In addition to tensor contractions, one of the most pronounced computational bottlenecks in the nonorthogonally spin-adapted forms of the quantum chemistry methods CCSDT and CCSDTQ, and their approximate forms—including CCSD(T) and CCSDT(Q)—are spin summations. At a first sight, spin summations are operations similar to tensor transpositions, but a closer look reveals additional challenges to high-performance calculations, including temporal locality and scattered memory accesses. This article explores a sequence of algorithmic solutions for spin summations, each exploiting individual properties of either the underlying hardware (e.g., caches, vectorization) or the problem itself (e.g., factorizability). The final algorithm combines the advantages of all the solutions while avoiding their drawbacks; this algorithm achieves high performance through parallelization and vectorization, and by exploiting the temporal locality inherent to spin summations. Combined, these optimizations result in speedups between 2.4× and 5.5× over the NCC quantum chemistry software package. In addition to such a performance boost, our algorithm can perform the spin summations \textit{in-place}, thus reducing the memory footprint by 2× over an \textit{out-of-place} variant.

CCS Concepts: • Mathematics of computing → Mathematical software; • Software and its engineering → Source code generation; • Computing methodologies → Linear algebra algorithms; Special-purpose algebraic systems; • Theory of computation → Vector/streaming algorithms;

Additional Key Words and Phrases: High-performance computing, tensor transposition, in-place

ACM Reference format:
Paul Springer, Devin Matthews, and Paolo Bientinesi. 2019. Spin Summations: A High-Performance Perspective. \textit{ACM Trans. Math. Softw.} 45, 1, Article 10 (March 2019), 22 pages.
https://doi.org/10.1145/3301319

1 INTRODUCTION

In quantum chemistry, especially in high-accuracy calculations utilizing the popular coupled cluster family of methods (Čížek 1966; Bartlett and Musiał 2007; Helgaker et al. 2013), tensor algebra plays a central role and often accounts for most of the computational effort. The main tensor operation utilized is a tensor contraction, the multidimensional analogue of matrix multiplication, which is typically expressed in terms of perfectly nested loops (Ma et al. 2011; Apra et al. 2014), “Loops

Financial support was provided by the Deutsche Forschungsgemeinschaft (DFG) through grant GSC 111 and by Intel Corporation through Parallel Computing Center grants to RWTH Aachen and UT Austin. D. Matthews received support from the Arnold and Mabel Beckman Foundation.

Authors’ addresses: P. Springer and P. Bientinesi, AICES, RWTH Aachen University, Schinkelstr. 2, 52062 Aachen, Germany; emails: [springer, pauldj]@aices.rwth-aachen.de; D. Matthews, ICES, University of Texas at Austin, Austin, TX 78712; email: dmathews@utexas.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0098-3500/2019/03-ART10 $15.00
https://doi.org/10.1145/3301319
over GEMM” (Di Napoli et al. 2014), “Transpose-Transpose-GEMM-Transpose” (Hirata 2003), and lately also in terms of a native GEMM-like tensor-tensor contraction kernel that mimics the design of a high-performance matrix-matrix multiplication (Springer and Bientinesi 2018; Matthews 2018).

Other essential tensor operations can consume a significant portion of the runtime, especially when they are poorly optimized relative to hand-tuned matrix multiplication kernels, or when they are practically bandwidth bound. Tensor transpositions (permutations) (Wei and Mellor-Crummey 2014; Lyakh 2015; Springer et al. 2016, 2017a) represent one of the bottlenecks, but recent work has shown that this cost can be minimized (Springer et al. 2016, 2017a, 2017b) or even eliminated entirely (Matthews 2018; Springer and Bientinesi 2018). Instead, in high-accuracy calculations based on the efficient nonorthogonal spin-adaptation scheme (Čížek 1966; Kucharski and Bartlett 1992; Matthews et al. 2013), there remains an additional and significant bottleneck in the form of the “spin summation” operation (Matthews et al. 2015). This operation is critical to forming the intermediates necessary to implement the fully factored form of the working equations, and also in the regularization of the tensor quantities to maintain numerical stability in the face of a linearly dependent basis (Kucharski and Bartlett 1992; Adams and Paldus 1979). The optimization of this important kernel is the focus of this work. In particular, we examine the set of spin summation operations summarized in Section 5.1, which cover all of the operations required for the nonorthogonally spin-adapted CCSDT (Noga and Bartlett 1987) and CCSDTQ (Kucharski and Bartlett 1992) quantum chemistry methods, as well as their approximate forms, such as CCSDT(Q) (Bomble et al. 2005).

Tensor transposition is related to spin summation in that the nonlocality (in both space and time) of data references must be taken into account to achieve high efficiency on modern hardware. The primary difference between transposition and spin summation, however, is that spin summation entails data reuse of both the input and output tensors, whereas transposition accesses the tensors only once. For factorizable summations (defined in the following), there is also a difference between algorithms that perform optimal and suboptimal numbers of floating-point operations. Thus, although specific optimization techniques from tensor transposition may be applied to the spin summation problem, a bottom-up analysis of algorithmic efficiency for this distinct class of problems is warranted.

The techniques and algorithms developed in this work are applicable to other important tensor operations beyond spin summations. A spin summation is essentially a linear sum of multiple tensor transpositions; this pattern appears in other tensor operations as well, such as in tensor (anti)symmetrization (Hanrath and Engels-Putzka 2010; Lai et al. 2012), and in the aggregation of intermediate tensor products produced in distinct orderings (Baumgartner et al. 2005; Hartono et al. 2009). Optimization of operations involving multiple transpositions is especially important for higher-dimensional tensors due to the proliferation of possible transpose variants.

Another important application for the techniques discussed in this article is machine learning (Sak et al. 2014), where pointwise tensor operations (e.g., ReLU, tanh, addition) are frequently “fused” together into a unified kernel, which is often required to handle multiple data layouts simultaneously.

To the best of our knowledge, this work represents the first discussion of spin summations from a high-performance perspective. The contributions made in this publication can be summarized as follows:

— The concepts of spin summations and the factorization thereof are introduced.
— For the calculation of spin summations, we discuss several algorithmic variants and provide the relative analyses.
— We present a high-performance open source algorithm for 3D and 4D spin summations used in computational chemistry, particularly in the nonorthogonally spin-adapted CCSDT and CCSDTQ methods and their various approximations.
— For this algorithm, we discuss a parallel and vectorized implementation\(^1\) that exploits the CPU’s cache hierarchy and reduces the amount of required floating-point operations while only using small buffers that fit into cache. Compared to the reference NCC quantum chemistry software (Matthews and Stanton 2015), part of the CFOUR package (Stanton et al. 2014), the combination of the optimizations results in speedups ranging from 2.4× to 5.5×.
— We illustrate that our high-performance algorithm can also be used for in-place spin summations of 2D, 3D, or 4D “square” tensors, reducing the memory footprint by 2×.

The remainder of this article is structured as follows. Section 2 reviews related work on tensor transposition and related transformations such as unpacking of antisymmetric tensors. Section 3 familiarizes the reader with spin summations, the nomenclature used, and the cache hierarchy of a modern CPU. Section 4 presents different implementations for spin summations and discusses their advantages and disadvantages. Section 5 introduces the exhaustive list of spin summations of interest and describes the measurement environment. Section 6 evaluates the performance of the different implementations and highlights their key features. Section 7 concludes our findings.

2 RELATED WORK

Matrix transposition (i.e., 2D tensor transposition) is a well-studied operation, including optimizations for blocking, vectorization, unrolling, and software prefetching (McCalpin and Smotherman 1995; Mateescu et al. 2012; Lu et al. 2006; Goldbogen 1981; Vladimirov 2013; Chatterjee and Sen 2000). For 3D tensors, the same optimizations are investigated in the context of out-of-place transpositions on CPUs (van Heel 1991; Jodra et al. 2015).

The optimization of arbitrary-dimensional tensor transpositions has gained more interest in recent years (Wei and Mellor-Crummey 2014; Lyakh 2015; Springer et al. 2016, 2017a). Our previous work on the Tensor Transposition Compiler (TTC) (Springer et al. 2016, 2017a) relied on code generation to yield a nearly optimal kernel for any given tensor transposition. Although the generated code attained high performance in the general case, it was only applicable to tensor transpositions that were known at compile time. For this reason, we developed HPTT (Springer et al. 2017b), a high-performance C++ library for tensor transpositions that maintains the desirable properties of TTC (e.g., explicit vectorization, loop reordering, autotuning) but avoids the code generation process.

Although there has been some focus on in-place tensor transpositions (Ding 2001; He and Ding 2002; Catanzaro et al. 2014), their performance—in the general case—has been found to be limited.

In the context of data reuse in multidimensional operations, algorithms for unpacking antisymmetric tensors into matrices were studied by Hanrath and Engels-Putzka (2010). They identify both “matrix-driven” and “tensor-driven” approaches, which are essentially equivalent to Algorithm 1 presented later. These approaches are also similar in terms of data access patterns to configuration interaction (CI) algorithms, which come in both “integral-driven” (Saunders and Lenthe 1983; Ansaloni et al. 2000; Klene and Robb 2000; Gan et al. 2003) and “configuration-driven” (Buenker and Krebs 1999; Jiang et al. 2009) varieties. In each of these cases, accesses to a 1D object is linearized at the expense of scattered accesses to another object. Blocking for the cache hierarchically has been

\(^1\)The source code is publicly available at https://www.github.com/springer13/spin-summations.
studied and successfully applied in the context of stencil algorithms (Datta et al. 2008; Nguyen et al. 2010); this is in several ways similar to the multidimensional tensor case.

3 BACKGROUND

To keep this publication self-contained, we introduce the used nomenclature and explain necessary architectural properties of a modern CPU.

3.1 Nomenclature

A $d$-dimensional tensor $A \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ is stored as a $d$-dimensional array adhering to the Fortran storage scheme (i.e., column-major). For instance, given a 3D tensor $A \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, the element $A_{i_1,i_2,i_3}$, for any index $(i_1, i_2, i_3) \in \{0, \ldots, n_1 - 1\} \times \{0, \ldots, n_2 - 1\} \times \{0, \ldots, n_3 - 1\}$, is placed in the linearized memory location $&A + i_1 + i_2 n_1 + i_3 n_1 n_2$ ($\&$ denotes the “address of” operator). Due to the nature of our problem, we are only concerned with 3D and 4D tensors for which the sizes of all dimensions are identical (i.e., $n_1 = n_2 = n_3 = n_4 = N$).

**Definition 1 (Permutation).** Given a bijective function

$$\phi : \{1, 2, \ldots, d\} \to \{1, 2, \ldots, d\},$$

we define the permutation $\pi$ that operates on $d$-tuples as

$$\pi : \mathbb{N}^d \to \mathbb{N}^d, \quad (i_1, i_2, \ldots, i_d) \mapsto (i_{\phi(1)}, i_{\phi(2)}, \ldots, i_{\phi(d)}).$$

(2)

Instead of defining $\pi$ explicitly, we use the shorter notation: $\pi_{\phi(1)\phi(2)\ldots\phi(d)}$.

**Example.** $\pi_{321}$ applied to an index $(i_1, i_2, i_3)$ inverts its order and yields the permuted index $\pi_{321}(i_1, i_2, i_3) = (i_3, i_2, i_1)$.

**Definition 2 (Tensor Permutation).** Given a tensor $A \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ and a permutation $\pi$, we formally express a tensor permutation as

$$B_{i_1, i_2, \ldots, i_d} \leftarrow A_{\pi(i_1, i_2, \ldots, i_d)} \quad \forall (i_1, i_2, \ldots, i_d) \in \mathbb{N}_{n_1} \times \mathbb{N}_{n_2} \times \cdots \times \mathbb{N}_{n_d}.$$  

(3)

Considering that this notation is quite verbose, we revert to a shorter form where $\pi$ acts as an operator on the entire tensor such that a tensor permutation can be written as $B \leftarrow \pi(A)$. Although we use the same symbol $\pi$ both as an operator that acts on indices as well as on tensors, these two cases are easily distinguishable by the context.

**Example.** A matrix transposition of $A, B \in \mathbb{R}^{N \times N}$ can be expressed via the tensor permutation: $B \leftarrow \pi_{21}(A)$.

**Definition 3 (Spin Summation).** In its most general form, a spin summation of a $d$-dimensional tensor $A$ is a sum of $n$ tensor permutations $\omega_1, \ldots, \omega_n$ each scaled by some $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$:

$$B \leftarrow \sum_{i=1}^{n} \alpha_i \omega_i(A) = (\alpha_1 \omega_1 + \alpha_2 \omega_2 + \cdots + \alpha_n \omega_n)(A).$$

(4)

**Example.** $B \leftarrow (\pi_{123} + \pi_{213} + \pi_{321} + \pi_{231} + \pi_{132} + \pi_{312})(A)$ is spin summation where all $\alpha_i$ are equal to 1.

Most of the spin summations that we are interested in exhibit a very important property, which we refer to as *factorizable.*
Definition 4 (Factorizable Spin Summation). A spin summation $\mathcal{B} \leftarrow \sum_{i=1}^{n} \alpha_i \omega_i(\mathcal{A})$ is factorizable if it can be separated into two successive spin summations such that

$$\mathcal{T} \leftarrow \sum_{i=1}^{\tilde{n}} \hat{\alpha}_i \hat{\omega}_i(\mathcal{A}),$$

$$\mathcal{B} \leftarrow \sum_{i=1}^{n} \tilde{\alpha}_i \tilde{\omega}_i(\mathcal{T}),$$

with $\tilde{n} + \hat{n} < n$ and $\mathcal{T} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$. This definition can be applied recursively, factoring the initial spin summation into more than two stages.

Example. The spin summation

$$\mathcal{B} \leftarrow (\pi_{123} + \pi_{132} + \pi_{213} + \pi_{312} + \pi_{321})(\mathcal{A})$$

(7)

can be factorized as

$$\mathcal{B} \leftarrow (\pi_{123} + \pi_{213})(\pi_{123} + \pi_{321} + \pi_{132})(\mathcal{A}).$$

(8)

Note that every spin summation $\mathcal{B} \leftarrow \sum_{i=1}^{n} \alpha_i \omega_i(\mathcal{A})$ is factorizable without the constraint $\tilde{n} + \hat{n} < n$. For instance, let $\hat{\omega}$ be an arbitrary permutation, then all factorizations of the following form are viable:

$$\mathcal{T} \leftarrow \hat{\omega}(\mathcal{A}),$$

(9)

$$\mathcal{B} \leftarrow \sum_{i=1}^{n} \alpha_i \tilde{\omega}_i(\mathcal{T}),$$

(10)

where $\tilde{\omega}_i = (\omega_i \circ \hat{\omega}^{-1})$ denotes the composition of $\omega_i$ and the inverse of $\hat{\omega}$.

3.2 Cache Hierarchy

The cache hierarchy of modern CPUs bridges the ever widening gap between the performance of the CPU’s floating-point units and its memory subsystem\(^2\) by buffering frequently used data in fast on-chip memory (i.e., caches and registers).

Caches exploit the fact that data, once loaded from main memory, is likely accessed again soon after; this effect is referred to as temporal locality. Thus, repeated accesses to the same memory location are served by the fast caches as opposed to the slow main memory. Another important feature of caches is that they fetch data at the granularity of a cacheline (typically 64 bytes = 16 single- = 8 double-precision elements). This means that an access to a memory location $l$ not only loads the element situated at $l$ but indeed its entire cacheline. Thus, a subsequent access to $l + 1$ (assuming that $l$ and $l + 1$ belong to the same cacheline) is served by the cache and does not require an additional load from main memory; this process is referred to as spatial locality.

The caches of a CPU are organized into multiple levels, which vary in size, bandwidth, and latency (Intel Corporation 2015). The Intel Xeon E5-2680 v3 CPU, for instance, has three levels: L1, L2, and L3, with decreasing speed and increasing size. Although L1 and L2 are private to each core, the L3 is shared between all the cores. Since modern CPUs use a cache coherency protocol (Patterson

\(^2\)The floating-point unit of one Intel Xeon E5-2680 v3 core can issue two fused-multiply-adds ($c \leftarrow a \times b + c$) per cycle; each fused-multiply-add operates on $3 \times 8$ single-precision elements simultaneously. The full Intel Xeon E5-2680 v3 CPU—assuming a turbo boost frequency of 3.1GHz—would require a staggering main memory bandwidth of 6,652GiB/s to keep all of its floating-point units—across all of its 12 cores—busy. This value is in sharp contrast to the CPU’s theoretical peak memory bandwidth of 65.3GiB/s.
and Hennessy 2007; Bryant et al. 2003) to keep all the caches of a CPU coherent, it is important that different cores do not write to the same cacheline; such a situation is referred to as false sharing and results in coherence traffic among the cores, which can significantly lower performance.

Modern write-back caches typically employ the write-allocate policy (Bryant et al. 2003): a write-miss (i.e., a write to a memory location that is not in any level of the cache hierarchy) fetches the corresponding cacheline from main memory before updating it; this additional memory traffic is referred to as write-allocate traffic. Such a mechanism is favorable if the data is be accessed again in the near future. So-called nontemporal store instructions avoid this write-allocate traffic by writing an entire cacheline without fetching it from main memory prior to the write (see Intel Corporation 2015) for more details).

3.3 Blocking

Any nontrivial real-world application has to utilize a CPU’s rich cache hierarchy to attain good performance. One of the most powerful optimization techniques to enable such a design is blocking. Blocking refers to an reordering of the memory accesses to make better use of the fast on-chip memory (i.e., caches and registers). This optimization often entails a significant modification of the original algorithm. A very prominent operation that makes extensive use of this technique is matrix-matrix multiplication. The interested reader is referred to Gunnels et al. (2001), Goto and Geijn (2008), and Van Zee and van de Geijn (2015) for further details.

4 ROAD TO HIGH PERFORMANCE

The following discussion introduces several implementations of the spin summations in Equations (7) and (8); this spin summation is small enough to serve as an example throughout this article, yet it is rich enough to demonstrate all features of interest. The implementations presented in Sections 4.1 through 4.5 employ optimization strategies targeting different features of modern CPUs. We analyze their advantages and disadvantages and lay the foundation for a high-performance implementation (see Section 4.6) that combines the advantages of its predecessors while avoiding their drawbacks.

4.1 Algorithm 1: Locality in \( B \)

Algorithm 1 is a direct translation of Equation (7) into code. Provided that the compiler is smart enough to accumulate the six elements from \( A \) (line 5) at the register level, this algorithm only requires one write to each memory location of \( B \), resulting in perfect temporal locality for \( B \). Moreover, the loops are ordered such that the output tensor \( B \) is traversed in a linear fashion. Although this ordering leads to a preferable memory access pattern for \( B \), it causes an unfavorable access pattern for \( A \). Phrased differently, this algorithm perfectly exploits both the spatial locality and temporal locality in \( B \), but it only exhibits very limited locality in \( A \). More precisely, the spatial locality in \( A \) is limited to the accesses of \( A_{i_1i_2i_3} \) and \( A_{i_1i_2i_3} \) that have the same stride-one index (i.e., \( i_1 \)) as the accesses to \( B \); all other accesses to \( A \) (i.e., \( A_{i_2i_1i_3}, A_{i_2i_3i_1}, A_{i_3i_1i_2}, A_{i_3i_2i_1} \)) have a different stride-one index. Likewise, temporal locality in \( A \) is only achieved if any of \( i_1, i_2, \) or \( i_3 \)

| ALGORITHM 1: Locality in \( B \) |
|--------------------------------|
| #pragma omp parallel for collapse(2) schedule(static) |
| for \( (i_3 = 0; i_3 < N; i_3++) \) do |
| for \( (i_2 = 0; i_2 < N; i_2++) \) do |
| for \( (i_1 = 0; i_1 < N; i_1++) \) do |
| \( B_{i_1i_2i_3} \leftarrow A_{i_1i_2i_3} + A_{i_1i_3i_2} + A_{i_2i_1i_3} + A_{i_2i_3i_1} + A_{i_3i_1i_2} + A_{i_3i_2i_1} \) |
are identical; these situations are negligible. Furthermore, due to the lack of spatial locality, this implementation is not vectorizable without relying on expensive gather memory operations.

Thanks to the selected loop order, this algorithm can be efficiently parallelized via a single OpenMP statement (line 1). This statement parallelizes all loops corresponding to the indices that are not the stride-one index of $\mathcal{B}$. This strategy is ideal in the sense that it avoids false sharing between the threads while still providing as much parallelism as possible.

### 4.2 Algorithm 2: Temporal Locality in $\mathcal{A}$ and $\mathcal{B}$

The key idea of Algorithm 2, outlined in this section, is to exploit the structure of spin summations and transform the scattered memory accesses to $\mathcal{A}$ into a structured form that makes use of the temporal locality in both the input and output tensor.

#### ALGORITHM 2: Temporal locality in $\mathcal{A}$ and $\mathcal{B}$

```plaintext
1 #pragma omp parallel for schedule(static,1)
2 for (i_3 = 0; i_3 < N; i_3++) do
3   for (i_2 = 0; i_2 ≤ i_3; i_2++) do
4     for (i_1 = 0; i_1 ≤ i_2; i_1++) do
5       for (π ∈ {π_{123}, π_{312}, π_{321}, π_{231}, π_{231}}) do
6         $\mathcal{B}_\pi(i_1, i_2, i_3) \leftarrow \mathcal{A}_\pi(i_1, i_2, i_3) + \mathcal{A}_\pi(i_1, i_2, i_3) + \mathcal{A}_\pi(i_1, i_2, i_3) + \mathcal{A}_\pi(i_1, i_2, i_3)
7     end for
end for
end for
end for
```

One noticeable difference to the previous algorithm is that this algorithm limits the range of the loops in lines 2 through 4 such that the loop variables $i_1$, $i_2$, and $i_3$ do not span the entire iteration space $N^3$ but only a smaller tetrahedron ($0 ≤ i_1 ≤ i_2 ≤ i_3 < N$). The smaller iteration space is required due to the mechanics introduced by the new loop in line 5. This loop iterates over all $d!$ permutations (line 5) of the loop variables $i_1$, $i_2$, and $i_3$ (e.g., $(i_1, i_2, i_3)$, $(i_1, i_3, i_2)$, $(i_2, i_1, i_3)$). The conceptual idea behind this loop is illustrated in Figure 1. Consider the spin summation $\mathcal{B} \leftarrow (\pi_{123} + \pi_{321} + \pi_{132})(\mathcal{A})$ (see Figure 1(a)): each node on the left (e.g., $\mathcal{B}_{i_1i_2i_3}$, $\mathcal{B}_{i_1i_2i_3}$) denotes a different entry of $\mathcal{B}$ for some fixed $i_1, i_2, i_3 \in \{0, 1, \ldots, N-1\}$ and is computed in one iteration of the innermost loop (see line 5 in Algorithm 2); likewise, the nodes on the right (e.g., $\mathcal{A}_{i_1i_2i_3}$, $\mathcal{A}_{i_1i_2i_3}$) denote different entries of $\mathcal{A}$. The edges between the nodes represent

---

3 We ignore special handling of cases where any of the indices $i_1$, $i_2$, or $i_3$ are identical for better readability. The actual implementation takes care of these cases.

4 This loop is only visible in this listing, the actual implementation fully unrolls this loop such that all $d!$ updates statements are explicit, and it omits redundant updates when any of $i_1$, $i_2$, or $i_3$ are equal.
dependencies; the number of outgoing edges of each node depends on the actual spin summation (compare Figure 1(a) and Figure 1(b)). For instance, in Figure 1(a), the first iteration of the innermost loop computes $B_{i_1 i_2 i_3} \leftarrow A_{i_1 i_2 i_3} + A_{i_3 i_1 i_2}$, whereas the second iteration computes $B_{i_1 i_2 i_3} \leftarrow A_{i_1 i_3 i_2} + A_{i_3 i_2 i_1} + A_{i_1 i_2 i_3}$.

The colors in Figure 1 identify different connected components; the separation into connected components is not yet important, but we revisit this property in Section 4.6.4.

Each iteration of the innermost loop accesses exactly six memory locations of $A$ (lines 6 and 7). Across all $d!$ iterations of this loop, this totals $6 \times d!$ accesses to $A$; however, only $d!$ of these memory locations are distinct. Thus, every element of $A$, once loaded, is reused exactly six times, which results in perfect temporal locality for both tensors (as long as at least $d!$ elements can be stored in cache). Phrased differently, no element of $A$ (or $B$) needs to be loaded twice if the memory subsystem would operate at the granularity of an element as opposed to a cacheline.

Figure 2 illustrates the underlying idea of Algorithm 2 on the 2D example $B \leftarrow (\alpha_1 \pi_{12} + \alpha_2 \pi_{21})(A)$. The two update statements (see the right-hand side of Figure 2) update distinct areas of $B$ (denoted by the shaded regions). Both elements $B_{i_1 i_2 i_3}$ and $B_{i_2 i_1 i_3}$ depend on both $A_{i_1 i_2 i_3}$ and $A_{i_3 i_2 i_1}$. Thus, temporal locality is achieved because both update statements are executed back-to-back, resulting in perfect reuse of the operands.

One of the fundamental problems of this algorithm, however, is its lack of spatial locality. As Figure 2 highlights, the next elements, accessed within the next iteration of the innermost loop, do not all belong to the previously loaded cachelines. In other words, this algorithm does not exhibit spatial locality for all memory accesses. Although this problem does not seem too severe for the 2D example, it becomes significantly accentuated in higher dimensions.

The missing spatial locality in $B$ is especially troublesome for performance for three reasons. First and foremost, although this algorithm provides perfect temporal locality at the element level (i.e., no element needs to be loaded from main memory more than once), it does not exhibit this property at the cacheline level (i.e., by the time another element—of a previously loaded—cacheline is accessed again, this cacheline might already have been evicted from the caches). Second, the scattered memory accesses within the innermost loop (see lines 6 and 7 of Algorithm 2) prevent efficient vectorization. Finally, parallelizing any of the loops associated to $i_1$, $i_2$, and $i_3$ causes false sharing between the threads and thus hampers performance.

4.3 Algorithm 3: Reduced FLOP-Count

So far, we have focused on optimizations that exploit the temporal and spatial locality inherent to spin summations. In contrast, this section aims to reduce the required amount of floating-point

---

5 Two arbitrary nodes $a$ and $b$ belong to the same connected component if and only if there exists a path from $a$ to $b$ or from $b$ to $a$. 

ACM Transactions on Mathematical Software, Vol. 45, No. 1, Article 10. Publication date: March 2019.
operations by exploiting the factorizability of spin summations (Definition 4). Algorithm 3 is implemented in terms of two dependent spin summations (lines 1–5 and 6–10) that are directly translated from their factorized form (Equation (8)); both of these implementations follow the design of Algorithm 1. Thus, the aforementioned drawbacks of this algorithm are also preserved.

We commence by observing that this algorithm requires a significant amount of additional memory, on the order of the problem size, to store the auxiliary tensor $T$. Second, this algorithm reduces the memory accesses to $A$ from $6N^3$ to $3N^3$, but it also adds three additional memory accesses (one write + two reads) per element of $T$, which have not been required before. Although the total amount of memory operations remained unchanged for this test case, other spin summations of interest significantly benefit from this optimization (see Section 6).

### 4.4 Algorithm 4: Reduced FLOP-Count Without Auxiliary Memory

Algorithm 4 is a combination of Algorithm 2 and Algorithm 3; its main goal is to combine their advantages while avoiding the need for an auxiliary tensor.

Instead of implementing each of the spin summations in Algorithm 3 in terms of Algorithm 1, we could alternatively use Algorithm 2, yielding Algorithm 4. This change allows the loops in lines 2 through 4 and 7 through 9 of Algorithm 3 to be fused and—more importantly—reduces the size of the auxiliary tensor $T$ from $N^3$ to merely $3!$ elements. Thus, accesses to $T$ become negligible and do not need to be accounted for any longer; this also reflects positively on the number of memory operations that have to go to main memory. More precisely, if one neglects the presence of any cache and all other side effects (e.g., false sharing), then one could come to the conclusion that Algorithm 4 is preferable over Algorithm 3 because it reduces the amount of memory operations

---

**ALGORITHM 3: Reduced FLOP-count**

```plaintext
#pragma omp parallel for collapse(2) schedule(static)
for (i3 = 0; i3 < N; i3++) do
  for (i2 = 0; i2 < N; i2++) do
    T[i2+i3] -= A[i2+i3] + A[i2+i3] + A[i2+i3]
  end for
#pragma omp parallel for collapse(2) schedule(static)
for (i3 = 0; i3 < N; i3++) do
  for (i2 = 0; i2 < N; i2++) do
    B[i2+i3] -= T[i2+i3] + T[i2+i3]
  end for
end for
```
from \((1+6)N^3\) to \((1+3)N^3\). However, in practice, this is not the case because the memory subsystem does not operate at the granularity of a single element but on an entire cacheline.

Although this algorithm reduces the FLOP-count and removes the requirement for an auxiliary tensor, the drawbacks mentioned in Section 4.2 still apply.

### 4.5 Algorithm 5: Blocking

Algorithm 5 capitalizes on the analysis of the previous sections and combines the benefits of Algorithms 1 through 4. It forms the foundation for the high-performance implementation presented in Section 4.6.

**Algorithm 5**: The blocked algorithm (left) is implemented in terms of a macrokernel (right). Each invocation of the macrokernel calculates \(d\) blocks of size \(bl^3\) of the output tensor. \(T\) refers to auxiliary memory. We ignore special handling of cases where any of \(\tilde{i}_1\), \(\tilde{i}_2\), or \(\tilde{i}_3\) are identical to improve the readability of the pseudocode.

Algorithm 4 already exhibits some desirable properties such as reduced FLOP-count and temporal locality for individual elements; however, this locality does not extend to entire cachelines. Thus, the main objective of Algorithm 5 is to provide temporal locality at the granularity of an entire cacheline (i.e., a cacheline is only loaded from main memory once). The key optimization to enable such a design is blocking (see Section 3.3).

Algorithm 5 is very similar to Algorithm 4 with the critical difference that Algorithm 5 operates on blocks of size \(bl^3\) as opposed to scalars. The calculation of these blocks is separated out into a so-called macrokernel (see Algorithm 5b). All invocations of the macrokernel are independent from one another and can therefore be computed in parallel by different threads. In contrast to the loop-based parallelism (Algorithms 1–4), this algorithm uses task-based parallelism. More precisely, a single thread (line 2 in Algorithm 5a) generates all the tasks (lines 6 and 7 in Algorithm 5a), which are then dynamically processed by any of the available threads.

The most important parameter in this algorithm is the blocksize \(bl^3\). This parameter is subject to several constraints imposed by our high-performance objective. First, \(bl^3\) should be a multiple of the cacheline size (i.e., 8 double-precision elements) to optimally exploit spatial locality. Second, all \(2 \times 3!\) blocks—per thread—must fit into the shared last-level cache. Moreover, \(bl^3\) should be as large as possible to facilitate more memory accesses to consecutive elements and (if possible) take

---

6For the course of this discussion, we assume that \(N\) is evenly divisible by the blocksize \(bl^3\). However, the actual implementation does not have this constraint and generalizes to nonsquare blocks and thus works for arbitrary sizes.
advantage of the adjacent cacheline prefetching feature of modern CPUs; moreover, larger blocks reduce the parallelization overhead introduced by the task-based parallelization scheme. These constraints respectively suggest $bl = 16$ and $bl = 8$ for 3D and 4D spin summations. More precisely, 3D and 4D spin summations respectively require 384KiB and 1,536KiB of cache per thread; this fits well into the available last-level cache of modern CPUs (e.g., an Intel Xeon E5-2680 v3 offers 2,560KiB of L3 per core).

The scratchpad memory $T$ (see Algorithm 5b) is allocated as a contiguous chunk of memory to avoid severe cache conflicts (caused by the limited associativity of the caches). Moreover, each thread initializes its own portion of $T$ individually to account for the nonuniform memory accesses of modern CPUs. We point out that—despite our best efforts to keep all accessed blocks in cache—this algorithm may still experience few redundant loads from main memory due to the accesses to $A$; we suggest an optimization for this scenario in Section 4.6.4.

The small—yet important—distinction between operating on blocks (Algorithm 5) as opposed to scalars (Algorithm 4) entails several positive effects on performance. First and foremost, provided that the blocks remain in some level of the cache hierarchy, this is the first algorithm that achieves temporal locality for entire cachelines for both $A$ and $B$. Second, this algorithm does not experience any false sharing between the threads, as $bl$ is chosen to be a multiple of the cache-line size. Finally, the abundantly available parallelism and the dynamic tasked-based scheduling together provide good load balancing; for instance, a tensor as small as $40^4$ elements (equivalent to 19.5MiB) already offers 70 independent tasks.

All in all, Algorithm 5 exhibits many desirable properties, but it does not offer spatial locality in the sense that a cacheline—once loaded into L1 cache—is consumed at once; most of the elements of this cacheline are accessed at a later stage. This delayed access necessitates an additional load of the cacheline from some higher level (i.e., L2 or L3) of the cache hierarchy. This disadvantage is addressed in the next section.

4.6 Algorithm 6: High Performance

We combine the underlying ideas of the previous implementations into a high-performance algorithm. All optimizations introduced in this section only affect the macrokernel (Algorithm 5b); the remainder is identical to Algorithm 5a.

4.6.1 Restructured Block Traversal. The main challenge for a high-performance macrokernel is the strided memory accesses of Algorithm 5b. For instance, each of the loop variables (i.e., $i_1$, $i_2$, $i_3$) associated to the loops in lines 2 through 4 of Algorithm 5b is a stride-one index for either the output or the input tensor at some iteration of the loop in line 5. This circumstance poses significant problems with respect to spatial locality and makes efficient vectorization impossible.

Algorithm 6 outlines the new macrokernel; despite the fact that it looks quite different from its predecessor (Algorithm 5b), they are indeed logically identical. The critical difference, however, is that Algorithm 6 traverses the $d!$ blocks (each of size $bl^d$) in a different order; identical blocks—those blocks for which $\tilde{i}_1$, $\tilde{i}_2$, and $\tilde{i}_3$ appear in the same order—are colored identically. Algorithm 6 can be deduced from Algorithm 5b via the following steps. (1) Pull the loop starts (i.e., $\tilde{i}_1$, $\tilde{i}_2$, and $\tilde{i}_3$) into the $A_{...}$ notation; this simple change renders all loops indistinguishable from one another (i.e., all loops have the same start position and loop trip count). (2) Now that the loops in lines 2 through 4 are indistinguishable, rename the loop variables for each update statement (lines 5–10) individually; different names correspond to different traversal orders. Any traversal is valid from a correctness perspective; however, some traversal orders are preferable from a high-performance perspective.

The blocks of $A$ are not consecutive in main memory and therefore may cause cache conflicts.
Thus, we rename the loop variables in each line such that both the input and output tensors are only accessed by at most two different stride-one indices (i.e., the leftmost indices: \(i_1\) and \(i_3\)).

Now that we only have to deal with at most two different stride-one indices (Algorithm 6), we can reorder the loops (lines 2–4) such that the loops associated to the stride-one indices are the innermost. This strategy greatly improves spatial locality and enables efficient vectorization along these loops (see the next section).

### 4.6.2 Vectorization

The vectorization scheme for spin summations is quite similar to that of tensor transpositions (Springer et al. 2016, 2017a) in the sense that both operate on 2D faces of arbitrary-dimensional tensors. However, in contrast to tensor transpositions, spin summations have to load, scale, and add multiple faces—not just one. All of these faces have to share at most two different stride-one indices to utilize fully vectorized memory operations (avoiding scatter and gather).

Figure 3 illustrates the vectorization on the example of the 2D spin summation \(B \leftarrow (\alpha_1 \pi_{12} + \alpha_2 \pi_{21})(\mathcal{A})\). The vectorized macrokernel proceeds as follows. (1) Load the two \(bl^2\) blocks of \(\mathcal{A}\) into the cache (mostly L1 and L2). (2) Add the corresponding transposed blocks \(\mathcal{A}^T\) to yield the output \(B\). Finally, (3) store these blocks to the correct locations in \(B\). All of these steps are fully vectorized (see Springer et al. 2016, 2017a) for details, exploiting spatial locality.

Given that our vectorization scheme relies on the fact that one can identify a traversal that exposes at most two different stride-one indices, the natural question is whether one can always restructure the traversal within the macrokernel such that this requirement is met. Although most of the spin summations of interest are vectorizable without any additional work, some require an additional regularization step. For instance, the spin summation \(B \leftarrow (\alpha_1 \pi_{123} + \alpha_2 \pi_{213} + \alpha_3 \pi_{321})(\mathcal{A})\) has three different stride-one indices; thus, any renaming strategy of the indices is bound to fail. However, such a situation can be avoided by a regularization step that transposes the input tensor \(\mathcal{A}\) prior to the actual spin summation and account for this change at a later stage (similar to Equations (9) and (10)). More precisely, instead of computing \(B \leftarrow (\alpha_1 \pi_{123} + \alpha_2 \pi_{213} + \alpha_3 \pi_{321})(\mathcal{A})\) directly, one could also transpose \(\mathcal{T} \leftarrow \pi_{213}(\mathcal{A})\) and compute the output as

\[
\text{Algorithm 6: High-Performance macro-kernel. Identical sub-tensors are highlighted by the same color. The notation } \mathcal{T}_{i_1,i_2,i_3} \text{ is equivalent to } \mathcal{T}_{i_1,i_2,i_3} \uparrow \iota_3; \text{ it is merely used to separate the loop variables, } i_1, i_2, i_3 \text{ from the offsets, } \tilde{i}_1, \tilde{i}_2, \tilde{i}_3.\]
Fig. 3. Schematic overview for the vectorization of the 2D spin summation $B \leftarrow (\alpha_1 \pi_{12} + \alpha_2 \pi_{21})(A)$. Same blocks are colored identically; each block denotes a 2D face consisting of $bl^2$ elements. The transpose of a block is indicated by a double edge.

$B \leftarrow (\alpha_1 \pi_{213} + \alpha_2 \pi_{123} + \alpha_3 \pi_{231})(T)$ that only has two different stride-one indices; this procedure is applicable to all the spin summations in which we are interested and enables efficient vectorization.

### 4.6.3 Nontemporal Stores

To avoid the write-allocate traffic (see Section 3.2) incurred by the writes to the output tensor $B$, we use nontemporal store instructions whenever possible. It is important to realize that nontemporal stores should only be applied to the writes to $B$ and not to $T$, as the latter is reused repeatedly. More importantly, failing to adhere to this policy results in poor performance because cachelines written via nontemporal stores are marked as invalid, and later accesses to those cachelines incur redundant loads from main memory.

To make efficient use of nontemporal stores, it is critical to reorder the loops of the macrokernel such that writes to the same cacheline occur soon after one another.$^8$ Such writes enable the underlying microarchitecture to write-combine successive writes to the same cacheline into a single memory transaction. Thus, we order the loops within the macrokernel such that the loop associated to the stride-one index of $B$ is the innermost loop (see Algorithm 6); failing to do so results in redundant reads/writes from/to main memory and thus significantly lowers performance.

### 4.6.4 Cache Optimization

Section 4.5 already pointed out that the blocking scheme might require redundant reads from main memory if the accesses to $A$ cause conflict misses (Bryant et al. 2003). This problem may be averted by (1) larger caches, (2) larger cache associativity, or (3) smaller cache requirements of the algorithm. Although we clearly cannot do anything about (1) and (2), there is some opportunity for (3).

Recall that the blocking scheme (see Section 4.5) requires each thread to keep all of its blocks—traversed within the macrokernel—in some level of the cache hierarchy. More precisely, computing a spin summation over $d$-dimensional double-precision tensors with $m$ threads and blocks of size $bl^d$ requires a minimal L3 cache size $C_{\text{min}}$:

$$C_{\text{min}} = bl^d \times m \times x \times 8 \text{ bytes},$$

where $x = 2 \times d!$ represents the number of blocks traversed within the macrokernel. To reduce $C_{\text{min}}$, one may try to decrease either $bl$, $m$, or $x$. As discussed previously, decreasing $bl$ is not desirable and leads to poor performance (data not shown). Similarly, measurements show that reducing

---

$^8$On an AVX-enabled architecture two nontemporal stores (each writing 32 bytes) are required to update an entire cacheline (64 bytes).
the number of threads \( m \) to a point where some CPU cores are idle also yields bad performance (data not shown) because the benefits of more last-level cache—per thread—are countered by the reduced parallelism. This leaves us with the last option: reducing \( x \) (i.e., using fewer blocks per macrokernel).

Consider the spin summation \( \mathcal{B} \leftarrow (\pi_{123} + \pi_{213})(\mathcal{A}) \) outlined in Figure 1(b), where identically colored nodes correspond to update statements that belong to the same connected component and therefore share operands. For instance, the two update statements \( \mathcal{B}_{i_1i_2i_3} \leftarrow \mathcal{A}_{i_1i_2i_3} + \mathcal{A}_{i_2i_1i_3} \) and \( \mathcal{B}_{i_2i_1i_3} \leftarrow \mathcal{A}_{i_2i_1i_3} + \mathcal{A}_{i_1i_2i_3} \) —highlighted in blue—share the same elements (i.e., \( \mathcal{B}_{i_1i_2i_3}, \mathcal{B}_{i_2i_1i_3}, \mathcal{A}_{i_1i_2i_3}, \mathcal{A}_{i_2i_1i_3} \)), but they do not use any of the elements belonging to other updates; the same analysis extends to blocks instead of elements. Thus, blocks of \( \mathcal{B} \) that belong to different connected components may be computed separately from one another without losing any spatial locality or temporal locality. For instance, separating a spin summation over \( d \)-dimensional tensors into \( c \) connected components reduces the amount of blocks \( x \) from \( 2 \times d! \) to \( 2 \times \frac{d!}{c} \). Thus, this technique can reduce the pressure on the cache hierarchy significantly.

This optimization is not applicable to all spin summations. For instance, for the spin summations \( \mathcal{B} \leftarrow (\pi_{123} + \pi_{321} + \pi_{132})(\mathcal{A}) \) (Figure 1(a)), a decomposition into independent connected components is not possible.

### 4.6.5 In-Place

In-place spin summations (i.e., \( \mathcal{A} \leftarrow \sum_{i=1}^{n} \alpha_{i\omega_1}(\mathcal{A}) \)) facilitate another effective opportunity to avoid the write-allocate traffic. Although high-performance implementations of in-place tensor transpositions for 2D, nonsquare tensors are a hard problem (Ding 2001; He and Ding 2002), our algorithm is (almost) immediately applicable to in-place, “hypersquare” spin summations. The support for in-place spin summations is mostly attributed to the auxiliary scratchpad memory \( T \) and the temporal locality of the algorithm. For instance, instead of directly computing the factorized spin summation \( \mathcal{T} \leftarrow (\pi_{123} + \pi_{321} + \pi_{132})(\mathcal{A}) \) followed by \( \mathcal{B} \leftarrow (\pi_{123} + \pi_{213})(\mathcal{T}) \), we change the latter statement to \( \mathcal{A} \leftarrow (\pi_{123} + \pi_{213})(\mathcal{T}) \). This is possible because all memory locations in \( \mathcal{A} \) that will be overwritten have already been read (and will not be accessed again). Some spin summations of interest, however, do not require any scratchpad memory because they are not factorizable (e.g., \( \mathcal{B} \leftarrow (\pi_{123} + \pi_{321} + \pi_{132})(\mathcal{A}) \)). For these spin summations, we introduce an additional preprocessing step—similar to the regularization step outlined in Section 4.6.2—that copies the blocks of \( \mathcal{A} \) into temporal blocks \( \mathcal{T} \). For instance, the in-place spin summation \( \mathcal{A} \leftarrow (\pi_{123} + \pi_{321} + \pi_{132})(\mathcal{A}) \) is computed in two stages: (1) \( \mathcal{T} \leftarrow \pi_{123}(\mathcal{A}) \) followed by (2) \( \mathcal{A} \leftarrow (\pi_{123} + \pi_{321} + \pi_{132})(\mathcal{T}) \); the amount of additional memory is very limited because the size of these blocks is chosen such that they all fit into the caches simultaneously.

### 4.7 Summary

Table 1 summarizes the properties of all previously mentioned algorithms. Although Algorithms 1 through 5 score positively on some properties and negatively on others, the high-performance algorithm (see Section 4.6) is the only one that scores positively across all properties.

### 5 Measurement Environment and Benchmark

We evaluate the performance of Algorithms 1 through 6 on a two-socket system consisting of two Intel Xeon E5-2680 v3 CPUs (12 cores each). The compiler of choice is Intel’s icpc 16.0.2 using the -O3 -qopenmp -xHost options. Moreover, we use one thread per physical core (i.e., 24 threads in total) and pin logically neighboring threads to physically neighboring cores.\(^9\)

\(^9\)The thread affinity is set via the environment variable KMP AFFINITY = compact, 1.
Table 1. Summary Over All Features of Each Implementations

| Algo | TMP Locality | SP Locality | VEC | LB | FLOP | MEM | IP | NTS |
|------|--------------|-------------|-----|----|------|-----|----|-----|
| 1    | -            | +           | -   | +  | -    | +   | -  | -   |
| 2    | +            | +           | -   | -  | +    | -   | -  | -   |
| 3    | -            | +           | -   | +  | +    | -   | -  | -   |
| 4    | +            | +           | -   | -  | +    | -   | -  | -   |
| 5    | +            | o           | o   | -  | +    | +   | +  | -   |
| 6    | +            | +           | +   | +  | +    | +   | +  | +   |

TMP locality, SP locality, VEC, LB, FLOP, MEM, IP, and NTS respectively stand for temporal locality, spatial locality, vectorization, load balancing, reduced FLOP-count, reduced auxiliary memory requirements, support for in-place summations, and support for nontemporal stores. “+,” “o,” and “-” respectively reflect positively, neutrally, or negatively on the corresponding property.

Given a spin summation (see Equation (4)) summing over \( n \) permutations of an input tensor \( \mathcal{A} \in \mathbb{R}^{N^d} \), we report the bandwidth (BW) and the floating-point performance (FP) as follows:

\[
BW = \frac{2 \times N^d \times \text{sizeof(double)}}{2^n \times \text{Time}} \text{ GiB/s},
\]

\[
FP = \frac{2 \times n \times N^d}{10^9 \times \text{Time}} \text{ GFLOP/s}.
\]

Notice that the bandwidth metric is especially conservative since we only account for one read of the input tensor \( \mathcal{A} \), independent of the number of permutations \( n \). The floating-point performance, however, accounts for the effective FLOP-count corresponding to the FLOPs performed by Algorithms 1 and 2.

To put our results into perspective, we respectively report the STREAM (McCalpin 1995) copy- (b ← a), scale- (b ← αa), add- (c ← a + b), and triad-bandwidth (c ← αa + b) for the full system: 101.2GiB/s, 101.3GiB/s, 107.6GiB/s, and 107.7GiB/s. Moreover, the total theoretical peak floating-point performance of our two-socket system is 1,113.6GFLOP/s; MKL attains 845GFLOP/s for a large matrix-matrix multiplication.

All performance results are based on the minimum execution time over several runs to get the most stable timings; the caches are cleared before every run (i.e., all measurements start on cold data). All computations use double-precision floating-point accuracy.

5.1 Benchmark

Table 2 lists all spin summations required for the nonorthogonally spin-adapted CCSDT (Noga and Bartlett 1987) and CCSDTQ (Kucharski and Bartlett 1992) quantum chemistry methods, as well as their approximate forms, such as CCSDT(Q) (Bomble et al. 2005). We test the performance of our implementations with three different problem sizes: small, medium, and large respectively corresponding to tensors roughly of size 70MiB, 320MiB, and 1,200MiB. Unless otherwise mentioned, we present results for the medium-size problems.

6 PERFORMANCE EVALUATION

Figure 4 summarizes the performance of the discussed algorithms across the set of 21 spin summations (see Table 2). We make the following key observations. (1) Algorithms 1 through 4 (■, □, △, ▼) show ambivalent performance for the different test cases with no clear loser nor winner among them. For instance, Algorithm 1 performs well on test cases 2, 3, 4, 19, and 20, but it greatly lags behind any of Algorithms 2 through 4 for test cases 1 and 5 through 17. (2) Although
Test Case $B ← A$ is implemented in terms of Algorithm −
(2 − $π_{231} - π_{132}$(A)).

Algorithm 3 ($\mathbb{B}$) is implemented in terms of Algorithm 1 ($\mathbb{B}$), they exhibit significantly different performance results for all test cases for which the FLOP-count can be reduced (i.e., 1, 5–17). For instance, the difference between the attained bandwidth of Algorithms 1 and 3 is largest for test case 5; this matches our expectations because this test case corresponds to the spin summation for which the FLOP-count is reduced the most. (3) Algorithm 5 ($\mathbb{B}$) is consistently faster than any of its predecessors; its superior performance can be attributed mostly to its improved temporal locality. (4) Algorithm 6 ($\mathbb{B}$), with all of its optimizations (e.g., vectorization, regularization), yields a significant improvement over its blocked but nonvectorized, predecessor ($\mathbb{B}$) across all spin summations. (5) Although test cases 6 through 17 require the same amount of FLOPS and memory accesses, they attain noticeably different bandwidth. This is a similar observation that we already made in a previous publication (Springer et al. 2017a), indicating that some transpositions are inherently more difficult than others. For instance, test cases 4, 8, 11, 14 through 17, and 21 require an additional regularization step (see Section 4.6.1) to enable vectorization. (6) The attained effective floating-point performance (see Figure 4(b)) greatly varies across the benchmark, reflecting that some spin summations benefit from the FLOP-count optimization more than others. More precisely, the theoretical FLOP-count reduction for test cases 1, 2 through 4, 5, 6 through 17, and 18 through 21 respectively is 2×, 1×, 4×, 2×, and 1×.

### 6.1 Optimizations

Figure 5 highlights the effect of various optimizations (see Sections 4.6.3–4.6.5) on the performance of Algorithm 6. The reference for the speedups is Algorithm 6 with all optimizations deactivated; similarly, the measurements labeled NTS, cache-opt (●) correspond to the results in Figure 4.

Table 2. Spin Summations

| ID | Test Case |
|----|-----------|
| 1  | $B ← (2 − π_{213}((2 − π_{321} - π_{132}$(A))) |
| 2  | $B ← (2 − π_{321} - π_{132}$(A)) |
| 3  | $B ← (2 − π_{213} - π_{132}$(A)) |
| 4  | $B ← (2 − π_{213} - π_{321}$(A)) |
| 5  | $B ← (2 − π_{213}((2 − π_{3214} - π_{1324})((2 − π_{4231} - π_{1432} - π_{1243}$(A))) |
| 6  | $B ← (2 − π_{3214} - π_{1324})((2 − π_{4231} - π_{1432} - π_{1243}$(A))) |
| 7  | $B ← (2 − π_{2134} - π_{1324})((2 − π_{4231} - π_{1432} - π_{1243}$(A))) |
| 8  | $B ← (2 − π_{2134} - π_{3214})((2 − π_{4231} - π_{1432} - π_{1243}$(A))) |
| 9  | $B ← (2 − π_{4231} - π_{1432})((2 − π_{3214} - π_{1324} - π_{1243}$(A))) |
| 10 | $B ← (2 − π_{2134} - π_{1432})((2 − π_{3214} - π_{1324} - π_{1243}$(A))) |
| 11 | $B ← (2 − π_{2134} - π_{4231})((2 − π_{3214} - π_{1324} - π_{1243}$(A))) |
| 12 | $B ← (2 − π_{4231} - π_{1432})((2 − π_{2134} - π_{1324} - π_{1243}$(A))) |
| 13 | $B ← (2 − π_{3214} - π_{1432})((2 − π_{2134} - π_{1324} - π_{1243}$(A))) |
| 14 | $B ← (2 − π_{3214} - π_{4231})((2 − π_{2134} - π_{1324} - π_{1243}$(A))) |
| 15 | $B ← (2 − π_{1432} - π_{1432})((2 − π_{2134} - π_{3214} - π_{1243}$(A))) |
| 16 | $B ← (2 − π_{1432} - π_{1432})((2 − π_{2134} - π_{3214} - π_{1243}$(A))) |
| 17 | $B ← (2 − π_{1432} - π_{1432})((2 − π_{2134} - π_{3214} - π_{1432}$(A))) |
| 18 | $B ← (2 − π_{4231} - π_{1432} - π_{1243}$(A)) |
| 19 | $B ← (2 − π_{3214} - π_{1324} - π_{1243}$(A)) |
| 20 | $B ← (2 − π_{2134} - π_{1324} - π_{1432}$(A)) |
| 21 | $B ← (2 − π_{2134} - π_{3214} - π_{4231}$(A)) |

Depending on the context, “2” is short for $2π_{123}$ or $2π_{1234}$. 
Fig. 4. Performance of all algorithms across the benchmark.

Fig. 5. Impact of various optimization techniques on the performance of Algorithm 6. Reference: Algorithm 6 without any optimizations. ▼ cache optimization; ▲ nontemporal stores; ◆ nontemporal stores and cache optimization; ● all optimizations and in-place.
The following is evident from Figure 5. First, the cache optimization (▼, see Section 4.6.4) only has an effect on the performance for 4D spin summations; this is expected because 3D spin summations only require a small portion of the available last-level cache, making cache conflicts less likely. Moreover, the positive effect of the cache optimization is evident both in the presence or absence of nontemporal stores. Second, activating nontemporal stores (▲, see Section 4.6.3) is critical for performance. The speedup due to non-temporal stores, in some case, can be as high as 1.5x; such a speedup is perfectly in line with the fact that the amount of data transferred from the caches is reduced by 33% (see Section 4.6.3). Finally, in-place spin summations (●, see Section 4.6.5) not only require less memory but also increase the performance over its out-of-place counterpart (●) by up to 1.2x.

6.2 Summary

Figure 6 summarizes the performance of Algorithm 6 for different problem sizes. Since Algorithm 2 constitutes the current implementation of spin summation in the NCC quantum chemistry software package (Matthews and Stanton 2015), Figure 6(b) reports the speedup of Algorithm 6 with respect to Algorithm 2.

---

10The maximum theoretical speedup of 1.5x assumes that the accessed data to both tensors is entirely cached and that the write-allocate traffic is avoided.
The average speedups for the small (▼), medium (▲), and large (◆) problems respectively vary between 2.4 and 4.3×, 2.4 and 3.8×, and 3.3 and 5.5×. Thus, our high-performance implementation substantially outperforms the reference for all spin summations and tested problem sizes. Although the speedups for the large problems are the highest, we point out that these speedups are caused by the combination of (1) slightly increased performance of Algorithm 6 (see Figure 6(a)) and (2) decreased performance of Algorithm 2 (i.e., the reference).

7 CONCLUSION

We tackled the problem of making the computation of 3D and 4D spin summations as efficient as possible. We presented a systematic way to restructure the memory accesses so that both the temporal and spatial locality, inherent to the operation, are exploited; the resulting algorithm also takes advantage of the factorizability of spin summations, thus significantly reducing the required FLOP-count. The lesson learned is that it is not one optimization in isolation that makes the difference but rather the combination of all of them. For instance, the main techniques introduced in Sections 4.1 through 4.4 (e.g., spatial locality, temporal locality, and reduced FLOP-count) show ambiguous performance results if applied in isolation. However, the integration of all of these ideas into Algorithm 6 yields noticeable speedups between 2.4× and 5.5× over the current reference implementation. Furthermore, our algorithm allows for spin summations to be performed in-place; this desirable property not only reduces the memory footprint by a factor of 2 but also reflects positively on the overall performance.

We plan to incorporate our high-performance implementation of spin summation based on Algorithm 6 into the NCC quantum chemistry software package. In concert with other ongoing work in optimizing tensor contraction (Matthews 2018; Springer and Bientinesi 2018), the speedups achieved in this work should allow for large-scale CCSDT, CCSDTQ, and CCSDT(Q) calculations to run at near-peak efficiency on modern multi- and many-core systems. This is exciting, given that previous experience has been that these calculations only achieved a small fraction of peak performance.

As already mentioned, the sums of tensor transpositions appear in other operations beyond spin summations. We believe the techniques detailed in this work are also applicable for tensor unpacking, and for the antisymmetrization operations required in open-shell calculations using an unrestricted Hartree-Fock reference.

ACKNOWLEDGMENT

We thank our colleagues in the HPAC research group for many fruitful discussions and valuable feedback.

REFERENCES

B. G. Adams and J. Paldus. 1979. Orthogonally spin-adapted coupled-cluster theory for closed-shell systems including triexcited clusters. Physical Review A 20, 1 (1979), 1.
Roberto Ansaloni, Gian Luigi Bendazzoli, Stefano Evangelisti, and Elda Rossi. 2000. A parallel full-CI algorithm. Computer Physics Communications 128, 1 (2000), 496–515. DOI: https://doi.org/10.1016/S0010-4655(99)00542-1
Edoardo Apra, Michael Klemm, and Karol Kowalski. 2014. Efficient implementation of many-body quantum chemical methods on the Intel® Xeon Phi coprocessor. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. IEEE, Los Alamitos, CA, 674–684.
Rodney J. Bartlett and Monika Musiał. 2007. Coupled-cluster theory in quantum chemistry. Reviews in Modern Physics 79, 1 (2007), 291–352.
G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, et al. 2005. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proceedings of the IEEE 93, 2 (2005), 276–292. DOI: https://doi.org/10.1109/JPROC.2004.840311
Yannick J. Bomble, John F. Stanton, Mihály Kállay, and Jürgen Gauss. 2005. Coupled-cluster methods including noniterative corrections for quadruple excitations. *Journal of Chemical Physics* 123 (Aug. 2005), 4101. DOI: https://doi.org/10.1063/1.1950567

Randal Bryant and David Richard O’Hallaron. 2003. *Computer Systems: A Programmer’s Perspective*. Vol. 2. Prentice Hall, Upper Saddle River, NJ.

R. J. Buenker and S. Krebs. 1999. The configuration-driven approach for multireference configuration interaction calculations. In *Recent Advances in Multireference Methods*, K. Hirao (Ed.). World Scientific, River Edge, NJ, 1–29.

Bryan Catanzaro, Alexander Keller, and Michael Garland. 2014. A decomposition for in-place matrix transposition. *ACM SIGPLAN Notices* 49, 8 (2014), 193–206.

S. Chatterjee and S. Sen. 2000. Cache-efficient matrix transposition. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA-6)*. 195–205. DOI: https://doi.org/10.1109/HPCA.2000.824350

J. Čížek. 1966. On the correlation problem in atomic and molecular systems. Calculation of wavefunction components in Ursell-type expansion using quantum-field theoretical methods. *Journal of Chemical Physics* 45, 11 (1966), 4256–4266. DOI: https://doi.org/10.1063/1.1727484

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, et al. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC’08)*. IEEE, Los Alamos, CA, 4:1–4:12. http://dl.acm.org/citation.cfm?id=1413570.1413575

Edoardo Di Napoli, Diego Fabregat-Traver, Gregorio Quintana-Orti, and Paolo Bientinesi. 2014. Towards an efficient use of the BLAS library for multidimensional tensor contractions. *Applied Mathematics and Computation* 235 (May 2014), 454–468.

C. H. Q. Ding. 2001. An optimal index reshuffle algorithm for multidimensional arrays and its applications for parallel architectures. *IEEE Transactions on Parallel and Distributed Systems* 12, 3 (2001), 306–315. DOI: https://doi.org/10.1109/71.914776

Zhengting Gan, Yuri Alexeev, Mark S. Gordon, and Ricky A. Kendall. 2003. The parallel implementation of a full configuration interaction program. *Journal of Chemical Physics* 119, 1 (2003), 47–59. DOI: https://doi.org/10.1063/1.1575193

Geoffrey C. Goldbogen. 1981. PRIM: A fast matrix transpose method. *IEEE Transactions on Software Engineering* 7, 2 (1981), 255–257.

Kazushige Goto and Robert A. Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software* 34, 3 (2008), 12.

John A. Gunnels, Greg M. Henry, and Robert A. Van De Geijn. 2001. A family of high-performance matrix multiplication algorithms. In *Proceedings of the International Conference on Computational Sciences (ICCS’01)*. 51–60.

M. Hanrath and A. Engels-Putzka. 2010. An efficient matrix-matrix multiplication based antisymmetric tensor contraction engine for general order coupled cluster. *Journal of Chemical Physics* 133, 6 (2010), 064108. DOI: https://doi.org/10.1063/1.3467878

A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, et al. 2009. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *Journal of Physical Chemistry A* 113, 45 (2009), 12715–12723. DOI: https://doi.org/10.1021/jp9051215

Y. He and C. H. Q. Ding. 2002. MPI and OpenMP paradigms on cluster of SMP architectures: The vacancy tracking algorithm for multi-dimensional array transposition. In *Proceedings of the ACM/IEEE 2002 Conference on Supercomputing*. 6. DOI: https://doi.org/10.1109/SC.2002.10065

T. Helgaker, P. Jørgensen, and J. Olsen. 2013. *Molecular Electronic-Structure Theory*. Wiley, New York, NY.

So Hirata. 2003. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *Journal of Chemical Physics A* 107 (2003), 9887–9897.

Intel Corporation. 2015. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Available at http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

Wanyi Jiang, Yuriy G. Khait, and Mark R. Hoffmann. 2009. Configuration-driven unitary group approach for generalized van Vleck variant multireference perturbation theory. *Journal of Physical Chemistry A* 113, 16 (2009), 4374–4380. DOI: https://doi.org/10.1021/jp9051215

Jose L. Jodra, Ibai Gurrutxaga, and Javier Muguerza. 2015. Efficient 3D transpositions in graphics processing units. *International Journal of Parallel Programming* 43, 5 (Oct. 2015), 876–891.

Michael Klene and Michael A. Robb. 2000. Parallel implementation of the CI-vector evaluation in full CI/CAS-SCF. *Journal of Chemical Physics* 113, 14 (2000), 5653–5665. DOI: https://doi.org/10.1063/1.1290014

ACM Transactions on Mathematical Software, Vol. 45, No. 1, Article 10. Publication date: March 2019.
Andrey Vladimirov. 2013. *Multithreaded Transposition of Square Matrices With Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors*. Available at https://hgpu.org/?p=11002.

Lai Wei and John Mellor-Crummey. 2014. Autotuning tensor transposition. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW’14)*. IEEE, Los Alamitos, CA, 342–351.

Received May 2017; revised July 2018; accepted November 2018