SIAM J. SCI. COMPUT.
Vol. 40, No. 3, pp. C305–C326

STRASSEN'S ALGORITHM FOR TENSOR CONTRACTION*

JIANYU HUANG†, DEVIN A. MATTHEWS‡, AND ROBERT A. VAN DE GEIJN†

Abstract. Tensor contraction (TC) is an important computational kernel widely used in numerous applications. It is a multidimensional generalization of matrix multiplication (GEMM). While Strassen's algorithm for GEMM is well studied in theory and practice, extending it to accelerate TC has not been previously pursued. Thus, we believe this to be the first paper to demonstrate how one can in practice speed up TC with Strassen's algorithm. By adopting a block-scatter-matrix format, a novel matrix-centric tensor layout, we can conceptually view TC as GEMM for a general stride storage, with an implicit tensor-to-matrix transformation. This insight enables us to tailor a recent state-of-the-art implementation of Strassen's algorithm to a recent state-of-the-art TC, avoiding explicit transpositions (permutations) and extra workspace, and reducing the overhead of memory movement that is incurred. Performance benefits are demonstrated with a performance model as well as in practice on modern single core, multicore, and distributed memory parallel architectures, achieving up to 1.3× speedup. The resulting implementations can serve as a drop-in replacement for various applications with significant speedup.

Key words. multilinear algebra, Strassen's algorithm, tensor contraction, matrix multiplication

AMS subject classification. 65F99

DOI. 10.1137/17M1135578

1. Introduction.

This paper builds upon a number of recent developments: the GotoBLAS algorithm for matrix multiplication (GEMM) [14] that underlies the currently fastest implementations of GEMM for CPUs; the refactoring of the GotoBLAS algorithm as part of the BLAS-like Library Instantiation Software (BLIS) [46, 45], which exposes primitives for implementing BLAS-like operations; the systematic parallelization of the loops that BLIS exposes so that high-performance can be flexibly attained on multicore and many-core architectures [37]; the casting of tensor contraction (TC) in terms of the BLIS primitives and avoiding the explicit tensor transposition (a permutation of the data elements in memory) required by traditional implementations by fusion with existing data movement in the GotoBLAS algorithm [32, 40]; the practical high-performance implementation of the classical Strassen's algorithm (STRASSEN) [42] in terms of variants of the BLIS primitives [20]; and the extension of this implementation [19] to a family of Strassen-like fast matrix multiplication algorithms [4]. Together, these results facilitate what we believe to be the first extension of Strassen's algorithm to TC.

Contributions. This work presents the first efficient implementation of Strassen's algorithm for TC, a significant problem with numerous applications.

- It describes how to extend Strassen's algorithm to TC without the explicit transposition of data that inherently incurs significant memory movement and workspace overhead.

*Submitted to the journal's Software and High-Performance Computing section July 10, 2017; accepted for publication (in revised form) March 14, 2018; published electronically May 31, 2018.
http://www.siam.org/journals/sisc/40-3/M113557.html

Funding: This work was partly supported by the National Science Foundation under grants ACI-1550493 and CCF-1714091, by Intel Corporation through an Intel Parallel Computing Center grant, and by a gift from Qualcomm. The second author is an Arnold O. Beckman postdoctoral fellow.

†Department of Computer Science and Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, TX 78712 (jianyu.huang@utexas.edu, rvdg@cs.utexas.edu).

‡Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, TX 78712 (dmatthews@utexas.edu).

- It provides a performance model for the cost of the resulting family of algorithms.
- It details the practical implementation of these algorithms, including how to exploit variants of the primitives that underlie BLIS and a data layout to memory for the tensors.
- It demonstrates practical speedup on modern single core and multicore CPUs.
- It illustrates how the local use of the Strassen's TC algorithm on each node improves performance of a simple distributed memory TC.

Together, these results unlock a new frontier for the research and application of Strassen's algorithm.

2. Background. We briefly review how high-performance GEMM is implemented first, before discussing the practical implementations of high-performance STRASSEN for GEMM.

2.1. High-performance GEMM. Let $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ be matrices of sizes $N_i \times N_p$, $N_p \times N_j$, and $N_i \times N_j$, respectively, and α and β be scalars. A general matrix-matrix multiplication (GEMM) in the BLAS interface [10] is expressed as $\mathbf{C} := \alpha\mathbf{AB} + \beta\mathbf{C}$. Written elementwise, $\mathbf{C}_{i,j} := \alpha \sum_{p=0}^{N_p-1} \mathbf{A}_{i,p} \cdot \mathbf{B}_{p,j} + \beta\mathbf{C}_{i,j}$, where $\cdot$ denotes scalar multiplication. We focus on the special case $\alpha = 1$ and $\beta = 1$ henceforth for brevity.

A key insight underlying modern high-performance implementations of GEMM is to organize the computations by partitioning the operands into blocks for temporal locality and to *pack* (copy) such blocks into contiguous buffers that fit into various levels of memory for spatial locality [14]. Figure 1 (left) illustrates the GOTOBLAS algorithm as implemented in BLIS. Cache blocking parameters $\{m_C, n_C, k_C\}$ determine the submatrix sizes of $\mathbf{B}_p$ ($k_C \times n_C$) and $\mathbf{A}_i$ ($m_C \times k_C$), such that they fit in various caches (we use the standard GEMM dimensions $\{m, n, k\}$ in defining blocking parameters for brevity and consistency with [46] but note that the meaning of $\{m, n, k\}$ alone is changed in section 2.3). During the computation, row panels $\mathbf{B}_p$ are contiguously packed into buffer $\widetilde{\mathbf{B}}_p$ to fit in the L3 cache. Blocks $\mathbf{A}_i$ are similarly packed into buffer $\widetilde{\mathbf{A}}_i$ to fit in the L2 cache. Register block sizes $\{m_R, n_R\}$ relate to submatrices in registers that contribute to $\mathbf{C}$. In the *microkernel* (the inner most loop), a small $m_R \times n_R$ microtile of $\mathbf{C}$ is updated by a pair of $m_R \times k_C$ and $k_C \times n_R$ slivers of $\mathbf{A}_i$ and $\mathbf{B}_p$. The above parameters can be analytically chosen [28].

2.2. High-performance STRASSEN. If the three operands are partitioned into quadrants,

$$\mathbf{A} = \left(\begin{array}{c|c} \mathbf{A}_0 & \mathbf{A}_1 \\ \hline \mathbf{A}_2 & \mathbf{A}_3 \end{array}\right), \mathbf{B} = \left(\begin{array}{c|c} \mathbf{B}_0 & \mathbf{B}_1 \\ \hline \mathbf{B}_2 & \mathbf{B}_3 \end{array}\right), \mathbf{C} = \left(\begin{array}{c|c} \mathbf{C}_0 & \mathbf{C}_1 \\ \hline \mathbf{C}_2 & \mathbf{C}_3 \end{array}\right),$$

then it can be checked that the operations

$$\begin{array}{ll} \mathbf{M}_0 = (\mathbf{A}_0 + \mathbf{A}_3)(\mathbf{B}_0 + \mathbf{B}_3); & \mathbf{C}_0 \mathrel{+}= \mathbf{M}_0; \mathbf{C}_3 \mathrel{+}= \mathbf{M}_0; \\ \mathbf{M}_1 = (\mathbf{A}_2 + \mathbf{A}_3)\mathbf{B}_0; & \mathbf{C}_2 \mathrel{+}= \mathbf{M}_1; \mathbf{C}_3 \mathrel{-}= \mathbf{M}_1; \\ \mathbf{M}_2 = \mathbf{A}_0(\mathbf{B}_1 - \mathbf{B}_3); & \mathbf{C}_1 \mathrel{+}= \mathbf{M}_2; \mathbf{C}_3 \mathrel{+}= \mathbf{M}_2; \\ \mathbf{M}_3 = \mathbf{A}_3(\mathbf{B}_2 - \mathbf{B}_0); & \mathbf{C}_0 \mathrel{+}= \mathbf{M}_3; \mathbf{C}_2 \mathrel{+}= \mathbf{M}_3; \\ \mathbf{M}_4 = (\mathbf{A}_0 + \mathbf{A}_1)\mathbf{B}_3; & \mathbf{C}_1 \mathrel{+}= \mathbf{M}_4; \mathbf{C}_0 \mathrel{-}= \mathbf{M}_4; \\ \mathbf{M}_5 = (\mathbf{A}_2 - \mathbf{A}_0)(\mathbf{B}_0 + \mathbf{B}_1); & \mathbf{C}_3 \mathrel{+}= \mathbf{M}_5; \\ \mathbf{M}_6 = (\mathbf{A}_1 - \mathbf{A}_3)(\mathbf{B}_2 + \mathbf{B}_3); & \mathbf{C}_0 \mathrel{+}= \mathbf{M}_6; \end{array} \tag{2.1}$$

compute $\mathbf{C} \mathrel{+}= \mathbf{AB}$, with seven instead of eight (sub)matrix multiplications, decreasing the total number of arithmetic operations by a factor of 7/8 (ignoring the total

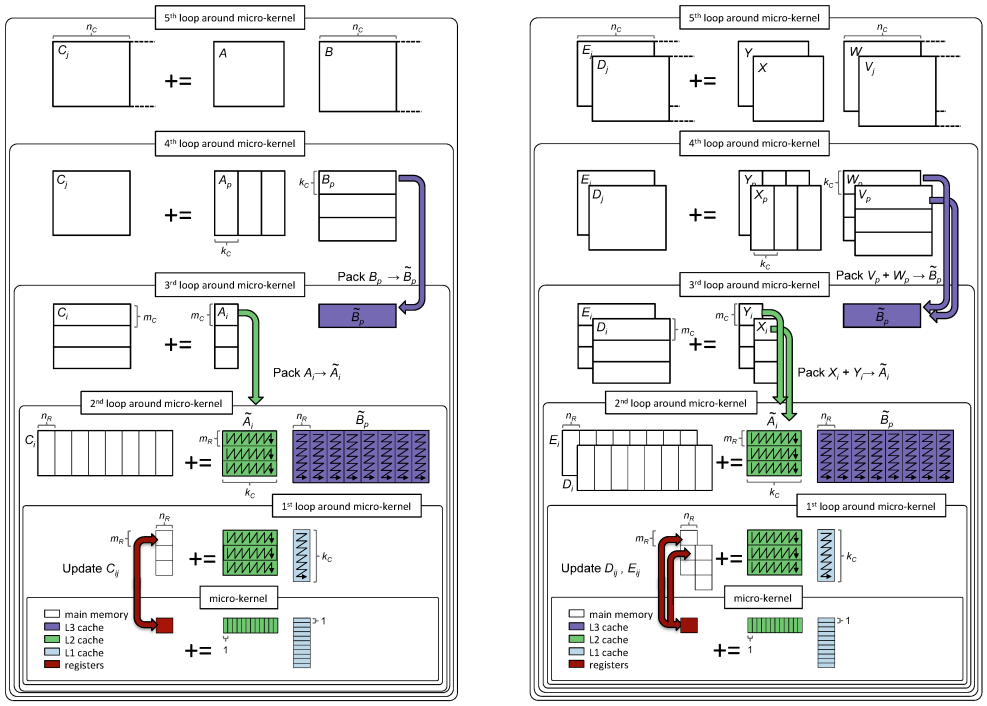


FIG. 1. *Figure from* [20] *(used with permission from the authors). Left: illustration of the BLIS implementation of the* GOTOBLAS *algorithm. All computation is cast in terms of a highly optimized microkernel. Right: modification that implements the representative computation* $\mathbf{M} = (\mathbf{X}+\mathbf{Y})(\mathbf{V}+\mathbf{W})$; $\mathbf{D}+= \mathbf{M}$; $\mathbf{E}+= \mathbf{M}$ *of each row of computations in* (2.1). $\mathbf{X}$, $\mathbf{Y}$ *are submatrices of* $\mathbf{A}$; $\mathbf{V}$, $\mathbf{W}$ *are submatrices of* $\mathbf{B}$; $\mathbf{D}$, $\mathbf{E}$ *are submatrices of* $\mathbf{C}$; $\mathbf{M}$ *is the intermediate matrix product. Note that the packing buffers* $\widetilde{\mathbf{A}}_i$ *and* $\widetilde{\mathbf{B}}_p$ *stay in cache.*

number of extra additions since they are lower-order terms). If all matrices are square and of size $N \times N$, theoretically this single step of STRASSEN [42] can be applied recursively, resulting in the classical STRASSEN with a cost of $O(N^{2.807})$.

In practice, only a few levels of the recursion are leveraged because the reduction in computations is quickly overwhelmed by the cost of extra additions and extra memory movements [4, 7, 20]. Furthermore, STRASSEN is known to experience degradation in numerical stability especially when more than two levels of recursion are incorporated [18, 8, 3].

Figure 1 (right) illustrates the modifications done in [20] to make STRASSEN practical. During the *packing* process, the additions of the submatrices $\mathbf{A}$ and $\mathbf{B}$ can be incorporated into the packing buffers $\widetilde{\mathbf{A}}_i$ and $\widetilde{\mathbf{B}}_p$, avoiding extra memory movement, and reducing workspace requirements. In the *microkernel*, a submatrix that contributes to $\mathbf{C}$ can be directly added to the appropriate parts of multiple submatrices of $\mathbf{C}$, once it is computed in machine registers. This optimization avoids the need for temporary intermediate matrices $\mathbf{M}_i$ and reduces extra memory movement. As shown in [20], this approach makes STRASSEN practical for smaller matrices and matrices of special shape (importantly, for rank-k updates, where N_p is relatively small comparing to N_i and N_j). This research is pushed further [19] by revealing that STRASSEN performs relatively better than most other Strassen-like fast matrix multiplication algorithms with one or two levels of recursions. For this reason, we do not

extend those Strassen-like fast matrix multiplication algorithms to TC in this paper, although it may be worthwhile in future work to pursue certain of these algorithms for highly nonsquare TC shapes.

2.3. High-performance tensor contraction. The definition and notation of tensors and TC are briefly reviewed before describing the tensor layouts that enable high-performance TC.

Tensor. The concept of matrices is extended to multiple dimensions through the use of tensors. For example, consider a three-dimensional (3-D) tensor $\boldsymbol{\mathcal{T}}$ of size $4 \times 6 \times 3$. $\boldsymbol{\mathcal{T}}$ can be thought of as a 3-D array of elements, where each element is given by indexing: $\boldsymbol{\mathcal{T}}_{i,j,k} \in \mathbb{R}$. The possible values for i, j, and k are determined by the lengths of the dimensions as given in the tensor size, i.e., $0 \le i < N_i = 4$, $0 \le j < N_j = 6$, and $0 \le k < N_k = 3$.

In general, a d-dimensional tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{N_{i_0}} \times \cdots \times \mathbb{R}^{N_{i_{d-1}}}$ has elements indexed as $\boldsymbol{\mathcal{T}}_{i_0,\ldots,i_{d-1}} \in \mathbb{R} \,\forall\, (i_0, \ldots, i_{d-1}) \in N_{i_0} \times \cdots \times N_{i_{d-1}}$, where $M \times \cdots \times N$ is a shorthand notation for the set of all tuples $(i, \ldots, j)$, $0 \le i < M \wedge \cdots \wedge 0 \le j < N$. The length of the dimension indexed by some symbol x is given by $N_x \in \mathbb{N}$. The indices may be collected in an ordered *index bundle* $I_d = (i_0, \ldots, i_{d-1})$, such that $\boldsymbol{\mathcal{T}}_{I_d} \in \mathbb{R} \;\forall\; I_d \in N_{i_0} \times \cdots \times N_{i_{d-1}}$. In general we will denote the dimension of a tensor $\boldsymbol{\mathcal{T}}$ as $d_{\boldsymbol{\mathcal{T}}}$ and the bundle length $N_{I_d} \in \mathbb{N}$ as the total length of an index bundle I_d, i.e., $N_{I_d} = \prod_{i \in I_d} N_i = N_{i_0} \cdot \cdots \cdot N_{i_{d-1}}$.

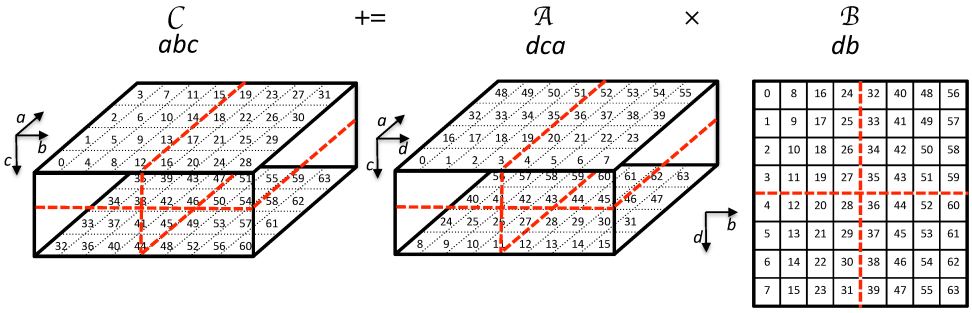
Tensor contraction. TC is the generalization of matrix multiplication to many dimensions. As an example, consider the TC illustrated in Figure 2(a), $\boldsymbol{\mathcal{C}}_{a,b,c} += \sum_{d=0}^{N_d-1} \boldsymbol{\mathcal{A}}_{d,c,a} \cdot \boldsymbol{\mathcal{B}}_{d,b}$. The summation is usually suppressed and instead implied by the Einstein summation convention, where indices that appear twice on the right-hand side are summed over. Additionally, possible scalar factors α and β are ignored for simplicity. In contrast to the definition of matrix multiplication in section 2.1, TC may have more than one index summed over and more than one nonsummed index in each of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$. The groups of indices that correspond to i, j, and p in the matrix case are grouped into index bundles I_m, J_n, and P_k. For this example, the bundles are (a, c), (b), and (d), respectively. Other than involving more indices, TC is precisely the same mathematical operation as matrix multiplication.

For general TCs, let $\boldsymbol{\mathcal{A}}$, $\boldsymbol{\mathcal{B}}$, and $\boldsymbol{\mathcal{C}}$ be tensors of any dimensionality satisfying $d_{\boldsymbol{\mathcal{A}}} + d_{\boldsymbol{\mathcal{B}}} - d_{\boldsymbol{\mathcal{C}}} = 2k$, $k \in \mathbb{N}$. Then, let I_m, J_n, and P_k be index bundles with $m = d_{\boldsymbol{\mathcal{A}}} - k$ and $n = d_{\boldsymbol{\mathcal{B}}} - k$. Last, let the index reordering $\Pi_{\boldsymbol{\mathcal{A}}}((i_0, \ldots, i_{d_{\boldsymbol{\mathcal{A}}}-1})) = (i_{\pi_{\boldsymbol{\mathcal{A}}}(0)}, \ldots, i_{\pi_{\boldsymbol{\mathcal{A}}}(d_{\boldsymbol{\mathcal{A}}}-1)})$ be defined by the bijective map $\pi_{\boldsymbol{\mathcal{A}}} \colon \{0, \ldots, d_{\boldsymbol{\mathcal{A}}} - 1\} \to \{0, \ldots, d_{\boldsymbol{\mathcal{A}}} - 1\}$, and similarly for $\Pi_{\boldsymbol{\mathcal{B}}}$ and $\Pi_{\boldsymbol{\mathcal{C}}}$. The general definition of TC is then given by

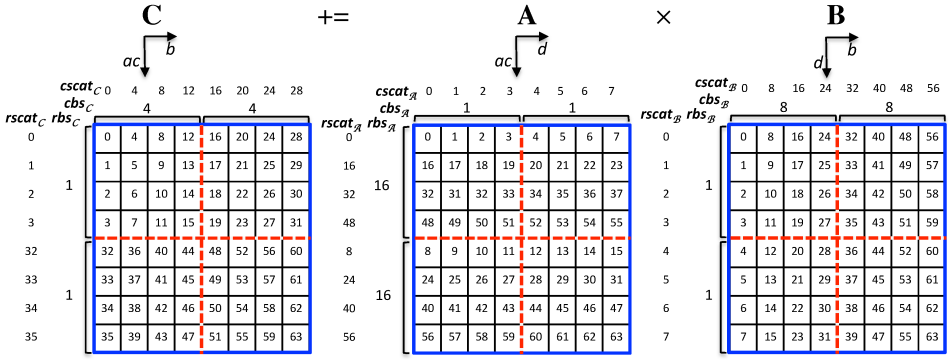
$$\boldsymbol{\mathcal{C}}_{\Pi_{\boldsymbol{\mathcal{C}}}(I_m J_n)} += \sum_{P_k \in N_{p_0} \times \cdots \times N_{p_{k-1}}} \boldsymbol{\mathcal{A}}_{\Pi_{\boldsymbol{\mathcal{A}}}(I_m P_k)} \cdot \boldsymbol{\mathcal{B}}_{\Pi_{\boldsymbol{\mathcal{B}}}(P_k J_n)},$$

where juxtaposition of two index bundles (e.g., $I_m J_n$) denotes concatenation. The indices in the bundles I_m and J_n are generally called *free*, *external*, or *uncontracted* indices, while the indices in the P_k bundle are called *bound*, *internal*, or *contracted* indices.[1] In the following we will suppress the explicit summation over P_k. The number of leading-order floating point operations required for TC is $2N_{I_m} \cdot N_{J_n} \cdot N_{P_k}$

[1]The preferred terms depend on context and the specific field of research. In some cases, these terms have specific meaning beyond the indication of how summation is performed; for example, in quantum chemistry the terms internal and external refer to the diagrammatic representation of TCs [6].



(a) Tensor contraction $\mathcal{C}_{a,b,c} \mathrel{+}= \mathcal{A}_{d,c,a} \cdot \mathcal{B}_{d,b}$ with $N_a = 4$, $N_b = N_d = 8$, and $N_c = 2$. The relative location of each data element in memory is given assuming a generalized column-major layout.



(b) Block scatter matrix view of **(a)**, where $\mathcal{A}_{d,c,a}$, $\mathcal{B}_{d,b}$, and $\mathcal{C}_{a,b,c}$ are mapped to matrices $\mathbf{A}_{i,p}$, $\mathbf{B}_{p,j}$, and $\mathbf{C}_{i,j}$: $rscat_{\mathcal{T}}$ and $cscat_{\mathcal{T}}$ denote the scatter vectors; $rbs_{\mathcal{T}}$ and $cbs_{\mathcal{T}}$ denote the block scatter vectors. Element locations are given by the sum of the row and column scatter vector entries.

FIG. 2. *An example to illustrate Strassen's algorithm for TC. The dashed lines denotes* STRASSEN 2×2 *partitions mapping from block scatter matrix view (bottom) to the original tensor (top). In this example the partitions are regular subtensors, but this is not required in general.*

$= 2(\prod_{i \in I_m} N_i) \cdot (\prod_{j \in J_n} N_j) \cdot (\prod_{p \in P_k} N_p)$. If the length of each dimension is $O(N)$, the TC operation requires $O(N^{m+n+k})$ flops.

The example illustrated in Figure 2(a) has index bundles as noted above and index reordering given by $\Pi_{\mathcal{A}}((i_0, i_1, i_2)) = (i_2, i_1, i_0)$, $\Pi_{\mathcal{B}}((i_0, i_1)) = (i_0, i_1)$, and $\Pi_{\mathcal{C}}((i_0, i_1, i_2)) = (i_0, i_2, i_1)$. Note that, for example, defining I_m as (c, a) would give different index reorderings—the choice of ordering withing the index bundles and the index reorderings is not unique. The number of floating point operations and memory accesses for this contraction is identical to that for a matrix multiplication of $(N_a \cdot N_c) \times N_d$, $N_d \times N_b$, and $(N_a \cdot N_c) \times N_b$ matrices, if performed entirely in place (i.e., without transposition).

General stride layouts. The well-known column-major and row-major matrix layouts may be extended to tensors as the generalized column- and row-major tensor layouts, where elements are stored contiguously along the first dimension or last dimension, respectively. However, in general we may assume only a *general tensor layout*, which extends the general matrix layout [46] by replacing matrix row and

column strides (e.g., $rs_{\mathbf{M}}$ and $cs_{\mathbf{M}}$) with a stride associated to each tensor dimension. For a d-dimensional tensor $\boldsymbol{\mathcal{T}}$ indexed by I_d, the strides $s_{\boldsymbol{\mathcal{T}};i_k} \in \mathbb{N}\ \forall\ 0 \leq k < d$ form the set $S_{\boldsymbol{\mathcal{T}}} = (s_{\boldsymbol{\mathcal{T}};i_0}, \ldots, s_{\boldsymbol{\mathcal{T}};i_{d-1}})$, which gives *g*eneral *s*tride element *loc*ations relative to $\boldsymbol{\mathcal{T}}_{0,\ldots,0}$,

$$\mathrm{LOC}_{GS}(\boldsymbol{\mathcal{T}}_{I_d}, S_{\boldsymbol{\mathcal{T}}}) = \sum_{k=0}^{d-1} i_k \cdot s_{\boldsymbol{\mathcal{T}};i_k}.$$

In general, the stride of the dimension indexed in $\boldsymbol{\mathcal{T}}$ by a particular symbol x is denoted by $s_{\boldsymbol{\mathcal{T}};x}$. The generalized column-major and row-major layouts can also be represented using a general stride layout, in which case $s_{\boldsymbol{\mathcal{T}};i_k} = \prod_{l=0}^{k-1} N_{i_l}$ and $s_{\boldsymbol{\mathcal{T}};i_k} = \prod_{l=k+1}^{d-1} N_{i_l}$, respectively.

In Figure 2(a), $\boldsymbol{\mathcal{C}}$ is stored in the generalized column-major layout. The numbers represent the location of the element $\boldsymbol{\mathcal{C}}_{a,b,c}$ relative to the element $\boldsymbol{\mathcal{C}}_{0,0,0}$ in the tensor storage layout. The strides are $s_{\boldsymbol{\mathcal{C}};a} = 1$, $s_{\boldsymbol{\mathcal{C}};b} = N_a = 4$, and $s_{\boldsymbol{\mathcal{C}};c} = N_a \cdot N_b = 32$. The element location of $\boldsymbol{\mathcal{C}}_{a,b,c}$ is $a \cdot s_{\boldsymbol{\mathcal{C}};a} + b \cdot s_{\boldsymbol{\mathcal{C}};b} + c \cdot s_{\boldsymbol{\mathcal{C}};c} = a + 4b + 32c$.

Block scatter matrix view. In [32] it is shown that tensors can be represented in a matrix-centric layout that allows for a simple but efficient implementation of TC using the BLIS framework. The main idea of that work is that the locations of tensor elements of $\boldsymbol{\mathcal{T}}$ can be described in a matrix format, the *s*catter *m*atrix layout, for an $N_i \times N_j$ matrix view of $\boldsymbol{\mathcal{T}}$, $\mathbf{T}$, very similarly to the general stride matrix layout,

$$\mathrm{LOC}_{SM}(\mathbf{T}_{i,j}, rscat_{\boldsymbol{\mathcal{T}}}, cscat_{\boldsymbol{\mathcal{T}}}) = rscat_{\boldsymbol{\mathcal{T}};i} + cscat_{\boldsymbol{\mathcal{T}};j}, \tag{2.2}$$

where $rscat_{\boldsymbol{\mathcal{T}}} \in \mathbb{N}^{N_i}$ and $cscat_{\boldsymbol{\mathcal{T}}} \in \mathbb{N}^{N_j}$. If we define the index bundle I_p of size p as the set of indices of $\boldsymbol{\mathcal{T}}$ that map to columns of $\mathbf{T}$ and the index bundle J_q of size $q = d_{\boldsymbol{\mathcal{T}}} - p$ as the set of indices that map to rows of $\mathbf{T}$, then by inspection of the general stride layout we can see that the scatter vector $rscat_{\boldsymbol{\mathcal{T}}}$ with respect to I_p is given by

$$rscat_{\boldsymbol{\mathcal{T}};i} = \sum_{k=0}^{p-1} i_k \cdot s_{\boldsymbol{\mathcal{T}};i_k},\ i = \sum_{k=0}^{p-1} i_k \cdot \prod_{l=0}^{k-1} N_{i_l},$$
$$\forall\,(i_0, \ldots, i_{p-1}) \in N_{i_0} \times \cdots \times N_{i_{p-1}};$$

and similarly for $cscat_{\boldsymbol{\mathcal{T}}}$ with respect to J_q.

The relative location of $\boldsymbol{\mathcal{C}}_{a,b,c}$ in Figure 2(a) or $\mathbf{C}_{i,j}$ in the matrix view of $\boldsymbol{\mathcal{C}}$ in Figure 2(b) is $rscat_{\boldsymbol{\mathcal{C}};i} + cscat_{\boldsymbol{\mathcal{C}};j}$ (e.g., $\mathrm{LOC}_{SM}(\boldsymbol{\mathcal{C}}_{2,3,1}) = \mathrm{LOC}_{SM}(\mathbf{C}_{6,3}) = rscat_{\boldsymbol{\mathcal{C}};6} + cscat_{\boldsymbol{\mathcal{C}};3} = 34 + 12$). Here, (1) $rscat_{\boldsymbol{\mathcal{C}};i} = a \cdot s_{\boldsymbol{\mathcal{C}};a} + c \cdot s_{\boldsymbol{\mathcal{C}};c} = a + 32c, i = a + c \cdot N_a = a + 4c \forall\,(a, c) \in N_a \times N_c$; (2) $cscat_{\boldsymbol{\mathcal{C}};j} = b \cdot s_{\boldsymbol{\mathcal{C}};b} = 4b, j = b \forall\,(b) \in N_b$. These scatter vectors are shown at the top and the left of the matrix view of $\boldsymbol{\mathcal{C}}$ in Figure 2(b).

The general definition of TCs gives a natural mapping from tensors to matrices through the index bundles I_m, J_n, and P_k. Thus, the bundle I_m defines $rscat_{\boldsymbol{\mathcal{A}}}$ and $rscat_{\boldsymbol{\mathcal{C}}}$, J_n defines $cscat_{\boldsymbol{\mathcal{B}}}$ and $cscat_{\boldsymbol{\mathcal{C}}}$, and P_k defines $cscat_{\boldsymbol{\mathcal{A}}}$ and $rscat_{\boldsymbol{\mathcal{B}}}$. If we define matrices $\mathbf{A}_{i,k}$, $\mathbf{B}_{k,j}$, and $\mathbf{C}_{i,j}$ and imbue them with scatter matrix layouts using the scatter vectors from the corresponding tensors, we can perform TC using the high-performance matrix multiplication algorithm introduced in section 2.1, without explicitly forming those matrices in extra working buffers and incurring the associated cost of data movement.

Since we are using the BLIS implementation of the GotoBLAS algorithm, we can leverage the fact that these matrices will be partitioned and packed to introduce

further optimizations. In the *microkernel* (Figure 1), the matrix $\mathbf{C}$ will be partitioned into $m_R \times n_R$ blocks and the matrices $\mathbf{A}$ and $\mathbf{B}$ will be partitioned into $m_R \times k_C$ and $k_C \times n_R$ slivers, respectively. If we further partition k_C into smaller increments of a new parameter k_R, on the order of m_R and n_R, then we will end up with only matrix blocks of very small size. As in [32], we can partition the scatter vectors into very small blocks of size m_R, n_R, and k_R as well and use optimized algorithms in the *packing* kernels (i.e., packing process in section 2.1) and *microkernel* when the scatter values for the current block are regularly spaced (i.e., strided). The regular strides for each $\{m,n,k\}_R$-sized block of $\{r,c\}scat_{\boldsymbol{\mathcal{T}}}$ (m_R for $rscat_{\boldsymbol{\mathcal{A}}}$ and $rscat_{\boldsymbol{\mathcal{C}}}$, n_R for $cscat_{\boldsymbol{\mathcal{B}}}$ and $cscat_{\boldsymbol{\mathcal{C}}}$, k_R for $cscat_{\boldsymbol{\mathcal{A}}}$ and $rscat_{\boldsymbol{\mathcal{B}}}$), or zero if no regular stride exists, are collected in a row/column block scatter vector $\{r,c\}bs_{\boldsymbol{\mathcal{T}}}$ of length $\lceil \frac{N_i}{\{m,n,k\}_R} \rceil$ and similarly for the other row/column scatter vectors. With these block scatter vectors, we can then utilize efficient `SIMD` vector load/store instructions for the stride-one index, or vector gather/scatter fetch instructions for the stride-n index, in a favorable memory access pattern.

In Figure 2(b), assuming $m_R = n_R = k_R = 4$, $rbs_{\boldsymbol{\mathcal{C}}} = (1,1)$, and $cbs_{\boldsymbol{\mathcal{C}}} = (4,4)$, since the regular strides for each four elements of $rscat_{\boldsymbol{\mathcal{C}}}$ and $cscat_{\boldsymbol{\mathcal{C}}}$ are 1 and 4, respectively.

3. Strassen for tensor contraction. The operations summarized in section 2.2 are all special cases of

$$\mathbf{M} = (\mathbf{X} + \delta\mathbf{Y})(\mathbf{V} + \epsilon\mathbf{W}); \quad \mathbf{D} \mathrel{+}= \gamma_0\mathbf{M}; \quad \mathbf{E} \mathrel{+}= \gamma_1\mathbf{M}; \tag{3.1}$$

for appropriately chosen $\gamma_0, \gamma_1, \delta, \epsilon \in \{-1, 0, 1\}$. Here, $\mathbf{X}$ and $\mathbf{Y}$ are submatrices of $\mathbf{A}$, $\mathbf{V}$ and $\mathbf{W}$ are submatrices of $\mathbf{B}$, and $\mathbf{D}$ and $\mathbf{E}$ are submatrices of $\mathbf{C}$. As in [20], this scheme can be extended to multiple levels of STRASSEN.

Instead of partitioning the tensor $\boldsymbol{\mathcal{A}}$ into subtensors $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ and so on for $\boldsymbol{\mathcal{B}}$ and $\boldsymbol{\mathcal{C}}$, we partition the matrix representations $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ (block scatter matrix view of $\boldsymbol{\mathcal{A}}$, $\boldsymbol{\mathcal{B}}$, $\boldsymbol{\mathcal{C}}$) as in the matrix implementation of STRASSEN. Figure 2 provides an example to illustrate the partition mechanism. Block scatter matrix layouts for these submatrices may be trivially obtained by partitioning the scatter and block scatter vectors of the entire matrices along the relevant dimensions. Once imbued with the appropriate layouts, these submatrices may then be used in the BLIS-based STRASSEN of [20] along with modifications to the packing kernels and microkernel as in [32].

In fusing these two methodologies, we need to further address the consideration of multiple block scatter vectors as required when packing and executing the microkernel. Methods for dealing with this issue are described in section 4.1. The advantage of using matrix partitions (which is enabled by the block scatter layout) instead of tensor partitions is primarily that only the product of the lengths of each index bundle, $\{N_{I_m}, N_{J_n}, N_{P_k}\}$, must be considered when partitioning, and not the lengths of individual tensor dimensions. For example, STRASSEN may be applied to any TC where *at least* one dimension in each bundle is even in our approach, whereas the *last* dimension (or rather, the dimension with the longest stride) should be even when using subtensors.[2] Additionally, when applying techniques such as zero-padding or dynamical peeling [21, 43] in order to address edge cases, the overhead is magnified for subtensor-based algorithms because the padding or peeling applies to only a single

[2] A dimension other than the last could also be chosen for partitioning, but the spatial locality of the partitioning would be destroyed.

tensor dimension; in our algorithm padding or peeling may be applied based on the length of the entire index bundle, which is necessarily longer and therefore incurs less overhead.

4. Implementations. We now detail the modifications to the block scatter matrix-based packing kernel and microkernel as described in [32] for STRASSEN.

4.1. Packing. When packing submatrices for STRASSEN using (3.1), multiple scatter and block scatter vectors must be considered. In our implementation, the block scatter vector entries for the corresponding blocks in both input submatrices (or all submatrices for L-level STRASSEN) are examined. If *all* entries are nonzero, then the constant stride is used in packing the current block.[3] Otherwise, the scatter vectors are used when packing the current block, even though one or more of the input submatrix blocks may in fact have a regular stride. In future work, we plan to exploit these cases for further performance improvements.

4.2. Microkernel. As in [20], we use assembly-coded microkernels that include the update to several submatrices of $\mathbf{C}$ from registers. In order to use this efficient update, *all* block scatter vector entries for the relevant submatrix blocks of $\mathbf{C}$ must be nonzero. Unlike in the packing kernel implementation, the case where only one or more of the submatrix blocks is regular stride would be more difficult to take advantage of, as the microkernel would have to be modified to flexibly omit or redirect individual submatrix updates.

4.3. Variations. We implement three variations of STRASSEN for TC on the theme illustrated in Figure 1 (right), extending [20].

- **Naive Strassen:** A classical implementation with temporary buffers. Submatrices of matrix representations of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ ($\mathbf{A}$ and $\mathbf{B}$) are explicitly copied and stored as regular submatrices. Intermediate submatrices $\mathbf{M}$ are explicitly stored and then accumulated into submatrices of matrix representation of $\boldsymbol{\mathcal{C}}$ ($\mathbf{C}$). We store the $\mathbf{M}$ submatrices as regular, densely stored matrices and handle their accumulation onto block scatter matrix layout submatrices of $\mathbf{C}$. Thus, the **naive Strassen** algorithm for TC is extremely similar to a TTGT-based STRASSEN algorithm (see section 7), except that the tensors are not required to be partitioned into regular subtensors.
- **AB Strassen:** The packing routines incorporate the summation of submatrices of matrix representations of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ with implicit tensor-to-matrix transformation into the packing buffers (see section 4.1), but explicit temporary buffers for matrices $\mathbf{M}$ are used.
- **ABC Strassen: AB Strassen**, but with a specialized microkernel (see section 4.2) that incorporates additions of $\mathbf{M}$ to multiple submatrices of matrix representation of $\boldsymbol{\mathcal{C}}$ with implicit matrix-to-tensor transformation. Thus, the **ABC Strassen** algorithm for TC requires no additional temporary buffers beyond the workspace already incorporated in conventional GEMM implementations.

5. Performance model. In [20], a performance model was proposed to predict the execution time T for variations of STRASSEN for matrices. In this section, we extend that performance model to estimate the execution time T of **ABC**, **AB**,

[3] Note that when nonzero, the block scatter vector entries for different submatrices will always be equal.

TABLE 1
Notation table for performance model.

τ_a	Time (in seconds) of one $a$rithmetic (floating point) operation.
τ_b	(Bandwidth) Amortized time (in seconds) of 8 bytes contiguous data movement from slow main memory to fast cache.
ρ_a	Penalty factor for $a$rithmetic operation effciency.
ρ_b	Penalty factor for $b$andwidth.
λ	Prefetching efficiency.
T	Total execution time (in seconds).
T_a	Time for $a$rithmetic operations (in seconds).
T_m	Time for $m$emory operations (in seconds).
$T_a^{\times}$	T_a for (sub)tensor contractions.
$T_a^{\boldsymbol{\mathcal{A}}+}$, $T_a^{\boldsymbol{\mathcal{B}}+}$, $T_a^{\boldsymbol{\mathcal{C}}+}$	T_a for extra (sub)tensor addtions/permutations.
$T_m^{\boldsymbol{\mathcal{A}}\times}$, $T_m^{\boldsymbol{\mathcal{B}}\times}$	T_m for reading (sub)tensors in packing routines (Figure 1).
$T_m^{\boldsymbol{\mathcal{C}}\times}$	T_m for reading *and* writing (sub)tensors in microkernel (Figure 1).
$T_m^{\boldsymbol{\mathcal{A}}+}$, $T_m^{\boldsymbol{\mathcal{B}}+}$, $T_m^{\boldsymbol{\mathcal{C}}+}$	T_m for reading *or* writing (sub)tensors, related to the temporary buffer as part of **naive Strassen** and **AB Strassen**.
W_a^X/W_m^X	Coefficient for the corresponding T_a^X/T_m^X.

and **naive** variations of L-level STRASSEN for TC and the high-performance non-STRASSEN TC routine we build on (see section 2.3; using TBLIS implementation [32, 31] introduced in section 6; denoted as TBLIS henceforth). Due to the high dimensionality of tensors and enormous types and combinations of permutations (transpositions) in TC, it is impractical to exhaustively search for every tensor shape and tensor problem size to find the best variation using empirical performance timings. Performance modeling helps us to better understand the memory footprint and computation of different STRASSEN implementations for TC and at least reduce the search space to pick the right implementation. In our model, besides input problem size, block sizes, and the hardware parameters such as the peak GFLOPS and bandwidth, T also depends on the shape of the tensors and the extra permutations in the packing routines and in the microkernel. In [19] we showed that a similar model is capable of predicting the best-performing fast matrix multiplication algorithms from a large set of candidates in most circumstances. The same predictive power should be applicable to TC as well, over a wide range of tensor shapes and sizes.

Assumptions. Similar to [20], we assume two layers of memory hierarchy: slow main memory and fast caches.[4] For write operations, the lazy write-back policy is enforced such that the time for writing into fast caches can be hidden. For read operations, the latency for accessing the slow main memory is counted, while the latency for accessing caches can be ignored.[5]

Notation. We summarize our notation in Table 1. The total execution time, T, can be decomposed into a sum of arithmetic time T_a and memory time T_m (② in Table 2).

Arithmetic operations. As shown in ③, T_a includes (sub)tensor contraction ($T_a^{\times}$) and (sub)tensor additions/permutations ($T_a^{\boldsymbol{\mathcal{A}}+}$, $T_a^{\boldsymbol{\mathcal{B}}+}$, $T_a^{\boldsymbol{\mathcal{C}}+}$). The corresponding

[4]The latency from multiple levels of cache for modern processors is hidden by hardware prefetching. Two layers of memory are good enough for modeling performance of regular applications such as GEMM.

[5]Either because it can be overlapped with computation or because it can be amortized over sufficient computations.

TABLE 2
Equations for computing the execution time T and effective *GFLOPS in our performance model.*

①	*Effective* GFLOPS $= 2 \cdot N_{I_m} \cdot N_{J_n} \cdot N_{P_k} / T \cdot 10^{-9}$
②	$T = T_a + T_m$
③	$T_a = W_a^{\times} \cdot T_a^{\times} + W_a^{\boldsymbol{\mathcal{A}}+} \cdot T_a^{\boldsymbol{\mathcal{A}}+} + W_a^{\boldsymbol{\mathcal{B}}+} \cdot T_a^{\boldsymbol{\mathcal{B}}+} + W_a^{\boldsymbol{\mathcal{C}}+} \cdot T_a^{\boldsymbol{\mathcal{C}}+}$
④	$T_m = W_m^{\boldsymbol{\mathcal{A}}\times} \cdot T_m^{\boldsymbol{\mathcal{A}}\times} + W_m^{\boldsymbol{\mathcal{B}}\times} \cdot T_m^{\boldsymbol{\mathcal{B}}\times} + W_m^{\boldsymbol{\mathcal{C}}\times} \cdot T_m^{\boldsymbol{\mathcal{C}}\times} + W_m^{\boldsymbol{\mathcal{A}}+} \cdot T_m^{\boldsymbol{\mathcal{A}}+} + W_m^{\boldsymbol{\mathcal{B}}+} \cdot T_m^{\boldsymbol{\mathcal{B}}+} + W_m^{\boldsymbol{\mathcal{C}}+} \cdot T_m^{\boldsymbol{\mathcal{C}}+}$
⑤	$\tau_a = 1/(\rho_a \cdot \text{Peak GFLOPS})$
⑥	$\tau_b = 8/(\rho_b \cdot \text{Bandwidth})$

TABLE 3
Various components of arithmetic and memory operations for TBLIS *TC and various implementations of* STRASSEN *TC. The time shown in the first column for* TBLIS *TC and L-level* STRASSEN *can be computed separately by multiplying the parameter in the τ column with the arithmetic/memory operation number in the corresponding entries. Here $N_{I_m} = \prod_{i \in I_m} N_i = N_{i_0} \cdot \cdots \cdot N_{i_{m-1}}$, $N_{J_n} = \prod_{j \in J_n} N_j = N_{j_0} \cdot \cdots \cdot N_{j_{n-1}}$, $N_{P_k} = \prod_{p \in P_k} N_p = N_{p_0} \cdot \cdots \cdot N_{p_{k-1}}$.*

	Type	τ	TBLIS	L-level
$T_a^{\times}$	-	τ_a	$2N_{I_m}N_{J_n}N_{P_k}$	$2\frac{N_{I_m}}{2^L}\frac{N_{J_n}}{2^L}\frac{N_{P_k}}{2^L}$
$T_a^{\boldsymbol{\mathcal{A}}+}$	-	τ_a	-	$2\frac{N_{I_m}}{2^L}\frac{N_{P_k}}{2^L}$
$T_a^{\boldsymbol{\mathcal{B}}+}$	-	τ_a	-	$2\frac{N_{P_k}}{2^L}\frac{N_{J_n}}{2^L}$
$T_a^{\boldsymbol{\mathcal{C}}+}$	-	τ_a	-	$2\frac{N_{I_m}}{2^L}\frac{N_{J_n}}{2^L}$
$T_m^{\boldsymbol{\mathcal{A}}\times}$	`r`	τ_b	$N_{I_m}N_{P_k}\lceil\frac{N_{J_n}}{n_c}\rceil$	$\frac{N_{I_m}}{2^L}\frac{N_{P_k}}{2^L}\lceil\frac{N_{J_n}/2^L}{n_c}\rceil$
$T_m^{\boldsymbol{\mathcal{B}}\times}$	`r`	τ_b	$N_{J_n}N_{P_k}$	$\frac{N_{J_n}}{2^L}\frac{N_{P_k}}{2^L}$
$T_m^{\boldsymbol{\mathcal{C}}\times}$	`r/w`	τ_b	$2\lambda N_{I_m}N_{J_n}\lceil\frac{N_{P_k}}{k_c}\rceil$	$2\lambda\frac{N_{I_m}}{2^L}\frac{N_{J_n}}{2^L}\lceil\frac{N_{P_k}/2^L}{k_c}\rceil$
$T_m^{\boldsymbol{\mathcal{A}}+}$	`r/w`	τ_b	$N_{I_m}N_{P_k}$	$\frac{N_{I_m}}{2^L}\frac{N_{P_k}}{2^L}$
$T_m^{\boldsymbol{\mathcal{B}}+}$	`r/w`	τ_b	$N_{J_n}N_{P_k}$	$\frac{N_{J_n}}{2^L}\frac{N_{P_k}}{2^L}$
$T_m^{\boldsymbol{\mathcal{C}}+}$	`r/w`	τ_b	$N_{I_m}N_{J_n}$	$\frac{N_{I_m}}{2^L}\frac{N_{J_n}}{2^L}$

coefficients W_a^X for TBLIS TC and L-level various STRASSEN TC are enumerated in Table 4. For example, one-level STRASSEN TC has coefficients $W_a^{\times} = 7$, $W_a^{\boldsymbol{\mathcal{A}}+} = 5$, $W_a^{\boldsymbol{\mathcal{B}}+} = 5$, and $W_a^{C+} = 12$, because it involves 7 submatrix multiplications, 5 additions with subtensors of $\boldsymbol{\mathcal{A}}$, 5 additions with subtensors of $\boldsymbol{\mathcal{B}}$, and 12 additions with subtensors of $\boldsymbol{\mathcal{C}}$. Note that T_a^X is calculated by multiplying the unit time τ_a with the arithmetic operation number in Table 3. We compute τ_a through ⑤. The penalty factor $\rho_a \in (0, 1]$ is introduced, due to the extra computations involved in $\{r, c\}scat_{\boldsymbol{\mathcal{T}}}$ and $\{r, c\}bs_{\boldsymbol{\mathcal{T}}}$, and the slow microkernel invocation when the corresponding entries in $rbs_{\boldsymbol{\mathcal{C}}}$ or $cbs_{\boldsymbol{\mathcal{C}}}$ are 0 (see section 4.2; nonregular stride access).

Memory operations. Based on the above assumptions, T_m can be broken down into three parts (④ in Table 2):

- updating the temporary buffer that are parts of **naive Strassen/AB Strassen** ($W_m^{\boldsymbol{\mathcal{T}}+} \cdot T_m^{\boldsymbol{\mathcal{T}}+}$);
- memory packing shown in Figure 1 ($W_m^{\boldsymbol{\mathcal{A}}\times} \cdot T_m^{\boldsymbol{\mathcal{A}}\times}$, $W_m^{\boldsymbol{\mathcal{B}}\times} \cdot T_m^{\boldsymbol{\mathcal{B}}\times}$);
- updating the submatrices of C shown in Figure 1 ($W_m^{\boldsymbol{\mathcal{C}}\times} \cdot T_m^{\boldsymbol{\mathcal{C}}\times}$).

The coefficients W_m^X are tabulated in Table 4. T_m^X is a function of block sizes $\{m_C, k_C, n_C\}$ in Table 3, and the bundle lengths $\{N_{I_m}/2^L, N_{J_n}/2^L, N_{P_k}/2^L\}$ because

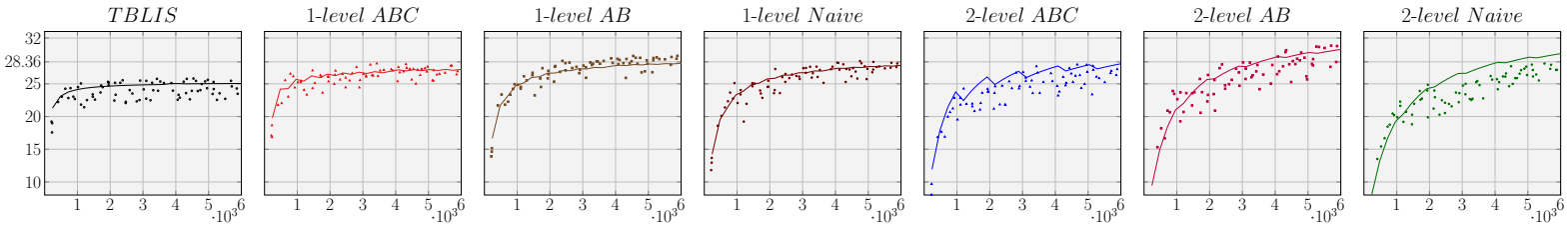
TABLE 4
Coefficient W_a^X/W_m^X mapping table for computing T_a^X/T_m^X in the performance model.

	TBLIS	1-level			2-level		
		ABC	AB	Naive	ABC	AB	Naive
$W_a^\times$	1	7	7	7	49	49	49
$W_a^{\mathcal{A}+}$	-	5	5	5	95	95	95
$W_a^{\mathcal{B}+}$	-	5	5	5	95	95	95
$W_a^{\mathcal{C}+}$	-	12	12	12	144	144	144
$W_m^{\mathcal{A}\times}$	1	12	12	7	194	194	49
$W_m^{\mathcal{B}\times}$	1	12	12	7	194	194	49
$W_m^{\mathcal{C}\times}$	1	12	7	7	144	49	49
$W_m^{\mathcal{A}+}$	-	-	-	19	-	-	293
$W_m^{\mathcal{B}+}$	-	-	-	19	-	-	293
$W_m^{\mathcal{C}+}$	-	-	36	36	-	432	432

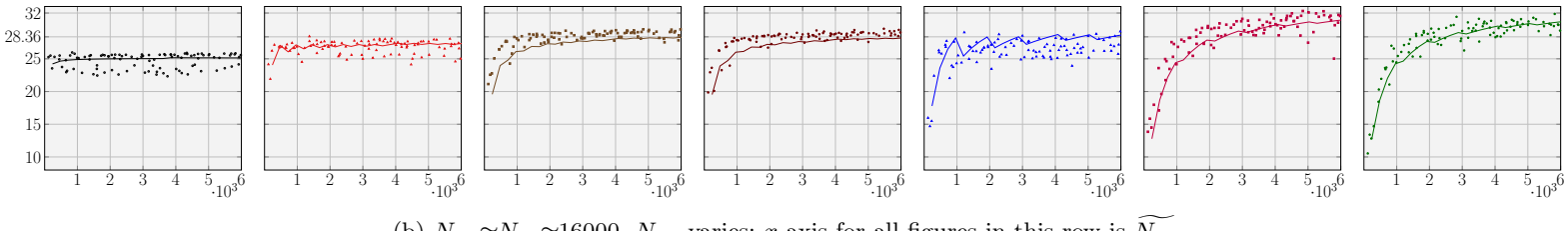
the memory operation can repeat multiple times according to which loop they reside in. Table 3 characterizes each memory operation term by its read/write type and the amount of memory in units of 64-bit double precision elements. In order to get T_m^X, the memory operation number needs to be multiplied by the bandwidth τ_b. We compute τ_b through ⑥. We penalize the effect of permutations without stride-one index accesss (see section 4.1; the corresponding entries in neither $rbs_{\mathcal{T}}$ or $cbs_{\mathcal{T}}$ are 1, i.e., using scatter/gather operation, or indirect memory addressing with (2.2)) by setting $\rho_b = 0.7$. A similar parameter is introduced in [40] for regular non-STRASSEN TC. Because of the software prefetching effects, $T_m^{\mathcal{C}\times} = 2\lambda \frac{N_{I_m}}{2^L} \frac{N_{J_n}}{2^L} \lceil \frac{N_{P_k}/2^L}{k_c} \rceil \tau_b$ has an extra parameter $\lambda \in (0.5, 1]$, which denotes the prefetching efficiency. $T_m^{\mathcal{C}\times}$ is a ceiling function proportional to N_{P_k}, since rank-k updates for accumulating submatrices of C recur $\lceil \frac{N_{P_k}/2^L}{k_c} \rceil$ times in the fourth loop (Figure 1).

Discussion. We can estimate the run time performance of various implementations, based on the performance model presented in Table 2. Here we define *effective* GFLOPS (① in Table 2) for TC as the metric to compare the performance of various STRASSEN TC and TBLIS TC. The theoretical peak GFLOPS and bandwidth information are given in section 6. In Figure 3, we demonstrate the modeled and actual performance for a wide range of synthetic tensor sizes and shapes: $N_{I_m} \approx N_{J_n} \approx N_{P_k}$; $N_{I_m} \approx N_{J_n} \approx 16000$, N_{P_k} varies; $N_{P_k} \approx 1024$, $N_{I_m} \approx N_{J_n}$ vary. How we generate synthetic data is detailed in section 6.1.1. In Table 5, we quantitatively show the model prediction accuracy.

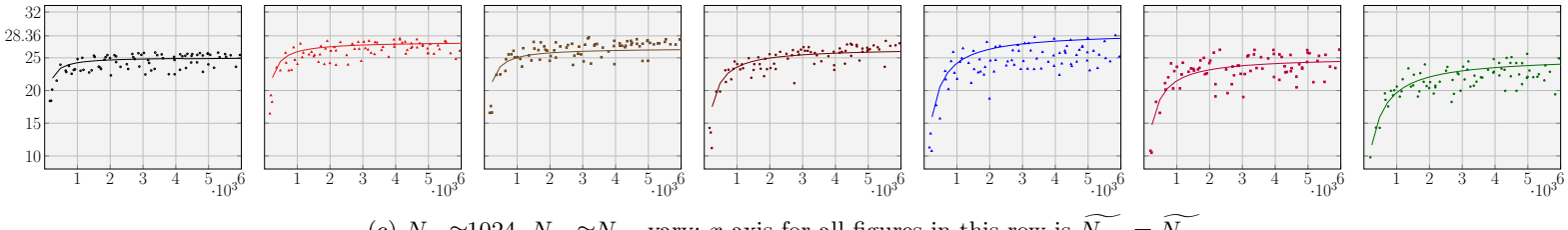
- The model can predict the relative performance for various implementations within 10% error bound.
- For $N_{I_m} \approx N_{J_n} \approx N_{P_k}$ (Figure 3(a)), the **ABC Strassen** implementations outperform TBLIS, when N_{I_m}, N_{J_n}, N_{P_k} are as small as $2k_C$, nearly 500, while **naive Strassen** cannot beat TBLIS until the problem size is larger than 2000.
- The "$N_{I_m} \approx N_{J_n} \approx 16000$, N_{P_k} varies" graphs (Figure 3(a)) show that when N_{P_k} is small, **ABC Strassen** performs best; when N_{P_k} is large, **AB Strassen** performs better. The coefficients W_m^X in Table 4 help to illustrate the reasons quantitatively. Two-level **AB Strassen** can achieve over 30% speedup comparing with TBLIS.



(a) $N_{I_m} \approx N_{J_n} \approx N_{P_k}$; x-axis for all figures in this row is $\widetilde{N_{I_m}} = \widetilde{N_{P_k}} = \widetilde{N_{J_n}}$.



(b) $N_{I_m} \approx N_{J_n} \approx 16000$, N_{P_k} varies; x-axis for all figures in this row is $\widetilde{N_{P_k}}$.



(c) $N_{P_k} \approx 1024$, $N_{I_m} \approx N_{J_n}$ vary; x-axis for all figures in this row is $\widetilde{N_{I_m}} = \widetilde{N_{J_n}}$.

FIG. 3. *Modeled performance (solid line) and actual performance (dots) of various implementations for synthetic data on single core. y-axis for all figures is* effective *GFLOPS* ($2 \cdot N_{I_m} \cdot N_{J_n} \cdot N_{P_k}/time$).

TABLE 5

Normalized root-mean-square error (NRMSE) between the actual and modeled performance for synthetic data on single core. NRMSE is defined as the root of mean square error normalized by the mean value of the measurements, which shows the model prediction accuracy.

TC shapes	NRMSE (%)						
	TBLIS	1-level			2-level		
		ABC	AB	Naive	ABC	AB	Naive
$N_{I_m} \approx N_{J_n} \approx N_{P_k}$	5.26	4.27	3.23	3.49	7.82	5.64	8.65
$N_{I_m} \approx N_{J_n} \approx 16000$, N_{P_k} varies	4.88	3.95	5.31	4.81	7.17	5.68	4.57
$N_{P_k} \approx 1024$, $N_{I_m} \approx N_{J_n}$ vary	4.55	4.64	5.39	5.23	9.08	7.65	7.26

- According to the model, when N_{P_k} is equal to appropriate multiple of k_C ($N_{P_k} = 2^L \cdot k_C$ for L-level), **ABC Strassen** achieves the best performance. We will leverage this observation in our distributed memory experiment.

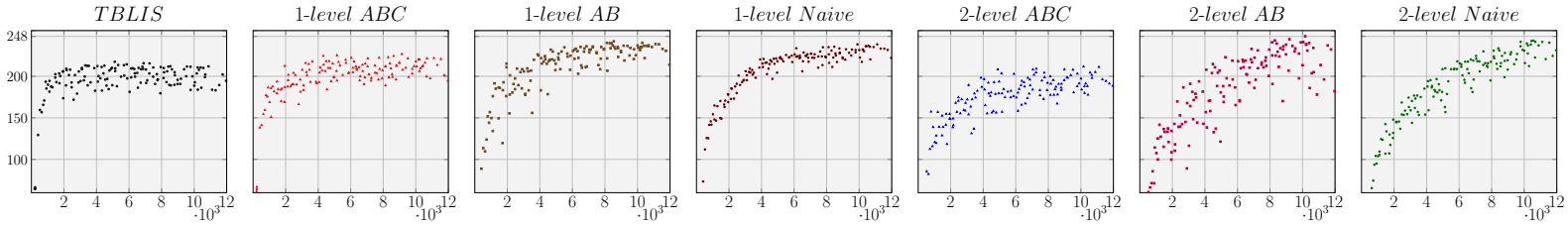
6. Experiments. We perform our experimental evaluations for synthetic data and real-world benchmarks on a single node and on a distributed memory architecture. The implementations are written in `C++`, utilizing `AVX` assembly, based on the open source TBLIS framework [31]. We compare against TBLIS's TC routine (marked as TBLIS) as well as the TTT routine from the MATLAB Tensor Toolbox [2] (linked with Intel MKL [22], marked as TTT) for single node and the TC routine from the Cyclops Tensor Framework [38] (also linked with Intel MKL, marked with CTF) for distributed memory.

We measure the CPU performance results on the Maverick system at the Texas Advanced Computing Center (TACC). Each node of that system consists of a dual-socket (10 cores/socket) Intel Xeon E5-2680 v2 (Ivy Bridge) processor with 256 GB memory (peak bandwidth: 59.7 GB/s with four channels) and a three-level cache (32 KB L1 data; 256 KB L2; 25.6 MB L3). The stable CPU clockrate is 3.54 GHz when a single core is utilized (28.32 GFLOPS peak, marked in the graphs) and 3.10 GHz when all 10 cores are in use (24.8 GFLOPS/core peak). We disable hyper-threading explicitly and set thread affinity with `KMP_AFFINITY=compact` which also ensures the computation and the memory allocation all reside on the same socket.

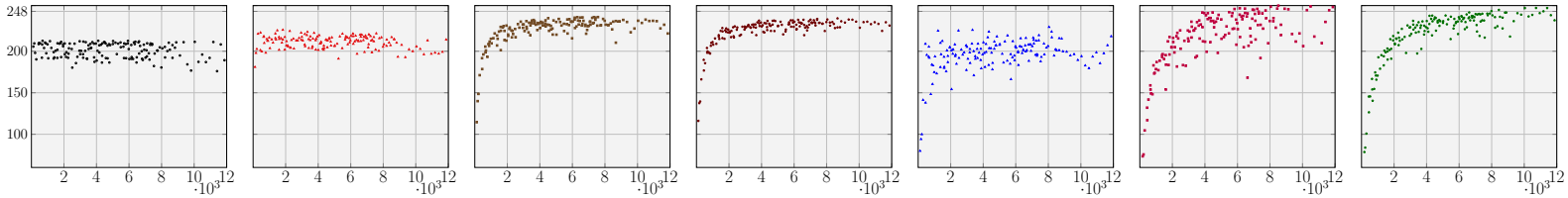
The cache blocking parameters, $m_C = 96$, $n_C = 4096$, $k_C = 256$, and the register block sizes, $m_R = 8$, $n_R = 4$, are consistent with parameters used for the standard BLIS DGEMM implementation for this architecture. We use the default value of $k_R = 4$ as defined in TBLIS. This makes the size of the packing buffer $\widetilde{\mathbf{A}}_i$ 192 KB and $\widetilde{\mathbf{B}}_p$ 8192 KB, which then fits the L2 cache and the L3 cache, respectively. Parallelization is implemented mirroring that described in [37], but with the number of threads assigned to each of the loops in Figure 1 automatically determined by the TBLIS framework.

6.1. Single node experiments.

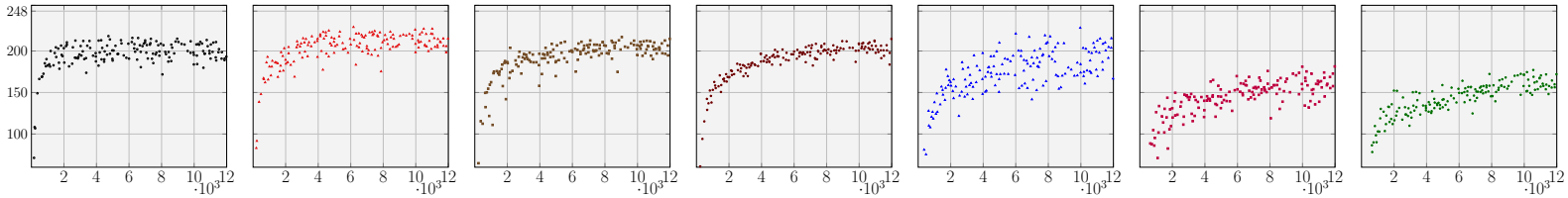
6.1.1. Synthetic tensor contractions. To evaluate the overall performance of various STRASSEN TC comparing against TBLIS TC for different tensor problem sizes, shapes, and permutations, we randomly generate TC test cases with 2-D to 6-D randomly permuted tensors as operands and test all these implementations for each synthetic test case, as shown in Figures 3 and 4. We choose step size 256 to sample uniformly $\{N_{I_m}, N_{J_n}, N_{P_k}\}$ for various tensor bundle lengths: *square*: $N_{I_m} \approx N_{J_n} \approx N_{P_k}$; *rank-$N_{P_k}$*: $N_{I_m} \approx N_{J_n} \approx 16000$, N_{P_k} varies; *fixed-N_{P_k}*: $N_{P_k} \approx 1024$, $N_{I_m} \approx N_{J_n}$ vary. For each bundle length $\{N_{I_m}, N_{J_n}, N_{P_k}\}$, we randomly generate three $\{I_m, J_n, P_k\}$ 1-D, 2-D, or 3-D bundles, such that the product of each index length is close to $\{N_{I_m}, N_{J_n}, N_{P_k}\}$. The order of $\{I_m, J_n, P_k\}$ is then randomly permuted.



(a) $N_{I_m} \approx N_{J_n} \approx N_{P_k}$; x-axis for all figures in this row is $\widetilde{N_{I_m}} = \widetilde{N_{P_k}} = \widetilde{N_{J_n}}$.



(b) $N_{I_m} \approx N_{J_n} \approx 16000$, N_{P_k} varies; x-axis for all figures in this row is $\widetilde{N_{P_k}}$.



(c) $N_{P_k} \approx 1024$, $N_{I_m} \approx N_{J_n}$ vary; x-axis for all figures in this row is $\widetilde{N_{I_m}} = \widetilde{N_{J_n}}$.

FIG. 4. *Actual performance (dots) of various implementations for synthetic data on one socket. y-axis for all figures is* effective *GFLOPS* ($2 \cdot N_{I_m} \cdot N_{J_n} \cdot N_{P_k}/time$).

The generated bundle lengths may not exactly match the original sampled bundle lengths. When we plot the actual performance of these synthetic test cases, we set effective bundle lengths $\widetilde{N_{I_m}} = \widetilde{N_{J_n}} = \widetilde{N_{P_k}} = (N_{I_m} \cdot N_{J_n} \cdot N_{P_k})^{1/3}$ for the *square* bundle lengths; $\widetilde{N_{P_k}} = N_{I_m} \cdot N_{J_n} \cdot N_{P_k}/(16000 \cdot 16000)$ for *rank-*N_{P_k} bundle lengths; and $\widetilde{N_{I_m}} = \widetilde{N_{J_n}} = (N_{I_m} \cdot N_{J_n} \cdot N_{P_k}/1024)^{1/2}$ for *fixed-*N_{P_k} bundle lengths.

For the *square* and *rank-*N_{P_k} tensor shapes on one core, TBLIS is rapidly outpaced by **ABC Strassen**, with a crossover point of about $500 \approx 2 \cdot k_C$. **ABC Strassen** is then shortly overtaken by **AB Strassen** and then by two-level **AB Strassen**. As predicted by the performance model, the **AB Strassen** implementation is best for very large problem sizes due to repeated updates to **C** in the **ABC Strassen** algorithm. The **naive Strassen** implementations are never the best in these experiments, although they may become more efficient than **AB Strassen** for extremely large, square problems. These trends are repeated in the 10-core experiments, although the crossover points are moved to larger tensor sizes.

For the *fixed-*N_{P_k} shapes, total performance is lower for **AB Strassen** and **naive Strassen** with scalability for the algorithms being especially impacted by the relatively smaller N_{I_m} and N_{J_n} sizes. For these shapes **ABC Strassen** is always the fastest method above the crossover point with standard TBLIS.

The actual performance data matches the predicted performance very well, with some variation due to the randomization of the tensor lengths and permutations. Using these performance models, it may be possible to analytically decide on which algorithm to apply for a given TC to achieve the highest performance, allowing an automated and seamless inclusion of STRASSEN into a TBLIS-like tensor framework.

6.1.2. Real-world benchmark. In Figure 5, we measure the performance of various implementations for a subset of TC from the Tensor Contraction Benchmark [39] on a single core and one socket. We present representative use cases where N_{P_k} is nearly equal to or larger than $2k_C$ (512), for which STRASSEN can show performance benefits, as illustrated in section 5. The right three test cases represent various regularly blocked TCs from coupled cluster with single and double excitations (CCSD) [36, 17, 35], a workhorse quantum chemistry computational method. The fourth case from the right illustrates the performance of TBLIS and STRASSEN TC for a pure matrix case. Comparing this case and the CCSD contractions highlights some of the performance issues that exist in the current implementation of the packing and matrix-to-block scatter matrix copy kernels (see section 4.1 for details). On one core, all STRASSEN implementations improve on TBLIS for these right four cases, and in parallel one-level STRASSEN implementations give a speedup as well, exceeding TTT performance especially in the case of **AB Strassen**. The gap between TBLIS and TTT for these contractions is due to TTT's use of Intel's MKL library, which is more highly optimized than the BLIS/TBLIS framework.

The left two benchmarks are again quantum chemistry applications using 3-D tensors that arise in density-fitting (DF) calculations [47, 12]. These contractions are also structurally equivalent to certain contractions from the coupled cluster with perturbative triples (CCSD(T)) method [34], where the *occupied* (see section 6.2) indices have been sliced. These cases show the improvement of TBLIS over TTT as noted in [32] but do not show a speedup from STRASSEN except for one-level **ABC Strassen** on one core. Our STRASSEN implementation performs the submatrix multiplications sequentially, with only parallelization of each submatrix multiplication step. A more comprehensive parallelization scheme, for example, using task-based parallelism [4], may show better performance. Additionally, since the DF/CCSD(T)

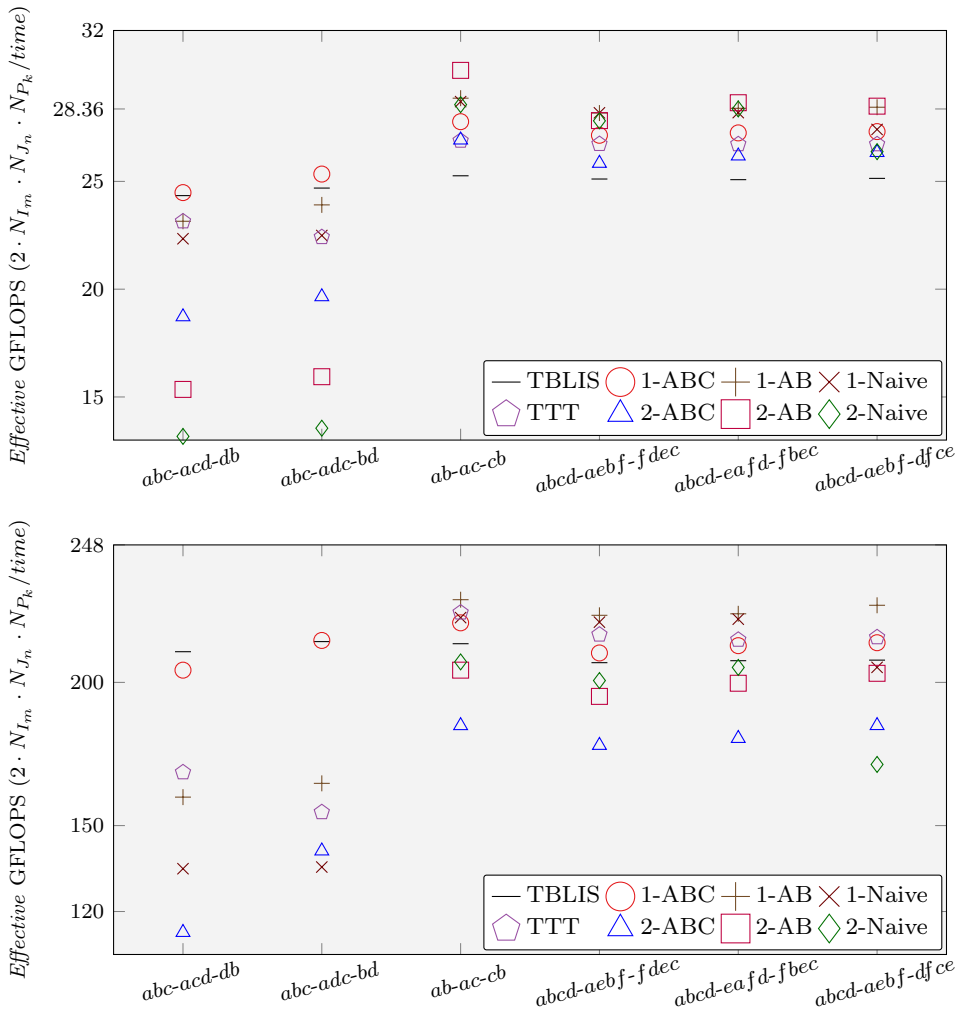


FIG. 5. *Performance for representative user cases of benchmark from* [40]. *TC is identified by the index string, with the tensor index bundle of each tensor in the order* $\mathcal{C}$-$\mathcal{A}$-$\mathcal{B}$, *e.g.,* $\mathcal{C}_{abcd} += \mathcal{A}_{aebf}\mathcal{B}_{dfce}$ *is denoted as abcd-aebf-dfce. Top: performance on single core. Bottom: performance on one socket.*

contractions are highly "nonsquare," an alternate fast matrix multiplication algorithm [4, 19] may perform better.

6.1.3. Shape-dependence experiments. The performance of the "particle-particle ladder" contraction from CCSD, $\mathcal{Z}_{abij} += \mathcal{W}_{abef} \cdot \mathcal{T}_{efij}$, is reported for a range of tensor shapes in Figure 6. In these experiments, the length of the *virtual* dimensions $\{a, b, e, f\}$ is varied with respect to the length of the *occupied* dimensions $\{i, j\}$ such that the total number of FLOPs is roughly similarly to a 16000×16000 matrix multiplication, and the ratio $N_a : N_i$ is used as a proxy for tensor shape. A ratio of 1:1 would reflect an extremely poor quality of basis set for the overall calculation but is common when the calculation employs *regular blocking*. The other end of the scale, with a ratio of $\sim 5 : 1$, would then correspond to *uneven blocking*. This type

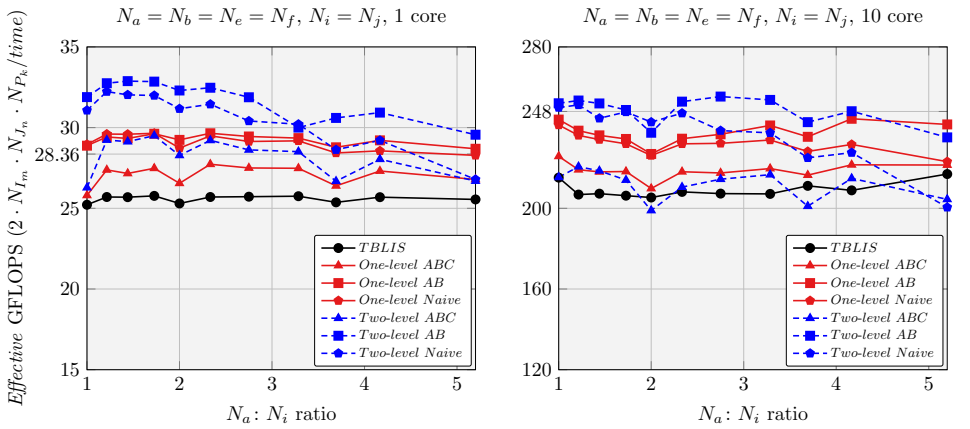


FIG. 6. *Performance for the contraction* $\boldsymbol{\mathcal{Z}}_{abij} \mathrel{+}= \boldsymbol{\mathcal{W}}_{abef} \cdot \boldsymbol{\mathcal{T}}_{efij}$ *with varying* N_a: N_i *ratio. Left: performance on single core. Right: performance on one socket.*

of blocking allows for better load balancing and lower overhead when N_a and N_i are very unequal in the overall calculation.

The performance of TBLIS and all of the one-level STRASSEN algorithms shows essentially no performance degradation across the entire range tested. The two-level STRASSEN algorithms show some performance degradation at larger ratios but still show improvement over TBLIS. Eventually, all STRASSEN algorithms will cross over and perform worse than TBLIS, as evidenced by the left two contractions in Figure 5 (these correspond to a ratio of about 22). However, the good performance of STRASSEN out to reasonably large ratios shows that it could be beneficial in both regular blocking and uneven blocking scenarios.

6.2. Distributed memory experiments. We demonstrate how to use the STRASSEN TC implementations to accelerate a distributed memory implementation of 4-D TC that exemplifies the two-particle "ring" terms from CCSD. In our tests we set the length of *virtual* indices $\{a, b, e\}$ to $10\times$ that of *occupied* indices $\{i, j, m\}$, which is a ratio commonly encountered in quantum chemistry calculations using popular basis sets such as 6-311++G** [25] and cc-pVTZ [13]. The problem sizes tested here correspond to calculations on systems with 80, 112, 160, 192, and 224 electrons (i.e., $N_i = N_j = N_m \in \{40, 56, 80, 96, 112\}$ and $N_a = N_b = N_e = 10 \cdot N_i$). We use the contraction $\boldsymbol{\mathcal{Z}}_{abij} := \boldsymbol{\mathcal{W}}_{bmej}\boldsymbol{\mathcal{T}}_{aeim}$ as a demonstration example to show the performance benefit.

We implement a SUMMA-like [44] algorithm for 4-D TC with MPI. Initially the tensors $\boldsymbol{\mathcal{W}}$, $\boldsymbol{\mathcal{T}}$, and $\boldsymbol{\mathcal{Z}}$ are distributed to a $P \times P$ mesh of MPI processes using a 2-D block distribution over the a, b, and e dimensions, with the i, j, and m dimensions stored locally (i.e., not distributed). After slicing $\boldsymbol{\mathcal{W}}$ and $\boldsymbol{\mathcal{T}}$ along the e dimension, the contraction is broken down into a sequence of K contractions of tensor slice pairs,

$$\boldsymbol{\mathcal{Z}} := \left(\boldsymbol{\mathcal{W}}_{e;0} \mid \cdots \mid \boldsymbol{\mathcal{W}}_{e;K-1} \right) \left(\begin{array}{c} \boldsymbol{\mathcal{T}}_{e;0} \\ \hline \vdots \\ \hline \boldsymbol{\mathcal{T}}_{e;K-1} \end{array} \right)$$

such that the e index length for each tensor slice pair $\{\boldsymbol{\mathcal{W}}_{e;p}, \boldsymbol{\mathcal{T}}_{e;p}\}$ is $N'_e = N_e/K$. For each tensor slice pair, $0 \le p < K$, $\boldsymbol{\mathcal{W}}_{e;p}$ is broadcast within rows of the mesh,

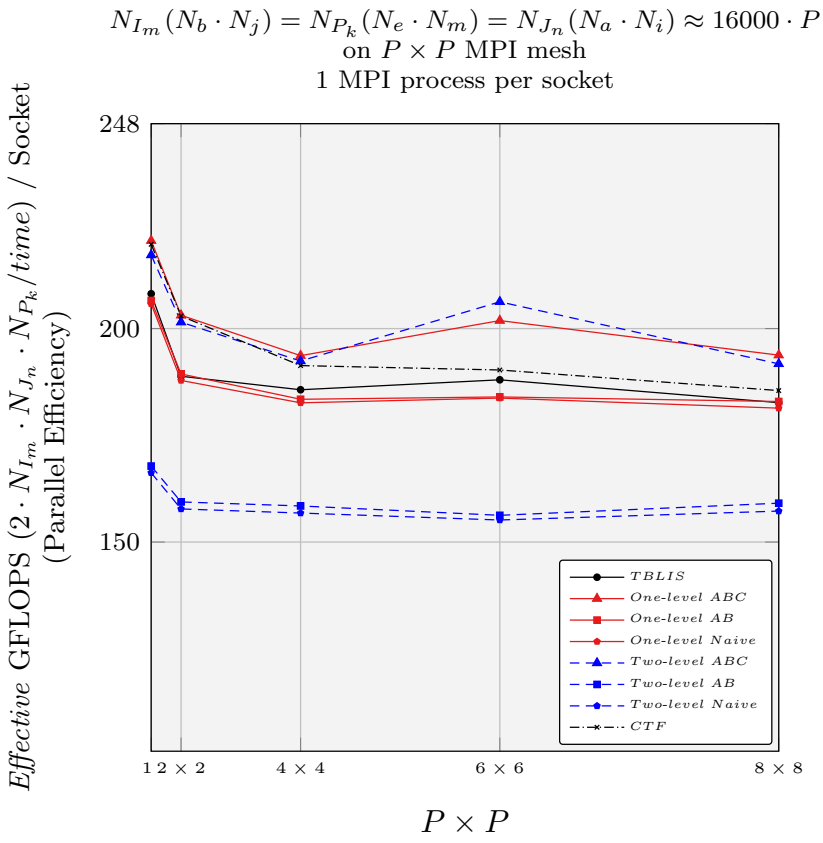


FIG. 7. *Weak scalability performance result of the various implementations for a 4-D CCSD application on distributed memory:* $\boldsymbol{\mathcal{Z}}_{abij} := \boldsymbol{\mathcal{W}}_{bmej}\boldsymbol{\mathcal{T}}_{aeim}$. *CTF: the performance of the Cyclops Tensor Framework* [38] *(linked with Intel MKL).*

and $\boldsymbol{\mathcal{T}}_{e;p}$ is broadcast within columns of the mesh. Then a local TC for the received tensor slice pair is performed to update the local block. Here TBLIS TC and various STRASSEN TC are used as a drop-in replacement for this local TC.

We perform the distributed memory experiment on the same machine as the single node experiment. The dual-socket processor has 10 cores on each socket. We run one MPI process for each socket and leverage all 10 cores in a socket with thread parallelism for all implementations. Figure 7 reports the weak scalability performance result on up to 640 cores (32 nodes, 64 sockets).

In our experiments on $P \times P$ mesh of sockets (MPI processes), the lengths of virtual indices are set to equal $N_a = N_b = N_e \approx 400\sqrt{P}$ and the lengths of occupied indices are set to equal $N_i = N_j = N_m \approx 40\sqrt{P}$, which make $N_{I_m} = N_{J_n} = N_{P_k} \approx 16000 \cdot P$. This guarantees the local memory buffer allocated to $\boldsymbol{\mathcal{Z}}$, $\boldsymbol{\mathcal{W}}$, $\boldsymbol{\mathcal{T}}$ is constant. Our experiments verify that the above SUMMA-like algorithm is weakly scalable on this constant local memory setup, regardless of which local TC implementation we use. The local e index length N'_e is chosen close to $N'_e = 1024/N_m$ (i.e., $N'_e \in \{25, 18, 12, 10, 9\}$) such that the local TC computations are performed with $N_{P_k} = N'_e \cdot N_m \approx 4 \cdot k_C$. The tensor slice pairs in the local TC computations matches the shape when **ABC Strassen** achieves the best performance. Therefore, the one-level and two-level **ABC Strassen** implementations outperform all other implementations.

We also tested the Cyclops Tensor Framework [38], which also uses a SUMMA or nested SUMMA algorithm but with possibly different block sizes and tensor distributions, as well as using the TTGT algorithm for local TC. We show it here as a reference for state-of-the-art performance.

7. Related work. To the best of our knowledge, this work represents the first implementation of Strassen's algorithm for TC. In the context of STRASSEN for matrices, there have been a variety of practical implementations [11, 21, 7, 4], including the closely related implementation of STRASSEN using the BLIS framework [20] which we extend.

For TC, recent work on high-performance TC [32, 40] serves as the motivation and basis for our present work, while other research has focused on algorithms using tensor slicing [9, 33, 26, 30] or on improving the efficiency of the so-called TTGT algorithm for TC [16, 15, 29, 41], where input tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are *T*ransposed (permuted) and then used in a standard GEMM algorithm, with the output then being *T*ransposed and accumulated onto the tensor $\boldsymbol{\mathcal{C}}$. TTGT could be used to construct a STRASSEN algorithm for TC by transposing subtensors into submatrices and vice versa and using a matrix implementation of STRASSEN instead of GEMM. However, we showed that this algorithm is essentially the same as our **naive Strassen** algorithm (see section 4.3), which is often less efficient than the other algorithms that we have implemented.

The GETT algorithm [40] is a high-performance TC implementation similar in many ways to the BLIS-based implementation by Matthews [32]. As in our current implementation, formation of linear combinations of input subtensors of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ and output to multiple subtensors of $\boldsymbol{\mathcal{C}}$ could be fused with the internal tensor transposition and microkernel steps of GETT. However, the implementation would be restricted to regular subtensors rather than more general submatrices (see section 3), which could have possible negative performance implications (e.g., false sharing).

8. Conclusions. We have presented what we believe to be the first paper to demonstrate how to leverage Strassen's algorithm for TC and have shown practical performance speedup on single core, multicore, and distributed memory implementations. Using a block scatter matrix layout enables us to partition the matrix view of the tensor, instead of the tensor itself, with automatic (implicit) tensor-to-matrix transformation, and the flexibility to facilitate Strassen's 2-D matrix partition to multidimensional tensor spaces. Fusing the matrix summation that must be performed for STRASSEN and the transposition that must be conducted for TC with the packing and microkernel operations inside high-performance implementation of GEMM avoids extra workspace requirements and reduces the cost of additional memory movement. We provided a performance model which can predict the speedup of the resulting family of algorithms for different tensor shapes, sizes, and permutations, with enough accuracy to reduce the search space to pick the right implementation. We evaluated our families of implementations for various tensor sizes and shapes on synthetic and real-world datasets, both observing significant speedups comparing to the baseline (TBLIS) and naive implementations (**naive Strassen**), particularly for smaller problem sizes ($N_{I_m}, N_{J_n}, N_{P_k} \approx 2k_C, 4k_C$), and irregular shape ($N_{P_k}$ is much smaller comparing to N_{I_m}, N_{J_n}). Together, this work demonstrates Strassen's algorithm can be applied for TC with practical performance benefit.

There are several avenues for future work:

- Higher-level tensor decomposition algorithms [24], such as Tucker decomposition, involve heavy use of TC. The impact of our performance improvements

with Strassen's algorithm for those algorithms is an interesting question. It may be possible to leverage our performance model to determine the best implementation for the tensor shape these algorithms require.

- So far, we target dense TC, which has numerous applications. However, the structure of the tensor operands may be symmetric [5] or sparse [1], which yields a number of new challenges, like more efficient storage or layout format. How to explore those structure patterns and combine with Strassen's algorithm can be investigated.
- More levels of Strassen's algorithm may lose precision due to numerical instability issues. It may be possible to combine with the techniques proposed in Extended and Mixed Precision BLAS [27] to get higher speedup and maintain precision.
- A number of recent papers explore practical implementations of Strassen-like fast matrix multiplications [4, 19] and other variations of Strassen's algorithm [23]. How to extend fast matrix multiplication with different partition block sizes for TC is an open question.

Source code availability. A number of implementations of the discussed Strassen's algorithm for tensor contraction are available under 3-clause (modified) BSD license from https://github.com/flame/tblis-strassen.

Acknowledgments. Access to the Maverick supercomputers administered by TACC is gratefully acknowledged. We thank Martin Schatz, for his help with distributed memory implementations, and the rest of the SHPC team (http://shpc.ices.utexas.edu) for their support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] B. W. Bader and T. G. Kolda, *Efficient MATLAB computations with sparse and factored tensors*, SIAM J. Sci. Comput., 30 (2007), pp. 205–231.

[2] B. W. Bader, T. G. Kolda, et al., *Matlab Tensor Toolbox Version* 2.6, http://www.sandia.gov/~tgkolda/TensorToolbox/.

[3] G. Ballard, A. R. Benson, A. Druinsky, B. Lipshitz, and O. Schwartz, *Improving the numerical stability of fast matrix multiplication*, SIAM J. Matrix Anal. Appl., 37 (2016), pp. 1382–1418.

[4] A. R. Benson and G. Ballard, *A framework for practical parallel fast matrix multiplication*, in Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2015, pp. 42–53.

[5] P. Comon, G. Golub, L.-H. Lim, and B. Mourrain, *Symmetric tensors and symmetric tensor rank*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 1254–1279.

[6] T. Crawford and H. Schaefer III, *An introduction to coupled cluster theory for computational chemists*, Rev. Comp. Chem., 14 (2000), pp. 33–136.

[7] P. D'Alberto, M. Bodrato, and A. Nicolau, *Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation*, ACM Trans. Math. Software, 38 (2011), pp. 2:1–2:30.

[8] J. Demmel, I. Dumitriu, O. Holtz, and R. Kleinberg, *Fast matrix multiplication is stable*, Numer. Math., 106 (2007), pp. 199–224.

[9] E. Di Napoli, D. Fabregat-Traver, G. Quintana-Orti, and P. Bientinesi, *Towards an efficient use of the BLAS library for multilinear tensor contractions*, Appl. Math. Comput., 235 (2014), pp. 454–468.

[10] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, *A set of level* 3 *basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.

[11] C. Douglas, M. Heroux, G. Slishman, and R. Smith, *GEMMW—a portable level* 3 *BLAS Winograd variant of Strassen's matrix-matrix multiplication algorithm*, J. Comput. Phys., 1994, pp. 1–10.

[12] B. I. Dunlap, *Robust and variational fitting*, Phys. Chem. Chem. Phys., 2 (2000), pp. 2113–2116.

[13] T. H. Dunning, Jr., *Gaussian basis sets for use in correlated molecular calculations.* I. *The atoms boron through neon and hydrogen*, J. Chem. Phys., 90 (1989), pp. 1007–1023.

[14] K. Goto and R. A. van de Geijn, *Anatomy of a high-performance matrix multiplication*, ACM Trans. Math. Software, 34 (2008), 12.

[15] M. Hanrath and A. Engels-Putzka, *An efficient matrix-matrix multiplication based anti-symmetric tensor contraction engine for general order coupled cluster*, J. Chem. Phys., 133 (2010), 064108.

[16] A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan, *Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry*, J. Phys. Chem. A, 113 (2009), pp. 12715–12723.

[17] T. Helgaker, P. Jorgensen, and J. Olsen, *Molecular Electronic-Structure Theory*, Wiley, New York, 2013.

[18] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, 2002.

[19] J. Huang, L. Rice, D. A. Matthews, and R. A. van de Geijn, *Generating families of practical fast matrix multiplication algorithms*, in Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium, 2017.

[20] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, *Strassen's algorithm reloaded*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, 2016, pp. 59:1–59:12.

[21] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson, *Implementation of Strassen's algorithm for matrix multiplication*, in Proceedings of the ACM/IEEE Conference on Supercomputing, Washington, DC, 1996.

[22] *Intel MKL*, https://software.intel.com/en-us/intel-mkl (2017).

[23] E. Karstadt and O. Schwartz, *Matrix multiplication, a little faster*, in Proceedings of the 29th Annual ACM Symposium on Parallelism in Algorithms and Architectures, New York, 2017.

[24] T. Kolda and B. Bader, *Tensor Decompositions and Applications*, SIAM Rev., 51 (2009), pp. 455–500.

[25] R. Krishnan, J. S. Binkley, R. Seeger, and J. A. Pople, *Selfconsistent molecular orbital methods.* XX. *A basis set for correlated wave functions*, J. Chem. Phys., 72 (1980), pp. 650–654.

[26] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc, *An input-adaptive and in-place approach to dense tensor-times-matrix multiply*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New York, ACM, 2015, pp. 76:1–76:12.

[27] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo, *Design, implementation and testing of extended and mixed precision BLAS*, ACM Trans. Math. Software, 28 (2002), pp. 152–205.

[28] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, *Analytical modeling is enough for high-performance BLIS*, ACM Trans. Math. Software, 43 (2016), pp. 12:1–12:18.

[29] D. I. Lyakh, *An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU*, Comput. Phys. Commun., 189 (2015), pp. 84–91.

[30] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal, *Optimizing tensor contraction expressions for hybrid CPU-GPU execution*, Cluster Comput., 16 (2011), pp. 131–155.

[31] D. A. Matthews, *Tensor-Based Library Instantiation Software*, https://github.com/devinamatthews/tblis (2016).

[32] D. A. Matthews, *High-performance tensor contraction without transposition*, SIAM J. Sci. Comput., 40 (2018), pp. C1–C24.

[33] E. Peise, D. Fabregat-Traver, and P. Bientinesi, *On the performance prediction of BLAS-based tensor contractions*, in High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation, S. A. Jarvis, S. A. Wright, and S. D. Hammond, eds., Lecture Notes in Comput. Sci. 8966, Springer, New York, 2014, pp. 193–212.

[34] K. Raghavachari, G. W. Trucks, J. A. Pople, and M. Head-Gordon, *A fifth-order perturbation comparison of electron correlation theories*, Chem. Phys. Lett., 157 (1989), pp. 479–483.

[35] G. E. Scuseria, A. C. Scheiner, T. J. Lee, J. E. Rice, and H. F. Schaefer, *The closed-shell coupled cluster single and double excitation (CCSD) model for the description of electron correlation. A comparison with configuration interaction (CISD) results*, J. Chem. Phys., 85 (1987), p. 2881–2890.

[36] I. Shavitt and R. J. Bartlett, *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*, Cambridge University Press, Cambridge, UK, 2009.

[37] T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, *Anatomy of high-performance many-threaded matrix multiplication*, in Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium, 2014.

[38] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, *A massively parallel tensor contraction framework for coupled-cluster computations*, J. Parallel Distributed Comput., 74 (2014), pp. 3176–3190.

[39] P. Springer and P. Bientinesi, *Tensor Contraction Benchmark V*0.1, https://github.com/hpac/tccg/tree/master/benchmark (2016).

[40] P. Springer and P. Bientinesi, *Design of a high-performance gemm-like tensor-tensor multiplication*, ACM Trans. Math. Software, 44 (2018), pp. 28:1–28:29.

[41] P. Springer, J. R. Hammond, and P. Bientinesi, *TTC: A High-Performance Compiler for Tensor Transpositions*, CoRR, arXiv:1603.02297, 2016.

[42] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[43] M. Thottethodi, S. Chatterjee, and A. R. Lebeck, *Tuning Strassen's matrix multiplication for memory efficiency*, in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, 1998, pp. 1–14.

[44] R. van de Geijn and J. Watts, *SUMMA: Scalable universal matrix multiplication algorithm*, Concurrency Practice Experience, 9 (1997), pp. 255–274.

[45] F. G. Van Zee, T. Smith, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. Gunnels, T. M. Low, B. Marker, L. Killough, and R. A. van de Geijn, *The BLIS framework: Experiments in portability*, ACM Trans. Math. Software, 42 (2016), 12.

[46] F. G. Van Zee and R. A. van de Geijn, *BLIS: A framework for rapidly instantiating BLAS functionality*, ACM Trans. Math. Software, 41 (2015), 14.

[47] J. L. Whitten, *Coulombic potential energy integrals and approximations*, J. Chem. Phys., 58 (1973), pp. 4496–4501.