shifted elements |

Example:

DECLARE\_reverseIndex(R,5,0);

v(R+1)=v(R); // copy elements 0..4 of vector v to 1..5

You can apply ordinary numbers (size\_t) to dimensions also. Example: v5=v(5);

The result of application of indices to a tensor is a new tensor referring the subset of data of the initial tensor. No memory allocation is occurred. The newly created tensor can be re-indexed again if needed.

Example:

**auto** diag=M(I,I); // No memory allocation. diag is logically an alias of M(I,I). diag(K) is ok.

BLAS usage and limitations

BLAS is Basic Linear Algebra Subprograms. Some of BLAS libraries like OpenBLAS are extremely fast. Unfortunately, BLAS subroutines are not convenient, especially in C++. BLAS was initially implemented in Fortran. There are many implementations of BLAS. Most of them support Fortran-like interface and can be linked with iTTL. Since BLAS libraries are often multi-threaded, the order of processing of dimensions elements is not always defined. So, all the above mentioned index types with the defined processing order cannot be used with BLAS libraries.

Besides, the fortran integer type is not the same as size\_t. So, when your BLAS library is using 32-bit integer for matrix dimensions, your dimension sizes for BLAS are limited to 231-1. Please choose the proper BLAS\_INTEGER type in blas\_tmpl.h header. The default is int.

Matrices in BLAS should have one continuous dimension. So, if T is a 3d tensor, T(I,J,1) is not a matrix in terms of BLAS even if I and J are indices of defaultIndex type.

Always use indices of defaultIndex type when possible. When not possible, the segmentIndex type is preferable. It will allow to use BLAS efficiently.

Linking BLAS

* You should set the proper BLAS\_INTEGER type in blas\_tmpl.h
* You should comment BLAS\_NEEDBUNDERSCORE defined in blas\_tmpl.h if your BLAS implementation does not contain ‘\_’ at the beginning of subroutine names

Basic functionality

Basic functionality has no restrictions on index type. All index types can be used. defaultIndex is the fastest since it allows BLAS usage. The gem\_logic.cpp illustrates the logic of General Tensor Multiplication (gem) and compares performance in different cases.

* Creating a tensor.

Examples:

tpp::TENSOR<3> T3({3,4,5});

tpp::MATRIX<> M({5000,5000});

size\_t dimensions[2]={300,400};

tpp::TENSOR M2(dimensions);

tpp::MATRIX<> ({{0.0, -1.0}, {1.0, 0.0}});

* Accessing elements

Examples:

M(2,3)=5;

**double** d=M(2,3);

* Initializing data with initialization list

Example:

tpp::TENSOR<4> t({2,2,2,2});

t(I,1,J,K)={{{1,2},{3,4}},{{5,6},{7,8}}};

This possibility exists just for convenience. It is not fast. If depth of nested initialization lists does not correspond the dimension of the tensor, compilation fails. If there are extra initialization parameters, the outOfBounds exception is thrown. If there is no initialization value for some element of the tensor being initialized, zero value is used.

* Memory access control

Incorrect application of indices leads to exceptions. Verification of indices occurs before processing of expressions.

* Aliasing

Example:

**auto** diag=M(I,I);

* Aliasing with index order changing

Example:

**auto** MT=M.template order\_indices<first\_desired\_valence, second\_desired\_valence>();

Note: The desired valence list should be a permutation of an existing valences of the original tensor. This possibility may be useful in a development of libraries.

* Removing dimensions by valences

Example:

size\_t offsets[2]={offset\_for\_first\_valence, offset\_for\_second\_valence};

**auto** V=T3.remove\_valences<first\_valence\_to\_remove, second\_valence\_to\_remove>(offsets);

This possibility may be useful in a development of libraries.

* Re-dimension

Examples:

tpp::TENSOR<3> T;

T.redim(2,3,4);

* Re-shape

Examples:

tpp::VECTOR<> V({17});

**auto** M=V.reshape(3,5); // M is a matrix 3x5 of first 15 elements of V. No memory allocation.

* Direct data access

Example:

**double** \*M\_Data=M.data\_ptr();

Note: data\_ptr() member function is available only if data of underlying tensor is continuous

* Copying

Examples:

V(I)=M(I,I); // copy diagonal elements

V(I)=M(I,K); // for each row sum through columns and copy

M(I,J)=V(I); // for each row of M copy V(row\_number) to all columns of M.

* Multiplication

Examples:

sc=A(I).dot(B(I)); // scalar multiplication

sc2=A(I).dot(M(I,J)); // sumj (sumi(Ai\*Mij))

R(I).gem(A(I,J),V(J)); // matrix-vector multiplication

R(I).gem(A(I),B(I)); // element-wise multiplication

R(I,J).gem(A(I,K),B(J,K),2,3); // R = 3\*R + 2\*A\*BT

* Scaling

Examples:

M(I,2).scal(3); // multiply column #2 by 3;

M(I,J).scal(V(I)); // multiply each row of M by the corresponding element of V

* Division

Example:

M(I,J).div(V(J)); // divide each column of M by the corresponding element of V

* Adding

Y(I).axpy(X(I),3); // Y = Y + 3\*X

* Shifting

Example:

V(I).shift(1); // V = V + 1

* Sum of elements

Example:

tr=M(I,I).sum(); // trace of M

* Absolute sum of elements

a=V(I).asum(); // a=sum(abs(Vi));

V(I).asum(M(I,J)); // Vi=sumj(abs(Mij))

* Sign

V(I).sign(X(I)); // Vi=sing(Xi)

* Get shape of tensor

size\_t shape[2];

M.get\_shape(shape);

* Get size of tensor

size=M.size(); // size is a product of the previously shape elements

* Releasing data

Example:

A.free();

free() method releases data of tensor. If there is no more tensors referring to data, data frees.

* Check if tensor refers to data

Example:

A.is\_allocated();

* Allocating memory for a copy of tensor

Example:

auto M=T(I,0,J).empty\_like();

This creates a new 2d tensor (matrix) M. Memory is allocated for the new matrix but not initialized. The new matrix M is indexed by I,J.

* Simple expressions

Examples:

R(I,J)+=A(J,I)/3;

R(I,J)+=2\*A(I,K)\*B(K,J)-V(J);

Note: complicated expressions are not supported. All supported expressions never allocate temporary data implicitly. However some expressions which can work without memory allocation are not supported. For example D=A\*(B+C) is not supported. If you don’t need to keep B unchanged, use two steps: B+=C; D=A\*B. If both B and C should remain unchanged, for fast computation, allocate temporary tensor for (B+C) explicitly. If you need to avoid extra memory allocation and you need to keep B and C unchanged, use D=A\*B+A\*C.

Other functionality

There are a lot of useful subroutines in BLAS. Most of them are not implemented in iTTL yet. Most of BLAS subroutines have hard restrictions in terms of iTTL and thus their usage is limited.

* Linear solving

Examples:

M(I,J).gesv(V(I));

This calculates M-1\*V and stores the result in V. The matrix M is overwritten after execution.

Note: last dimensions of M and V should be continuous and it should be derivable from the indices applied.

M(I,J).gesv(V(K,J));

This calculates (M-1)T\*Vk and stores the result in Vk.  The matrix M is overwritten after execution.

Note: last dimensions of M and V should be continuous and it should be derivable from the indices applied.

**auto** lu=M(I,J).lu(); // create LU factorization object for M(I,J). Memory allocation occurs.

lu.solve(V(I));

This calculates M-1\*V and stores the result in V. Neither M nor lu are overwritten. If V is continuous the processing is faster. Elements of V may correspond to high dimension of some plain tensor, for example: V(J)=T(J,2,3). In this case the processing is not fast.

**auto** lu=M(I,J).lu(); // create LU factorization object for M(I,J). Memory allocation occurs.

lu.solve(V(J));

This calculates (M-1)T\*V and stores the result in V. Neither M nor lu are overwritten. If V is continuous the processing is faster.

**auto** lu=M(I,J).lu(); // create LU factorization object for M(I,J). Memory allocation occurs.

lu.solve(V(K,J));

This calculates (M-1)T\*Vk and stores the result in Vk. Neither M nor lu are overwritten. If J-valence of V is continuous the processing is faster. If V is a plain matrix the processing is even more fast.

**auto** lu=M(I,J).lu(); // create LU factorization object for M(I,J). Memory allocation occurs.

lu.solve(V(J,K));

This calculates (M-1)T\*Vk and stores the result in Vk. Neither M nor lu are overwritten. In this case J-valence of V is not continuous. The processing is not fast.

There is no hard restrictions exist for lu.solve() method.

Note: gesv and solve methods return BLAS\_INTEGER. If the result is not 0, an error happens. An error may occur if the initial matrix has zero determinant. The result of lu() method can be verified using lu.info() method.

C++ Usage Example

The file l1\_procs.h contains an example of optimization procedure template. The BR\_solve\_one procedure template optimizes

When iTTL is linked with OpenBLAS this optimization solves problem with in 2 minutes on Intel Core i7 with 6 cores.

Note: the procedure template temporary allocates data. If *ridge* is 0 the procedure template is just a robust regression.

Environment

The library was tested only in Linux/gcc environment. The library requires at least C++11.

Bug report

Please report bugs to tpptensor@mail.ru