An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU.

Dmitry I. Lyakh*

National Center for Computational Sciences, Oak Ridge National Laboratory†, Oak Ridge TN, 37831

Abstract

An efficient parallel tensor transpose algorithm is suggested for shared-memory computing units, namely, multicore CPU, Intel Xeon Phi, and NVidia GPU. The algorithm operates on dense tensors (multidimensional arrays) and is based on the optimization of cache utilization on x86 CPU and the use of shared memory on NVidia GPU. From the applied side, the ultimate goal is to minimize the overhead encountered in the transformation of tensor contractions into matrix multiplications in computer implementations of advanced methods of quantum many-body theory (e.g., in electronic structure theory and nuclear physics). A particular accent is made on higher-dimensional tensors that typically appear in the so-called multireference correlated methods of electronic structure theory. Depending on tensor dimensionality, the presented optimized algorithms can achieve an order of magnitude speedup on x86 CPUs and 2-3 times speedup on NVidia Tesla K20X GPU with respect to the naïve scattering algorithm (no memory access optimization). The tensor transpose routines developed in this work have been incorporated into a general-purpose tensor algebra library (TAL-SH).

Keywords: tensor transpose, array reordering, tensor contraction, many-body theory, electronic structure, multireference, NVidia GPU, Intel Xeon Phi.

I. Introduction

* Corresponding author's email: quant4me@gmail.com (permanent) or liakhdi@ornl.gov (work).

Efficient, massively parallel *tensor algebra* algorithms are instrumental for enabling the use of *correlated* (post-Hartree-Fock) quantum many-body methods [1] in high-impact simulations in quantum chemistry, solid-state and nuclear physics. In particular, being implemented in a general-purpose parallel library, those algorithms can help us increase both the size and the accuracy of quantum many-body simulations, thus improving our predictive power in the search for efficient photo-catalytic nano-systems, energy storage materials, etc. To date, in electronic structure theory (on a post-Hartree-Fock level), several software packages exist [2-11], which are capable of performing massively parallel tensor algebra operations. In particular, the NWChem [2-3] software suite, based on the Global Arrays (GA) library [12-14], exploits the automated code generation module TCE (Tensor Contraction Engine) [15-17] (together with its later extensions [38-41]) in order to implement optimized tensor contraction routines. The ACESIII software package [4-7], the first electronic-structure software equipped with an internal *domain-specific* language SIAL (Super-Instruction Assembly Language) [7], offers a developer a set of higher-level parallel abstractions in order to facilitate automatic parallelization of the evaluation of tensor expressions encountered in conventional quantum many-body methods (MBPT2, CCSD, EOM-CCSD [1]). Similarly to TCE in NWChem, an automated equation/code generation plug-in has also been developed for the SIAL language [18], although rather limited capabilities of SIAL prevented its extensive deployment (the next generation ACES code, ACES IV, is expected to have a more flexible implementation of the SIAL language). Among the new experimental attempts aiming at massive parallelism, we need to mention the CYCLOPS tensor algebra library [19] that exploits a cyclic distribution of tensors (multidimensional arrays) in pursuit of better load balancing and communication regularization. A conceptually close, massively parallel RRR framework, based on the block-cyclic distribution of tensors, was reported very recently [41]. Besides, let us mention the C++ tensor algebra code by Hanrath [20], a shared-memory tensor algebra library by Epifanovsky et. al. [37] (used in Q-Chem [21]), and a shared-memory (accelerator-enabled) tensor

algebra library TAL-SH developed by the present author.

Technically, regardless of the actual choice of a correlated many-body method, tensor contractions form the most computationally expensive part of its computer implementation. A *tensor contraction* is a reduction over pairs of indices shared by two tensors. Fortunately, the binary tensor contraction operation can be reduced to a *matrix-matrix multiplication* (extremely well parallelized on both x86 Intel/AMD CPU, Intel Xeon Phi, and NVidia GPU) via several *tensor transpose* (tensor dimension reordering) operations. For example, the following tensor contraction

$$Z_{a_1,a_2,a_3,i_1,i_2,i_3} += S_{a_1,a_2,c_1,c_2,i_1,i_2,i_3,k_1} H_{k_1,a_3,c_1,c_2} \quad (1)$$

can be evaluated as

1. Tensor transpose: $S_{a_1,a_2,c_1,c_2,i_1,i_2,i_3,k_1} \rightarrow S_{a_1,a_2,i_1,i_2,i_3,k_1,c_1,c_2}$;
2. Tensor transpose: $H_{k_1,a_3,c_1,c_2} \rightarrow H_{k_1,c_1,c_2,a_3}$;
3. Matrix multiplication: $Z_{(a_1,a_2,i_1,i_2,i_3);(a_3)} += S_{(a_1,a_2,i_1,i_2,i_3);(k_1,c_1,c_2)} H_{(k_1,c_1,c_2);(a_3)}$;
4. Tensor transpose: $Z_{a_1,a_2,i_1,i_2,i_3,a_3} \rightarrow Z_{a_1,a_2,a_3,i_1,i_2,i_3}$.

In step 3 above, multiple indices grouped within parentheses are combined into a single (composite) index, thus transforming the tensors into three matrices. Formally, on average, a *tensor transpose* involves an order of magnitude less operations than a matrix multiplication. Thus, one might expect steps 1, 2, 4 to take much less time than step 3. In practice, however, this is often not the case because the tensor transpose performs element-wise data scattering/gathering and cannot be executed efficiently on modern computer architectures if implemented straightforwardly (due to a high rate of cache misses). To improve efficiency, one can try using in-place reordering algorithms [22] (if memory bound) or employ optimized generic sorting algorithms, especially for tensors of lower dimensionality (say, up to four). However, there is a whole class of high-end correlated many-body theories [23-33] where the tensor dimensionality can grow really high, thus increasing the size of the sorted keys.

Consequently, the main objective of this work is to introduce an alternative, cache-efficient, parallel *tensor transpose algorithm* (as a part of the *tensor contraction* operation) capable of running on multicore compute nodes, possibly equipped with the Intel Xeon Phi and NVidia GPU accelerators. Such an algorithm will reduce the overhead of tensor element reordering during the transformation of tensor contractions into matrix-matrix multiplications.

In the context of tensor algebra for electronic structure theory, this problem has been addressed before by Hammond [34] and Hanrath [20]. In particular, Hammond employed an automatic code generator in order to sample the loop optimization space for tensor transposes of relatively low rank. He found [34] that the optimal solution delivers a significant performance gain over the implementation used in TCE [15-17] on a single CPU core.

Here, we extend the previous studies on tensor transpose algorithms, emphasizing the case of *higher-rank* tensors with small and irregular dimensions that has not been carefully investigated before. Apparently, for higher-rank tensors, an exhaustive sampling of the loop optimization space would be prohibitive in terms of the time required. Below, we present a formal analysis of the tensor transpose problem and implement a universal multithreaded (OpenMP based) subroutine for multicore CPU and Intel Xeon Phi as well as a universal CUDA kernel for NVidia GPU, thus covering all possible tensor transpose cases (arbitrary tensor rank, size, and permutation). The performance of the corresponding routines is subsequently analyzed on multicore CPUs (AMD Opteron 6274, AMD Opteron 6378, Intel Xeon E5-2670), Intel Xeon Phi 5110P, and NVidia Tesla K20X GPU. The routines have been put together into a general-purpose tensor algebra library under the working name TAL-SH (with three submodules, CP-TAL, PH-TAL, NV-TAL, for CPU, Xeon Phi, and NVidia GPU, respectively).

II. Algorithm

A general rank-n tensor $T(i_1,\ldots,i_n)$ is a map $\{i_1,\ldots,i_n\} \rightarrow (\mathbb{R}|\mathbb{C})$, where $i_1,\ldots,i_n$ are integers, and real/complex numbers are assumed to be represented in either single or double precision when storing

the corresponding array in computer memory. Formally, the *tensor transpose* (tensor dimension reordering) problem can be defined as

$$D(i_1,\ldots,i_n) = S\left(i_{j_1},\ldots,i_{j_n}\right) \quad \forall\{i_1,\ldots,i_n\}, \tag{2}$$

where n is the number of dimensions (tensor rank), each tensor index i_k independently runs over the integer range $1 \leq i_k \leq N_k$, and $\{j_1,\ldots,j_n\}$ is the required index permutation (dimension k of the input tensor S becomes dimension j_k in the output tensor D). For n=2 the (non-trivial) problem reduces to the matrix transpose whereas for n=1 it is always a trivial copy. If a general tensor transpose (Eq. 2) is implemented as a straightforward nest of loops over consecutive dimensions of the input/output tensor, one will end up scattering/gathering individual tensor elements to/from very different computer memory locations, which is not cache-friendly, unless the leading dimensions of both tensors coincide and the aggregate size of those leading dimensions is comparable to or exceeds the cache line length.

To improve cache utilization, we present a tensor transpose algorithm based on a generalization of cache-efficient matrix transpose algorithms (see Ref. [35], for example). The subsequent parallelization of the algorithm is done only for *shared-memory* systems (multicore CPU, Intel Xeon Phi, NVidia GPU). A distributed tensor transpose can easily be implemented on top of our developments if the tensors (arrays) are distributed among MPI processes in terms of dense *tensor blocks* obtained by segmentation of index ranges (tiling techniques used in Global Arrays [12-14] or ACES III [4-7] are an example). In such a case, each tensor block will stay on the same node. Only the internal element order in a tensor block will be modified according to a given permutation, that is, a tensor block will be transposed into another tensor block. In the current work, we restrict ourselves to *dense* tensor blocks, although *partially ordered* tensor block storage schemes (with partial order relations between tensor dimensions) are extremely important when dealing with higher-rank tensors encountered in *multireference* quantum many-body theory [23-33]. The implementation of the corresponding (partially

ordered) tensor transpose algorithms will be reported elsewhere. For dense tensor blocks, we can assume either a Fortran-like or a C-like array element storage, where the fastest changing index will be called the *leading* or *minor* (first in Fortran, last in C) whereas the most slowly changing one will be referred to as *senior* (last in Fortran, first in C). For the sake of clarity, we will adopt the Fortran convention here, thus having a tensor block (or, simply, a tensor) $T(i_1,...,i_n)$ stored in memory *contiguously* as a Fortran array.

As we have mentioned, the tensor transpose problem for rank-2 tensors is reduced to the well-studied *matrix transpose*, for which *cache-efficient* algorithms are widely available [35]. Those algorithms are based on matrix *tiling* such that the tiles can be transposed consecutively (or in parallel) by utilizing only a handful of cache lines for each tile. In this case, two-dimensionality of matrix tiles maps perfectly onto the two-dimensional structure of the cache, represented by the cache line *length* and the number of utilized cache lines (*width*). Provided that the length of the senior dimension of the matrix is neither a half nor a multiple of the L1 cache size, one can obtain close to the optimal performance on modern processors (with cache associativity higher or equal to two). Here and in the following discussion, we express the cache size in units equal to the size of a tensor element (4 or 8 bytes for real, 8 or 16 bytes for complex numbers, depending on the precision used).

An attempt to straightforwardly adopt the cache-efficient matrix transpose algorithm for tensors of dimensionality higher than two will quickly fail however, since maintaining a manageable size of tensor *sub-blocks* (higher-dimensional counterparts of matrix tiles) with uniformly extended dimensions will result in very small index segments, leading to more frequent cache misses. Hence, in order to keep utilizing the CPU cache efficiently, one has to explore a *non-uniform* partitioning of a tensor into tensor sub-blocks by grouping/segmenting its *specific* dimensions, depending on a given index permutation. For NVidia GPU, the equivalent of the efficient CPU cache utilization is the coalescence of global memory accesses per thread warp. The corresponding algorithms are formalized

below.

Let us filter out certain tensor transpose cases for which an efficient algorithm will be obvious. We will need some formal definitions:

Definition 1: A *multi-index* of length $m \leq n$ is a subset of m distinct tensor indices, $\{i_{l_1},...,i_{l_m}\}$, placed in a specific order, where n is the tensor rank (total number of tensor dimensions).

The position of each index in a multi-index is fixed. Any possible combination of index values in a multi-index $\{i_{l_1},...,i_{l_m}\}$ is a particular *value* of that multi-index, the union of which forms the multi-index *range*. The basic set operations (union, intersect, etc.) will apply to multi-indices, with additional index reordering whenever necessary (specified in each particular case).

Definition 2: The *volume* of multi-index $\{i_{l_1},...,i_{l_m}\}$ is defined as $Vol(\{i_{l_1},...,i_{l_m}\}) \equiv \prod_{k=1}^{m} R_{l_k}$, where $i_{l_k} \in range(i_{l_k})$ and R_{l_k} is the extent of dimension l_k.

First, any trivial index permutation, $\{1,...,n\}$, will result in a *direct copy* of the input tensor into the output tensor. This special case will serve as a *reference* since the performance is only limited by the memory bandwidth (no multi-index translation overhead, minimal cache misses, etc.). Another group of special (easy) cases can be formed by those tensor transposes for which the index permutation $\{j_1,...,j_n\}$ and the tensors obey the following condition:

$$\exists m: \quad (j_k = k, 1 \leqslant k \leqslant m) \text{ AND } (Vol(\{i_1,...,i_m\}) \geqslant L), \tag{3}$$

where L is the cache line length (or the warp size for GPU accelerators) measured in words (the size of the word is equal to the size of a tensor element). In this case, one will have $Vol(\{i_{m+1},...,i_n\})$ independent tasks, each copying a contiguous segment of $Vol(\{i_1,...,i_m\})$ consecutive elements from the input tensor into a contiguous segment of the output tensor. Since the segment length is larger or equal to the cache line length (or the warp size for GPU accelerators), this kind of tensor transposes is expected to be already efficient (low rate of cache misses). Furthermore, any transpose involving *small*

tensors, as compared to the L1 cache size, can be done without any special treatment.

Another important special case is when the first (leading) dimension of the input tensor is different from the first (leading) dimension of the output tensor and the extents of both are greater than or equal to the cache line length (the warp size for GPU accelerators), L: $(j_1 \neq 1)$ AND $(R_1 \geqslant L)$ AND $\left(R_{j_1} \geqslant L\right)$. In this case, for each value of the *external* multi-index, $\{i_1,\ldots,i_n\} \setminus \{i_1, i_{j_1}\}$, a cache-efficient matrix transpose corresponding to the indices i_1 and i_{j_1} needs to be performed:

$$D\left(i_1,\ldots,i_{j_1},\ldots\right) = S\left(i_{j_1},\ldots,i_1,\ldots\right). \tag{4}$$

Hence, there will be $Vol\left(\{i_1,\ldots,i_n\} \setminus \{i_1, i_{j_1}\}\right)$ independent tasks, each of which is a (cache-efficient) matrix transpose.

If neither of the above conditions applies, the tensor transpose will be treated generically, as formalized below (for both x86 CPU and NVidia GPU). Few more definitions:

> **Definition 3**: The *minor input multi-index* of length *m* is a multi-index consisting of the first *m* indices from the input tensor in their original order: $M_m^I \equiv \{i_{j_1},\ldots,i_{j_m}\}$. The *minor output multi-index* of length *k* is a multi-index consisting of the first *k* indices from the output tensor in their original order: $M_k^O \equiv \{i_1,\ldots,i_k\}$.
>
> **Definition 4**: The *minor multi-index* is the union of the minor input and minor output multi-indices: $M_{mk} \equiv M_m^I \cup M_k^O = \{i_{j_1},\ldots,i_{j_m}\} \cup \{i_1,\ldots,i_k\}$, where non-redundant indices from M_k^O are appended to M_m^I. The *external multi-index* comprises the remaining tensor indices: $\bar{M}_{mk} \equiv \{i_1,\ldots,i_n\} \setminus \left(\{i_{j_1},\ldots,i_{j_m}\} \cup \{i_1,\ldots,i_k\}\right)$ (in the order dictated by the output tensor here).

The above definitions allow us to formulate *heuristic* grouping/segmenting rules for tensor dimensions, thus inducing a split of the tensor into smaller blocks with a shape/volume appropriate for an efficient CPU cache utilization or for coalescing global memory accesses on an NVidia GPU. As before, *L* will designate either the cache line length (for x86 CPU) or the warp size (for NVidia GPU)

expressed in words (the word size is equal to the tensor element size). The typical value for L is 8/16 double/single precision real numbers for x86 CPU and 32 real numbers for NVidia GPU (warp size). Also, for NVidia GPU, B will designate the size (in words) of the shared-memory buffer allocated per CUDA thread block.

Given a tensor and a permutation, we need to optimize the following *parameters* in our algorithm:

(a) The length of the minor input multi-index, m⩽n;

(b) The length of the minor output multi-index, k⩽n;

(c) Segment length for the last index of the minor input multi-index, $s_I \leqslant R_{j_m}$;

(d) Segment length for the last index of the minor output multi-index, $s_O \leqslant R_k$.

The heuristic optimization criteria/constraints are:

(1) $Vol(M_m^I) \geqslant L$;

(2) $Vol(M_k^O) \geqslant L$;

(3) CPU: $Vol(M_{mk}) \leqslant pL^2 \leq L1\_cache\_size,\ \ 1 \leqslant p \leqslant (\approx 8)$;

NVidia GPU: $Vol(M_{mk}) \leqslant B \geqslant L^2$;

(4) The last index (dimension) of the minor input/output multi-index can be split into segments only if it is not an internal index in the output/input multi-index, respectively;

(5) Try keeping $Vol(M_m^I) \approx Vol(M_k^O)$.

Given an arbitrary tensor transpose, a special function returns the optimized parameters m, k, s_I, and s_O. They are not guaranteed to result in the best possible performance but it should be reasonable in most cases as illustrated in numerical results. The rationale behind the above optimization criteria/constraints lies in a proper reshaping of an arbitrary tensor transpose into the form similar to the special case of multiple matrix transposes (Eq. 4) that can utilize the cache efficiently. First, the minor input multi-index is packed into a single *super-index J*:

$$S\left(i_{j_1},\ldots,i_{j_m};i_{j_{m+1}},\ldots,i_{j_n}\right)\rightarrow S\left(J;i_{j_{m+1}},\ldots,i_{j_n}\right). \quad (5)$$

Similarly, the minor output multi-index is packed into a single *super-index I*:

$$D(i_1,\ldots,i_k;i_{k+1},\ldots,i_n)\rightarrow D(I;i_{k+1},\ldots,i_n). \quad (6)$$

If too long, subject to constraint (4), super-indices *J* and *I* can be segmented by dividing into segments the last constituent index, i_{j_m} and i_k, respectively, in order to fulfill the first three optimization criteria (an analog of tiling in matrix transposes). Having combined the tensor indices in such a way, the principal difference from the special case of multiple matrix transposes, depicted by Eq. 4, is that the super-indices *J* and *I* can generally *overlap* (they can share the same tensor indices). If they do not overlap, such that $Vol(M_{mk})=Vol(M_m^I)\cdot Vol(M_k^O)$, we can symbolically write

$$\forall X:\ \ D(I,J,X)=S(J,I,X)\ \ \forall I,J, \quad (7)$$

where the original index ordering has been relaxed to some extent and *X* is the super-index obtained by packing the indices of the external multi-index. Similarly to Eq. 4, these are just independent matrix transposes, each corresponding to a certain value of *X*. We can also invoke a geometrical interpretation using an *n*-dimensional Cartesian space (*n* is the tensor rank). In this space, a tensor occupies an *n*-dimensional volume equal to the number of elements in the tensor. Having specified the minor multi-index $M_{mk}(s_I,s_O)$, we can divide the tensor into smaller blocks extended over the dimensions involved in the minor multi-index (taking into account possible segmentation of those) while being flat (segment size of 1) over the dimensions contained in the external multi-index. The dimensionality of these smaller blocks is equal to the length of the minor multi-index, $d\leq(\mathrm{m}+\mathrm{k})$: The blocks can be line segments, plain rectangles, 3-dimensional parallelepipeds, and so on). Consequently, we will have a set of independent tasks, each task being a tensor transpose of a small *d*-dimensional block that can be executed in a cache-friendly manner.

Having defined the minor multi-indices, $M_m^I\equiv\{i_{j_1},\ldots,i_{j_m}\}$ (input), $M_k^O\equiv\{i_1,\ldots,i_k\}$ (output), and

their union $M_{mk} \equiv M_m^I \cup M_k^O$, together with the index segmentation (if any), each tensor block, thus obtained, is independently transposed by reorganizing the loops over the tensor indices. After having segmented (at most two) minor tensor indices (i_{j_m} and i_k), we impose the following order of loops in the multicore x86 CPU algorithm (AMD/Intel CPU, Intel Xeon Phi):

Algorithm 1

Loops over the segments of i_{j_m} and i_k (sets the current bounds for i_{j_m} and i_k ranges);

Loop nest over the indices of the external multi-index $\bar{M}_{mk} \equiv \{i_1,\ldots,i_n\} \setminus \big(\{i_{j_1},\ldots,i_{j_m}\} \cup \{i_1,\ldots,i_k\}\big)$;

Loop nest over consecutive indices of $M_{mk} \setminus M_m^I$ (the leftmost index is the innermost);

Loop nest over consecutive indices of M_m^I (the leftmost index is the innermost).

$$D(i_1,\ldots,i_n) = S\big(i_{j_1},\ldots,i_{j_n}\big)$$

The first two loop nests define the coarse grain parallelism by spawning a set of independent tasks, each corresponding to the transpose of a *d*-dimensional tensor sub-block accomplished by the last two loop nests (note that in the last two loop nests the indices i_{j_m} and i_k run within their current bounds due to possible segmentation). In our OpenMP implementation, the iteration space of the first two loop nests is distributed across the OpenMP threads, each thread obtaining a contiguous segment of the iteration space. Let us analyze the cache utilization in each independent task (individual transpose of a tensor sub-block). Since the innermost loop nest runs over the indices of M_m^I (the leading *m* indices of the input tensor), the number of cache misses during consecutive $Vol(M_m^I)$ reads is minimal (the fourth loop nest is optimal for reading). At the same time, the corresponding writes to array *D* can cause up to $Vol(M_m^I)$ cache misses, each write occurring to a distinct cache line (the first invocation of the fourth loop nest can be the least optimal for writing). Then, in the best case, all those freshly opened write

cache lines will be kept open simultaneously such that, after the next iteration in the third loop nest (it will shift the writing position in D by one), the following $Vol(M_m^I)$ writes will be done to the same (already opened) cache lines. In certain cases (appropriate tensor shapes/sizes and alignment), one can achieve the optimal performance, that is, the number of cache misses will be equal to that of the ideal case (direct copy of the input tensor into the output tensor without any permutation). Conversely, for certain tensor shapes/sizes, the $Vol(M_m^I)$ writes in the fourth loop nest may be mapped to the same cache line, leading to multiple cache line replacements (this can happen when the elementary increments $i_{j_1}{+}{+}$ shift the write position in D by a multiple of the cache size). In this case, the performance will be poor, similar to the naïve implementation of the tensor transpose via element-wise scattering. On average, the number of collisions (cache line replacements) in write caching is likely to be proportional to $Vol(M_m^I)$ (the total extent of the fourth loop nest). At the same time, in order to maintain efficient read/write caching when it is feasible, $Vol(M_m^I)$ and $Vol(M_k^O)$ cannot be less than L. In practice, to get a compromise, both volumes should be chosen to lie between L and $4L$, unless the tensor transpose falls into one of simpler categories. This explains the heuristic rules formulated above. As shown in numerical results, these rules can provide an order of magnitude speedup (on x86 CPU), as compared to the naïve element-wise scattering algorithm, especially for higher-rank tensors (the main accent of this work).

On NVidia GPU, an efficient tensor transpose algorithm is expected to coalesce global (read/write) memory accesses per thread warp, which turns out to be a harder task than optimizing the L1 cache utilization per thread on x86 CPU (including Intel Xeon Phi). Hereafter, the input/output tensors are assumed to reside in the GPU global memory. A proper tensor transpose algorithm should read/write L consecutive tensor elements per warp from/to global memory (L = warpSize). In order to achieve a high degree of coalescence for global memory accesses, our CUDA kernel utilizes an intermediate

buffer allocated in the GPU shared memory (the buffer size is designated as *B*). According to the heuristic rules formulated above, the lower bound for the buffer size is L^2. The upper bound is also limited due to the necessity of keeping the occupancy of streaming multiprocessors at a sufficient level (we should note however that this is not very important since the tensor transpose algorithm mostly depends on the efficiency of data movement). The GPU tensor transpose algorithm exploits the same coarse grain parallelism as the CPU algorithm. Namely, the first two loop nests of Algorithm 1 are distributed among CUDA thread blocks (they are independent). However, the inner part, executed within each thread block, differs as shown below:

Algorithm 2

Loops over the segments of i_{j_m} and i_k (sets the current bounds for i_{j_m} and i_k ranges);

Loop nest over the indices of the external multi-index $\bar{M}_{mk} \equiv \{i_1,\dots, i_n\} \setminus \big(\{i_{j_1},\dots, i_{j_m}\} \cup \{i_1,\dots, i_k\}\big)$;

Read $Vol(M_{mk})$ elements from the input tensor into the shared-memory buffer;

__syncthreads();

Write $Vol(M_{mk})$ elements from the shared-memory buffer to the output tensor;

__syncthreads();

As seen from Algorithm 2, each thread block reads $Vol(M_{mk})$ elements from the input tensor by dividing the range of the minor multi-index M_{mk} into segments of length $Vol(M_m^I)$. These segments are contiguous in memory occupied by the input tensor and each segment can be read in a coalesced way by one of the thread warps available (fine grain parallelism). This also explains the heuristic rule $Vol(M_m^I) \geqslant L$ (to achieve coalescence). In this way, we load a part of the input tensor defined by the minor multi-index (and index segmentation) into the shared-memory buffer. This part consists of

multiple contiguous segments of length $Vol(M_m^I)$. Then, we upload (by all thread warps in parallel) the content of the shared-memory buffer into the output tensor by segments of length $Vol(M_k^O)$, which are contiguous in memory occupied by the output tensor but scattered in the shared-memory buffer. If $Vol(M_k^O) \geqslant L$, the uploading process can benefit from coalescing global memory writes. In the writing process, the translation of addresses is done via special (small) tables stored in shared-memory as well. Since the GPU shared memory has a good extent of access randomization (no need of coalescence, but conflicts are still possible), one may expect such an algorithm to be more efficient than the naïve implementation via straightforward tensor element scattering. In practice, as one will see in numerical results, our GPU algorithm is only 2-3 times faster on average. The high level of concurrency inherent to NVidia GPUs seems to partially circumvent the scalar inefficiency of the scattering tensor transpose algorithm in this case.

III. Numerical results

In order to test performance of our tensor transpose algorithms, we ran a series of random tensor transposes with double precision real numbers. We varied the following parameters in our sample:

1. Tensor volume (number of elements): 37M, 77M, 141M;
2. Tensor dimensionality (rank): 2, 3, 4, 5, 6, 7, 8, 15;
3. Dimension spread (ratio between the largest and the smallest dimensions): [1, 5, 15].

While a specific tensor rank can always be imposed exactly, actual tensor sizes and dimension spreads were only approximately equal to the above listed values (as close as possible). For each combination of {tensor size; tensor rank; dimension spread}, we obtained seven random permutations (trivial permutations were excluded), performed the corresponding tensor transposes together with their inverses, and recorded the timings for our algorithm and the straightforward scattering algorithm, in which the tensor elements are consecutively read from the input tensor and written into their new

locations in the output tensor. Then, the average bandwidths were calculated for our algorithm and for the scattering algorithm. Both values were compared to the reference case of the *direct copy* of the input tensor into the output tensor (without any permutation) that was also averaged over the random sample. The total sample size is 504=3*8*3*7. For the sake of completeness, we will also provide the average bandwidths obtained for the *reverse* permutations exclusively, that is, for permutations like {2,1}, {3,2,1}, {4,3,2,1}, {5,4,3,2,1}, etc., which are believed to represent the worst case.

Our OpenMP implementation for multicore AMD processors was compiled with the GNU Fortran compiler v.4.8.2 (O3 optimization), the OpenMP implementation for multicore Intel Xeon and many-core Intel Xeon Phi processors was compiled with the Intel Fortran compiler v.2015.0.090 (O3 optimization), and the GPU CUDA C implementation was compiled with the CUDA 5.5 nvcc compiler (O3 optimization).

Figure 1 illustrates the average bandwidths obtained on a single node of the Cray XK7 HPC system Titan [36] at the National Center for Computational Sciences at the Oak Ridge National Laboratory. The node consists of two NUMA nodes equipped with AMD Opteron 6274 (Interlagos), totaling in 16 integer cores operating under the 2.2 GHz clock, and an NVidia Tesla K20X GPU accelerator (Kepler architecture). The performance of the tensor transpose algorithms on NVidia Tesla K20X will be shown later, for now focusing on AMD CPU only. As one can see, our efficient tensor transpose algorithm is on average 2 times slower than the direct copy when using 2 OpenMP threads, while only about 50% slower with 8 and 16 OpenMP threads. In all cases, it is significantly faster than the naïve scattering algorithm (≈7x speedup). Note that the memory channels get saturated already with 8 OpenMP threads. Figure 2 demonstrates the performance of the tensor transpose algorithms for reverse permutations exclusively. It is only slightly worse than the random permutation case. Figure 3 provides a more detailed analysis where the tensor transpose bandwidths are provided for each tensor rank separately (the size of the sample is 63 for each tensor rank), using 16 OpenMP threads. As one can see,

the highest bandwidth is achieved for tensor ranks of 3 to 6.

Figures 4, 5, 6 show the results obtained on a single node of the Linux Cluster at the HPC center of the University of Florida equipped with AMD Opteron 6378 processors (2.4 GHz), totaling in 64 cores per node. Although the performance of our efficient tensor transpose algorithm, as compared to the direct copy, is even better than for AMD 6274, the inefficiency of the scattering algorithm is less pronounced in the case of random permutations. Yet, for reverse permutations, our algorithm still delivers a significant speedup with 64 OpenMP threads. And again, the highest bandwidth is achieved for tensor ranks of 3 to 6, as one can see in Figure 6.

Figure 7, 8, 9 show the average bandwidths obtained on a single node of the Beacon computer cluster located at the National Institute for Computational Sciences at the Oak Ridge National Laboratory. Beacon is a heterogeneous machine equipped with Intel Xeon E5-2670 (2.6 GHz) CPU and Intel Xeon Phi 5110P MIC accelerators. Focusing on CPU only, the superiority of our algorithm over the naïve scattering algorithm is obvious, although we are quite behind the direct copy case on lower numbers of cores (2, 4, 8). According to Figure 9, our algorithm significantly outperforms the naïve scattering algorithm for higher tensor ranks.

Finally, Figures 10-13 show the results obtained on accelerators (Intel Xeon Phi 5110P with 236 OpenMP threads and NVidia Tesla K20X with 256 threads per CUDA block). Clearly, our efficient algorithm favors Intel Xeon Phi, delivering up to 30x speedup over the scattering algorithm for some cases, as can be seen from Figure 12 (mostly due to higher L1 cache associativity). It is also only about 2.5 times slower than the direct copy in the case of random permutations, as compared to about 5.5 times on NVidia Tesla K20X. For reverse permutations, the performance on Intel Xeon Phi and NVidia Tesla K20X is similar. As we already mentioned in the previous section, the high level of concurrency pertinent to NVidia GPU alleviates the scalar inefficiency of the scattering algorithm to a big extent, thus diminishing the superiority of our efficient algorithm. On average, we only observe about 2x-3x

speedup on NVidia Tesla K20X. Also, the use of the intermediate shared memory buffer on NVidia GPU introduces a noticeable overhead, thus putting the performance of our algorithm well behind the direct copy. Besides, contrary to Intel Xeon Phi and AMD/Intel CPU, our GPU tensor transpose algorithm favors lower tensor ranks, as can be seen from Figure 13.

IV. Conclusions

Based on the results obtained on our random samples, the presented (efficient) tensor transpose algorithms show a significant speedup with respect to the naïve scattering algorithm, especially for multicore CPU and many-core Intel MIC devices. It turned out that the optimization of the L1 cache utilization per CPU core is easier than achieving a coalescence of global memory accesses on NVidia GPU (for the tensor transpose problem). Interestingly, the high level of concurrency exposed by NVidia GPU alleviates the scalar inefficiency of the naïve scattering algorithm, thus diminishing the superiority of our efficient GPU algorithm. Nevertheless, we still achieve a steady 2x-3x speedup over the scattering algorithm on Kepler architecture. Furthermore, while the bandwidths obtained on multicore CPUs and NVidia GPU in each specific tensor transpose case had little variability, the performance on Intel Xeon Phi could vary noticeably, depending on the previous operation (we took special precautions in order to produce meaningful numbers).

Overall, the suggested tensor transpose algorithm shows a significant potential for improving the performance of the tensor contraction operation, the cornerstone of any efficient computer implementation of correlated quantum many-body methods.

V. Acknowledgements

The initial part of this work (open-source multi-CPU OpenMP implementation) was started during author's postdoctoral work at the Quantum Theory Project at the University of Florida (group of Prof. R. J. Bartlett) with the financial support from the Air Force Office for Scientific Research. The author is thankful for that opportunity. The computational resources of the HPC Center of the University of

Florida are also appreciated. The rest of the work was accomplished in the Scientific Computing group (led by Dr. T. Straatsma) at the National Center for Computational Sciences at the Oak Ridge National Laboratory. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. A part of the presented results was obtained using the computational resources of the Beacon project, supported by the National Science Foundation under grant No. 1137097 and by the University of Tennessee: Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or the University of Tennessee.

References

1. I. Shavitt, R. J. Bartlett, Many-Body Methods in Chemistry and Physics, Cambridge University Press, Cambridge, 2009.
2. http://www.nwchem-sw.org/index.php/Main_Page
3. M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, W.A. de Jong. Comput. Phys. Commun. 181, 1477 (2010).
4. http://www.qtp.ufl.edu/aces
5. V. Lotrich, N. Flocke, M. Ponton, A. Yau, A. Perera, E. Deumens, R. J. Bartlett, J. Chem. Phys, 128, 194104 (2008).
6. B. A. Sanders, R. J. Bartlett, E. Deumens, V. Lotrich, M. Ponton. Proceedings of the ACM/IEEE SC2010 Conference, Nov. 2010, New Orleans LA, USA.
7. E. Deumens, V. F. Lotrich, A. Perera, M. J. Ponton, B. A. Sanders, R. J. Bartlett. WIREs Comput. Mol. Sci. 1, 895 (2011).
8. http://daltonprogram.org
9. http://www.psicode.org

10. http://www.mpqc.org

11. http://www.msg.ameslab.gov/gamess/gamess.html

12. http://hpc.pnl.gov/globalarrays

13. J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Apra. Int. J. High Perform. C. 20, 203 (2006).

14. H. J. J. van Dam, W. A. de Jong, E. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev. WIREs Comput. Mol. Sci. 1, 888 (2011).

15. http://www.nwchem-sw.org/index.php/TCE

16. S. Hirata, J. Phys. Chem. A 107, 9887 (2003).

17. S. Hirata, Theor. Chem. Acc. 116, 2 (2006).

18. D. I. Lyakh, R. J. Bartlett. 50th Sanibel Symposium, St. Simon's Island, GA, USA, Feb 24 – Mar 2, 2010. Book of abstracts.

19. http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-210.html

20. M. Hanrath, A. Engels-Putzka. J. Chem. Phys. 133, 064108 (2010).

21. http://www.q-chem.com/

22. C. H.Q. Ding. IEEE Transactions on Parallel and Distributed Systems 12, 306 (2001).

23. P. Piecuch, N. Oliphant, L. Adamowicz. J. Chem. Phys. 99, 1875 (1993).

24. L. Adamowicz, J.-P. Malrieu, V. V. Ivanov. J. Chem. Phys. 112, 10075 (2000).

25. D. I. Lyakh, V. V. Ivanov, L. Adamowicz. J. Chem. Phys. 122, 024108 (2005).

26. V. V. Ivanov, L. Adamowicz, D. I. Lyakh. Collect. Czech. Chem. Commun. 70, 1017 (2005).

27. V. V. Ivanov, D. I. Lyakh, L. Adamowicz. Phys. Chem. Chem. Phys. 11, 2355 (2009).

28. M. Hanrath. J. Chem. Phys. 123, 084102 (2005).

29. A. Engels-Putzka, M. Hanrath. J. Chem. Phys. 134, 124106 (2011).

30. M. Kallay, P. G. Szalay, P. R. Surjan. J. Chem. Phys. 117, 980 (2002).

31. H.-S. Hu, K. Kowalski. J. Chem. Theory Comput. 9, 4761 (2013).

32. D. I. Lyakh, M. Musial, V. F. Lotrich, R. J. Bartlett. Chem. Rev. 112, 182 (2012).

33. D. I. Lyakh, R. J. Bartlett. J. Chem. Phys. 133, 244112 (2010).

34. This information was deduced from the abstract of the PhD thesis of Dr. Jeff Hammond "Coupled-cluster response theory: Parallel algorithms and novel applications" and another unlabeled paper (found via GOOGLE) authored by him.

35. M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran. Proceedings of the 40th Annual Symposium on Foundations of Computer Science, Oct 17-19, 1999, New York City, NY. P. 285. DOI: 10.1109/SFFCS.1999.814600.

36. https://www.olcf.ornl.gov/titan

37. E. Epifanovsky, M. Wormit, T. Kus, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, A. I. Krylov. J. Comput. Chem. 34, 2293 (2013).

38. A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, A. Sibiryakov. Mol. Phys. 104, 211 (2006).

39. A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan. J. Phys. Chem. A 113, 12715 (2009).

40. P.-W. Lai, K. Stock, S. Rajbhandari, S. Krishnamoorthy, P. Sadayappan. Proceedings of SC'13, article #13. DOI: 10.1145/2503210.2503290.

41. S. Rajbhandari, A. Nikam, P.-W. Lai, K. Stock, S. Krishnamoorthy, P. Sadayappan. Proceedings of SC'14, P. 375, DOI: 10.1109/SC.2014.36.

Figure captions

1. Tensor transpose bandwidths obtained on AMD 6274 by the efficient and scattering algorithms, compared to the reference case of direct copy. Random sample consists of arbitrary permutations, excluding trivial ones.
2. Tensor transpose bandwidths obtained on AMD 6274 by the efficient and scattering algorithms, compared to the reference case of direct copy. Random sample consists of reverse permutations only (believed to be the worst case).
3. Tensor transpose bandwidths obtained on AMD 6274 (16 cores) by the efficient and scattering algorithms, compared to the reference case of direct copy: Dependence on the tensor rank. Random sample consists of arbitrary permutations, excluding trivial ones.
4. Tensor transpose bandwidths obtained on AMD 6378 by the efficient and scattering algorithms, compared to the reference case of direct copy. Random sample consists of arbitrary permutations, excluding trivial ones.
5. Tensor transpose bandwidths obtained on AMD 6378 by the efficient and scattering algorithms, compared to the reference case of direct copy. Random sample consists of reverse permutations only (believed to be the worst case).
6. Tensor transpose bandwidths obtained on AMD 6378 (16 cores) by the efficient and scattering algorithms, compared to the reference case of direct copy: Dependence on the tensor rank. Random sample consists of arbitrary permutations, excluding trivial ones.
7. Tensor transpose bandwidths obtained on Intel Xeon E5-2670 by the efficient and scattering algorithms, compared to the reference case of direct copy. Random sample consists of arbitrary permutations, excluding trivial ones.
8. Tensor transpose bandwidths obtained on Intel Xeon E5-2670 by the efficient and scattering algorithms, compared to the reference case of direct copy. Random sample consists of reverse

permutations only (believed to be the worst case).

9. Tensor transpose bandwidths obtained on Intel Xeon E5-2670 (16 cores) by the efficient and scattering algorithms, compared to the reference case of direct copy: Dependence on the tensor rank. Random sample consists of arbitrary permutations, excluding trivial ones.
10. Tensor transpose bandwidths obtained on Intel Xeon Phi 5110P and NVidia Tesla K20X by the efficient and scattering algorithms, compared to the reference case of direct copy. Random sample consists of arbitrary permutations, excluding trivial ones.
11. Tensor transpose bandwidths obtained on Intel Xeon Phi 5110P and NVidia Tesla K20X by the efficient and scattering algorithms, compared to the reference case of direct copy. Random sample consists of reverse permutations only (believed to be the worst case).
12. Tensor transpose bandwidths obtained on Intel Xeon Phi 5110P by the efficient and scattering algorithms, compared to the reference case of direct copy: Dependence on the tensor rank. Random sample consists of arbitrary permutations, excluding trivial ones.
13. Tensor transpose bandwidths obtained on NVidia Tesla K20X by the efficient and scattering algorithms, compared to the reference case of direct copy: Dependence on the tensor rank. Random sample consists of arbitrary permutations, excluding trivial ones.