Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure

Thomas Herault, Yves Robert, George Bosilca, Robert J Harrison, Cannada A Lewis, Edward F Valeev, Jack J Dongarra

▶ **To cite this version:**

Thomas Herault, Yves Robert, George Bosilca, Robert J Harrison, Cannada A Lewis, et al.. Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure. IPDPS 2021 - IEEE International Parallel and Distributed Processing Symposium, May 2021, Portland, OR, United States. pp.1-10, 10.1109/IPDPS49936.2021.00062 . hal-03508930

HAL Id: hal-03508930
https://inria.hal.science/hal-03508930

Submitted on 3 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure

Thomas Herault*, Yves Robert*†, George Bosilca*, Robert J. Harrison‡,
Cannada A. Lewis§, Edward F. Valeev¶, Jack J. Dongarra*
*ICL, University of Tennessee, TN, USA. {herault,yrobert,bosilca,dongarra}@icl.utk.edu
†ENS Lyon, France ‡IACS, Stony Brook University, NY, USA. robert.harrison@stonybrook.edu
§Sandia Ntl. Lab., CA, USA. canlewi@sandia.gov ¶Dept. of Chemistry, Virignia Tech, VA, USA. valeev76@vt.edu

***Abstract*—Many domains of scientific simulation (chemistry, condensed matter physics, data science) increasingly eschew dense tensors for block-sparse tensors, sometimes with additional structure (recursive hierarchy, rank sparsity, etc.). Distributed-memory parallel computation with block-sparse tensorial data is paramount to minimize the time-to-solution (*e.g.,* to study dynamical problems or for real-time analysis) and to accommodate problems of realistic size that are too large to fit into the host/device memory of a single node equipped with accelerators. Unfortunately, computation with such irregular data structures is a poor match to the dominant imperative, bulk-synchronous parallel programming model. In this paper, we focus on the critical element of block-sparse tensor algebra, namely binary tensor contraction, and report on an efficient and scalable implementation using the task-focused PaRSEC runtime. High performance of the block-sparse tensor contraction on the Summit supercomputer is demonstrated for synthetic data as well as for real data involved in electronic structure simulations of unprecedented size.**

***Index Terms*—electronic structure, tensor contraction, block-sparse matrix multiplication, distributed memory, multi-GPU nodes, PaRSEC.**

I. Introduction

The current path to exascale computing relies on an extensive use of accelerators. As of today, the Summit and Sierra systems [1] are number 2 and 3 on the TOP500 list [2]. Both systems are distributed-memory platforms where each node is equipped with several high performance NVIDIA accelerators. For instance Summit nodes include 6 NVIDIA V100 GPUs, interconnected at the node level by multiple NVLinks. The forthcoming Frontier exascale system [1] is announced with four AMD Radeon GPUs per node. On Summit, more than 97% of the overall compute performance is on the GPU side. The emerging trend remains consistent across all state-of-the-art platforms equipped with accelerated nodes: these machines draw most of their computing power out of the accelerators; hence, it is crucial, for any efficient and scalable algorithm, to be able to extract the most performance out of the accelerators to achieve high overall efficiency.

The existence of highly capable hardware only translates in application performance if software support exists. The community effort is well on its way to implement dense linear algebra libraries for multi-GPU accelerated nodes. Several ongoing projects aim at designing dense linear algebra kernels, not only to achieve high TOP500 performance, but to allow a broad range of applications to benefit from the computing power lying in the accelerators. While most projects are conducted by vendors (Intel, AMD, NVIDIA, Cray), some academic projects, such as SLATE [3], are publicly available, and provide efficient CPU or GPU implementations for most traditional dense linear routines. Recently, support for a limited number of operations in a multiple-accelerator setting has been added, with some matrix-size constraints. For instance the current matrix product $C = A \times B$ is limited to problems where the entire C matrix can reside in the memory of the accelerators. A similar academic effort proposes a distributed multi-accelerators prototype for matrix-matrix multiplication without any size restriction within the PaRSEC task-based runtime system [4].

Achieving good performance for dense linear algebra kernels is only a first step to achieving exascale performance for general scientific applications. This can be seen by looking at the performance discrepancies between two of the most widely used benchmarks in HPC, the HPL (High Performance LINPACK) benchmark used in the Top500 list, and HPCG (High Performance Conjugate Gradient) benchmark, more representative of the behavior of a typical scientific application. On Summit, the performance of HPCG is 50 times lower than that of HPL. This is because HPCG involves a communication-bound kernel with sparse fine-grained computational kernels, as opposed to a computation-bound kernel with dense Level3-BLAS routines for HPL.

This paper aims at complementing the insight gained from the HPCG benchmark by exploring another important and widely used computational kernel in High Performance Computing. We consider how the *binary contraction of block-sparse tensors*, a key paradigmatic operation for a variety of physical simulation and data-science domains, can be implemented efficiently on large-scale distributed-memory multi-GPU accelerated platforms. To assess the performance, we consider a mix of synthetic problem setups and contractions taken from actual simulations of electronic structure of molecules. The binary tensor contraction will be mapped, as is typically done, onto the GEneral Matrix Multiplication (GEMM) $C \leftarrow \alpha AB + \beta C$. While the dense matrix multiplication is a formidable, but manageable, challenge on

This research was supported by the Exascale Computing Project (17-SC-20-SC), and the NSF projects #1931347, #1931384, and #1931387; it used resources of the Oak Ridge Leadership Computing Facility at ORNL, which is supported by the U.S. D.o.E. under Contract No. DE-AC05-00OR22725.

distributed memory heterogeneous platforms for the relevant problem sizes [4], the block-sparse matrix multiplication adds several new challenges. First, the rows and columns of the three matrices are tiled nonuniformly, due to the nonuniform structure of the underlying physical problem. Second, the matrices are block-sparse, with the fill degree greatly varying with the particular simulation from 100% (for high-precision simulation on compact molecules) to a few percent even for modestly-sized simulations. Third, the aspect ratios of the matrices can vary greatly from 1 (square) to 100s (tall-and-skinny, or short-and-wide); the particular paradigmatic example from the electronic structure domain that we focus on, involves a large square matrix B and short-and-wide matrices A and C, with aspect ratios on the order of 100. All these characteristics decrease potential data-reuse and arithmetic intensity, and dramatically complicate the design of an efficient algorithm targeting multi-GPU accelerated nodes. The main contribution of this work is the design of a generic and flexible implementation of this block-sparse kernel, and its analysis on a large multi-GPU platform.

The rest of the paper is organized as follows. Section II surveys the motivating science application. Section III overviews the main design principles of our algorithm. Section IV discusses the main details of the prototype implementation, which is publicly available with all benchmarks used in this work [5]. In Section V, we report preliminary performance results. Section VI briefly discusses related work, before we conclude in Section VII.

II. Motivating Science Application

Our goal is to deploy the distributed memory block-sparse matrix multiplication in the context of electronic structure applications for quantum mechanical simulation of molecules and materials from first principles. Accurate simulation of electronic structure, via the coupled-cluster [6] and many-body Green's function approaches, is feasible but *expensive*, *i.e.,* such *many-body* methods have high-order polynomial operation and space complexity; for the foundational Coupled-Cluster Singles and Doubles method (CCSD), these are N^6 and N^4, respectively, with N proportional to the system size. The high complexity limits the applicability of conventional (naive) formulations of predictive methods to systems with a few (5-10) atoms on a single workstation, and a few dozen (50-100) atoms on a supercomputer [7]. However, the recent emergence of robust fast/reduced-scaling formulations has greatly extended the applicability of such methods to hundreds of atoms on a single workstation in a matter of days [8]. Modern state-of-the-art HPC platforms should make it possible to deploy reduced-scaling coupled-cluster (CC) methods with time-to-solution measured in minutes rather than in days. The complex tensor algebra involved in the CCSD method can be reduced for our purposes to a single representative term, usually the most expensive one (accounting routinely for 90% or more of the work), colloquially known as the *ABCD* term:

$$R_{ab}^{ij} = \sum_{cd} T_{cd}^{ij} V_{ab}^{cd} + \ldots, \tag{1}$$

where the elements of tensor T are the model parameters to be refined iteratively (in typically 10-20 iterations) to make tensor R vanish. Tensor V is fixed (does not change between iterations). Ranges of all indices are proportional to system size N, hence each tensor has N^4 space complexity, and the operation has N^6 operation complexity.

The tensor contraction in Equation (1) can be viewed as a multiplication of matrix T (with fused indices ij and cd playing the role of row and column indices, respectively; in subsequent sections such *matricized* tensor T will serve as matrix A in $C = C + AB$) with square matrix V (with cd and ab row and column indices; this will serve as matrix B). In practice the range of *unoccupied* indices (a, b, c, d) has rank U that is a factor of 5-20 times larger than the corresponding rank O of the *occupied* indices ij, hence transposes of matricized tensors T and R are tall-and-skinny matrices, with aspect ratios of 25-400.

To set the scale for target calculations we consider predictive calculations of the electronic structure of large molecules using many-body theory. Central to this problem is solving a set of coupled non-linear equations for the amplitudes t_{ij}^{ab} in which i, j label one-electron states occupied in a zeroth-order approximation to the wave function and a, b label excited states with about 10 such states per electron. In a fully dense calculation, which remains of interest for calibration and benchmarking, the number of amplitudes grows as the fourth power of the number of electrons. Thus, a calculation on just 1000 electrons exceeds the aggregate memory of all GPUs in Summit just to hold the solution. Intermediates and other quantities multiply the required memory. Reduced scaling calculations significantly reduce the amount of data, but our ambitions extend to systems with at least $O(10^4)$ electrons for which again, predictive calculations with controlled error would greatly exceed available GPU memory.

In the conventional formulation of CCSD, all tensors are generally dense (modulo prefactor-reducing block-sparsity due to discrete geometric symmetries; here we only focus on block-sparsity due to dynamical structure of the physical problem that can lead to the reduction of complexity). The formulation of dense matrix multiplication on distributed-memory systems [9], including for rectangular matrices [10], is relatively well understood and makes possible strongly scalable CCSD implementations [7], [11]. Extending these advances to *reduced-scaling* coupled-cluster variants in which tensors have complex block-sparse structure is nontrivial due to the physically-motivated nonuniform tiling of index ranges (*e.g.,* it is not in general possible to partition the basis into even chunks without sacrificing locality). This leads to the loss of the near-perfect load balance that makes traditional communication-optimal algorithms attain strong scaling. Parallel computation with irregularly-tiled and/or data-sparse tensorial data structures is also a poor match to imperative, bulk-synchronous parallel programming style and execution models due to the irregular (and potentially dynamic) structure of the data. In this work, we demonstrate how these challenges can be addressed by modern task-based dataflow-style scheduling

to achieve high performance on a distributed-memory heterogeneous cluster with multi-GPU nodes. The block-sparse evaluation of the ABCD term in Equation (1) in the so-called atomic orbital formulation will serve as the target performance benchmark; the reference CPU-only implementation of this term was developed in the open-source Massively Parallel Quantum Chemistry (MPQC) program [12].

III. Design Principles

As already mentioned, the problem is generated from a 4-dimension tensor, but can be viewed as a matrix multiplication, $C \leftarrow C + AB$, with the following characteristics:

- The matrices are composed of heterogeneous tiles: the size of the tiles strongly vary across rows and columns. and many of them are too small to provide high computational intensity.
- The matrices are block-sparse. A significant fraction of the tiles in A and B are zero tiles (which opens the possibility for some tiles of C to be zero tiles too). The non-zero tiles are dense, thus efficient dense linear algebra GEMM kernels can be used for the non-zero tile products. Only non-zero tiles are stored in memory.
- The matrices have very different sizes: A and C are short-and-wide, while B is square. More precisely, A has size $M \times K$, B has size $K \times N$, and C has size $M \times N$, where $M \ll K = N$ (typically $N = 100M$). As for tile indices, A has $M^{(t)}$ tile rows (of various heights) and B has $N^{(t)}$ tile columns (of various widths) All these characteristics dramatically complicate the problem.
- As pointed out in Section I, designing an efficient algorithm for matrix multiplication on multi-GPU accelerated distributed memory platforms is already a difficult task when A, B and C are dense and square. Here, the heterogeneity of tile sizes further hardens the management of GPU memory and diminishes the peak performance of the kernels, while the sparsity decreases data reuse across different GEMMs.

The target platform is composed of P processors, or nodes, each equipped with g GPUs. We aim at executing the block-sparse matrix product on a $p \times q$ process grid, where $pq \leq P$. For square and dense matrices, the traditional algorithm uses a square 2D-grid with $p = q$, a 2D-cyclic distribution of the three matrices, and computes C in place while A and B are communicated through the network. The significantly larger size of B in front of that of A and C requires changing the traditional algorithm. In order to minimize network traffic, we need to avoid circulating the largest of the matrices, so B will be stationary. A solution is to distribute full columns of B to processors, meaning that the distribution of B becomes uni-dimensional on a flat $1 \times q$ grid (where $q = P$). Each column of B is then entirely held by a single node, as opposed to partitioned across grid rows. However, this alternative is known to increase the communication volume related to A; this is why 2D-grids are generally preferred for matrix multiplication.

Yet another alternative is to duplicate the columns of B and to use a $p \times q$ processor grid with $p \geq 2$. In this last solution, each grid row computes the product of an horizontal slice of A by the whole matrix B. More precisely, A is segmented into p horizontal slices, and all p grid rows work independently on their own slice, without any communication and in full parallelism. The price to pay is to replicate each column of B p times in memory, one time per grid row, which puts pressure on CPU memory, but not on GPU memory which is the actual bottleneck for the computational perspective. We investigate this last solution and keep the number p of grid rows as a trade-off parameter: using $p = 1$ avoids the replication of B but increases the communication volume of A; using $p \geq 2$ requires p copies of each column of B but decreases the communication volume of A by a factor p.

The algorithm targets a 2D-grid of $p \times q$ processors where p is a parameter and $q = \lfloor \frac{P}{p} \rfloor$, where P is the total number of available processors, so that $pq \leq P$ (see Figure 1). The matrix A is distributed with a standard 2D-cyclic distribution. Let $A^{(k)}$ be the slice of A distributed on row grid number k where $0 \leq k \leq p-1$: $A^{(k)}$ is composed of tile rows of A of index i such that $i \bmod p = k$. Let $C^{(k)}$ be the corresponding slice of C (same row indices as $A^{(k)}$). Row grid number k computes the product $C^{(k)} \leftarrow C^{(k)} + A^{(k)}B$. All these products are independent and are executed in parallel. Therefore, we focus on the description of the algorithm on a single grid row, and keep using A instead of $A^{(k)}$ to ease notations. Recall that A now has $\frac{M^{(t)}}{p}$ tile rows (assume p divides $M^{(t)}$ for simplicity). To ease reading, we will denote the algorithm in terms of rows and columns, but remember that all operations are *tiled*, and we use **row** to denote a *tile row* and **column** to denote a *tile column*. The main operation of the algorithm on a processor row of size $1 \times q$ is the following:

- Assign columns of B to the q processors, and on each processor partition assigned columns into blocks, using the load-balancing algorithm detailed in Section III-1.
- On each processor in parallel, compute the column blocks one after the other. The size of a column block is monitored so that its size does not exceed 50% of a GPU memory. Hence each block will be transferred from the CPU to the GPU only once. See Section III-2 for details.
- The operation within each block is segmented to avoid GPU memory overflow. Communications from CPU to GPU are carefully monitored throughout execution to limit the number of A tiles transferred to GPU, in order to ensure that no tile of B and C is ever flushed back to CPU before all computations involving it, are completed. See Section III-3 for details.

The overhead induced by the algorithm is of the same order as the number of non-zero B tiles, and has a negligible cost on execution. See the companion report [13] for details.

1) Column Assignment: To load-balance the product $C \leftarrow C+AB$, let f_k be the total number of floating point operations (flop) corresponding to column k of B in the product, for $1 \leq k \leq N^{(t)}$, assuming that non-zero tiles are dense. We sort the columns by non-decreasing values of f_k and assign them to the q processors in a mirrored cyclic distribution: the first q columns are assigned to the q processors in that order, and the next q columns are assigned to the q processors in

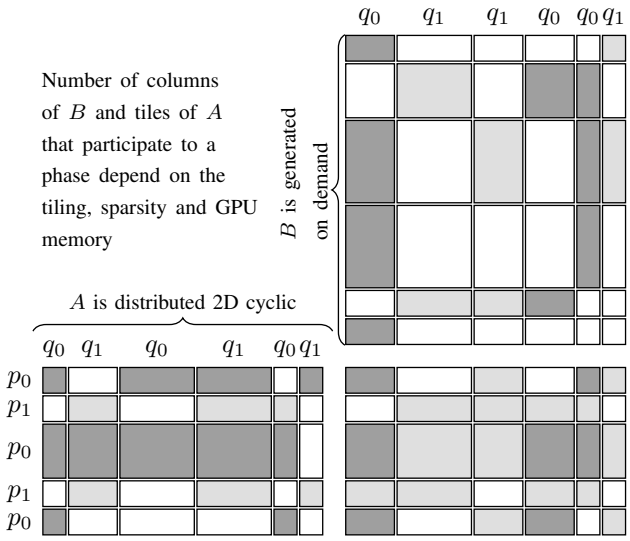


Fig. 1. Representation of a phase of the algorithm for the process at the position ($p = 0, q = 0$) in the 2×2 process grid. Dark grey represent data loaded and used for computations by this process, light grey by other processes.

reverse order, and the process repeats every $2q$ columns. The mirroring (reverse) pass is used to compensate the imbalance due to the initial forward pass.

Let $\mathcal{B}_q$ denote the subset of columns assigned to processor q. This node will be in charge to compute the same columns of the product C. Note that C will therefore follow the same row distribution as A and the same column distribution as B. The assignment algorithm ensures that each node receives a set of columns involving approximately the same amount of floating point operations, at the granularity of the columns of B, aiming at providing a good load-balance of the computations.

2) Partition into Blocks: Once the columns of B have been assigned to the processors, they are divided into blocks which are assigned to GPUs. While the assignment of columns across nodes is intended to load-balance computations, the partitioning into blocks on each node aims at monitoring GPU memory usage. Locally, each processor computes a partition of its columns into blocks whose size fits in half the memory of one GPU. The goal is to enforce that each column of B, together with the local C tiles in that column, will be transferred only once to the GPU. The algorithm sorts local columns (B columns assigned to the node) by non-increasing memory size (volume of data for the column and local C tiles) and allocates these columns in that order to the GPUs, using a worst-fit algorithm. Each GPU starts with an empty block which is filled as the worst-fit algorithm progresses. A new block is created and assigned to a GPU in a round-robin fashion when the current column does not fit anywhere, in order to ensure no GPU is assigned more than one block than any other GPU. During execution, blocks are transferred from CPU to GPUs in sequence: the transfer of the next block cannot start before operations on the current block are completed (although some overlap is made possible by the implementation as a task system, see Section IV). This is to avoid new B tiles flushing out current B tiles still in use, which is critical for performance [4]. Again, the size of a block (including C tiles) is computed so as not to exceed 50% of the GPU memory.

3) Segmentation into Chunks: There remains approximately 50% of GPU memory for A tiles, depending on the space occupied by B and C. How to organize the transfer of A tiles to maximize re-use within a block? Say there are c columns of B in the block. We would like to work with groups of several rows of A in parallel, say r rows, and to segment the transfer of these tiles by chunks of k tiles per row: this mimics the traditional algorithm that maximizes re-use by allowing b chains of GEMMs to progress in parallel (one per column) and enforcing a total of brk GEMMS with only rk transfers of A tiles. The value of the chunk depth k is computed for each new chunk of A so that rk tiles of A fit in the remaining memory of the GPU. Unfortunately, there is no guarantee that such a nice re-use will be achieved for our problem, because of the sparsity pattern of the tiles. It may well be the case that a tile of A is used only once instead of c times in the block, if $c-1$ out of the c potential products involving it are with zero tiles of B. Still, this segmentation should improve re-use, and we implement it into chunks of r rows of A. However, due to the heterogeneity of the tiles, we cannot load k tiles per row any longer; instead, we build chunks greedily by adding one tile per row of A in a cyclic fashion until half the remaining GPU memory, *i.e.,* 25% of total GPU memory, is exhausted. The other half of remaining memory, *i.e.,* the last quarter of total GPU memory, is saved to prefetch the next chunk of A tiles, to increase the overlap of communications with computations. Owing to this careful GPU memory management, chunks can proceed with minimal gap due to communications of A tiles, and without any flushing of B and C tiles back to CPU memory.

IV. Implementation

This algorithm has been implemented using an inspector-executor strategy over the Parameterized Task Graph (PTG) language [14] over the PaRSEC runtime system [15]. The implementation, as well as the benchmarks used to evaluate it are available at [5]. See [13] for more information on PaRSEC and PTG. The idea behind PTG is to define the Directed Acyclic Graph (DAG) of tasks as a concise and parameterized collection of tasks that exchange data through flows. Tasks are defined using task classes (a rudimentary templating approach), and task classes express synthetic conditions to enable input and output flows that carry the data. When the algorithm is regular, these conditions are fixed by a few parameters of the problem (*e.g.,* the input matrix size, the tile size). In our case, however, the problem is irregular, both because the matrices are block-sparse and because they are irregularly tiled. Thus, an inspector phase computes first what tasks exist, and how the data must flow between them. Then, a generic PTG that takes as input an execution plan produced by this inspector phase, allows the runtime system to execute it. This is sufficient to obtain a correct implementation of the irregular block-sparse matrix product. However, in order to implement the algorithm described above, one needs to be able to control the flow of

data across node boundaries; so we introduce, in addition to the necessary data flow, a control flow that constraints the choices of the runtime scheduler to those allowed by the algorithm (Section III). Thus, the algorithm representation can be seen as the superposition of two DAGs, having the same nodes (the tasks) but different sets of edges. One DAG, the *dataflow* DAG, represents the tasks and the data flow between them, a pure dataflow description of the algorithm as an unhindered rendition of the potential parallelism. The second DAG, the *control* DAG, represents a set of performance constraints, that are application and architecture specific, and that are necessary for the runtime to provide a finer control of the existing parallelism, in order to constraint when data transfers happen. This is the way chosen to optimize the execution of the tasks represented by the dataflow DAG.

The control flow DAG is also expressed within the PTG, and depends on the GPU memory, and the sparsity of the input matrices. Thus, it is also computed during the inspection phase, and provided as part of the execution plan. Note, however, that communications between nodes and transfers between the main RAM to GPUs are not explicit: they are deduced from the dataflow and realized in the background (*i.e.,* in parallel of task executions) by the runtime system. As a consequence, when the algorithm *reserves* 50% of a GPU memory to receive tiles of B and C when building a block, this is really implemented by constraining, with control flow, which tasks are ready to execute on that GPU, so they cannot refer more than 50% of the GPU memory if they were scheduled together on that GPU. Data transfers happen at the granularity of tiles, and tasks are scheduled as soon as the data they need is available on the GPU. The same applies to node-to-node transfers: although processes sharing the same row in the process grid need to have a copy of their share of the matrix A, this broadcast happens in the background, at the tile granularity, and tasks can be scheduled as soon as the data they need becomes available.

In addition to the data flow, PaRSEC programmers need to provide a description of the data to the runtime system. In our case, the matrices A and C are given using the data collections library available in PaRSEC. The matrix B, however, is stored implicitly: generation functions allow to instantiate any tile when needed. We extended PaRSEC's data collection library by developing a new data collection that instantiates the tasks corresponding to the tile generation on demand, when a tile needs to be instantiated. The usual mechanisms within the PaRSEC runtime system to manage the life-cycle of these data is then used to cache them as long as they are needed by any task, and discarded after this. The algorithm ensures that each tile of B is instantiated at most once per node that needs it (as noted, columns of B are replicated between processes that share the same column in the process grid), and since the generation routine does not have a CUDA implementation, these tasks are always executed on the CPUs.

Last, implicit data movement allows the runtime system to select the 'best' source of data, when multiple sources are available. This happens, for example, when two GPU devices need the same tile of A in our algorithm. One GPU needs to pull it from main memory, but the second may use the copy already on the first one, leveraging the fast NVlink to implement a device-to-device copy, thereby reducing the pressure on the PCI-Express bus to allow other memory transfers. This feature comes directly from the runtime system and does not require any modification of the algorithm itself.

V. Performance Evaluation

All performance measurements presented below were run on Summit, hosted at Oak Ridge National Laboratory. Summit holds 4,600 IBM AC922 compute nodes, each containing two POWER9 CPUs and 6 NVIDIA Volta V100 GPUs. The POWER9 CPUs have 22 cores running at 3.07 GHz, and 42 cores per node are made available to the application. Dual NVLink 2.0 connections between CPUs and GPUs provides a 25GB/s transfer rate in each direction on each NVLink, yielding an aggregate bidirectional bandwidth of 100GB/s.

PaRSEC, the proposed GEMM implementation and the driver program were all compiled in optimized (Release) mode, using XLC 16.1.1-2, CUDA 9.2.148, Spectrum MPI 10.3.0.0 available on the Summit programming environment. The BLAS3 GEMM kernel was the one provided in the cuBLAS library shipped with CUDA. We measured the practical peak of the GEMM kernel in this version of cuBLAS and this hardware at 7.2Teraflop/s per GPU. To obtain this value, we ran a single GEMM operation on large matrices that were pre-initialized in the GPU memory, repeated the operation 10 times, and took the fastest run measured.

All performance evaluation results presented below are obtained by measuring the time of executing the implementation described in Section IV, with the matrix A distributed between the nodes in a 2D-cyclic fashion, C empty (the necessary tiles will be allocated and initialized to zero when needed), and B generated on demand, on the cores. The time to generate B and inspect the execution, as well as the time to move data of C back and forth to the GPU, are all taken into account in the measurements presented below. Moreover, it is important to point out that, due to the target domain science, in most cases, the matrices A and C are too large to fit in GPU memory.

Each point is measured 5 to 10 times, and all figures showing performance present a Tukey box plot at the mark. On most figures, the measured variability is so small that the box plot is hidden by the mark or the line placed at the mean value, highlighting the stability of the distributed algorithm.

A. Synthetic Benchmarks

First, we consider matrices with random sparsity, in order to understand the performance of the implementation in a controlled setup. We set the number of nodes to 16, and start from a square and dense problem ($M = K = N$), then increase N and K (keeping $K = N$ to mimic the aspect ratios of the matrices involved in the target coupled-cluster ABCD contraction), and also decrease the density. Irregularity of tiling is set randomly to be uniform between 512 and 2048 (in each dimension), and both input matrices (A and B) have the target

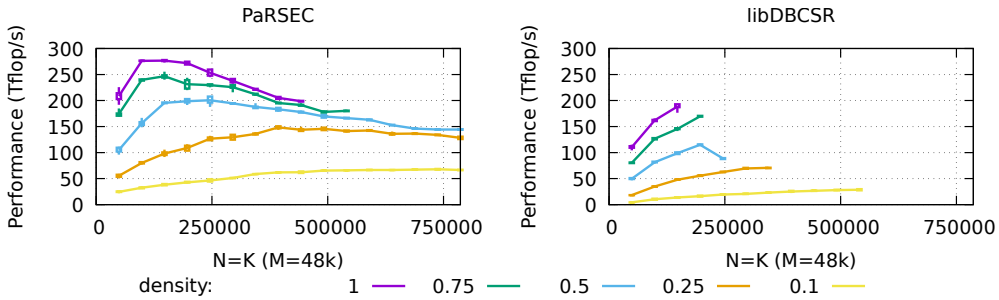


Fig. 2. Performance as a function of the matrix size (N and K) and density, on 16 nodes of Summit, for the PaRSEC implementation (left) and the libDBCSR implementation (right). Peak performance of GEMM for the 16 nodes is estimated at 672Tflop/s ($16 \times 6\text{GPU} \times 7\text{Tflop/s}$)

density (the density of C being computed from the shape and non-zero tiles of A and B). To decide which tiles are zero in A and B, an iterative algorithm selects uniformly a non-zero tile to eliminate, until eliminating another tile would draw the density of the matrix (element-wise) under the threshold.

We also compare the PaRSEC implementation of the GEMM algorithm with libDBCSR. libDBCSR [16] is a sparse matrix library that provides a block-sparse matrix-matrix product operation in distributed and on top of CUDA accelerators. We implemented the same synthetic benchmark on top of libDBCSR to serve as a basis for comparison. The benchmark is available in the same repository as the repository holding the implementation of the algorithm presented in this paper [5]. libDBCSR does not allow to manage multiple GPU per MPI process, so we deployed the runs with one process per GPU (*i.e.,* 96 processes). Each process gets allocated 6 cores and 1 GPU. As the performance and the capacity of libDBCSR depends on the process grid, for each problem size, we ran with all process grids achievable with 96 processes, and kept the best performing parameters. In most cases, the process grid (4×24) was the best performing one, but a few points are obtained with grids of (6×16) and (8×12).

Figure 2 depicts the performance as a function of N, K and the density of the problem. As stated above, this figure shows an average behavior over randomly uniform input data. Several conclusions can be drawn. First, density is the critical parameter of performance: it has more impact than the problem size or shape. This is expected because a lesser matrix density provides less opportunity for data reuse, shifting the block-sparse GEMM from compute-intensive to data-intensive (in this instance GPU transfer-intensive). We will revisit this topic later.

Considering the performance of libDBCSR, first the problems considered quickly become too large for this platform when using libDBCSR: for a dense problem (density = 1), problems of size $(48k, 192k, 192k)$ or more result in an error when trying to allocate the memory on some CUDA devices. To the best of our understanding, the algorithm used in libDBCSR does not manage the problem considered here, and assumes that a part of the data bigger than the available memory on each GPU should fit in memory. As the density gets lower, larger problems can be treated, but they all eventually reach a limit of capacity, while the algorithm described in this work focuses on managing problems that are much larger and do not make any assumption on the amount of memory required on each GPU. Second, even for problems that are manageable by both implementations, PaRSEC outperforms libDBCSR in all our experiments. This is because libDBCSR focuses on very small blocks (down to 6×6), and on square matrices [17], while the algorithm we present is designed to manage a large B matrix and works best with larger blocks. Note that in the square dense case ($M = N = K = 48k$), the PaRSEC implementation (203 TFlops/s) still outperforms libDBCSR (109 TFlops/s) by a factor 2. As libDBCSR does not leverage more than one GPU accelerator per MPI process, it was needed to create 96 processes to take advantage of the 96 GPUs; PaRSEC on the other hand runs with only 32 processes, each MPI process managing 3 GPUs. Thus, the libDBCSR application needs to communicate much more between processes than the PaRSEC one. We assume that this is the main reason behind the performance difference.

Focusing on the PaRSEC implementation, the performance reaches only half the GEMM-peak of the GPUs, even in the dense case. Comparing with the results that were obtained in [4] on the same machine, using the same runtime system, at this problem size and number of nodes, 80% to 90% of the GEMM-peak should be achievable. This difference is due to the problem shape, which required a different algorithm: tiles of B are generated on demand, but the size of B does not allow (in the application case) to keep all of them in memory until the completion of the algorithm. It is thus necessary to minimize the number of times that tiles of B are generated, and this drives the design of the algorithm to work on columns of B, while the traditional GEMM algorithms for square matrices, *e.g.*, [4], [18] work on square submatrices of C. As a consequence, the algorithm is not designed to perform optimally on square dense problems. As the A and C matrices become short and wide, the algorithm becomes more efficient, but the shape of the matrices themselves reduce the amount of reuse for the tiles of A and C, and thus limits the performance achievable in the dense case.

Last, the algorithm requires most tiles of A to be replicated on the processes that share the same row position in the process grid, and the corresponding data broadcasts are expensive

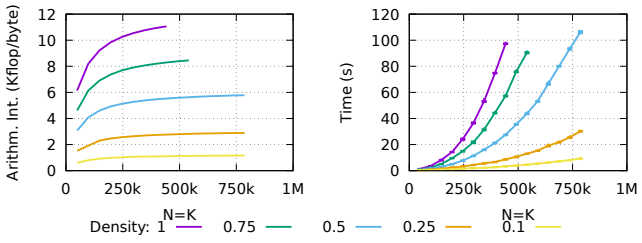


Fig. 3. Theoretical arithmetic intensity (left) and time to completion (right) of the synthetic matrix multiplication problem, on 16 nodes of Summit.

relative to the number of floating point operations, when the problem is square. In that case, the processes start by computing products with the tiles of A that are local, but if those are not enough to completely overlap the communications, execution stalls until the required tiles are received. When N increases, the number of operations available to overlap this communication increases, allowing the algorithm to reach higher performance. This increase in operations / bytes is illustrated in the arithmetic intensity (Figure 3).

Increasing K causes the tiles of A to be transferred to the GPU multiple times, since the part of A assigned to each GPU plus the column of B do not fit on the GPU memory anymore. This reduces the performance by increasing communication costs and reducing the effective arithmetic intensity. The maximum arithmetic intensity (*i.e.,* number of floating point operation divided by the aggregate size of A, B, and C) is depicted in Figure 3. The maximum intensity is an upper bound on the effective intensity since it can only be realized if A, B, and C were loaded only once to the device memory. As seen previously, the algorithm needs to load tiles of A multiple times, as the available memory on the GPU does not allow to keep all the input data, effectively decreasing the arithmetic intensity. Figure 3 also provides an explanation for the performance increase at the beginning of the curves in Figure 2, when columns of B and rows of A can fit together on the GPU, and also explains why the dominating element for the performance is the density of the matrices: as the sparsity increases, the number of operations relative to the amount of data to load decreases significantly, and as could be expected, the problem shifts from a compute-intensive problem to a data-intensive problem. In addition to this, each tile loaded on the GPU has a lower chance to get re-used for another product, as the number of tiles in the other matrix that correspond to it decreases with the density.

Although the effective arithmetic intensity and the measured performance inevitably decrease with the density of the problem, the time to solution remains dominated by the number of operations; since the number of operations decreases faster than the performance, as is illustrated in Figure 3, the time to solution also decreases with the density.

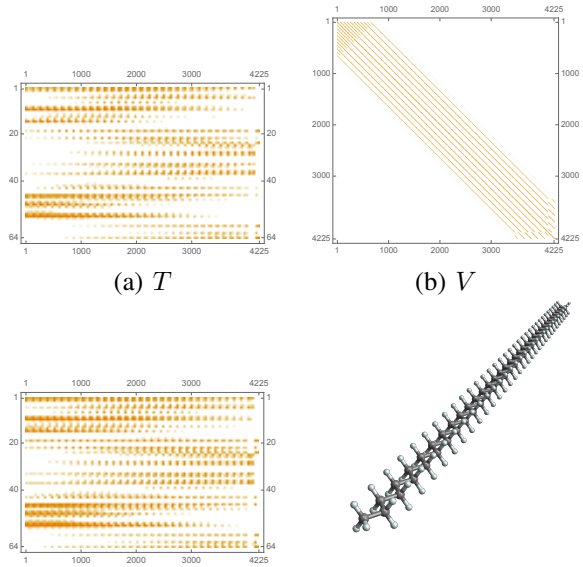
B. Practical Example: Evaluation of the ABCD coupled-cluster tensor contraction for the molecule $C_{65}H_{132}$

In this section, we use the new implementation of block-sparse matrix multiplication to evaluate the time-determining step of the CCSD electronic structure model (Equation (1)). Since problem sizes and traits vary greatly in practical applications, we decided to use an example that would be most challenging for reaching high absolute performance, namely a quasi-1-dimensional system and small atomic orbital (AO) basis, where the sparsity of tensors is maximized while the optimal (from the data compression perspective) tile size is small. The molecule we chose, $C_{65}H_{132}$, is representative of applications to 1-d polymers and quasi-linear molecules (such as some proteins); the choice of the def2-SVP AO basis is representative of medium-precision simulations in chemistry and condensed phase.

The ABCD term was evaluated using the AO-based formalism [19]. The input tensor T representing its initial state in the coupled-cluster simulation was evaluated in AO basis using the Laplace transform approximation, with the occupied orbitals localized and both occupied and the AO basis clustered to group spatially-close orbitals together [20]; the clustering defines tiling of the corresponding index ranges. The CPU-only implementation in MPQC evaluates tensor V on the fly, as needed; due to the lack of publicly-available efficient kernels for direct evaluation of AO integrals on GPUs (such kernels are under development by some of us) the GPU benchmarks used block-sparse V with the actual sparsity pattern determined by the CPU-only code but the tiles filled with random data. The sparse "shape" of tensor R was determined from the sparse shapes of tensors T and V as described previously [21].

Due to the quasi-1-dimensional structure and compact basis the T and V tensors in Eq. (1) are indeed very sparse (Figure 4). Note that the index range extents $O = 196$ and $U = 1570$ are much larger than would be practical for conventional CCSD: using dense tensors, the operation count for the ABCD term evaluation would be $2O^2U^4 \approx 0.47$ Exaflop, whereas the use of sparsity allows to evaluate this contraction in ≈ 1 Petaflop (see Table I). Reduction of the operation cost by more than two orders of magnitude illustrates the power of reduced scaling formulations of the electronic structure methods; clearly, the only way to deploy efficiently accurate electronic structure methods on leadership-scale machines is to focus on the reduced-scaling formalisms.

Unlike element-sparse representation, block-sparse representation of tensors introduces an additional degree of freedom, namely tiling of the index ranges. The tiling has dual purpose, to maximize the degree of sparsity and to control performance traits such as the amount of data parallelism for tile-level kernels and the amount of task-level parallelism for tensor-level operations. Using tiles that are 'too' large will reduce the degree of sparsity (in the limiting case of 1 tile per dimension the representation is dense) and greatly increase the operation count; using tiles that are 'too' small will decrease the amount of data parallelism exploitable by the tile-level kernels (in the limiting case of 1 element per tile, the representation becomes element-sparse, typically used for sparse matrix computation). These two objectives are contradictory, thus in practice for models with user-controllable tiling like the AO-basis CCSD, tiling should be optimized to balance its



(a) T (b) V (c) R (d) the molecule $C_{65}H_{132}$

Fig. 4. Pictorial representation of matricized block-sparse tensors T, V, and R for the $C_{65}H_{132}$ example (tiling v_1 is shown, with the aspect ratio is adjusted to make each tiles appear square). The extreme sparsity of the tensors is due to the quasi-one-dimensional shape of the molecule.

	Tiling v_1	Tiling v_2	Tiling v_3
$M \times N \times K$	$26576 \times 2464900 \times 2464900$		
#flop	877 Teraflop	923 Teraflop	1237 Teraflop
#flop (opt.)	850 Teraflop	899 Teraflop	1209 Teraflop
#GEMM tasks	1899971	468368	67818
#GEMM tasks (opt.)	1843309	455159	66315
Average #rows/block	700	[500;2500]	[1000;5000]
Average #columns/block	700	[500;2500]	[1000;5000]
Density of T	9.8%	10.2%	13.2%
Density of V	2.4%	2.6%	3.1%
Density of R (opt.)	14.9%	16.1%	21.7%

TABLE I
RELEVANT PROBLEM TRAITS FOR THE $C_{65}H_{132}$ TEST CASE WITH THE THREE VARIANTS OF TILING..

effects on the operation count and performance.

To evaluate the impact of the tiling on performance, we consider three representative tilings of the index ranges. Since the k-means-based clustering algorithm that determines the range tilings is quasirandom [20] and cannot ensure uniform tiling (this would necessarily violate locality in all practical applications), these tilings are generated by specifying the target number of clusters for each index range. Table I summarizes the difference between the three different tilings, from the most fine-grained one (v_1) to the most coarse-grained one (v_3). Tiling granularity impacts the tile size and the sparsity of the problem: a large grain tiling provides more irregular but larger average and minimum tile sizes, and increases the number of computations, as illustrated in the table and in Figure 5.

Figure 6 shows the execution time the ABCD contraction (Eq. (1)) for the $C_{65}H_{132}$ test case with the three tilings using between 3 and 108 V100 GPUs on Summit. Dotted lines represent a perfect strong scaling with respect to the 3 GPUs computation (first point). Time to solution decreases with the number of GPUs, from 272s at 3 GPUs for v_1, down to 34.9s, at 108 GPUs. Similar trends are observed for the other tilings. The parallel efficiency is not 1, however, as can be observed by the difference with the theoretical scaling curve: for v_1, at

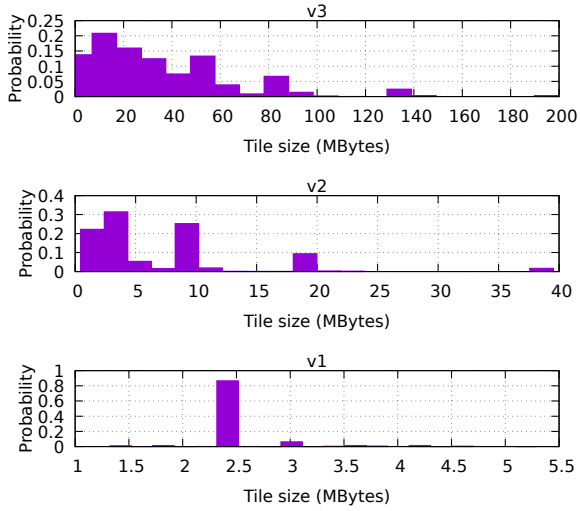


Fig. 5. Tile size distribution for the three tilings of the $C_{65}H_{132}$ test case. All input matrices use a similar block distribution..

108 GPUs, that parallel efficiency is down to 21%, when it is higher for v_2 (36.5%), or v_3 (35.2%). The cost of broadcasting tensor T, which is needed on all ranks that share the same rows, grows with the number of nodes and thus limits the scalability of the approach due to the compute time on each node becoming comparable to the communication time.

More interestingly, we observe that the overall time to completion in the cases v_2 and v_3 are similar, while for tiling v_3 the contraction involves 34% more flops compared to tiling v_2! Both tiling choices lead to significantly lower time to solution than the most fine-grained tiling v_1, which has the lowest flop count of all three tilings. This is a good demonstration of the dual aspect of tiling: larger tiles lead to higher performance of tile-level kernels but reduce the amount of sparsity and thus increase the operation count. This is justified by Figure 7, which shows the performance *per GPU* in the same experiment. Clearly, by increasing tiling it is possible to *trade sparsity for performance*; the problem of how to determine the optimal tiling is left to future studies.

The performance per GPU follows an inverse trend with the tiling size: as tiles grow bigger (v_3), each GPU kernel involves more flops. Moreover, the practical peak performance of these GPUs is around 7 Teraflop/s, while we observe up to 2.5Teraflop/s for the v_3 tiling, which represents 35% of the peak performance, degrading to 11% at 108 GPUs (a 30% parallel efficiency, as noted before). This shows that the arithmetic intensity (number of computation per bytes loaded) is too low to fully exploit the GPUs. Since a peak performance on a single tile can be obtained for tiles of 728×728, which is around the average tile size for tiling v_1, the problem does not reside in the tile sizes themselves, but in the tile re-use: the sparsity of the matrices V and T keep the re-use of data loaded on the GPU to a low amount, and GPU I/O dominates the execution time. Similar trends are observed in [17], where the performance at scale goes down to 30 Gflop/s per node (representing 3% of the GEMM-peak of that system).

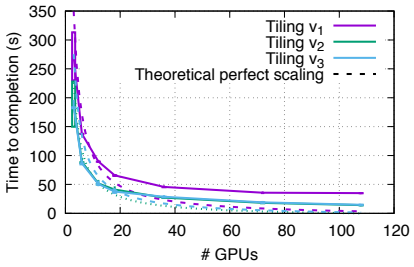


Fig. 6. Time to completion for $C_{65}H_{132}$.

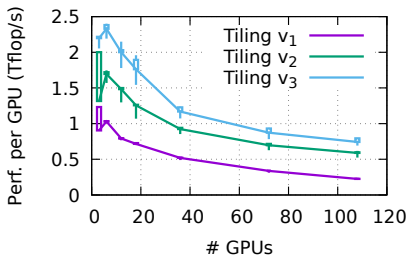


Fig. 7. Performance *per GPU* for $C_{65}H_{132}$

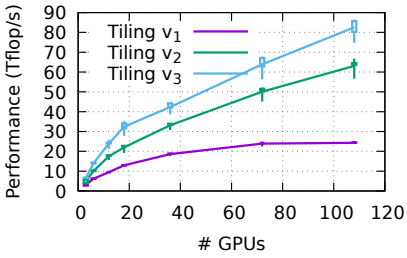


Fig. 8. Performance for $C_{65}H_{132}$

As observed previously, the performance per GPU decreases with the number of GPUs, due to the added number of nodes involved that introduce communications: when going from 3 to 6 GPUs, the computation can remain on the same node, and performance improves slightly; however upon further increase of the number of compute units, more nodes need to be introduced, increasing the total amount of communications. However, Figure 8 shows that, overall, the performance continue to increase up to 108 GPUs, when the completion time is less than a minute, even for the finest grain case. Because GPU I/O dominates the performance per GPU, increasing the amount of computation (even significantly, when comparing v_2 and v_3) does not impact the time to solution, because these added computations can be done in parallel with the data transfers. In the worst case, reducing the computations, but also reducing data reuse by increasing sparsity (v_1), increases the time to completion instead of decreasing it.

To compare the performance of the new GPU implementation of the state-of-the-art tensor contraction, we used the CPU-only code implemented in the MPQC package to evaluate the ABCD contraction for the $C_{65}H_{132}$ test case (no GPU version exists unfortunately). The computations utilizing $\{8,16\}$ nodes of Summit (total of 672 compute cores) completed in $\{308,158\}$ seconds, respectively. The corresponding GPU implementation using the most performant tiling v_3 on all GPUs available on the same set of nodes of Summit would reduce the time to solution by a factor ≈ 10. The estimated efficiency of the CPU-only computation is rather low: assuming 2 Teraflop/s CPU peak performance per node for the 16-node job leads to an efficiency of $\approx 17\%$ relatively to peak. Since the known performance heuristics of the CPU-only code in the MPQC package are established primarily for x86 architecture, it is likely that the CPU-only performance on Summit can be improved. Nevertheless the comparison is fair: MPQC is well-documented as a state-of-the-art coupled-cluster code [7], [22], and its CPU-only performance on Summit is an accurate reflection of the current state-of-the-art of chemistry codes on Summit.

VI. Related work

Due to lack of space, we only discuss a few closely related references in this section. A full review of related work is available in the companion report [13].

Block-Sparse Matrix Product on GPU: There are few works directly targeting block-sparse matrix product on distributed systems using accelerators: [23] uses tensor flow [24] to implement a block-sparse matrix-product on a single GPU; SuiteSparse [25] includes matrix-product operations for block-sparse and sparse matrices, on single node GPU; [26], [27] present Chunk and Tasks, a distributed algorithm for block-sparse matrix product on GPUs, using quad-trees to represent the sparsity and reduce the memory overheads. This algorithm focuses on the product of square matrices, at scales that are much smaller than the problem considered in this paper.

In Section V we have compared our approach with a CUDA-enabled version of Distributed Block Compressed Sparse Row (DBCSR) library [17], [28], a block-sparse matrix library used by the CP2K framework [29]. DBCSR originally targeted square block-sparse matrices, thus it uses the Cannon algorithm to schedule communications between nodes, re-orders columns and rows to balance the work between nodes, and uses dynamic scheduling of work on the GPU to orchestrate computations. Matrices in CP2K typically have blocks of small sizes (this assumption does not apply to our data), thus DBCSR generates JIT-compiled optimized kernels for these particular block sizes. DBCSR was recently generalized to tensor contractions [18], which required introduction of modified versions of the Cannon algorithm with partial replication of data; however, the two target aspect ratios considered do not match ours, and no absolute performance data was presented.

To the best of our knowledge, our algorithm is the first algorithm published in the literature that is capable of minimizing the transfers from CPU to GPU memory for arbitrary matrix sizes and shapes, specifically targets multi-GPU nodes by taking advantage of the NVlink device-to-device communication capability when the opportunity arises, and leverages the shape of the large matrix to reduce node-to-node communications.

Electronic Structure: Distributed-memory algorithms for coupled-cluster and other many-body electronic structure methods have been in development since late 1980s and are now available in several packages (see [7] for a recent review of CCSD implementations), most notably in NWChem (a flagship distributed-memory quantum chemistry code), ACESIII, and GAMESS. Unfortunately very little of this capability can be executed on distributed-memory heterogeneous platforms. NWChem has a CUDA-based implementation of perturbative triples correction to CCSD, also known as (T), that has been demonstrated on a GPU-equipped distributed-memory platform and can take advantage of multiple GPUs and multiple CPU cores on each node (however, the CCSD code is CPU only) [30]. Very recently a distributed memory implementation

of (T) in MPQC was demonstrated that can take advantage of multiple GPUs per node [22]. GAMESS has demonstrated a GPU-capable implementation of select terms in the CCSD code on 1 node with 1 GPU [31].

VII. Conclusion

In this paper, we focused on the block-sparse tensor contraction, a paradigmatic kernel in many scientific applications, whose characteristics (heterogeneity, sparsity, reduced computational intensity) make it a challenging candidate for distributed multi-GPU platforms. We have provided a highly-tuned algorithm that carefully orchestrates task executions and data transfers between CPU and GPUs and between nodes to maximize resource occupancy. The flexibility and programmability of the underlying PaRSEC runtime greatly improved the algorithm implementation while providing a highly efficient support for multi-GPU distributed-memory platforms. The resulting implementation takes advantage of the GPUs, a major source of computing power, and obtains an efficiency and performance yet unrealized in the domain. Although comparison with existing tools to solve the same problem are not straightforward because these tools do not run on the same hardware, the deployment on a real case shows a factor 10 of speedup using the same nodes. This shows that our new algorithm offers promising perspectives to solve problems of unprecedented scale and complexity.

Future work will aim at modeling the interactions between the tiling and the performance, in order to increase the efficiency of the algorithm. We will also extend the experiments to larger problems, representative of more complex molecular structures. Although we focused the evaluation on a representative of the most sparse cases, different molecules have the potential to provide much denser and compute-intensive input matrices, thereby (likely) enabling our algorithm to reach higher peak performance.

References

[1] Oak Ridge National Laboratory, "Oak Ridge Leadership Computing Facility," https://www.olcf.ornl.gov/.

[2] Top500, "Top 500 Supercomputer Sites," June 2020, https://www.top500.org/lists/2029/06/.

[3] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library," in *SC'2019*. ACM Press, 2019.

[4] T. Herault, Y. Robert, G. Bosilca, and J. J. Dongarra, "Generic matrix multiplication for multi-GPU accelerated distributed-memory platforms over PaRSEC," in *10th ScalA Workshop @SC*. IEEE, 2019, pp. 33–41.

[5] T. Herault, Y. Robert, G. Bosilca, R. J. Harrison, C. A. Lewis, and E. F. Valeev, "Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure: software artifact," https://bitbucket.org/herault/irr-gemm-gpu-over-parsec, commit 17c88d2, April 2020.

[6] I. Shavitt and R. Bartlett, *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*, ser. Cambridge Molecular Science. Cambridge University Press, 2009.

[7] C. Peng, J. A. Calvin, F. Pavošević, J. Zhang, and E. F. Valeev, "Massively Parallel Implementation of Explicitly Correlated Coupled-Cluster Singles and Doubles Using TiledArray Framework," *J. Phys. Chem. A*, vol. 120, no. 51, pp. 10 231–10 244, Dec. 2016.

[8] C. Riplinger, P. Pinski, U. Becker, E. F. Valeev, and F. Neese, "Sparse maps—A systematic infrastructure for reduced-scaling electronic structure methods. II. Linear scaling domain based pair natural orbital coupled cluster theory," *J Chem Phys*, vol. 144, no. 2, Jan. 2016.

[9] R. A. Van De Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[10] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication," in *IPDPS*. IEEE, 2013.

[11] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *JPDC*, vol. 74, no. 12, pp. 3176–3190, 2014.

[12] C. Peng, C. Lewis, X. Wang, M. Clement, F. Pavosevic, J. Zhang, V. Rishi, N. Teke, K. Pierce, J. Calvin, J. Kenny, E. Seidl, C. Janssen, and E. Valeev, "The Massively Parallel Quantum Chemistry Program (MPQC), Version 4.0.0," http://github.com/ValeevGroup/mpqc, 2018.

[13] T. Herault, Y. Robert, G. Bosilca, R. Harrison, C. Lewis, E. Valeev, and J. Dongarra, "Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure (revised version)," INRIA, Tech. Rep. 9365, 2020, available at https://hal.inria.fr/hal-02970659v1.

[14] A. Danalis, G. Bosilca, A. Bouteiller, T. Hérault, and J. J. Dongarra, "PTG: an abstraction for unhindered parallelism," in *4th WOLFHPC workshop @SC*. IEEE, 2014, pp. 21–30.

[15] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *IEEE Comp. Science Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[16] liDBCSR, "A sparse matrix library," https://www.cp2k.org/dbcsr, 2020.

[17] O. Schütt, P. Messmer, J. Hutter, and J. VandeVondele, *GPU-Accelerated Sparse Matrix–Matrix Multiplication for Linear Scaling Density Functional Theory*. John Wiley & Sons, Ltd, 2016, ch. 8, pp. 173–190.

[18] I. Sivkov, P. Seewald, A. Lazzaro, and J. Hutter, "DBCSR: A blocked sparse tensor algebra library," in *PARCO*, ser. Advances in Parallel Computing, vol. 36. IOS Press, 2019, pp. 331–340.

[19] R. Kobayashi and A. P. Rendell, "A direct coupled cluster algorithm for massively parallel computers," *Chem. Phys. Lett.*, vol. 265, no. 1-2, pp. 1–11, Jan. 1997.

[20] C. A. Lewis, J. A. Calvin, and E. F. Valeev, "Clustered Low-Rank Tensor Format: Introduction and Application to Fast Construction of Hartree–Fock Exchange," *J. Chem. Theory Comput.*, vol. 12, no. 12, pp. 5868–5880, Dec. 2016.

[21] J. A. Calvin, C. A. Lewis, and E. F. Valeev, "Scalable task-based algorithm for multiplication of block-rank-sparse matrices," in *IA3 '15*. ACM Press, 2015, pp. 1–8.

[22] C. Peng, J. Calvin, and E. F. Valeev, "Coupled-Cluster Singles, Doubles and Perturbative Triples with Density Fitting Approximation for Massively Parallel Heterogeneous Platforms," *Int. J. Quant. Chem.*, vol. 12, no. 119, p. e25894, 2019.

[23] S. Gray, A. Radford, and D. P. Kingma, "Gpu kernels for block-sparse weights," *arXiv preprint arXiv:1711.09224*, vol. 3, 2017.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[25] T. Davis *et al.*, "SuiteSparse : a suite of sparse matrix software," http://faculty.cse.tamu.edu/davis/suitesparse.html, Apr 2020.

[26] E. H. Rubensson, E. Rudberg, and P. Sałek, "A hierarchic sparse matrix data structure for large-scale hartree-fock/kohn-sham calculations," *J. Computational Chemistry*, vol. 28, no. 16, pp. 2531–2537, 2007.

[27] E. H. Rubensson and E. Rudberg, "Locality-aware parallel block-sparse matrix-matrix multiplication using the chunks and tasks programming model," *Parallel Computing*, vol. 57, pp. 87 – 106, 2016.

[28] U. Borštnik, J. VandeVondele, V. Weber, and J. Hutter, "Sparse matrix multiplication: The distributed block-compressed sparse row library," *Parallel Computing*, vol. 40, no. 5-6, pp. 47–58, Apr. 2014.

[29] CP2K, "Open source molecular dynamics," https://www.cp2k.org, 2020.

[30] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal, "Optimizing tensor contraction expressions for hybrid CPU-GPU execution," *Clust. Comput*, vol. 16, no. 1, pp. 131–155, 2013.

[31] A. Asadchev and M. S. Gordon, "Fast and Flexible Coupled Cluster Implementation," *J. Chem. Theory Comput.*, vol. 9, no. 8, pp. 3385–3392, Jul. 2013.