Index notation with macros

The `@tensor` macro

The prefered way to specify (a sequence of) tensor operations is by using the `@tensor` macro, which accepts an index notation format, a.k.a. Einstein notation (and in particular, Einstein's summation convention).

This can most easily be explained using a simple example:

```
using TensorOperationsXD
α=randn()
A=randn(5,5,5,5,5,5)
B=randn(5,5,5)
C=randn(5,5,5)
D=zeros(5,5,5)
@tensor begin
    D[a,b,c] = A[a,e,f,c,f,g]*B[g,b,e] + α*C[c,a,b]
    E[a,b,c] := A[a,e,f,c,f,g]*B[g,b,e] + α*C[c,a,b]
end
```

In the second to last line, the result of the operation will be stored in the preallocated array `D`, whereas the last line uses a different assignment operator `:=` in order to define a new array `E` of the correct size. The contents of `D` and `E` will be equal.

Following Einstein's summation convention, the result is computed by first tracing/ contracting the 3rd and 5th index of array `A`. The resulting array will then be contracted with array `B` by contracting its 2nd index with the last index of `B` and its last index with the first index of `B`. The resulting array has three remaining indices, which correspond to the indices `a` and `c` of array `A` and index `b` of array `B` (in that order). To this, the array `C` (scaled with `α`) is added, where its first two indices will be permuted to fit with the order `a,c,b`. The result will then be stored in array `D`, which requires a second permutation to bring the indices in the requested order `a,b,c`.

In this example, the labels were specified by arbitrary letters or even longer names. Any valid variable name is valid as a label. Note though that these labels are never interpreted as existing Julia variables, but rather are converted into symbols by the `@tensor` macro. This means, in particular, that the specific tensor operations defined by the code inside the `@tensor` environment are completely specified at compile time. Alternatively, one can also choose to specify the labels using literal integer constants, such that also the following code specifies the same operation as above. Finally, it is also allowed to use primes (i.e. Julia's `adjoint` operator) to denote different indices, including using multiple subsequent primes.

```
@tensor D[å'',ß,c'] = A[å'',1,-3,c',-3,2]*B[2,ß,1] + α*C[c',å'',ß]
```

The index pattern is analyzed at compile time and expanded to a set of calls to the basic tensor operations, i.e. `TensorOperationsXD.add!`, `TensorOperationsXD.trace!` and `TensorOperationsXD.contract!`. Temporaries are created where necessary, but will by default be saved to a global cache, so that they can be reused upon a next iteration or next call to the function in which the `@tensor` call is used. When experimenting in the REPL where every tensor expression is only used a single time, it might be better to use `TensorOperationsXD.disable_cache`, though no real harm comes from using the cache (except higher memory usage). By default, the cache is allowed to take up to the minimum of either one gigabyte or 25% of the total machine memory, though this is fully configurable. We refer to the section on Cache for temporaries for further details.

Note that the `@tensor` specifier can be put in front of a full block of code, or even in front of a function definition, if index expressions are prevalent throughout this block. If a certain part of the code is nonetheless to be interpreted literally and should not be transformed by the `@tensor` macro, it can be annotated using `@notensor`, e.g.

```
@tensor function f(args...)
    some_tensor_expr
    some_more_tensor_exprs
    @notensor begin
        some_literal_indexing_expression
    end
    ...
end
```

Note that `@notensor` annotations are only needed for indexing expressions which need to be interpreted literally.

Contraction order and `@tensoropt` macro

A contraction of several tensors `A[a,b,c,d,e]*B[b,e,f,g]*C[c,f,i,j]*...` is generically evaluted as a sequence of pairwise contractions, using Julia's default left to right order, i.e. as `( (A[a,b,c,d,e] * B[b,e,f,g]) * C[c,f,i,j]) * ...)`. Explicit parenthesis can be used to modify this order. Alternatively, if one respects the so-called NCON style of specifying indices, i.e. positive integers for the contracted indices and negative indices for the open indices, the different factors will be reordered and so that the pairwise tensor contractions contract over indices with smaller integer label first. For example,

```
@tensor D[:] := A[-1,3,1,-2,2]*B[3,2,4,-5]*C[1,4,-4,-3]
```

will be evaluated as `(A[-1,3,1,-2,2]*C[1,4,-4,-3])*B[3,2,4,-5]`. Furthermore, in that case the indices of the output tensor (`D` in this case) do not need to be specified (using `[:]` instead), and will be chosen as `(-1,-2,-3,-4,-5)`. Any other index order for the output tensor is of course still possible by just explicitly specifying it.

A final way to enforce a specific order is by giving the `@tensor` macro a second argument of the form `order=(list of indices)`, e.g.

```
@tensor D[a,b,c,d] := A[a,e,c,f]*B[g,d,e]*C[g,f,b] order=(f,e,g)
```

This will now first perform the contraction corresponding to the index labeled `f`, i.e. the contraction between `A` and `C`. Then, the contraction corresponding to index labeled `e` will be performed, which is between `B` and the result of contracting `A` and `C`. If these objects share other contraction indices, in this case `g`, that contraction will be performed simultaneously, irrespective of its position in the list.

Furthermore, there is a `@tensoropt` macro which will optimize the contraction order to minimize the total number of multiplications (cost model might change or become configurable in the future). The optimal contraction order will be determined at compile time and will be hard coded in the expression resulting from the macro expansion. The cost/size of the different indices can be specified in various ways, and can be integers or some arbitrary polynomial of an abstract variable, e.g. `x`. In the latter case, the optimization assumes the assymptotic limit of large `x`.

```
@tensoropt D[a,b,c,d] := A[a,e,c,f]*B[g,d,e]*C[g,f,b]
# cost x for all indices (a,b,c,d,e,f)
@tensoropt (a,b,c,e) D[a,b,c,d] := A[a,e,c,f]*B[g,d,e]*C[g,f,b]
# cost x for indices a,b,c,e, other indices (d,f) have cost 1
@tensoropt !(a,b,c,e) D[a,b,c,d] := A[a,e,c,f]*B[g,d,e]*C[g,f,b]
# cost 1 for indices a,b,c,e, other indices (d,f) have cost x
@tensoropt (a=>x,b=>x^2,c=>2*x,e=>5) D[a,b,c,d] := A[a,e,c,f]*B[g,d,e]*C[g,f,b]
# cost as specified for listed indices, unlisted indices have cost 1 (any symbol for
```

Because of the compile time optimization process, the optimization cannot use run-time information such as the actual sizes of the tensors involved. If these sizes are fixed, they should be hardcoded by specifying the cost in one of the ways as above. The optimization algorithm was described in Physical Review E 90, 033315 (2014) and has a cost that scales exponentially in the number of tensors involved. For reasonably sized tensor network contractions with up to around 30 tensors, this should still be sufficiently fast (at most a few seconds) to be performed once at compile time, i.e. when the contraction is first invoked. Information of the optimization process can be obtained during compilation by using the alternative macro `@tensoropt_verbose`.

The optimal contraction tree as well as the associated cost can be obtained by

```
@optimalcontractiontree D[a,b,c,d] := A[a,e,c,f]*B[g,d,e]*C[g,f,b]
```

where the cost of the indices can be specified in the same various ways as for `@tensoropt`. In this case, no contraction is performed and the tensors involved do not need to exist.

Dynamical tensor network contractions with `ncon` and `@ncon`

Tensor network practicioners are probably more familiar with the network contractor function `ncon` to perform a tensor network contraction, as e.g. described in NCON. In particular, a graphical application TensorTrace was recently introduced to facilitate the generation of such `ncon` calls. TensorOperationsXD.jl now provides compatibility with this interface by also exposing an `ncon` function with the same basic syntax

```
ncon(list_of_tensor_objects, list_of_index_lists)
```

e.g. the example of above is equivalent to

```
@tensor D[:] := A[-1,3,1,-2,2]*B[3,2,4,-5]*C[1,4,-4,-3]
D ≈ ncon((A,B,C),([-1,3,1,-2,2], [3,2,4,-5], [1,4,-4,-3]))
```

where the lists of tensor objects and of index lists can be given as a vector or a tuple. The `ncon` function necessarily needs to analyze the contraction pattern at runtime, but this can be an advantage, in case where the contraction is determined by runtime information and thus not known at compile time. A downside from this, besides the fact that this can result in some overhead (though that is typical negligable for anything but very small tensor contractions), is that `ncon` is type-unstable, i.e. its return type cannot be inferred by the Julia compiler.

The full call syntax of the `ncon` method exposed by TensorOperationsXD.jl is

```
ncon(tensorlist, indexlist, [conjlist, sym]; order = ..., output = ...)
```

where the first two arguments are those of above. Let us first discuss the keyword arguments. The keyword argument `order` can be used to change the contraction order, i.e. by specifying which contraction indices need to be processed first, rather than the strictly increasing order `[1,2,...]`. The keyword argument `output` can be used to specify the order of the output indices, when it is different from the default `[-1, -2, ...]`.

The optional positional argument `conjlist` is a list of `Bool` variables that indicate whether the corresponding tensor needs to be conjugated in the contraction. So while

```
ncon([A,conj(B),C], [[-1,3,1,-2,2], [3,2,4,-5], [1,4,-4,-3]]) ≈
    ncon([A,B,C], [[-1,3,1,-2,2], [3,2,4,-5], [1,4,-4,-3]], [false, true, false])
```

the latter has the advantage that conjugating `B` is not an extra step (which creates an additional temporary), but is performed at the same time when it is contracted. The fourth positional argument `sym`, also optional, can be a constant unique symbol that enables `ncon` to hook into the global cache structure for storing and recycling temporaries. When it is not specified, the cache cannot be used in any deterministically meaningful way.

As an alternative solution to the optional positional arguments, there is also an `@ncon` macro. It is just a simple wrapper over an `ncon` call and thus does not analyze the indices at compile time, so that they can be fully dynamical. However, it will transform

```
@ncon([A, conj(B), C], indexlist; order = ..., output = ...)
```

into

```
ncon(Any[A, B, C], indexlist, [false, true, false], some_unique_sym, order = ..., ou
```

so as to get the advantages of cache for temporaries and just-in-time conjugation (pun intended) using the familiar looking `ncon` syntax.

As a proof of principle, let us study the following method for computing the environment to the `W` isometry in a MERA, as taken from Tensors.net, implemented in three different ways:

```
function IsoEnvW1(hamAB,hamBA,rhoBA,rhoAB,w,v,u)
    indList1 = Any[[7,8,-1,9],[4,3,-3,2],[7,5,4],[9,10,-2,11],[8,10,5,6],[1,11,2],[1
    indList2 = Any[[1,2,3,4],[10,7,-3,6],[-1,11,10],[3,4,-2,8],[1,2,11,9],[5,8,6],[5
    indList3 = Any[[5,7,3,1],[10,9,-3,8],[-1,11,10],[4,3,-2,2],[4,5,11,6],[1,2,8],[7
    indList4 = Any[[3,7,2,-1],[5,6,4,-3],[2,1,4],[3,1,5],[7,-2,6]]
    wEnv = ncon(Any[hamAB,rhoBA,conj(w),u,conj(u),v,conj(v)],indList1) +
                         ncon(Any[hamBA,rhoBA,conj(w),u,conj(u),v,conj(v)],indList2)
                         ncon(Any[hamAB,rhoBA,conj(w),u,conj(u),v,conj(v)],indList3)
                         ncon(Any[hamBA,rhoAB,v,conj(v),conj(w)],indList4);
    return wEnv
end

function IsoEnvW2(hamAB,hamBA,rhoBA,rhoAB,w,v,u)
    indList1 = Any[[7,8,-1,9],[4,3,-3,2],[7,5,4],[9,10,-2,11],[8,10,5,6],[1,11,2],[1
    indList2 = Any[[1,2,3,4],[10,7,-3,6],[-1,11,10],[3,4,-2,8],[1,2,11,9],[5,8,6],[5
    indList3 = Any[[5,7,3,1],[10,9,-3,8],[-1,11,10],[4,3,-2,2],[4,5,11,6],[1,2,8],[7
    indList4 = Any[[3,7,2,-1],[5,6,4,-3],[2,1,4],[3,1,5],[7,-2,6]]
    wEnv = @ncon(Any[hamAB,rhoBA,conj(w),u,conj(u),v,conj(v)],indList1) +
                         @ncon(Any[hamBA,rhoBA,conj(w),u,conj(u),v,conj(v)],indList2)
                         @ncon(Any[hamAB,rhoBA,conj(w),u,conj(u),v,conj(v)],indList3)
                         @ncon(Any[hamBA,rhoAB,v,conj(v),conj(w)],indList4);
    return wEnv
end

@tensor function IsoEnvW3(hamAB,hamBA,rhoBA,rhoAB,w,v,u)
    wEnv[-1,-2,-3] :=
        hamAB[7,8,-1,9]*rhoBA[4,3,-3,2]*conj(w[7,5,4])*u[9,10,-2,11]*conj(u[8,10,5,6

        hamBA[1,2,3,4]*rhoBA[10,7,-3,6]*conj(w[-1,11,10])*u[3,4,-2,8]*conj(u[1,2,11,
        hamBA[1,2,3,4]*rhoBA[10,7,-3,6]*conj(w[-1,11,10])*u[3,4,-2,8]*conj(u[1,2,11,
        hamAB[5,7,3,1]*rhoBA[10,9,-3,8]*conj(w[-1,11,10])*u[4,3,-2,2]*conj(u[4,5,11,
        hamBA[3,7,2,-1]*rhoAB[5,6,4,-3]*v[2,1,4]*conj(v[3,1,5])*conj(w[7,-2,6])
    return wEnv
    end
end
```

All indices appearing in this problem are of size `χ`. For tensors with `ComplexF64` eltype and values of `χ` in `2:2:32`, the reported minimal times using the `@belapsed` macro from BenchmarkTools.jl are given by

χ	IsoEnvW1: ncon	IsoEnvW2: @ncon	IsoEnvW3: @tensor
2	0.000154413	0.000348091	6.4897e-5
4	0.000208224	0.000400065	9.5601e-5
6	0.000558442	0.00076453	0.000354621
8	0.00138887	0.00150175	0.000982109
10	0.00506386	0.00365188	0.00288137
12	0.0126571	0.00959403	0.00818371
14	0.0292822	0.0216231	0.0184712
16	0.0531353	0.0410914	0.0359749
18	0.225333	0.0774705	0.0688475
20	0.43358	0.139873	0.129315
22	0.601685	0.243468	0.221995
24	0.902662	0.459746	0.427615
26	1.2379	0.66722	0.622856
28	1.84234	1.08766	1.0322
30	2.58548	1.53826	1.44854
32	3.85758	2.44087	2.34229

Throughout this range of `χ` values, method 3 that uses the `@tensor` macro is consistenly the fastest, both at small `χ`, where the type stability and the fact that the contraction pattern is analyzed at compile time matters, and at large `χ`, where the caching of temporaries matters. The direct `ncon` call has neither of those two features (unless the fourth positional argument is specified, which was not the case here). The `@ncon` solution provides a hook into the cache and thus is competitive with `@tensor` for large `χ`, where the cost is dominated by matrix multiplication and allocations. For small `x`, `@ncon` is also plagued by the runtime analysis of the contraction, but is even worse then `ncon`. For small `x`, the unavoidable type instabilities in `ncon` implementation seem to make the interaction with the cache hurtful rather than advantageous.

Multithreading and GPU evaluation of tensor contractions with `@cutensor`

Every index expression will be evaluated as a sequence of elementary tensor operations, i.e. permuted additions, partial traces and contractions, which are implemented for strided arrays as discussed in Package features. In particular, these implementations rely on Strided.jl, and we refer to this package for a full specification of which arrays are supported. As a rule of thumb, `Array`s from Julia base, as well as `view`s thereof if sliced with a combination of `Integer`s and `Range`s. Special types such as `Adjoint` and `Transpose` from Base are also supported. For permuted addition and partial traces, native Julia implementations are used which could benefit from multithreading if `JULIA_NUM_THREADS>1`. The binary contraction is performed by first permuting the two input tensors into a form such that the contraction becomes equivalent to one matrix multiplication on the whole data, followed by a final permutation to bring the indices of the output tensor into the desired order. This approach allows to use the highly efficient matrix multiplication (`gemm`) from BLAS, which is multithreaded by default. There is also a native contraction implementation that is used for e.g. arrays with an `eltype` that is not `<:LinearAlgebra.BlasFloat`. It performs the contraction directly without the additional permutations, but still in a cache-friendly and multithreaded way (again relying on `JULIA_NUM_THREADS>1`). This implementation can sometimes be faster even for `BlasFloat` types, and the use of BLAS can be disabled globally by calling `disable_blas()`. It is currently not possible to control the use of BLAS at the level of individual contractions.

Since TensorOperationsXD v2.0, the necessary implementations are also available for `CuArray` objects of the CUDA.jl library. This implementation is essentially a simple wrapper over the CUTENSOR library of NVidia, and as such has certain restrictions as a result thereof. Native Julia alternatives using CUDA.jl or KernelAbstractions.jl might be provided in the future.

Mixed operations between host arrays (e.g. `Array`) and device arrays (e.g. `CuArray`) will fail. However, if one wants to harness the computing power of the GPU to perform all tensor operations, there is a dedicated macro `@cutensor`. This will transfer all arrays to the GPU before performing the requested operations. If the output is an existing host array, the result will be copied back. If a new result array is created (i.e. using `:=`), it will remain on the GPU device and it is up to the user to transfer it back. Arrays are transfered to the GPU just before they are first used, and in a complicated tensor expression, this might have the benefit that transer of the later arrays overlaps with computation of earlier operations.

« TensorOperationsXD.jl Functions »

Powered by Documenter.jl and the Julia Programming Language.