THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Compiling structured tensor algebra

Citation for published version:
Ghorbani, M, Huot, M, Hashemian, S & Shaikhha, A 2023, 'Compiling structured tensor algebra', *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, 229, pp. 204-233.
https://doi.org/10.1145/3622804

Digital Object Identifier (DOI):
10.1145/3622804

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
Proceedings of the ACM on Programming Languages

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 06. Aug. 2024

Compiling Structured Tensor Algebra

MAHDI GHORBANI, University of Edinburgh, UK
MATHIEU HUOT, University of Oxford, UK
SHIDEH HASHEMIAN, University of Edinburgh, UK
AMIR SHAIKHHA, University of Edinburgh, UK

Tensor algebra is essential for data-intensive workloads in various computational domains. Computational scientists face a trade-off between the specialization degree provided by dense tensor algebra and the algorithmic efficiency that leverages the structure provided by sparse tensors. This paper presents StructTensor, a framework that symbolically computes structure at compilation time. This is enabled by Structured Tensor Unified Representation (STUR), an intermediate language that can capture tensor computations as well as their sparsity and redundancy structures. Through a mathematical view of lossless tensor computations, we show that our symbolic structure computation and the related optimizations are sound. Finally, for different tensor computation workloads and structures, we experimentally show how capturing the symbolic structure can result in outperforming state-of-the-art frameworks for both dense and sparse tensor algebra.

CCS Concepts: • **Mathematics of computing** → **Mathematical software**; • **Software and its engineering** → **Compilers**; *Domain specific languages.*

Additional Key Words and Phrases: Tensor algebra, Structured tensors, Code generation, Program analysis, Program synthesis

ACM Reference Format:
Mahdi Ghorbani, Mathieu Huot, Shideh Hashemian, and Amir Shaikhha. 2023. Compiling Structured Tensor Algebra. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 229 (October 2023), 30 pages. https://doi.org/10.1145/3622804

1 INTRODUCTION

Linear and tensor algebra are the key drivers of data-intensive computations in many domains, such as physics simulations [Martín-García 2008; Ran et al. 2020], computational chemistry [Hirata 2006; Titov et al. 2013], bioinformatics [Cichocki et al. 2009], and deep learning [Hirata 2003; Smith and Gray 2018]. Due to their importance, many specialization attempts have been made throughout the entire system stack from hardware to software layers. The tensor accelerators [Hegde et al. 2019] and TPUs [Jouppi et al. 2017] provide efficient tensor processing at the hardware level. On the software level, there have been advances in providing highly-tuned kernels [Dongarra et al. 1990], and compilation frameworks that globally optimize tensor computations [Gareev et al. 2018].

There is a trade-off for using tensor algebra frameworks between the specialization degree and the flexibility of the structure such as sparsity (Figure 1). On one side of the spectrum, extensive research has been done on dense tensors without leveraging any structure. Such tensors appear in deep neural networks and computational physics. As all the memory access patterns are known at compilation time, one can provide heavily-tuned implementations without any knowledge about

Authors' addresses: Mahdi Ghorbani, mahdi.ghorbani@ed.ac.uk, University of Edinburgh, Edinburgh, UK; Mathieu Huot, mathieu.huot@stx.ox.ac.uk, University of Oxford, Oxford, UK; Shideh Hashemian, s.hashemian@sms.ed.ac.uk, University of Edinburgh, Edinburgh, UK; Amir Shaikhha, amir.shaikhha@ed.ac.uk, University of Edinburgh, Edinburgh, UK.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/10-ART229
https://doi.org/10.1145/3622804

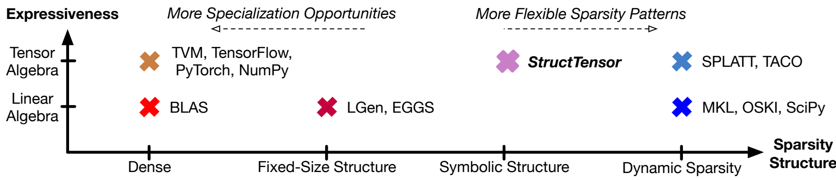

Fig. 1. Comparison of different linear and tensor algebra frameworks.

the content of tensors. As a result, a high-performance engineer or compiler has enough reasoning power to make decisions on parallelization, vectorization, and tiling.

However, many real-world applications involve tensors that exhibit specific structures (e.g., symmetry, lower/upper triangular, Toeplitz-like) which can be exploited to significantly reduce computational costs. Sparse tensor algebra, for instance, only captures the pattern of zero/non-zero elements in tensors at runtime, and has been at the centre of recent interest [Kjolstad et al. 2017; Strout et al. 2018]. However, postponing decisions about memory access patterns to runtime hinders the specialization power of compilers [Augustine et al. 2019].

Moreover, many real-world cases, such as using covariance matrices for training in machine learning (ML), convolutions computed with Toeplitz-like matrices [Hansen 2002], population growth modeled with Leslie matrices [Hansen 1989], pruned neural networks [Blalock et al. 2020], and immutable graphs, have statically known structures. As a partial remedy, there have been efforts [Spampinato and Püschel 2016; Tang et al. 2020a] to statically determine the structure of matrices during the compilation time. However, these are limited to fixed-size matrices. In practice, especially when dealing with machine learning problems, datasets often have varying dimensions. Using fixed sizes during compilation has two major drawbacks. Firstly, any change in the data size necessitates recompilation. Secondly, capturing intricate structures is challenging since structure patterns are defined globally over the matrix. It is not possible to capture global patterns using a divide-and-conquer approach and locally determined patterns using fixed-size windows. Therefore, a new system capable of capturing structure for variable-size tensors is required.

To resolve the dilemma between using tensor algebra frameworks focusing on either dense or sparse tensors (Figure 1), and also overcoming the fixed-size analysis limitations, this paper introduces StructTensor. StructTensor provides compile-time tracking of sparsity and redundancy structure in a *symbolic* way through various tensor operations on variable-sized tensors. On the one hand, the underlying compiler can use this symbolic information to specialize the code at the level of dense computations. On the other hand, the compiler can leverage this symbolic information to eliminate unnecessary and redundant computation. StructTensor enables high-performance computation for ML training over large volumes of data using structure propagation capabilities.

In StructTensor, all tensor computations and structure information are translated to a *single* intermediate language called Structured Tensor Unified Representation (STUR). STUR propagates the structure throughout the computation at compile time, followed by efficient C++ code generation.

Specifically, we make the following contributions:

- We present StructTensor, the first framework that supports structured computation for tensor algebra on variable-sized tensors (Section 3). For a tensor T, the structure handled by StructTensor comes as a pair (T_U, T_R), where
  - T_U tracks the symbolic sparsity structure of the tensor,
  - T_R tracks the redundancy structure, which captures repetition patterns (e.g., symmetric).
- We propose STUR, a unified intermediate representation (IR) that can express structured and non-structured tensor computations. It allows capturing non-optimized tensors as well as their symbolic structures T_U, T_R in a single IR (Section 4).

$$e ::= e \cdot e \mid e \odot e \mid e \otimes e \mid e + e \mid e \oplus e \mid e^T$$

$\mathcal{G}$	General dense matrix
$\mathcal{S}$	Symmetric matrix
$\mathcal{D}$	Diagonal matrix
$\mathcal{R}_n$	Matrix of non-zeros only at n^{th} row
C_n	Matrix of non-zeros only at n^{th} column
$\mathcal{H}_{n,m}$	Matrix of non-zeros only at element $[n][m]$
$\mathcal{Z}$	All zeros matrix

$$\frac{e\colon C_n}{e \cdot e^T\colon \mathcal{S}} \qquad \frac{e_1\colon C_n \quad e_2\colon \mathcal{R}_m \quad n \neq m}{e_1 \cdot e_2\colon \mathcal{Z}}$$

$$\frac{e_1\colon C_n \quad e_2\colon \mathcal{R}_n}{e_1 \cdot e_2\colon \mathcal{G}} \qquad \frac{e_1\colon \mathcal{R}_n \quad e_2\colon C_m}{e_1 \cdot e_2\colon \mathcal{H}_{n,m}}$$

$$\frac{e_1\colon \mathcal{R}_n \quad e_2\colon \mathcal{R}_m}{e_1 \cdot e_2\colon \mathcal{R}_n} \qquad \frac{e_1\colon C_n \quad e_2\colon C_m}{e_1 \cdot e_2\colon C_m}$$

$$\frac{e_1\colon \mathcal{D} \quad e_2\colon \mathcal{D}}{e_1 \odot e_2\colon \mathcal{D}} \qquad \frac{e_1\colon \mathcal{D} \quad e_2\colon \mathcal{D}}{e_1 \otimes e_2\colon \mathcal{D}}$$

$$\frac{e_1\colon \mathcal{D} \quad e_2\colon \mathcal{D}}{e_1 \cdot e_2\colon \mathcal{D}} \qquad \frac{e_1\colon \mathcal{D} \quad e_2\colon \mathcal{D}}{e_1 \oplus e_2\colon \mathcal{D}}$$

Fig. 2. The grammar for structured linear algebra operations and a subset of structure inference rules.

- We show how StructTensor uses the structure information in its compilation process (Section 5). It leverages STUR and rewrites tensors in 3 steps (also see Figure 4):
  (1) StructTensor propagates the structure on the AST of the tensor computation (Section 5.1)
  (2) Several optimizations are applied to the unified representation for tensor computations and their structure leading to computations over a compressed tensor (Section 5.2)
  (3) StructTensor generates C++ code for the tensor computation over the compressed form, as well as reconstructing the uncompressed tensor (Section 5.5)
- We give a mathematical view on the problem of lossless tensor computations that we study in this paper, and show the soundness of our rewrites and structure inference in Section 6.
- We experimentally evaluate StructTensor for different tensor workloads and structures (Section 7). We show that StructTensor leverages the structure to generate computation over compressed tensors and outperforms state-of-the-art dense and sparse tensor frameworks.

2 BACKGROUND

Structured Linear Algebra. There are several well-known structures for matrices, such as diagonal, symmetric, and lower/upper triangular. Incorporating these structures into matrix calculations can significantly decrease computational time. Figure 2 represents a set of linear algebra operations and well-known matrix structures.

Example - Diagonal Matrices Kronecker Product. Consider the example of the Kronecker product between two diagonal matrices $A_{n\times n}$ and $B_{m\times m}$:

$$(A \otimes B)[mr + v][ms + w] = A[r][s] \cdot B[v][w] \tag{1}$$

where the resulting matrix has dimension $nm \times nm$. Therefore, the computational cost would be $O(n^2m^2)$. However, if the computation is structure-aware, one can leverage the fact that all non-zero elements are on the diagonal of matrices. Therefore, it is sufficient to only perform the multiplication over the diagonal, which results in the following computation:

$$(A \otimes B)[mr + v][mr + v] = A[r][r] \cdot B[v][v] \tag{2}$$

where the result is diagonal as well. As a result, the computational complexity for this Kronecker product reduces to $O(nm)$. Furthermore, since the structure of the result is known (diagonal), this information can be used in further computations efficiently. Figure 2 shows a subset of inference rules that can be used to determine the output structure based on input structures.

Sparsity and Redundancy Structures. Structures that distinguish zero and non-zero values are referred to as *sparsity structures*, such as diagonal and lower/upper triangular. Other structures like symmetric and circulant can capture redundancy patterns and are thus referred to as *redundancy structures*. By knowing the structures, it is possible to reconstruct the whole matrix by storing only

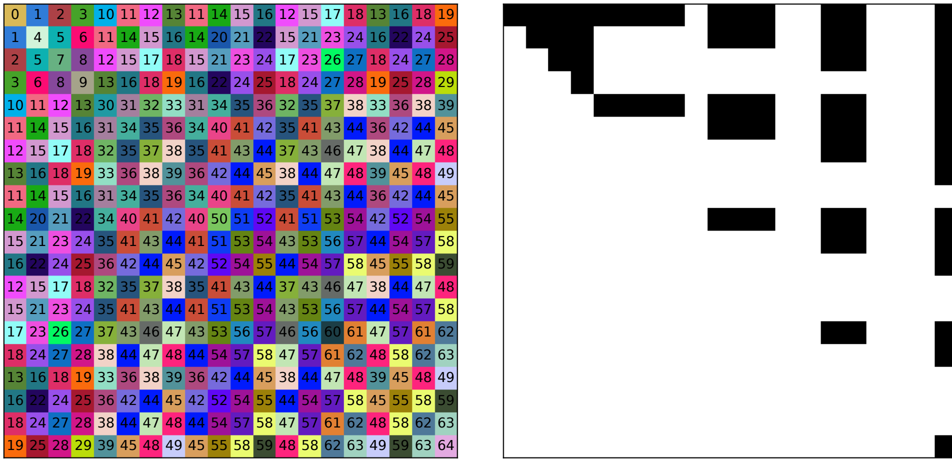

Fig. 3. The covariance matrix of Polynomial Regression degree-2. Its redundancy pattern is shown on the left and its unique elements in compressed format are on the right.

non-zero unique elements. The lower triangular part of a symmetric matrix, and the first column of a circulant matrix are the unique elements of those matrices. Knowing and propagating structures in many cases helps reduce computational costs [Spampinato and Püschel 2016].

Intricate Structures. The aforementioned well-known structures cannot cover more complicated patterns. Consider the example of creating the covariance matrix for the polynomial regression degree two model. Figure 3 shows the covariance matrix created for a polynomial regression degree-two model on a vector of length 4. Various colors in this figure are used for visualization purposes; elements with the same color/number have the same unique value. As it is represented in Figure 3, there are only 65 distinct elements even though the covariance matrix dimension is 20×20. The redundancy pattern in this matrix is sophisticated and cannot be captured using the existing well-known structures (e.g., symmetric). Utilizing such structure information enhances the performance of machine learning tasks that require covariance matrix creation such as training polynomial regression, PCA, and factorization machine models.[1]

Tensor Structures. The complicated structure of the mentioned covariance matrix can be captured in the form of a higher-order tensor structure. StructTensor decomposes the covariance matrix computation for higher-degree machine learning models into basic tensor operations. Then, it captures the global structure across these basic operations by composing them non-trivially. For example, the covariance matrix used for training a polynomial regression degree two contains all degree-2, degree-3, and degree-4 interactions between features in itself. Redundancy occurs because these interactions are calculated multiple times. For example, degree-2 interactions inside the matrix form a symmetric structure. Degree-3 and degree-4 interactions have some complex forms of redundancy as well as being symmetric. These interactions cover all the elements inside the covariance matrix. By representing degree i interactions as a tensor of order-i, the computation can be done more efficiently. This way, the only unique elements in degree two, three, and four tensors $T2(i,j), T3(i,j,k), T4(i,j,k,l)$ reside in $\{T2(i,j) \mid 1 \le i \le j \le n\}$, $\{T3(i,j,k) \mid 1 \le i \le j \le k \le n\}$, $\{T4(i,j,k,l) \mid 1 \le i \le j \le k \le l \le n\}$, respectively. This structure in the new data format forms a generalized symmetric pattern that is easy to capture and leads to efficient computation. In

[1]We consider factorization-related optimizations, techniques that aim to improve the performance by leveraging low ranks of tensors, orthogonal to structured tensors; our technique appears after such transformations, as can be seen in Section 7.5.

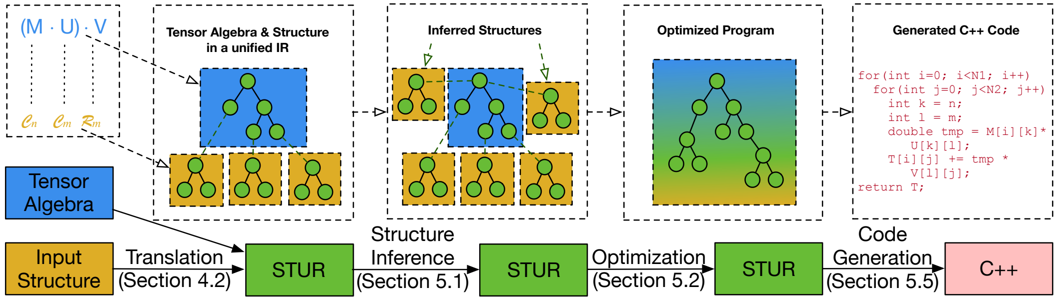

Fig. 4. StructTensor architecture overview.

the following sections, it is explained how StructTensor handles sophisticated structures and generates efficient C++ code for the computations.

3 OVERVIEW

In this section, the overall architecture of StructTensor is described (cf. Figure 4).

Input. StructTensor gets linear algebra and tensor algebra expressions, as well as the structure of their tensor parameters as input. These operations and structures subsume the ones that are mentioned in Figure 2. All tensor expressions, as well as their structures, are represented in the StructTensor unified representation, called STUR. Sparsity and redundancy structures are captured in a unique set and redundancy map, respectively. The unique set only contains non-zero elements, and must contain each distinct value appearing in the original tensor. The redundancy map represents the mapping from redundant and non-zero elements outside the unique set to their corresponding value in the unique set. These two subsume polyhedral set and map data structures that maintain the fixed-size and static structure information of matrices in LGen [Spampinato and Püschel 2016] (`SInfo` and `AInfo`).

Structure Inference. Afterwards, we infer the structure of the intermediate and output tensor expressions by propagating the structure through STUR. A predefined set of inference rules for operations and structures is provided in STUR. We apply these rules to infer the output structure, which is represented by a unique set and a redundancy map. The inference rules set is extensible to cover arbitrary operations and structures as well.

Optimizations. Various optimizations are applied to STUR expressions. Throughout the structure inference process, several intermediate unique sets and redundancy maps are created. The tensor inlining optimization removes intermediate sets and maps. Moreover, input structures and previous optimizations by STUR can produce repetitive, trivial, or contradictory conditions over iterators. Through logical simplifications, only distinct and relevant conditions are kept for the code generation step.

Code Generation. Finally, the optimized STUR is fed to the C++ code generator. The code generator assumes an order for iterators, calculates loop nest boundaries, and generates an efficient structure-aware C++ code. The generated code performs the computations over compressed format with lower computational cost thanks to the unique set. By utilizing the redundancy map, the final output tensor can be reconstructed for the user.

Example - Polynomial Regression Degree-2. We consider the covariance matrix creation for polynomial regression degree two. The covariance matrix for a vector of length n is defined as:

$$\Sigma(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x}^T = \mathbf{x} \otimes \mathbf{x}$$

The multiplication of $\mathbf{x}$ with its transpose is equivalent to the outer product of $\mathbf{x}$ with itself, and thus can be represented using the vector outer product operation $\otimes$. The vector $\mathbf{x}$ is formed by

concatenating the feature vector $\mathbf{f}$ with the vector obtained from the interaction of the features. In terms of linear algebra, this is expressed as:

$$\mathbf{x} = \mathbf{f} \oplus vec(\mathbf{f} \otimes \mathbf{f})$$

Here, $\oplus$ is the vector concatenation, $\otimes$ is the vector outer product, and $vec(M)$ flattens the matrix M of size $n \times m$ into a vector of size $n \cdot m$. By inlining the definition of $\mathbf{x}$ and using the distributive property of vector concatenation over vector outer product, we have:

$$\begin{aligned} \Sigma(\mathbf{x}) &= (\mathbf{f} \oplus vec(\mathbf{f} \otimes \mathbf{f})) \quad \otimes \quad (\mathbf{f} \oplus vec(\mathbf{f} \otimes \mathbf{f})) \\ &= \big((\mathbf{f} \otimes \mathbf{f}) || (\mathbf{f} \otimes vec(\mathbf{f} \otimes \mathbf{f}))\big) \quad // \quad \big((vec(\mathbf{f} \otimes \mathbf{f}) \otimes \mathbf{f}) || (vec(\mathbf{f} \otimes \mathbf{f}) \otimes vec(\mathbf{f} \otimes \mathbf{f}))\big) \end{aligned}$$

The operations || and // correspond to matrix horizontal and vertical concatenation, respectively.

StructTensor improves the performance for this computation in two levels of granularity. At the coarse-level, it applies common sub-expression elimination at the addition level and detects that $(\mathbf{f} \otimes vec(\mathbf{f} \otimes \mathbf{f}))$ and $(vec(\mathbf{f} \otimes \mathbf{f}) \otimes \mathbf{f})$ are computing the same elements but in a different layout. Thus, it only performs one of the computations, and it is sufficient to compute the following vector outer product terms:

$$\begin{aligned} M1 &= \mathbf{f} \otimes \mathbf{f} \\ M2 &= \mathbf{f} \otimes vec(\mathbf{f} \otimes \mathbf{f}) \\ M3 &= vec(\mathbf{f} \otimes \mathbf{f}) \otimes vec(\mathbf{f} \otimes \mathbf{f}) \end{aligned}$$

At a finer-level, for each of the terms $M1, M2, M3$, StructTensor detects a generalized symmetric structure. For $M1$, it detects a standard symmetric structure where it is sufficient to only keep the upper half of the matrix. For $M2$ and $M3$, there are $\sim 5\times$ and $\sim 23\times$ redundant elements, the patterns of which can be seen in Figure 3. To be more specific, StructTensor reduces the total number of elements from $n^4 + 2n^3 + n^2$ to $\frac{n^4+10n^3+35n^2+26n}{24}$. After inferring such structures, StructTensor generates the following C++ code for each of these terms:

```
// Computation for M1
for(int i=0; i<n; ++i){
 for(int j=i; j<n; ++j){
  M1[i][j]=f[i]*f[j];
}}
```

```
// Computation for M2
for(int i=0; i<n; ++i){
 for(int j=i; j<n; ++j){
  for(int k=j; k<n; ++k){
   int col=j*n+k;
   M2[i][col]=f[i]*f[j]*f[k];
}}}
```

```
// Computation for M3
for(int i=0; i<n; ++i){
 for(int j=i; j<n; ++j){
  for(int k=j; k<n; ++k){
   for(int l=k; l<n; ++l){
    int r=i*n+j;
    int c=k*n+l;
    M3[r][c]=f[i]*f[j]*f[k]*f[l];
}}}}
```

Note that the nested iterations only cover a subset of the full range, which is when all iterators are from 0 to n. The generated loop bounds are computed based on the unique sets. We will elaborate on this in Section 5.

In the next sections, we will thoroughly elaborate on how StructTensor works.

4 STUR: STRUCTURED TENSOR UNIFIED LANGUAGE

In this section, we present the syntax of the unified intermediate representation, STUR. We detail the grammar for unique sets, redundancy maps, and compressed tensor computations in this section. Moreover, we elaborate on the representation of linear algebra in STUR in this section.

4.1 The Syntax of STUR

Grammar. Figure 5 shows the grammar of STUR, which covers the grammar for tensors, unique sets, and redundancy maps computations. Each program (P) is made of several rules (R). Each rule is in the form of an assignment from a body (B) to access to a collection (A). The collection can be a tensor (T), compressed tensor (T_C), unique set (T_U), or redundancy map (T_R), whereas the access index can be multiple index variables ($\overline{X}$), an index variable, or a constant value. The assignment body is represented as a sum of factor products (F). Each factor restricts the domain of values that

Program	p	::=	$r \mid r; p$	List of rules.
Rule	r	::=	$A := B$	Head (access) and body (Sum of Products).
Body	B	::=	$f \mid f + B$	Sum of factor products.
Factor	f	::=	$e \mid e * f$	Product of expressions.
Expression	e	::=	$x\ \theta\ i \mid i\ \theta\ x \mid A$	Comparison ($\theta \in \{<, \leq, =, \geq, >\}$) or access.
Index	i	::=	$x \mid c \mid i \diamond i$	Variable, constant, or arithmetic ($\diamond$) over indices.
Access	A	::=	$T(\overline{x}) \mid T_C(\overline{x}) \mid T_U(\overline{x}) \mid T_R(\overline{x})$	Tensor, compressed tensor, unique set, and redundancy map access.

Fig. 5. Grammar of STUR. The meta variables x and T range over the name of indices and tensors, respectively.

$$\frac{T(x_1, \ldots, x_n) := e}{FV(T(x_1, \ldots, x_n)) \subseteq FV(e)} \qquad FV(e1 + e2) = FV(e1) \cap FV(e2) \qquad FV(x\theta t) = \{x\} \cup FV(t)$$

$$FV(e1 * e2) = FV(e1) \cup FV(e2) \qquad FV(x) = \{x\}$$

$$FV(T(x_1, \ldots, x_n)) = \{x_1, \ldots, x_n\} \qquad FV(t1 \diamond t2) = FV(t1) \cup FV(t2) \qquad FV(c) = \emptyset$$

Fig. 6. Free variable rules.

an index variable can have. This is achieved through a collection access or a comparison term. STUR treats tensors and compressed tensors as multi-dimensional arrays. Unique sets and redundancy maps are considered as sets in STUR. When a computation involves multiplying a set by a tensor, only the values in that tensor corresponding to the specified set will be used in computations.
Example - Simple Tensor Operation. Consider the following STUR program:

$$\begin{aligned} T_3(x, y) &:= T_1(x, y) * T_2(x, y) \\ T_4(x) &:= T_3(x, y) * (x = y) \end{aligned}$$

This program consists of two rules, with two input tensors T_1 and T_2. The first rule constructs an order-2 tensor T_3, i.e. a matrix, which is computed by performing an element-wise multiplication of the two input matrices. The second rule constructs an order-1 tensor, i.e. a vector, that contains the elements of the diagonal of the matrix T_3. This is achieved by (1) restricting the range of x and y by the comparison term $x = y$, and (2) existentially quantifying over y by not including it in the head of the rule. The second step sums over the second dimension after the range is restricted.
Sum-of-Product Semantics. The addition and multiplication in STUR are defined based on the underlying collection; when the domains of two collections overlap, for unique sets and redundancy maps, the addition and multiplication are defined as set union and intersection, whereas, for tensors and compressed tensors, they are defined as real number addition and multiplication. When the domain of two collections is disjoint, addition is impossible, and multiplication will act as an outer product; tensor outer product for tensors and cartesian product for sets. Each rule can existentially quantify the free variables of the body by not including them in the rule head. This is done by summing over the dimensions corresponding to the free variables. This is known as *marginalization* in the AI community. All the free variables in the head must be defined in the body (cf. Figure 6).
Example - Problematic Addition. To better illustrate the free variable rules, consider:

$$T1(i) \quad := \quad T2(i, j) + T3(i, k)$$

where $T2(i, j) := M2(i, j) * (0 \leq i < n) * (j = i)$ and $T3(i, k) := M3(i, k) * (n \leq i < 2 * n) * (k = i)$. In this computation, k is unbounded for $T2$, and j is unbounded for $T3$. Therefore, if we do not sum over k in $T2$ and j in $T3$, it is impossible to perform the computation.
Syntactic Sugar. To simplify the presentation, we consider the following syntactic sugar, where $\boldsymbol{x}$ corresponds to a list of arguments $x_1, \ldots, x_k$:

$$\begin{aligned} (a \leq x < c) &\equiv (a \leq x) * (x < c) \\ (x = y = z) &\equiv (x = y) * (y = z) \\ \boldsymbol{x}\theta\boldsymbol{b} &\equiv (x_1\theta b_1) * \ldots * (x_k\theta b_k) \end{aligned}$$

Unique Sets. The sparsity structure of all the distinct elements of a tensor is encoded in a unique set. The grammar of STUR already captures the definition of unique sets (cf. Figure 5); a unique set is provided as a sum of products of comparison factors or access to other unique sets (and redundancy maps). Unique sets enhance performance by restricting index boundaries for structured tensors. For unstructured tensors, the whole tensor is stored in its unique set, and the dimensions of the tensor are then given as input to its unique set.

Example - Chess Pattern Unique Set. We elaborate on the aforementioned definitions through an example. Consider the case of a matrix T of size $m \times n$ with a chess-board sparsity pattern. The set comprehension representation of a unique set for this matrix is:

$$\begin{aligned} UniqueSet \quad &:= \quad \{(i,j) | (0 \leq i' < (m/2)) \wedge (0 \leq j' < (n/2)) \wedge (i = i' * 2) \wedge (j = j' * 2 + 1)\} \cup \\ & \quad\quad \{(i,j) | (0 \leq i' < (m/2)) \wedge (0 \leq j' < (n/2)) \wedge (i = i' * 2 + 1) \wedge (j = j' * 2)\} \end{aligned}$$

The first set specifies the elements of the even rows (with an odd column index), and the second one specifies the ones for odd rows (with an even column index). In STUR, the $\wedge/\cap$ and $\cup/\vee$ operators are translated to $*$ and $+$, respectively. Therefore the described unique set is translated to:

$$\begin{aligned} T_U(i,j) \quad &:= \quad (0 \leq i' < (m/2)) * (0 \leq j' < (n/2)) * (i = i' * 2) * (j = j' * 2 + 1) + \\ & \quad\quad (0 \leq i' < (m/2)) * (0 \leq j' < (n/2)) * (i = i' * 2 + 1) * (j = j' * 2) \end{aligned}$$

Here, i' and j' are free variables that appear on the right-hand side but do not appear on the head. Thus, we existentially quantify over i' and j', which is achieved by summing over them in STUR.

Redundancy Maps. The remaining non-zero elements that do not appear in the unique set domain are covered by the redundancy map. A redundancy map keeps the association between these repetitive elements' indices and their corresponding indices in the unique set. Similar to the unique set, it is represented as a sum of products of comparison factors or access to other unique sets and redundancy maps. For a tensor of order-k, the redundancy map has $2k$ index variables. The first k index variables correspond to the indices of the redundant elements, whereas the second k ones correspond to the indices from the unique set. The redundancy map enables the capability of reconstructing the full matrix from the compressed version that only contains unique elements. Restricting the computations to the elements of the unique set and reconstructing the uncompressed final result once, when all the calculation is over, can significantly improve the performance.

Example - Identical Row Matrix Redundancy Map. Consider a matrix T of size $m \times n$, where all rows are the same. The unique set and redundancy maps for this matrix are defined as follows:

$$\begin{aligned} T_U(i,j) \quad &:= \quad (i = 0) * (0 \leq j < n) \\ T_R(i,j,i',j') \quad &:= \quad (0 < i < m) * (0 \leq j < n) * (i' = 0) * (j' = j) \end{aligned}$$

The first rule specifies that the unique elements are only in the first row ($i = 0$). The first factor of the second rule ($0 < i < m$) restricts the range of redundant elements to the ones from all the rows except the first one. The last two terms specify the index of the element from the unique set by specifying the first row ($i' = 0$) and the same column as the redundant element ($j' = j$).

Compressed Tensor. StructTensor leverages the structure for better performance by representing the tensor in a lossless compressed format. This is achieved by combining the original tensor with the unique set to extract only the unique elements. The compressed tensor T_C for the original tensor T is defined as:

$$T_C(\boldsymbol{x}) := T(\boldsymbol{x}) * T_U(\boldsymbol{x})$$

Naïvely executing the computation using this formula can even make the performance worse. After performing simplifications, the code generator produces code that iterates over the domain provided by the unique set for operations and excludes all the other elements of that tensor. This way, no extra computational cost is imposed while computing the result. When the computation is done, the uncompressed tensor T is retrievable by using T_R.

$$\begin{aligned}
[\![e^T]\!](i,j) &:= [\![e]\!](j,i) \\
[\![e_1 + e_2]\!](i,j) &:= [\![e_1]\!](i,j) + [\![e_2]\!](i,j) \\
[\![e_1 \cdot e_2]\!](i,j) &:= [\![e_1]\!](i,k) * [\![e_2]\!](k,j) \\
[\![e_1 \odot e_2]\!](i,j) &:= [\![e_1]\!](i,j) * [\![e_2]\!](i,j) \\
[\![e_1 \otimes e_2]\!](i,j) &:= [\![e_1]\!](i',j') * [\![e_2]\!](i'',j'') \\
\text{where } &(m,n) = dims(e_2), \\
&i' = \lfloor i/m \rfloor, j' = \lfloor j/n \rfloor \\
&i'' = i\%m, j'' = j\%n
\end{aligned}
\qquad
\begin{aligned}
[\![e_1 \oplus e_2]\!](i,j) &:= [\![e_1]\!](i,j) + [\![e_2]\!](i',j') \\
\text{where } &(m,n) = dims(e_1) \\
&i' = i - m, j' = j - n \\
[\![e_1 \cdot e_2]\!]() &:= [\![e_1]\!](i) * [\![e_2]\!](i) \\
[\![e_1 + e_2]\!](i) &:= [\![e_1]\!](i) + [\![e_2]\!](i) \\
[\![e_1 \oplus e_2]\!](i) &:= [\![e_1]\!](i) + [\![e_2]\!](i') \\
\text{where } &(n) = dims(e_1), i' = i - n \\
[\![e_1 \otimes e_2]\!](i,j) &:= [\![e_1]\!](i) * [\![e_2]\!](j)
\end{aligned}$$

Fig. 7. Representation of linear algebra operations in STUR. Here, $\cdot$ and $\otimes$ are overloaded; they correspond to matrix multiplication/Kronecker product for matrices and inner/outer product for vectors, respectively.

$$\frac{e\colon \mathcal{Z}}{T_U(i,j) := \emptyset} \qquad \frac{e\colon \mathcal{G}}{T_U(i,j) := (0 \le i < m) * (0 \le j < n)}$$

$$\frac{e\colon \mathcal{H}_{n1,n2} \quad 0 \le n1 < m \quad 0 \le n2 < n}{T_U(i,j) := (i = n1) * (j = n2)} \qquad \frac{e\colon \mathcal{D} \quad m = n}{T_U(i,j) := (0 \le i < n) * (i = j)}$$

$$\frac{e\colon \mathcal{R}_r \quad 0 \le r < m}{T_U(i,j) := (i = r) * (0 \le j < n)} \qquad \frac{e\colon \mathcal{S} \quad m = n}{\begin{aligned} T_U(i,j) &:= (0 \le i \le j < n) \\ T_R(i,j,i',j') &:= (0 \le j < i < n) * \\ &\quad\ (i' = j) * (j' = i) \end{aligned}}$$

$$\frac{e\colon \mathcal{C}_c \quad 0 \le c < n}{T_U(i,j) := (0 \le i < m) * (j = c)}$$

Fig. 8. Representation of well-known matrix structures in STUR. In all cases, $[\![e]\!] = T$ and T_U and T_R correspond to its unique set and redundancy map, respectively. Furthermore, we assume that $(m,n) = dims(e)$ for all cases and $T_R(i,j,i',j') := \emptyset$ unless stated explicitly. For the definition of the different structures, cf. Figure 2.

Example - Upper Triangular Matrix Compressed Tensor. A $n \times n$ upper triangular matrix compressed tensor is calculated as follows:

$$\begin{aligned}
T_U(i,j) &:= (0 \le i \le j < n) \\
T_C(i,j) &:= T(i,j) * (0 \le i \le j < n)
\end{aligned}$$

When $T(i,j)$ appears in a computation, the optimizer converts it to $T_C(i,j)$ and only uses elements with $0 \le i \le j < n$. Therefore, only half of the elements are used in the computation, which leads to a $\sim 2\times$ speed up.

4.2 Translating Structured Linear Algebra to STUR

Operations. The representation of linear algebra operations is shown in Figure 7. Imagine the matrices M_1 and M_2 with dimensions $m_1 \times n_1$ and $m_2 \times n_2$ are represented in STUR as $[\![e_1]\!]$ and $[\![e_2]\!]$, respectively. $[\![e_1 \bullet e_2]\!](i,j)$ represents the element at row i column j of $M_1 \bullet M_2$, where $\bullet$ is an operator. Each operation is translated to a sum of product format in STUR. For instance, if $\bullet = \oplus$, then the direct sum of M_1 and M_2 is translated by the top-right rule in Figure 7. This rule specifies that an element at the position (i,j) in the output is taken from $[\![e1]\!]$ when $i < m$ and $j < n$ and from $[\![e2]\!]$ when $i \ge m$ and $j \ge n$.

Structures. Figure 8 shows the representation of well-known matrix structures in STUR. Structures are translated to a combination of a unique set and a redundancy map. For example, if a matrix M has a symmetric structure, the upper triangular section of that is counted as the unique set. Therefore, the lower triangular region is considered as the redundant section; the redundancy map keeps the transformations from the lower triangular region to the corresponding upper region. Hence, if the representation of M in STUR is T, the unique set and redundancy map are provided by the last rule of Figure 8. When $j < i$, according to $T_R(i,j,i',j')$, the values of $T(i,j)$ will be retrieved from values in $T(j,i)$.

Algorithm 1 Overview of Compilation and Code Generation Algorithm

1: **function** COMPILE($inputs_U$, $inputs_R$, $expr$, $flags$)
2: $expr \leftarrow$ NORMALIZE($expr$)
3: $expr_U, expr_R \leftarrow$ STRUCTURE_INFERENCE($expr$)
4: $expr_C \leftarrow$ COMPRESS($expr, expr_U, expr_R$)
5: $expr_C, expr_R \leftarrow$ INLINE($expr_C, expr_R, inputs_U, inputs_R$)
6: $expr'_C \leftarrow$ RECONSTRUCT($expr_C, expr_R$)
7: $expr_C, expr'_C \leftarrow$ LOGICAL_SIMPLIFICATION($expr_C, expr'_C$)
8: **return** CODE_GEN($expr_C, flags$) + CODE_GEN($expr'_C, flags$)
9: **end function**

$$\frac{T(\boldsymbol{x}) := M(\boldsymbol{y}) * V(\boldsymbol{z}) \quad \boldsymbol{x} = \boldsymbol{y} \cup \boldsymbol{z}}{T_U(\boldsymbol{x}) := M_U(\boldsymbol{y}) * V_U(\boldsymbol{z})} \qquad \frac{T(\boldsymbol{x}) := M(\boldsymbol{x}) + V(\boldsymbol{x})}{T_U(\boldsymbol{x}) := M_U(\boldsymbol{x}) + V_U(\boldsymbol{x})} \qquad \frac{T(\boldsymbol{x}) := M(\boldsymbol{y}) \quad \boldsymbol{x} \subseteq \boldsymbol{y}}{T_U(\boldsymbol{x}) := M_U(\boldsymbol{y})}$$

Fig. 9. Inference rules for unique sets. The redundancy maps of the inputs are empty. $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ represent the sequence of variables for tensor accesses T, M, and V and their corresponding unique sets, T_U, M_U, and V_U.

5 COMPILATION

In this section, we show how StructTensor symbolically computes and propagates the structure. Optimizing the tensor computation using the inferred structure leads to structure-aware code generation. A pseudocode for the compilation algorithm is provided in Algorithm 1.

5.1 Structure Inference

Inference for Unique Sets. Figure 9 shows the inference rules for the output structure. These rules follow the assumption that input redundancy maps are empty and only capture sparsity patterns. Since all operations are translated to multiplication, addition, or projection in STUR, providing inference rules only for these operations is sufficient. Moreover, having more than two operands is handled by breaking them into sub-expressions with two operands and storing the results in intermediate sets. This procedure is called normalization (cf. Algorithm 1, line 2).

In the case of multiplication, the result is non-zero only if both inputs are non-zero values. Having a zero operand in multiplication makes the final result zero. Therefore, the unique set of output is calculated from the intersection of non-zero elements in both operands. Addition on the other hand is non-zero even if one of the operands is non-zero. Hence, any element in the unique set of operands can lead to a non-zero element in the output. Consequently, the unique set of output is the union over the unique set of its operands. Since projection is defined by summing up all values, it follows the same rule as addition. Any non-zero element in the unique set of the input can create a non-zero element in the output. So the output unique set for projection is computed by the union over all unique set values in that dimension (cf. Algorithm 1, line 3).

Example - Unique Set Computation. Consider the following tensor computation:

$$A(i,j) \quad := \quad B(i,k) * C(k,j) + D(i,j)$$

where all the inputs are $n \times n$ diagonal matrices. Therefore, input unique sets are:

$$\begin{aligned} B_U(i,k) &:= (0 \le i < n) * (0 \le k < n) * (i = k) \\ C_U(k,j) &:= (0 \le k < n) * (0 \le j < n) * (k = j) \\ D_U(i,j) &:= (0 \le i < n) * (0 \le j < n) * (i = j) \end{aligned}$$

StructTensor will normalize the computation and infers the unique set as follows:

$$\begin{array}{llll} (1) & E(i,j,k) := B(i,k) * C(k,j) & & E_U(i,j,k) := B_U(i,k) * C_U(k,j) \\ (2) & F(i,j) := E(i,j,k) & & F_U(i,j) := E_U(i,j,k) \\ (3) & A(i,j) := F(i,j) + D(i,j) & & A_U(i,j) := F_U(i,j) + D_U(i,j) \end{array}$$

By applying inlining and logical simplifications, the unique sets can be computed as follows:

Special Cases:

$$\frac{T(\boldsymbol{x}) := M(\boldsymbol{y}) * M(\boldsymbol{z}) \quad \boldsymbol{x} = \boldsymbol{y} \cup \boldsymbol{z} \quad \boldsymbol{y} \cap \boldsymbol{z} = \emptyset \quad \boldsymbol{x'} = \boldsymbol{y'} \cup \boldsymbol{z'} \quad \boldsymbol{y'} \cap \boldsymbol{z'} = \emptyset}{\begin{aligned} T_U(\boldsymbol{x}) &:= M_U(\boldsymbol{y}) * M_U(\boldsymbol{z}) * (\boldsymbol{y} \leq \boldsymbol{z}) \\ T_R(\boldsymbol{x}, \boldsymbol{x'}) &:= M_R(\boldsymbol{y}, \boldsymbol{y'}) * M_R(\boldsymbol{z}, \boldsymbol{z'}) + M_U(\boldsymbol{y}) * (\boldsymbol{y} = \boldsymbol{y'}) * M_R(\boldsymbol{z}, \boldsymbol{z'}) + \\ &\quad M_R(\boldsymbol{y}, \boldsymbol{y'}) * M_U(\boldsymbol{z}) * (\boldsymbol{z} = \boldsymbol{z'}) + M_U(\boldsymbol{y}) * (\boldsymbol{y} = \boldsymbol{y'}) * M_U(\boldsymbol{z}) * (\boldsymbol{z} = \boldsymbol{z'}) * (\boldsymbol{y} > \boldsymbol{z}) \end{aligned}}$$

$$\frac{T(\boldsymbol{x}) := M(\boldsymbol{y}) * V(\boldsymbol{z}) \quad \boldsymbol{x} = \boldsymbol{y} \cup \boldsymbol{z} \quad \boldsymbol{y} \cap \boldsymbol{z} = \emptyset \quad \boldsymbol{x'} = \boldsymbol{y'} \cup \boldsymbol{z'} \quad \boldsymbol{y'} \cap \boldsymbol{z'} = \emptyset}{\begin{aligned} T_U(\boldsymbol{x}) &:= M_U(\boldsymbol{y}) * V_U(\boldsymbol{z}) \\ T_R(\boldsymbol{x}, \boldsymbol{x'}) &:= M_R(\boldsymbol{y}, \boldsymbol{y'}) * V_R(\boldsymbol{z}, \boldsymbol{z'}) + M_U(\boldsymbol{y}) * (\boldsymbol{y} = \boldsymbol{y'}) * V_R(\boldsymbol{z}, \boldsymbol{z'}) + M_R(\boldsymbol{y}, \boldsymbol{y'}) * V_U(\boldsymbol{z}) * (\boldsymbol{z} = \boldsymbol{z'}) \end{aligned}}$$

$$\frac{T(\boldsymbol{x}) := M(\boldsymbol{x}) + V(\boldsymbol{x}) \quad M_U(\boldsymbol{x}) = V_U(\boldsymbol{x}) \quad M_R(\boldsymbol{x}, \boldsymbol{x'}) = V_R(\boldsymbol{x}, \boldsymbol{x'})}{\begin{aligned} T_U(\boldsymbol{x}) &:= M_U(\boldsymbol{x}) \\ T_R(\boldsymbol{x}, \boldsymbol{x'}) &:= M_R(\boldsymbol{x}, \boldsymbol{x'}) \end{aligned}}$$

$$\frac{T(\boldsymbol{x}) := M(\boldsymbol{x}) + V(\boldsymbol{x}) \quad M_U(\boldsymbol{x}) \cap V_U(\boldsymbol{x}) = \emptyset \quad M_R(\boldsymbol{x}, \boldsymbol{x'}) \cap V_R(\boldsymbol{x}, \boldsymbol{x'}) = \emptyset}{\begin{aligned} T_U(\boldsymbol{x}) &:= M_U(\boldsymbol{x}) + V_U(\boldsymbol{x}) \\ T_R(\boldsymbol{x}, \boldsymbol{x'}) &:= M_R(\boldsymbol{x}, \boldsymbol{x'}) + V_R(\boldsymbol{x}, \boldsymbol{x'}) \end{aligned}}$$

$$\frac{T(\boldsymbol{x}) := V(\boldsymbol{y}) * (\boldsymbol{y} = \boldsymbol{x} - \boldsymbol{d})}{\begin{aligned} T_U(\boldsymbol{x}) &:= V_U(\boldsymbol{y}) * (\boldsymbol{y} = \boldsymbol{x} - \boldsymbol{d}) \\ T_R(\boldsymbol{x}, \boldsymbol{x'}) &:= V_R(\boldsymbol{y}, \boldsymbol{y'}) * (\boldsymbol{y} = \boldsymbol{x} - \boldsymbol{d}) * (\boldsymbol{y'} = \boldsymbol{x'} - \boldsymbol{d}) \end{aligned}} \qquad \frac{T(x) := b \leq x < c}{\begin{aligned} T_U(x) &:= x = b \\ T_R(x, x') &:= (b < x < c) * x' = b \end{aligned}}$$

General Cases:

$$\frac{T(\boldsymbol{x}) := M(\boldsymbol{y}) * V(\boldsymbol{z}) \quad \boldsymbol{x} = \boldsymbol{y} \cup \boldsymbol{z} \quad \boldsymbol{x'} = \boldsymbol{y'} \cup \boldsymbol{z'}}{\begin{aligned} T_U(\boldsymbol{x}) &:= M_U(\boldsymbol{y}) * V_U(\boldsymbol{z}) + M_U(\boldsymbol{y}) * V_R(\boldsymbol{z}, \boldsymbol{z'}) + M_R(\boldsymbol{y}, \boldsymbol{y'}) * V_U(\boldsymbol{z}) \\ T_R(\boldsymbol{x}, \boldsymbol{x'}) &:= M_R(\boldsymbol{y}, \boldsymbol{y'}) * V_R(\boldsymbol{z}, \boldsymbol{z'}) \end{aligned}}$$

$$\frac{T(\boldsymbol{x}) := M(\boldsymbol{x}) + V(\boldsymbol{x})}{\begin{aligned} T_U(\boldsymbol{x}) &:= M_U(\boldsymbol{x}) + M_R(\boldsymbol{x}, \boldsymbol{x'}) + V_U(\boldsymbol{x}) + V_R(\boldsymbol{x}, \boldsymbol{x'}) \\ T_R(\boldsymbol{x}, \boldsymbol{x'}) &:= \emptyset \end{aligned}} \qquad \frac{T(\boldsymbol{x}) := M(\boldsymbol{y}) \quad \boldsymbol{x} \subseteq \boldsymbol{y}}{\begin{aligned} T_U(\boldsymbol{x}) &:= M_U(\boldsymbol{y}) + M_R(\boldsymbol{y}, \boldsymbol{y'}) \\ T_R(\boldsymbol{x}, \boldsymbol{x'}) &:= \emptyset \end{aligned}}$$

Fig. 10. Inference rules for unique sets and redundancy maps. The priority for the rules is top-down. The default case considers $T_R(\boldsymbol{x}, \boldsymbol{x'}) := \emptyset$ and $T_U(x) := \lfloor T(x) \rfloor$ (cf. Section 6). The last three rules subsume the ones shown in Figure 9.

$$\begin{aligned} E_U(i, j, k) &:= (0 \leq i < n) * (0 \leq k < n) * (i = k) * (0 \leq k < n) * (0 \leq j < n) * (k = j) \\ &:= (0 \leq i < n) * (0 \leq j < n) * (0 \leq k < n) * (i = j = k) \\ F_U(i, j) &:= (0 \leq i < n) * (0 \leq j < n) * (0 \leq k < n) * (i = j = k) \\ &:= (0 \leq i < n) * (0 \leq j < n) * (i = j) \\ A_U(i, j) &:= (0 \leq i < n) * (0 \leq j < n) * (i = j) + (0 \leq i < n) * (0 \leq j < n) * (i = j) \\ &:= (0 \leq i < n) * (0 \leq j < n) * (i = j) \end{aligned}$$

The final unique set specifies a diagonal structure, as expected.

Inference for Redundancy Maps. The redundancy map of the output is required to have access to every element in the output tensor and reconstruct it. Figure 10 shows the rules to infer output redundancy map and unique sets. The rules for the tensor outer product, addition, direct sum, and repetition are provided. The cases where there is no rule in Figure 10 are handled as follows. First, the redundancy maps of inputs are set to empty by combining all non-zero elements from the unique set and the redundancy map into the unique set. Then, rules from Figure 9 are applied to the computation to calculate their unique set (cf. Algorithm 1, line 3).

$$\frac{T(\boldsymbol{x}) := M(\boldsymbol{x}) + V(\boldsymbol{x}) \qquad M_U(\boldsymbol{x}) = V_U(\boldsymbol{x}) \quad M_R(\boldsymbol{x}, \boldsymbol{x'}) = V_R(\boldsymbol{x}, \boldsymbol{x'})}{T_C(\boldsymbol{x}) := M_C(\boldsymbol{x}) * M_U(\boldsymbol{x}) + V_C(\boldsymbol{x}) * M_U(\boldsymbol{x})} \qquad \frac{T(\boldsymbol{x}) := M(\boldsymbol{y}) \quad \boldsymbol{x} \subseteq \boldsymbol{y}}{T_C(\boldsymbol{x}) := M_U(\boldsymbol{y}) * M_C(\boldsymbol{y}) + M_R(\boldsymbol{y}, \boldsymbol{y'}) * M_C(\boldsymbol{y'})}$$

Fig. 11. A subset of inference rules for compressed tensors. All compression rules (except the left rule in this figure) can be constructed from the unique set rules (cf. Figure 10). This is achieved by applying the following modifications (as can be seen in the second rule of this figure): 1) replace $M_U(\boldsymbol{x})$ with $M_C(\boldsymbol{x}) * M_U(\boldsymbol{x})$ and 2) replace $M_R(\boldsymbol{x}, \boldsymbol{x'})$ with $M_C(\boldsymbol{x'}) * M_R(\boldsymbol{x}, \boldsymbol{x'})$. The default case considers $T_C(\boldsymbol{x}) := T_U(\boldsymbol{x}) * T(\boldsymbol{x})$.

Tensor Compression. By using the inferred unique sets and redundancy maps, StructTensor computes the compressed tensor (cf. Algorithm 1, line 4). Figure 11 shows a subset of inference rules for compressed tensor computation. The compressed tensor for an input is simply computed by multiplying it by its unique set. The expression corresponding to the reconstruction of tensor $T(\boldsymbol{x})$ from unique elements is created by multiplying $T(\boldsymbol{x'})$ by $T_R(\boldsymbol{x}, \boldsymbol{x'})$ (cf. Algorithm 1, line 6).
Example - Structure Inference. Assume the following definition:

$$T(x_1, x_2, x_3, x_4, x_5) := M(x_1, x_2, x_3) * V(x_4, x_5)$$

Furthermore, consider the dimensions as $(d, d, d) = dims(M)$ and $(n, n) = dims(V)$. Imagine M and V have a diagonal and symmetric structures respectively. Therefore, the unique set, redundancy map, and compressed tensor of inputs are:

$$\begin{aligned}
M_U(x_1, x_2, x_3) &:= (0 \le x_1 < d) * (0 \le x_2 < d) * (0 \le x_3 < d) * (x_1 = x_2 = x_3) \\
V_U(x_4, x_5) &:= (x_4 \le x_5) * (0 \le x_4 < n) * (0 \le x_5 < n) \\
M_R(x_1, x_2, x_3, x_1', x_2', x_3') &:= \emptyset \\
V_R(x_4, x_5, x_4', x_5') &:= (x_4 > x_5) * (0 \le x_4 < n) * (0 \le x_5 < n) * (x_4' = x_5) * (x_5' = x_4) \\
M_C(x_1, x_2, x_3) &:= M(x_1, x_2, x_3) * (0 \le x_1 < d) * (0 \le x_2 < d) * \\
&\quad\ (0 \le x_3 < d) * (x_1 = x_2 = x_3) \\
V_C(x_4, x_5) &:= V(x_4, x_5) * (x_4 \le x_5) * (0 \le x_4 < n) * (0 \le x_5 < n)
\end{aligned}$$

By following the second rule in Figure 10, StructTensor infers the output unique set, redundancy map, and compressed tensor as follows:

$$\begin{aligned}
T_U(x_1, x_2, x_3, x_4, x_5) &:= M_U(x_1, x_2, x_3) * V_U(x_4, x_5) \\
T_R(x_1, x_2, x_3, x_4, x_5, x_1', x_2', x_3', x_4', x_5') &:= M_R(x_1, x_2, x_3, x_1', x_2', x_3') * V_R(x_4, x_5, x_4', x_5') + \\
&\quad\ M_U(x_1, x_2, x_3) * V_R(x_4, x_5, x_4', x_5') * (x_1' = x_1) * (x_2' = x_2) * (x_3' = x_3) + \\
&\quad\ M_R(x_1, x_2, x_3, x_1', x_2', x_3') * V_U(x_4, x_5) * (x_4' = x_4) * (x_5' = x_5) \\
T_C(x_1, x_2, x_3, x_4, x_5) &:= M_C(x_1, x_2, x_3) * M_U(x_1, x_2, x_3) * V_C(x_4, x_5) * V_U(x_4, x_5)
\end{aligned}$$

Sound Reasoning. We focused on our inference rules being sound rather than complete. This means that our approach to the unique set and redundancy map is conservative; thus we consider the elements that might be non-zero after computation as non-zero since we do not know the actual values of tensors in compile time. For example, if tensor $T1$ and $T2$ include values -1 and 1 in their 0-th index, the zero values resulted by $T1 + T2$ in the 0-th index (and possibly, any other index) will be treated as non-zero. Thus, StructTensor will perform computations using these zero values to make sure the computation is correct.
Extensions. The set of rules provided in Figures 9-11 is minimal and meant to be extensible. For example, the rule for self-multiplication can be easily extended to consider higher-degree self-multiplications. The idea is that our rewriting system will always choose the most specialized (and therefore optimizing) rule first.

5.2 Optimizations

Rule Inlining. The intermediate compressed tensors, unique sets, and redundancy maps are materialized. Inlining these definitions can improve performance in various ways. First, one can avoid the materialization overhead. Second, inlining is a transformation enabling opportunities for further optimizations such as common sub-expression elimination. During inlining, the tensor index variable needs to be alpha renamed to avoid capturing free variables (cf. Algorithm 1, line 5).
Logical Simplifications. After inlining, the factors inside unique sets, redundancy maps, and compressed tensors might be repetitive or result in $\emptyset$ or other simple rules. Logical simplification removes repetitive conditions. For instance, $C(x) * C(x) \rightarrow C(x)$ where $C(x)$ is a boolean. Furthermore, there could be contradicting conditions that will result in $\emptyset$ (e.g., $(a \leq b) * (a > b)$ leads to $\emptyset$). $\emptyset$ acts as an absorbing element for multiplication (i.e., $\emptyset * e = \emptyset$), and it is the neutral element for addition (i.e., $\emptyset + e = e$). We use a fixed-point bottom-up traversal strategy to detect and apply them (cf. Algorithm 1, line 7).
Example - Structure Optimization. Continuing the previous example, after inlining, the unique set, redundancy map, and compressed tensor are transformed as follows:

$$\begin{aligned}
T_U(x_1,x_2,x_3,x_4,x_5) \;&:=\; (0 \leq x_1 < d) * (0 \leq x_2 < d) * (0 \leq x_3 < d) * (x_1 = x_2 = x_3) * \\
&\quad (x_4 \leq x_5) * (0 \leq x_4 < n) * (0 \leq x_5 < n) \\
T_R(x_1,x_2,x_3,x_4,x_5, x_1',x_2',x_3',x_4',x_5') \;&:=\; \emptyset * (x_4 > x_5) * (0 \leq x_4 < n) * (0 \leq x_5 < n) * (x_4' = x_5) * (x_5' = x_4) + \\
&\quad (0 \leq x_1 < d) * (0 \leq x_2 < d) * (0 \leq x_3 < d) * (x_1 = x_2 = x_3) * \\
&\quad (x_4 > x_5) * (0 \leq x_4 < n) * (0 \leq x_5 < n) * (x_4' = x_5) * (x_5' = x_4) * \\
&\quad (x_1' = x_1) * (x_2' = x_2) * (x_3' = x_3) + \\
&\quad \emptyset * (x_4 \leq x_5) * (0 \leq x_4 < n) * (0 \leq x_5 < n) * (x_4' = x_4) * (x_5' = x_5) \\
T_C(x_1,x_2,x_3,x_4,x_5) \;&:=\; M(x_1,x_2,x_3) * (0 \leq x_1 < d) * (0 \leq x_2 < d) * (0 \leq x_3 < d) * \\
&\quad (x_1 = x_2 = x_3) * (0 \leq x_1 < d) * (0 \leq x_2 < d) * (0 \leq x_3 < d) * \\
&\quad (x_1 = x_2 = x_3) * V(x_4,x_5) * (x_4 \leq x_5) * (0 \leq x_4 < n) * \\
&\quad (0 \leq x_5 < n) * (x_4 \leq x_5) * (0 \leq x_4 < n) * (0 \leq x_5 < n)
\end{aligned}$$

By further simplifying the redundancy map by propagating the rules for $\emptyset$, we obtain:

$$\begin{aligned}
T_R(x_1,x_2,x_3,x_4,x_5, x_1',x_2',x_3',x_4',x_5') \;&:=\; (0 \leq x_1 < d) * (0 \leq x_2 < d) * (0 \leq x_3 < d) * (x_1 = x_2 = x_3) * \\
&\quad (x_4 > x_5) * (0 \leq x_4 < n) * (0 \leq x_5 < n) * (x_4' = x_5) * (x_5' = x_4) * \\
&\quad (x_1' = x_1) * (x_2' = x_2) * (x_3' = x_3)
\end{aligned}$$

T is diagonal on the first three dimensions and symmetric on the last two of them, as expected. Also, by further simplifying the compressed tensor by propagating $C(x) * C(x) \longrightarrow C(X)$, we obtain:

$$\begin{aligned}
T_C(x_1,x_2,x_3,x_4,x_5) \;&:=\; M(x_1,x_2,x_3) * (0 \leq x_1 < d) * (0 \leq x_2 < d) * (0 \leq x_3 < d) * \\
&\quad (x_1 = x_2 = x_3) * V(x_4,x_5) * (x_4 \leq x_5) * (0 \leq x_4 < n) * (0 \leq x_5 < n)
\end{aligned}$$

5.3 Use Case 1: Structured Linear Algebra

In this section, we show that StructTensor can recover the output structure of linear algebra operations by using the inference and optimization rules presented earlier.
Example - Upper Triangular Hadamard Product by Symmetric (UHS). Assume two square matrices M and N with dimension $n \times n$ and with upper triangular and symmetric structures, respectively. The element-wise multiplication is represented in STUR as follows:

$$A(i,j) := M(i,j) * N(i,j)$$

The unique sets and redundancy maps of the input matrices are represented as follows:

$$\begin{aligned}
M_U(i,j) &:= (0 \leq i \leq j < n) \\
N_U(i,j) &:= (0 \leq i \leq j < n) \\
M_R(i,j,i',j') &:= \emptyset \\
N_R(i,j,i',j') &:= (0 \leq j < i < n) * (i' = j) * (j' = i)
\end{aligned}$$

The output unique set will be:

$$\begin{array}{rcl} A_U(i,j) & := & M_U(i,j) * N_U(i,j) + M_U(i,j) * N_R(i,j,i',j') + M_R(i,j,i',j') * N_U(i,j) \\ \text{(inlining)} & := & \big((0 \le i \le j < n) * (0 \le i \le j < n)\big) \\ & & +\big((0 \le i \le j < n) * (0 \le j < i < n) * (i' = j) * (j' = i)\big) \\ & & +\big(\emptyset * (0 \le i \le j < n)\big) \\ \text{(simplification)} & := & (0 \le i \le j < n) + \emptyset + \emptyset \\ & := & (0 \le i \le j < n) \end{array}$$

Example - Row Matrix Multiplied by Diagonal (RMD). Consider an $m \times n$ row matrix M, which only has elements in its r-th row, that is being multiplied by an $n \times n$ diagonal matrix N. The computation is expressed as:

$$A(i,j) := M(i,k) * N(k,j)$$

The unique set and redundancy maps are as follows:

$$\begin{array}{rcl} M_U(i,k) & := & (i = r) * (0 \le k < n), \quad M_R(i,k,i',k') := \emptyset \\ N_U(k,j) & := & (k = j) * (0 \le j < n), \quad N_R(k,j,k',j') := \emptyset \end{array}$$

Since there is no redundancy map, the rules of Figure 9 are enough to infer the output unique set. The output redundancy map is naturally $\emptyset$. Therefore, the output unique set is:

$$\begin{array}{rcl} A_U(i,j) & := & M_U(i,k) * N_U(k,j) \\ \text{(inlining)} & := & (i = r) * (0 \le k < n) * (k = j) * (0 \le j < n) \\ \text{(simplification)} & := & (i = r) * (0 \le j < n) * (0 \le j < n) \\ \text{(simplification)} & := & (i = r) * (0 \le j < n) \end{array}$$

As it is shown in Figure 8, this unique set corresponds to a row matrix.

5.4 Use Case 2: Structured Tensor Algebra

StructTensor uses inference rules in STUR and produces an optimized code for tensor algebra. Similar to the case of linear algebra, a programmer provides the tensor algebra program as input alongside the structure for input tensors. As there are very many possibilities for the structure of higher-order tensors, we expect the programmer to provide them in STUR.

Example - Diagonal Tensor Times Vector (DTTV). This example shows how the output structure for TTV (Tensor Times Vector) is inferred in STUR. TTV is defined in STUR as follows:

$$A(i,j) := M(i,j,k) * N(k)$$

Consider the unique set and redundancy map of the inputs to be as follows:

$$\begin{array}{rcl} M_U(i,j,k) & := & (i = j) * (j = k) * (0 \le i < m) * (0 \le j < m) * (0 \le k < m) \\ M_R(i,j,k,i',j',k') & := & \emptyset, \quad V_U(k) := (0 \le k < m), \quad V_R(k,k') := \emptyset \end{array}$$

where the dimensions are $(m,m,m) = dims(M)$ and $(m) = dims(V)$. Computation is divided into two steps, multiplication followed by reduction.

$$\begin{array}{rcl} B(i,j,k) & := & M(i,j,k) * N(k) \\ A(i,j) & := & B(i,j,k) \end{array}$$

Therefore, the output unique set should be calculated in two steps as well.

$$\begin{array}{rcl} B_U(i,j,k) & := & M_U(i,j,k) * N_U(k) \\ \text{(inlining)} & := & (i = j) * (j = k) * (0 \le i < m) * (0 \le j < m) * (0 \le k < m) * (0 \le k < m) \\ \text{(simplification)} & := & (i = j) * (j = k) * (0 \le i < m) \\ A_U(i,j) & := & B_U(i,j,k) \\ \text{(inlining)} & := & (i = j) * (j = k) * (0 \le i < m) \\ \text{(simplification)} & := & (i = j) * (0 \le i < m) \end{array}$$

Algorithm 2 Code Generation Algorithm

```
1:  function CODE_GEN(expr, flags)
2:      code ← ""
3:      for factor in expr.body do                          ▷ expr.body is a sum of products
4:          vars ← variables(factor)
5:          for var in vars do                               ▷ loop bounds generation
6:              lower, upper ← GET_BOUNDS(factor, var)
7:              if lower == upper then
8:                  GEN_IF_EQUAL(code, factor, var, lower, flags)
9:              else
10:                 GEN_LOOP(code, factor, var, lower, upper, flags)
11:             end if
12:         end for
13:         te_factor ← EXTRACT_COMPUTATION(factor)
14:         GEN_ADDITION(code, te_factor, flags)
15:     end for
16:     return code
17: end function
```

5.5 Code Generation

As the final step, StructTensor generates C++ code for the optimized STUR expression (cf. Algorithm 1, line 8). Algorithm 2 provides the pseudocode for code generation. To generate loop nests, first, the index variables are ordered following the same syntactic order as the input tensor expression[2] (cf. Algorithm 2, line 4). Afterwards, StructTensor computes the range of each index variable based on the output compressed tensor. The range for each index variable is computed as the maximum of all its lower bounds and the minimum of all its upper bounds provided as logical expressions in the unique set (cf. Algorithm 2, line 6).

Knowing the lower and upper bound of variables is enough to generate the loop nests. If the lower and upper bounds are the same, instead of a loop from lower to upper, only an if condition will be generated. StructTensor expects the bounds for inputs; an expression without input bounds is invalid in STUR. (cf. Algorithm 2, lines 7-11). The last step is the code generation for the input tensor expression. The code generator iterates over the compressed tensor body, which is a sum of product factors (named as $factor$). Each factor includes a series of tensors multiplied together (named as te_factor, line 13). In the deepest loop nest, the value of the result is updated by the value of the tensor expression appearing in the te_factor (cf. Algorithm 2, line 14).

Algorithm 2 focuses on generating code based on a tensor expression. Therefore, this algorithm is useful for both tensor compression and reconstruction. To reconstruct the final result, we pass the reconstruction tensor expression $T(\boldsymbol{x}) := T(\boldsymbol{x}') * T_R(\boldsymbol{x}, \boldsymbol{x}')$ (cf. Algorithm 1, line 6) to the code generator. In order to avoid redundant computations, we do not inline the reconstruction expression; we only inline the compressed tensor and redundancy map (cf. Algorithm 1, line 5).

StructTensor hoists loop-invariant expressions outside the loops whenever possible. Moreover, input data layout can be passed as a mapping from the dimension of the tensor to a lower dimensional space to StructTensor to reduce memory consumption. This information is provided in $flags$ to the code generation algorithm. However, there is no data-layout optimization applied on the output [Chou et al. 2018; Kandemir et al. 1999; Schleich et al. 2023]; an n dimensional output

[2] The optimal variable ordering is an NP-complete problem that is beyond the scope of this work.

```
for(int i = 0; i < n; ++i) {
  auto y_i = y[i];
  auto x_i = x[i];
  for(int j = i; j < n; ++j) {
    y_i[j] = x_i * x[j];
  }
}
```

```
for(int i = 0; i < n; ++i) {
  int ip = j;
  auto y_i = y[i];
  auto y_ip = y[ip];
  for(int j = 0; j < i; ++j) {
    int jp = i;
    y_i[j] = y_ip[jp];
  }
}
```

Fig. 12. Code snippets for linear regression covariance matrix creation: compressed computation (left), result matrix reconstruction (right). Note that the code motion is enabled.

tensor is stored in an n dimensional C++ array. Further optimizations such as multithreading and vectorization are also left for the future.

Example - Vector Self-Outer-Product. The outer product of a vector with size n by itself in STUR is defined as follows.

$$y(i,j) := x(i) * x(j) \qquad x_U(i) := 0 \le i < n \qquad x_R(i,i') := \emptyset \qquad x_C(i) := x(i) * (0 \le i < n)$$

Based on Figure 10 and Figure 11, the output structure is:

$$\begin{aligned}
y_U(i,j) &:= (0 \le i < n) * (0 \le j < n) * (i \le j) \\
&:= 0 \le i \le j < n \\
y_R(i,j,i',j') &:= (0 \le i < n) * (0 \le j < n) * (i > j) * (i' = i) * (j' = j) \\
&:= (0 \le j < i < n) * (i' = i) * (j' = j) \\
y_C(i,j) &:= x(i) * (0 \le i < n) * x(j) * (0 \le j < n) * (0 \le i < n) * (0 \le j < n) * (i \le j) \\
&:= x(i) * x(j) * (0 \le i \le j < n)
\end{aligned}$$

The structure is fed to the code generator, and compressed computational loop nests are generated based on the compressed tensor of y. Since all the elements in the matrix y should be accessible in the end, the reconstruction code is generated based on the redundancy map of the output. Figure 12 shows the computation and reconstruction code snippets for this program.

5.6 Frontend

StructTensor provides two frontends for the users. The first frontend allows users to perform linear algebra operations defined in Figure 7 using predefined structures provided in Figure 8. Examples provided in Section 5.3 use this frontend. The second frontend allows the user to generate code for arbitrary sum-of-products (SoP) tensor computations given an arbitrary SoP unique set and redundancy structure, following the STUR grammar. The example provided in Section 5.4 utilizes this frontend.

To be more specific, StructTensor requires the following information as input for STUR:

- Tensor computation expression (required)
- Dimension information of input tensors (required): The size of each dimension of the tensor, without which, code generation is impossible.
- Unique set of input tensors (optional): Bounds over all unique non-zero elements of the tensor should be provided in the unique set. If the unique set is not provided, we assume that the tensor is fully dense.
- Redundancy map of input tensors (optional): The mapping from repetitive non-zero values to their unique corresponding values should be provided in the redundancy map. If the redundancy map is not provided, we assume that there is no redundancy structure.

StructTensor uses STUR as its frontend language. The tensor computation expression, unique sets, redundancy maps, and compressed tensors are expressed in this language. The dimension information is provided as a list of symbols rather than fixed static values, as StructTensor is not

$$M2(i,col) := f(i) * f(j) * f(k) * (col = j * n + k)$$

$$f_U(i) := 0 \leq i < n$$

(a) Input computation and unique set.

$$M2_U(i,col) := f_U(i) * f_U(j) * f_U(k) * (col = j * n + k) * (0 \leq i \leq j \leq k < n)$$

$$M2_C(i,col) := f_C(i) * f_U(i) * f_C(j) * f_U(j) * f_C(k) * f_U(k) * (col = j * n + k) * (0 \leq i \leq j \leq k < n)$$

(b) Unoptimized unique set and compressed tensor.

$$M2_C(i,col) := (0 \leq i < n) * (i \leq j < n) * (j \leq k < n) * (col = j * n + k) * f(i) * f(j) * f(k)$$

(c) Optimized compressed tensor.

```
for(int i=0; i<n; ++i){
 for(int j=i; j<n; ++j){
  for(int k=j; k<n; ++k){
    int col = j * n + k;
    M2[i][col] = f[i] * f[j] * f[k];
}}}
```

(d) Generated C++ code.

Fig. 13. Step-by-step code generation procedure from inputs for $M2$ in covariance matrix creation.

limited to fixed-sized tensors as opposed to frameworks such as LGen [Spampinato and Püschel 2016] and EGGS [Tang et al. 2020b]. All intermediate and output tensors' structures are inferred following our program reasoning rules using the mentioned input information.

5.7 Putting It All Together

We provide a step-by-step elaboration on inference and code generation of the motivating example (Section 3). We only show the procedure for $M2$ where $M2 = \mathbf{f} \otimes vec(\mathbf{f} \otimes \mathbf{f})$; $M1$ and $M3$ are handled similarly. First, the representation of $M2$ as well as inputs structure is provided to STUR (cf. Figure 13a). Then, STUR generates the unoptimized unique set and compressed tensor for $M2$ computation (cf. Figure 13b) by following the generalized version of the first rule (self-outer product of vectors) in Figure 10. After optimization, the representation in Figure 13c is obtained.

As mentioned in Section 4, all tensor expressions in STUR are provided as a sum of products. $M2_C(i, col)$ only has one product inside its body. The code generator reads the body and extracts all the variables inside it (i, j, k, and col). Then, the code generator proceeds to detect the boundaries for each variable, which are as follows: $i \rightarrow (0, n)$, $j \rightarrow (i, n)$, $k \rightarrow (j, n)$, and $col \rightarrow (j*n+k, j*n+k+1)$. Here $a \rightarrow (lower, upper)$ means that $lower \leq a < upper$. As the next step, loops, if conditions, or assignments will be generated based on the detected boundaries. This will result in the generation of the three-level nested loops (cf. Figure 13d, white background). The last boundary (for variable `col`) is of the form $upper = lower + 1$, which corresponds to an assignment (cf. Figure 13d, light gray background). Finally, the computation is generated for the expression (cf. Figure 13d, dark gray background), which results in the code provided in Section 3 and Figure 13d is generated.

6 SOUNDNESS OF STRUCTURE INFERENCE

We present a simple and more mathematical view of what unique sets and redundancy maps compute. We then derive several properties that they satisfy. Furthermore, we state a soundness theorem for our structure inference, formally anchoring the fact that they do not lose information and compute the same as the original tensor.

6.1 Abstract View on the Tensor Compression Problem

We can view the problem of representing a tensor in a compressed way as follows. Let $\mathcal{T}_{n_1,\ldots,n_k}(\mathbb{R})$ be the real vector-space of $n_1 \times \ldots \times n_k$ real tensors. Given a fixed $T \in \mathcal{T}_{n_1,\ldots,n_k}(\mathbb{R})$, the problem is finding a pair of linear maps $(R_T, P_T) : \mathcal{T}_{n_1,\ldots,n_k}(\mathbb{R}) \rightarrow \mathcal{T}_{n_1,\ldots,n_k}(\mathbb{R})$ such that the following simple equation holds:

$$R_T \circ P_T(T) = T \tag{3}$$

We call such a pair (R_T, P_T) a solution to the reduction problem for T. The intuition is that P_T represents a projection, $P_T(T)$ represents a compressed version of T, and R_T ensures that this compression is lossless. Note that there are many solutions to this problem, e.g., P_T could simply be the identity, and we are naturally interested in solutions that approximately minimize the number of non-zeros elements of the compressed tensor $P_T(T)$. Finding optimal solutions is also feasible but would make us lose time overall for the final computations we are interested in.

Note that, as P_T and R_T are linear, they can be represented by $n_1 \times \ldots \times n_k \times n_1 \times \ldots \times n_k$ tensors. In this work, we will restrict to the cases where P_T, R_T can be represented with tensors valued in the Boolean semiring $(\{0,1\}, +, \times)$ where + is max and $\times$ is min. Both satisfy that, when seen as matrices, the sum of the elements on each row is at most 1. With the correspondence between $\{0,1\}$-valued matrices and relations between finite sets, such matrices represent partial (the sum of the elements on a column can be 0) injective (the sum of the elements on a row is at most one) relations ({0,1}-valued matrix). We further impose P_T to be a projection. In such a case, P_T will be an *orthogonal* projection and will verify $P_T \circ P_T = P_T$ (an orthogonal projection is automatically a partial injective relation). Any partial injective relation F is a morphism and will therefore satisfy $F(A+B) = F(A) + F(B)$. It will satisfy the additional nice property that $F(A \odot B) = F(A) \odot F(B)$ for all A, B, where $\odot$ is the Hadamard product. In fact, partial injective relations are characterized as those $\{0,1\}$-valued matrices F such that $F(A \odot B) = F(A) \odot F(B)$ for all vectors A, B.

We denote by $\lfloor - \rfloor : \mathcal{T}_{n_1,\ldots,n_k}(\mathbb{R}) \to \mathcal{T}_{n_1,\ldots,n_k}(\mathbb{R})$ the function sending a real r at position $(i_1, \ldots, i_k)$ to 1 at the same position if $r \neq 0$, and to 0 otherwise. Note that this extends a monoid homomorphism $(\mathbb{R}, \times, 1) \to (\{0,1\}, \times, 1)$ but this is not a linear transformation as it does not commute with addition, i.e. $\lfloor A + B \rfloor \neq \lfloor A \rfloor + \lfloor B \rfloor$ in general. It still implies that $\lfloor T \otimes S \rfloor = \lfloor T \rfloor \otimes \lfloor S \rfloor$, $\lfloor T \oplus S \rfloor = \lfloor T \rfloor \oplus \lfloor S \rfloor$ and $\lfloor T \odot S \rfloor = \lfloor T \rfloor \odot \lfloor S \rfloor$, where $\otimes$ is the Kronecker product, and $\oplus$ the direct sum. Additionally, for any partial injective relation F and any A, we have $\lfloor F(A) \rfloor = F(\lfloor A \rfloor)$.

We indistinguishably use tensors and the linear maps they represent, assuming fixed bases of the underlying vector spaces at play. From the abstract setting, the tensors T_U, T_R, T_C we infer for a tensor T, and use in our unified IR, are derived as follows:

$$T_U := \lfloor P_T(T) \rfloor \qquad T_R := R_T - P_T \qquad T_C := P_T(T) \tag{4}$$

In other words, T_U is the support of the compressed tensor $T_C = P_T(T)$, and T_R is, up to a small optimization, the support of the tensor representation of R_T. From Equation 3 and the definitions of T_U, T_R, T_C (Equation 4), we obtain several properties that are key in reconstructing T, and useful for optimizations.

Proposition. The following properties are valid equations, where $\boldsymbol{x}$ is the list of free variables:

(1) $T_U(\boldsymbol{x}) * T_R(\boldsymbol{x}, \boldsymbol{x'}) = \emptyset$. This optimization ensures that unique elements should not be mapped to any other element by T_R.
(2) $T_C(\boldsymbol{x}) = T_U(\boldsymbol{x}) * T(\boldsymbol{x})$ is the equation defining the compressed tensor in our IR.
(3) $T_C(\boldsymbol{x}) * T_R(\boldsymbol{x}, \boldsymbol{x'}) = \emptyset$. Elements that are stored in the compressed format should only exist in the unique set.
(4) $T_U(\boldsymbol{x}) * T_U(\boldsymbol{x}) = T_U(\boldsymbol{x})$. This is another simple optimization.
(5) $T(\boldsymbol{x}) = T_C(\boldsymbol{x}) + T_R(\boldsymbol{x}, \boldsymbol{x'}) * T_C(\boldsymbol{x'})$. This is the reconstruction process of the tensor T given its compressed format T_C, its unique set T_U, and its redundancy map T_R. This captures the key lossless property of the contraction.

Proof Sketch. Note the following extra elementary properties that we will use in the proofs. $A \odot \lfloor A \rfloor = A$ for any A and $P(A) \odot B = A \odot P(B) = P(A) \odot P(B)$ for any A, B and orthogonal projection P (which is a restatement of a well-known characterization of orthogonal projections in terms of inner products).

(2)

$$
\begin{aligned}
T \odot T_U &= T \odot \lfloor P_T(T) \rfloor && \text{definition of } T_U \\
&= T \odot P_T(\lfloor T \rfloor) && \text{property of partial injective relation } P_T \\
&= P_T(T) \odot P_T(\lfloor T \rfloor) && \text{Property of orthogonal projection } P_T \\
&= P_T(T \odot \lfloor T \rfloor) && \text{property of partial injective relation } P_T \\
&= P_T(T) = T_C && \text{fundamental property of } \lfloor - \rfloor \text{, definition of } T_C
\end{aligned}
$$

(3) is easily obtained from (1) and the fact that T_R commutes with $\lfloor - \rfloor$.

(4)
$$
\begin{aligned}
T_U \odot T_U &= \lfloor P_T(T) \rfloor \odot \lfloor P_T(T) \rfloor && \text{definition of } T_U \\
&= P_T(\lfloor T \rfloor) \odot P_T(\lfloor T \rfloor) && \text{property of partial injective relation } P_T \\
&= P_T(\lfloor T \rfloor \odot \lfloor T \rfloor) && \text{property of partial injective relation } P_T \\
&= P_T(\lfloor T \rfloor) = T_U && \text{property of } \lfloor - \rfloor \text{, definition of } T_U
\end{aligned}
$$

(5)
$$
\begin{aligned}
T_C + T_R(T_C) &= P_T(T) + (R_T - P_T)(P_T(T)) && \text{definition of } T_R, T_C \\
&= P_T(T) + R_T P_T(T) - P_T P_T(T) && \text{linearity} \\
&= P_T(T) + T - P_T(T) = T && \text{Equation 3, } P_T P_T = P_T \text{, simplification}
\end{aligned}
$$

6.2 Soundness Results

Given a tensor T, we write $\mathbb{I}_T : (R_T, P_T) \to (T_U, T_R, T_C)$ for the mapping sending a pair of linear maps verifying Equation 3 to the tensors defined by Equation 4. We say that (T_U, T_R, T_C) implements the solution (R_T, P_T) to the reduction problem for T.

In this work, we never explicitly construct P_T and R_T, but we do inductively define implementations (T_U, T_R, T_C) of solutions to the reduction problem for T, by induction on the structure of T. The first natural question is therefore whether the inferred tuple (T_U, T_R, T_C) is sound, that is whether it is obtained as the implementation of a solution for the reduction problem for T. This is a sufficient condition, as by Property 5 above we can reconstruct T from such a tuple (T_U, T_C, T_R).

Theorem. Let M, N be tensors. Assume given pairs (R_M, P_M) and (R_N, P_N) that are solutions to the reduction problem for M and N, respectively. Let (M_U, M_R, M_C) and (N_U, N_R, N_C) be their respective implementations. Further, assume that T is given by the premise of an inference rule from Figure 10. Then, there exists a solution (R_T, P_T) for the reduction problem for T such that its implementation (T_U, T_R, T_C) is given by the conclusion of the inference rule corresponding to T's definition.

Proof Sketch. We sketch the proof for some of the important cases. For the second rule, let $P_T := P_M \otimes P_V$ and $R_T := R_M \otimes R_T$. Then $R_T P_T(T) = (R_M \otimes R_T)(P_M \otimes P_T)(M \otimes V) = R_M P_M(M) \otimes R_V P_V(V) = M \otimes V = T$. In addition, $\lfloor P_T(T) \rfloor = \lfloor P_M(M) \otimes P_V(V) \rfloor = \lfloor P_M(M) \rfloor \otimes \lfloor P_V(V) \rfloor = M_U \otimes V_U = T_U$. Finally,

$$
\begin{aligned}
R_T - P_T &= R_M \otimes R_V - P_M \otimes P_V \\
&= (M_R + P_M) \otimes (V_R + P_V) - P_M \otimes P_V \\
&= M_R \otimes V_R + P_M \otimes V_R + M_R \otimes P_V
\end{aligned}
$$

Now, note that we define P_M to be the orthogonal projections onto M_U, i.e. P_M can be represented as $P_M = M_U \odot (-)$. This can be represented in STUR as $M_U(z) * (z = z')$. The first rule is proved from the second by noting that $(M \otimes M)_{im+i', jn+j'} = M_{i,j} * M_{i',j'} = M_{i',j'} * M_{i,j} = (M \otimes M)_{i'm+i, j'n+j}$ when M is an $m \times n$ matrix. This immediately generalizes to the case where M is an arbitrary tensor. The third rule is straightforward. For the fourth rule, we have $T = M \oplus V$. Let $R_T := R_M \oplus R_V$ and $P_T := P_M \oplus P_V$. The remainder of the proof is the same as for rule 2, where we replace $\otimes$ by $\oplus$. □

As a direct corollary of the theorem above, we obtain the soundness of our structure inference.

Table 1. Tensor kernels for StructTensor evaluation. In MTTKRP, j is fixed for tensor D. TACO allows specifying if each dimension is Sparse (S) or Dense (D). TACO (Smart) is the most efficient storage format.

Kernel	Structure of B	B_U in STUR	TACO (Smart)
TTM	Diagonal (plane)	$(0 \le i < n_i) * (i = j) * (0 \le l < n_l)$	$(D, S, D) * (D, D) \to (D, S, D)$
$A(i, j, k) :=$	Fixed j	$(0 \le i < n_i) * (j = J) * (0 \le l < n_l)$	$(D, S, D) * (D, D) \to (D, S, D)$
$B(i, j, l) * C(k, l)$	Upper half cube	$(0 \le i < n_i) * (i \le j < n_j) * (0 \le l < n_l)$	$(D, S, D) * (D, D) \to (D, S, D)$
THP	Diagonal (plane)	$(0 \le i < n_i) * (i = j) * (0 \le l < n_l)$	$(D, S, D) * (D, D, D) \to (D, S, D)$
$A(i, j, k) :=$	Fixed i	$(i = I) * (0 \le j < n_j) * (0 \le l < n_l)$	$(S, D, D) * (D, D, D) \to (S, D, D)$
$B(i, j, k) * C(i, j, k)$	Fixed j	$(0 \le i < n_i) * (j = J) * (0 \le l < n_l)$	$(D, S, D) * (D, D, D) \to (D, S, D)$
MTTKRP	Fixed i, j	$(i = I) * (0 \le k < n_k) * (0 \le l < n_l)$	$(S, D, D) * (D, D) * (D, S) \to (S, D)$
$A(i, j) := B(i, k, l)$	Fixed i	$(i = I) * (0 \le k < n_k) * (0 \le l < n_l)$	$(S, D, D) * (D, D) * (D, D) \to (S, D)$
$* C(k, j) * D(l, j)$	Fixed j	$(0 \le i < n_i) * (0 \le k < n_k) * (0 \le l < n_l)$	$(D, D, D) * (D, D) * (D, S) \to (D, S)$
SpMV	Leslie	$(i{=}0) * (0{\le}j{<}n_j) + (1{\le}i{<}n_i) * (j{=}i - 1)$	$(D, S) * (D) \to (D)$
$A(i) :=$	Upper triangular	$(0 \le i < n_i) * (i \le j < n_j)$	$(D, S) * (D) \to (D)$
$B(i, j) * C(j)$	Diagonal	$(0 \le i < n_i) * (i = j)$	$(D, S) * (D) \to (D)$

Corollary [Soundness of Inference]. Let $T := f(M, V)$ be given by a premise of an inference rule from Figure 10. Assume Property (5) holds for M, V. Then, Property (5) holds for T, where T_U, T_R are computed by the conclusion of the same inference rule.

A fundamental optimization is inlining definitions, which is sound in our language.

Proposition [Substitution Lemma]. Let $T_1(\boldsymbol{x}) := B_1(\boldsymbol{x}, \boldsymbol{x'})$ and $T_2(\boldsymbol{y}) := B_2(\boldsymbol{y}, \boldsymbol{y'})$. Then $T_2(\boldsymbol{x}) := B_2[B_1/T_1](\boldsymbol{x}, \boldsymbol{x'}, \boldsymbol{y'})$ is semantically valid whenever $\boldsymbol{x'} \cap \boldsymbol{y'} = \boldsymbol{x} \cap \boldsymbol{y'} = \emptyset$. Note that a substitution such as $T * S[T_1 + T_2/S] = T * (T_1 + T_2)$ introduces temporary parentheses, and the code renormalizes the term to $T * T_1 + T * T_2$.

7 EXPERIMENTAL RESULTS

In this section, we experimentally evaluate StructTensor by considering several tensor processing kernels over different tensor structures. We focus on structured tensor computations targeting machine learning applications on a large volume of data. We study the following questions:

- How advantageous is symbolic sparsity over dynamic sparsity for tensors and matrices?
- Does StructTensor perform better than the state-of-the-art tensor processing frameworks for real-world structured computations after leveraging the redundancy structure?
- How do the optimizations benefit StructTensor?
- Is it worthwhile to perform the computation over a compressed tensor and then decompress the result in real-world applications?

7.1 Experimental Setup

We use a server with 18.04.5 LTS Ubuntu equipped with 10 cores 2.2 GHz Intel Xeon Silver 4210 CPU, and 220 GB of main memory. C++ code is compiled with GCC 7.5.0 using C++17 and -O3, -pthread, -mavx2, -ffast-math, and -ftree-vectorize flags. All experiments are run on a single thread, and the average run time of five runs is reported. As the competitors, we use the latest version of TACO[3] [Kjolstad et al. 2017], Numpy 1.23.4 [Harris et al. 2020], PyTorch 1.12.1 [Paszke et al. 2019], and TensorFlow 2.10.0 [Abadi et al. 2015] with and without the XLA backend.

7.2 Sparsity Structure

To evaluate the effectiveness of StructTensor for the sparsity structure, we consider four tensor and matrix kernels: Tensor Times Matrix (TTM), Tensor Hadamard Product (THP), Matricized Tensor Times Khatri-RAllo Product (MTTKRP), and Sparse Matrix-Vector Multiplication (SpMV). All of the selected kernels play an important role in real-world computation; for example, population growth can be modeled by multiplying a Leslie structured matrix by an age vector (cf. Figure 14).

[3] https://github.com/tensor-compiler/taco/tree/2b8ece4c230a5f

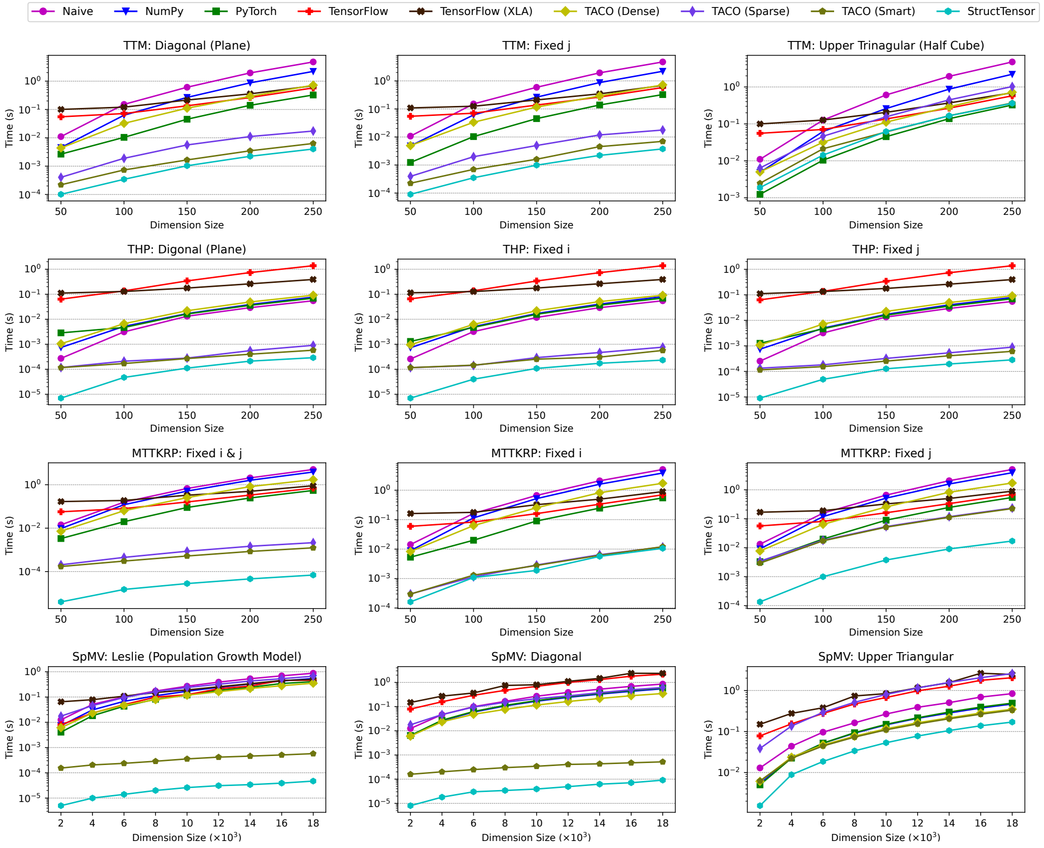

Fig. 14. Tensor operation kernels run time comparison using different sparsity structures. The naïve implementation and NumPy show similar performance. In most cases, StructTensor shows significantly better performance in comparison with all other competitors, including the best sparse format used in TACO.

StructTensor is evaluated against NumPy, PyTorch, TensorFlow (w/ and w/o the XLA backend), and TACO with three different formats: fully dense, fully sparse, and smart. In the smart version, we use the most efficient sparse format for TACO based on the sparsity of input and output tensors at each dimension. Table 1 shows the definition of these kernels, the different input structures we considered, their representation in STUR, and the data format at each dimension for TACO (Smart).[4] Additionally, we consider a naïve version that does not leverage the symbolic structure. Note that TACO is the only framework that features an appropriate sparse data layout. The remaining frameworks are only suitable for dense computation and do not perform as well on sparse kernels. Therefore, we focus on comparing our framework to TACO.

Figure 14 represents the run time of each implementation on each kernel. In all kernels i) the naïve implementation performs similar to NumPy, ii) TACO dense outperforms the naïve implementation and NumPy by a small margin, iii) TACO smart is outperforming all other competitors as well as other data format selections for TACO as expected, and iv) StructTensor outperforms (in 10 out of 12 experiments) or performs on par with TACO smart (despite generating a more optimized

[4]The output data format in MTTKRP for both dimensions in TACO (Smart) for the case of fixed i, j and all cases of TACO (Sparse) should be (S, S). However, because of a bug in TACO (GitHub Issue #518), we had to use (D, S) or (S, D) instead.

code). In all of these experiments, an efficient input data layout that only keeps unique elements is provided for inputs, and code motion is applied for StructTensor.

A major advantage of StructTensor over TACO is the compilation-time knowledge of the sparsity patterns, while TACO needs to handle it at runtime. Therefore, the generated code by StructTensor has direct access to tensor elements, while TACO uses indirect access. Furthermore, having direct access can activate the compiler's auto-vectorization and enhance the computation further for StructTensor, while this is absent for TACO. We confirmed the application of auto-vectorization of the kernels shown in Figure 14 by 1) using GCC auto-vectorization detection flags (e.g., `-fopt-info-vec-all`) and 2) confirming the usage of AVX instructions (e.g., `vmulsd`, `vpaddd`, etc.) in the generated assembly code. As an example, consider the THP kernel, where B is a diagonal on the first two dimensions ($i = j$ in $B(i, j, k)$). The generated code by StructTensor compared to the code generated by TACO for the smart version (that uses (dense, sparse, dense) for the tensor B and the output tensor) is as follows:

```
// StructTensor generated code


for(int i=0; i<min(sizeI, sizeJ); ++i){
 auto &cm1 = A[i];
 auto &cm2 = C[i];
 auto &cm3 = cm1[i]; // j = i;
 auto &cm4 = cm2[i]; // j = i;
 for (int k=0; k<sizeK; ++k)
  cm3[k] += (B[i*sizeK+k] * cm4[k]);
}
```

```
// TACO generated code (smart format)
int32_t jA = 0;
for(int32_t i=0; i<sizeI; i++){
 for(int32_t jB=pos[i]; jB<pos[i+1]; jB++){
  int32_t j = crd[jB];
  int32_t jC = i * sizeI + j;
  for (int32_t k=0; k<sizeK; k++) {
   int32_t kA = jA * sizeK + k;
   int32_t kB = jB * sizeK + k;
   int32_t kC = jC * sizeK + k;
   A_vals[kA] = B_vals[kB] * C_vals[kC];
  }
  jA++;
}}
```

As shown in the generated code, StructTensor directly accesses the tensor elements. However, TACO retrieves the sparse elements using their position (`pos`) and coordinate (`crd`) mappings.

7.3 Redundancy Structure

We consider 7 structured matrix and tensor kernels that correspond to the tasks of one-dimensional full convolution of a kernel of size 16 with a signal, creating and addition of covariance matrices for three machine learning models: linear regression, polynomial regression degree-2, and polynomial regression degree-3. The computation of the convolution kernel is modeled as sparse-matrix vector multiplication (SpMV) of a Toeplitz structured matrix with a dense vector. As the competitors, we consider TACO (fully dense, fully sparse, and smart formats), PyTorch, TensorFlow (w/ and w/o the XLA backend), NumPy, and a naïve implementation without leveraging the structure. We also evaluate the built-in kernel of Python frameworks for the one-dimensional full convolution kernel.

Figure 15 shows the run time comparison. The first two plots illustrate the performance comparison of one-dimensional full convolution using different frameworks with both 1) built-in convolution functions (if exist) and 2) implementing it as a SpMV of Toeplitz structured matrix with the signal. The Toeplitz structure used for the kernel contains both sparsity and redundancy structures. StructTensor outperforms all other frameworks significantly in the SpMV implementation. The second best performance is for TACO in the smart format using a (dense, sparse) layout and the vector using a dense layout. Leveraging existing sparsity in the Toeplitz matrix enhances TACO's performance. Moreover, StructTensor outperforms the built-in convolution functions of all Python frameworks. Crucially, it is competitive with the built-in convolution function provided by the Numpy library, which has a hand-tuned implementation.

The next six plots in Figure 15 illustrate the run time of different regression models using various frameworks. The inputs to these kernels are dense vectors corresponding to continuous features.

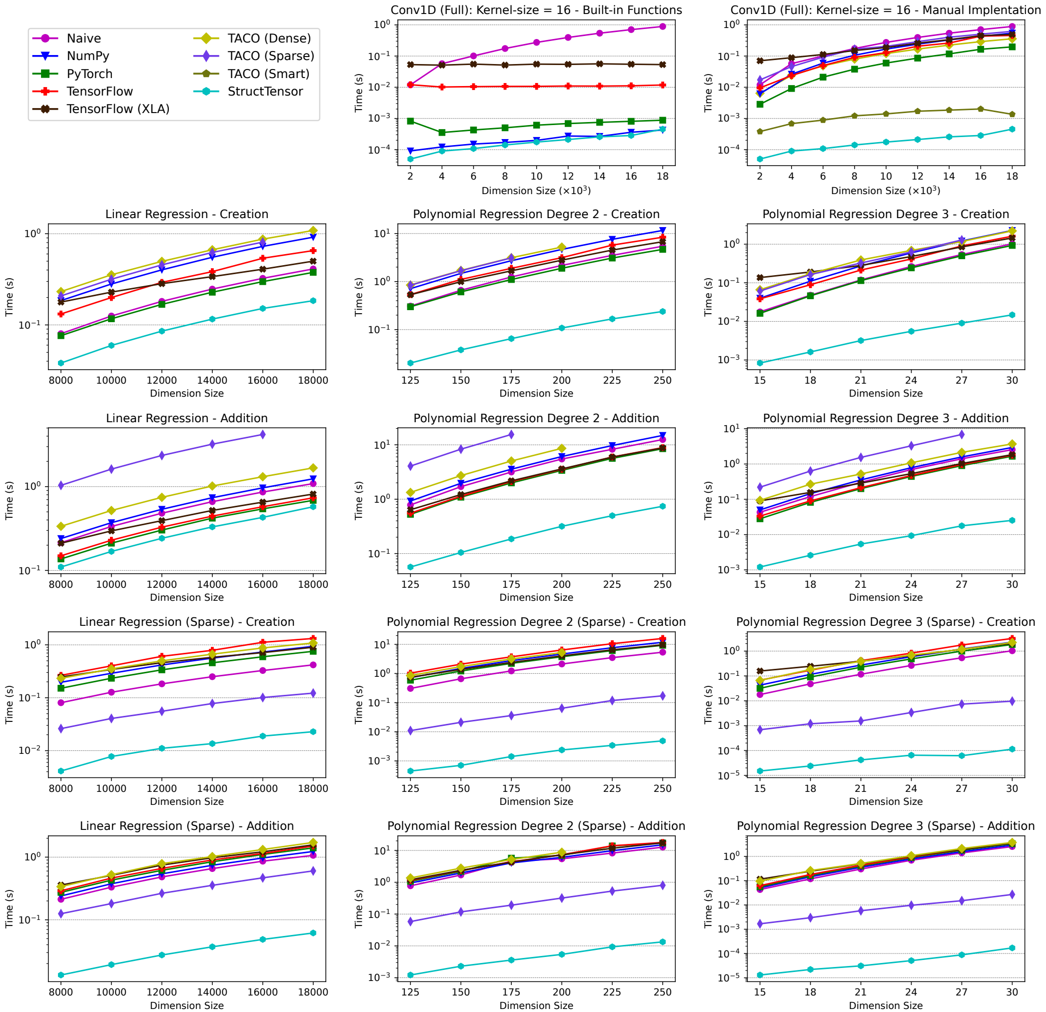

Fig. 15. The run time comparison of one-dimensional full convolution and the covariance matrix creation and addition for linear regression and polynomial regression degree-2 and degree-3 with dense and sparse feature vectors. In all kernels, StructTensor significantly outperforms the competitors thanks to avoiding redundant computation. It is also competitive with the built-in convolution function provided by Numpy.

The missing numbers mean that the process was killed due to a segmentation fault. For example, TACO in fully dense and fully sparse formats can only handle the polynomial regression degree-2 up to 200 and 175 features, respectively, because it gets killed for more than that (due to excessive memory allocation). The smart format of TACO for these kernels is the same as the dense format of TACO; therefore not shown on the plots separately. In all kernels, StructTensor outperforms all other competitors. For the kernels with more redundancy (i.e., polynomial regression degree-2 and degree-3), the performance gap between StructTensor and competitors increases due to an increase in repetitive computation.

The last six plots in Figure 15 show the performance of the same regression kernels by using sparse feature vectors (33.3% sparsity) corresponding to categorical features. Previous kernels

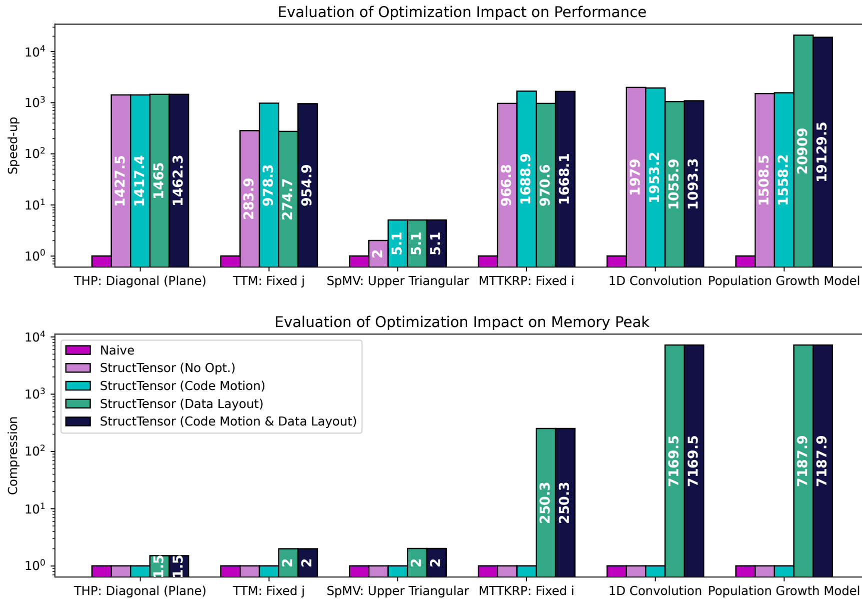

Fig. 16. Impact of different optimizations for six kernels. The length of the vector in the one-dimensional convolution kernel, population growth model, and SpMV is $18K$. The kernel size for the convolution is 16. Each dimension of THP, TTM, and MTTKRP has a size of $1K$.

only contain redundancy patterns; however, these kernels leverage both sparsity and redundancy patterns. The smart format of TACO for these kernels is the same as the sparse format of TACO; therefore not shown on the plots separately. StructTensor outperforms all the competitors by up to 6 orders of magnitude despite the lack of data-layout optimizations, loop tiling, and vectorization. The sparse version of TACO is the runner-up thanks to leveraging the existing sparsity in the input vectors. The run time of all other competitors is similar to their run time in the regression kernels with dense feature vectors since they still perform the computation over all elements of the inputs.

7.4 Optimization Impact

We evaluate the effectiveness of StructTensor and the impact of optimizations against the naïve implementation for six kernels: 1) THP with a plane diagonal structure (cf. Table 1), 2) TTM where the j index is fixed (cf. Table 1), 3) SpMV with upper triangular structure, 4) MTTKRP where the i index is fixed (cf. Table 1), 5) one-dimensional full convolution using Toeplitz matrix, and 6) population growth model employing a Leslie matrix. Memory peak is computed using the massif [Nethercote et al. 2006] tool of Valgrind [Nethercote and Seward 2007].

As shown in Figure 16, StructTensor performs up to 4 orders of magnitude faster than a naïve implementation of all six kernels, even without any optimizations. Enabling code motion either improves the performance by a great ratio (e.g., TTM, SpMV, MTTKRP) or has a negligible impact (e.g., THP, convolution, population growth model). Storing unique elements of the sparse tensor in an array data layout can sometimes cause performance degradation (e.g., convolution) by limiting the compiler optimization opportunities existing in the primary layout. In other kernels, data layout modification removes the zero elements and gets rid of unnecessary dimensions of the sparse tensor, and has a consistently positive impact on run time. Utilizing both optimization techniques enhances performance significantly for all kernels.

As shown in Figure 16, the memory peak can be reduced by simply changing the input data layout, while code motion has no effect on it, as expected. Storing only the unique elements in all

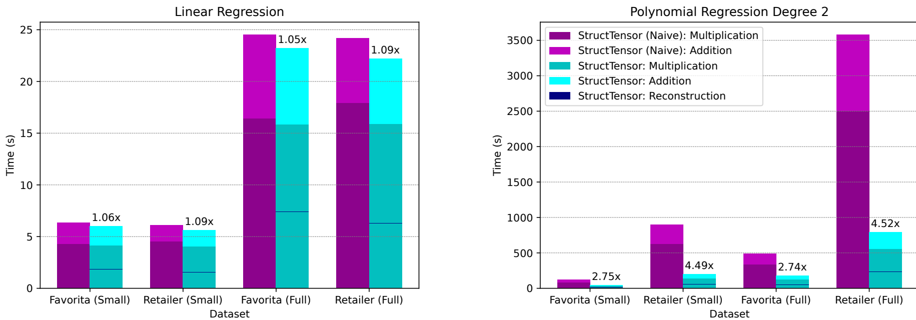

Fig. 17. Run time of in-database machine learning over Favorita (6 continuous features) and Retailer (13 continuous features) datasets. The small version of these datasets contains 25% of rows. Here, we compute the covariance matrix of linear and polynomial regression degree-2 over the join of multiple relations. Reconstruction time is negligible; thus, it is not visible on the chart.

six kernels resulted in a significant decrease in the memory peak. In kernels where modifying the data layout can cause performance degradation, memory compression can pay off. For instance, in the case of one-dimensional convolution, a data layout transformation results in almost 2x worse performance, while the memory peak improves by ∼7200x.

7.5 End-to-End Experiments

We evaluate StructTensor on an in-database machine learning task. This benchmark does not compare StructTensor to other frameworks since the end-to-end experiment is built on top of covariance matrix creation and addition, which are benchmarked separately against all other competitors (cf. Figure 15). We use StructTensor to create a covariance matrix for linear and polynomial regression degree-2. We consider the following real-world datasets: Retailer [Khamis et al. 2020] with 13 continuous features and Favorita [Favorita 2017] with 6 continuous features. Both of them have a small (consisting of 25% of the data elements) and full version. StructTensor is compared with a naïve implementation that does not leverage the redundancy structure.

We use the idea of a semi-ring covariance matrix data-structure [Nikolic and Olteanu 2018; Shaikhha et al. 2022] for both implementations. A semi-ring is a type of data structure that includes a domain D, two associative binary operators (+) for addition and (*) for multiplication, and identity elements (0) for addition and (1) for multiplication. The addition operator must be commutative, and both left and right multiplication must be distributive over addition. Additionally, any element in D multiplied by 0 must equal 0. We employed and extended the definition of the semi-ring covariance matrix provided by [Shaikhha et al. 2022]. This data structure pushes aggregates before joins, which heavily improves both memory consumption and run time. In a semi-ring covariance matrix, different degrees of interaction are decoupled and stored separately. This means that for polynomial regression degree-2 (cf. Section 3), we store degree-2 ($M1$), degree-3 ($M2$), and degree-4 ($M3$) interactions separately. Furthermore, degree-3 interactions are calculated twice in both $(\mathbf{f} \otimes vec(\mathbf{f} \otimes \mathbf{f}))$ and $(vec(\mathbf{f} \otimes \mathbf{f}) \otimes \mathbf{f})$. As the naïve version does not use the structure information, it cannot detect such coarse-grained redundancy information.

Figure 17 represents the run time of StructTensor in comparison with the naïve version. StructTensor produces a structure-aware code that reduces the computations by avoiding redundant computation. Hence, there should be a reconstruction phase that rebuilds the final result and put all elements that have not been computed in their corresponding positions based on the redundancy map, which is negligible in comparison with the rest of the computation.

Table 2. Comparison of different tensor processing frameworks. FS: Fixed-size, SY: Symbolic.

Framework	LA	TA	Dense	Sparse	Redundancy	Loop Opts.
Dense TA (TensorFlow)	●	●	●	○	○	●
Dense LA (BLAS)	●	○	●	○	○	●
Sparse TA (TACO, SPLATT)	●	●	●	●	○	●
Sparse LA (MKL, OSKI)	●	○	●	●	○	●
Static Sparse LA (EGGS)	●	○	●	◐ (FS)	◐	●
Structured LA (LGen)	●	○	●	◐ (FS)	●	●
Symbolic Sparse LA (Sympiler)	◐	○	●	● (SY)	◐	●
StructTensor	●	●	●	◐ (SY)	●	◐

8 RELATED WORK

Table 2 compares various optimized linear and tensor algebra frameworks. These frameworks vary in their applications, support of data layouts, structure awareness, supported structures, approach to capturing the structure, and loop optimization levels. We will elaborate on them in this section.

Dense Tensor Algebra. Polyhedral frameworks such as isl [Verdoolaege 2010] provide advanced scheduling and code generation capabilities. CLooG [Bastoul 2004] provide efficient loop nest code generation for dense and affine tensor algebra computations. These polyhedral frameworks are used in tensor compilers. Tensor Comprehension [Vasilache et al. 2018] uses isl and provides a polyhedral-based DSL supporting generalized Einstein notation that leads to optimized Cuda code generation for dense deep learning computation. PolyBlocks [PolyMage 2023] also uses isl and the MLIR infrastructure to transform high-level Python code to an optimized low-level GPU code.

TensorFlow [Abadi et al. 2015] provides dense tensor operation for large-scale machine learning applications.[5] All these works provide efficient kernels for dense linear and tensor algebra but for structured tensors, still, they do unnecessary and/or redundant computations.

Sparse Tensor Algebra. The sparse polyhedral framework [Strout et al. 2018] extends the ability of polyhedral compilation to support sparse tensor algebra as well [Mohammadi et al. 2019]. TACO [Kjolstad et al. 2017] handles sparse and dense computation over tensor algebra. JAX [Bradbury et al. 2018] has preliminary support for sparse tensors based on a TACO-inspired design in the MLIR infrastructure. However, unlike StructTensor, none of these works supports redundancy-aware computation. Moreover, sparsity is handled in run time, leading to irregular memory access that are hard to optimize for the compiler [Tang et al. 2020a] (cf. Section 7.2).

Specialized Sparse Linear Algebra. [Augustine et al. 2019] take a different approach by breaking the irregular sparsity patterns into sub-computations with regular structures so they can remove indirect access and provide vectorization to the linear algebra code. EGGS [Tang et al. 2020a] further specializes the computation to a sparsity pattern and creates the expression tree of the result by unrolling the entire computation. Performing common-subexpression elimination over the expression tree can partially remove redundancies, but cannot detect symmetric-style patterns. StructTensor infers redundancy patterns at compilation time to specialize the generated code.

Structured Linear Algebra. LGen [Spampinato and Püschel 2016] proposes a polyhedral-based technique for code generation of small-scale structured linear algebra. They provide i) a predefined set of operations required for basic linear algebra, ii) a predefined set of matrix structures, iii) inference rules for the combination of the defined structures and operations, and iv) a fixed-size polyhedral set and polyhedral map to maintain the unique elements (called `SInfo`) and redundancy information (called `AInfo`) respectively. However, it does not support higher-order tensor computations and is limited to fixed-size small-scale matrices. To extend LGen to support higher-order

[5] There is limited support for sparse tensor processing in TensorFlow, but it was shown to be suboptimal [Chou et al. 2018].

tensors, one should redesign their intermediate representation and add support for more generalized tensor computations such as Einsum. To handle the variable size computation for higher-order tensors, one should extend the representation of polyhedral sets/maps to parametric lengths.

Sympiler [Cheshmi et al. 2017] utilizes symbolic analysis at compile-time (similar to StructTensor) to produce high-performance code (e.g., using parallelisation, tiling, etc.) for structured variable-size matrix computations involving sparse data. However, it is limited to specific kernels (e.g., sparse direct solvers) and only captures the structure for matrices. To extend their system for higher-order tensors, a wider range of structures, or computing other kernels, the user must provide i) the symbolic inspector (acts similar to StructTensor's unique set inference), ii) a specific inspection strategy for that specific structure, and iii) how it can be coupled with the specific numerical kernels and the internal AST of Sympiler. However, in StructTensor one can perform arbitrary generalized Einsum computations empowered by structure inference using program reasoning. As opposed to StructTensor, Sympiler does not use any kind of generalized program reasoning to provide the inspector set but rather asks the user to provide it for the specific structure and kernel inside the Sympiler's code. That said, one can see StructTensor and Sympiler as complementing each other; StructTensor provides unique set inference for higher-order tensor expressions and can generate the AST of Sympiler to benefit from the optimizations provided by Sympiler. To the best of our knowledge, no previous work supports structure for higher-order tensors.

Declarative Data Processing Languages. STUR is closely connected to logic languages (e.g., Datalog and Prolog) with two main differences: STUR i) does not allow recursive definitions, and ii) enables aggregations over tensors in addition to sets. Dyna [Eisner and Filardo 2010] extends Datalog by adding support for maps to real numbers in addition to maps to boolean values (i.e., sets). FAQ [Khamis et al. 2016] is the main source of inspiration for STUR, which allows for a combination of different semi-rings. The main focus of FAQ has been on efficient algorithms for evaluating sparse tensor contractions appearing in database query engines [Khamis et al. 2020] (e.g., worst-case optimal joins). STUR provides additional constructs for arithmetic operations over the indices and restricting the ranges, which are crucial for efficient structured tensor computations.

9 CONCLUSION & OUTLOOK

In this paper, we presented StructTensor, a compiler for structured tensor algebra. We considered two classes of structures: (1) sparsity patterns, and (2) redundancy structures. We proposed STUR, a unified IR that is expressive enough for tensor computation and captures both forms of structures. We have shown the soundness of transformations and inference rules. Finally, the experimental results show that StructTensor outperforms the state-of-the-art tensor processing libraries.

We see three future directions. First, StructTensor focuses on structure-specific optimizations, and except for code motion, does not support advanced loop transformations such as tiling. Such optimizations are used for unstructured block structure that is necessary for deep learning applications such as sparse neural networks. It would be also interesting to use polyhedral-based optimizations to generate efficient loop nests. Second, gradient and Jacobian computations can also result in structured tensors. The integration of sparse automatic differentiation [Shaikhha et al. 2023] in StructTensor is an interesting direction. Finally, for storing output compressed tensors, we currently use $n-d$ arrays, and allocate the memory for the entire uncompressed tensor. Using better layouts can significantly reduce the memory pressure, and possibly improve data locality.

ACKNOWLEDGEMENT

The authors thank Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh. Shaikhha acknowledges a gift from RelationalAI.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. 2019. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 625–639. https://doi.org/10.1145/3314221.3314615

Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*. IEEE Computer Society, 7–16. https://doi.org/10.1109/PACT.2004.10018

Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. 2020. What is the State of Neural Network Pruning? (2020). https://proceedings.mlsys.org/book/296.pdf

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 13, 13 pages. https://doi.org/10.1145/3126908.3126936

Stephen Chou, Fredrik Kjolstad, and Saman P. Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 123:1–123:30. https://doi.org/10.1145/3276493

Andrzej Cichocki, Rafal Zdunek, Anh Huy Phan, and Shun-ichi Amari. 2009. *Nonnegative Matrix and Tensor Factorizations - Applications to Exploratory Multi-way Data Analysis and Blind Source Separation*. Wiley. https://doi.org/10.1002/9780470747278

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16, 1 (1990), 1–17. https://doi.org/10.1145/77626.79170

Jason Eisner and Nathaniel Wesley Filardo. 2010. Dyna: Extending Datalog for Modern AI. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6702)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.). Springer, 181–220. https://doi.org/10.1007/978-3-642-24206-9_11

Corporacion Favorita. 2017. Corp. Favorita Grocery Sales Forecasting: Can you accurately predict sales for a large grocery chain?

Roman Gareev, Tobias Grosser, and Michael Kruse. 2018. High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach. *ACM Trans. Archit. Code Optim.* 15, 3 (2018), 34:1–34:27. https://doi.org/10.1145/3235029

Per Christian Hansen. 2002. Deconvolution and Regularization with Toeplitz Matrices. *Numer. Algorithms* 29, 4 (2002), 323–378. https://doi.org/10.1023/A:1015222829062

Poul Einer Hansen. 1989. Leslie matrix models. *Mathematical Population Studies* 2, 1 (1989), 37–67.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Kartik Hegde, Hadi Asghari Moghaddam, Michael Pellauer, Neal Clayton Crago, Aamer Jaleel, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 319–333. https://doi.org/10.1145/3352460.3358275

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

So Hirata. 2006. Symbolic algebra in quantum chemistry. *Theoretical Chemistry Accounts* 116, 1 (2006), 2–17.

Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley,

Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 1–12. https://doi.org/10.1145/3079856.3080246

Mahmut T. Kandemir, Alok N. Choudhary, U. Nagaraj Shenoy, Prithviraj Banerjee, and J. Ramanujam. 1999. A Linear Algebra Framework for Automatic Determination of Optimal Data Layouts. *IEEE Trans. Parallel Distributed Syst.* 10, 2 (1999), 115–135. https://doi.org/10.1109/71.752779

Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies. *ACM Trans. Database Syst.* 45, 2 (2020), 7:1–7:66. https://doi.org/10.1145/3375661

Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 13–28. https://doi.org/10.1145/2902251.2902280

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

José M. Martín-García. 2008. xPerm: fast index canonicalization for tensor computer algebra. *Comput. Phys. Commun.* 179, 8 (2008), 597–603. https://doi.org/10.1016/j.cpc.2008.05.009

Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary W. Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 594–609. https://doi.org/10.1145/3314221.3314646

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (jun 2007), 89–100. https://doi.org/10.1145/1273442.1250746

Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. 2006. "Building Workload Characterization Tools with Valgrind". In *2006 IEEE International Symposium on Workload Characterization*. 2–2. https://doi.org/10.1109/IISWC.2006.302723

Milos Nikolic and Dan Olteanu. 2018. Incremental View Maintenance with Triple Lock Factorization Benefits. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 365–380. https://doi.org/10.1145/3183713.3183758

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

PolyMage. 2023. PolyBlocks. https://www.polymagelabs.com/technology/. Accessed: 2023-08-04.

Shi-Ju Ran, Emanuele Tirrito, Cheng Peng, Xi Chen, Luca Tagliacozzo, Gang Su, and Maciej Lewenstein. 2020. *Tensor network contractions: methods and applications to quantum many-body systems*. Springer Nature.

Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing Tensor Programs on Flexible Storage. *Proc. ACM Manag. Data* 1, 1 (2023), 37:1–37:27. https://doi.org/10.1145/3588717

Amir Shaikhha, Mathieu Huot, and Shideh Hashemian. 2023. ∇SD: Differentiable Programming for Sparse Tensors. *arXiv preprint arXiv:2303.07030* (2023).

Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–33. https://doi.org/10.1145/3527333

Daniel G. A. Smith and Johnnie Gray. 2018. opt_einsum - A Python package for optimizing contraction order for einsum-like expressions. *J. Open Source Softw.* 3, 26 (2018), 753. https://doi.org/10.21105/joss.00753

Daniele G. Spampinato and Markus Püschel. 2016. A basic linear algebra compiler for structured matrices. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, Björn Franke, Youfeng Wu, and Fabrice Rastello (Eds.). ACM, 117–127. https://doi.org/10.1145/2854038.2854060

Michelle Mills Strout, Mary W. Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.

2857721

Xuan Tang, Teseo Schneider, Shoaib Kamil, Aurojit Panda, Jinyang Li, and Daniele Panozzo. 2020a. EGGS: Sparsity-Specific Code Generation. *Comput. Graph. Forum* 39, 5 (2020), 209–219. https://doi.org/10.1111/cgf.14080

Xuan Tang, Teseo Schneider, Shoaib Kamil, Aurojit Panda, Jinyang Li, and Daniele Panozzo. 2020b. EGGS: Sparsity-Specific Code Generation. *Comput. Graph. Forum* 39, 5, 209–219. https://doi.org/10.1111/cgf.14080

Alexey V Titov, Ivan S Ufimtsev, Nathan Luehr, and Todd J Martinez. 2013. Generating efficient quantum chemistry codes for novel architectures. *Journal of chemical theory and computation* 9, 1 (2013), 213–221.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

Sven Verdoolaege. 2010. *isl*: An Integer Set Library for the Polyhedral Model. In *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6327)*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

Received 2023-04-14; accepted 2023-08-27