Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200058

DBCSR: A Blocked Sparse Tensor Algebra Library

Ilia SIVKOV [a,1], Patrick SEEWALD [a,2], Alfio LAZZARO [a,3], and Jürg HUTTER [a,4]
[a] *University of Zurich, Department of Chemistry, Switzerland*

Abstract. Advanced algorithms for large-scale electronic structure calculations are mostly based on processing multi-dimensional sparse data. Examples are sparse matrix-matrix multiplications in linear-scaling Kohn-Sham calculations or the efficient determination of the exact exchange energy. When going beyond mean field approaches, e.g. for Moller-Plesset perturbation theory, RPA and Coupled-Cluster methods, or the GW methods, it becomes necessary to manipulate higher-order sparse tensors. Very similar problems are also encountered in other domains, like signal processing, data mining, computer vision, and machine learning. With the idea that the most of the tensor operations can be mapped to matrices, we have implemented sparse tensor algebra functionalities in the frames of the sparse matrix linear algebra library DBCSR (Distributed Block Compressed Sparse Row). DBCSR has been specifically designed to efficiently perform blocked-sparse matrix operations, so it becomes natural to extend its functionality to include tensor operations. We describe the newly developed tensor interface and algorithms. In particular, we introduce the tensor contraction based on a fast rectangular sparse matrix multiplication algorithm.

Keywords. sparse matrix-matrix multiplications, sparse tensor algebra, multi-threading, MPI parallelization, accelerators

1. Introduction

Most, if not all the modern scientific simulation packages utilize matrix algebra operations. Often, due to the nature of simulated systems, the structure of matrices and tensors is sparse with a low degree of nonzero elements ($< 10\%$). Applications exploiting the sparsity include the linear scaling density functional theory [1], cubic scaling RPA algorithm and a similar approach to fast, quadratic scaling Hartree-Fock exchange [2] in the quantum chemistry CP2K framework [3]. The first method works with sparse matrices, while the other two algorithms rely on contractions involving sparse 3-rank tensors. Due to the nature of the studied chemical systems, this naturally leads to a blocked sparsity pattern, with chemically motivated block sizes. Therefore, the implementation of such methods requires convenient and effective tools and libraries to work also with block-sparse matrices and tensors, with a range of occupancy between 0.01% up to dense.

[1] E-mail: ilia.sivkov@chem.uzh.ch
[2] E-mail: patrick.seewald@chem.uzh.ch
[3] E-mail: alazzaro@cray.com, now at Cray Switzerland GmbH, Switzerland
[4] E-mail: hutter@chem.uzh.ch

The highly optimized linear algebra library DBCSR (Distributed Block Compressed Sparse Row) has been specifically designed to efficiently perform block-sparse and dense matrix operations, covering a range of occupancy between 0.01% up to dense. It is parallelized using MPI and OpenMP, and can exploit GPU accelerators using CUDA. The more detailed description of these features can be found in the previous works [4,5,6]. Here we give an overview of the library in section 2.

Although DBCSR supports multiplications of rectangular matrices, the implemented algorithm was inefficient whenever the resulting matrix has a much smaller size than input matrix sizes ($<$ 1000 smaller). This matrix multiplication can be used for the realization of tensor contraction since the tensor contraction can be mapped to matrix-matrix multiplications [7]. In section 3 we present an optimized implementation for such a case. Additionally, we have developed the tensor algebra operations as an extension of the DBCSR library. In section 4 we present an overview of the new functionalities. The main operation which is used in tensor algebra is a contraction between two tensors over a set of indices. In many of methods, the rank of tensors is no more than 4 and therefore the non-trivial contractions can be performed over 1-3 indices. Finally, section 5 reports the conclusion.

1.1. Related Work

Other implementations of tensor libraries are described in Ref. [8,9,10,11,12], while Ref. [13] presents an overview of tensor algebra applications. The proposed tensor library implementation in DBCSR differs from these implementations since it is specifically targeting block-sparse tensor contractions with a wide range of occupancy between $0.01-10\%$ by optimally exploiting block sparsity. Existing parallel sparse tensor libraries have limited parallel scalability [10], do not prove to be more efficient compared to the dense case [8], or have low sequential efficiency [9]. For matrix-matrix multiplications, the DBCSR library already provides an efficient and scalable solution without the above-mentioned shortcomings.

2. The DBCSR Library

DBCSR is written in Fortran and is freely available under the GPL license from https://github.com/cp2k/dbcsr. DBCSR matrices are stored in a blocked compressed sparse row (CSR) format distributed over a two-dimensional grid of P MPI processes. Matrix-matrix multiplication is implemented by means of the Cannon algorithm [14]. As part of this work, two novel implementations are specifically introduced for rectangular matrix multiplications similar to one iteration of CARMA algorithm [15] (see section 3) and for the tensor contraction algorithm (see section 4). The latter uses the same idea as for the rectangular matrix multiplication with a slightly different implementation.

In the Cannon algorithm, only the matrices A and B are communicated for the multiplication $C = C + A \times B$. The amount of communicated data by each process scales as $\mathscr{O}(1/\sqrt{P})$. These communications are implemented with asynchronous point-to-point MPI calls, using the MPI Funneled mode [6]. The local multiplication will start as soon as all the data has arrived at the destination process, and it is possible to overlap the local computation with the communication if the network allows that.

The local computation consists of pairwise multiplications of small dense matrix blocks, with dimensions $(m \times k)$ for A blocks and $(k \times n)$ for B blocks. It employs a cache-oblivious matrix traversal to fix the order in which matrix blocks need to be computed, in order to improve memory locality. First, the algorithm loops over A matrix row-blocks and then, for each row-block, over B matrix column-blocks. Then, the corresponding multiplications are organized in batches. Multiple batches can be computed in parallel on the CPU by means of OpenMP threads or alternatively executed on a GPU. A static assignment of batches with a given A matrix row-block to threads is employed in order to avoid race conditions. Processing the batches has to be highly efficient. For this reason, specific libraries were developed that outperform vendor BLAS libraries, namely `LIBCUSMM` for GPU and `LIBXSMM` for CPU/KNL systems [16,17].

For GPU execution, data is organized in such a way that the transfers between the host and the GPU are minimized. A double-buffering technique, based on CUDA streams and events, is used to maximize the occupancy of the GPU and to hide the data transfer latency [5]. When the GPU is fully loaded, the computation may be simultaneously done on the CPU. `LIBCUSMM` employs an auto-tuning framework in combination with a machine learning model to find optimal parameters and implementations for each given set of block dimensions. For a multiplication of given dimensions (m,n,k), `LIBCUSMM`'s CUDA kernels are parametrized over 7 parameters, affecting:

- algorithm (different matrix read/write strategies)
- amount of work and number of threads per CUDA block
- number of matrix element computed by one CUDA thread
- tiling sizes

yielding $\approx$ 30,000 - 150,000 possible parameter combinations for each of about $\approx$ 75,000 requestable (m,n,k)-kernels. These parameter combinations result in vastly different performances. We use machine learning to derive a performance model from a subset of tuning data that accurately predicts performance over the complete kernel set. The model uses regression trees and hand-engineered features derived from the matrix dimensions, kernel parameters, and GPU characteristics and constraints. To perform the multiplication the library uses Just-In-Time (JIT) generated kernels or dispatches the already generated code. In this way, the library can achieve a speedup in the range of 2–4x with respect to batched DGEMM in cuBLAS.

`DBCSR` operations include sum, dot product, and multiplication of matrices, and the most important operations on single matrices, such as transpose and trace. Additionally, the library includes some of the linear algebra methods, such as the sign matrix algorithm [1] and matrix inverse. These methods were ported from `CP2K` to `DBCSR`. The sign matrix algorithm is used in the linear scaling density functional theory in order to find a ground state of the quantum systems. As associated methods, we have ported the matrix-vector multiplication operation and an interface to some SCALAPACK operations.

3. Optimized Rectangular Matrix Multiplication Algorithm and Implementation

Despite the Cannon algorithm gives in general good performance for the sparse matrix multiplication of any size, it loses its efficiency in the case where the size of the resulting matrix C ($S_C = O_C MN$) is much smaller than the sizes of the input A ($S_A = O_A MK$)

and/or B ($S_B = O_B KN$) matrices, where M, N, K are the dimensions of the dense matrices and O_A, O_B, O_C their occupancy values. This is a direct consequence of the algorithm since it requires the communication of A and B data on a 2D grid of P processors, while C remains local to each processor. In particular, for the multiplication of two rectangular matrices the Cannon algorithm requires to communicate per each processor [5]

$$T_w = \frac{S_A + S_B}{\sqrt{P}} = \frac{K(O_A M + O_B N)}{\sqrt{P}}. \tag{1}$$

Therefore, the communication will be dominated by one of the dimension whenever is much larger than the other two. We can distinguish the two important cases:

1. $M \ll K$ and $N \ll K$, which corresponds to $S_C \ll \{S_A, S_B\}$
2. $K \ll M$ and $K \leq N$, which corresponds to $S_B \ll \{S_A, S_C\}$

According to Ref. [18], a communication-optimal algorithm for this case is obtained by dividing the original matrix multiplication into smaller tasks such that each task is local to a process subgroup. Inspired by this idea, we redistribute the matrices on a linear MPI grid (see Figure 1) and perform the multiplication locally. We describe the implementations for the two cases in the following subsections. We also report the results of some tests we performed for a variety of matrix, block sizes and occupancy values of our interests ($10\% - 50\%$ often present in CP2K). We used double precision matrices with sizes of the order $M, N = \mathcal{N}$, $K = \mathcal{N}^2$ and $\mathcal{N} = 10^3$. The calculations were performed using the Cray XC50 "Piz Daint" supercomputer at the Swiss National Supercomputing Centre (CSCS). Each node of the system is equipped by a CPU Intel Xeon E5-2690 v3 @ 2.60GHz (12 cores, 64GB DRAM) and a GPU NVIDIA Tesla P100 (16GB HBM). For the MPI configuration, we used 1 rank per node and 12 OpenMP threads per rank. Each multiplication was performed 100 times to exclude the fluctuations of performance due to hardware glitches.

3.1. $S_C \ll \{S_A, S_B\}$

Matrices A and B are redistributed on a linear MPI grid and the A matrix is transposed, such as the longest dimension K is now distributed over the P processors (see Figure 1a). Then a local multiplication is executed, which gives $\widetilde{C}_i = A_i^T \cdot B_i$, with $i = 1, ..., P$. Here $\widetilde{C}_i$ corresponds to a partial result of the full, undistributed, matrix C. Therefore we have to reduce all $\widetilde{C}_i$ and redistribute the result according to the original 2D grid distribution and sum to the input C matrix to get the final distributed C matrix result over the 2D grid (see Figure 2). This algorithm runs in P steps, where for each step we send and receive the proper C data and run the local reduction. It is implemented with MPI asynchronous communications, such as we do overlap the communication of the data with the local reduction. In the end, each processor requires S_C data. Including the initial redistribution of the A and B matrices, we get that the total amount of data communicated by each processor is:

$$T'_w = \overbrace{\left(\frac{S_A + S_B}{P}\right)}^{\text{2D} \rightarrow \text{1D grid}} + S_C. \tag{2}$$

[5] Here we assume a uniform distribution of the non-zero elements in the matrices without losing generality.

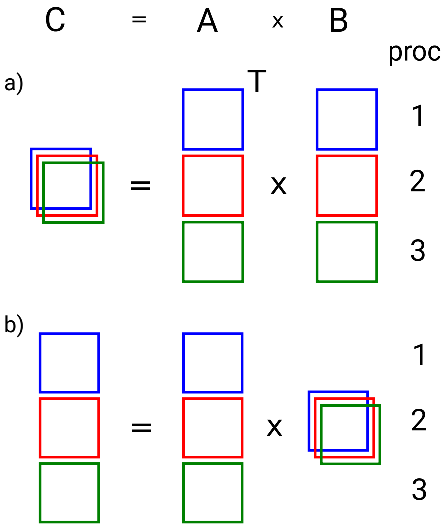

Figure 1. Communication-avoiding algorithm for the rectangular matrix-matrix multiplication. a) Middle dimension K is the largest (case 1), C is replicated and A and B are distributed on a linear grid. b) Outer dimension M is the largest (case 2), C and A are distributed on a linear grid and B is cloned or distributed.

We can now consider the ratio with the Cannon algorithm (Eq. 1), which leads to a reduction in communicated data $\sqrt{P}/(1+RP)$, where $R = S_C/(S_A+S_B)$. Therefore, the ratio scales as $\mathscr{O}(1/\sqrt{P})$. Finally, it is important to note that by multiplication of the sparse matrices even with high sparsity the result might be dense (so called Birthday Paradox [19]). We can evaluate an upper limit on the O_C by combining the Eq. 1 and Eq. 2 such that $T_w < T'_w$:

$$O_C < \frac{1}{MN}\left(T_w - \frac{S_A+S_B}{P}\right) \tag{3}$$

If we omit redistribution costs and assume that $O_A = O_B = O$ then we can write:

$$\frac{O_C}{O} < \frac{K(M+N)}{MN\sqrt{P}}. \tag{4}$$

As an example, for $M,N = \mathscr{N}$, $K = \mathscr{N}^2$ and $\mathscr{N} = 10^3$, $P = 100$ we get $O_C/O < 2\cdot 10^2$.

The results of the tests are presented in Figure 3a. Overall, the new implementation gives a speed-up up to 3x with respect to the Cannon algorithm for high occupancy ($> 10\%$), which becomes negligible when we reach the upper limit reported in the Eq. 3. As expected, the speed-up decreases with the number of processors.

3.2. $S_B \ll \{S_A, S_C\}$

Matrices A and B are redistributed in a linear MPI grid, such as the longest dimension K is now distributed over the P processors (see Figure 1b). A virtual column-grid is created for the A matrix to be compatible with the row-grid of the matrix B. Then the standard Cannon algorithm is executed over this virtual topology made of P steps. Virtual column-

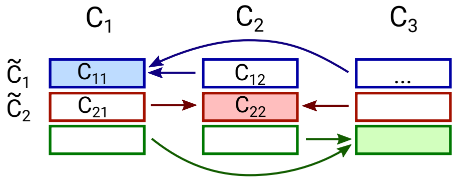

Figure 2. Reduce operation of the matrix C after the local multiplication when $S_C \ll \{S_A, S_B\}$.

grid does not require communication of the A data and therefore only the communication of the matrix B is required. Finally, C result is redistributed to the original 2D grid and accumulated to the input C matrix. The total amount of communicated data by each processor is:

$$T''_w = \overbrace{\left(\frac{S_A + S_B + S_C}{P}\right)}^{\text{2D} \rightarrow \text{1D} \rightarrow \text{2D grid}} + S_B. \tag{5}$$

Also in this case the ratio of the communicated data with respect to Cannon implementation scales as $\mathcal{O}(1/\sqrt{P})$.

The results of the tests are presented in Figure 3b. Overall, the new implementation gives a speed-up of up to 20% with respect to the Cannon algorithm for high occupancy ($> 10\%$) or up to 100% for the matrices close to dense ($\sim 50\%$). The time for the redistribution and the overhead introduced by the virtual grid creation limits the benefit of the new implementation. For the same reasons, the benefit of the new algorithm is negligible or even worse for low occupancy.

4. Sparse Tensor Algebra Implementation

DBCSR was originally developed to enable linear scaling electronic structure methods that are mainly based on the multiplication of sparse square matrices. Similar strategies employing sparse data can also be employed for methods beyond density functional theory that provide better accuracy at significantly higher computational costs than Kohn-Sham density functional theory. In the case of the electron correlation methods MP2 and RPA, the canonical implementation scales at least quartic with system sizes, thereby preventing the study of large systems (hundreds to thousands of atoms). An initial DBCSR-based cubic scaling implementation of RPA was reported in Ref. [2], enabling calculations of thousands of atoms. Here we report strategies to optimize and generalize this initial implementation by extending the DBCSR library to multi-dimensional tensors. A generalized implementation of tensor operations in DBCSR instead of specialized implementations in the application code is desirable to manage code complexity and to easily extend the current implementation to other methods such as Hartree-Fock exchange or GW. The formalism of our RPA implementation was already described in Ref. [2] and here we emphasize the general characteristics of the tensor operations appearing in this and similar methods. We describe the requirements we pose for a tensor framework that should provide all relevant tensor operations in a general API.

As in DBCSR, the sparsity of the tensors is based on the representation of molecular orbitals in terms of a localized atom-centered basis. A blocked sparsity pattern is

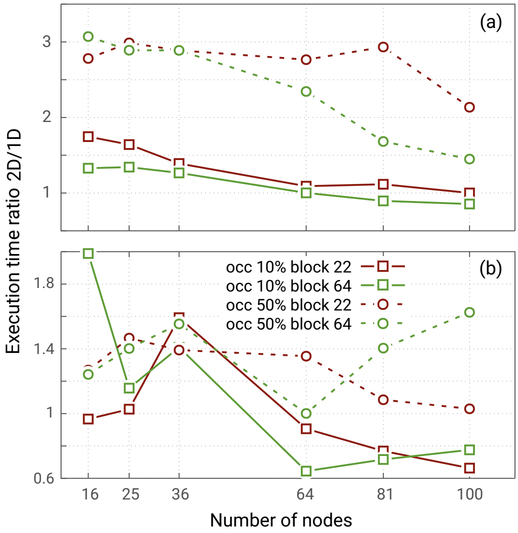

Figure 3. Execution time ratios for two types of considered rectangular matrix multiplications on a 1D grid in comparison with the regular 2D Cannon algorithm. (a) The first type, $S_C \ll \{S_A, S_B\}$, shows significant improvement for higher occupancy (50%) and less pronounced for the lower one (10%). (b) The second type, $S_B \ll \{S_A, S_C\}$, shows moderate to high speedup for higher occupancy (50%) and poor to moderate behavior for the lower occupancy (10%). In both cases, the benefit of the implementation reduces with the number of nodes, as expected.

equally important for the tensor implementation to efficiently incorporate sparse data while keeping the significant overhead for the handling of sparse indices low. Tensors can have arbitrary ranks, most relevant are tensors with ranks between 2 and 4. The main operation is tensor contraction where a sum over one or more indices of two tensors is performed. Our implementation is based on the property that tensor contractions are isomorphic to matrix-matrix multiplications [7]. This allows us to implement tensor contraction by mapping tensors to matrices – the contraction is then internally performed by a call to the existing implementation of sparse matrix-matrix multiplication.

Recasting tensor contraction in terms of matrix-matrix multiplication imposes some requirements on the distribution and the matrix representation of the tensors, most importantly that one matrix dimension represents the indices to sum over and the other matrix dimension represents all other indices. If these requirements are not met, conversion steps are required before and after matrix-matrix multiplication which involves the redistribution of all tensor data. In order to avoid these relatively expensive redistribution steps, the tensor API gives the caller tight control over the distribution and the matrix representation of tensors, such that tensors can be created in a compatible layout and the redistribution step can be skipped in a tensor contraction. Data redistribution is then only strictly needed if a tensor appears in multiple contraction expressions involving sums over different indices.

While this approach of mapping tensors to matrices allows for an implementation of tensor operations as thin layers around an existing matrix library, the resulting matrices are problematic since one dimension is much larger than the other dimension. For the example of a 3 rank tensor with size $N \times N \times N$, two tensor dimensions are mapped to one matrix dimension such that one matrix dimension grows quadratically with the size of the other dimension. The DBCSR library must thus be extended in a way that it can efficiently store and multiply tall-and-skinny matrices contrary to the previous target of square matrices.

One limitation of the DBCSR matrix format is the index data replicated on all MPI ranks which contain information about block sizes and the distribution of blocks along each of the matrix dimensions. If the size of the matrix index corresponds to the number of atoms N in a system, this limits the scalability of DBCSR to a few tens of millions of particles [1]. For 3-rank tensors where the largest matrix dimension grows as N^2, this limit is already hit at a few thousand atoms, representing a much bigger issue in practice. Thus an extension to the DBCSR matrix format must be provided to store large tensors without exhausting memory due to replicated index data. Another challenge is to multiply tall-and-skinny matrices communication-efficiently, where the algorithm described in section 3 comes into play.

Our requirements for memory-efficient storage and communication-efficient multiplication can both be met by dividing the largest matrix dimension, resulting in smaller and approximately square submatrices that can be handled by DBCSR. The storing of the full matrix index and the multiplication acting on submatrices are managed by an in-between tall-and-skinny matrix layer on top of DBCSR that serves as a basis for the tensor implementation. The tall-and-skinny matrix layer is designed in a way that the index data is not explicitly stored but provided by externally defined function objects, to avoid the above-mentioned limitation of the DBCSR format. The matrix index is thus handled in the tensor layer and is calculated on the fly from the tensor index. Due to the fully distributed sparse data layout, the matrix index calculation happens only when accessing a locally present non-zero block and does not add any overhead.

The main difference between the implementation of tall-and-skinny matrix multiplication and the one implemented in DBCSR internally (see section 3) is that instead of relying on a linear process grid, the grid may have arbitrary dimensions. The submatrices are obtained on MPI subgroups by dividing any of the two grid dimensions by an arbitrary factor. This ensures that an optimal split factor can always be chosen, independently of the total number of processes, for any grid dimensions. Thus n-rank tensors can be represented on an arbitrary n-dimensional process grids where the grid dimension should be chosen as balanced as possible for a load-balanced distribution of data. The multiplication algorithm for contraction can then be run directly without additional costly redistribution steps (for tall-and-skinny matrices, the bandwidth cost of redistributing data exceeds the bandwidth cost for the multiplication [15]).

5. Conclusion

We have presented a new implementation for the rectangular matrix-matrix multiplication algorithm in the DBCSR library that is able to speed-up the execution up to 3x for matrix sizes and occupancy values of $10\% - 50\%$ which are often present in CP2K calculations. We have described the newly developed tensor operations that generalize the DBCSR library to multidimensional tensor contraction for low-scaling electronic structure methods beyond density functional theory. These functionalities are the basic building block for the CP2K quantum chemistry and solid-state physics software package.

Acknowledgments

We thank Shoshana Jakobovits (CSCS Swiss National Supercomputing Centre) for her work on the LIBCUSMM code. This work was supported by grants from the Swiss National Supercomputing Centre (CSCS) under projects S238 and UZHP and received funding from the Swiss University Conference through the Platform for Advanced Scientific Computing (PASC).

References

[1] Joost VandeVondele, Urban Borstnik, and Juerg Hutter. Linear scaling self-consistent field calculations for millions of atoms in the condensed phase. *The Journal of Chemical Theory and Computation*, 8(10):3565–3573, 2012.

[2] Jan Wilhelm, Patrick Seewald, Mauro Del Ben, and Juerg Hutter. Large-scale cubic-scaling RPA correlation energy calculations using a Gaussian basis. *Journal of Chemical Theory and Computation*, 2016. `http://dx.doi.org/10.1021/acs.jctc.6b00840`.

[3] Juerg Hutter, Marcella Iannuzzi, Florian Schiffmann, and Joost VandeVondele. CP2K: Atomistic Simulations of Condensed Matter Systems. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 4(1):15–25, 2014.

[4] Urban Borstnik, Joost VandeVondele, Valery Weber, and Juerg Hutter. Sparse Matrix Multiplication: The Distributed Block-Compressed Sparse Row Library. *Parallel Computing*, 40(5-6):47–58, 2014.

[5] Ole Schütt, Peter Messmer, Juerg Hutter, and Joost VandeVondele. GPU Accelerated Sparse Matrix Matrix Multiplication for Linear Scaling Density Functional Theory. In *Electronic Structure Calculations on Graphics Processing Units*. John Wiley and Sons, 2015.

[6] Alfio Lazzaro, Joost VandeVondele, Jürg Hutter, and Ole Schütt. Increasing the Efficiency of Sparse Matrix-Matrix Multiplication with a 2.5D Algorithm and One-Sided MPI. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '17, pages 3:1–3:9, New York, NY, USA, 2017. ACM.

[7] Edgar Solomonik, James Demmel, and Torsten Hoefler. Communication lower bounds for tensor contraction algorithms. Technical report, ETH-Zürich, 2015.

[8] Cannada A Lewis, Justus A Calvin, and Edward F Valeev. Clustered Low-Rank Tensor Format: Introduction and Application to Fast Construction of Hartree-Fock Exchange. *arXiv preprint arXiv:1510.01156*, 2015.

[9] Edgar Solomonik and Torsten Hoefler. Sparse Tensor Algebra as a Parallel Programming Model. *arXiv preprint arXiv:1512.00066*, 2015.

[10] Evgeny Epifanovsky, Michael Wormit, Tomasz Ku, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I. Krylov. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of Computational Chemistry*, 34(26):2293–2309, 2013.

[11] Samyam Rajbhandari, Akshay Nikam, Pai-Wei Lai, Kevin Stock, Sriram Krishnamoorthy, and P Sadayappan. Framework for distributed contractions of tensors with symmetry. *Preprint, Ohio State University*, 2013.

[12] Walter Landry. Implementing a High Performance Tensor Library. *Scientific Programming*, 11(4):273–290, December 2003.

[13] Tamara G. Kolda and Brett W. Bader. Tensor Decompositions and Applications. *SIAM Rev.*, 51(3):455–500, August 2009.

[14] Lynn Elliot Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[15] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 261–272, Washington, DC, USA, 2013. IEEE Computer Society.

[16] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *Proceedings of the International Confer-*

ence for High Performance Computing, Networking, Storage and Analysis, SC '16, pages 84:1–84:11, Piscataway, NJ, USA, 2016. IEEE Press.

[17] Iain Bethune, Andreas Glöss, Jürg Hutter, Alfio Lazzaro, Hans Pabst, and Fiona Reid. Porting of the DBCSR Library for Sparse Matrix-Matrix Multiplications to Intel Xeon Phi Systems. In *Advances in Parallel Computing, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*, pages 47–56, 2017.

[18] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 261–272. IEEE, 2013.

[19] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.