

New Algorithm for Tensor Contractions on Multi-Core CPUs, GPUs, and Accelerators Enables CCSD and EOM-CCSD Calculations with over 1000 Basis Functions on a Single Compute Node

Ilya A. Kaliman* and Anna I. Krylov

A new hardware-agnostic contraction algorithm for tensors of arbitrary symmetry and sparsity is presented. The algorithm is implemented as a stand-alone open-source code *libxm*. This code is also integrated with general tensor library *libtensor* and with the *Q-Chem* quantum-chemistry package. An overview of the algorithm, its implementation, and benchmarks are presented. Similarly to other tensor software, the algorithm exploits efficient matrix multiplication libraries and assumes that tensors are stored in a block-tensor form. The distinguishing features of the algorithm are: (i) efficient repackaging of the individual blocks into large matrices and back, which affords efficient graphics processing

unit (GPU)-enabled calculations without modifications of higher-level codes; (ii) fully asynchronous data transfer between disk storage and fast memory. The algorithm enables canonical all-electron coupled-cluster and equation-of-motion coupled-cluster calculations with single and double substitutions (CCSD and EOM-CCSD) with over 1000 basis functions on a single quad-GPU machine. We show that the algorithm exhibits predicted theoretical scaling for canonical CCSD calculations, $O(N^6)$, irrespective of the data size on disk. © 2017 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.24713

Introduction

High-level quantum-chemistry calculations of large systems are extremely costly, owing to the steep computational scaling of the many-body theories.^[1,2] For example, solving the electronic Schrödinger equation exactly, using the full configuration interaction ansatz, scales factorially with the number of electrons/orbitals.^[1] Approximate approaches, such as coupled-cluster (CC) hierarchy of methods,^[3–5] enable practical calculations of moderate-size molecules, but their polynomial scaling limits the size of systems that can be tackled.^[2] For example, CCSD (CC with single and double substitutions) and CCSDT (CC with single, double, and triple substitutions) scale as $O(N^6)$ and $O(N^8)$, respectively.^[1]

According to Moore's law,^[6] the computational power of central processing units grows exponentially; that is, the number of transistors doubles approximately every two years. Large quantum-chemistry calculations that could only be performed using supercomputers 10 years ago are now possible on small clusters or even single workstation computers. At the same time, most processing power now comes from difficult-to-program graphics processing units (GPUs) and various hardware floating point accelerators (e.g., Intel Xeon Phi). Thus, one of the main challenges of modern algorithms is the efficient use of the new, increasingly complex hardware that can deliver large performance gains.

Using GPUs to accelerate electronic-structure calculations is not a new idea. For example, *NWChem*^[7,8] contains an implementation of non-iterative part of the CCSD(T) method on GPUs. Advanced electronic-structure codes, which exclusively target GPUs, also exist; one of them is *TeraChem*.^[9] The developers of

this code harnessed all available computing power by targeting exclusively CUDA technology and by optimizing every step of the calculation. Recently the SIAL environment used by ACES III package^[10] has been extended to GPUs.^[11] We pursue a different paradigm. We rely on extensive electronic-structure codes already available in packages such as *Q-Chem*.^[12,13] Our aim is to achieve performance gains by accelerating the most time-consuming operations, which dominate the calculation for very large problems.

Due to rapidly changing programming models and complex hardware, adapting advanced scientific codes to keep up with the hardware progress became an unrealistic endeavor. Optimizing quantum-chemistry codes can be a somewhat simpler task, once we recognize that the most computationally demanding part in many-body methods is operations on tensors. Tensors in quantum chemistry can be viewed as large (up to many terabytes or 2^{40} bytes) multidimensional arrays of floating point numbers exhibiting high degree of symmetry and sparsity. They represent everything from electron-repulsion integrals (ERIs) to wave-function amplitudes. Efficient storage and operations on tensors are thus the core of any modern high-performance quantum-chemistry code.

I. A. Kaliman, A. I. Krylov

Department of Chemistry, University of Southern California, Los Angeles, California 900890482 E-mail: ilya.kaliman@gmail.com

Contract grant sponsors: U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program; Contract grant sponsor: U.S. Air Force of Scientific Research (AFOSR); Contract grant number: FA9550-16-1-0051.

© 2017 Wiley Periodicals, Inc.

While tensors can be modified in various ways in the course of an *ab initio* calculation, one set of operations is of a particular importance. These are tensor contractions, which dominate the calculation even for medium-size systems. In large calculations, contractions take virtually all processing time and the contributions from other operations are negligible. Thus, efficiently carrying out tensor contractions is the key to achieving the best performance on modern hardware for large-scale quantum-chemistry calculations.

Any tensor contraction can be represented as matrix multiplication. Due to the ubiquity of matrix multiplication operations, many hours have been spent by software engineers optimizing these routines to the extreme. Matrix multiplication is a commonly used operation for benchmarking and measuring the performance of computer hardware. Reformulating various multi-index tensor contractions as multiplications of matrices allows one to use standard *BLAS*^[14] routines for all compute-intensive operations. Thus, various computer hardware can be viewed as a set of abstract devices that perform matrix multiplications. This strategy may allow one to carry out the computations as efficiently as the hardware and vendor's *BLAS* implementation permit.

Highly optimized *BLAS* routines are commonly used by most modern quantum-chemistry codes^[10,12,15–17] and tensor libraries.^[18–21] While the idea of using *BLAS* is obvious, doing so with maximum efficiency is not trivial. Maximum *BLAS* performance is only attained for large matrices. However, huge size and rich internal structure of the tensors arising in many-body theories results in complicated data patterns. To manage this complexity, large tensors are usually stored as a collection of relatively small blocks, following the block-tensor design first outlined by Windus and Pople.^[22] Consequently, the developers often use *BLAS* for operations on relatively small matrices, which deteriorates the performance.^[18] Furthermore, within this paradigm, programming other devices beyond CPU is problematic, as the programmers are required to manually manage data transfer to the device (e.g., GPU) to achieve performance gains. When using small matrices, one needs to deal with the programming details of that particular device; this makes porting software to new platforms tedious. The problem becomes even harder once we start dealing with more than one device on the same system, for example, in the case of computations on multiple GPUs.

A possible solution to this problem is to hide the implementation details of the matrix multiplication engine. This can be done using very large matrices to carry out computations. The advantage of using large matrices arises from the fact that all hardware vendors provide matrix multiplication routines that efficiently hide the details of data transfer to and from the device (or multiple devices) when computations are sufficiently large. To achieve the best performance on modern hardware, the size of such matrices must be at least on the order of gigabytes.

There are a number of tensor libraries and frameworks that enable complex operations on tensors. These include *libtensor*,^[18] Cyclops Tensor Framework (CTF),^[19] *TiledArray*,^[20] *TensorFlow*.^[21] They target various architectures and application domains ranging from single-node^[18] to massively parallel^[19] modalities.

Each has its strengths and weaknesses. The goal of the present work is not to introduce a new tensor library. Instead, we present an algorithm that solves the complexity problem of modern hardware for tensor contraction operations common in quantum chemistry by leveraging the standard matrix-multiplication routines. Due to extremely large size of tensors involved, all data have to be stored on hard disk. Efficient data transfer is achieved using fully asynchronous input/output (I/O) operations. Our algorithm speeds up the most important and time-consuming step in quantum-chemistry calculations, that is, contractions of large sparse tensors with complex symmetries. We present tests and illustrate the performance of our implementation by performing CCSD and EOM-CCSD calculations. We show that large CCSD calculations (with more than 1000 basis functions) can be routinely executed on a single multi-GPU server or workstation. The algorithm can be used as is or integrated within other tensor libraries or quantum-chemistry codes. In this work, we integrated the code with the C++ tensor library *libtensor*^[18] and the *Q-Chem* quantum-chemistry package.^[12,13] This is the first application of this algorithm to quantum-chemistry problems. An efficient implementation of the presented algorithm is released as a stand-alone open-source code, *libxm*.^[23]

The structure of the article is as follows. In the next section, we provide a concise summary of how tensors are represented and stored in electronic structure codes. Tensor Contraction Algorithm Overview section describes the tensor contraction algorithm implemented in *libxm*. Code Structure and Integration with *Libtensor* and *Q-Chem* section describes details of the computer implementation and the integration of *libxm* with *libtensor* and *Q-Chem*. Coupled-Cluster Calculations and Benchmarks section presents the results of performance benchmarks on CPU and GPU servers. Our concluding remarks are given in Conclusions section.

Tensors in Quantum Chemistry

In many-body theories, tensors (multidimensional array of numbers) represent everything from wave-function amplitudes to one- and two-electron integrals. The main challenges of handling tensors in quantum chemistry include large data size (up to many terabytes), their symmetry and sparsity, and many different types of contractions that need to be performed in a typical calculation. To achieve the best possible performance, practical codes must use all available properties of tensors that reduce their size and the amount of computations.

Efficient use of symmetry and sparsity, which arises naturally in many-body theories, affords greatest savings. In electronic structure theory, we exploit permutational symmetry, spin symmetry, and molecular point group symmetry. Permutational symmetry of ERIs, $v_{pqrs} = \langle pq || rs \rangle$, allows one to reduce the size of the stored data eightfold. Spin-symmetry also results in significant reduction of the data, especially in the case of closed-shell molecules. Symmetry and sparsity of tensors in quantum chemistry is described in detail in Refs. [22] and [18].

One strategy of exploiting symmetry and sparsity of tensors is using block-tensor design,^[18,22,24] which is illustrated in Figure 1. Block tensors allow one to take advantage of both

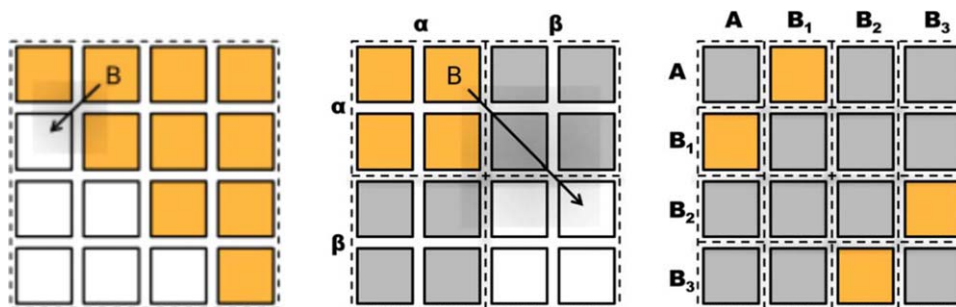


Figure 1. Block tensor structure. Permutational symmetry (left pane), spin symmetry (middle), molecular point-group symmetry (right). The left panel illustrates permutational symmetry of a Fock-like operator ($F_{pq} = F_{qp}$). The center panel illustrates spin-symmetry in a spin-conserving 2-index tensor, such as a Fock matrix or a single excitation EOM operator. The right panel illustrates point group symmetry of a B_1 2-index tensor, such as single excitation EOM operator corresponding to a B_1 transition. Orange blocks represent canonical or source blocks that are stored on disk; white blocks can be constructed from canonical blocks by permuting elements and multiplying by a scalar; grey blocks are zero-blocks. Reproduced with permission from Epifanovsky, et al., *J. Comput. Chem.* 2013, 34, 2293. [Color figure can be viewed at wileyonlinelibrary.com]

symmetry and sparsity, while making storage access efficient due to the data locality (the blocks are stored continuously in memory).

Block representation of tensors facilitates handling of both symmetry and sparsity. Space savings due to symmetry are achieved by storing only unique tensor blocks. The non-unique blocks are obtained by copying and permuting elements of the canonical block. For non-unique blocks, only the information about the permutation of elements and the scalar multiplier of the block needs to be stored.

To take advantage of sparsity (by sparsity here we mean blocks that only contain zero elements), we mark blocks as being zero and do not store any data at all. For example, for some spin-superblocks all blocks can simply be marked as zero and no data need to be stored.

Storage of tensors

Even when taking advantage of symmetry and sparsity, large tensors cannot be fully stored in fast random access memory (RAM). Even for medium-size CC calculations, data size is measured in hundreds of gigabytes. There are several ways of dealing with data size: distributed memory,^[19] storage on hard disk,^[18] and on-the-fly or “direct” calculations.^[7]

Distributed storage. Tensors can be distributed across multiple machines so that each node stores only a part of the data in RAM. This approach is used by codes that target large clusters of computers and supercomputers. For single-workstation machines, which are targeted here, this technique is not applicable. We note, however, that the greatest advantage of this approach is that tensors can be stored fully in fast memory. Even for the cases where tensors are stored on a remote machine in RAM, modern network technologies like Infiniband Remote Direct Memory Access provide better performance compared to accessing local spinning disks. Solid state drives (SSDs) can alleviate the problems of spinning disks somewhat, as they can perform random data access faster.

The disadvantage of distributed storage is that one needs hundreds of nodes to store data during large calculations. For example, if each node has 64 Gb of RAM, one needs over a 100

nodes to store in RAM the electron repulsion integrals for a job with 1000 basis functions. Moreover, algorithms that involve distributed storage are often complicated, as each node does not have all the data required for the calculation. The performance of such algorithms can be largely affected by the topology of a cluster interconnect.^[25] The need for large number of nodes can be alleviated using resolution-of-identity^[26,27] and Cholesky decomposition^[28,29] approaches, which reduce the size of the data.^[30]

On-the-fly calculation. Storage can often be avoided altogether using direct methods in which the integrals are recomputed each time when needed. This approach (used, for example, by *NWChem*^[7,8] and *ACES III*^[10]) affords scalability up to hundreds of thousands of processors. The big disadvantage is that the amount of calculations grows significantly. This is less of a problem for supercomputers where the overall computing power is often much larger compared to the performance of storage and communication subsystems. However, such schemes are computationally prohibitive on a single machine. Moreover, non-linear nature of some many-body methods precludes one from recalculating all large objects. For example, the CCSD and EOM amplitudes are calculated by an iterative algorithm; thus, they cannot be recomputed on-the-fly.

Disk storage. Calculating data once and storing it for later use is the optimal strategy from the point of view of the amount of calculations. The integrals and other quantities can be computed beforehand and stored on a hard drive until needed later. On modern systems, hard disk space is a virtually unlimited resource. Disks of several Tb per unit are common, so one can easily buy a server or workstation with over 10 Tb of storage. Moreover, SSDs can make storage subsystems much faster. If cost is prohibitive, one can use small SSDs as a cache for conventional disks to improve the performance. It should be noted that good hardware RAID arrays with fast write-back cache can give similar performance to SSDs, while costing an order of magnitude less compared to enterprise-class SSDs. The greatest disadvantage of the external storage is access speed, which is orders of magnitude slower compared to RAM or even fast network technologies, such as Infiniband. Fortunately, in virtually all high-level quantum-chemistry methods the amount of

calculations scale steeper than the data size. For example, for the conventional CCSD method, the calculations scale as $O(N^6)$, versus $O(N^4)$ scaling of the data size. A properly designed algorithm can hide the communication overhead by overlapping the data access with the calculations. Storage on disk is the only viable option for quantum-chemistry calculations on a single machine. Below we show that our implementation completely hides disk I/O using fully asynchronous data access.

Tensor Contraction Algorithm Overview

The basic principle of the algorithm is that any tensor contraction can be reformulated as a multiplication of two matrices, which is exploited in many electronic-structure codes. Let us consider a simple case of the contraction of two dense 4-index tensors (dense tensors have no symmetry or sparsity). Four-index quantities are very common in all many-body methods. For example, a contraction over 2 indices can be written as follows:

$$C_{klab} = \sum_{ij} A_{ijkl} B_{ijab} = \sum_L A_{Li}^M B_{Lj}^M = C_{ij}^M, \quad (1)$$

where superscript M denotes matricized (unfolded) tensors and $L=i \times j_{\max} + j$, $I=k \times l_{\max} + l$, $J=a \times b_{\max} + b$, where j_{\max} , l_{\max} , and b_{\max} are corresponding tensor dimension sizes. By unfolding we mean the process of converting a multi-dimensional tensor into matrix form. The reverse process is called folding.

As one can see, once we unfold the tensors, this contraction is equivalent to multiplication of matrices A^M (with dimensions $i_{\max} \times j_{\max}$ and $k_{\max} \times l_{\max}$) and B^M (with dimensions $a_{\max} \times b_{\max}$, $i_{\max} \times j_{\max}$). It is now clear that the contraction can be performed using one of the optimized routines for matrix multiplication, such as provided by the *BLAS* library.^[14] This analogy can be easily extended to arbitrary contractions of tensors. The naive version of the algorithm would thus involve the following steps:

1. Unfold tensor A to form matrix A^M so that contraction indices are in rows;
2. Unfold tensor B to form matrix B^M so that contraction indices are in columns;
3. Unfold tensor C to form matrix C^M so that rows and columns are consistent with the product of A^M and B^M ;
4. Perform matrix multiplication $C^M = \alpha \cdot A^M \times B^M + \beta \cdot C^M$ using the optimized routine (e.g., *BLAS dgemm*);
5. Fold C^M back to form tensor C .

In this way, one can perform arbitrary contractions using only matrix multiplications (some of the contractions may require data reshuffling, that is, permutation of the tensor dimensions so that the contraction indices are grouped together). Such approach is used in many quantum-chemistry codes and tensor libraries, such as *libtensor*.^[18] Our algorithm generalizes this scheme so that the same code can be executed on different hardware, that is, multi-core CPUs or GPUs. This is possible because the optimized *BLAS* matrix multiplication routines are readily available from vendors such as NVIDIA

or Intel. For example, for our benchmark calculations on GPU we use *cuBLAS-XT*, an implementation of *BLAS* routines from NVIDIA. It automatically uses multiple GPUs available in the system (up to 4 in our benchmarks). No modification of the code or the algorithm is necessary to benefit from such computer devices. This strategy allows one to achieve the best performance by reformulating tensor contractions as multiplications of large matrices and using optimized vendor libraries to perform the actual computations on CPUs and/or GPUs.

Of course, the naive implementation outlined above is very inefficient due to the following issues:

1. Tensors are often too large to fit in RAM;
2. The resulting temporary matrices are too large to fit in RAM;
3. All operations are performed sequentially and can become bottlenecks;
4. Symmetry or sparsity is not used in the naive implementation.

Data size is the main challenge of a large quantum-chemistry calculation. For a single workstation, the only option is to store all data on the hard drive and perform the contraction in batches, using matrices that fit in fast memory. To tackle the data transfer issue, we carry out contractions in chunks, as multiplications of carefully assembled big matrices. Such operations are then asynchronously offloaded to an accelerator, while reading and preparing the next piece of data from the hard disk. Because the computational cost of matrix multiplication scales as $O(N^3)$, whereas the data size scales as $O(N^2)$, it is always possible to find N for which the problem will be compute-bound. A vendor-optimized *BLAS* matrix multiplication routine can be used to take advantage of available resources in the most efficient way. For example, multithreaded Intel MKL *BLAS* can be used on multi-core CPUs, whereas NVIDIA CUDA implementation or other libraries can be used if one or more GPUs are available. The algorithm can benefit from multiple accelerator devices in the system with no modifications.

The implemented algorithm works by performing all work in parallel using threads as shown on Figure 2. Its four main elements are:

1. Prefetching data blocks from the disk storage;
2. Permuting the elements to form the temporary matrices;
3. Multiplying the matrices using the optimized matrix-multiplication routines;
4. Folding the result back to the block-tensor format.

Steps 1, 2, and 4 are performed on CPU in parallel. Rate-limiting step 3 can be performed either on CPUs or offloaded to GPUs or other hardware accelerators.

For each tensor, a buffer consisting of several parts, is allocated in RAM. Each block of memory is referred in *libxm* code^[23] as *blk_a1*, *blk_a2*, *blk_a3*, *blk_a4* for tensor A ; *blk_b1*, *blk_b2*, *blk_b3*, *blk_b4* for tensor B ; *blk_c1*, *blk_c2*, *blk_c3* for tensor C .

The algorithm works iteratively with the buffers used as follows:

1. Memory blocks *blk_a4* and *blk_b4* are used for prefetching blocks for the next algorithm iteration from the disk for tensors A and B , respectively.

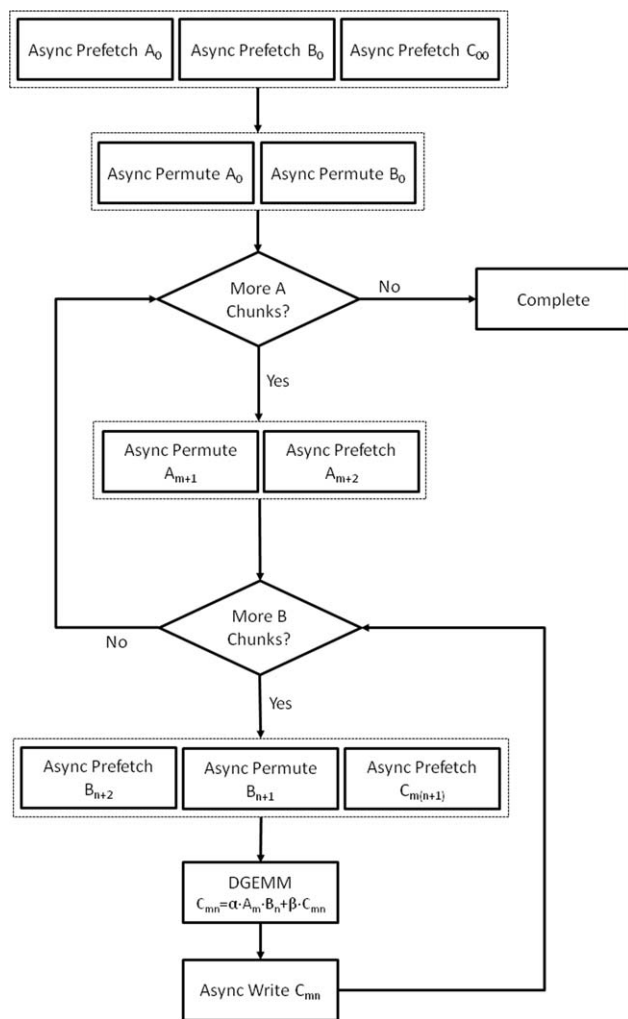


Figure 2. Flowchart of the tensor contraction algorithm. The two conditionals represent loops over tensors *A* and *B*, respectively. The operations in dotted boxes are performed in parallel using *POSIX* threads.

- blk_a3* and *blk_b3* store the blocks prefetched in the previous iteration; they are permuted and packed into *blk_a2* and *blk_b2*. These are the matrices that will be passed to *dgemm* routine in the next iteration.
- Buffers *blk_a1* and *blk_b1* store the completed matrices prepared in the previous iteration. These are passed to the matrix multiplication routine in the current iteration.

For tensor *C*, there is no need to perform the permutations as only the canonical blocks are needed to form the result. Instead, an additional buffer, *blk_c3*, is used for unfolding the result and writing it back to disk.

The algorithm works by processing tensors in batches. The tensors *A* and *B* are split, so that the batch fits in the preallocated buffers, as described above. The iterations go over the larger tensor once. Let us assume that *A* is the larger of the two tensors. For each batch of *A*, the algorithm iterates over the entire tensor *B* once. As soon as the first chunk of *A* and all chunks of *B* are contracted, the next chunk from *A* is used. In total, there are $M \times N$ iterations, where M and N are the number of chunks in

tensors *A* and *B*. The process can also be performed the other way around, that is, going through *B* once and through *A* many times. But for efficient caching, it is more advantageous to go over a smaller tensor many times as it is more likely to stay in the OS filesystem cache. For optimal performance, the total size of all buffers is automatically set to 25% of the available RAM. If desired, a custom buffer size can be specified,

As only canonical blocks in resulting tensor *C* need to be computed, many operations can be avoided by efficiently packing the resulting matrices into smaller ones by omitting blocks that are not used. For example, if $C_{pqrs} = C_{qprs} = C_{pqsr}$, we only need to compute the blocks that are unique (marked orange in Fig. 3). We can pack the resulting unfolded tensor into a smaller matrix, as shown on Figure 3. The packing is performed automatically by scanning the unfolded tensor for rows and columns of blocks that can be discarded. In some cases, it is not possible to fully pack the tensor and the resulting packed unfolded tensor contains spots of “white” non-canonical blocks. In this case, extra computations are performed and blocks that are not needed are discarded after the computation.

It is critical for the performance that the permutation of elements and unfolding of the blocks into matrices is done as efficiently as possible. The naive algorithm goes over all elements of the block and copies them one-by-one into a matrix, applying the correct permutation. In our experience, such trivial implementation is not sufficiently fast to achieve efficient overlap of permutations with matrix multiplications. The key observation, which we exploit to significantly increase the performance of this step, is that in many cases the order of the permuted data in memory is the same as the order in the original block. We can use this fact to avoid copying individual elements and instead copy entire rows of data using standard *memcpy* *C* function, which is very efficient. This way instead of looping over all elements of a block (which is $\sim N^4$ for 4-index tensor), we only loop over all rows (which is on order N^3 for 4-index tensor). Although the amount of the data copied is the same as in the trivial algorithm, thanks to the efficiency of *memcpy*, this change alone increases the throughput of permutations 4–5 fold for larger block sizes. This is one of the reasons for using a larger block size of at least 32 elements, versus 16 in the original *libtensor* code. Once the elements are copied, the scalar multiplier (e.g., -1 for asymmetric blocks) is applied to the resulting matrix.

The size of the buffer required to fully overlap the calculations and data transfer can be estimated as follows. First, let us assume that the matrices are square with dimension N . Consider an abstract device able to achieve 1 TFLOP/s (10^{12} floating point operations per second) on double-precision matrix multiplications. Such performance was demonstrated on the NVidia GPUs and Intel Xeon Phi devices. The typical block size in our calculations is several megabytes (e.g., 32^4 elements for 4-index tensors). We assume that the read speed is close to a sequential read and that the storage subsystem is capable of 100 Mb/s reading speed, which is a conservative estimate for what is achievable using SAS disks with hardware RAID or enterprise-class SSDs. We can also assume that, due to symmetry/sparsity of a typical tensor, the

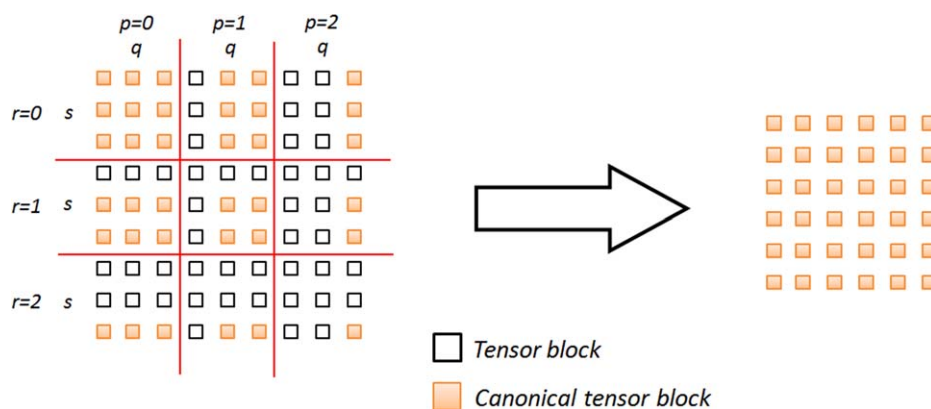


Figure 3. Tensor packing scheme used by *libxm*. Orange blocks are canonical blocks that need to be computed. White blocks are obtained by permuting the elements of the corresponding canonical block so they can be omitted from the computation. [Color figure can be viewed at wileyonlinelibrary.com]

block is actually read from disk only half of the time and the other half it is available instantly from the filesystem cache. That gives us an effective bandwidth close to 200 Mb/s, which is consistent with what we have observed in our benchmarks. Reading the data from the disk and the formation of temporary matrices from tensors require $O(N^2)$ operations. However, on modern CPUs data reshuffling in memory is usually much faster compared to disk access. We consider disk read to be the time-limiting step in our estimation.

To form resulting matrix C^M , we need to compute N^2 elements. Each element requires $2 \times N$ FLOPs: N multiplications and $N - 1$ additions. The total required operations for the matrix multiplication is $2 \times N^3$. We assume double width floating point type, which is 8 bytes per element. This gives the resulting matrix size of $8 \times N^2$ bytes. Making matrix multiplication time equal to the time required to read the next chunk from the disk, we arrive at the following formula:

$$2 \times N^3 / P = 8 \times N^2 / R, \quad (2)$$

where P is the matrix multiplication performance in operations per second and R is disk read speed in bytes per second. Inserting our estimates for P and R , we have:

$$2 \times N^3 / 10^{12} = 8 \times N^2 / (200 \times 1024^2), \quad (3)$$

which gives $N = 20,000$ or about 3 Gb per matrix. Our algorithm requires 11 such buffers (for asynchronous prefetching and permutation of tensors A , B , and C). So the total memory requirements are about 32 Gb of RAM. At least the same amount of RAM is desirable for filesystem cache. The total RAM of 64 Gb RAM is needed on such hypothetical system to fully overlap the disk I/O and the computations.

The main advantage of the presented algorithm is that there are no bottlenecks as all operations are performed in parallel. Given large enough buffer, matrix multiplication will always be a limiting step for large calculations, as it takes $O(N^3)$ time versus $O(N^2)$ for all other operations. Multiplication of temporary matrices can be performed either using multi-core CPUs or GPUs. Thanks to efficient BLAS routines, the computing resources will always be used in the optimal way. These

properties of the algorithm ensure that its performance will improve with each new generation of computer hardware.

Code Structure and Integration with *Libtensor* and *Q-Chem*

Libxm is a stand-alone implementation of the tensor-contraction algorithm described in this article. To run quantum-chemistry calculations, one needs to integrate it with a full-fledged software package that performs other necessary operations. Our code targets high-level calculations of large systems. For maximum speed, the package should perform all steps as efficiently as possible, even though the majority of time will be spent in the new tensor-contraction code. *Q-Chem*^[12,13] is a state-of-the-art quantum-chemistry package with many features. Recently, a new framework called *libtensor*^[18] was introduced in *Q-Chem* to hide tensor-related complexities from the developers of many-body models. While *libtensor* shows excellent performance for moderate-size systems, its performance deteriorates when the data size is larger than the available fast memory. The *libtensor* code was designed to run on multicore CPUs^[18] and is not adapted for other devices such as GPUs.

Libtensor provides an excellent framework for integrating with other software. Recently, it was integrated with CTF^[19] to take advantage of distributed memory systems.^[25] *Libtensor* provides an easy way to create and manage tensors with complex symmetries. It was a natural choice for integration with our new code *libxm*.

We have supplemented an internal memory allocator of *libtensor* with the one from *libxm*. This allocator is used when the calculations are performed using the new code, as it provides better performance when tensors do not fit in fast memory. *Libtensor* provides a flexible template-based allocator framework, so extending the allocation routines is straightforward. Tensors are always stored on disk when new disk-based memory allocator is used. Data caching to RAM is managed by the operating system and does not require any additional work from the developer. The total data size is only limited by the available disk space and filesystem limits. The provided allocation routines are similar to the ones from the C standard library. The `xm_allocator_allocate` function takes the allocation

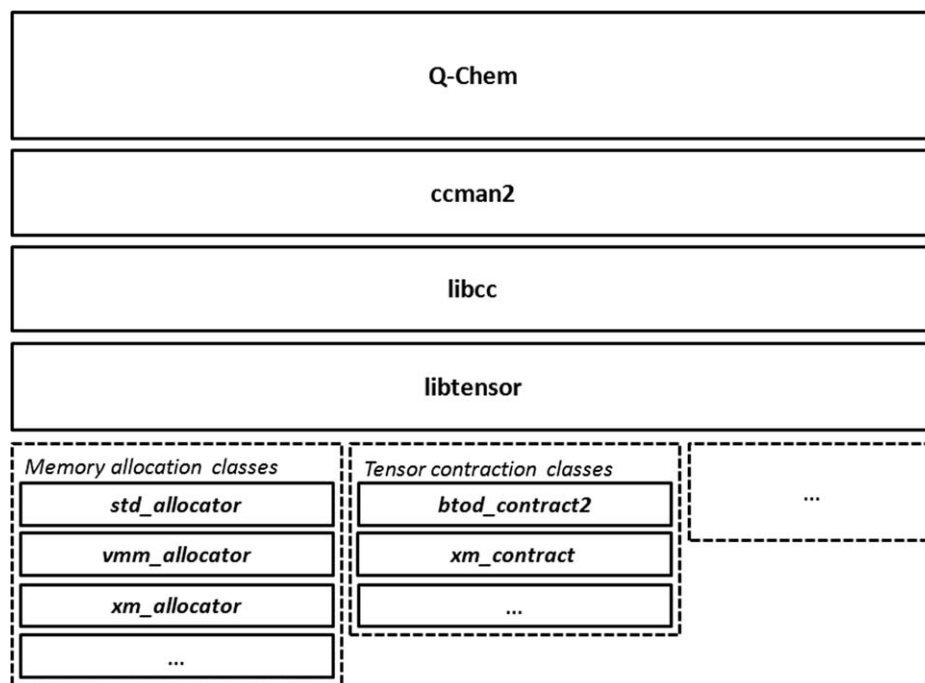


Figure 4. *libxm* integration with the *Q-Chem* package.

size and returns a virtual pointer to the data. The actual data can be retrieved to physical memory using *xm_allocator_read* function. To write the data back, the *xm_allocator_write* function is used. When creating a new tensor, only the unique (canonical) blocks need to be allocated and stored. Other blocks are either marked as zero (using *xm_tensor_set_zero_block*) or reference an arbitrary permutation of a canonical block. These functions are not used directly when using the new code through *libtensor* interface. All memory allocation and tensor creation are managed using high-level C++ objects from *libtensor*. *Libxm* supports both single and double-precision data, as well as complex numbers.

Once the tensors are allocated and filled with data, they are passed to the *xm_contract* function, which is very similar to the BLAS matrix multiplication function *dgemm*:

```
xm_contract(alpha, A, B, beta, C, "ijab", "abcd",
"ijcd");
```

Here, *alpha* and *beta* are scalars and *A*, *B*, and *C* are tensors of the *xm_tensor* type. This code performs the following operation:

$$C_{ijcd} := \alpha \cdot \sum_{ab} A_{ijab} B_{abcd} + \beta \cdot C_{ijcd}. \quad (4)$$

All internal contraction functions in *libtensor* are replaced by the *xm_contract* routine when the new code is enabled. No other *Q-Chem* code directly calls low-level *libtensor* routines. This means that any code that uses *libtensor* automatically benefits from the new functionality. These include all CC and EOM-CC codes^[18] as well as the suite of the ADC methods^[31] in *Q-Chem*. In a similar fashion, *libxm* can be interfaced to other electronic structure codes.

The *libxm* code was extensively tested in several ways. Numerous native *libxm* tests were primarily used during the

code development stage. The integrated code was checked by running the full test suite from *libtensor*. Finally, it was verified that the results of all end-test electronic structure calculations using the new code match the reference outputs digit-by-digit.

The overall structure of the *Q-Chem/libxm* code is shown in Figure 4. *Q-Chem's* layered architecture greatly simplifies code modifications. The changes made at lower levels are automatically available to all methods higher in the stack.

Coupled-Cluster Calculations and Benchmarks

Hardware details

All benchmark calculations using GPUs were performed on a Dell R730 server with two 4-core Intel Xeon E5-2623 CPUs, 384 Gb of RAM, and two NVidia Tesla K80 cards. Each Tesla K80 card contains two K40-class GPUs (giving a total of four GPUs in the system) and 24 Gb of GDDR5 memory. The system also has 8 × 1.2 Tb SAS hard drives in RAID-0 configuration, giving rise to about 8 Tb of usable scratch space. The NVidia CUDA toolkit version 7.5 was installed on this machine.

We note that the code needs at least as many CPU cores as there are GPUs, that is, one CPU core per GPU. Having more CPU cores does not affect the timing, unless one uses a version of BLAS able to use both GPU and CPU. Our GPU benchmark machine has eight CPU cores and four GPUs, so four CPU cores are used during GPU calculations.

Several CPU-only systems were also used for benchmarking. These include Dell R720 and R820 servers with 2 and 4 CPU sockets giving rise to a total of 8 to 32 CPU cores per server. The amount of RAM in these nodes ranges from 48 to 768 Gb.

Table 1. Time in seconds for large matrix multiplication with double-precision numbers (*dgemm*) on CPU (speedup relative to 1 CPU core is shown in parenthesis).

1 CPU core	2 CPU cores	4 CPU cores	8 CPU cores
1683 (1×)	882 (1.9×)	494 (3.4×)	285 (5.9×)
Square matrices with dimension $N = 32,768$ elements.			

Fast SAS disks in hardware RAID-0 configuration are used for scratch space in all configurations.

Benchmarking matrix multiplication on CPUs and GPUs

The core part of the presented algorithm is multiplication of large in-memory matrices. Since this step is the time-limiting step, the matrix multiplication performance largely determines the overall performance of the algorithm. In this subsection, we present the benchmarking results for optimized matrix multiplication routines provided by the *BLAS* libraries. We compare the performance of NVidia's *cuBLAS-XT* on GPUs with Intel's MKL implementation on multicore CPUs. Double-precision numbers are used in all calculations and benchmarks. Intel MKL Version 11.1.3 was used for the CPU benchmarks and NVidia CUDA 7.5 was used for GPUs. The GPU-system described above was used for the tests. The results are shown in Tables 1 and 2. The numerical values obtained using CPUs and GPUs are identical in all our tests. We note that the performance results can vary for different matrix sizes and dimensions. Nevertheless, these tests provide a good sense of the performance that GPUs give for double-precision matrix multiplication. All tests show execution time for one *dgemm* call. The input and output matrices resided in the host memory and were filled with pseudorandom data. Several calculations were repeated and the best result is shown here.

In these tests, *cuBLAS-XT* on GPUs outperforms Intel MKL *BLAS* on CPUs. Both MKL and CUDA implementations scale well with the number of cores/devices. Users should carefully choose GPU tile size. The tile size parameter determines the batch size when the matrix is transferred to the device memory from the host memory. If the matrix is smaller than the tile size, the remainder is filled with zeros and unnecessary work is performed. Our extensive testing shows that the tile size of 4096 is optimal for large electronic structure calculations. This tile size was used in all GPU benchmark calculations.

We note that the tile size parameter is only used by *cuBLAS-XT*, a particular *BLAS* implementation we used for GPU calculations. In future versions of *cuBLAS-XT* the optimal value may change or the implementation may ignore it and select some value automatically (as it probably should). At present our recommendation is that the tile size does not exceed σ^2 , where σ is the number of occupied orbitals.

We also observe that in real-world electronic-structure calculations the actual performance of CUDA *BLAS dgemm* is somewhat worse than in these synthetic benchmarks. A likely explanation is that competing threads reading the data from disk storage and permuting tensor elements create additional pressure on the memory subsystem. This negatively affects data transfer to and from the GPU cards. New socket design

of the upcoming Intel's Knights Landing architecture may be less affected, as it offers much higher memory bandwidth. As noted before, the code does not require any modifications to benefit from new hardware architectures.

Coupled-cluster benchmarks on multicore CPUs

We begin by benchmarking the new code, *libxm*, against the original *libtensor*-based implementation.^[18] This series of tests show the CCSD performance for both codes on multicore CPUs. The original *libtensor* implementation is not designed to run on GPUs.

The main difference in algorithms in the *libxm* code and in *libtensor* is that the latter flattens individual blocks and uses the resulting small matrices for operations. In the case of *libxm*, multiple blocks are asynchronously flattened and combined into one big matrix, which is used for contractions. The parallelism in *libtensor* is achieved by performing operations on several separate blocks in parallel, whereas *libxm* is parallelized using multi-threaded *dgemm* *BLAS* function from Intel MKL on CPUs or CUDA *BLAS* on GPUs for multiplication of the resulting large matrices.

We used two systems for comparison: methylated uracil dimer with a water molecule (test 3 from Ref. [18]) and a nucleobase tetramer, AATT.^[32] For AATT, we carried out benchmark calculations using Frozen Natural Orbitals (FNO) approach,^[33–36] which reduces the size of virtual space (as was done in Ref. [18]), and with the full virtual space. Core electrons were frozen in these tests. The Cartesian geometries can be found in Ref. [18].

The results are summarized in Table 3. As one can see, the data size increases from left to right, while the amount of RAM is increased as we go from the top of the table to the bottom. For the cases when data is much larger than the RAM size, the new code significantly outperforms *libtensor*. This is due to a more robust handling of I/O, which is fully overlapped with calculations. For the cases where the data fully fits in RAM, the original code performs slightly better due to more efficient utilization of symmetry and sparsity of tensors. The speedup factor gradually grows if one goes from the bottom-left corner (smallest data-RAM ratio) to top-right corner (highest data-RAM ratio). For example, the *libtensor* AATT benchmark without FNO did not finish after 36 days on a 12 core node.

The less than ideal performance of the new code for small systems is explained by the following:

Table 2. Time in seconds for large matrix multiplication with double-precision numbers (*dgemm*) on GPU for different tile sizes (speedup relative to eight CPU cores is shown in parenthesis).

Tile size	1 GPU	2 GPU	3 GPU	4 GPU
2048	241 (1.2×)	94 (3×)	136 (2.1×)	100 (2.8×)
4096	107 (2.7×)	61 (4.7×)	81 (3.5×)	64 (4.5×)
8192	126 (2.2×)	63 (4.5×)	49 (5.8×)	33 (8.6×)
Square matrices with dimension $N = 32,768$ elements. Input and output matrices are in host memory with data transfer to GPU handled internally by CUDA libraries.				

Table 3. Benchmark comparison of *libtensor* and *libxm* codes.

Node Spec	mUracil dimer + water molecule, 489 BSF (250 Gb data size)		AATT with FNO, 968 BSF (800 Gb data size)		AATT, 968 BSF (3.1 Tb data size)	
8 cores, 48 Gb RAM	lt: 5.4 hrs xm: 3.2 hrs	1.7×	–	–	–	–
12 cores, 128 Gb RAM	lt: 2.6 hrs xm: 2.1 hrs	1.2×	lt: 105.9 hrs xm: 20.9 hrs	5.1×	lt: >36 days xm: 102 hrs	>8×
16 cores, 380 Gb RAM	lt: 0.6 hrs xm: 0.7 hrs	0.8×	lt: 10.3 hrs xm: 8.8 hrs	1.2×	lt: 197.0 hrs xm: 31.3 hrs	6.3×
32 cores, 768 Gb RAM	lt: 0.5 hrs xm: 0.6 hrs	0.8×	lt: 5.6 hrs xm: 5.8 hrs	0.9×	lt: 39.7 hrs xm: 29.9 hrs	1.3×

Time per one CCSD iteration is shown.

1. Due to interfacing with *Q-Chem*, extra copying of data is performed, which takes a noticeable fraction of time for small systems. For large systems the relative time needed for copying is negligible compared to the overall calculation time.
2. The time spent in contraction function is relatively insignificant for small tasks and the performance improvements are not noticeable. However, for large systems virtually all time is spent in *xm_contract* and the performance of other parts of the code does not contribute much to the overall calculation time even when executed on slower CPUs.
3. Small calculations do not benefit significantly from acceleration due to small size of the tensors and matrices.

Coupled-cluster benchmarks on GPUs

In this section, we show a series of coupled-cluster calculations, which demonstrate the capabilities of the new code on GPU-enabled systems.

Water ladder benchmark. Owing to the importance of water, water clusters are popular in benchmark studies. Here, we present the “water ladder” benchmark, which demonstrates scalability of the code with the system size. We consider a series of water clusters of the increasing size. We generated the input structures by removing a required number of water molecules from the initial water cluster structure containing 23 water molecules. The Cartesian geometries and relevant energies are given in Supporting Information. All-electron CCSD calculations with the cc-pVTZ basis set were performed to measure how our algorithms scales with the data size. All calculations were performed using the GPU machine described above. Table 4 and Figure 5 present the results of these calculations.

For $(\text{H}_2\text{O})_{18}$, the total number of basis functions exceeds 1000 and the total data size is close to 5 Tb. A single CCSD iteration takes about a day (24.4 hrs) for this system on our GPU machine. One CCSD step for the same calculation takes approximately 39.2 hrs when not using GPU acceleration. Figure 5 shows the sixth root of the total single-point energy calculation time versus the number of water molecules in the

cluster (data from the last column in Table 4). CCSD scales as N^6 , where N is the number of orbitals. As illustrated in Figure 5, the algorithm scales perfectly with system size, that is the dependence is linear and does not depend on the size of data on disk. Our algorithm can fully hide disk I/O even when data size is over $10 \times$ the available RAM (see “Data Size” column in Table 4). Additionally, Table 5 shows time for CCSD energy calculation for $(\text{H}_2\text{O})_8$ using different basis sets. The algorithm shows good performance both on CPU and GPU, with CPU being about $1.5 \times$ slower. Such low speedup for GPU can be explained by this system having too small occupied space (40 orbitals). This leads to unfavorable matrix dimensions making matrix multiplications closer to vector products. In comparison, for the AATT benchmark (see next section) the GPU speedup is closer to $2 \times$ due to larger occupied space.

We have also performed a single-point CCSD calculation for a $(\text{H}_2\text{O})_{20}$ cluster using the cc-pVTZ basis set with frozen core. The wall clock time for one CCSD step using our code on a GPU-machine is 27.8 hrs. We contrast this to the calculations using *NWChem*. In Ref. [8], such calculations were performed on a supercomputer using 96,000 processors. While the absolute time per step was shorter, the calculations required over 100 Tb of memory^[8] versus 6 Tb for our code and much more computing resources and power.

Although the algorithm does not take full advantage of symmetry because merging small blocks into large matrices sometimes requires keeping zero blocks, the code handles point group symmetry reasonably well. As an example, consider the CCSD calculation of the oligoporphyrin dimer molecule with cc-pVDZ basis set (942 basis functions) with D_{2h} symmetry. The use of point-group symmetry reduces storage requirements sixfold. The single CCSD iteration takes about 12 hrs on our GPU node, to be compared with 13.2 hrs on 12 cores for *libtensor* and 810 sec on 1024 cores for *NWChem*.^[18] We note that the performance of the algorithm for systems with point-group symmetry can be improved. A new version of the algorithm, which shows better performance for high-symmetry cases, is included in Ref. [23].

AATT nucleobase tetramer. Table 6 shows the benchmark results for the nucleobase tetramer, AATT, on CPU and GPU using new *libxm* code on a GPU machine. The CPU timings are for the calculations using eight cores on the same machine

Table 4. "Water ladder" benchmark on a GPU machine.

<i>N</i> water molecules	Number of basis-functions	Data size (Tb)	Number of CCSD iterations	Time per one CCSD iteration (hrs)	Time per CCSD energy calculation (hrs)
1	58	<0.1	7	<0.1	<0.1
2	116	<0.1	8	<0.1	<0.1
3	174	<0.1	8	<0.1	0.1
4	232	<0.1	8	<0.1	0.2
5	290	<0.1	8	<0.1	0.4
6	348	<0.1	8	0.1	0.8
7	406	0.1	9	0.2	1.7
8	464	0.2	9	0.3	2.8
9	522	0.3	9	0.5	6.1
10	580	0.4	9	1.0	12.2
11	638	0.6	9	2.0	19.7
12	696	0.9	9	3.3	32.5
13	754	1.2	9	5.8	55.8
14	812	1.6	9	8.0	78.1
15	870	2.1	9	9.8	97.4
16	926	2.7	9	14.1	138.1
17	986	3.5	9	17.3	178.1
18	1044	4.8	9	24.4	235.3

All-electron CCSD calculations with cc-pVTZ basis set.

but with the GPUs disabled. Calculations with frozen core and with all electrons active are shown. The 6-311+G(d,p) basis set was used with a total of 968 basis functions. In the all-electron calculation, there are 136 occupied and 830 virtual orbitals. In the calculation with frozen core, there are 38 frozen occupied, 98 active occupied, and 830 active virtual orbitals. In the FNO calculation, there are 98 active occupied and 552 active virtual orbitals.

The data in Table 6 shows that enabling GPUs allows one to run large CCSD calculations about twice faster compared to using only CPUs on the same machine. We can also compare the data in Table 6 with the last column in Table 3 for the AATT benchmark with frozen core. In this case GPU gives about 1.7x speedup over 32 Xeon E5-4640 2.4 GHz CPU cores. We note that the monetary cost of the GPU machine is about half of that for the 32 core machine from Table 3.

Even though GPUs give good speedups for large compute-bound contractions like T_2 amplitudes with vvv integrals, the maximum attainable speedup is limited by other contractions in the calculation. For example, the largest contraction in a CCSD

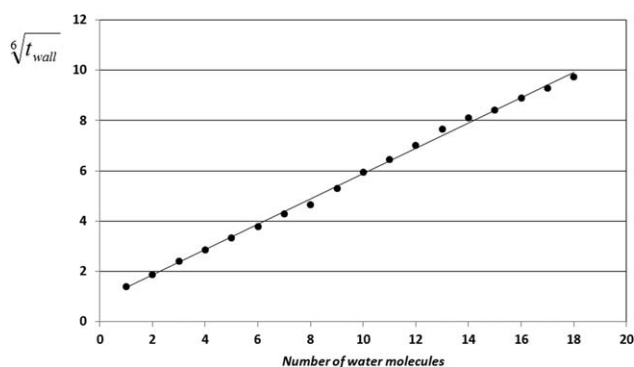


Figure 5. Sixth root of calculation wall clock time versus the number of water molecules for a single-point CCSD calculation. Libxm algorithm on a GPU machine.

calculation takes about 5 hrs on GPU. The same contraction takes 15 hrs if performed without GPU acceleration. The 3x speedup in this case is consistent with matrix multiplication benchmarks shown in Table 2. Nevertheless, many contractions over 1 and 3 indices in CCSD are I/O bound and can take significant portion of the calculation. Even though these contractions do not contribute to the $O(N^6)$ scaling of CCSD, they still take considerable time. The share of these contractions in the overall calculation time becomes smaller as the problem size increases.

We found that for the CCSD tests, an extra K80 card (that is, going from 2 GPUs to 4 GPUs) gives about 20% performance improvement. This is consistent with our matrix multiplication benchmarks from Table 2. For example, for the test calculation of $(\text{H}_2\text{O})_{16}$ with four GPUs enabled, a CCSD step takes 14 hrs 6 min, whereas with two GPUs it takes 16 hrs 42 min.

EOM-CCSD calculations on GPU. Owing to the layered architecture of *Q-Chem* (see Fig. 4) a large variety of correlated methods automatically benefits from the new algorithm. In the previous subsection, we demonstrated CCSD calculations on GPUs. Table 7 compares timings for the EOM-CCSD calculations for water clusters on GPUs with regular CCSD. The benchmark EOM calculations were performed for four EOM states (in the EOM calculations, time per iteration is roughly proportional to the number of states requested).

The timings demonstrate that EOM-CCSD calculations can be efficiently carried out on a single node. The EOM-EE-CCSD

Table 5. Time for CCSD energy calculation for $(\text{H}_2\text{O})_8$ with different basis sets.

Basis set	Time on GPU (min)	Time on CPU (min)
cc-pVDZ (192 b.f.)	1.0	1.4
cc-pVTZ (464 b.f.)	17.3	26.3
cc-pVQZ (920 b.f.)	480.1	712.9

All electrons are active.

Table 6. Nucleobase tetramer AATT benchmarks with and without GPU acceleration for all-electron and frozen-core calculations (data size is shown in parenthesis).

	Frozen Core (3.1 Tb)	All-electron (4.0 Tb)
8 Xeon E5-2623 3.0 GHz CPU cores	30.8	62.7
2 K80 GPUs	17.8	34.0
Time per one CCSD iteration in hours.		

calculations take the longest time to complete. Unfortunately, the EOM-EA-CCSD and EOM-EE-CCSD calculations for 18 water molecules did not fit the available scratch disk space on the node. To enable such calculations, resolution-of-identity^[26,27] and Cholesky decomposition^[28,29] approaches can be used^[30]; this extension is the subject of future work.

Conclusions

We presented a new general contraction algorithm for tensors of arbitrary symmetry and sparsity. A production-ready stand-alone open-source implementation is available as *libxm* code.^[23] The most recent version of the algorithm can be found in Ref. 23. This software is hardware agnostic and works without modifications on multi-core CPUs, GPUs, and other floating-point accelerators. Our implementation is integrated with the open-source library *libtensor*^[18] and incorporated in the *Q-Chem* package.^[12,13] In a similar fashion, *libxm* can be interfaced to other packages, either directly, or via *libtensor*. *Libxm* enables fast large-scale electronic structure calculations on regular workstations. The new code affords all-electron CCSD and EOM-CCSD calculations with over a 1000 basis functions on a single compute node. We demonstrated that such large CCSD calculations of water clusters with no point group symmetry take less than 10 days when using GPUs to accelerate the calculations on modern computers.

The algorithm is future-proof because it does not depend on peculiarities of computer hardware and can work efficiently on any future devices, as long as vendor-optimized matrix-multiplication routines are available. While *libxm* was designed to accelerate a variety of many-body electronic structure methods, it can also be used in other domains of science that deal with contractions involving very large tensors.

According to Moore's Law,^[6] the number of transistors in CPUs doubles approximately every two years. Disk storage and

Table 7. Iteration timings for the CCSD and EOM-CCSD methods.

<i>N</i> water molecules	Time per CCSD iteration (hrs)	Time per EOM-IP iteration (hrs)	Time per EOM-EA iteration (hrs)	Time per EOM-EE iteration (hrs)
12	3.3	0.5	2.8	5.4
14	8.0	0.7	6.3	13.9
16	14.1	1.8	9.4	23.7
18	24.4	4.1	–	–
Water clusters with the cc-pVTZ basis set, all electrons active, four EOM states.				


RAM sizes also grow exponentially, even at a higher rate.^[37] Here, we have shown that as long as sufficient amount of fast memory is available, the algorithm can provide best performance, while fully hiding scratch-disk I/O. Given the exponential growth of the computing power and storage capacities, we anticipate that CCSD calculations with several thousand basis functions will soon be practical on a single server. This promises a bright future for large-scale electronic structure calculations on workstation PCs.

Acknowledgment

We are thankful to Dr. Evgeny Epifanovsky for his help in interfacing the code with *libtensor* and with the *Q-Chem* package. This work also benefited from scientific interactions within *MolSSI* initiative supported by the National Science Foundation (*molssi.org*).

Keywords: coupled-cluster · equation-of-motion coupled-cluster · GPGPU · tensor computations · many-body theories · electronic structure

How to cite this article: I. A. Kaliman, A. I. Krylov. *J. Comput. Chem.* **2017**, *38*, 842–853. DOI: 10.1002/jcc.24713

 Additional Supporting Information may be found in the online version of this article.

- [1] T. Helgaker, P. Jørgensen, J. Olsen, *Molecular Electronic Structure Theory*; Wiley: Chichester, West Sussex, England, **2000**.
- [2] M. Head-Gordon, *J. Phys. Chem.* **1996**, *100*, 13213.
- [3] R. J. Bartlett, *Mol. Phys.* **2010**, *108*, 2905.
- [4] J. F. Stanton, J. Gauss, *Adv. Chem. Phys.* **2003**, *125*, 101.
- [5] J. D. Watts, J. Gauss, R. J. Bartlett, *J. Chem. Phys.* **1993**, *98*, 8718.
- [6] Moore's law. Available at: https://en.wikipedia.org/wiki/Moore's_law, accessed on July 20, **2016**.
- [7] R. Kobayashi, A. P. Rendell, *Chem. Phys. Lett.* **1997**, *265*, 1.
- [8] E. Aprà, A.P. Rendell, R.J. Harrison, V. Tipparaju, W.A. deJong, S.S. Xantheas. Liquid water: Obtaining the right answer for the right reasons. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*; ACM: New York, NY, **2009**; pp. 66:1–66:7.
- [9] I. S. Ufimtsev, T. J. Martinez, *Comput. Sci. Eng.* **2008**, *10*, 26.
- [10] E. Deumens, V. F. Lotrich, A. Perera, M. J. Ponton, B. A. Sanders, R. J. Bartlett, *WIREs Comput. Mol. Sci.* **2011**, *1*, 895.
- [11] N. Jindal, V. Lotrich, E. Deumens, B. A. Sanders, *Int. J. Parallel Program.* **2016**, *44*, 309.
- [12] A. I. Krylov, P. M. W. Gill, *WIREs Comput. Mol. Sci.* **2013**, *3*, 317.
- [13] Y. Shao, Z. Gan, E. Epifanovsky, A. T. B. Gilbert, M. Wormit, J. Kussmann, A. W. Lange, A. Behn, J. Deng, X. Feng, D. Ghosh, M. Goldey, P. R. Horn, L. D. Jacobson, I. Kaliman, R. Z. Khaliullin, T. Kus, A. Landau, J. Liu, E. I. Proynov, Y. M. Rhee, R. M. Richard, M. A. Rohrdanz, R. P. Steele, E. J. Sundstrom, H. L. Woodcock, III, P. M. Zimmerman, D. Zuev, B. Albrecht, E. Alguire, B. Austin, G. J. O. Beran, Y. A. Bernard, E. Berquist, K. Brandhorst, K. B. Bravaya, S. T. Brown, D. Casanova, C. M. Chang, Y. Chen, S. H. Chien, K. D. Closser, D. L. Crittenden, M. Diedenhofen, R. A. DiStasio, Jr., H. Do, A. D. Dutoi, R. G. Edgar, S. Fatehi, L. Fusti-Molnar, A. Ghysels, A. Golubeva-Zadorozhnaya, J. Gomes, M. W. D. Hanson-Heine, P. H. P. Harbach, A. W. Hauser, E. G. Hohenstein, Z. C. Holden, T. C. Jagau, H. Ji, B. Kaduk, K. Khistyayev, J. Kim, J. Kim, R. A. King, P. Klunzinger, D. Kosenkov, T. Kowalczyk, C. M. Krauter, K. U. Laog, A. Laurent, K. V. Lawler, S. V. Levchenko, C. Y. Lin, F. Liu, E. Livshits, R. C. Lochan, A. Luenser, P. Manohar, S. F. Manzer, S. P. Mao, N. Mardirossian, A. V. Marenich, S. A. Maurer, N. J. Mayhall, C. M. Oana, R. Olivares-Amaya, D. P. O'Neill, J. A. Parkhill, T. M. Perrine, R.

- Peverati, P. A. Pieniazek, A. Prociuk, D. R. Rehn, E. Rosta, N. J. Russ, N. Sergueev, S. M. Sharada, S. Sharma, D. W. Small, A. Sodt, T. Stein, D. Stuck, Y. C. Su, A. J. W. Thom, T. Tsuchimochi, L. Vogt, O. Vydrov, T. Wang, M. A. Watson, J. Wenzel, A. White, C. F. Williams, V. Vanovschi, S. Yeganeh, S. R. Yost, Z. Q. You, I. Y. Zhang, X. Zhang, Y. Zhou, B. R. Brooks, G. K. L. Chan, D. M. Chipman, C. J. Cramer, W. A. Goddard, III, M. S. Gordon, W. J. Hehre, A. Klamt, H. F. Schaefer, III, M. W. Schmidt, C. D. Sherrill, D. G. Truhlar, A. Warshel, X. Xu, A. Aspuru-Guzik, R. Baer, A. T. Bell, N. A. Besley, J. D. Chai, A. Dreuw, B. D. Dunietz, T. R. Furlani, S. R. Gwaltney, C. P. Hsu, Y. Jung, J. Kong, D. S. Lambrecht, W. Z. Liang, C. Ochsenfeld, V. A. Rassolov, L. V. Slipchenko, J. E. Subotnik, T. Van Voorhis, J. M. Herbert, A. I. Krylov, P. M. W. Gill, M. Head-Gordon, *Mol. Phys.* **2015**, *113*, 184.
- [14] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, *ACM Trans. Math. Softw.* **2002**, *28*, 135.
- [15] H.-J. Werner, P. J. Knowles, G. Knizia, F. R. Manby, M. Schütz, *WIREs Comput. Mol. Sci.* **2012**, *2*, 242.
- [16] J. M. Turney, A. C. Simmonett, R. M. Parrish, E. G. Hohenstein, F. A. Evangelista, J. T. Fermann, B. J. Mintz, L. A. Burns, J. J. Wilke, M. L. Abrams, N. J. Russ, M. L. Leininger, C. L. Janssen, E. T. Seidl, W. D. Allen, H. F. Schaefer, R. A. King, E. F. Valeev, C. D. Sherrill, T. D. Crawford, *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2012**, *2*, 556.
- [17] J. F. Stanton, J. Gauss, M. E. Harding, P. G. Szalay. CFour. with contributions from A. A. Auer, R. J. Bartlett, U. Benedikt, C. Berger, D. E. Bernholdt, Y. J. Bomble, L. Cheng, O. Christiansen, M. Heckert, O. Heun, C. Huber, T.-C. Jagau, D. Jonsson, J. Jusélius, K. Klein, W. J. Lauderdale, F. Lipparini, D. A. Matthews, T. Metzroth, L. A. Mück, D. P. O'Neill, D. R. Price, E. Prochnow, C. Puzzarini, K. Ruud, F. Schiffmann, W. Schwalbach, C. Simmons, S. Stopkowitz, A. Tajti, J. Vázquez, F. Wang, J. D. Watts; and the integral packages MOLECULE (J. Almlöf and P.R. Taylor), PROPS (P.R. Taylor), ABACUS (T. Helgaker, H.J.Aa. Jensen, P. Jørgensen, and J. Olsen), and ECP routines by A.V. Mitin and C. van Wüllen. For the current version, see <http://www.cfour.de>, accessed on July 20, **2016**.
- [18] E. Epifanovsky, M. Wormit, T. Kuš, A. Landau, D. Zuev, K. Khistyayev, P. Manohar, I. Kaliman, A. Dreuw, A. I. Krylov, *J. Comput. Chem.* **2013**, *34*, 2293.
- [19] E. Solomonik, D. Matthews, J. Hammond, J. Demmel. Cyclops tensor framework: reducing communication and eliminating load imbalance in massively parallel contractions. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, MA, **2013**.
- [20] J. A. Calvin, E. F. Valeev. TiledArray: A general-purpose scalable block-sparse tensor framework. Available at: <https://github.com/ValeevGroup/tiledarray>, accessed on July 20, **2016**.
- [21] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, **2015**. Available at: <https://www.tensorflow.org>, accessed on July 20, 2016.
- [22] T. L. Windus, J. A. Pople, *Int. J. Quantum Chem.* **1995**, *56*, 485.
- [23] I. Kaliman, Libxm tensor contraction code, **2016**. Available at: <https://github.com/ilyak/libxm>, accessed on July 20, 2016.
- [24] S. Hirata, *J. Phys. Chem. A* **2003**, *107*, 9887.
- [25] K. Z. Ibrahim, E. Epifanovsky, S. W. Williams, A. I. Krylov, Cross-scale efficient tensor contractions for coupled cluster computations through multiple programming model backends. Technical Report LBNL-1005853, Lawrence Berkeley National Lab, **2016**. Available at: <https://publications.lbl.gov/islandora/object/ir:1005853>.
- [26] Y. Jung, A. Sodt, P. M. W. Gill, M. Head-Gordon, *Proc. Nat. Acad. Sci. USA* **2005**, *102*, 6692.
- [27] F. Weigend, M. Kattannek, R. Ahlrichs, *J. Chem. Phys.* **2009**, *130*, 164106.
- [28] N. H. F. Beebe, J. Linderberg, *Int. J. Quantum Chem.* **1977**, *12*, 683.
- [29] F. Aquilante, T. B. Pedersen, R. Lindh, *Theor. Chem. Acc.* **2009**, *124*, 1.
- [30] E. Epifanovsky, D. Zuev, X. Feng, K. Khistyayev, Y. Shao, A. I. Krylov, *J. Chem. Phys.* **2013**, *139*, 134105.
- [31] A. Dreuw, M. Wormit, *WIREs Comput. Mol. Sci.* **2015**, *5*, 82.
- [32] K. B. Bravaya, E. Epifanovsky, A. I. Krylov, *J. Phys. Chem. Lett.* **2012**, *3*, 2726.
- [33] T. L. Barr, E. R. Davidson, *Phys. Rev. A* **1970**, *1*, 644.
- [34] C. Sosa, J. Geertsen, G. W. Trucks, R. J. Bartlett, *Chem. Phys. Lett.* **1989**, *159*, 148.
- [35] A. G. Taube, R. J. Bartlett, *Collect. Czech. Chem. Commun.* **2005**, *70*, 837.
- [36] A. Landau, K. Khistyayev, S. Dolgikh, A. I. Krylov, *J. Chem. Phys.* **2010**, *132*, 014109.
- [37] C. Walter. Kryder's law. Available at: <http://www.scientificamerican.com/article/kryders-law>, accessed on July 20, **2016**.

Received: 23 August 2016
Revised: 3 November 2016
Accepted: 3 December 2016
Published online in Wiley Online Library