```julia
module TensorOperationsXD

using TupleTools
using Strided
using Strided: AbstractStridedView, UnsafeStridedView
using LinearAlgebra
using LinearAlgebra: mul!, BLAS.BlasFloat
using LRUCache
using Requires

# Exports
#----------
# export macro API
export @tensor, @tensoropt, @tensoropt_verbose, @optimalcontractiontree, @notensor,
@ncon
export @cutensor

export enable_blas, disable_blas, enable_cache, disable_cache, clear_cache,
cachesize

# export function based API
export ncon
export tensorcopy, tensoradd, tensortrace, tensorcontract, tensorproduct, scalar
export tensorcopy!, tensoradd!, tensortrace!, tensorcontract!, tensorproduct!

# Convenient type alias
const IndexTuple{N} = NTuple{N,Int}

# An exception type for reporting errors in the index specificatino
struct IndexError{S<:AbstractString} <: Exception
    msg::S
end

# Index notation
#-----------------
@nospecialize
include("indexnotation/verifiers.jl")
include("indexnotation/analyzers.jl")
include("indexnotation/preprocessors.jl")
include("indexnotation/ncon.jl")
include("indexnotation/instantiators.jl")
include("indexnotation/postprocessors.jl")
include("indexnotation/parser.jl")
include("indexnotation/poly.jl")
include("indexnotation/optdata.jl")
include("indexnotation/optimaltree.jl")
include("indexnotation/tensormacros.jl")
include("indexnotation/indexordertree.jl")
@specialize

# Implementations
#------------------
include("implementation/indices.jl")
include("implementation/tensorcache.jl")
include("implementation/stridedarray.jl")
include("implementation/diagonal.jl")

# Functions
#------------
include("functions/simple.jl")
include("functions/ncon.jl")
include("functions/inplace.jl")

# Global package settings
#-------------------------
# A switch for enabling/disabling the use of BLAS for tensor contractions
const _use_blas = Ref(true)
use_blas() = _use_blas[]
function disable_blas()
    _use_blas[] = false
    return
end
function enable_blas()
    _use_blas[] = true
    return
end

# A cache for temporaries of tensor contractions
const _use_cache = Ref(true)
use_cache() = _use_cache[]

function default_cache_size()
    return min(1<<32, Int(Sys.total_memory())>>2)
end

# methods used for the cache: see implementation/tensorcache.jl for more info
function memsize end
function similar_from_indices end
function similarstructure_from_indices end

taskid() = convert(UInt, pointer_from_objref(current_task()))

const cache = LRU{Any, Any}(; by = memsize, maxsize = default_cache_size())

"""
    disable_cache()

Disable the cache for further use but does not clear its current contents.
Also see [`clear_cache()`](@ref)
"""
function disable_cache()
    _use_cache[] = false
    return
end

"""
    enable_cache(; maxsize::Int = ..., maxrelsize::Real = ...)

(Re)-enable the cache for further use; set the maximal size `maxsize` (as number of
bytes)
or relative size `maxrelsize`, as a fraction between 0 and 1, resulting in
`maxsize = floor(Int, maxrelsize * Sys.total_memory())`. Default value is `maxsize
= 2^30` bytes, which amounts to 1 gigabyte of memory.
"""
function enable_cache(; maxsize::Int = -1, maxrelsize::Real = 0.0)
    if maxsize == -1 && maxrelsize == 0.0
        maxsize = default_cache_size()
    elseif maxrelsize > 0
        maxsize = max(maxsize, floor(Int, maxrelsize*Sys.total_memory()))
    else
        @assert maxsize >= 0
    end
    _use_cache[] = true
    resize!(cache; maxsize = maxsize)
    return
end

"""
    clear_cache()

Clear the current contents of the cache.
"""
function clear_cache()
    empty!(cache)
    return
end

"""
    cachesize()

Return the current memory size (in bytes) of all the objects in the cache.
"""
cachesize() = cache.currentsize

# Initialization
#-----------------
function __init__()
    resize!(cache; maxsize = default_cache_size())

    @require CUDA="052768ef-5323-5732-b1bb-66c8b64840ba" begin
        if CUDA.functional() && CUDA.has_cutensor()
            const CuArray = CUDA.CuArray
            const CublasFloat = CUDA.CUBLAS.CublasFloat
            const CublasReal = CUDA.CUBLAS.CublasReal
            for s in (:handle, :CuDefaultStream, :CuTensorDescriptor, :cudaDataType,
                    :cutensorContractionDescriptor_t, :cutensorContractionFind_t,
                    :cutensorContractionPlan_t,
                    :CUTENSOR_OP_IDENTITY, :CUTENSOR_OP_CONJ, :CUTENSOR_OP_ADD,
                    :CUTENSOR_ALGO_DEFAULT,  :CUTENSOR_WORKSPACE_RECOMMENDED,
                    :cutensorPermutation, :cutensorElementwiseBinary,
:cutensorReduction,
                    :cutensorReductionGetWorkspace, :cutensorComputeType,
                    :cutensorGetAlignmentRequirement,
:cutensorInitContractionDescriptor,
                    :cutensorInitContractionFind, :cutensorContractionGetWorkspace,
                    :cutensorInitContractionPlan, :cutensorContraction)
                eval(:(const $s = CUDA.CUTENSOR.$s))
            end
            include("implementation/cuarray.jl")
            @nospecialize
            include("indexnotation/cutensormacros.jl")
            @specialize
        end
    end
end

# Some precompile statements
#---------------------------
function _precompile_()
    AVector = Vector{Any}
    for N = 1:8
        @assert precompile(Tuple{typeof(isperm), NTuple{N,Int}})
    end
    @assert precompile(Tuple{typeof(_intersect), Base.BitArray{1},
Base.BitArray{1}})
    @assert precompile(Tuple{typeof(_intersect), Base.BitSet, Base.BitSet})
    @assert precompile(Tuple{typeof(_intersect), UInt128, UInt128})
    @assert precompile(Tuple{typeof(_intersect), UInt32, UInt32})
    @assert precompile(Tuple{typeof(_intersect), UInt64, UInt64})
    @assert precompile(Tuple{typeof(_isemptyset), Base.BitArray{1}})
    @assert precompile(Tuple{typeof(_isemptyset), Base.BitSet})
    @assert precompile(Tuple{typeof(_isemptyset), UInt128})
    @assert precompile(Tuple{typeof(_isemptyset), UInt32})
    @assert precompile(Tuple{typeof(_isemptyset), UInt64})
    @assert precompile(Tuple{typeof(_ncontree!), AVector, Vector{Vector{Int64}}})
    @assert precompile(Tuple{typeof(_setdiff), Base.BitArray{1}, Base.BitArray{1}})
    @assert precompile(Tuple{typeof(_setdiff), Base.BitSet, Base.BitSet})
    @assert precompile(Tuple{typeof(_setdiff), UInt128, UInt128})
    @assert precompile(Tuple{typeof(_setdiff), UInt32, UInt32})
    @assert precompile(Tuple{typeof(_setdiff), UInt64, UInt64})
    @assert precompile(Tuple{typeof(_union), Base.BitArray{1}, Base.BitArray{1}})
    @assert precompile(Tuple{typeof(_union), Base.BitSet, Base.BitSet})
    @assert precompile(Tuple{typeof(_union), UInt128, UInt128})
    @assert precompile(Tuple{typeof(_union), UInt32, UInt32})
    @assert precompile(Tuple{typeof(_union), UInt64, UInt64})
    @assert precompile(Tuple{typeof(_nconmacro), Int, Int, Int})
    @assert precompile(Tuple{typeof(addcost), Power{:χ, Int64}, Power{:χ, Int64}})
    @assert precompile(Tuple{typeof(degree), Power{:x, Int64}})
    @assert precompile(Tuple{typeof(instantiate_contraction), Int, Int, Expr, Int,
AVector, AVector, Int})
    @assert precompile(Tuple{typeof(instantiate_generaltensor), Int, Int, Expr,
Int, AVector, AVector, Int})
    @assert precompile(Tuple{typeof(instantiate_linearcombination), Int, Int, Expr,
Int, AVector, AVector, Int})
    @assert precompile(Tuple{typeof(instantiate), Int, Int, Expr, Int, AVector,
AVector, Int})
    @assert precompile(Tuple{typeof(instantiate), Int, Int, Expr, Int, AVector,
AVector})
    @assert precompile(Tuple{typeof(instantiate), Expr, Bool, Expr, Int64, AVector,
AVector})
    @assert precompile(Tuple{typeof(instantiate), Nothing, Bool, Expr, Bool,
AVector, AVector, Bool})
    @assert precompile(Tuple{typeof(instantiate), Symbol, Bool, Expr, Bool,
AVector, AVector})
    @assert precompile(Tuple{typeof(instantiate_eltype), Expr})
    @assert precompile(Tuple{typeof(instantiate_scalar), Expr})
    @assert precompile(Tuple{typeof(instantiate_scalar), Float64})
    @assert precompile(Tuple{typeof(disable_blas)})
    @assert precompile(Tuple{typeof(disable_cache)})
    @assert precompile(Tuple{typeof(enable_blas)})
    @assert precompile(Tuple{typeof(enable_cache)})
    @assert precompile(Tuple{typeof(expandconj), Expr})
    @assert precompile(Tuple{typeof(expandconj), Symbol})
    @assert precompile(Tuple{typeof(getallindices), Expr})
    @assert precompile(Tuple{typeof(getallindices), Int})
    @assert precompile(Tuple{typeof(getallindices), Symbol})
    @assert precompile(Tuple{typeof(getindices), Symbol})
    @assert precompile(Tuple{typeof(getindices), Expr})
    @assert precompile(Tuple{typeof(gettensorobject), Int})
    @assert precompile(Tuple{typeof(getlhs), Expr})
    @assert precompile(Tuple{typeof(getrhs), Expr})
    @assert precompile(Tuple{typeof(isindex), Expr})
    @assert precompile(Tuple{typeof(isindex), Symbol})
    @assert precompile(Tuple{typeof(isindex), Int})
    @assert precompile(Tuple{typeof(hastraceindices), Expr})
    @assert precompile(Tuple{typeof(isassignment), Expr})
    @assert precompile(Tuple{typeof(isdefinition), Expr})
    @assert precompile(Tuple{typeof(isgeneraltensor), Expr})
    @assert precompile(Tuple{typeof(istensor), Expr})
    @assert precompile(Tuple{typeof(istensorexpr), Expr})
    @assert precompile(Tuple{typeof(isnconstyle), Array{AVector, 1}})
    @assert precompile(Tuple{typeof(isscalarexpr), Expr})
    @assert precompile(Tuple{typeof(isscalarexpr), Float64})
    @assert precompile(Tuple{typeof(isscalarexpr), LineNumberNode})
    @assert precompile(Tuple{typeof(isscalarexpr), Symbol})
    @assert precompile(Tuple{typeof(istensorexpr), Expr})
    @assert precompile(Tuple{typeof(isgeneraltensor), Expr})
    @assert precompile(Tuple{typeof(decomposetensor), Expr})
    @assert precompile(Tuple{typeof(normalizeindex), Int})
    @assert precompile(Tuple{typeof(normalizeindex), Symbol})
    @assert precompile(Tuple{typeof(normalizeindex), Expr})
    @assert precompile(Tuple{typeof(mulcost), Power{:χ, Int64}, Power{:χ, Int64}})
    @assert precompile(Tuple{typeof(ncontree), Vector{AVector}})
    @assert precompile(Tuple{typeof(optdata), Expr})
    @assert precompile(Tuple{typeof(optdata), Expr, Expr})
    @assert precompile(Tuple{typeof(optimaltree), Vector{AVector}, Base.Dict{Any,
Power{:χ, Int64}}})
    @assert precompile(Tuple{typeof(parsecost), Expr})
    @assert precompile(Tuple{typeof(parsecost), Int64})
    @assert precompile(Tuple{typeof(parsecost), Symbol})
    @assert precompile(Tuple{typeof(storeset), Type{Base.BitArray{1}}, AVector,
Int64})
    @assert precompile(Tuple{typeof(storeset), Type{Base.BitArray{1}}, Array{Int64,
1}, Int64})
```

```
    @assert precompile(Tuple{typeof(storeset), Type{Base.BitArray{1}},
Base.Set{Int64}, Int64})
    @assert precompile(Tuple{typeof(storeset), Type{Base.BitSet}, AVector, Int64})
    @assert precompile(Tuple{typeof(storeset), Type{Base.BitSet}, Array{Int64, 1},
Int64})
    @assert precompile(Tuple{typeof(storeset), Type{Base.BitSet}, Base.Set{Int64},
Int64})
    @assert precompile(Tuple{typeof(storeset), Type{UInt128}, AVector, Int64})
    @assert precompile(Tuple{typeof(storeset), Type{UInt128}, Array{Int64, 1},
Int64})
    @assert precompile(Tuple{typeof(storeset), Type{UInt128}, Base.Set{Int64},
Int64})
    @assert precompile(Tuple{typeof(storeset), Type{UInt32}, AVector, Int64})
    @assert precompile(Tuple{typeof(storeset), Type{UInt32}, Array{Int64, 1},
Int64})
    @assert precompile(Tuple{typeof(storeset), Type{UInt32}, Base.Set{Int64},
Int64})
    @assert precompile(Tuple{typeof(storeset), Type{UInt64}, AVector, Int64})
    @assert precompile(Tuple{typeof(storeset), Type{UInt64}, Array{Int64, 1},
Int64})
    @assert precompile(Tuple{typeof(storeset), Type{UInt64}, Base.Set{Int64},
Int64})
    @assert precompile(Tuple{typeof(tensorify), Expr})
    @assert precompile(Tuple{typeof(extracttensorobjects), Any})
    @assert precompile(Tuple{typeof(_flatten), Expr})
    # @assert precompile(Tuple{typeof(processcontractions), Any, Any, Any})
    @assert precompile(Tuple{typeof(defaultparser), Expr})
    @assert precompile(Tuple{typeof(defaultparser), Any})
    @assert precompile(Tuple{typeof(unique2), AVector})
    @assert precompile(Tuple{typeof(unique2), Array{Int64, 1}})
    @assert precompile(Tuple{typeof(use_blas)})
    @assert precompile(Tuple{typeof(use_cache)})
end
_precompile_()

end # module
```