Cache for temporaries

Contracting a sequence of tensors is provably most efficient (in terms of number of computations) by contracting them pairwise. However, this requires that several intermediate results need to be stored. In addition, if the contraction needs to be performed as a BLAS matrix multiplication (which is typically the fastest choice), every tensor typically needs an additional permuted copy that is compatible with the implementation of the contraction as multiplication. All these temporary arrays, which can be large, put a a lot of pressure on Julia's garbage collector, and the total time spent in the garbage collector can become significant.

That's why there is now a functionality to store intermediate results in a package wide cache, where they can be reused upon a next run, either a next iteration if the tensor contraction appears within the body of a loop, or on the next function call if it appears directly within a given function. This mechanism only works with the `@tensor` macro, not with the function-based interface.

The `@tensor` macro expands the given expression and immediately generates the code to create the necessary temporaries. It associates with each of them a random symbol (`gensym()`) and uses this as an identifier (together with the `Threads.threadid` of where it it is being evaluated) in a package wide global cache structure `TensorOperations.cache`, the implementation of which is a least-recently used cache dictionary from LRUCache.jl. Thereto, it estimates the size of each object added to the cache using `Base.summarysize` and discards objects once a certain memory limit is reached.

Enabling and disabling the cache

The use of the cache can be enabled or disabled using

`TensorOperations.enable_cache` — Function

```
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.

`TensorOperations.disable_cache` — Function

```
disable_cache()
```

Disable the cache for further use but does not clear its current contents. Also see `clear_cache()`

Furthermore, the current total size of all the objects stored in the cache can be obtained using the method `cachesize`, and `clear_cache` can be triggered to release all the objects currently stored in the cache, such that they can be removed by Julia's garbage collector.

`TensorOperations.cachesize` — Function

```
cachesize()
```

Return the current memory size (in bytes) of all the objects in the cache.

`TensorOperations.clear_cache` — Function

```
clear_cache()
```

Clear the current contents of the cache.

Cache and multithreading

The `LRU` cache currently is thread safe, but requires that temporary objects are allocated for every thread that is running `@tensor` expressions. If the same tensor contraction is evaluated by several threads (simultaneoulsy or not, this we cannot know), every thread needs to have its own set of temporary variables, so the 'same' temporary will be stored in the cache multiple times. As indicated above, this is accomplished by associating with every temporary a randoms symbol and the `Threads.threadid` of the thread where the expression is being evaluated. If you have `JULIA_NUM_THREADS>1` but always run tensor expressions on the main execution thread, no additional copies of the temporaries will be created.

« Functions Implementation »