Efficient parallel sparse tensor contraction

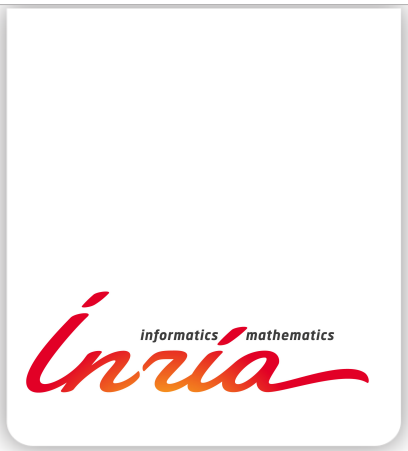
Somesh Singh, Bora Uçar

Efficient parallel sparse tensor contraction

Somesh Singh, Bora Uçar

RESEARCH REPORT
N° 9551
July 2024
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9551--FR+ENG



Efficient parallel sparse tensor contraction

Somesh Singh*, Bora Uçar†

Project-Team ROMA

Research Report n° 9551 — July 2024 — 27 pages

Abstract: We investigate the performance of algorithms for sparse tensor-sparse tensor multiplication (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 by first converting the input tensors into matrices, then invoking high performance variants of SpGEMM, and finally reconverting the resultant matrix into a tensor. Alternatively, one can carry out the scalar operations underlying SpGETT in the realm of tensors without matrix formulation. 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 new fine-tuned hashing method, proposed in this paper. We evaluate the methods on real world tensors, contracting a tensor with itself along various dimensions. Our proposed hashing-based method for SpGETT consistently outperforms the state-of-the-art method, achieving a 25% reduction in sequential execution time on average and a 21% reduction in parallel execution time on average across a variety of input instances.

Key-words: tensor contraction, hashing

* LabEx MILYON and LIP (UMR5668 Université de Lyon - ENS de Lyon - UCBL - CNRS - Inria), 46 allée d'Italie, ENS Lyon, Lyon F-69364, France.

† CNRS and LIP (UMR5668 Université de Lyon - ENS de Lyon - UCBL - CNRS - Inria), 46 allée d'Italie, ENS Lyon, Lyon F-69364, France.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Contraction efficace et parallèle des tenseurs creux

Résumé : Nous étudions les performances des algorithmes de multiplication de tenseurs creux à creux (SpGETT). Cette opération, également appelée contraction, est un analogue d'ordre supérieur à la multiplication de matrices creuses (SpGEMM). En conséquence, SpGETT peut être réalisé en convertissant d'abord les tenseurs d'entrée en matrices, puis en invoquant des variantes hautes performances de SpGEMM, et enfin en reconvertissant la matrice résultante en tenseur. Alternativement, il est possible de réaliser les opérations scalaires sous-jacentes à SpGETT directement dans le domaine des tenseurs, sans formulation matricielle. Nous discutons des éléments constitutifs des deux approches et proposons une méthode basée sur le hachage pour éviter les opérations coûteuses de recherche ou de redirection. Nous présentons les résultats de performances en utilisant les approches actuelles basées sur SpGEMM, les approches SpGETT existantes, ainsi qu'une approche SpGETT soigneusement mise en œuvre avec une nouvelle méthode de hachage améliorée, proposée dans cet article. Nous évaluons ces méthodes sur des tenseurs réels, en contractant un tenseur avec lui-même selon différentes dimensions. La méthode basée sur le hachage proposée pour SpGETT surpasse systématiquement l'état de l'art, atteignant une réduction moyenne de 25% du temps d'exécution séquentielle et une réduction moyenne de 21% du temps d'exécution parallèle sur une variété d'instances d'entrée.

Mots-clés : contraction des tenseurs, hachage

Contents

1 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 [4], 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 a 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 the 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 [3, 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 contraction 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 3, where one is based on sorting, and 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 4 and propose a hashing scheme (Section 5) 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 lookup time. When all elements to be hashed are unique and known beforehand, static hashing approaches with expected linear time construction, linear memory overhead, and worst case constant time lookup are 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 key 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 applicable, we introduce a parallel dynamic hashing method in Section 5 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 2, we investigate the SpGEMM-based approach to sparse tensor contraction in Section 3. We then describe in Section 4 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 5, as it is of independent interest. We then compare the proposed SpGETT algorithm with two current state of the art SpGETT implementations in Section 6, one using SpGEMM routines and the other natively working in tensors. Section 7 concludes the paper.

2 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 index into $\boldsymbol{\mathcal{A}}$. 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 $\boldsymbol{\mathcal{A}}$ 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 $\boldsymbol{\mathcal{A}}$ 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 this 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 convenient, it is not necessary to have the first p modes of $\boldsymbol{\mathcal{A}}$ in S_R. 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 specified contraction modes, can be formulated as 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

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

where 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 $c_{ij,kl}$ can 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 [36] 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 easiest one for a user.

2.1 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.

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

```
1 parfor nonempty A_{i,:} do
2     for nonzero a_{ik} in A_{i,:} do
3         for nonzero b_{kj} in B_{k,:} do
4             v ← a_{ik} · b_{kj}
5             if c_{ij} ∈ C_{i,:} then
6                 c_{ij} ← c_{ij} + v
7             else
8                 insert c_{ij} in C_{i,:}
9                 c_{ij} ← v
```

As can be seen in Algorithm 1, Gustavson's algorithm proceeds row-wise on matrix $\mathbf{A}$. For each nonzero a_{ik} in a 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 a 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 lookup for a scalar multiplication to be added to an existing entry in $\mathbf{C}$ (Line 6 of Algorithm1), or creating a new entry in $\mathbf{C}$ (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].

2.2 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] and 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. The Sparse Polyhedral Framework [44] generates code for sparse tensor contractions. The Cyclops Tensor Framework [39] enables automatic parallelization of sparse tensor computations, including sparse tensor contractions, expressed algebraically.

The Tensor Toolbox [4] 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 the order-agnostic coordinate (COO) format. It uses 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 the Intel Optane DC Persistent Memory Module (PMM). We compare our proposed methods against Sparta on homogeneous DRAM memory in Section 6. Our tensor contraction method uses a novel dynamic hashing method for representing the input tensors as well as the sparse accumulator, which complements 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.

2.3 Related work on hashing

As discussed before, hashing methods are 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 has been constructed, it is only queried for the existence of items and retrieving them. In dynamic hashing, the set of items

is 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 lookups. Such hashing methods which enable worst case constant time lookups are referred to as *perfect hashing* methods. 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 lookup 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 dynamic hashing method. It uses a set of ℓ hash functions. We describe the well-known case of $\ell = 2$, which has been extensively studied and is well-understood. For each arriving item, two hash values are computed, giving the two possible locations for the item. The item is placed in one of the locations. For lookup, one has to check then only two locations. Assuming that the hash functions take $O(1)$-time to compute, the lookups are thus asymptotically optimal. Insertion of an item in Cuckoo hashing is $O(1)$-time in the average case. If both the possible locations for an incoming item x are already occupied then one of the locations is picked randomly, and the item y at that location is replaced with x, and y is moved to its alternate location. If the other location for y is occupied, we replace the item z there with y and try to find a position for the displaced element z in the same fashion. This is continued until a free location is found or a rehash is necessary (in the case of cycles). Rehashing takes $O(n)$ in the average case, where n is the number of items. For the above complexity bounds to hold, one needs to maintain the invariant that no more than half of the total locations are occupied.

We introduce a dynamic perfect hashing method supporting the insertion of items one by one, and performing worst-case constant time lookups on the updated hash data structure. This is imperative to support 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, our proposed tensor contraction method is the first to use a perfect hashing method for sparse tensor contractions.

3 SpGETT via reduction to SpGEMM

Consider the tensor contraction operation $\mathcal{C}_{e_A,e_B} = \mathcal{A}_{e_A,c_A}\mathcal{B}_{c_B,e_B}$, with external modes e_A and e_B corresponding to, respectively, the external modes of $\mathcal{A}$ and $\mathcal{B}$. The contraction modes c_A and c_B of $\mathcal{A}$ and $\mathcal{B}$ 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 converting the resultant matrix to a 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 $\mathbf{A} = \mathcal{A}_{e_A \times c_A}$ and $\mathbf{B} = \mathcal{B}_{c_B \times e_B}$. As discussed before, many tuples of indices in external or contraction modes may be zero in $\mathcal{A}$ or $\mathcal{B}$. Typically, the number of empty rows or columns in $\mathbf{A}$ and $\mathbf{B}$ can be much more than the number of nonzeros in the matrices, if all tuples in external or contraction modes are created as rows or columns in these matrices. As this sparsity will cause slowdowns in SpGEMM software, either special SpGEMM libraries should be developed [18] or $\mathbf{A}$ and $\mathbf{B}$ should contain only nonempty rows and columns. We discuss the latter approach so that one can invoke any high performance SpGEMM library.

The nonempty subtensors $\mathcal{A}_{e_A,:}$ define the nonzero columns of $\mathbf{A}$, and the nonempty subtensors $\mathcal{B}_{:,e_B}$ 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 nonempty subtensors $\boldsymbol{\mathcal{A}}_{:,c_A}$, and the columns of $\mathbf{B}$ should correspond to nonempty subtensors $\boldsymbol{\mathcal{B}}_{c_B,:}$. After the multiplication, the resultant matrix $\mathbf{C}$ should be converted to the tensor $\boldsymbol{\mathcal{C}}$ by mapping each nonempty row index i of $\mathbf{C}$ to the corresponding $|e_A|$-tuple $\mathbf{i}$, and each nonempty 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 *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 nonempty subtensor, and also need the inverse of this map for the tensorization of $\mathbf{C}$. A similar discussion holds for $|e_B|$-tuples defining nonempty subtensors $\boldsymbol{\mathcal{B}}_{:,e_B}$ and the columns of $\mathbf{C}$.

Typically, sorting or hashing is 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.

3.1 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, by increasing index, into L using contraction indices as key, while keeping a reference to the original nonzero as auxiliary data
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 A and the row ids in the matrix 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 Steps 4 and 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.

3.2 SB-Hmat: Hashing for SpGETT-via-SpGEMM

This method uses hashing for effecting consistent matricization of $\mathcal{A}$ and $\mathcal{B}$ 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 $\mathcal{A}$ and $\mathcal{B}$ together into a single hash table. We do this by inserting all contraction indices of $\mathcal{A}$ and $\mathcal{B}$ 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 $\mathcal{A}$ and $\mathcal{B}$ 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 $\mathcal{A}$ and $\mathcal{B}$. For each nonzero in $\mathcal{A}$ and $\mathcal{B}$, 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 $\mathcal{A}$ and $\mathcal{B}$ separately. We create separate hash tables for $\mathcal{A}$ and $\mathcal{B}$ using the external indices of the nonzeros as the hash key. As in the sorting-based algorithm, we create $\mathbf{A}$ and $\mathbf{B}$ in the coordinate format, which we call $\mathrm{COO_A}$ and $\mathrm{COO_B}$ respectively. Again, we maintain a global counter to assign consecutive ids to the distinct external indices, $\mathbf{e_A}$, of $\mathcal{A}$. We first scan the nonzeros of $\mathcal{A}$. For each nonzero in $\mathcal{A}$ 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 as row-id in $\mathrm{COO_A}$. Furthermore, we write the id of contraction indices of the nonzero, $\mathbf{c_A}$, as the column-id in $\mathrm{COO_A}$ 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 $\mathcal{A}$, for assigning ids to the external indices of $\mathcal{B}$ and for creating the $\mathbf{B}$ by populating $\mathrm{COO_B}$ 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 $\mathcal{C}$.

4 SB-TC: SpGETT natively on input tensors

We present SB-TC to carry out parallel SpGETT $\mathcal{C}_{e_A,e_B} = \mathcal{A}_{e_A,c_A}\mathcal{B}_{c_B,e_B}$ natively on tensors. Recall that for this tensor contraction to be feasible, the contraction modes c_A and c_B of $\mathcal{A}$ and $\mathcal{B}$, respectively, must be the same size and ordered consistently. Recall also that for nonzeros $a_{\mathbf{e_A},\mathbf{c_A}}$ and $b_{\mathbf{c_B},\mathbf{e_B}}$, the indices $\mathbf{c_A}$ and $\mathbf{c_B}$ refer to the indices in the contraction modes c_A of $\mathcal{A}$ and c_B of $\mathcal{B}$, respectively. SB-TC 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.

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 A using the indices in the e_A modes as the key
2  Create a hash data structure for B using the indices in the c_B modes as the key
3  parfor nonempty e_A subtensor A_{e_A,:} do
4      Initialize SPA                          // Hash for the nonzeros in C_{e_A,:}
5      for nonzero a_{e_A,c_A} in A_{e_A,:} do
6          for nonzero b_{c_B,e_B} in B_{c_B,:} do
7              v ← a_{e_A,c_A} · b_{c_B,e_B}
8              if SPA.lookup(e_B) = True then
9                  add v to SPA(e_B)
10             else
11                 SPA.insert(e_B, v)
12     for each tuple e_B in SPA with value v do                    // v ≠ 0
13         set c_{e_A,e_B} = v
```

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 from each other, in parallel, indicated by the parallel **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 an appropriate subtensor $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$. in $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$. 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.

A parallel **parfor** loop goes over all the nonempty subtensors $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ of $\boldsymbol{\mathcal{A}}$ (Line 3 in Algorithm 3). In order to populate a subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ of the output tensor $\boldsymbol{\mathcal{C}}$, the algorithm loops over all the nonzeros in a subtensor $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ of $\boldsymbol{\mathcal{A}}$ (Line 5 in Algorithm 3). Doing so efficiently requires all nonzeros of $\boldsymbol{\mathcal{A}}$ having 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 (Line 1 in Algorithm 3). In this hash data structure, each unique external index of $\mathbf{e_A}$ of $\boldsymbol{\mathcal{A}}$ maps to a distinct location. Furthermore, a location in the hash data structure points to a 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 $\mathcal{H}_{\mathcal{A}}$ corresponds to a row of the matrix $\mathbf{A}$ without explicit conversion. Looping over the nonzeros in $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ reduces to going over the contiguous array pointed to by the location of $\mathbf{e_A}$ in $\mathcal{H}_{\mathcal{A}}$.

Furthermore, for each nonzero $a_{\mathbf{e_A},\mathbf{c_A}}$ in the subtensor $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ of $\boldsymbol{\mathcal{A}}$, the algorithm goes over all the nonzeros in the corresponding subtensor $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$ of $\boldsymbol{\mathcal{B}}$ (Line 6 in Algorithm 3), and multiplies $a_{\mathbf{e_A},\mathbf{c_A}}$ with every nonzero $b_{\mathbf{c_B},\mathbf{e_B}}$ in $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$ (Line 7 in Algorithm 3). Recall that for every nonzero $a_{\mathbf{e_A},\mathbf{c_A}}$ in $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$, its contraction indices $\mathbf{c_A}$ must be equal to the contraction indices $\mathbf{c_B}$ in the nonzeros in $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$. To facilitate accessing the nonzeros in a subtensor $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$ of $\boldsymbol{\mathcal{B}}$ efficiently, it is required that all nonzeros in $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$ 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 (Line 2 in Algorithm 3). 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 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 external modes, e_B. Such a location in $\mathcal{H}_\mathcal{B}$ corresponds to a row of the matrix $\mathbf{B}$ without explicit conversion. Looping over the nonzeros in $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$ reduces to going over the contiguous array pointed to by the location of $\mathbf{c_B}$ in $\mathcal{H}_\mathcal{B}$.

As the algorithm proceeds, in order to assemble the product of pairs of nonzeros, $a_{\mathbf{e_A},\mathbf{c_A}}$ and $b_{\mathbf{c_B},\mathbf{e_B}}$ in the output subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$, we need to perform *lookup* (Line 8) and *insert* (Line 11) operations. In order to construct the subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ efficiently, we accumulate the results in a sparse accumulator (SPA) (Line 4, Algorithm 3). The efficiency of the algorithm is contingent on the efficiency of the lookup and insert operations. Therefore, we maintain the SPA as a dynamic hash structure which supports constant time lookup and efficient insertion, while also being space efficient. SPA uses the indices of the nonzeros of $\boldsymbol{\mathcal{B}}$ in the external modes as the hash key. Once the entire output subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ is ready in the SPA, the contents of the SPA are written out to $\boldsymbol{\mathcal{C}}$ 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 instructions.

4.1 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.

4.1.1 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 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 number of nonzeros 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. 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.

4.1.2 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 a 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. We count flops for each nonempty subtensor $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ of $\boldsymbol{\mathcal{A}}$. For determining the flops of a subtensor $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$, we sweep over the nonzeros in the subtensor and sum the number of nonzeros in the appropriate subtensor $\boldsymbol{\mathcal{B}}_{\mathbf{c_B},:}$. After

collating the flops of all the subtensors $\boldsymbol{\mathcal{A}}_{\mathbf{e_A},:}$ in an array, we perform a prefix sum. We then determine the starting subtensor (and the number of subtensors implicitly) of $\boldsymbol{\mathcal{A}}$ for each of the threads such that each thread is assigned a close to average number of flops per thread, as outlined in the original scheme.

4.2 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.

4.2.1 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. This optimization is performed on the fly, as the algorithm proceeds, and does not require any preprocessing.

4.2.2 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 its product with the nonzeros in subtensor $\boldsymbol{\mathcal{B}}_{c_B,:}$ can be written directly to the output tensor at its unique position in the corresponding subtensor $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ This is so as there are no prior entries in the sparse accumulator and each of the products all have unique positions in the sparse accumulator. 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 $\boldsymbol{\mathcal{A}}$. This requires performing preprocessing, which takes time $O(nnz(\boldsymbol{\mathcal{A}}))$.

5 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.

5.1 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 $n_h < d$, 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

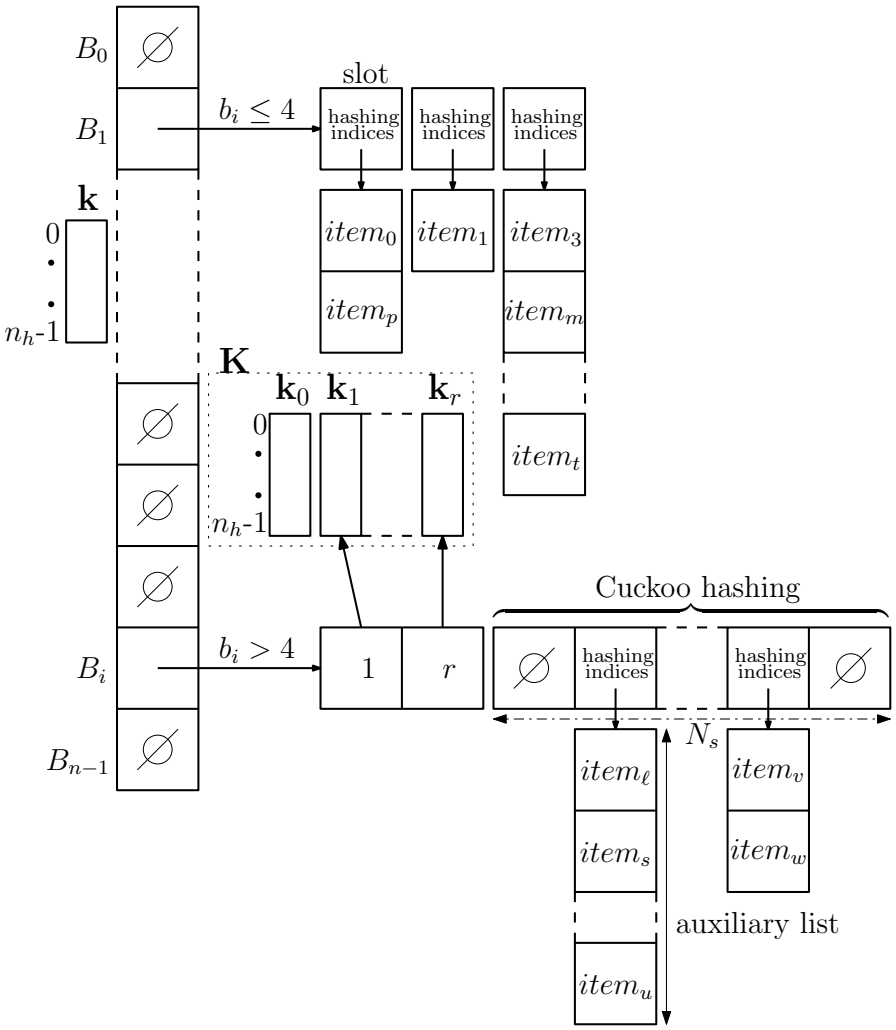


Figure 1: SBhash data structure. A n_h-tuple $\mathbf{k}$ is used as key for the first level of hashing. Two types of buckets are shown, a bucket with $b_i \leq 4$ and a bucket with $b_i > 4$. For a bucket with $b_i > 4$, indices of two n_h-tuples from $\mathbf{k}_0$ to $\mathbf{k}_r$ in $\mathbf{K}$ are stored. Empty buckets and slots are denoted by a $\varnothing$. In SB-TC, an item is $\{\langle \text{indices} \rangle, \text{val}\}$. In SB-Hmat, an item is the *id* of nonzero in coordinate representation of tensor.

that all map to that 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 the indices of $\mathbf{x}$ in the hashing dimensions H. This function has been used before [7, 15].

Furthermore, consider that to a bucket B_i, b_i distinct n_h-tuples are mapped as a result of the first-level hashing. If $b_i = 0$, then nothing is to be stored at that bucket. For nonempty buckets having b_i up to 4, we maintain the slots in the bucket as a contiguous array which store the distinct n_h-tuples mapped to the bucket along with their auxiliary list. As shown in Figure 1, we maintain a pre-populated set of random n_h-tuples, $\mathbf{K}$, of size 32. For buckets having $b_i > 4$, for mapping the n_h-tuples to slots we perform a second level of hashing. We use a variant of the standard Cuckoo hashing [30] adapted to the needs of the SpGETT operation. At each such bucket, we store ids of two random keys in $\mathbf{K}$, which we use for hashing the items mapped to the bucket. The number of slots of such a bucket is maintained at the smallest power of 2 greater than twice b_i. 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 4, the lookup operations need to be performed one by one during the contraction operation. The insertions on the other hand can either be one by one, or 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 is of independent interest, we do not discuss its generality.

5.2 Cuckoo hashing specifications

We use 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 on one side, and N_s vertices on the other. Each item chooses ℓ slots by applying the ℓ hash functions. A perfect hashing will be obtained if we can perfectly match each item to a unique slot. While the standard Cuckoo hashing uses a random walk for insertions, we carefully implement a deterministic method for insertion, which is described below. We have $\ell = 2$.

5.2.1 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 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 $N_s \geq 2b_i$. For an item $\mathbf{x}$ in bucket B_i, the evaluation of the two hash functions, $h_1(\mathbf{x})$ and $h_2(\mathbf{x})$ gives its two possible locations in the bucket.

The standard random walk-based insertion method for Cuckoo hashing can be used in our setting with two possible locations per item. With this method, if both the possible locations of an item are occupied, we randomly pick one of the two locations, and displace the item at that location and move it to its alternate location choice. We continue this until a free location is found or a rehash is necessary. This approach has the advantage that it does not use any extra storage; however it can visit certain items multiple times.

As stated above, insertion in Cuckoo hashing can be reduced to finding a matching on a bipartite graph, where the items are vertices on the left and their possible locations are vertices on the right. There is an edge from an item to each of its possible locations. Every item, on the left, must be matched to exactly one location, on the right. Our deterministic approach for insertion efficiently solves this underlying matching problem in order to assign a unique location to every item in a bucket. Since for an item $\mathbf{x}$, its possible locations can be determined by evaluating the hash functions $h_1(\mathbf{x})$ and $h_2(\mathbf{x})$, so we do not store the edges of the underlying graph explicitly but evaluate the hash functions on the fly on demand.

For inserting a new item $\mathbf{x}$, its two possible locations are computed. If one of the locations is free, the item is assigned to that location. If both the possible locations, $h_1(\mathbf{x})$ and $h_2(\mathbf{x})$ are occupied, then the location $h_1(\mathbf{x})$ is picked to displace the item at that location. Note that since every item has only two possible location choices, so there are no further choices to be made as we walk through the underlying graph to find an augmenting path to augment the current matching. If we fail to find an augmenting path, then the location $h_2(\mathbf{x})$ is picked to displace the item at that location and find the augmenting path. The traversal of the graph can be performed using either breadth-first or depth-first traversal since both are equivalent in this scenario. In order to make the algorithm efficient and to avoid additional overheads, we maintain a single *visited* array of size number of slots, N_s only once across different BFS/DFS traversals for augmentations, and use it multiple times. We only access the locations that are required during the current traversal. We use a marking scheme so as to tell apart the visited/unvisited flags at an array index, across different traversals. Furthermore, we also need to maintain the augmenting path for every augmentation. We again maintain a single array of size number of items, n_I, to store the items in the augmenting path, and reuse it across different augmentations; we do not reinitialize the array. This is akin to a stack in a DFS traversal. Thus, our scheme avoids additional computational overhead in exchange for additional storage for bookkeeping, which is $O(N_s + n_I)$. We only spend the time in finding the augmenting path. Thus, the worst-case complexity of finding a location for a new item is proportional to the sum of the cost of traversing the paths corresponding to its two possible location choices, since $\ell = 2$.

If we do not find a free location for an item, then we need to rehash the items in the bucket. In order to perform a rehash, we update the hash function. To update the hash function, we first update the number of slots to be the smallest power of 2 greater than twice the number of items, in order to create sufficient slots to perform 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 in $\mathbf{K}$. The items are then inserted one by one as discussed above.

5.2.2 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 with the given item.

5.3 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 the items—assigning items to threads and every thread handling an 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 5.1. 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.

5.4 Memory requirement of SBhash in SpGETT-via-SpGEMM and SB-TC

As discussed in Sections 3.2 and 4, SBhash is used in SpGETT-via-SpGEMM and SB-TC. We describe the memory requirements of SBhash in both the methods.

In SB-Hmat, we use SBhash for assigning ids to distinct contraction indices. For this, SBhash uses $(nnz(\boldsymbol{\mathcal{A}}) + nnz(\boldsymbol{\mathcal{B}}))$ buckets for the first level hashing. The number of buckets is an upper bound on the number of unique contraction indices that are inserted into the hash data structure. At a bucket, the total number of locations, used for Cuckoo hashing, are upper bounded by twice the number of occupied locations. Thus, the total number of locations, across all buckets is upper bounded by twice the total number of occupied locations, which is much less than the number of nonzeros, which we call N_{TOS}. Furthermore, at each occupied location, an auxiliary list stores an integer id of the nonzeros in the coordinate format of tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$. The total size of all the auxiliary lists is equal to the total number of items inserted in the hash data structure, which is equal to $(nnz(\boldsymbol{\mathcal{A}}) + nnz(\boldsymbol{\mathcal{B}}))$. Thus, the total space requirement is upper bounded by $2 \times (nnz(\boldsymbol{\mathcal{A}}) + nnz(\boldsymbol{\mathcal{B}})) + N_{TOS}$ integers. Next, in SB-Hmat, we also use SBhash for assigning ids to the distinct external indices of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ separately. Following the same analysis as above, the total space requirements of the hash data structures for the external indices of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are upper bounded by $2 \times nnz(\boldsymbol{\mathcal{A}}) + N_{TOS}$ integers, and by $2 \times nnz(\boldsymbol{\mathcal{B}}) + N_{TOS}$ integers respectively.

For SB-TC, we use SBhash to create hash data structures $\mathcal{H}_{\mathcal{A}}$ and $\mathcal{H}_{\mathcal{B}}$ for tensors $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, respectively. $\mathcal{H}_{\mathcal{A}}$ uses $nnz(\boldsymbol{\mathcal{A}})$ buckets for the first level hashing. The number of buckets is an upper bound on the number of distinct external indices of $\boldsymbol{\mathcal{A}}$. At a bucket, the total number of locations used for Cuckoo hashing are upper bounded by twice the number of occupied locations. Thus, the total number of locations across all buckets is upper bounded by twice the total number N_{TOS} of occupied locations. Furthermore, in order to improve locality, at each occupied location the auxiliary list stores the indices and the values of the nonzeros of $\boldsymbol{\mathcal{A}}$. The total size of all the auxiliary lists is equal to the total number of items inserted in the hash data structure, which is equal to $nnz(\boldsymbol{\mathcal{A}})$. Therefore, the total space requirement of all auxiliary lists combined is $d \times nnz(\boldsymbol{\mathcal{A}})$ integers and $nnz(\boldsymbol{\mathcal{A}})$ double-type nonzero values. Putting it all together, the total space requirement of $\mathcal{H}_{\mathcal{A}}$ is upper bounded by $(d+1) \times nnz(\boldsymbol{\mathcal{A}}) + N_{TOS}$ integers and $nnz(\boldsymbol{\mathcal{A}})$ double-type values. Performing a similar analysis for $\mathcal{H}_{\mathcal{B}}$, the total space requirement of $\mathcal{H}_{\mathcal{B}}$ is upper bounded by $(d+1) \times nnz(\boldsymbol{\mathcal{B}}) + N_{TOS}$ integers and $nnz(\boldsymbol{\mathcal{B}})$ double-type values.

Furthermore, we also use SBhash for maintaining the sparse accumulator per subtensor, $\boldsymbol{\mathcal{C}}_{\mathbf{e_A},:}$ of $\boldsymbol{\mathcal{C}}$. For this, the number of buckets in SBhash, for the first level hashing, is set to the estimated

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

number of nonzeros $\widetilde{nnz}(\mathcal{C}_{\mathbf{e_A},:})$, which is an upper bound on the number of nonzeros in $\mathcal{C}_{\mathbf{e_A},:}$. At a bucket, the total number of locations, used for Cuckoo hashing, are upper bounded by twice the number of occupied locations. Thus, the total number of locations, across all buckets is upper bounded by twice the total number of occupied locations, N_{TOS}. In order to improve locality, at each occupied location the auxiliary list stores the external indices $\mathbf{e_C}$ (= $\mathbf{e_B}$) of the nonzero along with the value of the nonzero. Thus, the total space requirement of all the auxiliary lists combined is $|e_B| \times nnz(\mathcal{C}_{\mathbf{e_A},:})$ integers and $nnz(\mathcal{C}_{\mathbf{e_A},:})$ double-type nonzero values. Putting it together, the total space requirement is upper bounded by $|e_B| \times nnz(\mathcal{C}_{\mathbf{e_A},:}) + \widetilde{nnz}(\mathcal{C}_{\mathbf{e_A},:}) + N_{TOS}$ integers and $nnz(\mathcal{C}_{\mathbf{e_A},:})$ double-type values.

6 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 [37]. Table 1 summarizes the characteristics of the real-life tensors in our test-suite. All our codes are available at `https://github.com/ssomesh/parTC`.

We perform a comparative study of the performance of different methods for SpGETT: SB-Smat, SB-Hmat, SB-TC, 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. 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 $|e_A| \times |e_A|$.

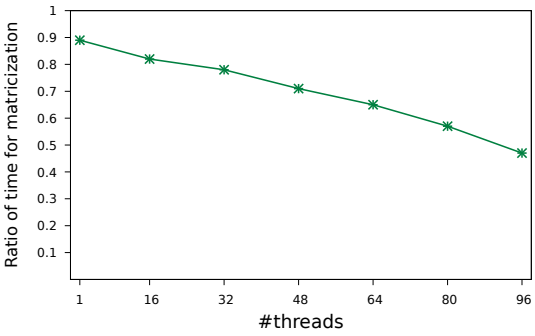


Figure 2: Geometric mean of the ratio of the time for matricization in SB-Hmat to that in SB-Smat, across all 85 input instances, with {1, 16, 32, 48, 64, 80, 96} threads. Lower value on the y-axis depicts better relative performance of matricization in SB-Hmat.

A d-dimensional input tensor can be contracted with itself along any combination of k modes, for $1 \leq k < d$. Thus, there are $\sum_{k=1}^{d-1} \binom{d}{k} = 2^d - 2$ distinct 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 1, 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 1 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 6.1. We then study the performance of the better of the two methods, SB-TC and Sparta in sequential and parallel settings in Section 6.2.

6.1 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 methods. We use CXS-parse [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 the Sparta library [26].

Figure 2 compares the performance of matricization in SB-Hmat and SB-Smat, across all input instances for different numbers 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,

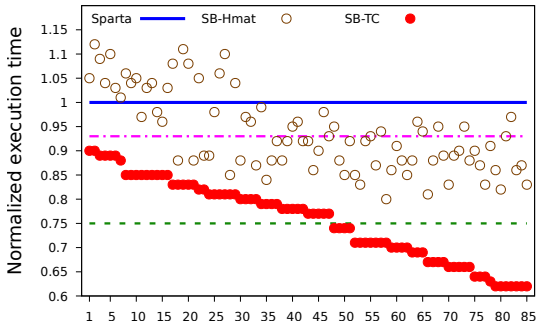


Figure 3: The run time of SB-Hmat and SB-TC normalized by that of Sparta for sequential execution.

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 s, while that in sequential SB-Hmat is 34.89 s. Furthermore, across all input instances, the geometric mean of the time for matricization in parallel SB-Smat with 96 threads is 5.87 s and that in parallel SB-Hmat with 96 threads is 2.70 s. Thus, we conclude that SB-Hmat is more suitable for SpGETT via SpGEMM.

6.2 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. All methods can benefit from ordering of matrices and tensors in the preprocessing step to improve data locality.

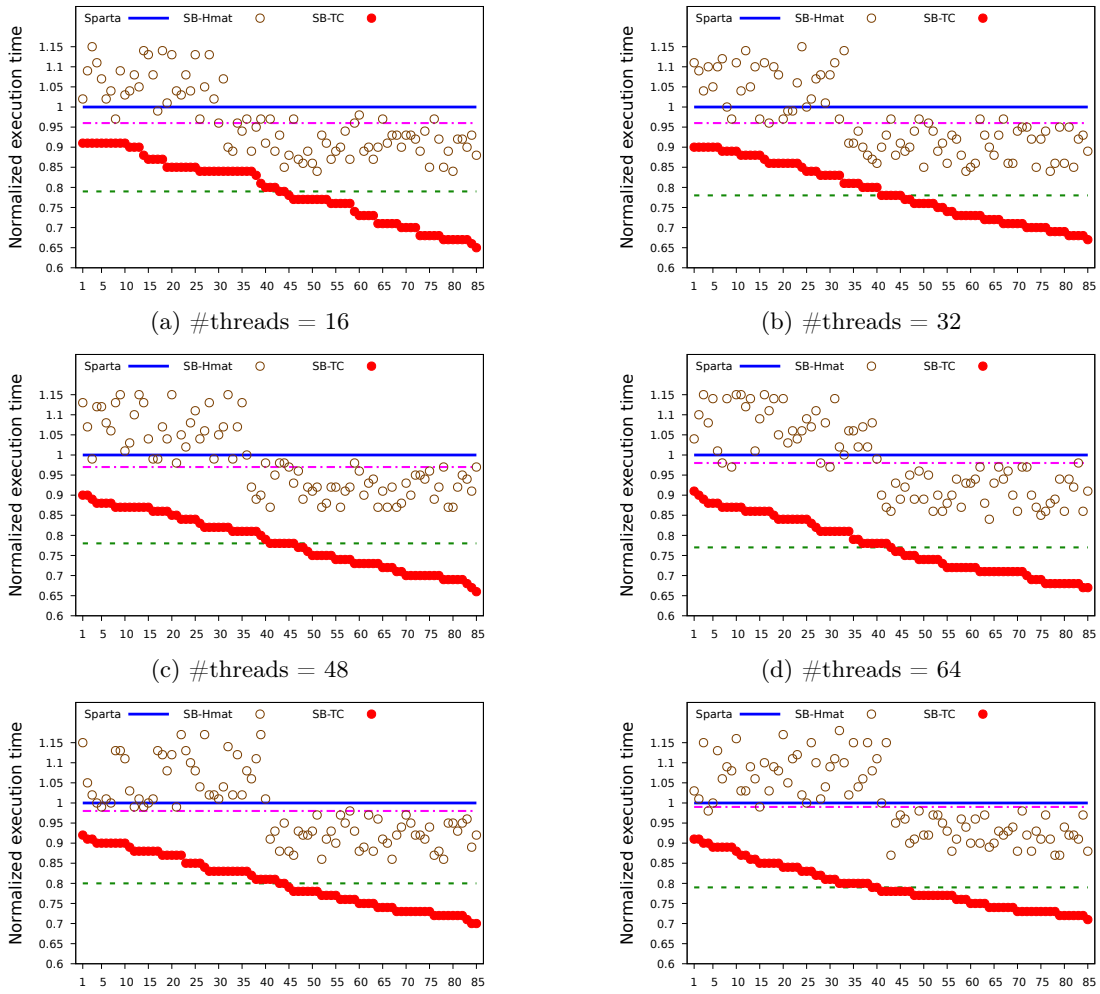
Tensor name	**Contraction modes**	**#flops** $\times 10^9$	**output *nnz*** $\times 10^9$	**Preprocessing (s)**	**SpGETT (s)**
enron	{1,3}	47.13	6.93	23.41	6383.67
chicago_crime	{2,3}	72.81	16.93	2.18	20424.06

Table 2: Run time (in s) of Sparta on two representative instances for which Sparta computed the tensor contraction in more than one hour.

We present in Table 2 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 s and the time for SpGETT is 6383.67 s. 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 s and the SpGETT time is 20424.06 s. 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 s, and on chicago_crime with contraction modes {2,3}, SB-TC completed in 16694.36 s. 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



(a) #threads = 16 (b) #threads = 32

(c) #threads = 48 (d) #threads = 64

(e) #threads = 80 (f) #threads = 96

Figure 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. The geometric mean of the ratio of SB-TC to Sparta is shown with the green dashed line and the geometric mean of the ratio of SB-Hmat to Sparta is shown with the dot-dashed magenta line.

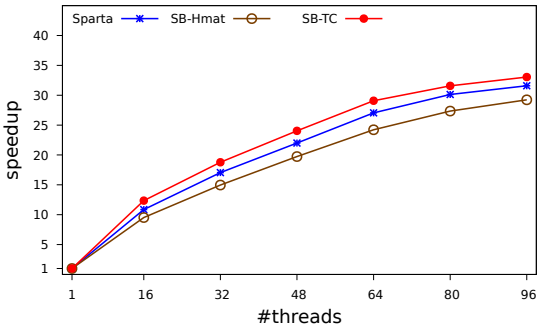


Figure 5: Overall scalability of Sparta, SB-Hmat and SB-TC on all the instances. Speedup of SB-TC, SB-Hmat and Sparta is with respect to their respective sequential run time.

175.08 s on average, SB-Hmat takes 162.64 s on average and SB-TC takes 132.44 s on average. For parallel execution with 96 threads, across all input instances, Sparta takes 5.66 s, SB-Hmat takes 5.71 s and SB-TC takes 4.13 s on average.

7 Conclusion

We have investigated two approaches to performing parallel sparse tensor-sparse tensor multiplication (SpGETT) on shared 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 used this hashing method to implement SB-Hmat, a state-of-the-art method to compute SpGETT via reduction to SpGEMM, and have shown through 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 comparison with the existing state-of-the-art parallel method for SpGETT. Overall, SB-TC obtains about 21% 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 preprocessing, in which matrices or tensors are reordered for better data locality. A suitable reordering method should have a low overhead and should be amenable to parallelization. Much work remains to be done on algorithms for such reordering methods.

Acknowledgment

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*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343619. doi: 10.1145/2925426.2926273. URL 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*, 38(6):C624–C651, 2016. doi: 10.1137/15M104253X. URL https://doi.org/10.1137/15M104253X.

[3] B. W. Bader and T. G. Kolda. Efficient MATLAB Computations with Sparse and Factored Tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007. doi: 10.1137/060676489.

[4] B. W. Bader, T. G. Kolda, et al. MATLAB Tensor Toolbox Version 3.6. Available online https://www.tensortoolbox.org/, September 2023.

[5] M. Bansal, O. Hsu, K. Olukotun, and F. Kjolstad. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi: 10.1145/3591236. URL 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*, 34(4):C123–C152, 2012. doi: 10.1137/110838844. URL 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*, 27, dec 2022. ISSN 1084-6654. doi: 10.1145/3568421. URL 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*, 32(2):145–163, March 2015. ISSN 1053-5888. doi: 10.1109/MSP.2013.2297439.

[9] E. Cohen. Structure Prediction and Computation of Sparse Matrix Products. *Journal of Combinatorial Optimization*, 2(4):307–332, Dec 1998. ISSN 1573-2886. doi: 10.1023/A:1009716300509. URL https://doi.org/10.1023/A:1009716300509.

[10] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006. doi: 10.1137/1.9780898718881. URL 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*, 78:33–46, 2018. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2018.06.009. URL 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*, 23(4):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)*, pages 175–185, Philadelphia, PA, 2020. SIAM.

[14] M. Fishman, S. R. White, and E. M. Stoudenmire. The ITensor Software Library for Tensor Network Calculations. *CoRR*, abs/2007.14822, 2020. URL `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*, 31(3):538–544, 1984. ISSN 0004-5411.

[16] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse Matrices in MATLAB: Design and Implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992. doi: 10.1137/0613024. URL `https://doi.org/10.1137/0613024`.

[17] F. G. Gustavson. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, sep 1978. ISSN 0098-3500. doi: 10.1145/355791.355796. URL `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*, 40(2):C258–C281, 2018. doi: 10.1137/17M1115873. URL `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)*, pages 1–10, 2014. doi: 10.1109/HiPC.2014.7116881.

[20] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133901. URL `https://doi.org/10.1145/3133901`.

[21] T. G. Kolda and B. W. Bader. Tensor Decompositions and Applications. *SIAM Review*, 51 (3):455–500, 2009. doi: 10.1137/07070111X. URL `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*, SC '18, New York, NY, USA, 2018. ACM. ISBN 978-1-5386-8384-2.

[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)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.SEA.2017.25. URL `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)*, pages 47–57, 2017. doi: 10.1109/CLUSTER.2017.75.

[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*, ICS '21, pages 190–202, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383356. doi: 10.1145/3447818.3460355. URL `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*, PPoPP '21, pages 318–333, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382946. doi: 10.1145/3437801.3441581. URL 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*, pages 370–381, 2014. doi: 10.1109/IPDPS.2014.47.

[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*, 90:102545, 2019. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2019.102545. URL 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*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290. doi: 10.1145/3295500.3356216. URL https://doi.org/10.1145/3295500.3356216.

[30] R. Pagh and F. F. Rodler. Cuckoo Hashing. In F. M. auf der Heide, editor, *Algorithms — ESA 2001*, pages 121–133. Springer Berlin Heidelberg, 2001.

[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*, PPoPP '20, page 362–375, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368186. doi: 10.1145/3332466.3374521. URL 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*, pages 1339–1348, Virtual Event, Canada, 2021. ACM.

[33] C. Psarras, L. Karlsson, J. Li, and P. Bientinesi. The landscape of software for tensor computations, 2022. URL 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*, 65(13):3551–3582, 2017. doi: 10.1109/TSP.2017.2690524.

[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)*, pages 265–274. IEEE, IEEE CPS, 2022. doi: 10.1109/IPDPSW55747.2022.00056.

[36] 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*, IA³ '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340014. doi: 10.1145/2833179.2833183. URL https://doi.org/10.1145/2833179.2833183.

[37] 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.

[38] E. Solomonik and T. Hoefler. Sparse Tensor Algebra as a Parallel Programming Model. *CoRR*, abs/1512.00066, 2015. URL `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*, 74(12):3176–3190, 2014. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2014.06.002. URL `https://www.sciencedirect.com/science/article/pii/S074373151400104X`. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

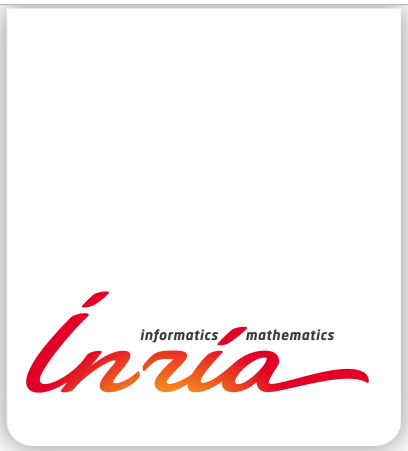
[40] P. Springer and P. Bientinesi. Design of a High-Performance GEMM-like Tensor–Tensor Multiplication. *ACM Trans. Math. Softw.*, 44(3), jan 2018. ISSN 0098-3500. doi: 10.1145/3157733. URL `https://doi.org/10.1145/3157733`.

[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*, 71(08):1968–1981, aug 2022. ISSN 1557-9956. doi: 10.1109/TC.2021.3113028.

[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)*, pages 27–38, 2021. doi: 10.1109/LLVMHPC54804.2021.00009.

[43] C. Uphoff and M. Bader. Yet Another Tensor Toolbox for Discontinuous Galerkin Methods and Other Applications. *ACM Trans. Math. Softw.*, 46(4), oct 2020. ISSN 0098-3500. doi: 10.1145/3406835. URL `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.*, 20(1), dec 2022. ISSN 1544-3566. doi: 10.1145/3566054. URL `https://doi.org/10.1145/3566054`.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399