Massachusetts Institute of Technology

SySTeC: A Symmetric Sparse Tensor Compiler

Radha Patel
MIT CSAIL
Cambridge, Massachusetts, USA
rrpatel@alum.mit.edu

Willow Ahrens
MIT CSAIL
Cambridge, Massachusetts, USA
willow@csail.mit.edu

Saman Amarasinghe
MIT CSAIL
Cambridge, Massachusetts, USA
saman@csail.mit.edu

Abstract

Symmetric and sparse tensors arise naturally in many domains including linear algebra, statistics, physics, chemistry, and graph theory. Symmetric tensors are equal to their transposes, so in the n-dimensional case we can save up to a factor of $n!$ by avoiding redundant operations. Sparse tensors, on the other hand, are mostly zero, and we can save asymptotically by processing only nonzeros. Unfortunately, specializing for both symmetry and sparsity at the same time is uniquely challenging. Optimizing for symmetry requires consideration of $n!$ transpositions of a triangular kernel, which can be complex and error prone. Considering multiple transposed iteration orders and triangular loop bounds also complicates iteration through intricate sparse tensor formats. Additionally, since each combination of symmetry and sparse tensor formats requires a specialized implementation, this leads to a combinatorial number of cases. A compiler is needed, but existing compilers cannot take advantage of both symmetry and sparsity within the same kernel. In this paper, we describe the first compiler which can automatically generate symmetry-aware code for sparse or structured tensor kernels. We introduce a taxonomy for symmetry in tensor kernels, and show how to target each kind of symmetry. Our implementation demonstrates significant speedups ranging from 1.36x for SSYMV to 30.4x for a 5-dimensional MTTKRP over the non-symmetric state of the art.

CCS Concepts: • **Mathematics of computing → Mathematical software**; • **Software and its engineering → Compilers**; • **Computing methodologies → Symbolic and algebraic manipulation**.

Keywords: Sparse, Symmetric, Structured Tensor, Compiler

1 Introduction

A symmetric tensor is a tensor that is invariant under a permutation of its indices. Tensors are often naturally symmetric because of the physical and chemical properties of substances and matter which produce symmetric interactions, structures, or reactions. Additionally, symmetry can also be induced as a mathematical consequence of how we use tensor operations (e.g. $A^T A$).

Real world tensors can also be sparse, meaning they are mostly zero or some other fill value. Special formats have been proposed to only store nonzeros and several systems, such as GraphBLAS [18], TACO [19], or Finch [4] have been developed to efficiently perform operations on sparse tensors, but none of them can handle symmetry automatically.

There are wide-ranging applications of symmetric sparse tensors, from mathematical optimization to scientific computing. In linear algebra, the hat matrix in linear regression and the Q matrix that is a result of QR factorization are both symmetric [16]. In statistics, matrices expressing covariance and other similarly commutative calculations are naturally symmetric [28]. In physics and chemistry computations, the properties of quantum tensor networks and computational fluid dynamics give way to multi-dimensional symmetry [17, 24]. In graph theory, adjacency matrices of undirected graphs, used in algorithms like single-source shortest path and to find connected components, are also symmetric [30].

Table 1. Supported features: ●= Yes, ◖= Partially. [1] = Only static sizes, [2] = Only one sparse tensor at a time, [3] = Only symbolic patterns, [4] = Only contractions.

	MKL	TCE	Cyclops	sBLACs	STUR	SySTeC
Supports Dense Tensors	●	●	●	◖[1]	●	●
Supports Sparse Tensors	●		◖[2]	◖[1,3]	◖[3]	●
Supports Structured Tensors				◖[1]	●	●
Supports General Einsums	◖[4]	◖[4]	●	●	●	●
Optimizes Redundant Reads	●					●
Optimizes Redundant Operations	●	●	●	●	●	●
Optimizes Redundant Storage	●	●	●	●	●	●

Optimizing for symmetry and sparsity at once is uniquely challenging. Symmetric optimizations require that we consider all combinatorial loop reorderings of the kernel and restrict iteration to a triangle, which can be complex and

error prone. Sparse optimizations require reformatting the data to store and process only nonzeros. Symmetry is a property defined on the coordinates of the tensor, and sparse formats obfuscate the relationship between tensor coordinates and where elements are stored in memory. Iterators over sparse tensor formats are often a performance bottleneck, and are especially sensitive to changes in loop ordering and loop bounds [6]. Additionally, since each combination of symmetry and sparse tensor formats requires a specialized implementation, this leads to a combinatorial number of cases, hand-writing solutions is not feasible in the general case. Libraries such as MKL or CuBLAS only support a small subsets of symmetric sparse matrix kernels [1, 2].

A compiler approach is necessary. Though several compilers have been developed to handle symmetric tensors, none of them apply to sparse tensors, and vice versa. Compilers like STUR [15] and Cyclops [33] and sBLACs [35] all optimize for symmetric tensors, but STUR and sBLACs cannot handle unstructured sparse tensors and Cyclops cannot handle more than one sparse tensor at a time. These compilers accelerate symmetric kernels by avoiding redundant computation and storage, but cannot avoid redundant memory operations. In some kernels, like symmetric sparse matrix-vector multiply, we can optimize memory bandwidth by restricting iteration to the upper triangle and performing all necessary updates to the output tensor in one pass.

We aim to fill the gap by presenting a granular approach to identify and exploit symmetry in sparse tensor kernels. Our specific contributions include:

1. To the best of our knowledge, SySTeC is the first system to automatically generate code for symmetric and sparse or otherwise structured (Triangular, Banded, Run-Length-Encoded) tensor operations.

2. A taxonomy of symmetry in tensor kernels, and strategies to utilize each kind of symmetry. We introduce the concepts of *visible* and *invisible input* and *output* symmetries. Capitalizing on these saves memory bandwidth, storage, and compute by reusing reads and writes to symmetric input and filtering redundant storage and computations.

3. We show how to extend traditional compiler optimizations to take advantage of symmetry, as well as introduce new compiler optimizations, such as *simplicial lookup tables* and *diagonal splitting*. Our compiler uses term rewriting to optimize redundancies, and is easily extensible to general operators beyond + and ∗.

4. We implement our compiler and evaluate it on several common tensor kernels, demonstrating speedups from 1.36x for SSYMV to 30.4x for a 5-dimensional MTTKRP with the symmetric code generated by the compiler over the naive implementation of these kernels.

2 Background

In this section, we introduce the terminology and syntax that will be used throughout the rest of the paper.

2.1 Symmetric Tensors

A matrix M is symmetric if $M[i_1, i_2] = M[i_2, i_1]$—i.e. the entries at permutations of the indices are equivalent. We can generalize this definition for tensors [12].

Definition 2.1 (Symmetry). Let T be an n-dimensional tensor. T is *symmetric* if for all permutations σ of $\{1, ..., n\}$,

$$T[i_1, ..., i_n] = T[i_{\sigma(1)}, ..., i_{\sigma(n)}].$$

In the case of matrices, symmetry is binary: a matrix is either symmetric or it is not. However, when dealing with higher-order tensors, this definition can be expanded with the notion of *partial symmetry*. A *partition* Π of a set A is a collection of non-empty, pairwise disjoint subsets, which we will refer to as *parts*, of A, such that each element of A belongs to exactly one subset within the collection [27]. We denote π_i to be the ith part of Π. We define partial symmetry relative to a chosen partition [29].

Definition 2.2 (Partial Symmetry). Let T be an n-dimensional tensor, and let Π be a partition of $\{1, ..., n\}$. Then T is *partially symmetric* if

$$T[i_1, ..., i_n] = T[i_{\sigma(1)}, ..., i_{\sigma(n)}]$$

for all permutations σ of $\{1, ..., n\}$ which only permute elements within their parts in Π.

Then, we can denote that T has Π symmetry.

Since the upper and lower triangles of a symmetric matrix are equal, our framework restricts our computations to one of the triangles to avoid redundant operations. We refer to the triangle that we choose to compute as the *canonical triangle* of the tensor. We choose the upper triangle in this work.

Definition 2.3 (Canonical). Let tensor $T[i_1, ..., i_n]$ have symmetry Π_T. Coordinates $[i_1, ..., i_n]$ are *canonical* if $i_p \leq i_q$ for any $p < q$ with i_p and i_q in the same part of Π_T. Otherwise, the coordinates are *non-canonical*. The *canonical triangle* of a tensor consists of all the canonical coordinates in the tensor.

Although computations in the triangles of a tensor are repeated, computations on diagonals are not, and so diagonals must often be handled separately.

Definition 2.4 (Diagonal). A *diagonal* of a tensor T consists of all coordinates $[i_1, ..., i_n]$ where the indices in a subset D of $\{i_1, ..., i_n\}$ where $|D| > 1$ are equal.

2.2 Sparse and Structured Tensor Programming

We will be using the program syntax and formats from Finch, a Julia-to-Julia compiler designed for optimizing loop nests over sparse or structured (Triangular, Banded, Run-Length-Encoded) multidimensional arrays [4]. Finch supports a wide

range of sparse and structured storage formats, as well as the control flow necessary to implement and execute symmetric kernels, such as conditionals and multiple outputs. Finch also simplifies the complexities of sparse data manipulation, enabling us to write **loop structures that appear dense but are compiled to be sparse** which makes it easier to focus the optimizations we apply to take advantage of symmetry. Figure 1 shows the syntax of Finch.

```
EXPR := LITERAL | VALUE | INDEX | VARIABLE | EXTENT | CALL | ACCESSS
STMT := ASSIGN | LOOP | DEFINE | SIEVE | BLOCK

 DECLARE := TENSOR .= EXPR(EXPR...)    #V is the set of all values
  FREEZE := @freeze(TENSOR)            #S is the set of all Symbols
    THAW := @thaw(TENSOR)              #T is the set of all types
  ASSIGN := ACCESS <<EXPR>>= EXPR
    LOOP := for INDEX = EXPR                 LITERAL := V
              STMT                             VALUE := S::T
            end                               TENSOR := S
  DEFINE := let VARIABLE = EXPR                INDEX := S
              STMT                          VARIABLE := S
            end                              EXTENT := EXPR : EXPR
   SIEVE := if EXPR                            CALL := EXPR(EXPR...)
              STMT                           ACCESS := TENSOR[EXPR...]
            end                                MODE := @mode(TENSOR)
   BLOCK := begin
              STMT...
            end
```

Figure 1. Finch Syntax [4, Figure 7]

Finch uses a hierarchical mode-by-mode fibertree description of tensor formats, where tensors are conceptualized as a vector of vectors of vectors, etc. [10, 36] This allows us to characterize each level of the tree as a separate vector type, expressing several common sparse and structured tensor formats as combinations of simple level formats. For example, CSR format is Dense(Sparse(Element(0.0))), or a dense vector of sparse vectors [26]. The 3-dimensional CSF format is Dense(Sparse(Sparse(Element(0.0)))) [31]. We refer the reader to literature on Finch for more information and examples of tensor formats [4, Figure 6 and Table 3].

Critically, accesses to sparse tensors in Finch syntax (e.g. x[i]) act as iterators over sparse tensors, and comparisons between indices (e.g. i < j) are lifted into loop bounds. Thus, the Finch code on left compiles to the code on the right.

```
                                    q = 1
x = Sparse(Element(0.0))            stop = min(nnz(x), 7 - 1)
@finch                              while i < stop
for i=_                 Finch          i = x.idx[q]
    if i < 7            ────────▶       if i <= stop
        s[] += x[i]                         s += x.val[q]
                                        q += 1
```

3 Techniques to Exploit Symmetry

We categorize the symmetry that presents itself in assignments in two groups: **input symmetry**, which involves one or more input tensors being symmetric and **output symmetry**, which consists of a symmetric output tensor.

Assignments can have either input or output symmetry, as well as both types of symmetry. We make the distinction because the techniques to exploit symmetry vary based on the type of symmetry.

Furthermore, we can subdivide output symmetry into two more intersecting types—visible and invisible, where **visible output symmetry** is between indices that are present in the output tensor and **invisible output symmetry** is between indices that are not explicitly present in the output tensor, but are still involved in the computation.

Example 3.1 (Visible and Invisible Output Symmetry). The assignment $B[i, j] = A[i, k] * A[j, k]$ exhibits visible output symmetry. Essentially, $B[i, j] = A[i, k] * A[j, k] = A[j, k] * A[i, k] = B[j, i]$. Thus, we know that B exhibits $\{\{i, j\}\}$ symmetry. Because the symmetry is preserved in the output, we refer to this symmetry as *visible*.

On the other hand, the assignment $B[i] = A[i, j] * A[i, k]$ exhibits invisible output symmetry. Let us rewrite the assignment with a temporary tensor T as follows.

$$T[i, j, k] = A[i, j] * A[i, k]$$
$$B[i] = \sum_{j,k} T[i, j, k]$$

Now the symmetry is more apparent: $T[i, j, k] = A[i, j] * A[i, k] = A[i, k] * A[i, j] = T[i, k, j]$. T (and thus B) exhibit $\{\{j, k\}\}$ symmetry. Because this symmetry is not seen in the output B, we refer to this symmetry as *invisible*.

The two core strategies we have identified to exploit symmetry to make better use of memory and compute are (1) reusing canonical reads to save on bandwidth and (2) filtering redundant computations, which are dissected in more detail in the following subsections.

3.1 Reusing Canonical Reads

When input tensors are symmetric, we can restrict reads to the canonical triangle and use the same read to perform multiple computations for the output. This is critical for sparse inputs, since iteration over sparse inputs is expensive, especially if we must iterate in multiple directions at once, which is particularly relevant for iteration bound and memory bound kernels (e.g. SSYMV). The efficiency of these kernels is often limited by the rate at which data can be transferred from the memory to the processor (memory bandwidth) rather than the rate at which the processor can perform calculations (compute throughput).

Let us take a look at what reusing canonical reads algorithmically entails for the sparse symmetric matrix-vector multiply (SSYMV) kernel given by $y[i] = A[i, j] * x[j]$. The optimization in Figure 2 limits accesses of the symmetric tensor to the canonical triangle and uses reads that are not on the diagonal for two assignments and reads that are on

```
                              for j=_, i=_
                                  if i < j
  for j=_, i=_                        a = A[i, j]
      y[i] += A[i, j] * x[j]          y[i] += a * x[j]
                                      y[j] += a * x[i]
                                  if i == j
                                      y[i] += A[i, j] * x[j]
```

Figure 2. On left, a naive SSYMV. On right, SSYMV that accesses only canonical triangle *and* reuses memory reads.

the diagonal for one assignment. Note that i and j are permuted in the second assignment and this makes up for not covering the iteration space where $i > j$.

As the number of axes of symmetry increase, the complexity of the symmetry-optimized kernel increases, but so do the optimization opportunities. For instance, suppose that A in the mode-1 TTM kernel [21] given by $C[i, j, l] = A[k, j, l] * B[k, i]$ is fully symmetry. The resulting kernel from restricting accesses of A to the canonical triangle and performing all necessary updates to the output tensor C is given by Listing 1.

```
 1  for l=_, i=_, k=_, j=_
 2      if j <= k && k <= l
 3          if j < k && k < l
 4              A = A[j, k, l]
 5              C[i, j, l] += A * B[k, i]
 6              C[i, j, k] += A * B[l, i]
 7              C[i, k, l] += A * B[j, i]
 8              C[i, k, j] += A * B[l, i]
 9              C[i, l, k] += A * B[j, i]
10              C[i, l, j] += A * B[k, i]
11          if j == k && k != l
12              A = A[j, k, l]
13              C[i, j, l] += A * B[k, i]
14              C[i, j, k] += A * B[l, i]
15              C[i, l, k] += A * B[j, i]
16          if j != k && k == l
17              A = A[j, k, l]
18              C[i, j, l] += A * B[k, i]
19              C[i, k, l] += A * B[j, i]
20              C[i, k, j] += A * B[l, i]
21          if j == k && k == l
22              C[i, j, l] += A[j, k, l] * B[k, i]
```

Listing 1. TTM kernel that accesses only the canonical triangle of A.

The monotonically increasing condition on line 2 of Listing 1 enforces that we only iterate over the canonical triangle of symmetric tensor A. With three axes of symmetry, there are more diagonals to consider as the number of equivalence groups increase: i.e. the diagonals represented by equivalence groups $\{(j = k)\}$, $\{(k = l)\}$, $\{(j = l)\}$, and $\{(j = k = l)\}$. We handle each of these diagonals separately in Listing 1, with

the exception of $\{(j = l)\}$ because our overarching monotonically increasing condition ensures that if $j = l$, then we are overlapping the diagonal represented by $\{(j = k = l)\}$, which we already handle.

Given n axes of symmetry, upon restricting a kernel to access only $\frac{1}{n!}$ of a tensor, we need to perform $n!$ assignments in each iteration to write to all the triangles of the output tensor in the case where none of the n indices are equivalent. However, if m indices are equivalent to each other (e.g. we read an element on a diagonal of the symmetric tensor) then we only perform $\frac{n!}{m!}$ assignments to avoid duplicate assignments. In other words, we perform the same number of assignments as unique permutations of the indices per iteration to make up for the fact that we are only covering $\frac{1}{n!}$ of the iteration space. The simplest solution to symmetrize code and handle these edge cases is to define every possible combination of equivalent indices and specify each assignment to the output, then optimize those statements.

3.2 Optimizing Multiple Triangular Assignments

The symmetrization process results in multiple assignments being performed with one read of the symmetric tensors. Explicitly representing multiple triangular assignments makes plain the redundancies of symmetry and allows us to easily optimize or filter them.

3.2.1 Visible Output Symmetry. Visible output symmetry involves indices that *are* used to index the output tensor. In the presence of visible output symmetry, we can restrict our kernel to compute the values comprising only the canonical triangle of the output. Afterwards, we can perform an extra post-processing step that consists of copying the canonical triangle of the output to the other triangles.

For example, let us consider the first block of the symmetrized TTM kernel given in Listing 1 that performs the assignments using coordinates of A in the canonical triangle that are not on a diagonal. We reorder the assignments to make the pattern from output symmetry more obvious in Listing 2. Swapping the second and third indices in the output tensor on the left-hand side lends an equivalent right-hand side for each expression. As depicted in Listing 3, we can exploit the output symmetry by only writing to the canonical triangle of the output tensor (i.e. if we index C as C[i, j, l], then only where j <= l), which reduces the number of computations that are done by a factor of 2. Then, we can copy the values from the canonical triangles to the other triangles of the output tensor in a separate loop nest (lines 7-9 of Listing 3), thus completing the remaining assignments.

```
 1  for l=_, j=_, k=_, i=_
 2      if j <= k && k <= l
 3          A = A[j, k, l]
 4          C[i, j, l] += A * B[k, i]
 5          C[i, l, j] += A * B[k, i]
 6          C[i, j, k] += A * B[l, i]
```

```
7        C[i, k, j] += A * B[l, i]
8        C[i, k, l] += A * B[j, i]
9        C[i, l, k] += A * B[j, i]
```

Listing 2. Before exploiting output symmetry in the conditional block of the TTM kernel that handles non-diagonal coordinates of A.

```
1  for l=_, j=_, k=_, i=_
2      if j <= k && k <= l
3          A = A[j, k, l]
4          C[i, j, l] += A * B[k, i]
5          C[i, j, k] += A * B[l, i]
6          C[i, k, l] += A * B[j, i]
7  for l=_, j=_, i=_
8      if j > l
9          C[i, j, l] = C[i, l, j]
```

Listing 3. After exploiting output symmetry in the conditional block of the TTM kernel that handles non-diagonal coordinates of A.

In general, if n indices are in the same part of a partition representing the visible symmetry of the output tensor, then we can reduce the number of operations by a factor of $n!$.

3.2.2 Invisible Output Symmetry. While visible output symmetry results in equivalent assignments to multiple locations, invisible output symmetry results in equivalent assignments to the same locations. We optimize redundant computation by replacing k additions with equivalent right-hand sides with a single addition that multiples the right-hand side by scalar k.

SYPRD is given by $y = x[i] * A[i, j] * x[j]$ where A is symmetric. SYPRD exemplifies invisible output symmetry because the output is a scalar (and thus any output symmetry must be with indices that are not present in the output). If we permute i, j, then we obtain an equivalent assignment.

$$y = x[i] * A[i, j] * x[j] = x[j] * A[j, i] * x[i]$$

Thus, instead of performing both non-diagonal assignments in Listing 4 (lines 5-6), we can optimize by only performing one assignment but multiplying it by a factor of 2, as depicted in Listing 4 (line 3). Note that this does not apply to the block that accesses the diagonal entries of A because i and j are equivalent and thus there is only one assignment.

```
1  for j=_, i=_
2      if i <= j
3          if i < j
4              A = A[i, j]
5              y[] += x[i] * A * x[j]
6              y[] += x[j] * A * x[i]
7          if i == j
8              y[] += x[i] * A[i, j] * x[j]
```

Listing 4. SYPRD before exploiting output symmetry.

```
1  for j=_, i=_
2      if i < j
3          y[] += 2 * x[i] * A[i, j] * x[j]
4      if i == j
5          y[] += x[i] * A[i, j] * x[j]
```

Listing 5. SYPRD after exploiting output symmetry

Invisible output symmetry often presents itself when there are multiple of the same operands in an assignment. Using the same process depicted in the prior section, we may need to swap around a few indices in the blocks accounting for the diagonals to make the invisible output symmetry more apparent. This normalization makes it easier to pinpoint when assignments are equivalent.

If n indices are in the same part of a partition representing the invisible symmetry of the output tensor, then we can reduce the number of operations by a factor of $n!$.

4 Symmetric Compiler Methodology

Given an assignment and a map of input tensors that are known to be symmetric and the partitions that represent their symmetries, to take advantage of symmetry, we need to first generate a kernel that reuses memory reads and then, filter the resulting redundant computations. For simple assignments, it is easy to do this by hand, but as the number of indices involved in a symmetry group, the dimensionality of the tensors, and the number of tensors in the assignment increase, writing a symmetric kernel becomes less intuitive and more akin to a trial-and-error process. In this section, we propose a mechanical, generalizable system to generate a symmetry-exploiting kernel that is applicable to any tensor assignment and which can be replicated in any compiler.

We divide this system in two phases to reflect the two core strategies of first capitalizing on memory bandwidth and then compute throughput. The first phase is *symmetrization* and consists of generating code to read only the canonical triangle(s) of the symmetric tensor(s). The second phase is *optimization* and consists of applying various transforms to reduce the number of memory accesses and operations that are performed.

4.1 Symmetrization

The process of symmetrization involves adding the appropriate control structures to limit the iteration space to the canonical triangles of the symmetric input tensors and determining which additional assignments will need to be made and and under what conditions to ensure that all the appropriate updates to the output tensor are performed.

We will use set S_T to represent the equivalent permutations of a fully or partially symmetric tensor T. If T is fully symmetric, S_T is the set of all permutations of $\{1, ..., n\}$. If T is partially symmetric with partition Π, S_T is the set of all permutations σ of $\{1, ..., n\}$ which only permute elements within their parts in Π.

Given an assignment

$$O[i_1, ..., i_n] = T_1[i_{1,1}, ..., i_{1,n_1}] \otimes ... \otimes T_m[i_{m,1}, ..., i_{m,n_m}],$$

let Π_i be the partition that defines the symmetry of T_i. Furthermore, we represent the symmetry groups as $S_{T_1}, ..., S_{T_m}$ and S_O.

To represent and easily distinguish which diagonal of a tensor we are accessing, we introduce the notion of *equivalence groups*—a term that we have formulated to represent the tensor generalizations of diagonals.

Definition 4.1 (Equivalence Group). Given a set of indices I, we define *equivalence group* E to represent a partition Π of indices $i \in I$ where for each part $\pi \in \Pi$, $i_n = i_m$ for all $n, m \in \pi$.

We define the notation symmetry group $S_T|E$ to represent the unique permutations of a tensor's indices given a particular equivalence group E.

Definition 4.2 (Unique Symmetry Group). Let $S_T|E$ represent the *unique symmetry group*, which given an equivalence group E, consists of $S_T|E = \{\pi \in S_n \mid \forall i, j \in \{1, 2, ..., n\}$, if i, j are both in the same subset of E, then $\pi(i) < \pi(j)\}$.

The four stages below delineate the process to systematically generate a symmetrized kernel for this assignment. We assume that in addition to the assignment itself, the client has also provided the partitions Π_i for each input tensor T_i as well as the loop order (i.e. the order in which they will be looping through the indices in the assignment).

1. **Identify Symmetry**: First, we determine the set of permutable indices P, which is given by

$$P = \bigcup_{i=1}^{m} \left(\bigcup \{\pi \in \Pi_i \mid |\pi| > 2\} \right)$$

and includes all indices in the tensor assignment that are in a symmetry group with more than one index. Note that this step overapproximates symmetry—for instance, if we have $\{\{1, 2\}, \{3, 4\}\}$ symmetry in a tensor, we obtain $P = \{1, 2, 3, 4\}$.

2. **Restrict Iteration Space**: We establish an ordering $p_1, ..., p_n$ of the permutable indices in P such that accessing any tensor T_i at entries where $p_1, ..., p_n$ are monotonically increasing (i.e. $p_1 \leq ... \leq p_n$) will only access the canonical triangle of all symmetric tensors. This ordering is a topological sort of the dependence graph between canonical indices and always exists.

3. **Define Assignments**: For each equivalence group E that can be constructed from P and satisfies the monotonically increasing condition established in step (2), we determine the unique symmetry group $S_P|E$ where S_P consists of all the permutations of P. Then we can apply each permutation $\sigma \in S_P|E$ to the original assignment to generate all the assignments that need to

be performed if the equivalence relationships defined by E are satisfied.

4. **Normalize Assignments**: Lastly, we normalize all assignments to make it easier to identify equivalent assignments or patterns across assignments during the optimization process. There are many ways to rewrite an expression and yield an equivalent result; namely, indices in a symmetric group of a symmetric tensor can be permuted and operands involved in commutative operations can be commuted. Standardizing tensor assignments can make it easier to programmatically identify equivalent assignments and distinguish patterns across assignments. Thus, we define the notion of a *normalized* assignment to be an assignment were (1) all tensors on the right-hand side have been ordered based on some predetermined sort order (e.g. alphabetical) and (2) for all symmetric tensors T_i in the assignment, all indices in the same part of the partition Π_i representing the symmetry of T_i are ordered based on some predetermined sort order (e.g. to be concordant with the loop order).

The resulting symmetrized kernel from applying these steps is depicted via mathematical pseudocode in Figure 3. We first enforce the monotonically increasing condition for the permutable indices (line 1) to restrict the iteration space to the canonical triangles of the symmetric tensors. We iterate through all possible equivalence groups of P (line 3) and for each, determine the set of unique permutations of P given the equivalence group (line 4). We apply each of these unique permutations to the initial assignment (line 6) to obtain all the assignments that are performed for the equivalence relationships represented by corresponding equivalence group.

```
1: for i₁ = 1 : _, i₂ = 1 : _, ... do
2:    if p₁ ≤ ... ≤ pₙ then
3:       for all E of P do
4:          Construct S_P|E
5:          for all σ ∈ S_P|E do
6:             (O[i₁, ..., iₙ] = T₁[i₁¹, ..., iₙ¹] ⊗ ... ⊗ /
7:                Tₘ[i₁ᵐ, ..., iₙᵐ]) [i → σ(i)]
8:          end for
9:       end for
10:   end if
11: end for
```

Figure 3. Pseudocode for Symmetrized Kernel

We can furthermore unroll the loops from lines 5-7 and lines 3-8 in Figure 3 to generate a more efficient kernel. Additionally, note that each equivalence group is exclusive (i.e. a coordinate only satisfies one of the equivalence groups), so when we do unroll the loops, the conditional blocks that are generated are exclusive.
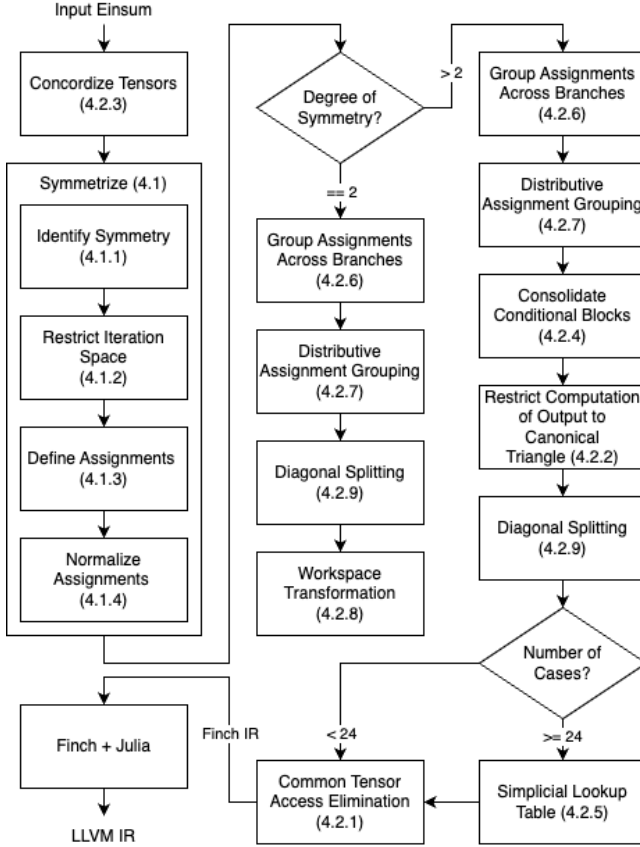
Figure 4. Symmetric Compiler Flow

4.2 Optimization

After symmetrizing the kernel such that it accesses only the canonical triangle(s) of the symmetric tensor(s), we shift to applying various transforms to reduce the number of computations performed. These transforms are the building blocks for filtering redundant code. Several of these transformations may be familiar, but these transformations are performed at the level of sparse tensor computation in Finch IR before Finch lowers to Julia and then LLVM IR. Therefore, we cannot rely on the Julia or LLVM compilers to perform these optimizations. The *simplicial lookup tables* and *diagonal splitting* are novel, as is the overall composition of these transforms with the goal of exploiting symmetry. The compiler is summarized in Figure 4.

4.2.1 Common Tensor Access Elimination. We replace repeated reads of the same element in a tensor with a single constant value. In particular, after normalizing the symmetrized kernel, all accesses to a fully symmetric tensor will be equivalent in each iteration of a loop. For a fully symmetric tensor of order n, this will entail reducing memory reads by a factor of $n!$. Accesses to other tensors might also be repeated and thus, can also be consolidated. This step is crucial for the Finch compiler because it understands each

tensor access as a separate iterator, and multiple redundant accesses would lead to intersecting multiple iterators in the loop. Thus, this step must be applied before running Finch.

```
y[i] += A[i, j] * x[j]          temp = A[i, j]
y[j] += A[i, j] * x[i]          y[i] += temp * x[j]
                                y[j] += temp * x[i]
```

4.2.2 Restrict Computation of Output to Canonical Triangle. Identify assignments with equivalent right-hand sides that update symmetric entries of the output tensor (i.e. coordinates with particular indices swapped) in the same conditional block. In this case, replace the symmetric assignments with just one assignment to the canonical coordinate of the output tensor. We also mark the indices across which the output tensor will need to be replicated. After the kernel is computed, the canonical triangle of the output tensor can be replicated to the noncanonical triangles if needed. We chose to do the replication in a separate loop as the main loop may access the same output location multiple times. By keeping these loops separate, we avoid repeating the out-of-order access required to replicate the output tensor.

```
for j=_, i=_                    for j=_, i=_
    if i <= j                       if i <= j
        y[i, j] += A[i, j] * x[j]       y[i, j] += A[i, j] * x[j]
        y[j, i] += A[i, j] * x[j]   for j=_, i=_
                                        if i > j
                                            y[i, j] = y[j, i]
```

4.2.3 Concordize Tensors. Transpose tensors to make the iteration of indices concordant [5]. A program is concordant when the order of indices in each tensor access match the order in which loops are nested around it. If necessary, transpose the tensor *and* reorder the loops to make iteration concordant. This step is critical for sparse tensors, as we can only iterate over hierarchical sparse formats with a concordant traversal. Concordant traversal is faster for dense tensors as well. Given a particular loop order and set of tensors, we prioritize transposing the tensors with fewer dimensions (and/or modifying the loop order if necessary) over those with more dimensions for efficiency.

```
for j=_, k=_, i=_               for k=_, i=_, j=_
    C[i, j] += A[i, k] * B[k, j]    C_T[j, i] += A[i, k] * B_T[j, k]
    C[k, j] += A[i, k] * B[i, j]    C_T[j, k] += A[i, k] * B_T[j, i]
```

4.2.4 Consolidate Conditional Blocks. Identify conditional blocks containing equivalent assignments and replace them with a single conditional block with an if-condition that is the union of the if-conditions of each of the conditional blocks. This transform improves the readability of the generated kernel and also prevents unnecessary specialization of cases during Finch compilation.

```
if i == j                          if (i == j) || (i < j)
    y[i] += A[i, j] * x[j]             y[i] += A[i, j] * x[j]
if i < j
    y[i] += A[i, j] * x[j]
```

4.2.5 Simplicial Lookup Table.
Given multiple conditional blocks with the same assignments but with different constant factors, we combine them into a single block and generate a lookup table that is used to determine the constant factor. We index into the lookup table using some product of primes based on which indices are equivalent.

```
if (i != k) && (k != l)            lookup_table = [2, 0, 1, 1, 0, 0, 1/3]
    C[l, j] += 2 * A[i, k, l] * B[i, j]   idx = 2 * (i == k) + 3 * (k == l) + 1
    C[k, j] += 2 * A[i, k, l] * B[l, j]   factor = lookup_table[idx]
    C[i, j] += 2 * A[i, k, l] * B[k, j]
if ((i != k) && (k == l))          C[l, j] += factor * A[i, k, l] * B[i, j]
      || ((i == k) && (k != l))    C[k, j] += factor * A[i, k, l] * B[l, j]
    C[l, j] += A[i, k, l] * B[i, j]       C[i, j] += factor * A[i, k, l] * B[k, j]
    C[k, j] += A[i, k, l] * B[l, j]
    C[i, j] += A[i, k, l] * B[k, j]
if (i == k) && (k == l)
    C[l, j] += A[i, k, l] * B[i, j]
```

4.2.6 Group Assignments Across Branches.
Many of the same assignments are performed in different branches in the code generated from the symmetrization process. Restructure and reorganize the generated code such that each assignment is called only once. This particular transform is beneficial when the total number of unique assignments (after applying the previous transforms) is less than the number of conditional blocks and we only apply it when this is the case; it also improves the readability of the generated kernel and prevents unnecessary specialization of cases during compilation.

```
if i < j                           if i < j || i == j
    y[i] += A[i, j] * x[j]             y[i] += A[i, j] * x[j]
    y[j] += A[i, j] * x[i]          if i < j
if i == j                              y[i] += A[i, j] * x[i]
    y[i] += A[i, j] * x[j]
```

4.2.7 Distributive Assignment Grouping.
Replace N equivalent additions in a conditional block with a single addition that multiples the right-hand side by N.

```
y[i] += A[i, j] * x[j]             y[i] += 2 * A[i, j] * x[j]
y[i] += A[i, j] * x[j]
```

4.2.8 Workspace Transformation.
Replace a write to the output tensor in an assignment with a write to a temporary variable defined just inside the innermost loop L that iterates through an index used to access the output tensor in the assignment. Accumulate updates in this temporary variable and write back the sum to the output tensor just at the end of this loop. This is worthwhile to do when there are more for loops inside L.

```
for j=_, i=_                       for j=_
    y[i] += A[i, j] * x[j]             temp = 0
    y[j] += A[i, j] * x[i]             for i=_
                                           y[i] += A[i, j] * x[j]
                                           temp += A[i, j] * x[i]
                                       y[j] += temp
```

4.2.9 Diagonal Splitting.
Moving specific conditional blocks into a separate loop nest. Because non-diagonal values form the bulk of the values in a tensor, we can think of assignments that involve the diagonal entries of a symmetric tensor as an edge case and compute them separately. In particular, we can move the conditional blocks involving non-diagonal entries in a separate loop nest.

```
for j=_, i=_                       for j=_, i=_
    if i != j                          if i != j
        y[i] += A[i, j] * x[j]             y[i] += A[i, j] * x[j]
    if i == j                          for j=_, i=_
        y[i] += A[i, j] * x[j]             if i == j
                                               y[i] += A[i, j] * x[j]
```

4.3 MTTKRP Demonstration

```
1:  for j = 1 : _, l = 1 : _, k = 1 : _, i = 1 : _ do
2:      if i ≤ k ≤ l then
3:          if E = {(i), (k), (l)} then
4:              for all σ ∈ {(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)} do
5:                  (i, k, l) = σ((i, k, l))
6:                  C[i, j] = A[i, k, l] * B[l, j] * B[k, j]
7:              end for
8:          end if
9:          if E = {(i = k), (l)} then
10:             for all σ ∈ {(1, 2, 3), (1, 3, 2), (3, 1, 2)} do
11:                 (i, k, l) = σ((i, k, l))
12:                 C[i, j] = A[i, k, l] * B[l, j] * B[k, j]
13:             end for
14:         end if
15:         if E = {(i), (k = l)} then
16:             for all σ ∈ {(1, 2, 3), (2, 1, 3), (3, 1, 2)} do
17:                 (i, k, l) = σ((i, k, l))
18:                 C[i, j] = A[i, k, l] * B[l, j] * B[k, j]
19:             end for
20:         end if
21:         if E = {(i = k = l)} then
22:             for all σ ∈ {(1, 2, 3)} do
23:                 (i, k, l) = σ((i, k, l))
24:                 C[i, j] = A[i, k, l] * B[l, j] * B[k, j]
25:             end for
26:         end if
27:     end if
28: end for
```

Figure 5. MTTKRP Symmetrization: We construct the unique symmetry groups given each equivalence group.

Let us apply this technique to the MTTKRP kernel given by $C[i, j] = A[i, k, l] * B[l, j] * B[k, j]$. If A is fully-symmetric, the set of permutable indices is given by $P = \{i, k, l\}$ and we

can establish ordering i, k, l—such that if these indices are monotonically increasing, we will only access the canonical triangle of A. The equivalence groups that can be constructed from P and which satisfy the that $i \le k \le l$ are $\{(i), (k), (l)\}$, $\{(i = k), (l)\}$, $\{(i), (k = l)\}$, and $\{(i = k = l)\}$.

Next, we determine the unique symmetry group $S_P|E$ for each equivalence group E. For instance, for equivalence group $\{(i = k), (l)\}$, $S_P|E = \{(1, 2, 3), (1, 3, 2), (3, 1, 2)\}$. Thus, the pseudocode in Figure 3 expands to Figure 5. The normalized equivalent is given by Listing 6.

```
1  function mttkrp(C, A, B)
2    for l=_, j=_, k=_, i=_
3      if i <= k && k <= l
4        if i != k && k != l
5          C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
6          C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
7          C[k, j] += A[i, k, l] * B[i, j] * B[l, j]
8          C[k, j] += A[i, k, l] * B[i, j] * B[l, j]
9          C[l, j] += A[i, k, l] * B[i, j] * B[k, j]
10         C[l, j] += A[i, k, l] * B[i, j] * B[k, j]
11       if i == k && k != l
12         C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
13         C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
14         C[l, j] += A[i, k, l] * B[i, j] * B[k, j]
15       if i != k && k == l
16         C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
17         C[k, j] += A[i, k, l] * B[i, j] * B[l, j]
18         C[k, j] += A[i, k, l] * B[i, j] * B[l, j]
19       if i == k && k == l
20         C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
```

Listing 6. Normalized Symmetric MTTKRP Kernel

After performing common tensor access elimination, distributive assignment grouping, consolidating conditional blocks, diagonal splitting, and lastly, concordizing tensors, we obtain the code given by Listing 7.

```
1  function mttkrp(C, A_nondiag, A_diag, B)
2    for l=_, k=_, i=_, j=_
3      A = A_nondiag[i, k, l], B_ji = B_T[j, i], B_jk =
           B_T[j, k], B_jl = B[j, l]
4      if i <= k && k <= l
5        if i != k && k != l
6          C_T[j, i] += 2 * A * B_jk * B_jl
7          C_T[j, k] += 2 * A * B_ji * B_jl
8          C_T[j, l] += 2 * A * B_ji * B_jk
9    for l=_, k=_, i=_, j=_
10     A = A_diag[i, k, l], B_ji = B_T[j, i], B_jk = B_T
           [j, k], B_jl = B[j, l]
11     if i <= k && k <= l
12       if (i == k && k != l) || (i != k && k == l)
13         C_T[j, i] += A * B_jk * B_jl
14         C_T[j, l] += A * B_ji * B_jk
15         C_T[j, k] += A * B_ji * B_jl
16       if i == k && k == l
17         C_T[j, i] += A * B_jk * B_jl
```

Listing 7. MTTKRP: Separate Loop Nests

5 Evaluation

5.1 Implementation

We implemented the SySTeC compiler in Julia and demonstrated that the performance of the generated kernels was competitive on the SSYMV, SYPRD, SSYRK, TTM, and MT-TKRP operations when compared against the naive Finch implementation[4], TACO [19], symmetric MKL [2], and SPLATT [31]. The implementation is available on github [1].

Provided a single Finch assignment and a list of symmetric tensors, SySTeC outputs an executable kernel in Finch IR that exploits symmetry. SySTeC uses RewriteTools[3], the same rewriting package used by Finch [5], to define a set of simplification rules and identify specific control structures, einsums, and operations to which these rules are applied.

SySTeC generates code in two phases according to Section 4: in the *symmetrization* phase, the compiler first generates an executable kernel that only accesses the canonical triangle. In the *optimization* phase, the compiler performs transforms to reduce operation count. Each transform from Section 4.2 has been mapped into a rewrite rule that is applied if applicable.

5.2 Results

All experiments were run on a single core of a 12-core 2-socket Intel Xeon CPU E5-2680 v3 running at 2.50GHz with 128GB of memory. We used v0.6.22 of the Finch library to implement the kernels and executed both the naive and optimized implementation generated by SySTeC. We compare all kernels to the column-major implementations in TACO, and additionally SSYMV to MKLSparse v1.1.0 and MTTKRP to SPLATT. We used Julia v1.10 to run the tests and all timings are the minimum of 10,000 runs or 5s of measurement, whichever happens first. The time to rearrange data before or after each kernel is not included in the timings, including transposition or replicating the output.

Table 2. Matrix collection used in the experiments, taken from Vuduc et. al [37].

Name	Dimension	Nonzeros	Name	Dimension	Nonzeros
bayer02	13935	63679	olafu	16146	1015156
bayer10	13436	94926	onetone2	36057	227628
bcsstk35	30237	1450163	orani678	2529	90185
coater2	9540	207308	raefsky3	21200	1488768
crystk02	13965	968583	raefsky4	19779	1328611
crystk03	24696	1751178	rdist1	4134	94408
ct20stif	52329	2698463	rim	22560	1014951
ex11	16614	1096948	saylr4	3564	22316
finan512	74752	596992	sherman3	5005	20033
gemat11	4929	33185	sherman5	3312	20793
goodwin	7320	324784	shyy161	76480	329762
lhr10	10672	232633	venkat01	62424	1717792
lnsp3937	3937	25407	vibrobox	12328	342828
memplus	17758	126150	wang3	26064	177168
nasasrb	54870	2677324	wang4	26068	177196

For the SSYMV, SYPRD, and SSYRK kernels, we evaluated with the matrix benchmark suite used by Vuduc et. al [37],

---

[1] https://github.com/radha-patel/symmetry-compiler

shown in Table 2. The asymmetric matrices in the suite were symmetrized by summing the transpose (i.e. $A + A^T$). To our knowledge, there does not exist a database of symmetric tensors so for the MTTKRP kernels, we generated uniformly distributed symmetric random sparse tensors of varying sizes and sparsities via an Erdős–Rényi distribution. The dense input matrices are also randomly generated. The numerical rank is the number of columns in the dense matrix, where applicable.

In Figures 6-11, we normalize all results to naive Finch; the red line specifies the performance of naive Finch (our baseline) and the purple line the speedup we expect.

5.2.1 SSYMV. The sparse symmetric matrix vector kernel is given by $y[i] = A[i, j] * x[j]$ where A is symmetric and CSC, and y and x are dense.

The optimized kernel accesses only $\frac{1}{2}$ of the values of A, but performs all of the computations. In cases where SSYMV is bandwidth bound, we can expect a speedup approaching 2x, however we don't expect any computational savings here. We find that SySTeC is 1.45, 1.45, and 1.90 times faster on average than the naive Finch implementation, TACO, and MKL's `mkl_dcsrsymv`, respectively (Figure 6). MKL was the only commercial sparse SYMV implementation for cpu we found, but is either not taking advantage of symmetry or not optimized for a single threaded case. TACO may be faster than naive Finch because it emits simpler loop bounds for SPMV that are more amenable to compiler optimizations.

5.2.2 Bellman-Ford Update. The sparse symmetric Bellman-Ford update kernel is given by $y[i]min = A[i, j] + d[j]$ where A is symmetric and CSC, and y and d are dense. Here, d represents shortest path lengths after k steps, A represents the edge distances, and y represents the shortest path lengths after $k + 1$ steps. This kernel is identical to SSYMV from a performance perspective, as seen in Figure 7, but is included to show that SySTeC can handle symmetrizing operations beyond + and *.

5.2.3 SYPRD. The symmetric triple product kernel is given by $y[] = x[j] * A[i, j] * x[i]$ where A is symmetric and CSF and y and x are dense.

The optimized kernel accesses $\frac{1}{2}$ of the values of A and performs $\frac{1}{2}$ of the computations because we have $\{\{i, j\}\}$ invisible symmetry in C. As n grows, we can expect a speedup of 2x. We find that SySTeC is 1.79 and 1.46 times faster on average than naive Finch and TACO (Figure 8). We may not have achieved 2x speedup everywhere because the symmetric code needs to terminate the iteration through the sparse matrix early to restrict to the triangle, which complicates the exit condition of the loop.

5.2.4 SSYRK. The sparse symmetric rank-k update is given by $C[i, j] = A[i, k] * A[j, k]$ where A is *not* symmetric, but

by nature of the computation, C *is* symmetric. A and C are both CSF.

The optimized kernel accesses all values of A because A is not symmetric, but performs only $\frac{1}{2}$ of the computations and writes to C because we exploit the $\{\{i, j\}\}$ output visible symmetry in C. Because SSYRK is compute-bound, we expect a speedup of 2x. We find that SySTeC is 2.20 times faster than naive Finch (Figure 9). TACO does not support the outer products implementation of SSYRK. We believe we exceed the expected speedup due to increased reuse of rows of A in the point of the triangle.

5.2.5 TTM. The tensor times matrix kernel is given by $C[i, j, l] = A[k, j, l] * B[k, i]$ where A is fully symmetric CSF, and B and C are dense.

The optimized kernel accesses only $\frac{1}{6}$ of the values of A and performs $\frac{1}{2}$ of the computations (and hence writes $\frac{1}{2}$ of the values to C) because we take advantage of the $\{\{j, l\}\}$ symmetry in C. We can therefore expect a speedup of at least 2x. We find that SySTeC is 2.09 and 1.13 times faster than naive Finch and TACO, respectively, with high density and low numerical rank. SysTeC underperforms naive Finch for high numerical rank because the overhead of initializing the dense output outweighs the cost of the computation.

5.2.6 MTTKRP. The MTTKRP (matricized tensor times Khatri-Rao product) kernel is used in tensor factorization, namely to compute the CPD (Canonical Polyadic Decomposition) of a tensor [21]. Typically, the factorization of an N-dimensional tensor requires N MTTKRP kernels, each executed on a different transposition of the tensor. When the tensor is symmetric, no transpose is required because all transpositions of the tensor are equivalent. While CPD typically involves separate factor matrices for each dimension, the symmetric CPD problem uses the same factor matrix for all dimensions. This means that the update step is identical in all the modes, and the algorithm is more efficient [20, Algorithm 2].

The assignments for the 3-, 4-, and 5-dimensional kernels are given below where A is CSF and B and C are dense.

$$C[i, j] = A[i, k, l] * B[k, j] * B[l, j]$$
$$C[i, j] = A[i, k, l, m] * B[k, j] * B[l, j] * B[m, j]$$
$$C[i, j] = A[i, k, l, m, n] * B[k, j] * B[l, j] * B[m, j] * B[n, j]$$

The optimized kernels our compiler implementation generates for MTTKRP consist of two loop nests, one that handles the triangles and another to handle the diagonals to simplify control flow logic. For the 3D case, the optimized kernel accesses only $\frac{1}{6}$ of the values of A and performs $\frac{1}{2}$ of the computations because we have $\{\{k, l\}\}$ invisible symmetry in C. For the 4D case, the optimized kernel accesses only $\frac{1}{4!} = \frac{1}{24}$ of the values of A and performs $\frac{1}{3!} = \frac{1}{6}$ of the computations because we have $\{\{k, l, m\}\}$ invisible symmetry
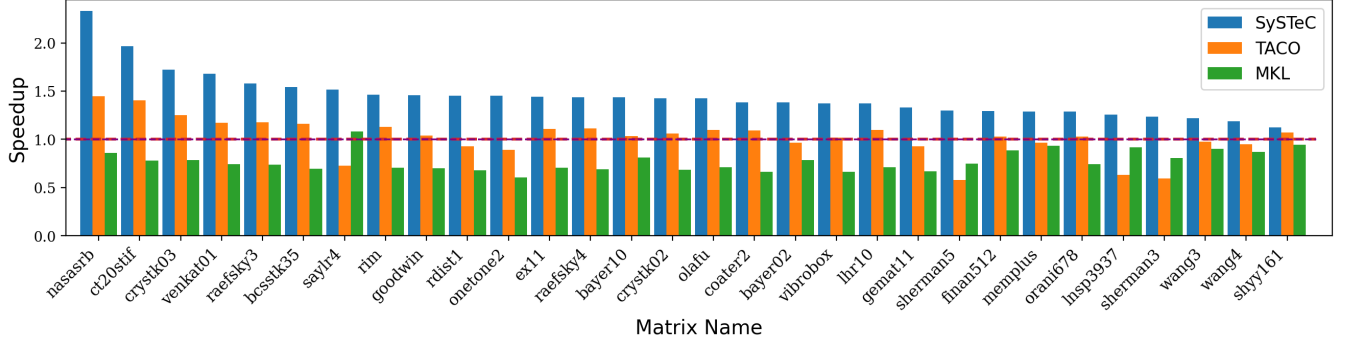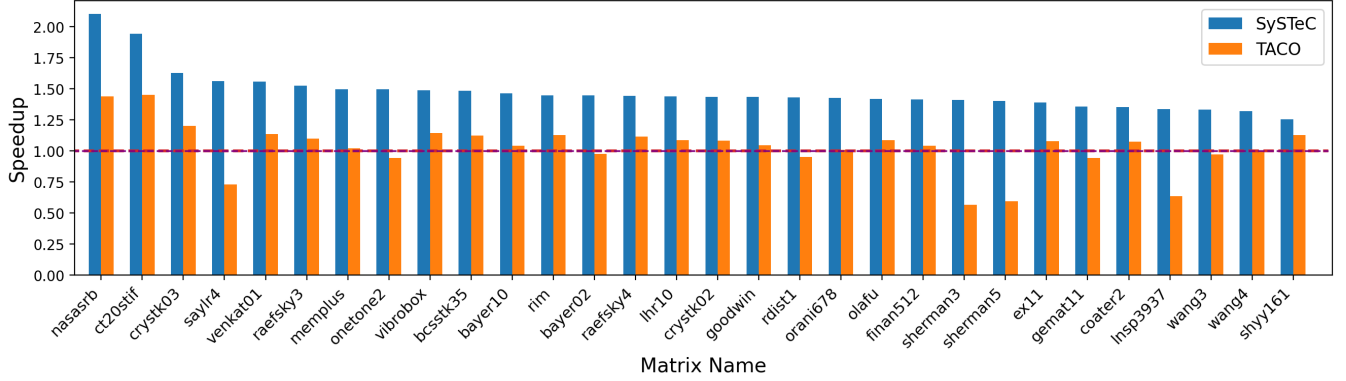
Figure 6. SSYMV Performance
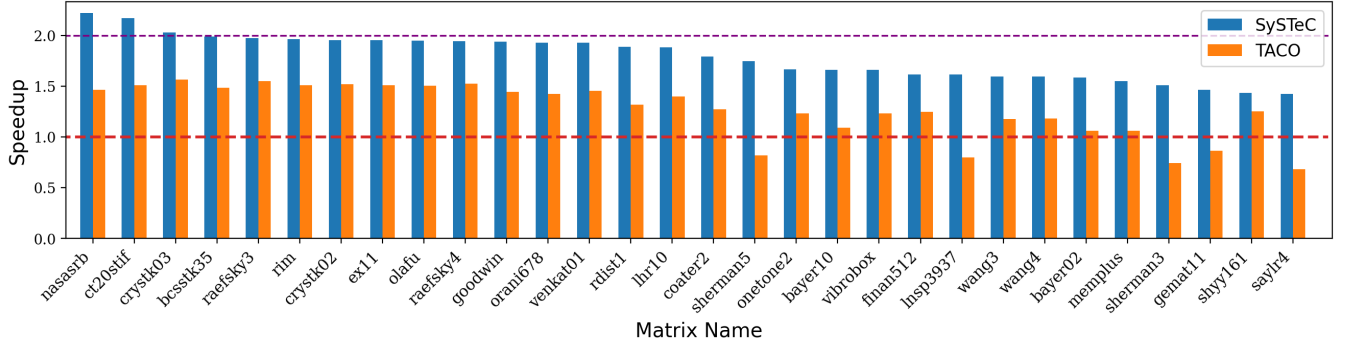

Figure 7. Bellman-Ford Step Performance

Figure 8. SYPRD Performance

in C. Lastly, for the 5D case, the optimized kernel accesses only $\frac{1}{5!} = \frac{1}{120}$ of the values of A and performs $\frac{1}{4!} = \frac{1}{24}$ of the computations because we have $\{\{k, l, m, n\}\}$ invisible symmetry in C. Thus, we expect speedups of 2x, 6x, and 24x and obtain maximal speedups of 3.38, 7.35, and 29.8 times for 3-, 4-, and 5-dimensional MTTKRP, respectively, with SySTeC over naive Finch (Figure 11). We attribute the above-expected speedups over naive Finch to register reuse of the input tensors in the symmetric code.

6 Related Work

Several tensor compilers support sparse tensor operations, but have no support for tensor symmetry, such as TACO [19], SparseLNR [13], SPARTA [23], and MLIR [8]. We therefore expect these systems would perform similarly to TACO in our evaluation.

Directly related techniques fall into three categories: Libraries which collect hand-specialized symmetric kernels, compilers which reduce symmetric problems to multiple
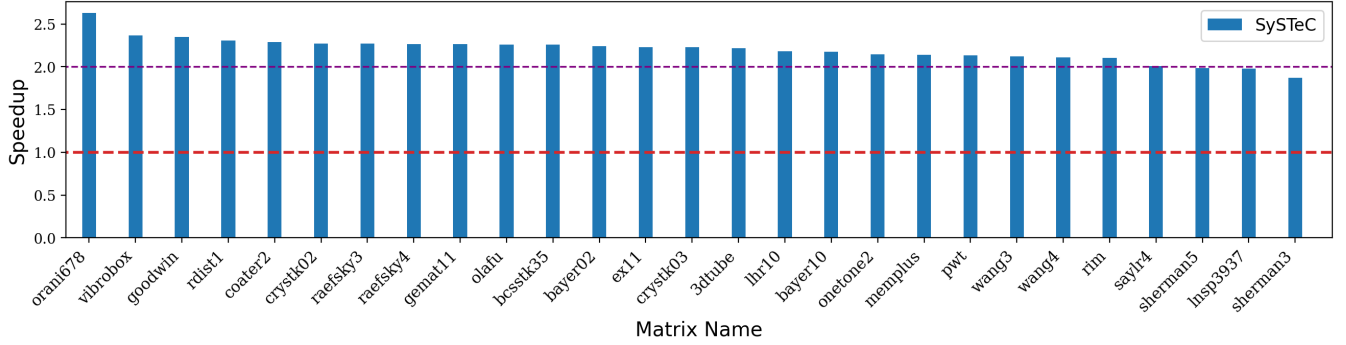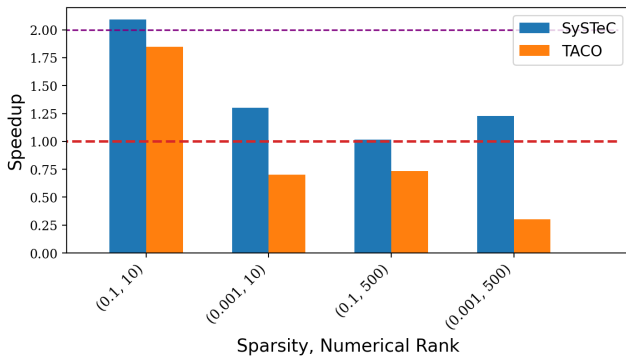
Figure 9. SSYRK Performance

Figure 10. TTM Performance

asymmetric or hand-written kernels, and compilers which produce direct solutions to symmetric problems.

Many libraries contain at least the dense symmetry-specific functions specified in the BLAS [7], such as ATLAS [11], MKL [2], and CuBLAS [1]. Of these, only MKL implements multiple sparse symmetric kernels. CuBLAS only handles sparse symmetry in SpMV. This reflects an implementation burden that further motivates our work.

Most of the work on symmetric compilers reduces symmetric problems to other kernels. The most notable of these is the Cyclops Tensor Framework (CTF). CTF reduces N-dimensional dense symmetric contractions to $N!$ separate triangular contractions, linearizes triangular indices which are preserved in the output, then dynamically loops over the remaining triangular indices and repeatedly calls matrix multiply, saving compute and storage [33, 34]. However, this approach does not extend to kernels which are not contractions (such as MTTKRP) or do not use + and ∗ (such as min-+ semiring multiplication [9]), and cannot benefit from within-kernel reuse of the redundant arguments that are produced by the problem reduction. This approach also requires transposing and reformatting arguments before running the kernel, which may be expensive in comparison to the cost of the kernel. It also only supports one sparse argument at a time (likely due to the complexity of sparse-sparse interactions in the triangular loops). The OpMin system provides an operation optimization process that identifies the optimal ordering of tensors in a tensor contraction, but does not produce the code to compute the contraction [22].

Few compilers produce code that directly computes symmetric kernels. Shi et. al. proposes to use an output-oriented loop structure which iterates through the unique inputs needed to compute the result, but the corresponding random access of the symmetric input resulted in poor performance [29]. STUR symbolically optimizes kernels based on the structure (triangular, banded, symmetric, etc.) of the arguments using a term rewriting approach, but cannot handle dynamic sparsity patterns, and is not as specialized to symmetry [15]. Similarly, Spampinato proposes a polyhedral approach (sBLACs) to generating structured code, but does not address dynamic sparsity or even tensors of dynamic sizes [35]. The ITensor library has several library routines for symmetric tensors within tensor networks, but these routines do not handle unstructured sparsity either [14].

Other works propose more specialized techniques for symmetric tensors which we do not attempt in this work. Solomonik also proposes a Strassen-like algorithm to reduce operation count with the symv, syr2, syr2k, and symm kernels [32], which is beyond the scope of this paper. The Blocked Compact Symmetric format proposed by Schatz et. al for the TTM kernel breaks tensors into blocks, processing only canonical blocks of symmetric tensors [27]. However, the implementation does not handle sparse tensors, and cannot optimize diagonal blocks.

7 Future Work

Several avenues for future research remain:

1. **Generalizing to More Types of Symmetry**: We can expand and adapt our methodology to encompass other forms of symmetry like antisymmetry, block symmetry, or cyclic symmetry that commonly arise in
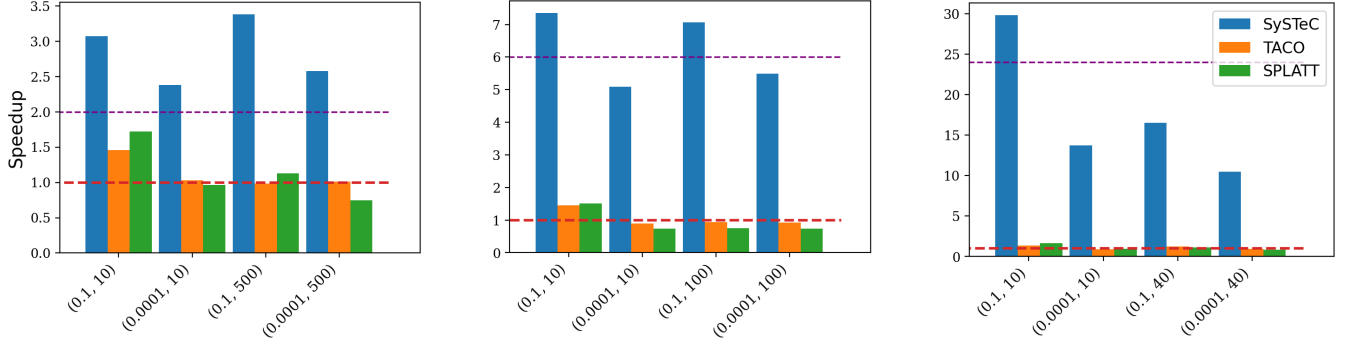
Figure 11. 3-, 4-, and 5-dimensional MTTKRP performance over varying sparsity and numerical rank

physics, mathematics, chemistry, and machine learning.

2. **Exploring Parallelization Opportunities**: Incorporating parallelization is important for increasing the practicality of our system, and are actively working towards that goal. In distributed memory settings, the block-cyclic distribution employed by Cyclops [34] would decompose the problem into several smaller, symmetric instances that we could process with SySTeC out of the box. In multicore settings, we plan to tag an outer loop as parallel. Of course, because iteration over the canonical triangle modifies different transpositions of the output at once, we will also need to make atomic updates to the output.

3. **Fine-Tuning Symmetric Optimizations**: Data formats tailored to symmetry that provide more efficient access when iterating over different transpositions of the canonical triangle could significantly improve performance. Symmetry-aware formats could also eliminate or simplify extra post-processing steps like replicating the canonical triangle of a tensor to the non-canonical triangles. Additionally, our implementation of Systec fully optimizes for every symmetry in the kernel, but we could develop approaches to partially optimize for symmetry only when it is beneficial.

8   Conclusion

In this paper, we demonstrated a systematic approach to exploit symmetry in arbitrary tensor kernels. We identified core strategies to exploit symmetry in tensor kernels, including memory read reuse and redundant computation filtering. We also proposed a detailed compiler methodology for mechanically generating and optimizing symmetric code. This methodology involved two stages: first, symmetrizing the kernel such that we only access the canonical triangle of symmetric inputs, and secondly, applying a set of transforms to further optimize the code. We ultimately implemented this methodology in a Julia-based compiler and evaluated

its performance on several common tensor kernels, showing significant speedups.

This work provides a strong foundation for exploiting symmetry in tensor kernels.

9   Data Availability Statement

The data that support the findings of this study are openly available in Zenodo at https://zenodo.org/records/13821280. [25].

A   Artifact Appendix

A.1   Abstract

This artifact provides all of the necessary tools to reproduce the program compilation and charts presented in the paper, with the exception of SSYRK (which takes too much time and memory), and MKL, which is too cumbersome to install. The datasets given to TTM and MTTKRP are smaller here than in the paper. We claim that all speedups should be within 1.5 of the expected, with few exceptions. We note that there was a bug in the TACO TTM results in the submitted version which has been fixed here, so the results may not match the submitted version, but do match the camera ready.

A.2   Artifact Check-list (Meta-Information)

- **Algorithm:** We present a new compilation algorithm, SySTeC, which accepts pointwise einsums with symmetry-annotated tensors and produces symmetry-optimized Finch tensor programs.
- **Program:** We benchmark on the SSYMV, SYPRD, TTM, and MTTKRP Kernels. We compare against naive Finch, TACO, and SPLATT, all of which are included.

- **Compilation:** We require Finch v0.6.32 and Julia v1.10.4 to compile.
- **Binary:** We provide a Docker image `systec-artifact.tar` with all dependencies pre-installed, which requires ARM hardware to run as it was built for Apple M2. If different hardware is used, one can build the docker image for that hardware using the provided instructions.
- **Data set:** Matrix datasets are downloaded automatically from http://sparse.tamu.edu/.
- **Run-time environment:** The artifact requires a Unix System (Mac or Linux).
- **Hardware:** We benchmark on an Intel Xeon CPU E5-2680 v3 running at 2.50GHz, and our benchmarks run well on Apple M2.
- **Execution:** We assume that TACO and SPLATT will run in single-threaded mode.
- **Metrics:** We measure runtime of all methods we test.
- **Output:** The numerical output of each kernel is collected and validated against comparison methods in the script itself. The experiments themselves output json files in the same subdirectory they run in. The artifact includes instructions to plot the results.
- **Experiments:** We claim that our speedup results should be within 1.5 of the expected, with few exceptions.
- **How much disk space required (approximately)?:** 4GB
- **How much time is needed to prepare workflow (approximately)?:** The artifact can be set up quickly with Docker, in around an hour. Building by hand also takes less than an hour, but it may take longer if your system needs special build flags.
- **How much time is needed to complete experiments (approximately)?:** The experiments take around an hour to run.
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** MIT
- **Workflow framework used?:** Docker
- **Archived (provide DOI)?:** Zenodo doi.org/10.5281/zenodo.13377504

A.3 Description

A.3.1 How Delivered. There are multiple sources:
Download from Zenodo: doi.org/10.5281/zenodo.13377504
Download from Github: https://github.com/radha-patel/symmetry-benchmarks/tree/systec-cgo-artifact
Disk space required: 4GB

A.3.2 Hardware Dependencies. These experiments require at least 24GB of memory and 4GB disk space. A network connection is required to download the matrix datasets.

We benchmark on an Intel Xeon E5-2695 v2 @ 2.40GHz, and our benchmarks run well on Apple M2.

A.3.3 Software Dependencies. A Unix (Mac or Linux) system is required to run the artifact. ARM hardware is required to use the pre-built docker image included with the artifact, or one can build their own docker image or build from source.

Dependencies which must be downloaded are documented in `Dockerfile`:

- Julia v1.10.4
- coreutils
- cmake
- gcc
- g++
- python
- python3
- python3-pip
- python3-venv
- git
- libblas-dev
- liblapack-dev
- poetry

Dependencies which are built from source with `make deps`:

- TACO https://github.com/tensor-compiler/taco/commit/1278503a1c859d557174a4ef2ae7a85295f39f69
- SPLATT https://github.com/ShadenSmith/splatt/commit/6cb86283c1fbfddcc67c2564e025691de4f784cf

Julia deps are also collected in a Project.toml file which can be installed with `julia setup.jl` or just `make env`.

- ArgParse v1.2.0
- BenchmarkTools v1.5.0
- DataStructures v0.18.20
- Finch v0.6.32
- JSON v0.21.4
- MatrixDepot v1.0.13
- SparseArrays v1.10.0
- SySTeC v0.1.0 https://github.com/radha-patel/SySTeC/commit/b0ec98927f0d2be01a61b48646cbadaa92040b0f
- TensorMarket v0.2.0

Python deps are collected in a pyproject.toml file which can be installed with `poetry install --no-root` or just `make env`.

- python = v3.9
- matplotlib = v3.9.0
- packaging = v24.0
- numpy = v1.26.4

A.3.4 Data Sets. All matrix datasets are automatically downloaded with a network connection from http://sparse.tamu.edu/

The tensor datasets are randomly generated by the benchmarks themselves.

A.4 Installation

A.4.1 Docker. Installation is greatly simplified if you are using the provided Docker image. First load the image with

```
docker image load systec-artifact.tar
```

or build it with

```
docker build -t systec-artifact .
```

Then, you can launch a shell in the container named "evaluator-container" with

```
docker run -it --name evaluator systec-artifact
    /bin/bash
```

Be sure to configure Docker to use enough memory (24GB) and disk space (100 GB) for the experiments.

A.4.2 Manual. Once the basic dependencies are installed on your system, you should be able to run `make` to build all the necessary dependencies.

Should you need to customize the workflow, `make  deps` builds the dependencies, `make  env` initializes Python and Julia Environments, and `make  kernels` builds the benchmarking kernels.

A.5 Experiment Workflow

There are three main phases to the experiments:

A.5.1 Run SySTeC to Compile the Kernels. From the toplevel directory, you can run the SySTeC compiler with

```
julia run_SySTeC.jl
```

This will compile all the kernels required for the experiments and output them into the `generated/` directory. The script itself can be inspected and modified if you wish to compile different kernels and compare the output code. The SySTeC compiler itself is contained in `deps/SySTeC/src/SySTeC.jl`.

A.5.2 Run the Experiements. You can run the experiments with

```
sh run_benchmarks.sh
```

Benchmarks took us a little over an hour to run. The benchmarks may be run individually with the individual commands in the `run_benchmarks.sh` script. Results will be generated in the toplevel directory. Corresponding reference results used in the publication are available with the filename suffix `_reference.json`.

A.5.3 Plot the Results. You can plot the results with

```
poetry run python plot_results.py
```

This will generate plots for whichever experiments you have run and saved results for. The plots will be saved in the `charts/` directory. Corresponding reference results used in the publication are available with the filename suffix `_reference.json`.

A.6 Evaluation and Expected Result

You should expect to see similar results to the paper, notably SSYMV (Figure 6), SYPRD (Figure 8), TTM (Figure 10), and MTTKRP (Figure 11). We expect that the more dramatic speedups will be seen in the MTTKRP benchmarks, as the compiler is able to exploit the most symmetry in 5-dimensional kernels. Note that we have reduced the size of the tensors in TTM and MTTKRP to keep the runtime and storage manageable, so these results may demonstrate slightly less speedup as more time is spent on diagonal edge cases.

If you are using our provided Docker image or using Docker to run the experiments, you can view the charts by copying them to the host.

```
docker cp evaluator:symmetry-benchmarks/charts .
```

A.7 Experiment Customization

The evaluator may optionally customize the kernel given to SySTeC by modifying the `run_SySTeC.jl` script. The size of the tensors in our TTM and MTTKRP experiments can be changed by modifying the scripts directly.

A.8 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] 2024. cuSPARSE. https://docs.nvidia.com/cuda/cusparse/index.html
[2] 2024. Developer Reference for Intel® oneAPI Math Kernel Library for Fortran. (April 2024). https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-fortran/2024-0/overview.html
[3] 2024. RewriteTools.jl. https://github.com/willow-ahrens/RewriteTools.jl
[4] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2024. Finch: Sparse and Structured Array Programming with Control Flow. *arXiv preprint arXiv:2404.16730* (2024).
[5] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2023)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3579990.3580020
[6] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3519939.3523442
[7] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719604

[8] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Transactions on Architecture and Code Optimization* 19, 4 (Sept. 2022), 50:1–50:25. https://doi.org/10.1145/3544559

[9] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017. Design of the GraphBLAS API for C. In *2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 643–652.

[10] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 123:1–123:30. https://doi.org/10.1145/3276493

[11] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (Jan. 2001), 3–35. https://doi.org/10.1016/S0167-8191(00)00087-9

[12] Pierre Comon, Gene Golub, Lek-Heng Lim, and Bernard Mourrain. 2008. Symmetric Tensors and Symmetric Tensor Rank. *SIAM J. Matrix Anal. Appl.* 30, 3 (Jan. 2008), 1254–1279. https://doi.org/10.1137/060661569 Publisher: Society for Industrial and Applied Mathematics.

[13] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. 2022. SparseLNR: accelerating sparse tensor computations using loop nest restructuring. In *Proceedings of the 36th ACM International Conference on Supercomputing*. ACM, Virtual Event, 1–14. https://doi.org/10.1145/3524059.3532386

[14] Matthew Fishman, Steven White, and Edwin Stoudenmire. 2022. The ITensor Software Library for Tensor Network Calculations. *SciPost Physics Codebases* (Aug. 2022), 4. https://doi.org/10.21468/SciPostPhysCodeb.4

[15] Mahdi Ghorbani, Mathieu Huot, Shideh Hashemian, and Amir Shaikhha. 2023. Compiling Structured Tensor Algebra. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 229:204–229:233. https://doi.org/10.1145/3622804

[16] Colin R Goodall. 1993. Computation Using the QR Decomposition. Elsevier. Section: 13.

[17] Howard H. Hu. 2012. Fluid Mechanics. Academic Press, 421–272. Section: Chapter 10 - Computational Fluid Dynamics.

[18] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and Jose Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. https://doi.org/10.1109/HPEC.2016.7761646

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

[20] Eleftherios Kofidis and Phillip A. Regalia. 2002. On the Best Rank-1 Approximation of Higher-Order Supersymmetric Tensors. *SIAM J. Matrix Anal. Appl.* 23, 3 (Jan. 2002), 863–884. https://doi.org/10.1137/S0895479801387413

[21] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (Aug. 2009), 455–500. https://doi.org/10/dzcrv6 Publisher: Society for Industrial and Applied Mathematics.

[22] Pai-Wei Lai, Huaijian Zhang, Samyam Rajbhandari, Edward Valeev, Karol Kowalski, and P. Sadayappan. 2012. Effective Utilization of Tensor Symmetry in Operation Optimization of Tensor Contraction Expressions. *Procedia Computer Science* 9 (Jan. 2012), 412–421. https://doi.org/10.1016/j.procs.2012.04.044

[23] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: high-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3437801.3441581

[24] Román Orús. 2019. Tensor networks for complex quantum systems. *Nature Reviews Physics* 1 (2019), 538–550.

[25] Radha Patel, Willow Ahrens, and Saman Amarasinghe. 2024. SySTeC: A Symmetric Sparse Tensor Compiler: The Artifact. https://doi.org/10.5281/zenodo.13821280

[26] Yousef Saad. 2003. *Iterative methods for sparse linear systems* (2nd ed.). SIAM, Philadelphia.

[27] Martin D Schatz, Tze Meng Low, Robert A van de Geijn, and Tamara G Kolda. 2013. Exploiting symmetry in tensors for high performance. *arXiv preprint arXiv:1301.7744* (2013). Publisher: Citeseer.

[28] James R Schott. 2016. *Matrix Analysis for Statistics.* John Wiley I& Sons.

[29] Jessica Shi, Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2021. An Attempt to Generate Code for Symmetric Tensor Computations. https://doi.org/10.48550/arXiv.2110.00186 arXiv:2110.00186 [cs].

[30] Harmanjit Singh and Richa Sharma. 2012. Role of adjacency matrix & adjacency list in graph theory. *International Journal of Computers & Technology* 3, 1 (2012), 179–183.

[31] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS '15)*. IEEE Computer Society, Washington, DC, USA, 61–70. https://doi.org/10.1109/IPDPS.2015.27

[32] Edgar Solomonik and James Demmel. 2021. Fast Bilinear Algorithms for Symmetric Tensor Contractions. *Computational Methods in Applied Mathematics* 21, 1 (Jan. 2021), 211–231. https://doi.org/10.1515/cmam-2019-0075 Publisher: De Gruyter.

[33] Edgar Solomonik and Torsten Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. *arXiv:1512.00066 [cs]* (Nov. 2015). http://arxiv.org/abs/1512.00066

[34] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 813–824. https://doi.org/10.1109/IPDPS.2013.112

[35] 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 '16)*. Association for Computing Machinery, New York, NY, USA, 117–127. https://doi.org/10.1145/2854038.2854060

[36] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[37] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. 2002. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 26–26. https://doi.org/10.1109/SC.2002.10025