Efficient Parallel Sparse Tensor Contraction

Somesh Singh and Bora Uçar

***Abstract*—We investigate the performance of algorithms for sparse tensor–sparse tensor multiplication operation (SpGETT). This operation, also called sparse tensor contraction, is a higher order analogue of the sparse matrix sparse matrix multiplication (SpGEMM) operation. Therefore, SpGETT can be performed first by converting the input tensors into matrices, then by calling high performance variants of SpGEMM, and last by reconverting the resulting matrix to a tensor. Alternatively, one can carry out the scalar operations underlying SpGETT in the realm of tensors without formulating using matrices. We discuss the building blocks in both approaches and formulate a hashing-based method to avoid costly search or redirection operations. We present performance results with the current state-of-the-art SpGEMM-based approaches, existing SpGETT approaches, and a carefully implemented SpGETT approach with a fine-tuned hashing method, proposed in this paper.**

***Index Terms*—hashing, tensor contraction**

I. Introduction

Tensors, or multidimensional arrays, are widely used in modeling and analyzing multidimensional data [8], [21], [34]. The breadth of the applications and their importance has led to the development of libraries covering different application needs. A common operation provided by those libraries is tensor tensor multiplication, also called tensor contraction, see for example Tensor Toolbox [3], Cyclops [38], and also others in a recent survey [33]. This operation takes two tensors and a set of indices along which to carry out the multiplication and produces another tensor. Suppose for example that $\boldsymbol{\mathcal{A}}$ is a 4-dimensional tensor of size $I \times J \times P \times Q$, and $\boldsymbol{\mathcal{B}}$ is 3-dimensional tensor of size $P \times Q \times L$. Two dimensions of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ have the lengths P and Q, and hence one can contract $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ in either or both of those dimensions. If we contract $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ on the dimension of length P, we will obtain $\boldsymbol{\mathcal{C}}$ of size $I \times J \times Q \times L$, where $c_{ijql} = \sum_{p=1}^{p=P} a_{ijpq} \cdot b_{pql}$, where the individual elements of the tensors are shown with the corresponding lower-case letters. Similarly, we obtain $\boldsymbol{\mathcal{C}}$ of size $I \times J \times L$, if we contract along the two agreeing dimensions, where $c_{ijl} = \sum_{p=1}^{P} \sum_{q=1}^{Q} a_{ijpq} \cdot b_{pql}$.

Tensor contraction is a higher order analogue of the ubiquitous matrix matrix multiplication (GEMM). In fact, tensor contraction can be cast as matrix matrix multiplication after suitably rearranging the tensors. For example, using the same tensors above, the results $c_{ijl} = \sum_{p=1}^{P} \sum_{q=1}^{Q} a_{ijpq} \cdot b_{pql}$, can be computed by rearranging $\boldsymbol{\mathcal{A}}$ into a matrix $\mathbf{A}$ of size $IJ \times PQ$ and $\boldsymbol{\mathcal{B}}$ into a matrix $\mathbf{B}$ of size $PQ \times L$ so that the matrix $\mathbf{C}$ is of size $IJ \times L$, where $\mathbf{C} = \mathbf{AB}$ contains the resulting elements of $\boldsymbol{\mathcal{C}}$ in a matrix. Given this relation with the GEMM, tensor contraction is dubbed GETT [40].

We investigate the GETT operation on large sparse tensors. This operation, called SpGETT, takes two sparse tensors and a set of contraction indices and performs the scalar multiply and add operations as summarized above. Much like the sparse variant of GEMM, called SpGEMM, SpGETT has many challenges due to low arithmetic intensity. We target efficient execution of SpGETT on shared memory parallel systems. Since SpGETT operation can be performed along different dimensions depending on the use case, an SpGETT library should provide a simple interface. As argued in previous work [4], [18], none of the dimensions should be favored or preferred in the interface. This is doable by adopting the well-known coordinate format, which stores all indices of a nonzero explicitly along with its value.

A first approach to implement SpGETT of two tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ converts them to matrices $\mathbf{A}$ and $\mathbf{B}$ such that the contractions indices are in the columns of $\mathbf{A}$ and in the rows of $\mathbf{B}$. Then an SpGEMM will compute the nonzeros of the output tensor. As the tensors can have multiple dimensions, each in the orders of millions, putting all possible tuples of contraction indices in the columns of $\mathbf{A}$ and the rows of $\mathbf{B}$ is not advisable. Instead one should use those tuples of contraction indices in which $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ have nonzeros. Finding the nonempty tuples of contraction indices for two tensors and numbering them consistently so that $\mathbf{A} \times \mathbf{B}$ computes the nonzeros of the desired tensor is a problem that does not appear in dense GETT nor in SpGEMM. We investigate two ways to tackle this problem in Section III, one is based on sorting, the other is based on hashing.

A more direct approach to implement SpGETT of two tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ keeps them as tensors and carries out the necessary multiply add operations. Two key problems here are to know which entries to multiply and where to add the result, which can again be tackled with a sorting or a hashing scheme. These two problems are raised by the multi-dimensional nature of both input and output tensors. While the first approach to SpGETT can rely on the existing high performance SpGEMM libraries, this second approach needs efficient implementation. We investigate SpGETT natively on tensors in Section IV and propose a hashing scheme (Section V) to store tensor nonzeros to avoid costly search and redirection operations. Parallelization of the multiply add operations is achieved by taking the hashing data structure into account.

Hashing arises in both types of algorithms for SpGETT. The hashing scheme should allow fast construction, and have low memory overhead and look-up time. When all elements to be hashed are unique and known before hand, static hashing approaches with expected linear time construction, linear memory overhead, and worst case constant time look-up are

• Somesh Singh is with LabEx MILYON and LIP (UMR5668 Université de Lyon - ENS de Lyon - UCBL - CNRS - Inria), 46 allée d'Italie, Lyon, France, F-69364. *E-mail: somesh.singh@ens-lyon.fr*.

• Bora Uçar is with CNRS and LIP (UMR5668 Université de Lyon - ENS de Lyon - UCBL - CNRS - Inria), 46 allée d'Italie, Lyon, France, F-69364. *E-mail: bora.ucar@ens-lyon.fr*.

possible, see for example earlier work [7], references therein and its predecessor [15]. In the SpGETT operation though the elements to hash are not known without preprocessing and have keys duplicates. For example, different nonzeros of the tensor $\boldsymbol{\mathcal{A}}$ can be in the same column of $\mathbf{A}$; or the nonzero positions of the output tensor $\boldsymbol{\mathcal{C}}$ are not available. As the static hashing methods are not possible, we introduce a parallel dynamic hashing method in Section V which works natively on tensors and enables fast SpGETT. The proposed hashing scheme is also effective and usable in converting tensors to matrices for the SpGETT via SpGEMM approach.

After presenting a brief background in Section II, we investigate the SpGEMM-based approach to sparse tensor contraction in Section III. We then describe in Section IV an adaptation of a well-known SpGEMM algorithm to the SpGETT case, which is the main contribution of this paper. A parallel fast hashing scheme to be used in this approach is proposed in a separate Section V, as it is of independent interest. We then compare the proposed SpGETT algorithm with two current state of the art SpGETT implementations in Section VI, one using SpGEMM routines and the other natively working in tensors. Section VII concludes the paper.

II. Background and Related Work

We briefly describe the terms and notations used in this paper. We denote tensors using boldface script letters as in $\boldsymbol{\mathcal{A}}$, matrices using boldface capital letters as in $\mathbf{A}$, vectors using boldface lower case letters as in $\mathbf{a}$ and scalars using lower case letters, as in a.

A tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ has d *modes* and is of *order* d. For an order d tensor $\boldsymbol{\mathcal{A}}$, one needs d indices to uniquely identify individual entries of $\boldsymbol{\mathcal{A}}$. For example a_{ijkl} is a nonzero of a 4D tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_I \times n_J \times n_K \times n_L}$. We refer to a subtensor obtained by fixing all except m indices of the tensor as a *m-order subtensor* of the tensor [43]. For example, in tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_I \times n_J \times n_K \times n_L}$, $\boldsymbol{\mathcal{A}}_{::k:} \in \mathbb{R}^{n_I \times n_J \times n_L}$ is a 3-order subtensor of $\boldsymbol{\mathcal{A}}$.

We use *Einstein notation* to represent tensor contractions whereby the indices (and modes) that appear in both input tensors are the *contraction* indices (and modes), and a summation over these indices is implied. The contraction indices (and modes) do not appear in the output tensor. The remaining indices (and modes) appear in the output tensor and are called the *external* indices. For example, consider the contraction of two 4D tensors, $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_I \times n_J \times n_P \times n_Q}$, $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{n_P \times n_Q \times n_K \times n_L}$, along two modes, P and Q to produce a 4D output tensor $\boldsymbol{\mathcal{C}}$. This operation is written as

$$\boldsymbol{\mathcal{C}}_{ijkl} = \boldsymbol{\mathcal{A}}_{ijpq}\boldsymbol{\mathcal{B}}_{pqkl} \quad \text{indicating}$$
$$c_{ijkl} = \sum_{p=1}^{n_P}\sum_{q=1}^{n_Q} a_{ijpq} \cdot b_{pqkl} \tag{1}$$

Indices $\{p,q\}$ are the contraction indices and $\{P,Q\}$ are the contraction modes, while $\{i,j,k,l\}$ are the external indices and $\{I,J,K,L\}$ are the external modes. More generally, we denote an ordered set of contraction modes of a tensor $\boldsymbol{\mathcal{A}}$ (for a specified order) using c_A and specific indices in the set c_A of contraction modes using boldface $\mathbf{c_A}$. Similarly, we denote an ordered set of external modes of a tensor $\boldsymbol{\mathcal{A}}$ (for a specified order) using e_A and specific indices in the set e_A of external modes using boldface $\mathbf{e_A}$.

A d-mode tensor with $d > 2$ can be *matricized*, or reshaped into a matrix. Consider a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$. The modes $S = \{1,2,\ldots,d\}$ can be partitioned into two disjoint sets $S_R = \{r_1,r_2,\ldots,r_p\}$ and $S_C = \{c_1,c_2,\ldots,c_{d-p}\}$, and mapped, respectively, to the rows and to the columns of a matrix $\mathbf{A} \in \mathbb{R}^{\{n_{r_1} \times n_{r_2} \times \cdots \times n_{r_p}\} \times \{n_{c_1} \times n_{c_2} \times \cdots \times n_{c_{d-p}}\}}$. We use $\mathbf{A} = \boldsymbol{\mathcal{A}}_{S_R \times S_C}$ to denote that the matrix $\mathbf{A}$ is obtained by matricizing the tensor $\boldsymbol{\mathcal{A}}$ with the partition S_R and S_C of the modes. Here it is convenient to refer to the rows and the columns of $\mathbf{A}$, respectively, by a p-tuple $\mathbf{r}$ and a $(d-p)$-tuple $\mathbf{c}$. In which case, a tensor nonzero $a_{i_1,i_2,\ldots,i_d}$ is mapped to a matrix nonzero $a_{\mathbf{rc}}$, where $\mathbf{r}$ corresponds to the indices in S_R and $\mathbf{c}$ corresponds to the remaining indices S_C. While it is also convenient to have the first p modes of $\boldsymbol{\mathcal{A}}$ to define S_R, it is not necessary. When the tensor $\boldsymbol{\mathcal{A}}$ is sparse, many rows and columns in $\mathbf{A} = \boldsymbol{\mathcal{A}}_{S_R \times S_C}$ will have only zeros, if all p-tuples from S_R and $(d-p)$-tuples from S_C are used as indices in $\mathbf{A}$.

The contraction of two tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ along a given set of contraction indices can be formulated as a matrix matrix multiplication by matricizing $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ suitably. The external indices of $\boldsymbol{\mathcal{A}}$ map to the rows of $\mathbf{A}$, and hence the contraction indices map to the columns of $\mathbf{A}$. Similarly, the contraction indices of $\boldsymbol{\mathcal{B}}$ map to the rows of $\mathbf{B}$, and the remaining indices map to the columns of $\mathbf{B}$. By slightly abusing the index notation, the sample contraction (1) can be written as

$$c_{ij,kl} = \sum_{p,q} a_{ij,pq} \cdot b_{pq,kl} \ ,$$

the two matrices can be recognized, and the whole computation can be succinctly written as $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. When $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are sparse, many $c_{ij,kl}$ will be zero, as the nonzeros in the 2-order subtensors $\boldsymbol{\mathcal{A}}_{ij::}$ do not necessarily share common indices with the nonzeros in the 2-order subtensors $\boldsymbol{\mathcal{B}}_{::kl}$.

There are a number of popular storage formats for sparse tensors, such as COO, F-COO [24], HiCOO [22], CSF [37] and its variant [29]. These formats each have certain advantages for certain operations, or for memory use [41]. The format COO corresponds to the well-known sparse matrix storage format called the coordinate format. In this format, each nonzero element is represented by storing its indices in all dimensions and its value separately. We use COO as the input and output format, as it is the most easier one for a user.

A. *Gustavson's algorithm for* SpGEMM

Gustavson's algorithm [17] is widely used for SpGEMM. Several multi-threaded CPU implementations of SpGEMM follow Gustavson's algorithm [2], [28] since it has less synchronization, lower memory traffic and simpler operations compared to the inner product and outer product formulations of SpGEMM. Gustavson's algorithm also underlies the algorithms for SpGETT investigated in this paper. We therefore summarize a parallel version of this algorithm in Algorithm 1.

As can be seen in Algorithm 1, Gustavson's algorithm proceeds row-wise on matrix $\mathbf{A}$. For each nonzero a_{ik} in a

Algorithm 1: Row-wise Gustavson's algorithm for SpGEMM $\mathbf{C} = \mathbf{A} \times \mathbf{B}$

1 **parfor** *nonempty* $\mathbf{A}_{i,:}$ **do**
2   **for** *nonzero* a_{ij} ***in*** $\mathbf{A}_{i,:}$ **do**
3   **for** *nonzero* b_{kj} ***in*** $\mathbf{B}_{k,:}$ **do**
4             $v \leftarrow a_{ij}.b_{kj}$
5   **if** $c_{ij} \in \mathbf{C}_{i,:}$ **then**
6                 $c_{ij} \leftarrow c_{ij} + v$
7   **else**
8   insert c_{ij} in $\mathbf{C}_{i,:}$
9                 $c_{ij} \leftarrow v$

row of $\mathbf{A}$, the kth row of matrix $\mathbf{B}$ is read and is scaled by a_{ik}. When the ith row of $\mathbf{A}$ is processed, the sum of the scaled rows of $\mathbf{B}$ generates the ith row of the output matrix $\mathbf{C}$. Since the rows of the output matrix $\mathbf{C}$ can be constructed independent of each other, Gustavson's algorithm exposes sufficient parallelism. Despite the highly parallel nature of Gustavson's algorithm, its efficient parallelization is challenging. This is so because neither the sparsity pattern of the output matrix, nor the number of nonzeros in the output matrix can be known without inspecting the input matrices. This may lead to load imbalance among the threads, since rows of the output matrix are assigned to threads, and different rows can necessitate varying number of operation counts. Accumulating the scaled rows of $\mathbf{B}$ in order to compute a row of $\mathbf{C}$ requires a method to quickly look-up for a scalar multiplication to be added to an existing entry in $\mathbf{C}$ (Line 6 of Algorithm1), or creating a new entry on $\mathbf{C}$ (Line 8). This is often implemented with a sparse accumulator (SPA) per row of the output matrix. SPA aids in efficient accumulation of intermediate products, which can be written back to the output matrix after all the nonzeros in the row of the output matrix have been computed. The design choice of the sparse accumulator depends on the sparsity of the inputs and output, and there are mainly four variants: using heap [2], [27], hashing [1], [11], sorting [6], and dense arrays [16], [31].

B. Related work on tensor contractions

A large body of prior work has tackled the problem of efficient tensor contractions. We primarily focus on the previous work targeting parallel sparse tensor contraction on shared memory systems.

TACO [20], COMET [42] are compilers for dense and sparse tensor computations, including tensor contractions. Given a tensor algebra expression and the preferred storage format, these automatically generate a tensor algebra kernel. Mosaic [5] is a sparse tensor algebra compiler that extends TACO. It generates a mix of natively generated TACO code and external library calls, which have high interoperability, for the specified tensor algebra expression. Sparse Polyhedral Framework [44] generates code for sparse tensor contractions. Cyclops Tensor Framework [39] enables automatic parallelization of sparse tensor computations, including sparse tensor contractions, expressed algebraically.

The Tensor Toolbox [3] provides a suite of tools in MATLAB for computations on tensors. It includes a method for sparse tensor contractions. ITensor [14] and Libtensor [19] are frameworks that support multithreaded, block-sparse tensor contractions. Sparta [26] is the current state-of-the-art for parallel element-wise sparse tensor contractions, using order-agnostic coordinate (COO) format. It employs a hash-based representation for input sparse tensors and implements a hash-based sparse accumulator. Furthermore, it proposes data placement strategies for optimizing sparse tensor contractions on tiered memory systems with DRAM and Intel Optane DC Persistent Memory Module (PMM). We compare our proposed methods against its implementation on homogeneous DRAM memory in Section VI. In SB-TC, we employ a novel dynamic hashing method for representing the input tensors as well as the sparse accumulator, which compliments the Gustavson's-like formulation of sparse tensor contractions. Athena [25] extends Sparta to efficiently perform a sequence of sparse tensor contractions on tiered memory systems.

C. Related work on hashing

As discussed before, hashing methods are in used in SpGETT via SpGEMM and also in the SpGETT approach working on tensors. We review some key concepts in the hashing methods suitable for our use case. Hashing is used to store and retrieve a set of items. A hash function maps the items, called keys, to a set of values which are then used for indexing the location of the keys in a table. In *static hashing*, one is given a set of distinct items in advance, and this set does not change. Once a static hashing structure constructed, it is only queried for the existence of items and retrieving them. In dynamic hashing, the set of items are not known in advance; they arrive throughout the execution of a program. In the typical use cases, the newly arrived items that are not present in the hash table are inserted into the table, and those that are present are retrieved. The static case allows one to develop worst case constant time look-ups. In this paper, the items will be the coordinates of tensor nonzeros, which are d-tuples for a d-dimensional tensor. The worst case time of $O(d)$ for a look-up is therefore asymptotically optimal.

There are several static [7], [13], [23], [32], [35] and dynamic [12], [30] perfect hashing methods proposed in the literature. Cuckoo hashing [30] is a well-known perfect hashing method with a constant time lookup in the worst case. It uses a set of c hash functions. For each arriving item, c values are computed, and one of them is used for indexing the location of the item. For look-up, one has to check then only c locations. Assuming that c is a small constant and the hash functions take $O(1)$-time to compute, the look-ups are thus asymptotically optimal. Insertions in some well-studied condition require expected constant time. SBhash introduced in this work is a dynamic perfect hashing method supporting incremental updates, which is tailored for operations involved in sparse tensor contractions The previous algorithms for sparse tensor contractions that employ hashing do not use perfect-hashing methods. To the best of our knowledge, SB-TC is the first to use a perfect-hashing method for sparse tensor contractions.

III. SpGETT via reduction to SpGEMM

Consider the tensor contraction operation $\boldsymbol{\mathcal{C}}_{e_A,e_B} = \boldsymbol{\mathcal{A}}_{e_A,c_A}\boldsymbol{\mathcal{B}}_{c_B,e_B}$, with external modes e_A and e_B corresponding to, respectively, those external modes of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$. The contraction modes c_A and c_B of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ have necessarily the same size, and are ordered consistently. We discuss here how to process the tensors so that the contraction can be computed first by invoking a high performance SpGEMM library and then by processing the resulting matrix to the tensor.

In order to use SpGEMM routines to compute $\mathbf{C} = \mathbf{AB}$ instead of the SpGETT above, the matrices should be defined according to matricizations $\mathbf{A} = \boldsymbol{\mathcal{A}}_{e_A \times c_A}$ and $\mathbf{B} = \boldsymbol{\mathcal{B}}_{c_B \times e_B}$. As discussed before, there may be many tuples of indices in external or contraction modes that are zero in $\boldsymbol{\mathcal{A}}$ or $\boldsymbol{\mathcal{B}}$. Typically the number of empty rows or columns in $\mathbf{A}$ and $\mathbf{B}$ will be much more than the number of nonzeros in the matrices, should all tuples in external or contraction modes are created as rows or columns in these matrices. As this sparsity will cause slow downs in SpGEMM software, either special SpGEMM libraries should be developed [18] or $\mathbf{A}$ and $\mathbf{B}$ should contain only non-empty rows and columns. We discuss the latter approach so that any high performance SpGEMM library can be invoked.

The nonempty subtensors $\boldsymbol{\mathcal{A}}_{e_A,:}$ define the nonzero columns of $\mathbf{A}$, and the nonempty subtensors $\boldsymbol{\mathcal{B}}_{:,e_B}$ define the nonzero rows of $\mathbf{B}$. In this case, to compute $\mathbf{C} = \mathbf{AB}$ the columns of $\mathbf{A}$ and the rows of $\mathbf{B}$ should be numbered consistently. That is, if an integer j is assigned to the column index $\mathbf{j}$ of a nonzero $a_{\mathbf{i},\mathbf{j}}$ in $\boldsymbol{\mathcal{A}}$ where $\mathbf{i}$ are the indices in the external modes and $\mathbf{j}$ are the indices in the contraction modes, then each nonzero $b_{\mathbf{j},\mathbf{k}}$ in $\boldsymbol{\mathcal{B}}$ should necessarily be assigned j as the row index. We refer to this requirement as the *consistency condition* on the matricization of two tensors. Furthermore, the rows of $\mathbf{A}$ should correspond to non-empty subtensors $\boldsymbol{\mathcal{A}}_{:,c_A}$, and the columns of $\mathbf{B}$ should corresponds to non-empty subtensors $\boldsymbol{\mathcal{B}}_{c_B,:}$. Once the multiplication is performed, the matrix $\mathbf{C}$ should be converted to the tensor $\boldsymbol{\mathcal{C}}$ by mapping each non-empty row index i of $\mathbf{C}$ to the corresponding $|e_A|$-tuple $\mathbf{i}$, and each non-empty column index j of $\mathbf{C}$ to the corresponding $|e_B|$-tuple $\mathbf{j}$. Converting the indices of the nonzeros of the matrix $\mathbf{C}$ to those of the tensor $\boldsymbol{\mathcal{C}}$ is referred to as the *tensorization*.

The matricizations of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ and the tensorization of $\mathbf{C}$ are coupled, as the nonzero rows of $\mathbf{A}$ and the nonzero columns of $\mathbf{B}$ define, respectively, the rows and the columns of $\mathbf{C}$. Therefore, we need to map an $|e_A|$-tuple $\mathbf{i}$ to an integer i, where $\boldsymbol{\mathcal{A}}_{\mathbf{i},:}$ is a non-empty subtensor, and also need the inverse of this map for the tensorization of $\mathbf{C}$. A similar discussion holds for $|e_B|$-tuples defining non-empty subtensors $\boldsymbol{\mathcal{B}}_{:,e_B}$ and the columns of $\mathbf{C}$.

Typically, sorting or hashing are used for operations similar to consistent matricizations of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, and the coupled tensorization of $\mathbf{C}$. We therefore discuss two schemes SB-Smat, which uses sorting, and SB-Hmat, which uses hashing, to perform SpGETT via a reduction to SpGEMM.

A. SB-Smat: *Sorting for SpGETT-via-SpGEMM*

The key steps in matricizing $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ for computing $\boldsymbol{\mathcal{C}}_{e_A,e_B} = \boldsymbol{\mathcal{A}}_{e_A,c_A}\boldsymbol{\mathcal{B}}_{c_B,e_B}$ via SpGEMM $\mathbf{C} = \mathbf{AB}$ are shown in Algorithm 2. This algorithm sorts the indices of the nonzeros of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ in the contraction modes together in order to obtain a consistent numbering of the columns of $\mathbf{A}$ and the rows of $\mathbf{B}$. Then, the indices of the nonzeros of each tensor in the external modes are sorted to obtain integer ids for each unique $|e_A|$-tuple and each unique $|e_B|$-tuple. It is necessary to keep a reverse map for the row and column ids of $\mathbf{C}$ for obtaining a tensor after SpGEMM.

Algorithm 2: Consistent matricization and coupled tensorization with sorting

1. Let $L_A(i)$ contain the c_A indices of the ith nonzero of $\boldsymbol{\mathcal{A}}$ and $L_B(j)$ contain the c_B indices of the jth nonzero of $\boldsymbol{\mathcal{B}}$
2. Sort L_A and L_B together into L, while keeping a reference to the original nonzero
3. Scan L (in the sorted order) to generate a unique integer id for each unique c_A-tuple, while keeping that integer id for the corresponding nonzero in $\boldsymbol{\mathcal{A}}$ or $\boldsymbol{\mathcal{B}}$
   `/* the column ids in the matrix` $\mathbf{A}$ `and the row ids in the matrix` $\mathbf{B}$ `are ready */`
4. Let $L_A(i)$ contain the e_A indices of the ith nonzero of $\boldsymbol{\mathcal{A}}$ and the integer id of the c_A indices of the same nonzero computed in Step 3
5. Sort L_A with respect to the e_A indices to obtain a unique integer id for each unique e_A-tuple, combine it with the integer id of the corresponding c_A indices for building $\mathbf{A}$ in the coordinate format. While doing so, keep a map from the unique integer id to a nonzero having the corresponding e_A indices for translating the row indices of the nonzeros of $\mathbf{C}$ to e_A indices in $\boldsymbol{\mathcal{C}}$
6. Perform Step 5 for the external indices of $\boldsymbol{\mathcal{B}}$ to obtain $\mathbf{B}$ and a map for translating the column indices of the nonzeros of $\mathbf{C}$ to e_B indices in $\boldsymbol{\mathcal{C}}$

Algorithm 2 creates $\mathbf{A}$ and $\mathbf{B}$ in the coordinate format, which are then converted to the CSC or CSR formats for invoking an existing SpGEMM library. The resulting matrix $\mathbf{C}$ is again in the CSC or CSR formats. A pass over the nonzeros is needed to translate the indices of the nonzeros of $\mathbf{C}$ to that of $\boldsymbol{\mathcal{C}}$ by using the maps created in Steps 5 and 6 of Algorithm 2. Note that when a $\mathbf{c_A}$ is nonzero and the corresponding $|c_B|$-tuple is zero, $\mathbf{B}$ will have an empty row. When $\mathbf{A}$ is processed in an SpGEMM row-by-row as in Algorithm 1, this do not create much overhead. Similarly, when a $|c_A|$-tuple is zero and the corresponding $|c_B|$-tuple is nonzero, $\mathbf{A}$ will have an empty column. This does not create an overhead for a row-by-row SpGEMM.

B. SB-Hmat: *Hashing for SpGETT-via-SpGEMM*

This method uses hashing for effecting consistent matricization of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ and for assigning ids to their external indices, $\mathbf{e_A}$ and $\mathbf{e_B}$. The aim is to take advantage of fast hashing methods when available. As arbitrary hashing methods cannot in general be competitive with the sorting approach, we propose an efficient hashing method later in the paper.

The key idea for the consistent matricization is to hash the contraction indices of the nonzeros in $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ together into a single hash table. We do this by inserting all contraction indices of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ in a single batch. Note that duplicates are likely in this batch, and hence static hashing methods are not well suited. Thus, we use a dynamic hashing approach as summarized below.

We maintain a counter to assign consecutive ids to unique contraction indices, $\mathbf{c_A}$. We also maintain an indirection array for all nonzeros in $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ combined, to store the id of contraction indices, $\mathbf{c_A}$ for every nonzero in both input tensors. We use the contraction indices as key for hashing. We scan the nonzeros of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$. For each nonzero in $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, we check if the contraction indices, $\mathbf{c_A}$, is present in the hash table. If it is not present, we insert $\mathbf{c_A}$ into the hash table, along with the new id that we assign to $\mathbf{c_A}$ using the counter, which is stored in the indirection array. If $\mathbf{c_A}$ for a nonzero is already present in the hash table, we retrieve its id from the hash table and write it to the index of the nonzero in the indirection array.

Next, in order to obtain the ids for the external indices of each nonzero, we process $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ separately. We create separate hash tables for $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ using the external indices of the nonzeros as hash key. As in the sorting-based algorithm, we create $\mathbf{A}$ and $\mathbf{B}$ in the coordinate format, which we call $\mathrm{COO}_\mathbf{A}$ and $\mathrm{COO}_\mathbf{B}$ respectively. Again, we maintain a global counter to assign consecutive ids to the distinct external indices, $\mathbf{e_A}$, of $\boldsymbol{\mathcal{A}}$. We first scan the nonzeros of $\boldsymbol{\mathcal{A}}$. For each nonzero in $\boldsymbol{\mathcal{A}}$ we check if the external indices, $\mathbf{e_A}$, is present in the hash table by performing a lookup. If it is not present, we insert $\mathbf{e_A}$ into the hash table, along with the new id that we assign to $\mathbf{e_A}$ using the counter. If $\mathbf{e_A}$ for a nonzero is already present in the hash table, we retrieve its id from the hash table and write it to the index of the nonzero in row-id in $\mathrm{COO}_\mathbf{A}$. Furthermore, we write the id of contraction indices of the nonzero, $\mathbf{c_A}$, as the column-id in $\mathrm{COO}_\mathbf{A}$ by accessing the indirection array previously populated. We also create a reverse map of external indices to their ids in order to be able to tensorize $\mathbf{C}$.

We follow the same procedure, as for $\boldsymbol{\mathcal{A}}$, for assigning ids to the external indices of $\boldsymbol{\mathcal{B}}$ and for creating the $\mathbf{B}$ by populating $\mathrm{COO}_\mathbf{B}$ appropriately.

As in the sorting-based algorithm, $\mathbf{A}$ and $\mathbf{B}$ are converted to the CSC or CSR formats for invoking an existing SpGEMM library. After that, the indices of the nonzeros of $\mathbf{C}$ are translated to $|e_A|$- and $|e_B|$-tuples for populating the nonzeros of $\boldsymbol{\mathcal{C}}$.

IV. SB-TC: SpGETT NATIVELY ON INPUT TENSORS

We present SB-TC to carry out parallel SpGETT $\boldsymbol{\mathcal{C}}_{e_A,e_B} = \boldsymbol{\mathcal{A}}_{e_A,c_A}\boldsymbol{\mathcal{B}}_{c_B,e_B}$ natively on tensors. It closely follows Gustavson's algorithm without explicitly matricizing the input tensors, and builds the output tensor subtensor by subtensor. The method is empowered by a novel parallel perfect hashing method to avoid expensive searching and sorting.

We present the proposed SpGETT method in Algorithm 3. This method can in principle use any dynamic hashing scheme. We therefore discuss this algorithm independent from the hashing approach, and defer the presentation of the proposed dynamic hashing scheme to the next section.

At a high level, SB-TC performs SpGETT such that the subtensors $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ of the resultant tensor are constructed independent of each other, in parallel, indicated by the parallel

Algorithm 3: SB-TC: SpGETT using Gustavson's algorithm-like formulation

Input : $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{d_A}}$, $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_{d_B}}$
e_A, c_A partitioning the modes $\{1, 2, \ldots, d_A\}$ of $\boldsymbol{\mathcal{A}}$ and
c_B, e_B partitioning the modes $\{1, 2, \ldots, d_B\}$ of $\boldsymbol{\mathcal{B}}$
Output: $\boldsymbol{\mathcal{C}}_{e_A,e_B} = \boldsymbol{\mathcal{A}}_{e_A,c_A}\boldsymbol{\mathcal{B}}_{c_B,e_B}$
1 Create a hash data structure for $\boldsymbol{\mathcal{A}}$ using the indices in the e_A modes as the key
2 Create a hash data structure for $\boldsymbol{\mathcal{B}}$ using the indices in the c_B modes as the key
3 **parfor** *nonempty* $\mathbf{e_A}$ *subtensor* $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ **do**
4 Initialize *SPA* /* Hash for the nonzeros in $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ */
5 **for** *nonzero* $a_{\mathbf{e_A},\mathbf{c_A}}$ ***in*** $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ **do**
6 **for** *nonzero* $b_{\mathbf{c_B},\mathbf{e_B}}$ ***in*** $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$ **do** /* $\mathbf{c_A} = \mathbf{c_B}$ */
7 $v \leftarrow a_{\mathbf{e_A},\mathbf{c_A}} \cdot b_{\mathbf{c_B},\mathbf{e_B}}$
8 **if** *SPA.lookup*($\mathbf{e_B}$) = *True* **then**
9 add v to *SPA*($\mathbf{e_B}$)
10 **else**
11 *SPA*.insert($\mathbf{e_B}, v$)
12 **for** *each tuple* $\mathbf{e_B}$ ***in*** *SPA with value* v **do**
13 set $c_{\mathbf{e_A},\mathbf{e_B}} = v$

parfor loop at Line 3 in Algorithm 3. In order to construct an output subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$, every nonzero in the subtensor $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ is to be multiplied (Line 7 in Algorithm 3) by all nonzeros in the appropriate subtensor $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$. The additive contributions from the product of pairs of nonzeros are assembled in a sparse accumulator (Lines 9 and 11). Finally, the contents of the sparse accumulator are written to output subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ (Line 13). The details of this algorithm are described below.

In order to efficiently produce a subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ of the output, the algorithm requires all nonzeros of $\boldsymbol{\mathcal{A}}$ having the same external indices $\mathbf{e_A}$ to be gathered. To accomplish this, we build a hash data structure $\mathcal{H}_\mathcal{A}$ for $\boldsymbol{\mathcal{A}}$ using the external indices of its nonzeros as the key. In this hash data structure, each unique external indices of $\boldsymbol{\mathcal{A}}$, $\mathbf{e_A}$, maps to a distinct location in the hash structure. Furthermore, a location in the hash data structure points to a dense contiguous array storing the nonzeros of $\boldsymbol{\mathcal{A}}$ that share the external indices $\mathbf{e_A}$. Note that all nonzeros in this array will have different indices in the contraction modes. Such a location in $\mathcal{H}_\mathcal{A}$ implicitly corresponds to a row of $\mathbf{A}$.

Since a nonzero $a_{\mathbf{e_A},\mathbf{c_A}}$ in the subtensor $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ is multiplied with all nonzeros in the subtensor $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$, the algorithm also requires all nonzeros in $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$ to be gathered. We thus build another hash data structure $\mathcal{H}_\mathcal{B}$ for $\boldsymbol{\mathcal{B}}$ using the contraction indices of its nonzeros as the key. In $\mathcal{H}_\mathcal{B}$, each unique tuple $\mathbf{c_B}$ of contraction indices maps to a distinct location. Furthermore, a location in the hash data structure points to a dense contiguous array storing the nonzeros of $\boldsymbol{\mathcal{B}}$ that share the same contraction indices $\mathbf{c_B}$. Note that all nonzeros in this array will have different indices in the contraction modes. Such a location in $\mathcal{H}_\mathcal{B}$ implicitly corresponds to a row of $\mathbf{B}$.

As all the nonzeros in $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are available at the outset, $\mathcal{H}_\mathcal{A}$ and $\mathcal{H}_\mathcal{B}$ can be built in parallel using batch insertion; a dynamic hashing scheme is necessary as distinct nonzeros can have the same contraction indices.

Given the hash data structures $\mathcal{H}_\mathcal{A}$ and $\mathcal{H}_\mathcal{B}$, computing the nonzeros of the output subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$, having the same indices in the e_A modes entails the following. We make a pass over the nonzeros mapped to a location in $\mathcal{H}_\mathcal{A}$ corresponding to $\mathbf{e_A}$, and for each nonzero performing a lookup of the contraction indices in $\mathcal{H}_\mathcal{B}$ to locate the corresponding location in $\mathcal{H}_\mathcal{B}$. We then go over the nonzeros in that slot of $\mathcal{H}_\mathcal{B}$ and multiply the values of the nonzeros in $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$. The hash data structure should thus have worst case constant time lookup to be efficient.

A sparse accumulator is needed in Algorithm 3 for nonzeros of $\boldsymbol{\mathcal{C}}$ having the same indices in the e_A modes—these correspond to a row of $\mathbf{C}$ in the SpGEMM formulation. This accumulator is again implemented as a dynamic hash data structure. For each different tuple of indices in the external modes e_A, a SPA is initialized as a hash table which uses the indices of the nonzeros of $\boldsymbol{\mathcal{B}}$ in the external modes as the key. Once all nonzeros of $\boldsymbol{\mathcal{A}}$ with the same indices $\mathbf{e_A}$ in the external modes e_A are processed, the sparse accumulator contains all nonzeros of $\boldsymbol{\mathcal{C}}$ which has the same $\mathbf{e_A}$ indices. Those nonzeros are written to $\boldsymbol{\mathcal{C}}$ out at Line 13. Since different threads write a chunk of nonzeros in $\boldsymbol{\mathcal{C}}$, a global counter is accessed atomically to reserve the positions of nonzeros produced by a thread—this can be implemented with light weight atomic fetch and add instruction.

A. Preprocessing

After creating the hash data structures $\mathcal{H}_\mathcal{A}$ and $\mathcal{H}_\mathcal{B}$, we perform two preprocessing operations before the start of the SpGETT computation (Line 3 in Algorithm 3), which aid in efficient computation of SpGETT with our proposed scheme.

1) Estimating the memory requirement of the output tensor: In the tensor contraction $\boldsymbol{\mathcal{C}}_{e_A,e_B} = \boldsymbol{\mathcal{A}}_{e_A,c_A}\boldsymbol{\mathcal{B}}_{c_B,e_B}$, in order to estimate the total number of nonzeros in the output tensor $\boldsymbol{\mathcal{C}}$ and the number of nonzeros per subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$, we apply the probabilistic estimation method proposed by Cohen [9]. While the method is originally proposed for SpGEMM involving sparse matrices, we apply it to for tensors. We use the hash data structures $\mathcal{H}_\mathcal{A}$ and $\mathcal{H}_\mathcal{B}$ of the input tensors, which have been already built, to conceptually matricize them and estimate the nonzero in their multiplication. Cohen's estimator does multiple rounds r to obtain a good estimate, where each round takes $O(nnz(\boldsymbol{\mathcal{A}} + nnz(\boldsymbol{\mathcal{B}}))$ time. In our tests $r = 2$ already produced a good estimate. For our use case, we empirically determined $r = 2$ to produce a good estimate as an upper bound for the number of nonzeros $\boldsymbol{\mathcal{C}}$ and in subtensors $\boldsymbol{\mathcal{C}}_{e_A,:}$. This estimation of the nonzeros is thus very practical and has a time complexity much less than computing $\boldsymbol{\mathcal{C}}$ or its nonzero pattern. Furthermore, its parallelization requires no communication among threads.

2) Load balancing: Like Gustavson's algorithm for SpGEMM, the parallel Gustavson's-like formulation of SpGETT suffers from load imbalance among threads due to disparity in the number of operations per subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ of the output tensor $\boldsymbol{\mathcal{C}}$. In order to mitigate the work load imbalance among threads, we make the assignment of subtensors to threads such that each thread gets assigned nearly equal operation count. We apply a parallel light-weight, load-balancing thread scheduling scheme proposed for Gustavson's algorithm for SpGEMM [28] to our formulation of SpGETT. This is made possible by the hash tables $\mathcal{H}_\mathcal{A}$ and $\mathcal{H}_\mathcal{B}$ already created, as a location corresponds to a row in the corresponding matricized view of the tensor.

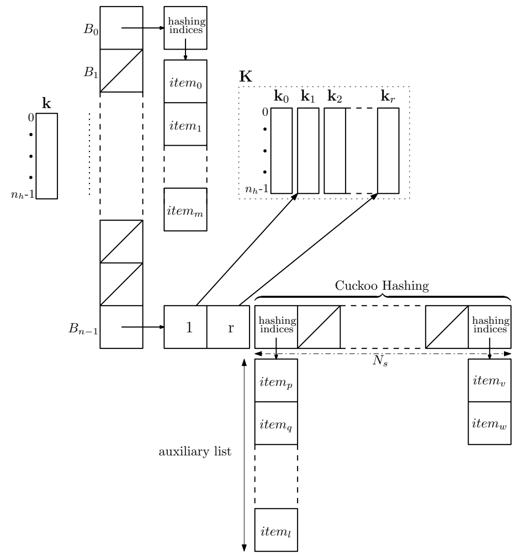


Fig. 1: SBhash data structure. In SB-TC, an item is $\{\langle$ indices$\rangle$, val$\}$. In SB-Hmat, an item is the *id* of nonzero in coordinate representation of tensor.

B. Optimizations for SpGETT in SB-TC

We make two observations in our proposed algorithm for SpGETT and apply optimizations that exploit these observations to enhance the performance of SB-TC. We discuss the two optimizations below.

1) Handling subtensors $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ of $\boldsymbol{\mathcal{A}}$ with a single nonzero: Consider subtensors $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ of $\boldsymbol{\mathcal{A}}$, having exactly one nonzero element. We observe that for such a subtensor of $\boldsymbol{\mathcal{A}}$, each product can be written directly to the output tensor at its unique position in the corresponding subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$. This is because there is no accumulation and thus there is no need for a sparse accumulator. As a result, we avoid maintaining a hash-based sparse accumulator for such subtensors of the output tensor.

2) Reducing the number of lookups to the sparse accumulator: For a subtensor $\boldsymbol{\mathcal{A}}_{e_A,:}$ of $\boldsymbol{\mathcal{A}}$ having more than one nonzero, we note that for the first nonzero in the subtensor with which we choose to scale the nonzeros in subtensor $\boldsymbol{\mathcal{B}}_{c_B,:}$, the partial products can be inserted at their unique positions in the sparse accumulator without having to perform a lookup. This is so as there are no prior entries in the sparse accumulator and the nonzeros in the subtensor $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$, all have a unique position. So, all the partial products can be safely inserted. We make use of this observation to reduce the number of lookups to the sparse accumulator. In each subtensor $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ of $\boldsymbol{\mathcal{A}}$, we determine the nonzero having the highest number of nonzeros in the corresponding subtensor $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$ and make it the first nonzero in that subtensor of $\boldsymbol{\mathcal{A}}$.

V. SBhash: A DYNAMIC PERFECT HASHING METHOD

We present a novel dynamic hashing method which we call SBhash. It is a perfect hashing method, that is, lookups to the hash data structure are answered in the worst case constant time per item size; as our data have d indices, the worst case constant time refers to $O(d)$ operations. The proposed hashing scheme allows fast insertion operations on a stream of d-tuples, using a given set H of $n_h = |H|$ indices, where $1 \leq n_h \leq d$, and one or more input items can have the same n_h-tuple. SBhash supports sequential insertions, as well as, batch insertions. Batch insertions can happen in parallel.

A. *Design of* SBhash

SBhash, as shown in Figure 1, has a two-level structure for perfectly hashing the nonzeros of a given tensor using a given set H of n_h indices. The first level is a set of buckets, to which each of the n_h-tuples of indices are mapped. Then, each bucket B_i has a set of slots, each of which uniquely corresponds to an n_h-tuple mapped to B_i. When $d > n_h$, there can be more than one nonzero whose n_h-tuple maps to a given slot in a bucket. A slot points to a collection of nonzeros that map to the slot. We maintain this collection as a contiguous array and it stores relevant information about each of the nonzeros. We refer to it as the *auxiliary list* of the slot.

For the first level, the hash function is defined as $h(\mathbf{k}, \mathbf{x}, p, n) := (\mathbf{k}^T \mathbf{x}_H \bmod p) \bmod n$, to find a bucket for a given nonzero $\mathbf{x}$. Here, n is the number of buckets, p is a prime number greater than n, the vector $\mathbf{k}$ is an n_h-tuple, and $\mathbf{x}_H$ is $\mathbf{x}$s indices in the hashing dimensions H. This function has been used before [7], [15].

For mapping the n_h-tuples to slots, in the second level of hashing, we use a variant of the standard Cuckoo hashing [30] adapted to the needs of the SpGETT operation. As this Cuckoo hashing variant is at the core of the proposed dynamic hashing scheme we discuss it in detail. This second level hashing needs to provide insert and lookup operations. As discussed before in Section IV, the lookup operations need to be performed one by one during the contraction operation. The insertions on the other hand can be one by one, and also in a batch as needed. Therefore, we detail the insertion, lookup, and parallel batch insertion below. While this hashing data structure can be used in other contexts and of independent interest, we do not discuss its generality.

B. *Our Cuckoo Hashing*

We employ a variant of the standard Cuckoo hashing. Given N_s slots and m items, each item has to be placed in one of the k slots chosen by ℓ random hash functions. This implicitly defines a bipartite graph where there are m vertices in one side, and N_s vertices in the other. Each items chooses ℓ slots by applying the ℓ hash functions. Then a perfect hashing will be obtained if we can perfectly match each item to a unique slot. While the standard Cuckoo hashing uses random walk for insertions, we carefully implement a deterministic method for insertion, which is described below. We use $\ell = 2$ in our setting. Furthermore, as seen in Figure 1 we maintain a pre-populated set of random n_h-tuples, $\mathbf{K}$. Consider that to a bucket B_i, b_i distinct n_h-tuples are mapped. If $b_i = 0$, then nothing is to be stored at that bucket. For buckets having b_i up to 4, we maintain the bucket as a packed dense array which stores the distinct n_h-tuples mapped to the bucket along with their auxiliary list. For buckets having $b_i > 4$, we store ids of two random keys in $\mathbf{K}$, which we use for hashing the items mapped to the bucket. The size of such a bucket is maintained at twice b_i. We store the n_h-tuples along with the auxiliary list.

1) Insertion: In order to obtain two different hash functions, we pick two keys by selecting two distinct n_h-tuples from the pre-populated set $\mathbf{K}$ of random n_h-tuples. In the bucket, we store the ids of the two n_h-tuples that we pick from $\mathbf{K}$. Consider a bucket B_i with N_s many slots having b_i distinct n_h-tuples mapped to it as a result of first level hashing. Let $\mathbf{k}_1^i$ and $\mathbf{k}_2^i$ be the two random hash keys for the bucket. Then the two hash functions defined for the bucket are: $h_1(\mathbf{k}_1^i, \mathbf{x}, p, N_s) := ({\mathbf{k}_1^i}^T \mathbf{x}_H \bmod p) \bmod N_s$ and $h_2(\mathbf{k}_2^i, \mathbf{x}, p, N_s) := ({\mathbf{k}_2^i}^T \mathbf{x}_H \bmod p) \bmod N_s$, where $N_s \geq 2b_i$.

When inserting a new item using our Cuckoo Hashing, we first check if one of its two possible slots is currently empty. If so, then the item is inserted into the empty slot. If both its possible slots are currently occupied, then we search for an *augmenting-path* starting at the new item with a breadth-first-search-like search. We maintain a *queue* to store the items to be processed, and repeat the following steps until the queue is empty or an augmenting path is found. We extract the item at the head of the queue ($item_q$). We traverse both its edges, one by one, by applying the two hash functions to the item—the edges are not stored explicitly. If the slot reached by an edge is already occupied and is unvisited, the slot is marked as visited and the item which was matched to the slot previously is enqueued. Additionally, in order to trace back the augmenting path, if we find one, we store the previous item for every visited slot; it corresponds to an edge from a slot to an item. If we reach a slot that is not already occupied, it means an augmenting path is found, so we stop. We then traverse this augmenting path to assign new slots to items.

If BFS ends without finding an unoccupied slot, then we need to change the hash functions and reinsert all the items from scratch. Now, the hash functions we use depend on the choice of the key as well as the number of slots. To update the hash function, we first increase the number of slots to the smallest power of 2 greater than $2\times$(# items), in order to create sufficient slots for the cuckoo hashing, and also to reduce the frequency of updating the number of slots. We then pick a pair of keys from the pool of keys. With the chosen keys, we compute the two hash functions for all the items. As a first step, we check for every item if one of its two possible slots is currently empty. If so, then the item is inserted into the empty slot. For each of the items that could not be assigned a slot in the first step, we find an augmenting path. If we fail to find an augmenting path for any of these unmatched items, we pick a different pair of keys and repeat.

While Cuckoo hashing is efficient, it may not be the best choice for a small number of items. We found empirically that keeping up to four items as a compact array for insertions and lookups is faster than using Cuckoo hashing. We therefore use the discussed Cuckoo hashing for managing buckets having at least five different n_h-tuples.

2) Lookup: The lookup to test if a given item is present is done straightforwardly. The two hash functions are evaluated for the item to find its potential slots. If both slots are empty, then the item is not present. Otherwise, if a slot contains an element it is compared withe given item.

C. Parallel Batch Insertion in SBhash

SBhash can perform a batch insertion of the items when they are available at the outset.

As a first step, it computes the bucket id for each item by applying the first level hash function to each item independently, in parallel. We populate an array, of size number of items, with the bucket ids of the items. Next, we insert the items into the SBhash data structure, in parallel. We parallelize the loop over items—assigning items to threads and every thread handling equal number of items. For each item, we determine its bucket id by performing a lookup on the array we populated in the first step. We then insert the item into its assigned bucket following the procedure described in Section V-A. Now, more than one item, handled by different threads, can potentially map to the same slot of a bucket. To ensure thread-safe behavior, when adding an item into a slot, we determine its position in the auxiliary list by atomically incrementing the number of items already present in the list. The auxiliary list is maintained as a dynamic array which is resized by doubling its size when full. The resizing of the list involves allocating a new list double the size, copying all of the existing items over to the new list, and inserting the new item in the next available position. This needs to be done as one atomic operation. To resize the list, a thread acquires a lock on the concerned slot. Thanks to the use of doubling resizing, the resizing is not frequent. Furthermore, if the insertion of an item invokes rehashing of the items of the bucket using Cuckoo hashing, a thread acquires a lock on the bucket, performs the rehashing and then releases the lock. This is because rehashing needs to be performed sequentially by one thread. Note that rehashing of the items of a bucket is infrequent and is required to be done only for a few buckets.

VI. Evaluation

We carry out the experiments on a machine having Intel Xeon E7-8890 v4 CPU with 96 cores (four sockets, 24 cores each), clock-speed 2.20GHz, 240 MB L3 cache and 1.5 TB memory. The machine runs Debian GNU/Linux 11 (64-bit). The codes are compiled with g++ version 13.2.0 with the flags `-O3`, `-std=c++17` and `-fopenmp` for OpenMP parallelization. We present experiments on real-life tensors from FROSTT [36]. Table I summarizes the characteristics of the real-life tensors in our test-suite. All our codes are available at /path/to/repository.

We perform a comparative study of the performance of different methods for SpGETT: SB-TC, SB-Hmat, SB-Smat, and the existing state-of-the-art method for sparse tensor contraction, Sparta [26]. We evaluate all methods on the SpGETT operation $\mathcal{C}_{e_A,e_A} = \mathcal{A}_{e_A,c_A}\mathcal{A}_{c_A,e_A}$, where $\mathcal{A}$ is a sparse tensor and $\mathcal{C}$ is the resultant tensor obtained by contracting $\mathcal{A}$ with itself along the specified contraction modes, c_A.

Tensor name	Order	Dimensions	*nnz*
nell-1	3	2,902,330 × 2,143,368 × 25,495,389	143,599,552
nell-2	3	12,092 × 9,184 × 28,818	76,879,419
delicious-4d	4	532,924 × 17,262,471 × 2,480,308 × 1,443	140,126,181
flickr-4d	4	319,686 × 28,153,045 × 1,607,191 × 731	112,890,310
enron	4	6,066 × 5,699 × 244,268 × 1,176	54,202,099
chicago_crime	4	6,186 × 24 × 77 × 32	5,330,673
uber	4	183 ×24 × 1,140 × 1,717	3,309,490
nips	4	2,482 × 2,862 × 14,036 × 17	3,101,609
vast-2015-mc1-5d	5	165,427 × 11,374 × 2 × 100 × 89	26,021,945
lbnl-network	5	1,605 × 4,198 × 1,631 × 4,209 × 868,131	1,698,825

TABLE I: Real-life sparse tensors in our test-suite, their order, the size in each dimension, and the number of nonzeros.

Recall that this operation can be viewed as an SpGEMM operation $\mathbf{C} = \mathbf{A}\mathbf{A}^T$, where $\mathbf{A}$ is a sparse matrix obtained by matricizing $\mathcal{A}$, such that e_A indexes the rows and c_A indexes the columns of $\mathbf{A}$; $\mathbf{C}$ is the resultant square matrix of size $|e_A| \times |e_A|$.

A d-dimensional input tensor can be contracted with itself along any combination of k modes, for $1 \leq k < d$. Thus, there are $\sum_{k=1}^{d-1}\binom{d}{k}$ or $(2^d - 2)$ distinct number of possible contractions. We refer to each of these contractions as an *input instance*. For example, for the five dimensional tensor lbnl-network, all 1-mode, 2-mode, 3-mode and 4-mode contractions with itself give rise to $(2^5 - 2) = 30$ input instances. Across all tensors in Table I, there are a total of 156 input instances. The time taken for SpGETT is contingent on the number and distribution of nonzeros in the input tensors and the resultant tensor, the contraction modes and the number of floating point operations. Testing SpGETT on the tensors from Table I with different contraction modes thus helps us cover various scenarios.

For all methods, we evaluate the performance of their parallel and sequential execution. For parallel execution, we consider thread counts of $\{16, 32, 48, 64, 80, 96\}$. For performance comparison, we only consider the input instances for which Sparta's sequential execution takes between one second and one hour to compute the tensor contraction. There are 85 such input instances, out of the total 156, which we use for parallel execution as well. For all methods, on every instance that we consider, we report the geometric mean of the execution time of three independent runs.

We begin the evaluation by first comparing the performance of SB-Smat and SB-Hmat in Section VI-A. We then study the performance of the better of the two methods, SB-TC and Sparta in sequential and parallel settings in Section VI-B.

A. Performance comparison of SB-Smat and SB-Hmat

We compare SB-Smat and SB-Hmat to identify the best performing variant of the method SpGETT via SpGEMM. These two methods differ only in their approach to matricization. The subsequent steps, sparse matrix–sparse matrix multiplication using a SpGEMM library, and the conversion of the resultant matrix to a tensor are common to both the

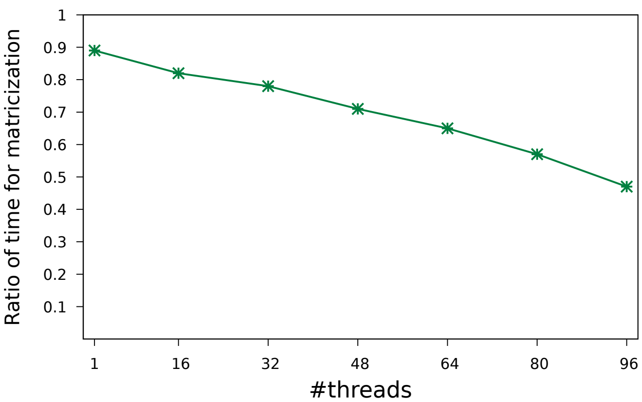


Fig. 2: Geometric mean of the ratio of the time for matricization in SB-Hmat to that in SB-Smat, across all 85 input instances, with $\{1,\ 16,\ 32,\ 48,\ 64,\ 80,\ 96\}$ threads. Lower value on the y-axis depicts better relative performance of matricization in SB-Hmat.

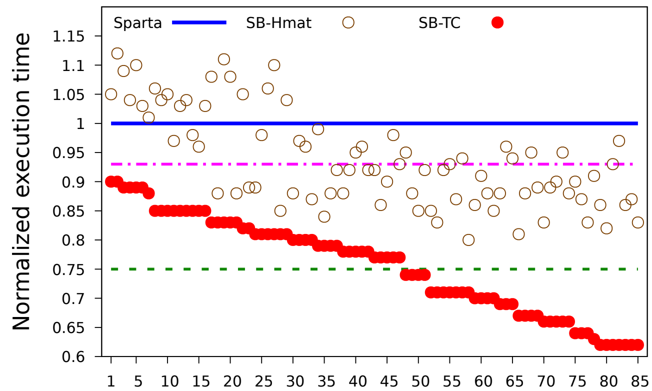


Fig. 3: The run time of SB-TC and SB-Hmat normalized by that of Sparta for sequential execution.

Tensor name	**Contraction modes**	**#flops** $\times 10^9$	**output** nnz $\times 10^9$	**Preprocessing (s)**	**SpGETT (s)**
enron	{1,3}	47.13	6.93	23.41	6383.67
chicago_crime	{2,3}	72.81	16.93	2.18	20424.06

TABLE II: Run time (in second) of Sparta on two representative instances for which Sparta computed the tensor contraction in more than one hour.

methods. We use CXSparse [10] library for SpGEMM. Thus, it suffices to compare the performance of the matricization step alone in the two methods to study which of the two has a superior performance. SB-Smat uses an implementation of quicksort available in Sparta library [26].

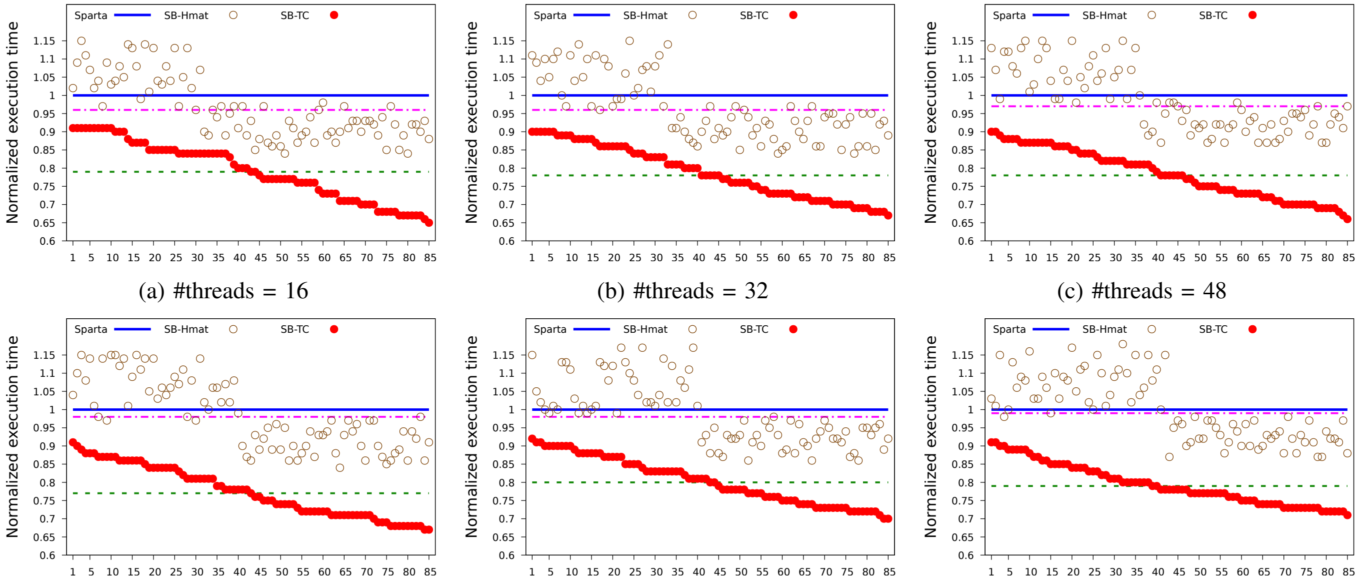
Figure 2 compares the overall performance of matricization in SB-Hmat and SB-Smat, across all input instances for different number of threads (on the x-axis). In the figure, the ratio of the run time of matricization in SB-Hmat to that in SB-Smat is computed for each of the 85 input instances at a given thread count, and the geometric mean of those 85 ratios is plotted. As seen here, the ratio is less than 1 for all thread counts, thus the matricization in SB-Hmat is consistently faster than that in SB-Smat. In order to give further insight into the performance, we note that the geometric mean of the time for matricization in sequential SB-Smat is 39.21 second, while that in sequential SB-Hmat is 34.89 second. Furthermore, across all input instances, the geometric mean of the time for matricization in parallel SB-Smat with 96 threads is 5.87 second and that in parallel SB-Hmat with 96 threads is 2.70 second. Thus, we conclude that SB-Hmat is more suitable for SpGETT via SpGEMM.

B. Performance comparison of SB-Hmat*,* SB-TC *and* Sparta

We start by comparing the sequential run time of all three methods. Figure 3 shows the relative performance of sequential execution of SB-TC and SB-Hmat with respect to Sparta on all input instances. In the y-axis we see the total run time of SB-TC, SB-Hmat, and Sparta normalized by the total run time of Sparta for all instances. On the x-axis, the instances are arranged in nondecreasing order by the ratio of the run time of SB-TC to Sparta from left to right. The figure also shows the geometric mean of the ratio of SB-TC's run time to that of Sparta (dashed line) and the geometric mean of the ratio of SB-Hmat's run time to that of Sparta (dot-dashed line). Lower values on the y-axis for SB-TC and SB-Hmat thus depict better performance with respect to Sparta. We see from the figure that for all input instances the performance of SB-TC is always better than the other two methods. SB-TC is 25% faster on average compared to Sparta across all instances, enjoying up to 38% better run time (flickr-4d with contraction modes $\{0{,}2\}$). We observe that SB-Hmat is faster than Sparta on 65 input instances out of the 85. SB-Hmat is up to 20% faster than Sparta (on delicious-4d with contraction modes $\{0{,}1\}$) and up to 12% slower than Sparta (on enron with contraction modes $\{1{,}2\}$). Overall, SB-Hmat is 7% faster than Sparta across all input instances.

We further note that across all the input instances, the time for the multiply add operations takes a majority of the total execution time. For Sparta, the preprocessing time accounts for 7.58% of the total time on average. For SB-TC, the preprocessing time accounts for 6.13% of the total time on average. For SB-Hmat, the time for preprocessing and postprocessing combined accounts for 13.44% of the total time on average. Therefore, the performance difference between Sparta and SB-TC in the run time is due mostly to the efficiency in the data access method during the multiply add operations. On the other hand, SB-Hmat has a less involved data access pattern than Sparta as it works on the CSR representation of matrices. Note that all methods can benefit from further ordering of matrices and tensors in the preprocessing step to improve data locality.

We present in Table II the breakdown of sequential execution time of Sparta on two instances that are representative of cases for which Sparta takes over an hour. The tensor enron contains 54.20 million nonzeros, while the tensor contraction requires 47.13 billion flops and the resultant tensor has 6.93 billion nonzeros. As we can observe, the preprocessing time is 23.41 seconds and the time for SpGETT is 6383.67 seconds.



(a) #threads = 16
(b) #threads = 32
(c) #threads = 48
(d) #threads = 64
(e) #threads = 80
(f) #threads = 96

Fig. 4: The run time of SB-TC and SB-Hmat normalized by that of Sparta for parallel execution with {16, 32, 48, 64, 80, 96} threads.

Similarly, chicago_crime contains 5.33 million nonzeros, while the tensor contraction requires 72.81 billion flops and the resultant tensor has 16.93 billion nonzeros. For this input instance, the preprocessing time of Sparta is 2.18 second and the SpGETT time is 20424.06 second. We also ran SB-TC on the two instances in order to demonstrate the relative performance of Sparta and SB-TC in extreme settings. On enron with contraction modes {1,3}, SB-TC completed in 4644.81 seconds, and on chicago_crime with contraction modes {2,3}, SB-TC completed in 16694.36 second. Here again, we observe that SB-TC is faster than Sparta.

Next, we present the performance of parallel execution of SB-Hmat, SB-TC and Sparta. Figure 4 shows the relative performance of parallel execution of SB-TC and SB-Hmat with respect to Sparta on all 85 input instances, for different thread counts. For each of the plots, y-axis shows the run time of SB-TC, SB-Hmat and Sparta normalized by the run time of Sparta for all instances. On the x-axis, the instances are arranged in nondecreasing order by the ratio of the run time of SB-TC to Sparta from left to right. The geometric mean of the ratio of SB-TC to Sparta (dashed line) and the geometric mean of the ratio of SB-Hmat to Sparta (dot-dashed line) are also shown in the figure. Lower values on the y-axis for SB-TC and SB-Hmat depict better performance with respect to Sparta. We observe from the figure that for parallel execution, SB-TC is consistently faster than Sparta for all instances, for all thread counts. With 16, 32, 48, 64, 80 and 96 threads (Figure 4a–4f), SB-TC is on average 21%, 22%, 22%, 23%, 20%, 21% faster, respectively than Sparta across all input instances. Over all instances across all thread counts, SB-TC is on average 21.48% faster than Sparta. SB-TC demonstrates best performance w.r.t. Sparta for 64 threads (Figure 4d). Furthermore, we observe that across all thread counts, SB-Hmat is on average faster than Sparta on 53 input instances out of the 85. SB-Hmat is on average 2.67% faster than Sparta across all input instances for all thread counts. We see from these figures that SB-TC is consistently faster than the other two approaches in all thread counts.

Last, we study the scalability of parallel execution of SB-TC, SB-Hmat and Sparta with respect to their respective sequential versions. Figure 5 presents the parallel scaling of SB-TC, SB-Hmat and Sparta over all the instances. We observe from the plot that all the three methods show similar parallel scaling, while the absolute run time of SB-TC is on average less than that of Sparta and SB-Hmat (combining the inference from Figure 3 and Figure 5), since SB-TC is faster than Sparta and SB-Hmat in sequential execution. This is because the total time is in general dominated by the multiplication step. Furthermore, the performance of multiplication is limited by the memory access latency, due to limited data locality particularly in the output tensor. Note that we run our experiments on a machine having 4 NUMA nodes having 24 cores each. The overall performance as well as the scalability is also affected by NUMA effects. None of the methods SB-Hmat, SB-TC, or Sparta implements optimizations for NUMA systems. To give further insight we note that for the sequential execution, across all input instances, Sparta takes 163.06 seconds on average, SB-Hmat takes 151.64 seconds on average and SB-TC takes 122.94 seconds on average. For parallel execution with 96 threads, across all input instances, Sparta takes 5.16 seconds, SB-Hmat takes 5.19 seconds and SB-TC takes 4.13 seconds on average.

VII. Conclusion

We have investigated two approaches to performing parallel sparse tensor sparse tensor multiplication (SpGETT) on shared

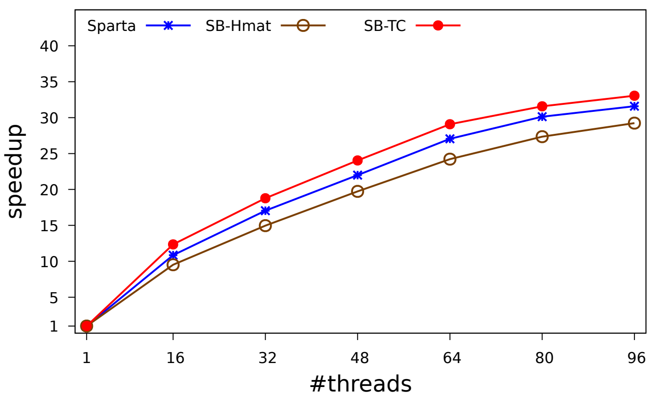


Fig. 5: Overall scalability of SB-TC, SB-Hmat and Sparta on all the instances. Speedup of SB-TC, SB-Hmat and Sparta is with respect to their respective sequential run time.

memory systems: i) SpGETT via reduction to SpGEMM, and ii) SpGETT natively on the input tensors. We have identified that a hashing scheme is needed in both approaches for efficiency and proposed SBhash, a parallel dynamic hashing method. We have then used this hashing method to implement SB-Hmat, a state-of-the-art method to compute SpGETT via reduction to SpGEMM and have showed by means of experiments that SB-Hmat is more efficient than a more readily available approach SB-Smat. We have also used SBhash to propose SB-TC, an efficient parallel hashing-based method, to perform SpGETT natively on the input tensors. We demonstrate the efficacy of SB-Hmat and SB-TC through a systematic evaluation, and by comparison with the existing state-of-the-art parallel method for SpGETT. Overall, SB-TC obtains about 20% better run time than the current state-of-the art methods on a machine with 96 cores.

The methods SB-Hmat and SB-TC can benefit from a preprocessing, in which matrices or tensors are reordered for better data locality. A suitable reordering should have a low overhead, and should be amenable to parallelization. What will be a suitable algorithm for such a reordering?

Acknowledgments

This work was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program “Investissements d'Avenir” (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

References

[1] P. N. Q. Anh, R. Fan, and Y. Wen, “Balanced hashing and efficient gpu sparse general matrix-matrix multiplication,” in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926273

[2] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, “Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication,” *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016. [Online]. Available: https://doi.org/10.1137/15M104253X

[3] B. W. Bader, T. G. Kolda *et al.*, “Matlab tensor toolbox version 3.6,” Available online https://www.tensortoolbox.org/, September 2023.

[4] B. W. Bader and T. G. Kolda, “Efficient MATLAB computations with sparse and factored tensors,” *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, December 2007.

[5] M. Bansal, O. Hsu, K. Olukotun, and F. Kjolstad, “Mosaic: An Interoperable Compiler for Tensor Algebra,” *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591236

[6] N. Bell, S. Dalton, and L. N. Olson, “Exposing fine-grained parallelism in algebraic multigrid methods,” *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012. [Online]. Available: https://doi.org/10.1137/110838844

[7] J. Bertrand, F. Dufossé, S. Singh, and B. Uçar, “Algorithms and data structures for hyperedge queries,” *ACM J. Exp. Algorithmics*, vol. 27, dec 2022. [Online]. Available: https://doi.org/10.1145/3568421

[8] A. Cichocki, D. P. Mandic, L. De Lathauwer, G. Zhou, Q. Zhao, C. Caiafa, and A.-H. Phan, “Tensor decompositions for signal processing applications: From two-way to multiway component analysis,” *IEEE Signal Processing Magazine*, vol. 32, no. 2, pp. 145–163, March 2015.

[9] E. Cohen, “Structure prediction and computation of sparse matrix products,” *Journal of Combinatorial Optimization*, vol. 2, no. 4, pp. 307–332, Dec 1998. [Online]. Available: https://doi.org/10.1023/A:1009716300509

[10] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718881

[11] M. Deveci, C. Trott, and S. Rajamanickam, “Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures,” *Parallel Computing*, vol. 78, pp. 33–46, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819118301923

[12] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, “Dynamic perfect hashing: Upper and lower bounds,” *SIAM Journal on Computing*, vol. 23, no. 4, pp. 738–761, 1994.

[13] E. Esposito, T. Mueller Graf, and S. Vigna, “RecSplit: Minimal perfect hashing via recursive splitting,” in *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Philadelphia, PA: SIAM, 2020, pp. 175–185.

[14] M. Fishman, S. R. White, and E. M. Stoudenmire, “The itensor software library for tensor network calculations,” *CoRR*, vol. abs/2007.14822, 2020. [Online]. Available: https://arxiv.org/abs/2007.14822

[15] M. L. Fredman, J. Komlós, and E. Szemerédi, “Storing a sparse table with $O(1)$ worst case access time,” *J. ACM*, vol. 31, no. 3, pp. 538–544, 1984.

[16] J. R. Gilbert, C. Moler, and R. Schreiber, “Sparse matrices in MATLAB: Design and implementation,” *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992. [Online]. Available: https://doi.org/10.1137/0613024

[17] F. G. Gustavson, “Two fast algorithms for sparse matrices: Multiplication and permuted transposition,” *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, sep 1978. [Online]. Available: https://doi.org/10.1145/355791.355796

[18] A. P. Harrison and D. Joseph, “High performance rearrangement and multiplication routines for sparse tensor arithmetic,” *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. C258–C281, 2018. [Online]. Available: https://doi.org/10.1137/17M1115873

[19] K. Z. Ibrahim, S. W. Williams, E. Epifanovsky, and A. I. Krylov, “Analysis and tuning of libtensor framework on multicore architectures,” in *2014 21st International Conference on High Performance Computing (HiPC)*, 2014, pp. 1–10.

[20] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, “The tensor algebra compiler,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133901

[21] T. G. Kolda and B. W. Bader, “Tensor decompositions and applications,” *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009. [Online]. Available: https://doi.org/10.1137/07070111X

[22] J. Li, J. Sun, and R. Vuduc, “HiCOO: Hierarchical storage of sparse tensors,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '18. New York, NY, USA: ACM, 2018.

[23] A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo, “Fast and Scalable Minimal Perfect Hashing for Massive Key Sets,” in *16th International Symposium on Experimental Algorithms (SEA 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 75. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 25:1–25:16. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7619

[24] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on gpus," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 47–57.

[25] J. Liu, D. Li, R. Gioiosa, and J. Li, "Athena: High-performance sparse tensor contraction sequence on heterogeneous memory," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 190–202. [Online]. Available: https://doi.org/10.1145/3447818.3460355

[26] J. Liu, J. Ren, R. Gioiosa, D. Li, and J. Li, "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*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 318–333. [Online]. Available: https://doi.org/10.1145/3437801.3441581

[27] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 370–381.

[28] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors," *Parallel Computing*, vol. 90, p. 102545, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016781911930136X

[29] I. Nisa, J. Li, A. Sukumaran-Rajam, P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356216

[30] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms — ESA 2001*, F. M. auf der Heide, Ed. Springer Berlin Heidelberg, 2001, pp. 121–133.

[31] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, "Speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 362–375. [Online]. Available: https://doi.org/10.1145/3332466.3374521

[32] G. E. Pibiri and R. Trani, "PTHash: Revisiting FCH minimal perfect hashing," in *44th SIGIR, International Conference on Research and Development in Information Retrieval*. Virtual Event, Canada: ACM, 2021, pp. 1339–1348.

[33] C. Psarras, L. Karlsson, J. Li, and P. Bientinesi, "The landscape of software for tensor computations," 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2103.13756

[34] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[35] S. Singh and B. Uçar, "An efficient parallel implementation of a perfect hashing method for hypergraphs," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE. IEEE CPS, 2022, pp. 265–274.

[36] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis, "FROSTT: The formidable repository of open sparse tensors and tools," Available at http://frostt.io/, 2017.

[37] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA¡sup¿3¡/sup¿ '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833179.2833183

[38] E. Solomonik and T. Hoefler, "Sparse tensor algebra as a parallel programming model," *CoRR*, vol. abs/1512.00066, 2015. [Online]. Available: http://arxiv.org/abs/1512.00066

[39] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3176–3190, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S074373151400104X

[40] P. Springer and P. Bientinesi, "Design of a high-performance GEMM-like tensor–tensor multiplication," *ACM Trans. Math. Softw.*, vol. 44, no. 3, jan 2018. [Online]. Available: https://doi.org/10.1145/3157733

[41] Q. Sun, Y. Liu, H. Yang, M. Dun, Z. Luan, L. Gan, G. Yang, and D. Qian, "Input-aware sparse tensor storage format selection for optimizing mttkrp," *IEEE Transactions on Computers*, vol. 71, no. 08, pp. 1968–1981, aug 2022.

[42] R. Tian, L. Guo, J. Li, B. Ren, and G. Kestor, "A High Performance Sparse Tensor Algebra Compiler in MLIR," in *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2021, pp. 27–38.

[43] C. Uphoff and M. Bader, "Yet another tensor toolbox for discontinuous Galerkin methods and other applications," *ACM Trans. Math. Softw.*, vol. 46, no. 4, oct 2020. [Online]. Available: https://doi.org/10.1145/3406835

[44] T. Zhao, T. Popoola, M. Hall, C. Olschanowsky, and M. Strout, "Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-iteration," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 1, dec 2022. [Online]. Available: https://doi.org/10.1145/3566054