FaSTCC: Fast Sparse Tensor Contractions on CPUs

Saurabh Raje
University of Utah
Salt Lake City, Utah, USA
saurabh.raje@utah.edu

Hunter McCoy
Northeastern University
Boston, Massachusetts, USA
mccoy.hu@northeastern.edu

Atanas Rountev
Ohio State University
Columbus, Ohio, USA
rountev@cse.ohio-state.edu

Prashant Pandey
Northeastern University
Boston, Massachusetts, USA
p.pandey@northeastern.edu

P. Sadayappan
University of Utah
Salt Lake City, Utah, USA
saday@cs.utah.edu

ABSTRACT

Sparse tensor contractions are a core computational primitive in scientific computing and machine learning. Effective optimization of such contractions through loop permutation/tiling remains an open challenge. Our work perform the first comprehensive comparative analysis of data access costs and memory requirements for loop permutations for sparse tensor contractions. Based on these insights, we develop FaSTCC, a novel hashing-based parallel implementation of sparse tensor contractions. FaSTCC introduces a new 2D tiled contraction-index-outer scheme and a corresponding tile-aware design. Using probabilistic modeling, our approach automatically chooses between dense and sparse output tile accumulators and selects suitable tile size. We evaluate FaSTCC across two CPU platforms and a range of real-world workloads, demonstrating significant speedups on benchmarks from FROSTT and from quantum chemistry.

CCS CONCEPTS

• **Computing methodologies** → **Linear algebra algorithms**; *Shared memory algorithms.*

ACM Reference Format:
Saurabh Raje, Hunter McCoy, Atanas Rountev, Prashant Pandey, and P. Sadayappan. 2025. FaSTCC: Fast Sparse Tensor Contractions on CPUs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3712285.3759841

1 INTRODUCTION

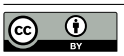
Tensor contractions generalize matrix-matrix multiplication to higher dimensions. Such contractions are important components of scientific computations and machine learning techniques. Loop permutation and loop tiling are key transformations for optimizing dense matrix/tensor computations and have been used for optimizing dense tensor contractions [18, 24, 29, 37].

The focus of our work are contractions on *sparse tensors.* A significant challenge to analysis and implementations of loop permutation and tiling for sparse tensor contractions is that any compact representation imposes restrictions on efficient element access patterns. For example, consider a D-mode tensor in the compressed sparse fiber (CSF) format [35]. Efficient access of nonzero elements along one mode, for fixed coordinates in the remaining $D-1$ modes, is only possible along the innermost mode in the CSF representation. With sparse matrices, since there are only two dimensions, loop transformations and loop tiling have been successfully employed for a number of sparse-matrix algorithms [4, 5, 23, 27, 28, 42].

Loop ordering and tiling for sparse tensor contractions. To our knowledge, no systematic analysis has previously been performed for loop permutation and tiling for optimizing higher-dimensional sparse tensor contractions. In related prior work, the TACO [19] tensor compiler automatically generates code for sparse tensor contractions on CSF tensors. TACO currently has some restrictions when the output is a sparse high-dimensional tensor, requiring a contraction-index-inner (CI) loop order (elaborated later in Sec. 2). Sparta [22] is a state-of-the-art sparse tensor contraction library that operates on input tensors in the COO format and uses a contraction-index-middle (CM) loop order. Neither framework uses loop tiling for optimizations.

A key goal of our work is to perform systematic analysis of the trade-offs between data access costs and memory requirements for sparse tensor contraction using different loop permutations, followed by an analysis for tiled execution of sparse tensor contractions. Although input tensors in a sparse tensor contraction can have an arbitrary number of modes, their indices can be categorized into three groups: (1) *contraction indices* – indices shared by both input tensors that are summed over during the contraction; (2) *external-left indices* – indices that appear in both the left input tensor and the output tensor; and (3) *external-right indices* – indices that appear in both the right input tensor and the output tensor. Thus an analysis of all possible loop orders is feasible even for tensors with an arbitrary number of modes.

It is well known that matrix-matrix multiplication $O_{ij} = A_{ik}B_{kj}$ allows six possible permutations of the three nested loops. Similarly, when tensor modes are grouped into contraction, external-left, and external-right indices, there are six possible loop orderings for a tensor contraction. However, due to the symmetry in Einstein notation, the order of operands (i.e., *LR* vs *RL*) has no semantic significance — making the distinction between external-left index and external-right index arbitrary. As a result, only three unique loop orders remain, corresponding to the position of the contraction index within the three-level loop structure.



SC '25, November 16–21, 2025, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/2025/11
https://doi.org/10.1145/3712285.3759841

In this work we systematically analyze the trade-offs among the three possible loop orders for sparse tensor contractions, focusing on data access overhead and memory usage. We show that the contraction-index-outer (CO) loop order minimizes data access overheads, but at the cost of significantly increased memory requirements for the output accumulators (where multiple contributions to sparse output elements are combined). However, by using tiling, the space requirement for output accumulators can be controlled, as well as made very efficient by matching its size to cache capacity.

FaSTCC [1]**.** Based on these insights, we develop FaSTCC (Fast Sparse Tensor Contractions on CPUs), a novel hashing-based parallel implementation of sparse tensor contractions that uses the CO loop order within tiles and 2D tiling along the output tensor dimensions. FaSTCC uses a probabilistic model based on the sparsity of input tensors to select between dense versus sparse accumulator for representing the tiles of the output tensor. Across a range of benchmark tensors, we show that FaSTCC performs much better than the current state-of-the-art frameworks, TACO [19] and Sparta [22], demonstrating the practical value of our approach.

Contributions. The novel contributions of this work are:

- We perform the first comprehensive comparative analysis of data access costs and memory requirements for loop permutations for sparse tensor contractions.
- We develop FaSTCC, to our knowledge the first multicore implementation of a 2D tiled scheme for sparse tensor contractions, with the option to use dense or sparse output accumulators.
- We develop a probabilistic modeling approach based on the estimated density of the output to automatically choose between dense and sparse output accumulator and to select tile size.
- We present an experimental evaluation on two multicore platforms, an 8-core desktop system and a 64-core server, across a range of benchmarks, to demonstrate *significant* performance improvements over state-of-the-art alternatives.

2 BACKGROUND

2.1 Sparse Tensor Contractions

A tensor T of order n is defined by a set of n modes, with mode $M_k = \{1, \ldots, N_k\}$ for $1 \le k \le n$. These modes define the index space of the tensor. $T_{i_1,\ldots,i_n}$ denotes the tensor element for a particular point in that index space, with $i_k \in M_k$. In a *sparse tensor* most $T_{i_1,\ldots,i_n}$ have zero numeric values; thus, standard dense representations of size $\prod_k N_k$ are wasteful. Instead, compact representations such as COO (COOrdinate format [34, 40]) or CSF (Compressed Sparse Fiber [35]) are used.

Consider two tensors L and R. A *tensor contraction* of L and R is a tensor O defined by

$$O_{l_1,\ldots,l_p,r_1,\ldots,r_q} = \sum_{c_1,\ldots,c_m} L_{l_1,\ldots,l_p,c_1,\ldots,c_m} * R_{c_1,\ldots,c_m,r_1,\ldots,r_q}$$

Here *contraction indices* $c_1, \ldots, c_m$ denote modes that are common to both tensors. These contraction indices are specified as part of the contraction definition. The remaining indices $l_1, \ldots, l_p$ and $r_1, \ldots, r_q$ are referred to as *external indices*.

Clearly, this is a higher-dimensional analog of matrix-matrix multiplication. In fact, the approach we define in this paper assumes that a pre-processing step has been applied to linearize the tuple $l_1, \ldots, l_p$ to a single index $l \in \mathbb{L}$. Similarly, $r_1, \ldots, r_q$ is linearized to an index $r \in \mathbb{R}$ and $c_1, \ldots, c_m$ is linearized to an index $c \in \mathbb{C}$. In our implementation such linearlization is applied as a pre-processing step, and the inverse delinearlization is applied as a post-processing step (both are accounted for in the measured execution time). Thus, the computation we aim to optimize is

$$O_{lr} = \sum_c L_{lc} * R_{cr},\ l \in \mathbb{L}, r \in \mathbb{R}, c \in \mathbb{C}$$

2.2 Sparse Tensor Representations

A variety of representations for sparse tensors have been considered in prior efforts. We outline the most relevant three representations.

The COO (Coordinate) format [40] stores a sparse tensor as a list of tuples, each of which describes a nonzero tensor element. For a tensor with n modes, each tuple contains $n + 1$ values, with the first n values representing index coordinates and the final one representing the numeric value of the tensor element. While the COO format is not as compact as other formats, it does support constant-cost insertions of new elements, since a new tuple can simply be appended to the end of the list. Due to its ease of construction COO is commonly used to read in input tensors and write out result tensors, with the tensor being converted from COO to a more optimized format for the targeted computations. Both Sparta [22] and FaSTCC consume COO input and produce COO output.

The CSF (Compressed Sparse Fiber) format [35] is based on a chosen outer-to-inner order of the tensor modes. CSF structures a sparse tensor as a tree. The internal nodes of this tree at a depth k represent the indices present in the k-th mode, and each leaf in the tree represents one nonzero element in the tensor.

Some approaches (e.g., Sparta [22]) employ hash tables to represent sparse tensors. A hash table maps a universe of keys to a universe of values. Internally, hash tables store keys by mapping them to an internal slot via a *hash function*, a function that deterministically maps input data to an output universe of a fixed size. Hash tables come in two categories, *open addressing* and *closed addressing* (or *chaining*). Open addressing tables use a hash function to map input keys to positions in a fixed size array: if the position chosen for a key is occupied, the key finds a new position via a probing scheme that determines which positions to probe in the array. Closed addressing schemes hash keys to a bucket data structure which can store any number of keys using chaining (or linked list). The chaining table used in Sparta is one such table, as keys are mapped to an internal linked list during insertion. Open addressing hash tables can achieve higher space efficiency and offer better data locality compared to chaining hash tables.

3 ANALYSIS OF LOOP ORDERS FOR SPARSE TENSOR CONTRACTION

In this section, we perform a comparative analysis of the data access costs for sparse tensor contraction for different *loop orders*. We first perform the analysis without considering tiling, and then in Section 5 extend the analysis for the tiled case.

A sparse tensor contraction is shown in an abstract form in Algorithm 1, using the notation from Section 2.

[1] FaSTCC code: https://github.com/HPCRL/fastcc

Algorithm 1: Abstract Sparse Tensor Contraction

for $l \in \mathbb{L}$ **do**
  for $c \in \mathbb{C}$ *with nonzero* L_{lc} **do**
    for $r \in \mathbb{R}$ *with nonzero* R_{rc} **do**
      $O_{lr} \leftarrow O_{lr} + L_{lc} * R_{rc}$

In the compressed sparse fiber (CSF) representation, tensor indices are typically ordered from left to right according to the outer-to-inner hierarchy, defining the tensor's layout. This means that efficient element access is possible only for loop orders that align with the tensor's layout: accessing elements in non-layout-compatible order requires costly searches. In contrast, using a hash table to store sparse tensor elements allows flexible and efficient access across different loop orders, depending on how the table is indexed.

Since the roles of the two input tensors ("left" or "right") are interchangeable in a contraction, only three unique loop orders need to be considered — determined by the position of the contraction index in the loop nest. Below, we examine these three loop orders and analyze their impact on data access costs in sparse tensor contractions. For each case, we first create appropriately indexed hash tables for the two input tensors.

3.1 Contraction-Inner (CI) Scheme

The input tensors are first placed in hash tables as follows:

$$HL : \mathbb{L} \rightarrow \mathcal{P}(\mathbb{C} \times \mathbb{V})$$

$$HR : \mathbb{R} \rightarrow \mathcal{P}(\mathbb{C} \times \mathbb{V})$$

$\mathbb{L}$ is the set of values for the left index l (assuming linearization if there are multiple left indices $l_1, l_2, \ldots$). Similarly, $\mathbb{R}$ is the set of values for the right index r and $\mathbb{C}$ is the set of values for the contraction index c. The numeric values are from set $\mathbb{V}$. We use $\mathcal{P}(X)$ denotes the powerset of X.

For each possible combination of l and r, this scheme requires the determination of the intersection of $HL(l)$ and $HR(r)$ in order to find pairs $\langle c, lv\rangle$ and $\langle c, rv\rangle$ with matching values of c.

Algorithm 2: Contraction-Inner (CI)

for $l \in \mathbb{L}$ **do**
  for $r \in \mathbb{R}$ **do**
    $sum \leftarrow 0$
    $update \leftarrow false$
    for $\langle c, lv\rangle \in HL(l) \wedge \langle c, rv\rangle \in HR(r)$ **do**
      $sum \leftarrow sum + lv * rv$
      $update \leftarrow true$
    if *update* **then**
      $Out.append(l, r, sum)$

With this scheme (Algorithm 2), the output tensor is constructed element-by-element by using a scalar variable *sum* to accumulate all contributions from matching pairs of nonzero elements from the two inputs. The TACO compiler [19] can automatically synthesize efficient tensor contraction code for the CI scheme, using CSF representations of the input tensors, where the contraction index is innermost in both input tensors. This scheme is therefore also called an "inner-inner" scheme. A CSF mode order with outer mode $\mathbb{L}$ and inner mode $\mathbb{C}$ allows iteration over l followed by c. In contrast to a hash table, CSF needs sorted indices in every dimension, and cannot resize dynamically. The cost of creating CSF is therefore $O(nnz * \log(nnz))$ where nnz is the number of nonzero elements. Furthermore, to obtain a nonzero value, this approach needs lookups in $2 \times n$ arrays where n is the order of the tensor.

3.2 Contraction-Middle (CM) Scheme

The input tensors are first placed in hash tables as follows:

$$HL : \mathbb{L} \rightarrow \mathcal{P}(\mathbb{C} \times \mathbb{V})$$

$$HR : \mathbb{C} \rightarrow \mathcal{P}(\mathbb{R} \times \mathbb{V})$$

Algorithm 3: Contraction-Middle (CM)

for $l \in \mathbb{L}$ **do**
  $WS \leftarrow \emptyset$
  for $\langle c, lv\rangle \in HL(l)$ **do**
    for $\langle r, rv\rangle \in HR(c)$ **do**
      $WS.upsert(r, lv * rv)$
  for $r \in WS.keys$ **do**
    $Out.append(l, r, WS(r))$

In Algorithm 3, the contraction index iterates in the middle loop. For each index $l \in \mathbb{L}$, all nonzero elements L_{lc} with external index l are extracted from the hash table HL. For each value c, nonzero elements R_{cr} are extracted from hash table HR by using c as the key. The product $L_{lc}R_{cr}$ is a contribution to O_{lr}. Accumulations to O_{lr} must be performed for each extracted R_{cr}. This must be done for all c corresponding to nonzero elements L_{lc}. A workspace WS is used for performing the accumulations to the appropriate output elements O_{l*}:

$$WS : \mathbb{R} \rightarrow \mathbb{V}$$

Either a dense array (along with some auxiliary data structures to keep track of which elements of the workspace are updated and become nonzero) or a sparse accumulator (using a hash table) may be used for WS. Update operation $WS.upsert(r, v)$ modifies the workspace as follows: if $r \notin WS.keys$, $WS(r)$ is set to v; otherwise, v is added to $WS(r)$.

3.3 Contraction-Outer (CO) Scheme

The input tensors are represented as follows:

$$HL : \mathbb{C} \rightarrow \mathcal{P}(\mathbb{L} \times \mathbb{V})$$

$$HR : \mathbb{C} \rightarrow \mathcal{P}(\mathbb{R} \times \mathbb{V})$$

In addition, a 2D workspace is used:

$$WS : (\mathbb{L} \times \mathbb{R}) \rightarrow \mathbb{V}$$

The workspace WS has keys that are pairs $\langle l, r\rangle \in \mathbb{L} \times \mathbb{R}$.

The CO scheme in Agorithm 4 has the contraction index as the outer loop. Both input tensors have their nonzero elements inserted into hash tables $HL(c)$ and $HR(c)$, indexed by the contraction index. For each value c of the contraction index that has nonzero elements in both $HL(c)$ and $HR(c)$, the product of each L_{cl} and R_{cr} must be formed and accumulated for output element O_{lr}. The workspace WS

Algorithm 4: Contraction-Outer (CO)

```
WS ← ∅
for c ∈ ℂ do
    for ⟨l, lv⟩ ∈ HL(c) do
        for ⟨r, rv⟩ ∈ HR(c) do
            WS.upsert(l, r, lv ∗ rv)
for ⟨l, r⟩ ∈ WS.keys do
    Out.append(l, r, WS(l, r))
```

is used to perform the accumulations. Operation $WS.upsert(l, r, v)$ updates the workspace as expected: if $(l, r) \notin WS.keys$, $WS(l, r)$ is set to v; otherwise, v is added to $WS(l, r)$.

3.4 Comparative Analysis of Loop Orders

We next compare the three schemes with respect to data access costs. For this analysis, we assume a dense workspace for performing accumulations for output tensor elements. For the output tensor, the number of accumulation operations is identical for all three schemes, and the main difference is the size of the workspace (which may in turn affect data access cost, if a small workspace can fit within cache but a large workspace requires DRAM accesses).

Data Access for Input Tensors: The input tensors are stored and accessed from hash tables. Each query incurs the cost of generating a hash value from the key and an access into the hash table to determine if the key exists. The payload for a successful query is not uniform, being directly proportional to the number of nonzero elements in the accessed slices of the tensor. We therefore separately quantify the number of hash table queries and the total volume (number of nonzero elements) of data retrieved over the full execution of the sparse tensor contraction.

CI: The CI scheme (Algorithm 2) computes the sparse inner product between every pair of left tensor slice $l \in \mathbb{L}$ and right tensor slice $r \in \mathbb{R}$. Thus, $O(L * R)$ queries to the input hash tables are required, where L and R are the extents of the respective index spaces of $\mathbb{L}$ and $\mathbb{R}$. For each such pair of slices from the left tensor and the right tensor, their nonzero elements must be co-iterated to find elements with matching values of the contraction index c. This is done efficiently if the nonzero elements are sorted in increasing value of c, but even so the volume of data access is very high: each slice of the right tensor must be accessed for each slice of the left tensor, with a total volume of $O(L * nnz_R)$, and similarly a volume of $O(R * nnz_L)$ for the left tensor, where nnz_L and nnz_R denote the number of nonzero elements in the left and right tensor, respectively.

CM: With the CM loop order (Algorithm 3), for each $l \in \mathbb{L}$ for the left tensor, the nonzero elements L_{lc} are accessed, along with queries to slices R_{cr} from the right tensor. The number of queries to the left tensor is L. Since each nonzero element in L causes a query to the right tensor, the total number of queries to the right tensor is nnz_L. The volume of data accessed for the left tensor is nnz_L because each nonzero element in the tensor is accessed once. Each element R_{cr} of the right tensor will be extracted for every nonzero L_{cl}. Therefore the total volume of data accessed for the right tensor is $\sum_{c \in \mathbb{C}} nnz_L(c) * nnz_R(c)$, where $nnz_L(c)$ and $nnz_R(c)$ denote the number of nonzero elements is the slices of the tensors for contraction index c. This sum can be approximated as $\frac{nnz_L}{C} * \sum_{c \in \mathbb{C}} nnz_R(c) = \frac{nnz_L * nnz_R}{C}$.

CO: With the CO loop order (Algorithm 4), each slice L_{c*} and R_{c*} is only accessed once. The number of hash table queries for the input tensors is $C + C = 2C$; the data volume is $nnz_L + nnz_R$.

Data Access for Output Tensor: For any of the loop orders, a temporary workspace must be used to accumulate contributions to nonzero output elements. While the size of the required workspace is affected by the loop order, the total number of accesses to the output workspace is independent of the loop order and equals the total number of multiply-accumulate operations.

Workspace Size: The three schemes impose very different demands on the size of the dense workspace. We quantify the space required.

CI: The output elements are processed one at a time and therefore only one scalar variable is needed for the workspace.

CM: If a dense workspace is used, a 1D array must be used, of size $\mathbb{R}$, i.e., R.

CO: With a dense workspace, a 2D array will be needed, whose size is the product of the ranges of $\mathbb{L}$ and $\mathbb{R}$, i.e., $L * R$.

Table 1: Comparison of data movement and space needed

Scheme	Queries	Data Volume	Size_Acc
CI	$O(L * R)$	$O(L * nnz_R + R * nnz_L)$	1
CM	$L + nnz_L$	$O(nnz_L + \frac{nnz_R * nnz_L}{C})$	R
CO	$O(2 * C)$	$nnz_L + nnz_R$	$L * R$

From the description of the schemes for the three loop orders and the analysis above, we can observe the following trade-offs:

- The *Contraction-Inner (CI)* scheme incurs the highest data access overhead because of lower reuse of data elements of the input tensors. However, the handling of the accumulations for the sparse output tensor is very straightforward and the scheme can be readily implemented for tensors of any dimensionality.
- The *Contraction-Middle (CM)* scheme is much more efficient with respect to the number of queries and the volume of data movement for input tensor elements than the CI scheme. However, the handling of accumulations requires a work-space, whose size depends on the extent R, for a dense workspace. In the case of very sparse high-dimensional output tensors, a dense workspace may be infeasible or inefficient to use if the product of the mode extents corresponding to $\mathbb{R}$ is very high. Sparta [22] implements the CM scheme as described in Section 7.2.
- The *Contraction-Outer (CO)* scheme is the most efficient in terms of accesses to input tensors. However, the required size for a dense output accumulator is problematic. Furthermore, even if the dense accumulator could fit in DRAM, update operations could be very slow due to the high latency to DRAM.

3.5 Tiled CO Scheme

From the above discussion, the CO loop order has the lowest number of data accesses but faces challenges with the output workspace. This challenge can be overcome by using 2D tiling along the linearized output tensor dimensions, so that the size of the output accumulator can be controlled by the chosen tile sizes.

Algorithm 5 outlines the 2D-tiled CO scheme. The output tensor's index space $\mathbb{L} \times \mathbb{R}$ is partitioned into $NL * NR$ tiles, where

Algorithm 5: 2D-Tiled CO scheme

```
Create NL left hash tables: HL_T[lt]
Create NR right hash tables: HR_T[rt]
for lt ← 0 to NL − 1 do
    for rt ← 0 to NR − 1 do
        Execute_Tiled_CO(lt, rt)
```

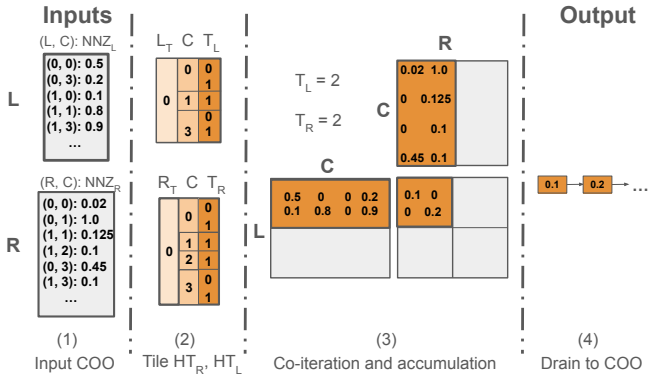


Figure 1: Intermediate steps of the FaSTCC contraction.

$NL = \lceil |\mathbb{L}|/T_L \rceil$ and $NR = \lceil |\mathbb{R}|/T_R \rceil$. Here T_L and T_R are tile sizes, selected as described later in the paper. The elements of the left input tensor are inserted from the input COO format into NL hash tables, where an element with index $\langle l, c \rangle$ is inserted into hash table $HL_T[lt]$, where $lt = \lfloor \frac{l}{TL} \rfloor$, and similarly for the right input tensor. A total of $NL * NR$ parallel invocations of instances of the 2D-tiled CO algorithm are dynamically scheduled on the available cores. Details are presented in the next section.

4 FASTCC DESIGN AND IMPLEMENTATION

In this section we present FaSTCC, an efficient parallel implementation of a 2D-tiled CO scheme for sparse tensor contractions. As introduced in the previous section, the $\mathbb{L} \times \mathbb{R}$ index space of the output tensor is partitioned into $NL * NR$ tiles, where $NL = \lceil |\mathbb{L}|/T_L \rceil$ and $NR = \lceil |\mathbb{R}|/T_R \rceil$.

4.1 Tiling and Workspace Design

Output tiles and input tiles. Consider an output data tile $Out_{i,j}$ where $0 \le i < NL$ and $0 \le j < NR$. The tile is indexed by intra-tile indices l and r such that $0 \le l < T_L$ and $0 \le r < T_R$. Element $Out_{i,j}(l, r)$ corresponds to $Out(i * T_L + l, j * T_R + r)$.

To compute $Out_{i,j}$, we need a 1D tile L_i of the left tensor L and a 1D tile R_j of the right tensor R. Here L_i corresponds to elements $L(c, l)$ such that $i * T_L \le l < (i + 1) * T_L$. Similarly, R_j corresponds to elements $R(c, r)$ such that $j * T_R \le r < (j + 1) * T_R$. To reflect this tiling of the inputs, we represent the left input tensor using NL hash tables of the form

$$HL_i : \mathbb{C} \to \mathcal{P}(\{0, \ldots, T_L - 1\} \times \mathbb{V})$$

and the right input tensor using NR hash tables of the form

$$HR_j : \mathbb{C} \to \mathcal{P}(\{0, \ldots, T_R - 1\} \times \mathbb{V})$$

Map HL_i represents input tile L_i while HR_j represents input tile R_j. The maps store intra-tile indices for the non-contraction data dimensions, together with the corresponding non-zero data values. For example, each $\langle l, lv \rangle \in HL_i(c)$ corresponds to an input element $L(c, i * T_L + l)$ with value lv.

FaSTCC algorithm. At the outermost level, Algorithm 6 iterates over output tiles $Out_{i,j}$. For every c such that both HL_i and HR_j contain some non-zero elements for c, the workspace WS accumulates the contributions to $Out_{i,j}$ due to c. The output tile is then "drained" to the output tensor, with appropriate remapping of intra-tile indices l and r.

Algorithm 6: FaSTCC contraction

```
for i ← 0 to NL − 1 do
    for j ← 0 to NR − 1 do
        WS ← ∅
        for c ∈ HL_i.keys do
            if c ∈ HR_j.keys then
                for ⟨l, lv⟩ ∈ HL_i(c) do
                    for ⟨r, rv⟩ ∈ HR_j(c) do
                        WS.upsert(l, r, lv * rv)
        for ⟨l, r⟩ ∈ WS.keys do
            Out.append(i * T_L + l, j * T_R + r, WS(l, r))
```

4.2 Implementation Details and Parallelization Techniques

The FaSTCC contraction has four steps: (1) construction of maps HL_i and HR_j, (2) iteration over matching positions of HL_i and HR_j, (3) accumulation of partial results in the workspace, and (4) draining the workspace into the output COO list. Next we describe the parallel implementation of each of these steps individually.

Parallel construction of hash tables. Recall that non-zero elements of the left operand tensor L are represented via hash tables $HL_i : \mathbb{C} \to \mathcal{P}(\{0, \ldots, T_L - 1\} \times \mathbb{V})$. Each element $L(l, c)$ with value lv is represented in map HL_i, where $i = \lfloor l/T_L \rfloor$, such that set $HL_i(c)$ contains a pair with the intra-tile index (e.g., l mod T_L) and the value lv. The representation of the right operand R is similar. This is illustrated in Figure 1, step 2 (tile hash tables). The example shows the computation of one output tile.

Construction of these hash tables is performed in parallel. Half the threads work on HL while the other half works on HR. This is implemented using OpenMP parallel regions with nested parallelism. Each thread in the left team reads the input tensor L in parallel, adding any points that are inside the thread's tile spaces to thread-local hash tables. For example, thread 0 in the left team is responsible for constructing all HL_i with i mod $num\_threads = 0$, thread 1 builds all HL_i with i mod $num\_threads = 1$, and so on.

Parallel co-iteration over HL_i and HL_j. We parallelize the co-iteration using a task queue. Each tile-tile contraction (i.e., each combination of i and j values that computes some output tile $Out_{i,j}$ in Algorithm 6) is defined as a separate task. These tasks are embarrassingly parallel, as each tile from the inputs is read-only, and each pair of input tiles $\{i, j\}$ uniquely writes one tile of output. Furthermore, since tasks are mapped to threads at run time, load imbalance

is much lower than it would have been if we partitioned the index space of the non-zero elements. We use Taskflow for implementing this task queue [15].

Parallel accumulation of partial results. The result of this contraction is accumulated into a thread-local tile WS_t. Based on an approach described later in Section 5.5, this can either be dense or sparse workspace. At the end of the accumulation, WS_t is drained into a thread-local COO linked list.

A dense tile structure includes: (1) a buffer *nnz* of size $T_L * T_R$ to hold non-zero elements of the tile, (2) a buffer *apos* of the same size to hold integers corresponding to the active positions in the tile, and (3) a bitmask *bm* with $\frac{T_L * T_R}{8}$ bits. An update at position p to this tile performs the following operations:

(1) Test and set bit p of bitmask bm
(2) If the initial value of $bm[p]$ was 0, append p to *apos*
(3) Update the non-zero element at $nnz[p]$ with the new value

The update operation takes constant time, and requires three random accesses to dense spaces. In case the accumulator is sparse, we use an open addressed hash table and the update operation lowers to an upsert operation on the hash table, which is expected to execute in constant time. This is shown in Figure 1, step 3, where the tile output data is computed from the input tile hash tables.

Parallel drain from tiles to COO output. Once the tile has been filled with data, the thread that owns that tile has to write the data to the COO list before working on the next tile. In case the tile is a sparse accumulator, the thread simply iterates over the underlying hash table and appends to the COO list each non-zero with key as co-ordinate and value as the data.

For dense tiles, we use array *apos* to perform the drain faster. The thread iterates over this array of active non-zero positions within the tile. It reads the non-zero elements from array *nnz* with positions as given in *apos*, i.e. $nnz[apos[iter]]$, and appends them to the COO list. Therefore, we iterate only over the non-zeros instead of iterating over the entire dense tile area of size $T_L * T_R$. This is shown in Figure 1, step 4 (drain to output COO) where the non-zeros from the tile are extracted and stored in the COO linked list.

COO representation for the output. With the above four steps, we construct one COO list per thread that represents disjoint sets of the sparse tensor data. The master thread concatenates these disjoint thread-local lists using pointer movements (no data copies) into one output COO result. We implement a memory pool layer to make the COO construction faster. Each thread gets heap allocations in chunks of 512MB as it pushes non-zeros to the thread-local COO. As the threads complete, they free up their local heap allocations.

5 SELECTION OF DENSE/SPARSE ACCUMULATOR AND TILE SIZE

In this section, we develop a modeling approach for (1) deciding whether a dense or sparse accumulator should be used for a given contraction, and (2) the size of the tile for the Tiled-CO scheme.

5.1 Estimation of Output Tensor Density

Chosing between dense and sparse accumulator, and the selection of tile size (for sparse accumulator in Sec 5.4) are both based on the density δ of the output sparse tensor. Exact computation of δ would essentially require as many operations as the actual tensor contraction. Hence we first estimate this density as follows.

We first assume a uniform random distribution of the nonzeros across the index spaces of input tensors L_{lc} and R_{cr}, $l \in \mathbb{L}$, $r \in \mathbb{R}$, $c \in \mathbb{C}$. An output element O_{lr} is nonzero if there exists at least one c such that both L_{lc} and R_{cr} are nonzero. The probability p_L that element L_{lc} is nonzero is its density:

$$p_L = \frac{nnz_L}{L * C}$$

Similarly, the probability p_R that element R_{cr} is nonzero is $\frac{nnz_R}{C*R}$. For some contraction index c, the probability that it contributes a nonzero to O_{lr} is

$$p_{overlap} = p_L * p_R$$

The probability that none of the C contraction indices contributes a nonzero is

$$P_{zero} = (1 - p_{overlap})^C$$

Hence, the probability that an output element O_{lr} is nonzero is

$$P_{nonzero} = 1 - P_{zero} = 1 - (1 - p_L * p_R)^C$$

This is also the probability density function (Φ_{res}) of the output tensor.

5.2 Choice of Dense or Sparse Accumulator

Dense accumulators enable more efficient update operations than sparse accumulators. However, from the analyses in Section 5.3 and Section 5.4, much larger tile sizes can be used with a sparse accumulator. If the output tensor is ultra-sparse, a significant fraction of tiles may not have any nonzeros if dense accumulators are used, because the tile size is limited by L3 cache capacity. However, with sparse accumulators, the tile size can be adaptively made larger for sparser output tensors. The analysis in Section 5.4 shows that the optimal tile size is inversely proportional to the square root of the density of the output tensor.

We compute the expected number of nonzeros in the output tensor tile. This expected number is used to chose between dense or sparse tiles. Based on the analysis in Section 5.1, the output probability density function is:

$$P_{nonzero} = 1 - (1 - p_L * p_R)^C$$

Given the assumption of uniform random distribution of the input nonzeros, this reduces to:

$$P_{nonzero} = 1 - (1 - \frac{nnz_L}{|L| * |C|} * \frac{nnz_R}{|R| * |C|})^{|C|}$$

The expected number of nonzeros in a region of size $T \times T$ of the output tensor is therefore:

$$E_{nnz}(T^2) = P_{nonzero} * T^2 = (1 - (1 - \frac{nnz_L}{|L| * |C|} * \frac{nnz_R}{|R| * |C|})^{|C|}) * T^2$$

Since we need all dense tiles (of data-type having DT bytes each) to fit in last level cache (L_3), we set $T^2 * N_{cores} * DT = L_3$. For contractions with $E_{nnz}(T^2) >= 1$, we use dense tiles; if $E_{nnz}(T^2) < 1$, we use sparse tiles.

5.3 Tile Size for Dense Accumulator

First, we extend the analysis from Section 3 to model the number of queries and accessed data volume from the input hash tables. Recall from the earlier section that we use L, R, and C to denote the extents of $\mathbb{L}$, $\mathbb{R}$, and $\mathbb{C}$ respectively. The analysis for each tile with a dense accumulator WS can be performed just as was done for the untiled CO algorithm in Section 3. For each tile, there would be $2 * C$ queries, resulting in a total query count of

$$N\_queries = 2 * C * NL * NR = \frac{2 * C * L * R}{T_L * T_R}$$

With Tiled-CO, each nonzero element in L will be accessed once per tile of R, i.e., $NR = R/T_R$ times in total across the entire execution. Similarly, each nonzero element of R will be accessed $NL = L/T_L$ times. Thus the total data volume of data access is

$$Data\_Vol = nnz_L * NR + nnz_R * NL = nnz_L * \frac{R}{T_R} + nnz_R * \frac{L}{T_L}$$

Thus, the number of queries as well as the total volume of data accessed are inversely related to the tile size, with larger tile sizes resulting in lower data access overhead.

Next, we discuss the impact of tile size on the accumulation operations. The total number of accumulation operations (i.e., updates to WS) is independent of the tile size. However, the access pattern for updating nonzeros in the dense accumulator can be expected to be very random, without any spatial or temporal locality. This is because we perform a sequence of outer products between nonzero elements L_{*c} and R_{c*}. For a given value of c, the elements O_{lr} cannot be expected to have any spatial locality unless there is pre-existing spatial locality among nonzeros in the input tensors. Further, for successive values of c, we do not expect any temporal locality. Therefore, in order to avoid expensive random access from DRAM, it is desirable to choose the tile size of a dense accumulator so that it can fit within cache. Since larger tile sizes reduce access overheads from the hash tables for the input tensors, we choose tile sizes to be as large as possible but still fit within cache.

The parallelization strategy is to have different threads concurrently operate on different tiles, and therefore we set the tile size $T = T_L = T_R$ such that $T * T * N\_cores = L3_{words}$, i.e., $T = \sqrt{\frac{L3_{words}}{N\_cores}}$, where $L3_{words}$ is the capacity of the L3 cache in double-precision words (cache capacity in bytes divided by eight). We demonstrate via experimental evaluation that such a choice is often the best or close to the best, with much smaller or larger tile sizes resulting in lower performance.

5.4 Tile Size for Sparse Accumulator

As described in Section 4, a hash table can be used as a sparse accumulator WS with the Tiled-CO scheme. The main benefit of using a sparse accumulator is that much larger tile sizes than $\sqrt{\frac{L3_{words}}{N\_cores}}$ can be used. Our main consideration is that the space occupied by the entries in the hash table stays within cache. If the density of the output sparse tensor is δ, the expected number of nonzeros in a $T * T$ region of the index space of the output tensor is $T * T * \delta$. Each hash table entry occupies 16 bytes: 8 bytes for a 64-bit key and 8 bytes for the double-precision data element. Aiming for 90% utilization of the hash table to avoid significant spills into DRAM gives $16 * T^2 * \delta = 0.9 * L3_{bytes}$, i.e., $T = \sqrt{\frac{L3\_bytes}{17.7 * \delta * N}}$.

5.5 Summary of Modeling Approach

Algorithm 7 describes the overall modeling approach. The output of this algorithm is (1) a choice whether to use a dense accumulator or a sparse accumulator, and (2) selection for tile size T. In the next section, we provide an experimental evaluation of this modeling.

Algorithm 7: Algorithm to determine whether to use sparse or dense tiles and the tile size. Inputs are number of cores (N_{cores}), last level cache size (L_3) and floating point width (DT)

$p_L = \frac{nnz_L}{|L|*|C|}$, $p_R = \frac{nnz_R}{|R|*|C|}$
$P_{res} = 1 - (1 - p_L * p_R)^{|C|}$
$T^2 = \frac{L_3}{N_{cores} * DT}$
$E_{nnz}(T^2) = P_{res} * T^2$
if $E_{nnz}(T^2) < 1$ **then**
  return $\langle sparse, \sqrt{\frac{L3\_bytes}{17.7 * \delta * N}} \rangle$
else
  return $\langle dense, T \rangle$

The algorithm selects between dense and sparse tile accumulators based on an estimate of the number of nonzeros that the tile may have. It computes this estimate with the assumption that nonzeros in the two sparse inputs follow a uniform random distribution. Given Φ_L and Φ_R as the density functions of the left and right operands, it computes Φ_{res}, the density function of the sparse result. The expected number of nonzeros in a tile is simply the product of Φ_{res} and per-core cache size.

If this expected number is less than 1, it means such a dense tile would likely be empty if constructed. In this scenario we use a sparse tile. Since this sparse tile is a hash table, its size is not simply the span of the co-ordinates that it indexes, but rather a function of the expected number of tile elements.

6 EXPERIMENTAL EVALUATION

Our experiments compare performance on two machines, using two state-of-the-art baselines, and 16 benchmark datasets:

Baselines: We compare our implementation FaSTCC against state-of-the-art compiler-generated code for sparse tensor contractions by TACO [19, 39] as well as Sparta [22], the state-of-the-art manually developed code for sparse tensor contractions.

Platforms: We evaluated performance on an 8-core Intel desktop system (Intel i7-11700F with 512KiB per-core L2 cache and a shared 16MiB L3 cache) and a 64-core AMD server (Ryzen Threadripper 3990X with 512KiB per-core L2 cache and shared 256MB L3 cache).

Datasets: We used the same sparse tensor benchmarks from the FROSTT [34] suite used by Li et al. [22] in evaluating Sparta. We further extended the benchmarking datasets by including sparse tensors arising with the state-of-the-art DLPNO (Domain Localized Pair Natural Orbital) method [30] from quantum chemistry.

Table 2: FROSTT tensor dimensions and size.

Tensor	Modes					NNZs
	0	1	2	3	4	
nips publications	2482	2862	14K	17		3.1M
chicago crime	6K	24	77	32		5.3M
vast 2015 mini challenge	165K	11.3K	2	100	89	26M
uber pickups	183	24	1140	1717		3.3M

6.1 Benchmarks

FROSTT benchmark suite. The FROSTT [34] benchmark suite is among the most commonly used datasets for evaluation of algorithms with sparse tensors. Table 2 shows the dimensionality, shape, and nonzero count of the tensors. Tensors from the FROSTT collection were used in experimental evaluation of the Sparta algorithm [22]. With each tensor, Liu et al. [22] performed two or three sparse tensor contraction evaluations, where the tensor was contracted with itself along one or more of its modes. For example, the 4D Chicago tensor was used in three evaluations, Chicago 0, Chicago 01, and Chicago 123, where the digits denote the modes that were contracted over. Since Chicago is a 4-mode tensor, the output tensor for these three cases would have 3+3=6 modes, 2+2=4 modes, and 1+1=2 modes, respectively. In our experiments, we use the same tensor contractions used by Liu et al. [22].

Sparse tensor contractions in quantum chemistry. In addition to the FROSTT benchmarks, we performed an evaluation on sparse tensor contractions that arise in linear scaling methods in quantum chemistry. Coupled cluster methods (CC) have traditionally been used in quantum chemistry to predict properties of large molecules. Recent efforts in computational chemistry have sought to reduce the asymptotic complexity of this method by considering only domain-localized (nearby) interactions of electron pairs. This is known as the DLPNO-CCSD method [3]. The bottleneck for scaling the DLPNO-CCSD method is in computing *four-centered integrals* from *three-centered integrals*. Specifically, we have the contractions:

$$Int_{ovov}(i,\mu,j,\nu) = TE_{ov}(i,\mu,k) \times TE_{ov}(j,\nu,k)$$

$$Int_{vvoo}(\mu,\nu,i,j) = TE_{vv}(\mu,\nu,k) \times TE_{oo}(i,j,k)$$

$$Int_{vvov}(\mu,\nu,i,\mu_1) = TE_{vv}(\mu,\nu,k) \times TE_{ov}(i,\mu_1,k)$$

The inputs are two 3D sparse tensors, which have to be contracted over one contraction index to produce a sparse 4D tensor.

In the following experiments, we benchmark these three contractions on two large molecules (Caffeine and Guanine) obtained using the TAMM system [25]. The first contraction is labeled *ovov*, the second *vvoo* and the third *vvov* in all the graphs that follow.

6.2 Tile Size Selection for Dense Accumulators

For experiments with dense accumulators, we select tile sizes as described in Section 5.3. The desktop system has a private 512KiB cache per core and a shared 16MiB L3 cache. Thus, each core has a share of 2 MiB of the L3 cache. With a wordsize of 8 bytes per double-precision word, the maximum tile size to stay within L3 cache is $\sqrt{\frac{2*1024*1024}{8}} = 512$. The 64-core server system has a 256 MiB shared L3 cache, giving each core a 4MiB share. Hence, the maximum tile size with dense accumulator for this system is $\sqrt{\frac{4*1024*1024}{8}} = 724$. We round this down to 512 since we need a power of 2 for the bitmask in the drain operation.

6.3 Estimating Density of Output Tensors

Based on the analysis in Section 5.2, we compute the expected number of nonzeros in the output tile for each contraction in the FROSTT tensor suite and the DLPNO contractions. The expected counts, model output, and running times are shown in Table 3. The times for dense and sparse accumulators are shown to compare the impact of the accumulator (the correct model decision) on the performance. For all contractions other than NIPS mode 2 and NIPS modes 2 and 3, the dense accumulator is selected by the model because the expected number of nonzeros in a dense tile is > 1. We therefore run FaSTCC with sparse tile accumulator for NIPS 2 and NIPS 23, and use a dense tile accumulator for all other cases.

Table 3: Model output for each experiment. First column is tensor name and contraction mode (in subscript). Time (seconds) for dense contraction is shown as $Time_D$, similarly for sparse. Model prediction for dense vs sparse accumulator is shown in column (D/S).

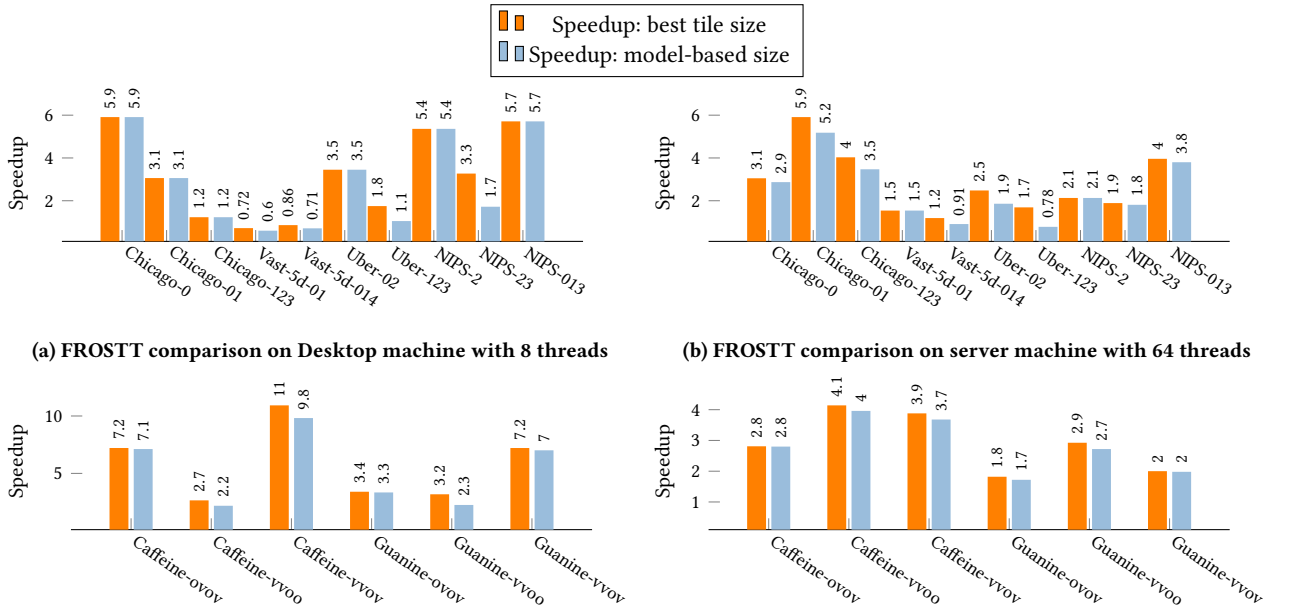
Tensor	p_L(%)	p_R(%)	$E_{nnz}(T^2)$	$Time_D$	$Time_S$	D/S
chic$_0$	1.46	1.46	4.79E+04	9.21	9.36	D
chic$_{01}$	1.46	1.46	65536	0.33	0.54	D
chic$_{123}$	1.46	1.46	6.55E+04	1.23	2.06	D
uber$_{02}$	0.04	0.04	2.00E+03	0.55	0.73	D
uber$_{123}$	0.04	0.04	6.55E+04	0.34	0.38	D
vast$_{01}$	7.78E-06	7.78E-06	7.38E+00	4.23	4.26	D
vast$_{014}$	7.78E-06	7.78E-06	6.54E+02	4.36	4.45	D
NIPS$_2$	1.83E-04	1.83E-04	3.08E-03	DNF	2.44	S
NIPS$_{23}$	1.83E-04	1.83E-04	5.24E-02	0.73	0.259	S
NIPS$_{013}$	1.83E-04	1.83E-04	2.65E+01	1.44	1.48	D
G-ovov	0.63	0.63	1.98E+04	0.315	0.566	D
G-vvoo	18.36	0.17	6.16E+04	11.28	12.12	D
G-vvov	18.36	0.63	6.55E+04	36.09	85.91	D
C-ovov	3.66	3.66	6.50E+04	0.219	0.566	D
C-vvoo	41.90	1.03	6.55E+04	3.79	4.305	D
C-vvov	41.90	3.66	65536	16.03	107.4	D

For sparse accumulators, we select the tile size as described in Section 5.4. These are rounded up to the next power of two. For NIPS 2, the resulting tile contains 1048576×1048576 elements, while for NIPS 23 the tile contains 262144×262144 elements.

6.4 Comparison with Sparta

Figure 2 shows the performance comparison with Sparta. TACO cannot generate code for parallel contraction of two sparse tensors with multi-threaded accumulation, and thus is not included in this comparison.

For most benchmarks, FaSTCC achieves significant speedups over Sparta. The Vast and Uber contractions have highly dense outputs with small dimensionality (order of 10K $L \times R$). The bottleneck for these contractions is creating hash tables HL_i and HR_j, which is the reason our approach does not show performance improvements over Sparta. Furthermore, Sparta uses a chaining-based hash table,



(a) FROSTT comparison on Desktop machine with 8 threads

(b) FROSTT comparison on server machine with 64 threads

(c) Quantum chemistry comparison on desktop machine with 8 threads.

(d) Quantum chemistry comparison on server machine with 64 threads.

Figure 2: Speedups over Sparta, with the best and model-chosen tile sizes on the FROSTT and quantum chemistry benchmarks.

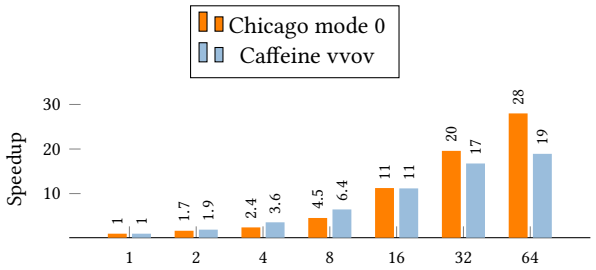


Figure 3: Factor improvement in run-time over single thread execution for the FaSTCC kernel from 1 to 64 threads.

which has fast insertions, while we use open addressing which is more expensive due to resizing costs at insertion.

6.5 Impact of Tile Sizes

Figure 4 compares execution times with different choices of tile sizes. As these measurements show, many benchmarks exhibit U-shaped patterns, implying that tile sizes that are too small or too large are not likely to achieve high performance. This highlihght the importance of selecting good tile sizes, which is the motivation for our modeling approach. As shown earlier in Figure 2a and Figure 2b, our modeling selects tile sizes that typically achieve performance close to that achieved with the best possible tile sizes.

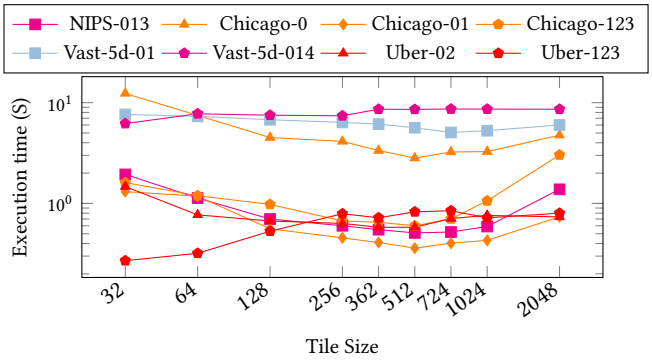
6.6 Comparison with TACO

Figure 5 compares the performance of TACO and FaSTCC on a single thread of the desktop machine, since TACO does not generate parallel code when the output tensor is sparse. The speedup achieved over TACO-generated code is shown on the y axis. FaSTCC is executed with the best tile configuration for the tensor. For several contractions, we can observe more than two orders of magnitude speedup over TACO-generated code.

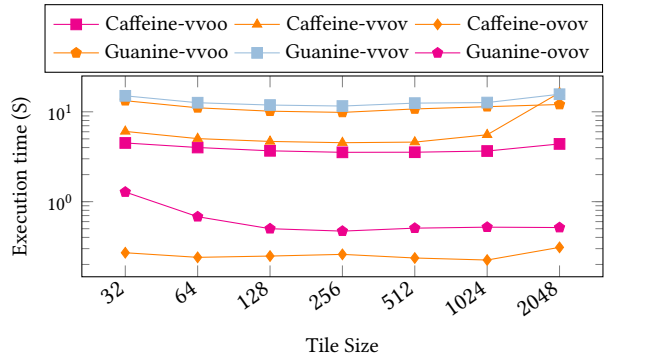
7 RELATED WORK

The approaches to efficient sparse tensor contractions fall into two categories: code generation and manual (library) implementations. Code generators [13, 19, 26, 41] such as TACO [19] perform ahead-of-time (AOT) compilation to produce kernels optimized for each instance of use of tensor contractions in an application program. On the other hand, library solutions [6, 8, 10, 16, 21, 22, 36] such as Sparta [22] expose pre-compiled library functions for sparse tensor contraction that are linked with programs that call those functions. FaSTCC falls in the second category and can be invoked by a client program that passes input tensors in multi-dimensional COO format and returns the computed tensor product in COO format. The implementation of FaSTCC is publicly available in the associated SC artifact, available at [31].

Sparse tensor contractions are only targeted by a subset [8, 10, 14, 19, 21, 22] of Tensor Algebra compilers and libraries, but have important applications in chemistry [1, 20] and quantum physics [11]. Sparse tensor contractions are also utilized in evaluating sparse tensor networks [17], where a sequence of contractions must be performed to contract a set of tensors [7, 32].

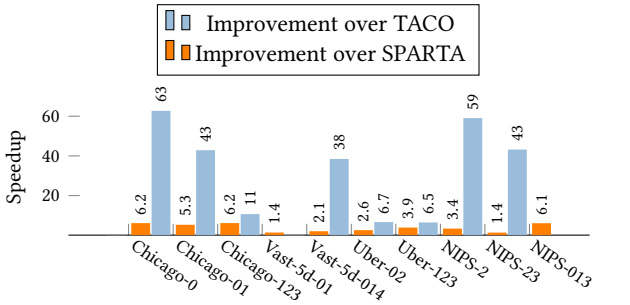


(a) Execution time variation with tile size: FROSTT

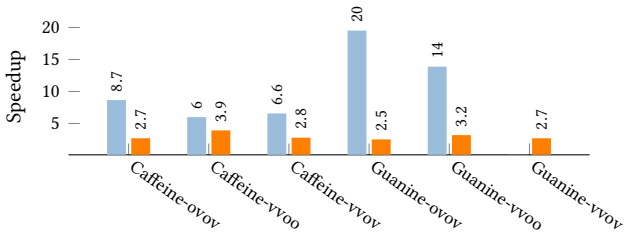


(b) Execution time variation with tile size: quantum chemistry

Figure 4: Execution time as a function of tile size.



(a) FROSTT benchmarks.



(b) Quantum chemistry benchmarks.

Figure 5: Speedups of FaSTCC for sequential execution.

7.1 Compiler-Based Code Generation

The work of Bik et al. represented the earliest attempt to automatically generate code for sparse matrix primitives [2]. The Tensor Algebra Compiler (TACO) [19] was the first compiler effort to combine dense and sparse linear algebra code generation. TACO generates efficient code for complex sparse tensor expressions that involve multiple operands and operators. It defines an abstraction of iteration graphs that determines how to co-iterate the hierarchical layout structures of multiple tensor operands. For the binary tensor contractions that are the focus of our work, TACO allows the programmer to insert a temporary dense workspace in which slices of the input tensors are contracted. Using a dense workspace provides constant-time reads and writes to the accumulator at the cost of increased space usage. The sparse polyhedral framework [44] optimizes the co-iteration between a pair of tensors for a tensor contraction using a Satisfiability Modulo Theory (SMT) solver.

Some recent efforts have introduced IR transformations for optimizing tensor product expressions with multiple sparse tensor contractions (sparse tensor networks). CoNST [32] uses an SMT solver to search over possible nested loop structures that implement a given binarization of a sparse tensor network with CSF (Compressed Sparse Fiber) tensors. It fuses common outer loops to reduce dimensionality of intermediate results, leading to reduced memory consumption. SparseLNR [7] splits the multi-term contraction of CSF tensors to pairwise contractions and also fuses loops to reduce intermediate dimensionality. Both systems rely on TACO to generate the C kernel that executes each pairwise tensor product.

7.2 Manual Implementations

Sparta [22] leverages the sparse accumulators and tables used in Athena [21] to efficiently perform sparse contractions. Sparta uses chaining hash tables to represent the tensors and implements the Contraction-Middle scheme discussed in Section 3.2. Pseudocode for Sparta's tensor contraction scheme is shown in Algorithm 8. Improvements over Sparta were reported by Feng et al. [9] by use of more advanced hashing techniques.

Algorithm 8: Sparse Tensor Contraction in Sparta

```
for L in L_Table do
    sparse_accumulator = []
    contraction_indices = L_Table[L]
    for c, l_val in contraction_indices do
        R_values = R_table[c]
        for R, r_val in R_values do
            sparse_accumulator[L,R] += l_val * r_val
    accumulators.push_back(sparse_accumulator)
```

Swift [8] and SpGETT [33] are two recently developed systems that implement hashmap-based sparse tensor contraction kernels on CPUs. While they improve upon the hashing scheme in comparison to Sparta, they do not explore other possible loop orders for the contraction. Furthermore, the accumulation is performed

over the entire span of external indices and tiled execution is not considered.

7.3 Custom Accelerators

Gamma [43] and Matraptor [38] are custom accelerators for sparse matrix operations. Gamma is an accelerator for multiplication of two sparse matrices. It uses a novel cache design and parallel processing elements to execute Gustavson's algorithm for inner-outer matrix multiplication [12]. Matraptor [38] uses a row-wise product and a novel data format called cyclic channel sparse row to perform sparse matrix-matrix multiplication efficiently.

8 CONCLUSION

This paper introduces FaSTCC, a library for fast sparse tensor contractions. By employing a novel 2D tiling scheme within the Contraction-Outer (CO) loop order, FaSTCC achieves reductions in memory overhead while enhancing data locality and computational efficiency. The developed adaptive approach for choosing between dense and sparse accumulators, based on a predictive modeling technique, enables high performance across diverse sparsity patterns and hardware platforms.

Experimental results show that FaSTCC outperforms state-of-the-art implementations, delivering improvements in running time and scalability across a variety of datasets. The contributions of this work, including a systematic analysis of loop orders together with the first (to our knowledge) contraction-outer sparse tensor contraction scheme using a tiling-based implementation, provide a strong foundation for further advancements in sparse tensor computation. Future work could extend these techniques to GPUs and heterogeneous systems, further broadening the applicability of FaSTCC to emerging computational challenges in fields such as machine learning, quantum chemistry, and physics.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation under awards 2009007, 2216903, 2217154, 2339521, 2517201, and 2513656. We thank the reviewers for their valuable feedback that helped us in improving this paper.

REFERENCES

[1] Edoardo Aprà et al. 2020. NWChem: Past, present, and future. *The Journal of Chemical Physics* 152, 18 (May 2020). https://doi.org/10.1063/5.0004997

[2] Aart J. C. Bik, Peter J. H. Brinkhaus, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. 1998. The automatic generation of sparse primitives. *ACM Trans. Math. Softw.* 24, 2 (June 1998), 190–225. https://doi.org/10.1145/290200.287636

[3] Jiri Brabec, Jakub Lang, Masaaki Saitow, Jiří Pittner, Frank Neese, and Ondřej Demel. 2018. Domain-Based Local Pair Natural Orbital Version of Mukherjee's State-Specific Coupled Cluster Method. *Journal of Chemical Theory and Computation* 14, 3 (2018), 1370–1382. https://doi.org/10.1021/acs.jctc.7b01184 arXiv:https://doi.org/10.1021/acs.jctc.7b01184 PMID: 29345924.

[4] Aydin Buluç. 2010. *Linear algebraic primitives for parallel computing on large graphs.* Ph. D. Dissertation. USA. Advisor(s) Gilbert, John R. AAI3398781.

[5] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel. 2011. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *2011 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 721–733. https://doi.org/10.1109/ipdps.2011.73

[6] Justus A. Calvin and Edward F. Valeev. 2015. Task-Based Algorithm for Matrix Multiplication: A Step Towards Block-Sparse Tensor Computing. arXiv:1504.05046 [cs.DC] https://arxiv.org/abs/1504.05046

[7] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. 2022. SparseLNR: Accelerating sparse tensor computations using loop nest restructuring. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 15, 14 pages. https://doi.org/10.1145/3524059.3532386

[8] Andrew Ensinger, Gabriel Kulp, Victor Agostinelli, Dennis Lyakhov, and Lizhong Chen. 2024. Swift: High-Performance Sparse Tensor Contraction for Scientific Applications. arXiv:2410.10094 [cs.DS] https://arxiv.org/abs/2410.10094

[9] Guofeng Feng, Weile Jia, Ninghui Sun, Guangming Tan, and Jiajia Li. 2024. POSTER: Optimizing Sparse Tensor Contraction with Revisiting Hash Table Design. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 457–459. https://doi.org/10.1145/3627535.3638500

[10] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. 2022. The ITensor Software Library for Tensor Network Calculations. *SciPost Phys. Codebases* (2022), 4. https://doi.org/10.21468/SciPostPhysCodeb.4

[11] E. Schuyler Fried, Nicolas P. D. Sawaya, Yudong Cao, Ian D. Kivlichan, Jhonathan Romero, and Alán Aspuru-Guzik. 2018. qTorch: The quantum tensor contraction handler. *PLOS ONE* 13, 12 (Dec. 2018), e0208510. https://doi.org/10.1371/journal.pone.0208510

[12] Fred G Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (1978), 250–269.

[13] So Hirata. 2003. Tensor Contraction Engine: Abstraction and Automated Parallel Implementation of Configuration-Interaction, Coupled-Cluster, and Many-Body Perturbation Theories. *The Journal of Physical Chemistry A* 107, 46 (2003), 9887–9897. https://doi.org/10.1021/jp034596z arXiv:https://doi.org/10.1021/jp034596z

[14] Rong Hu, Haotian Wang, Wangdong Yang, Renqiu Ouyang, Keqin Li, and Kenli Li. 2024. BCB-SpTC: An Efficient Sparse High-Dimensional Tensor Contraction Employing Tensor Core Acceleration. *IEEE Transactions on Parallel and Distributed Systems* PP (12 2024), 1–15. https://doi.org/10.1109/TPDS.2024.3477746

[15] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Trans. Parallel Distrib. Syst.* 33, 6 (June 2022), 1303–1320. https://doi.org/10.1109/TPDS.2021.3104255

[16] Khaled Ibrahim, Samuel Williams, and Evgeny Epifanovsky. 2015. Analysis and tuning of libtensor framework on multicore architectures. *2014 21st International Conference on High Performance Computing, HiPC 2014* (06 2015). https://doi.org/10.1109/HiPC.2014.7116881

[17] Raghavendra Kanakagiri and Edgar Solomonik. 2024. Minimum Cost Loop Nests for Contraction of a Sparse Tensor with a Tensor Network. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 169–181. https://doi.org/10.1145/3626183.3659985

[18] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2019. A code generator for high-performance tensor contractions on GPUs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 85–95.

[19] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM on Programming Languages* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[20] Christoph Köppl and Hans-Joachim Werner. 2016. Parallel and Low-Order Scaling Implementation of Hartree–Fock Exchange Using Local Density Fitting. *Journal of Chemical Theory and Computation* 12, 7 (2016), 3122–3134. https://doi.org/10.1021/acs.jctc.6b00251 arXiv:https://doi.org/10.1021/acs.jctc.6b00251 PMID: 27267488.

[21] Jiawen Liu, Dong Li, Roberto Gioiosa, and Jiajia Li. 2021. Athena: High-performance sparse tensor contraction sequence on heterogeneous memory. In *Proceedings of the 35th ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 190–202. https://doi.org/10.1145/3447818.3460355

[22] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 318–333.

[23] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient Block Algorithms for Parallel Sparse Triangular Solve. In *49th International Conference on Parallel Processing - ICPP (ICPP '20)*. ACM, 1–11. https://doi.org/10.1145/3404397.3404413

[24] Devin A Matthews. 2018. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing* 40, 1 (2018), C1–C24.

[25] Erdal Mutlu, Ajay Panyala, Nitin Gawande, Abhishek Bagusetty, Jeffrey Glabe, Jinsung Kim, Karol Kowalski, Nicholas P. Bauman, Bo Peng, Himadri Pathak, Jiri Brabec, and Sriram Krishnamoorthy. 2023. TAMM: Tensor algebra for many-body methods. *The Journal of Chemical Physics* 159, 2 (07 2023), 024801. https://doi.org/10.1063/5.0142433

[26] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2020. COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. In *Languages and Compilers*

for Parallel Computing: 33rd International Workshop, LCPC 2020, Virtual Event, October 14-16, 2020, Revised Selected Papers. Springer-Verlag, Berlin, Heidelberg, 87–103. https://doi.org/10.1007/978-3-030-95953-1_7

[27] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 68–78. https://doi.org/10.1109/ipdps49936.2021.00016

[28] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*. ACM, 90–106. https://doi.org/10.1145/3503221.3508431

[29] Nvidia. 2020. cuTENSOR: A high-performance CUDA library for tensor primitives. https://docs.nvidia.com/cuda/cutensor/index.html.

[30] Peter Pinski, Christoph Riplinger, Edward F Valeev, and Frank Neese. 2015. Sparse maps—A systematic infrastructure for reduced-scaling electronic structure methods. I. An efficient and simple linear scaling local MP2 method that uses an intermediate basis of pair natural orbitals. *The Journal of chemical physics* 143, 3 (2015).

[31] Saurabh Raje, Hunter McCoy, Atanas Rountev, Prashant Pandey, and Ponnuswamy Sadayappan. 2025. *FaSTCC: Fast Sparse Tensor Contractions on CPUs*. https://doi.org/10.5281/zenodo.16930011

[32] Saurabh Raje, Yufan Xu, Atanas Rountev, Edward F. Valeev, and P. Sadayappan. 2024. CoNST: Code Generator for Sparse Tensor Networks. *ACM Trans. Archit. Code Optim.* 21, 4, Article 82 (Nov. 2024), 24 pages. https://doi.org/10.1145/3689342

[33] Somesh Singh and Bora Uçar. 2024. *Efficient parallel sparse tensor contraction*. Technical Report RR-9551. Inria Lyon. https://hal.science/hal-04659658

[34] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. http://frostt.io/

[35] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 61–70.

[36] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190. https://doi.org/10.1016/j.jpdc.2014.06.002 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[37] Paul Springer and Paolo Bientinesi. 2018. Design of a high-performance GEMM-like tensor–tensor multiplication. *ACM Transactions on Mathematical Software (TOMS)* 44, 3 (2018), 1–29.

[38] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. https://doi.org/10.1109/micro50266.2020.00068

[39] TACO [n. d.]. TACO: The Tensor Algebra Compiler. http://tensor-compiler.org/

[40] R.P. Tewarson. 1973. *Sparse Matrices*. Academic Press. https://books.google.com/books?id=I-FQAAAAMAAJ

[41] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High Performance Sparse Tensor Algebra Compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 27–38. https://doi.org/10.1109/LLVMHPC54804.2021.00009

[42] Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. 2018. swSpTRSV: a fast sparse triangular solve with sparse level tile layout on sunway architectures. *ACM SIGPLAN Notices* 53, 1 (Feb. 2018), 338–353. https://doi.org/10.1145/3200691.3178513

[43] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, 687–701. https://doi.org/10.1145/3445814.3446702

[44] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-iteration. *ACM Trans. Archit. Code Optim.* 20, 1, Article 16 (Dec. 2022), 26 pages. https://doi.org/10.1145/3566054