A System to Exploit Symmetry in Common Tensor Kernels

by

Radha Patel

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Radha Patel Department of Electrical Engineering and Computer Science May 10, 2024
Certified by:	Saman Amarasinghe Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

A System to Exploit Symmetry in Common Tensor Kernels

by

Radha Patel

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Symmetric tensors arise naturally in many domains including linear algebra, statistics, physics, chemistry, and graph theory. Symmetry arises through both mathematical properties and scientific phenomena. Taking advantage of symmetry in matrices saves a factor of two, but taking advantage of symmetry in a tensor of order n can save a factor of $n!$ in memory accesses and operations. However, implementing this symmetry by hand significantly increases the complexity; for instance, leveraging symmetry in 2D BLAS nearly doubles the implementation burden, and this burden escalates further in the case of higher-dimensional tensors. Existing compilers to compute those kernels either do not take advantage of symmetry or do not take advantage of it to the extent possible. My thesis will identify and categorize methods to exploit symmetry in common and uncommon tensor kernels. We will depict a methodology to systematically generate and optimize symmetric code and will present a compiler in Julia that automates this process. Our symmetric implementation demonstrates significant speedups ranging from 1.36x for SSYMV to 7.95x for a 4-dimensional MTTKRP over the naive implementation of these kernels.

Thesis supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Firstly, a huge thanks to my research mentor Willow Ahrens, whose guidance has been invaluable throughout my journey as both an M.Eng. student and an undergraduate researcher in the COMMIT group. Working closely with Willow has enabled me to grow tremendously both academically and professionally. This project simply would not have been possible without Willow's mentorship, insight, and constant encouragement.

I would also like to thank my research advisor Saman Amarasinghe, whose wisdom and support have been instrumental in my development as a performance engineer. My interest in this field was first sparked by his course 6.106 Software Performance Engineering and my learning has only deepened from being involved in the COMMIT group. I would like to thank the other members of the COMMIT group as well, especially Teo Collin and Changwan Hong, for the valuable discussions and advice along the way.

Finally, I would like to thank my parents, Anjana and Rahul Patel, my sister Krishna Patel, and my friends for being there along the way and believing in me. As you know, I could not have done it without your support.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A symmetric tensor is a tensor that is invariant under a permutation of its indices (Figure 1.1). Tensors are often naturally symmetric because of the physical and chemical properties of substances and matter which produce symmetric interactions, structures, or reactions. In addition, mathematical operations often induce symmetry (e.g. computing A^TA), a consequence of the way we use tensor algebra.

$$A = \begin{bmatrix} 4 & -3 & 5 & 0 & 7 \\ -3 & 6 & -2 & 9 & -8 \\ 5 & -2 & 0 & -4 & -6 \\ 0 & 9 & -4 & -1 & -3 \\ 7 & -8 & -6 & -3 & 2 \end{bmatrix}$$

Figure 1.1: Example of a 5x5 symmetric matrix. Note that for all valid indices i, j, $A_{ij} = A_{ji}$.

There are applications of symmetric tensors in many domains. For example, in linear algebra, the hat matrix in linear regression and the Q matrix that is a result of QR factorization are both symmetric [10]. In statistics, matrices expressing covariance and other similarly commutative calculations are naturally symmetric [33]. In physics and chemistry computations, the properties of quantum tensor networks and computational fluid dynamics give way to multi-dimensional symmetry [28, 15]. In graph theory, all adjacency matrices of undirected graphs, used in algorithms like single-source shortest path and to find connected components, are also symmetric [36].

Given the ubiquitous nature of symmetric tensors, it is worthwhile to explore and implement systems to take advantage of symmetry. Optimizations with symmetric tensors can be categorized as being either (or both) storage-based or compute-based where the former typically avoid redundant storage and the latter avoid redundant computation. For instance, a storage-based optimization for the SSYMV kernel given by $y = Ax$ where A is symmetric would be storing and accessing only one triangle of matrix A (i.e. $\frac{1}{2}$ the values) to compute all of the output y. On the other hand, a compute-based optimization for the SSYRK kernel given by $C = A * A^T$ where A is not symmetric would be computing the values for just one triangle of C (i.e. $\frac{1}{2}$ the values) because C is symmetric, and then replicating the values to the other triangle of C or alternatively, storing C in a symmetry-aware format that does not need replication.

Existing compilers that take advantage of symmetry can be grouped into two classes: frameworks that restructure tensor operations based on symmetry to reduce computation and specialized libraries or systems that provide hand-written symmetric kernels that can be recombined to produce symmetry-aware programs, as will be described further in Section 2.1. The former encompasses systems that reorder tensor contractions to minimize operation count [18] as well as systems that utilize blocking to store and operate on only the unique values of a symmetric tensor [32, 39]. However, after reordering or blocking, such systems generally use programs that are not symmetry-aware to actually compute the output. On the other hand, the specialized libraries have a limited number of symmetrized tensor kernels [19]; hand-writing more specialized, symmetric kernels is cumbersome. Overall, there is a lack of a generalizable framework for how to generate symmetry-aware code. I seek to fill this void by providing such a framework that is applicable to arbitrary tensor kernels with any type of symmetry.

Thus, my specific contributions in this thesis are

1. An identification and categorization of strategies to exploit symmetry in tensor kernels. These include saving memory bandwidth by reusing memory reads of the symmetric input and compute bandwidth by filtering redundant computations. To the best of my knowledge, this thesis is the first to systematically depict how to take advantage of symmetry at a granular level, providing context for some of the broader optimizations previously described in the field.

2. A set of compiler techniques and automatic code transformations to systematically generate and optimize symmetric code using the aforementioned strategies to exploit symmetry. The first phase, symmetrization, involves generating code to read only the canonical triangle(s) of symmetric input tensors, while the second phase, optimization, focuses on reducing the number of memory accesses and operations.

3. An implementation of the compiler and the experimental results of the implementation on several common tensor kernels. We demonstrate speedups ranging from 1.36x for SSYMV to 7.95x for a 4-dimensional MTTKRP with the symmetric code generated by the compiler over the naive implementation of these kernels.

1.1 Thesis Overview

The remainder of this thesis is organized as follows

- **Chapter 2: Background** discusses existing technologies to take advantage of symmetry and where they could be extended or improved.

- **Chapter 3: Terminology for Discussing Symmetry** presents a set of terminology to use when discussing symmetry.

- **Chapter 4: Techniques to Exploit Symmetry** describes the core strategies that form the groundwork for optimizing symmetric kernels.
- **Chapter 5: Symmetric Compiler** provides a step-by-step generalized methodology to symmetrizing a kernel and then optimizing it that applies to any kernel.
- **Chapter 6: Evaluation** compares the performance of kernels to the symmetric version generated by the compiler in Finch.
- **Chapter 7: Conclusion** concludes with some future directions to extend the applicability of the compiler to more types of kernels and integrate it with existing libraries.

Chapter 2

Background

2.1 Related Work

There are several existing solutions to optimize computation with symmetric tensors. The most common application of these solutions has been for the Coupled Cluster (CC) method, a numerical technique widely used for describing quantum many-body systems which involves hundreds to thousands of tensor contractions. Tensors in CC have a high-dimensional structure with permutational symmetry or skew-symmetry, which arises from the requirement that the wave-function for fermions (bosons) be antisymmetric (symmetric) under the interchange of particles [40].

Given the steep computational cost of quantum chemistry methods, a lot of effort has been put into designing algorithms that enable efficient use of computational resources. Overall, CC motivates tensor contraction algorithms that exploit symmetry in tensors, efficiently support contractions among tensors of diverse dimensions and shapes, and are suitable for long and repeated contraction sequences [40]. The goal remains to minimize communication (i.e. the number of words of data moved across network by any given processor) done to contract tensors.

There are a few generalized existing systems focused on minimizing operation count in kernels with symmetric tensors. For example, the OpMin system provides an operation optimization process that identifies the best sequence of two-tensor contractions to achieve a multi-tensor contraction, performs common sub-expression elimination to reduce computational costs, and factorizes via the distributive law across multiple tensors [18]. Lai et. al present a comparison of the opcounts that the restructured tensor contraction OpMin generates would hypothetically involve with the actual opcounts (determined via performance counters inserted into the soure code) that high performance quantum chemistry computation suites NWChem [43] and PSI3 [37] use to compute that same contraction. However, the OpMin system does not actually provide a means to generate the code itself.

Blocking, a common technique in tensor operations, involves partitioning large tensors into smaller blocks to facilitate efficient computation. By breaking down tensors into manageable chunks, blocking reduces memory overhead and improves cache utilization, enhancing overall computational performance. Additionally, blocking enables parallelization of tensor operations by distributing blocks across multiple processors or computing nodes. This approach is widely employed when dealing with symmetric tensors. Schatz et. al propose Blocked Compact Symmetric Storage which involves storing only the unique blocks of symmetric tensors and a format for an algorithm-by-blocks for several common tensor kernels that exploits this storage format [32]. In this algorithm-by-blocks, knowledge of the symmetry of the operands is concealed from the implementation; all computation is done as if the blocks are nonsymmetric. Whilst most of the blocks are indeed nonsymmetric, the blocks that intersect with a diagonal (i.e. the boundary between the canonical triangle[1] and the other triangles) must be symmetric. Redundant computation *could* be avoided if a symmetry-aware program was used for such blocks.

The Cyclops Tensor Framework (CTF), a popular library for distributed tensor computations, utilizes a different scheme for blocking wherein values in a tensor are distributed cyclicly to processors so that each processor owns every element of the tensor with a defined cyclic phase. This algorithm ensures that each the sub-tensor owned by any processor has the same symmetry and structure as the whole tensor, thus remaining susceptible to classical linear algebra optimizations [39]. Furthermore, the mapping scheme is defined such that the distributed contraction algorithm can treat the blocks as nonsymmetric. Here too, we speculate that it could be possible to use fewer blocks and less padding, and thus more efficient computation, if a symmetry-aware program was utilized instead.

However, while these systems describe how to most optimally restructure tensor operations via either reordering tensor contractions or dividing the computation into blocks to reduce operation count and maximize throughput, respectively, they do not in fact describe how to actually *compute* the tensor operations described (in OpMin's case) or if they do, it is via a program that is *not* symmetry aware. Many existing libraries or papers that do provide symmetry-aware algorithms are highly specialized to specific kernels. The LAPACK (Linear Algebra Package) BLAS (Basic Linear Algebra Subprograms) provide a set of low-level routines that are optimized for efficiency and are widely used as building blocks in higher-level numerical libraries and applications. There are routines for vector, matrix-vector, and matrix-matrix vector operations, including specialized routines for symmetric matrices [19]. Solomonik also proposes a specialized method to minimize operation count with the symv, syr2, syr2k, and symm kernels by taking advantage of multiplication on a ring by reformulating the kernels with a symmetric intermediate and introducing an additive inverse [38]. Furthermore, Cai et. al propose a new technique for computing the Matricized Khatri-Rao Product (MTTKRP) kernel that involves fusing the computations of the Khatri-Rao product and the multiplication of it with the matricized tensor, instead of completing one after the other [4].

While there are several specialized algorithms for specific kernels to exploit symmetry that have been proposed, there is overall a lack of literature describing how to programatically generate these algorithms for arbitrary tensor kernels. A generalized system that can be applied to any type of symmetric tensor used in any application and result in significant performance savings does not yet exist.

The closest such system that exists was proposed by Shi et. al. Shi proposes a system of representing and accessing packed symmetric tensors in storage using simplical numbers and details how to generate an *output-oriented* loop structure for any tensor kernel that

[1]Defined in Section 3.

iterates through all coordinates needed to sequentially compute the output [34]. However, the output-oriented approach randomly accesses the symmetric input, thus forgoing many possible memory savings and resulting in poor performance.

I seek to fill the gap in literature surrounding symmetric tensors by presenting a granular approach to identify symmetry in a tensor kernel and generate code that successfully exploits this symmetry. In this thesis, we will provide a generalizable framework to do the aforesaid that can be applicable to any tensor kernel with any type of symmetry by analyzing the phenomena with symmetry that emerges in different kernels and the means in which it can be capitalized.

2.2 Common Symmetric Tensor Kernels

Below is a set of tensor kernels pervasive across various applications and domains that we will be using to demonstrate many of the techniques described later in the paper. In these kernels, we make the distinction of tensors being dense/sparse to be orthogonal to normal (i.e. not symmetric)/symmetric—we find that it is usually the sparse tensor that is symmetric in applications of these kernels.

2.2.1 SSYMV

Sparse symmetric matrix-vector multiplication (SSYMV) is given by

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

where matrix A is sparse symmetric, and vectors y and x are dense.

SSYMV is often used within iterative solvers, such as the conjugate gradient method, which is used to solve systems of linear equations, because the coefficient matrix is symmetric and positive-definite [25]. Furthermore, in graph theory, many problems are represented as operations on symmetric matrices (like adjacency matrices of undirected graphs), where SSYMV can be used for finding properties such as centrality measures [42]. In structural engineering, finite element analysis often leads to large, sparse, symmetric system matrices because of the inherent symmetries present in many physical structures (e.g from the conservation of energy and equilibrium), causing SSYMV operations to be a common occurrence [46]. There are also applications in machine learning. Principal Component Analysis (PCA), a dimensionality reduction technique used for analyzing high-dimensional data, involves computing covariance matrices which are inherently symmetric [8]. In Support Vector Machines (SVMs), the Gram matrix is symmetric because it represents the pairwise inner products of feature vectors [26].

2.2.2 SYPRD

The symmetric triple product (SYPRD), also known as the "quadratic form" in literature [27] is given by

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

where matrix A is sparse symmetric, vector x is dense, and y is a scalar.

SYPRD is very common in optimization problems, especially when the objective function is a quadratic form [27]. In statistics, symmetric quadratic forms are used in the analysis of variance (ANOVA) to determine the variability within and between groups of data. They are also crucial in estimating the parameters of a linear regression model, where the matrix often represents covariances or correlations [3]. In graph theory, because the adjacency matrix for an undirected graph is symmetric, graphs and kinematic chains can be represented by a SYPRD kernel to detect graph isomorphism [13].

2.2.3 SSYMM

Sparse symmetric matrix-matrix multiplication (SSYMM) is given by

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

where matrix A is sparse symmetric and matrices B and C are dense.

SSYMM is also used in finite element analysis for simplex meshes to update stiffness matrices during the simulation of structures under various load conditions [24]. In spectral clustering and graph partitioning, the Laplacian matrix, is symmetric and sparse. SSYMM is involved in operations that combine various graph Laplacians or when adjusting weights in graph structures [30]. Quantum mechanics and computational chemistry often deal with Hamiltonian and overlap matrices that are symmetric and sparse. SSYMM operations are essential in calculating the properties of quantum systems over time or under different conditions [9]. In portfolio optimization and other financial computations, covariance matrices, which are symmetric, are multiplied to adjust risk models or to scale financial properties across different scenarios [29]. Finally, in 3D reconstruction and image processing, SSYMM can be part of the computation for transforming matrices when aligning images or constructing spatial structures from image data [47].

2.2.4 SSYRK

Sparse symmetric rank-K update is given by

$$C[i,j] = A[i,k] * A[j,k]$$

where A is a sparse (not necessarily symmetric) matrix and C is a dense symmetric matrix because AA^T for any matrix A is symmetric.

SSYRK is employed in optimization algorithms such as the Broyden–Fletcher–Goldfarb–Shanno (BFGS) and its limited memory variant (L-BFGS), which are used for nonlinear optimization problems. The update step in these algorithms, which refines the approximation of the Hessian matrix, can be formulated as a rank-k update [14]. SSYRK is also useful in control theory for updating Lyapunov or Riccati equations that describe the evolution of system errors or state estimations in control systems [44]. In 3D modeling and rendering, updating transformation matrices to reflect changes in orientation or position is accomplished using SSYRK [47]. SSYRK also plays a significant role in updating covariance matrices in Cholesky-based matrix inversion algorithms [17].

2.2.5 TTM

The mode-1, mode-2, and mode-3 variations of the tensor times matrix kernel [16] are given by

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

where A is a sparse fully symmetric tensor, B is a dense matrix, and C is a dense tensor.

TTM operations are commonplace in tensor decomposition methods such as CANDECOMP/PARAFAC (CP) and Tucker decompositions [16]. In these decompositions, fully symmetric tensors often arise when modeling inherently symmetric phenomena, such as diffusion processes [22] or signal correlations [7]. In multilinear subspace learning, symmetric tensors can capture the higher-order correlations in data [23]. TTM operations are also relevant in algorithms used for feature extraction and dimensionality reduction [31]. In quantum chemistry, the electron correlation problem involves symmetric tensors representing the interactions between electrons and in physics, tensors are used to represent various properties such as inertia, the stress-energy of a system, or the moments of a distribution; TTM is used to then apply transformations to these kernels [6].

2.2.6 MTTKRP

Matricized tensor times Kronecker product is given by

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

where A is a sparse fully symmetric tensor and B, C, and D are dense matrices. We consider the specialized case where B and C are equivalent as this causes more symmetric properties to emerge.

This specialized case of MTTKRP emerges in CP decomposition when the same factor matrix is used for all modes and is a common bottleneck [21]. A common application of CP decomposition is to speed-up convolutions within CNNs [20]. CP decomposition with structured matrices is also used in signal processing application such as Wiener-Hammerstein system identification and cumulant-based wireless communication channel estimation [11]. Furthermore, in quantum mechanics, fully symmetric tensors often represent interaction tensors or density matrices, where the matrices being equivalent can reflect operations involving particles of the same type or symmetry in interaction terms [12]. In the study of materials, especially when analyzing isotropic materials or symmetric properties like thermal conductivity or elasticity, the symmetric tensor and equivalent matrices can model properties that are uniform in all directions, allowing for simplified and more efficient computational modeling [35].

2.2.7 Other Tensor Kernels

The aforementioned kernels were chosen because they each demonstrate different symmetric properties and serve as effective examples to illustrate the methodology we describe in later sections to exploit symmetry. Our methodology can be generalized to apply to tensor contractions with a variable number of tensors of any dimensionality and any form of symmetry. These example kernels also only involve the multiplication operator, but our methodology can be expanded to work with any commutative operator used within the kernel.

2.3 Finch

Throughout this thesis, 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 multi-dimensional arrays [1]. Finch supports a variety of tensor storage formats and provides a clean interface for writing dense loops with enhanced control structures to read and update tensors. We use Finch because of the support for a range of sparse storage formats and enhanced control flow, which is necessary to implement and execute the symmetric kernels our compiler generates. Finch also handles the complexities of sparse data manipulation, enabling us to write readable, dense loop structures, which makes it easier to focus the optimizations we apply to take advantage of symmetry.

Finch uses a fiber-tree style tensor abstraction [1] depicted in Figure 2.1 where a multi-dimensional tensor is represented as a nested vector datastructure, where each level of the nesting corresponds to a dimension of the tensor.

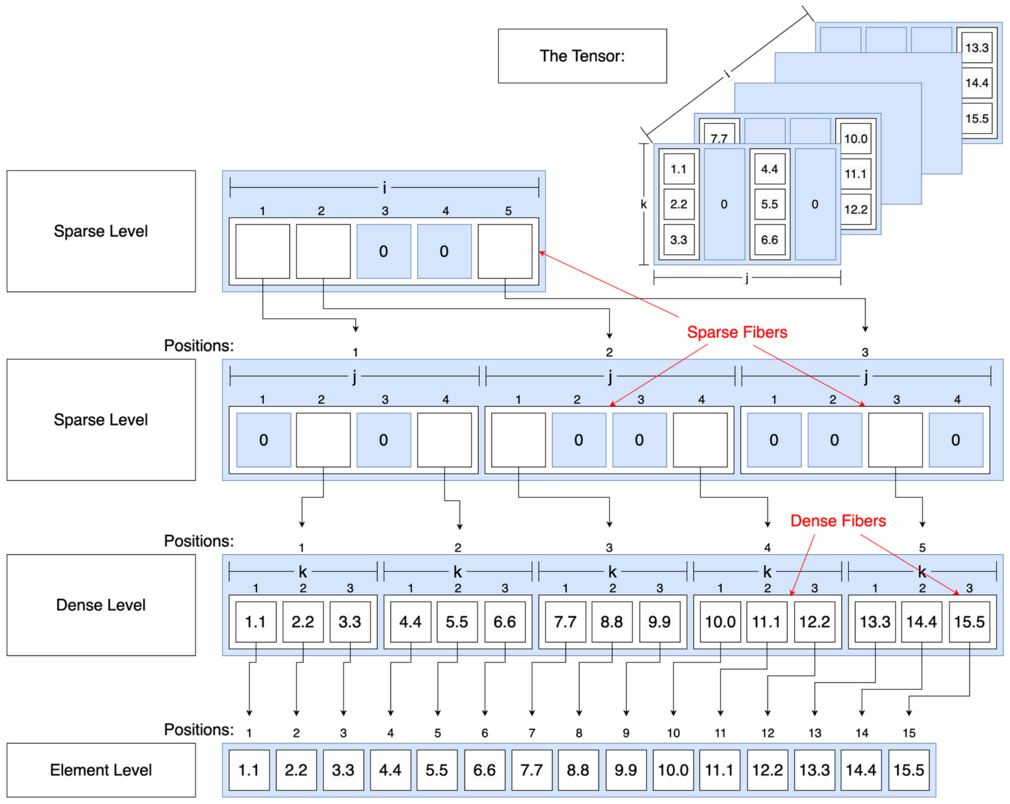


Figure 2.1: The fiber tree representation of a tensor with two sparse outer levels and a dense inner level in Finch.)

The Finch syntax mirrors most imperative languages with for-loops and control flow, as shown in Figure 2.2 [1].

```
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 2.2: The syntax of the Finch language [1], which will be used throughout the rest of thesis when describing kernels.)

To fully illustrate the utility of using Finch to write dense code, but have it compile as sparse code, let us take a look at Listings 2.1 and 2.2 which demonstrate the code that we would write with Finch and the code that Finch actually executes, respectively. The kernel in Listing 2.1 works with sparse data, but the code itself appears dense because Finch handles the sparse data manipulation for us, which we can see in the generated code in Listing 2.2. Moreover, Finch is also able to lift conditions to the highest possible loop level.

```
A = Tensor(Dense(SparseList(Element(0.0)), rand(10, 10))
s = Scalar(0.0)

@finch begin
    s .= 0
    for j=_, i=_
        if i <= j
            s[] += A[i, j]
        end
    end
end
```

Listing 2.1: Program to compute the sum of the values in the upper triangle of a sparse matrix, written in Finch.

```
s = ((ex.bodies[1]).bodies[1]).tns.bind
A_lvl = ((ex.bodies[1]).bodies[2]).body.body.body.rhs.tns.bind.lvl
```

```
A_lvl_2 = A_lvl.lvl
A_lvl_ptr = A_lvl_2.ptr
A_lvl_idx = A_lvl_2.idx
A_lvl_2_val = A_lvl_2.lvl.val
result = nothing
s_val = 0
for j_4 = 1:A_lvl.shape
    A_lvl_q = (1 - 1) * A_lvl.shape + j_4
    A_lvl_2_q = A_lvl_ptr[A_lvl_q]
    A_lvl_2_q_stop = A_lvl_ptr[A_lvl_q + 1]
    if A_lvl_2_q < A_lvl_2_q_stop
        A_lvl_2_i1 = A_lvl_idx[A_lvl_2_q_stop - 1]
    else
        A_lvl_2_i1 = 0
    end
    phase_stop = min(j_4, A_lvl_2.shape, A_lvl_2_i1)
    if phase_stop >= 1
        if A_lvl_idx[A_lvl_2_q] < 1
            A_lvl_2_q = Finch.scansearch(A_lvl_idx, 1, A_lvl_2_q, A_lvl_2_q_stop - 1)
        end
        while true
            A_lvl_2_i = A_lvl_idx[A_lvl_2_q]
            if A_lvl_2_i < phase_stop
                A_lvl_3_val = A_lvl_2_val[A_lvl_2_q]
                s_val = A_lvl_3_val + s_val
                A_lvl_2_q += 1
            else
                phase_stop_3 = min(A_lvl_2_i, phase_stop)
                if A_lvl_2_i == phase_stop_3
                    A_lvl_3_val = A_lvl_2_val[A_lvl_2_q]
                    s_val += A_lvl_3_val
                    A_lvl_2_q += 1
                end
                break
            end
        end
    end
end
s.val = s_val
result = (s = s,)
result
```

Listing 2.2: The code that Finch actually generates and runs to compute the sum of the values in the upper triangle of a sparse matrix.

Chapter 3

Terminology for Discussing Symmetry

In this section, we introduce the terminology that will be used throughout the rest of the thesis. We adopt and adapt several definitions as well as the notation used to describe them from existing literature to maintain consistency in communication.

3.1 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 [5].

Definition 3.1.1 (Symmetry) Let T be an n-dimensional tensor and let S_n be the set of all permutations of $\{1, ..., n\}$. Then T is *symmetric* if for all $\sigma \in S_n$,

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

Conventionally, symmetric is synonymous to *fully symmetric*.

Example 3.1.1 (Symmetry) A three-dimensional tensor T is *symmetric* if

$$T[i_1, i_2, i_3] = T[i_1, i_3, i_2] = T[i_2, i_1, i_3] = T[i_2, i_3, i_1] = T[i_3, i_1, i_2] = T[i_3, i_2, i_1].$$

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 [32]. We denote π_i to be the i^{th} part of Π.

We define partial symmetry relative to a chosen partition [34].

Definition 3.1.2 (Partial Symmetry) Let T be an n-dimensional tensor, and let Π be a partition of $\{1, ..., n\}$. Then T is *partially symmetric* if for each part $\pi_i \in \Pi$ and each permutation σ_i of the elements in π_i,

$$T[i_1, \ldots, i_n] = T[i'_1, \ldots, i'_n]$$

where

$$i'_j = \begin{cases} i_{\sigma_1(j)} & \text{if } j \in \pi_1 \\ \vdots & \vdots \\ i_{\sigma_m(j)} & \text{if } j \in \pi_m. \end{cases}$$

Then, we can denote that T has Π symmetry.

Note that parts of size one correspond to indices that are not transposable, as they may not be interchanged with any other index. Also note that the term *partial symmetric* encapsulates the term *fully symmetric* because a partition of A can consist of a single set with all the indices in A. Furthermore, the term *partial symmetry* technically also encapsulates an unsymmetric tensor because we can have a partition where $|\pi_i| = 1$ for all i; however, to distinguish unsymmetric tensors, although we will use the S_T notation for all tensors T to describe their symmetry groups (irregardless of whether they actually demonstrate the presence of symmetry), we will only use the term *partial symmetry* to describe tensors where there exists $|\pi_i| > 1$.

Example 3.1.2 (Partial Symmetry) A three-dimensional tensor T has $\{\{1, 3\}, \{2\}\}$ partial symmetry if

$$T[i_1, i_2, i_3] = T[i_3, i_2, i_1].$$

Although conventionally, *symmetric* is synonymous to *fully symmetric*, throughout this thesis, for simplicity, we will also use the term *symmetric* to describe tensors that exhibit partial symmetry. Additionally, we will use set S_T to interchangeably represent the symmetry of a fully or partially symmetric tensor T and a 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, S_T is $S_{\pi_1} \times ... \times S_{\pi_m}$ where $\pi_i \in \Pi$ and Π is a partition of $\{1, ..., n\}$.

A tensor T with symmetry S_T has many groups of coordinates with equivalent values. To standardize tensor operations, we can choose to work with a specific coordinate from each group of equivalent coordinates; let us refer to the coordinate that is commonly used or referenced in tensor-related computations, operations, or representations as the *canonical* coordinate. We define it as follows.

Definition 3.1.3 (Canonical) Let tensor $T[i_1, ..., i_n]$ have symmetry S_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 S_T. Otherwise, the coordinates are *non-canonical.*

Furthermore, let us define the region of a tensor consisting of canonical coordinates.

Definition 3.1.4 (Canonical Triangle) The *canonical triangle* of a tensor consists of all the canonical coordinates in the tensor.

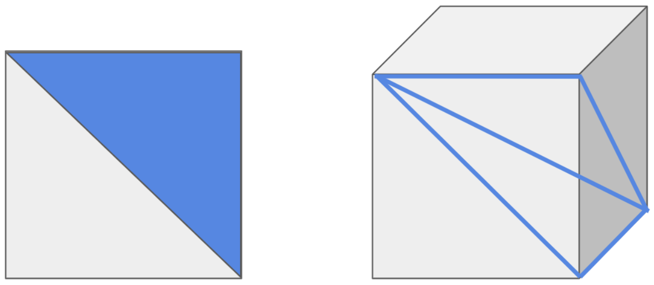


Figure 3.1: The canonical triangles of a matrix and 3-dimensional tensor if the zero coordinate is in the (front) top left corner of a tensor. In this case, the canonical triangle of the matrix is equivalent to the upper triangle.

The canonical triangle of a tensor forms a geometric triangle in two-dimensions, a prism in three-dimensions (Figure 3.1), and a polytope in higher dimensions. We term the edges and faces of these geometric figures the diagonals.

Definition 3.1.5 (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.

Note that this is different from the conventional definition of a diagonal, which involves only the coordinates where all indices $i_1 = i_2 = ... = i_n$. Our definition enables there to be multiple diagonals in a tensor since a diagonal is defined by the subset of indices that are equivalent and there are multiple subsets of indices of size greater than one for a tensor of dimension three and greater.

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 3.1.6 (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 can simplify the representation of an equivalent group to only include parts where there are more than one indices. Moreover, note that since the order of the subsets in E or the order of the indices in E do not matter, permutations of the subsets of E or the indices in subsets of E still result in the same equivalence group.

A reminder about the distinction between symmetry groups and equivalence groups: symmetry groups indicate which permutation of indices produce coordinates at which there are equivalent entries in a tensor, while equivalence groups indicate which indices used to access a tensor are constrained to be equivalent in value. To make the distinction clearer, we will use $=$ in equivalence groups to indicate the equivalent indices.

Example 3.1.3 (Equivalence Group) Given equivalence group $\{(i = j)\}$, indices i and j of $T[i, j, k]$ are equivalent. All the entries of T that satisfy this equivalence group comprise the diagonal of T where $i = j$.

Example 3.1.4 (Equivalent Symmetry Permutations) Suppose we access fully symmetric tensor $T[i,j,k]$ at equivalence group $\{(i=j)\}$. The symmetry of T is described by

$$S_T = \{\{1,2,3\},\{1,3,2\},\{2,1,3\},\{2,3,1\},\{3,1,2\},\{3,2,1\}\}.$$

However, because we are accessing entries on the diagonal where $i=j$ (i.e. indices 1 and 2 are the same), the permutations $\{1,2,3\}$ and $\{2,1,3\}$, $\{1,3,2\}$ and $\{2,3,1\}$, and $\{3,1,2\}$ and $\{3,2,1\}$ result in accesses of the same entries.

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 3.1.7 (Unique Symmetry Group) Let $S_T|E$ represent the *unique symmetry group*, which given an equivalence group E, consists of

$$S_T|E = \{\pi \in S_n \mid \forall i,j \in \{1,2,...,n\},\ \text{if } i,j \text{ are both in the same subset of } E,\ \text{then } \pi(i) < \pi(j)\}.$$

3.2 Tensor Operations

We will now shift to discussing terminology to describe tensor operations. We consider the basic unit for computation with tensors to be an *assignment*.

Definition 3.2.1 (Assignment) A tensor *assignment* is an operation of the form

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

where tensor O is the output tensor, tensors $T_1,...,T_m$ are the input tensors, and $\otimes$ is some binary operation (e.g. $+$, $\times$, etc.). An assignment consists of computing a value using specific entries in the input tensor and assigning it to a a specific coordinate of the output tensor.

If the binary operator $\otimes$ is multiplication, an assignment is equivalent to the expression $i_{1,1}\dots i_{1,n_1},\quad \dots,\quad i_{m,1}\dots i_{m,n_m} \to i_1 \dots i_n$ in the Einstein summation convention (einsum) [41] where the reduction is across the indices that do not present themselves in the output. We choose to use the assignment notation we have introduced, however, as it closely parallels the syntax to write assignment expressions in the Finch language (Section 2.3).

Example 3.2.1 (Assignments) The assignment that forms the basis of the MTTKRP kernel is

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

Chapter 4

Techniques to Exploit Symmetry

4.1 Types of Symmetry

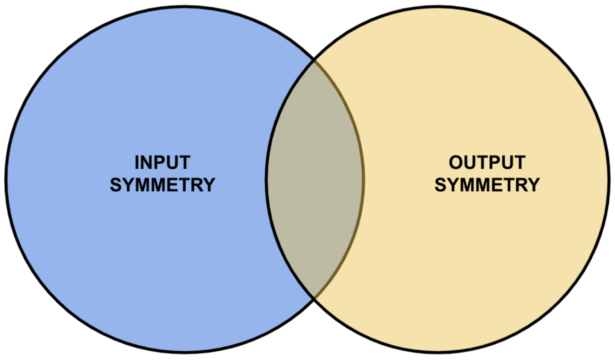


Figure 4.1: Assignments can have input and/or output symmetry.

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 (Figure 4.1). We make the distinction because the techniques to exploit symmetry vary based on the type of symmetry.

It is possible to have input symmetry, but not output symmetry as the assignment itself may not preserve the symmetry present in the input tensors: the computation involved in the assignment may alter or distribute the symmetry differently across the output indices. On the other hand, an assignment may result in a symmetric output tensor even if the input tensors do not possess similar symmetry due to the specific combination of indices involved in the assignment and the resulting tensor contraction: the symmetry in this case arises from the mathematical properties of the assignment itself, rather than inherent symmetry present in the input tensors.

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 4.1.1 (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 B is indexable by both i and j, 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 B is not indexable by both j and k, we refer to this symmetry as *invisible.*

We may also have output symmetry between indices that are and are not present in the output tensor (i.e. "visible-invisible" output symmetry), but we restrict our optimizations to just exploit symmetry in groups of indices that are all present in the output tensor and groups of indices that are all not present in the output tensor. The replication strategies, as described in the subsequent section, are different based on the type of symmetry, and for simplicity, we choose to not account for the intermediate category.

4.2 Core Strategies to Exploit Symmetry

We take an input-oriented approach to exploit symmetry in which we iterate through only the coordinates needed to sequentially access the symmetric input whilst randomly accessing the output and nonsymmetric inputs. This is in contrast to the output-oriented approach explored by Shi et. al which involves iterating through all the coordinates needed to sequentially compute the output [34].

Logically, in the presence of *input symmetry*, we choose to iterate over and access only the canonical triangle(s) of the symmetric tensor(s). Analogously, in the presence of *output symmetry*, we choose to update only the canonical triangle of the output tensor. Furthermore, depending on whether the output symmetry is *visible* or *invisible*, the post-processing we do to replicate the output tensor (i.e. fill in the redundant values) will differ. A mechanical approach to exploit all types of symmetry will be described in Chapter 5, but in this section, we will highlight the core strategies underlying this approach.

The two core strategies we have identified to exploit symmetry to make better use of memory and compute bandwidth are

1. reusing memory reads to save on bandwidth and

2. filtering redundant computations,

which are dissected in more detail in the following subsections.

4.2.1 Reusing Memory Reads

Given symmetric input tensors, we can restrict reads to the canonical triangle and use the same memory read to perform multiple computations for the output. The key motivation for reusing memory reads is to save memory bandwidth, which is particularly relevant for memory bandwidth bound kernels (e.g. SSYMV) that have a high ratio of memory operations to floating-point operations. 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 bandwidth).

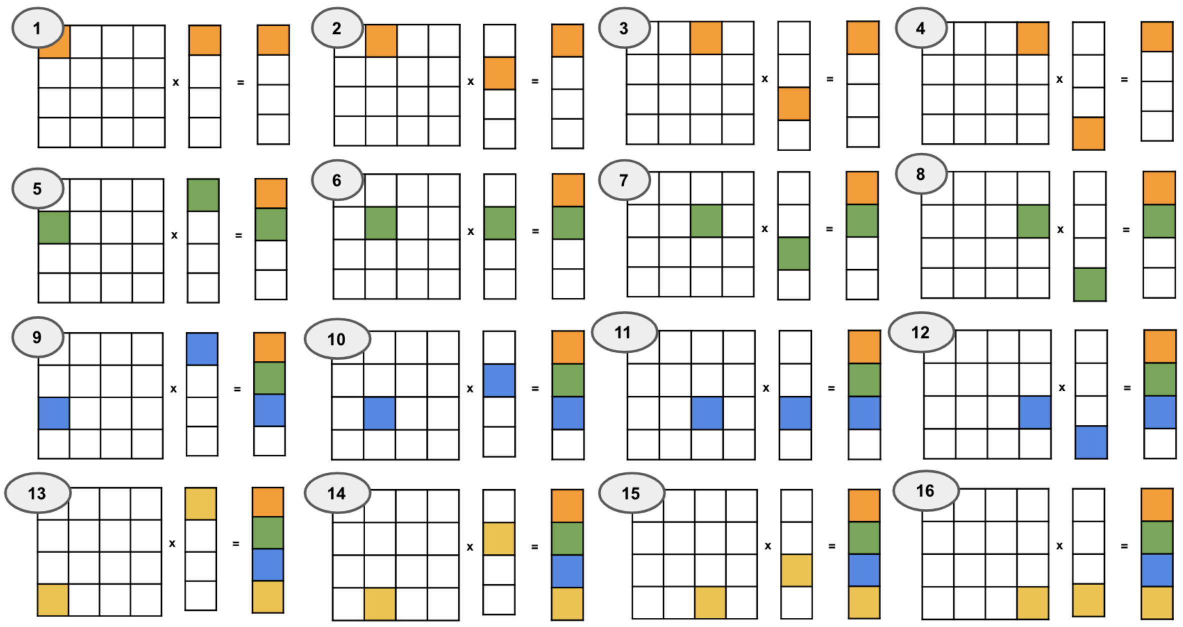


Figure 4.2: The naive SSYMV kernel iterates through all the elements of the matrix.

Let us take a look at what reusing memory reads algorithmically entails for the SSYMV kernel given by $y[i] = A[i, j] * x[j]$. In Figure 4.3, essentially, for every access of a coordinate in the canonical triangle, we perform all the assignments that the canonical coordinate and its non-canonical equivalents would be involved in. Note that the values on the diagonal get handled differently—i.e. they are involved in only assignment, whereas all other values are involved in two assignments.

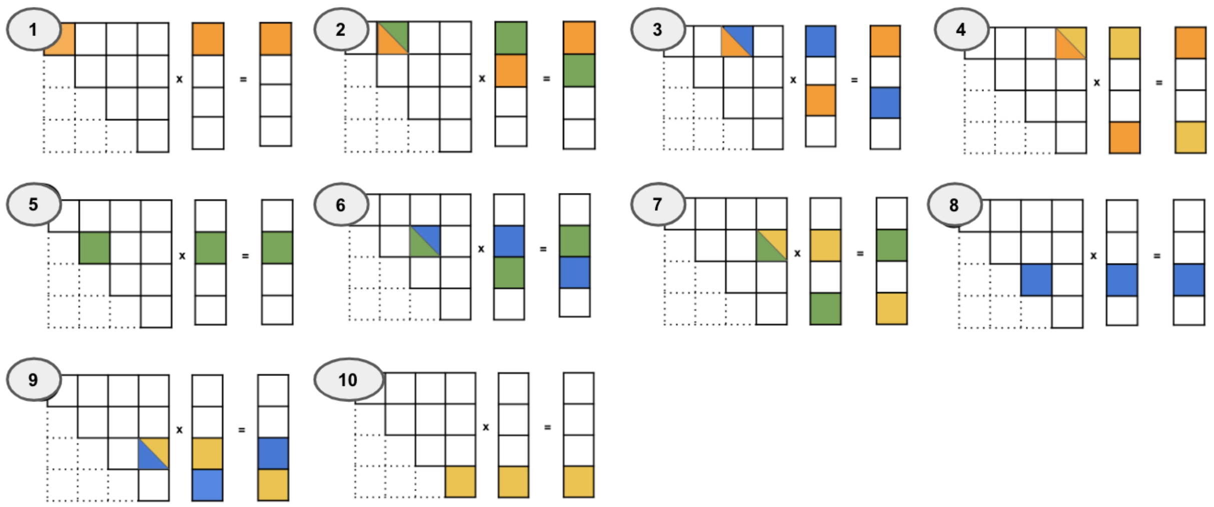


Figure 4.3: The SSYMV kernel optimized to reuse memory reads iterates through only the canonical triangle of the matrix.

The code segments in Listings 4.1 and 4.2 illustrate the implementations for the diagrams in Figures 4.2 in 4.3. Listing 4.2 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 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 $i > j$.

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

Listing 4.1: Naive SSYMV

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

Listing 4.2: SSYMV that accesses only canonical triangle *and* reuses memory reads.

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 [16] 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 Figure 4.3.

```
for l=_, i=_, k=_, j=_
    if j <= k && k <= l
        if j < k && k < l
            A = A[j, k, l]
            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 4.3: TTM kernel that accesses only the canonical triangle of A.

The monotonically increasing condition on line 2 of Listing 4.3 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 4.3, 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. Specifically, the block from lines 3-10 handles the equivalence group $\{(j), (k), (l)\}$, lines 11-15 handles $\{(j = k)\}$, lines 16-20 handles $\{(k = l)\}$, and lines 21-22 handle $\{(j = k = l)\}$.

We can generalize the examples for SpMV and TTM to any kernel. To restrict reads to a particular triangle, we can enclose the loop body of the kernel in an if-clause that checks that the indices corresponding to the axes of symmetry are in the canonical triangle (i.e. given our definition of the canonical triangle, that some order of these indices are monotonically increasing).

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 the exact assignments to the output that need to be performed for each.

Evidently, as the number of dimensions increase, the complexity of the symmetrized kernel also increases. It becomes impractical to hand-write the kernel, pointing to the need to utilize an automated system. Chapter 5 describes a systematic method to determine which assignments need to be performed and when to symmetrize a kernel.

Note that whether or not there is a speedup from this transform alone depends on the nature of the kernel—we reduce the memory bandwidth needed to read the symmetric input tensors, but we are performing out-of-order updates to the output. Regardless of whether there is a performance enhancement, rewriting in this format enables patterns across multiple assignments to be identified and capitalized, as discussed in the subsequent section.

4.2.2 Filtering Redundant Computations

The symmetrization process results in multiple assignments being performed with one read of the symmetric tensors. Having multiple assignments in each loop iteration (as opposed to just one assignment per loop iteration) makes it easier to identity and exploit patterns that emerge across assignments. Namely, it enables us to identify where symmetry emerges in the output tensor and consequently, take advantage of it.

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 4.3 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 4.4. 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 4.5, 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 $\frac{1}{2}$. 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 4.5), thus completing the remaining assignments.

```
for l=_, j=_, k=_, i=_
    if j <= k && k <= l
        A = A[j, k, l]
        C[i, j, l] += A * B[k, i]
        C[i, l, j] += A * B[k, i]
        C[i, j, k] += A * B[l, i]
        C[i, k, j] += A * B[l, i]
        C[i, k, l] += A * B[j, i]
        C[i, l, k] += A * B[j, i]
```

Listing 4.4: Before exploiting output symmetry in the conditional block of the TTM kernel that handles non-diagonal coordinates of A.

```
for l=_, j=_, k=_, i=_
    if j <= k && k <= l
        A = A[j, k, l]
        C[i, j, l] += A * B[k, i]
        C[i, j, k] += A * B[l, i]
        C[i, k, l] += A * B[j, i]
for l=_, j=_, i=_
    if j > l
        C[i, j, l] = C[i, l, j]
```

Listing 4.5: After exploiting output symmetry in the conditional block of the TTM kernel that handles non-diagonal coordinates of A.

Let us now see how we can exploit output symmetry in the conditional blocks that handle the diagonal coordinates of A. We will look at the second conditional block of Listing 4.3 which handles the diagonal represented by equivalence group $\{(j = k)\}$, replicated below in Listing 4.6. It is slightly less apparent that we even have output symmetry because every assignment does not have an analogous assignment where the second and third indices are swapped, like we did in Listings 4.4-4.10. However, given that `j == k`, we can swap the `j` and `k` indices in the assignment `C[i, l, k] += A * B[j, i]` on line 7 to obtain `C[i, l, j] += A * B[k, i]`. It is evident now that there is output symmetry across the second and third indices of C in lines 5 and 7. We also observe that the second and third indices of `C`, `j` and `k`, on line 6 are equivalent, and thus there is no analogous assignment.

```
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, l, k] += A * B[j, i]
```

Listing 4.6: TTM kernel that handles the coordinates on the diagonal represented by $\{(j = k)\}$ in A

Given these observations, we can write the second block as shown in Listing 4.7, with only two assignments and replication of C afterwards. Here we aim to show that it may not be immediately clear in a mathematically correct symmetrized kernel where the output symmetry is; we may need to swap around a few indices in the blocks accounting for the diagonals to make the visible output symmetry more apparent.

```
for l=_, i=_, k=_, j=_
    if j <= k && k <= l
        if j == k && k != l
            A = A[j, k, l]
            C[i, j, l] += A * B[k, i]
            C[i, j, k] += A * B[l, i]
for l=_, j=_, i=_
    if j > l
        C[i, j, l] = C[i, l, j]
```

Listing 4.7: TTM kernel that handles the coordinates on the diagonal represented by $\{(j = k)\}$ in A after replicating output symmetry and with post-processing replication of C

The resulting TTM kernel after exploiting the visible output symmetry in C thoughout the entire kernel is given by Listing 4.8.

```
for l=_, i=_, k=_, j=_
    if j <= k && k <= l
        if j < k && k < l
            A = A[j, k, l]
            C[i, j, l] += A * B[k, i]
            C[i, j, k] += A * B[l, i]
            C[i, l, k] += A * B[j, i]
        if j == k && k != l
            A = A[j, k, l]
            C[i, j, l] += A * B[k, i]
            C[i, j, k] += A * B[l, i]
        if j != k && k == l
            A = A[j, k, l]
            C[i, j, l] += A * B[k, i]
            C[i, k, l] += A * B[j, i]
        if j == k && k == l
            C[i, j, l] += A[j, k, l] * B[k, i]
for l=_, j=_, i=_
    if j > l
        C[i, j, l] = C[i, l, j]
```

Listing 4.8: Entire TTM kernel after exploiting visible output symmetry.

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 assignment operations by $\frac{1}{n!}$.

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 filter 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 nondiagonal assignments in Figure 4.9 (lines 5-6), we can optimize by only performing one assignment but multiplying it by a factor of 2, as depicted in figure 4.9 (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 4.9: 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]
6
7
8
```

Listing 4.10: 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 assignment operations by $\frac{1}{n!}$.

After symmetrizing to reuse memory reads and filtering redundant computation, there are further optimizations that do not necessarily exploit symmetry, but instead exploit some of the properties that materialize as a result of the optimizations *to* exploit symmetry described previously. These additional optimizations result in fewer memory accesses, improved cache locality, and better branch prediction and will be described in the subsequent chapter.

Chapter 5

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 chapter, 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.

5.1 Methodology

We divide this system in two phases to reflect the two core strategies of first capitalizing on memory bandwidth and then compute bandwidth. 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.

5.1.1 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.

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.

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
$$P = \bigcup_{i=1}^{m} \Big(\bigcup \{\pi \in \Pi_i \,|\, |\pi| > 2\} \Big)$$
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 programatically 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 5.1. 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 $i_1 = 1 : \_, i_2 = 1 : \_, ...$ do                          ▷ Loop through all indices
2:     if $p_1 \leq ... \leq p_n$ then
3:         for each $E$ of $P$ do          ▷ Iterate through all possible equivalence groups of P
4:             Construct $S_P|E$                             ▷ Determine all unique permutations
5:             for each $\sigma \in S_P|E$ do
6:                 $(O[i_1, ..., i_n] = T_1[i^1_1, ..., i^1_n] \otimes ... \otimes T_m[i^m_1, ..., i^m_n])\ [i \to \sigma(i)]$
7:             end for
8:         end for
9:     end if
10: end for
```

Figure 5.1: Pseudocode for Symmetrized Kernel

We can furthermore unroll the loops from lines 5-7 and lines 3-8 in Figure 5.1 to generate a more efficient kernel. We demonstrate how to do this for MTTKRP in Section 5.2. 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.

5.1.2 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 and are illustrated in more detail in Table 5.1.

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 $\frac{1}{n}$. The same entry of other tensor operands may also be read multiple times and can be replaced with a constant.

Distributive Assignment Grouping

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

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 is replicated to the noncanonical triangles. Note that equivalent index variables may need to be swapped with each another in the assignments in a conditional block to make it easier to identify the output symmetry.

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. Note that equivalent index variables may need to be swapped with each another in the assignments in a conditional block to make it easier to identify whether or not the assignments are equivalent to those in another conditional block. This transform improves the readability of the generated kernel and also prevents unnecessary specialization of cases during compilation.

Group Assignments Across Branches

Restructure and reorganize the generated code such that each assignment happens only once. Many of the same assignments are performed in different branches in the code generated from the symmetrization process. For each assignment, we identify all the equivalence conditions under which that particular assignment is performed. We create a new block that is branched to if any of the equivalence conditions *and* the overarching monotonically increasing conditional requirement is satisfied and in which the assignment is performed. We also merge blocks where the branching conditions are the same. 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.

Concordize Tensors

Transpose tensors to make the iteration of indices concordant [2]—in other words, so that the order of the indices with which the tensor is accessed aligns with the loop order of the kernel. If necessary, transpose the tensor *and* reorder the loops to make iteration concordant.

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.

Multiple Loop Nests

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.

Technique	Before	After

Common Tensor Access Elimination	`y[i] += A[i, j] * x[j]`
	`y[j] += A[i, j] * x[i]`
		`A = A[i, j]`
		`y[i] += A * x[j]`
		`y[j] += A * x[i]`

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

Restrict Computation of Output to Canonical Triangle	`for j=_, i=_`
	`    if i <= 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] += A[i, j] * x[j]`
		`for j=_, i=_`
		`    if i > j`
		`        y[i, j] = y[j, i]`

Consolidate Conditional Blocks	`if i == j`
	`    y[i] += A[i, j] * x[j]`
	`if i != j`
	`    y[i] += A[i, j] * x[j]`
		`if (i == j) || (i != j)`
		`     y[i] += A[i, j] * x[j]`

Group Assignments Across Branches	`if i != j`
	`    y[i] += A[i, j] * x[j]`
	`    y[j] += A[i, j] * x[i]`
	`if i == j`
	`    y[i] += A[i, j] * x[j]`
		`if i != j || i == j`
		`    y[i] += A[i, j] * x[j]`
		`if i != j`
		`    y[i] += A[i, j] * x[i]`

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

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

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

Table 5.1: Optimization Transforms

5.2 MTTKRP Demonstration

Let us take a look at how we would apply the methodology described in the prior section for MTTKRP given by

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

where tensor A is fully-symmetric and B is not symmetric.

We begin with the symmetrization phase. The set of permutable indices is given by $P = \{i, k, l\}$. We establish an ordering of the permutable indices in P—i, k, l—such that if these indices are monotonically increasing, we will only access the canonical triangle of A. We will use the pseudocode given in 5.1 to generate the code. Our set up thus far is depicted in 5.2.

```
1:  for $j = 1 : \_, l = 1 : \_, k = 1 : \_, i = 1 : \_$ do
2:      if $i \leq k \leq l$ then
3:          for each $E$ of $P$ do
4:              Construct $S_P|E$
5:              for each $\sigma \in S_P|E$ do
6:                  $(i, k, l) = \sigma((i, k, l))$
7:                  $C[i, j] = A[i, k, l] * B[l, j] * B[k, j]$
8:              end for
9:          end for
10:     end if
11: end for
```

Figure 5.2: MTTKRP Symmetrization: Set up pseudocode.

All the equivalence groups that can be constructed from P and which satisfy the that $i \leq k \leq l$ are delineated below.

$$\{(i), (k), (l)\}$$
$$\{(i = k), (l)\}$$
$$\{(i), (k = l)\}$$
$$\{(i = k = l)\}$$

Note that the monotonically-increasing requirement for the permutable indices naturally enforces that if two-nonadjacent indices p_i and p_{i+k} in the canonical order are equivalent, all the indices that are between p_i and p_{i+k} in the canonical order are also equivalent. Thus, we do not include the case $\{(i = l), (k)\}$ because it is encapsulated by $\{(i = k = l)\}$.

We can unroll the for loop given by lines 3-8 in Figure 5.2 as shown in Figure 5.3.

```
for $j = 1 : \_, l = 1 : \_, k = 1 : \_, i = 1 : \_$ do
    if $i \leq k \leq l$ then
        if $E = \{(i), (k), (l)\}$ then
            Construct $S_P|E$
            for each $\sigma \in S_P|E$ do
                $(i, k, l) = \sigma((i, k, l))$
                $C[i, j] = A[i, k, l] * B[l, j] * B[k, j]$
            end for
        end if
        if $E = \{(i = k), (l)\}$ then
            Construct $S_P|E$
            for each $\sigma \in S_P|E$ do
                $(i, k, l) = \sigma((i, k, l))$
                $C[i, j] = A[i, k, l] * B[l, j] * B[k, j]$
            end for
        end if
        if $E = \{(i), (k = l)\}$ then
            Construct $S_P|E$
            for each $\sigma \in S_P|E$ do
                $(i, k, l) = \sigma((i, k, l))$
                $C[i, j] = A[i, k, l] * B[l, j] * B[k, j]$
            end for
        end if
        if $E = \{(i = k = l)\}$ then
            Construct $S_P|E$
            for each $\sigma \in S_P|E$ do
                $(i, k, l) = \sigma((i, k, l))$
                $C[i, j] = A[i, k, l] * B[l, j] * B[k, j]$
            end for
        end if
    end if
end for
```

Figure 5.3: MTTKRP Symmetrization: Unroll block looping through equivalence groups.

Next, we determine the unique symmetry group $S_P|E$ for each equivalence group E.

$$
\begin{aligned}
&E = \{(i), (k), (l)\} \rightarrow S_P|E = \{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}\\
&E = \{(i = k), (l)\} \rightarrow S_P|E = \{(1, 2, 3), (1, 3, 2), (3, 1, 2)\}\\
&E = \{(i), (k = l)\} \rightarrow S_P|E = \{(1, 2, 3), (2, 1, 3), (3, 1, 2)\}\\
&E = \{(i = k = l)\} \rightarrow S_P|E = \{(1, 2, 3)\}
\end{aligned}
$$

We can add these symmetry groups to our pseudocode as shown in Figure 5.4.

```
 1: for $j = 1 : \_, l = 1 : \_, k = 1 : \_, i = 1 : \_$ do
 2:     if $i \leq k \leq l$ then
 3:         if $E = \{(i), (k), (l)\}$ then
 4:             for each $\sigma \in \{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$ do
 5:                 $(i,k,l) = \sigma((i,k,l))$
 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 each $\sigma \in \{(1,2,3),(1,3,2),(3,1,2)\}$ do
11:                 $(i,k,l) = \sigma((i,k,l))$
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 each $\sigma \in \{(1,2,3),(2,1,3),(3,1,2)\}$ do
17:                 $(i,k,l) = \sigma((i,k,l))$
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 each $\sigma \in \{(1,2,3)\}$ do
23:                 $(i,k,l) = \sigma((i,k,l))$
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 5.4: MTTKRP Symmetrization: We construct the unique symmetry groups given each equivalence group.

Now we unroll the blocks given by lines 4-6, 9-11, 14-16, and 19-21 in 5.4 to obtain 5.5.

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

Figure 5.5: MTTKRP Symmetrization: We unroll the blocks looping through the permutations in the symmetry groups.

Lastly, we normalize all the assignments. The symmetrized code generated by the above process after normalization is given in Listing 5.1.

```
function mttkrp(C, A, B)
    for l=_, j=_, k=_, i=_
        if i <= k && k <= l
            if i != k && k != l
                C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
                C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
                C[k, j] += A[i, k, l] * B[i, j] * B[l, j]
                C[k, j] += A[i, k, l] * B[i, j] * B[l, j]
                C[l, j] += A[i, k, l] * B[i, j] * B[k, j]
                C[l, j] += A[i, k, l] * B[i, j] * B[k, j]
            if i == k && k != l
                C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
                C[i, j] += A[i, k, l] * B[k, j] * B[l, j]
                C[l, j] += A[i, k, l] * B[i, j] * B[k, j]
            if i != k && k == l
                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 5.1: Normalized Symmetric MTTKRP Kernel

We proceed to the optimization phase. First, we apply the common tensor access elimination transform in Listing 5.2 where we replace repeated accesses of particular coordinates of A and B with constants (line 3).

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

Listing 5.2: MTTKRP: Common Tensor Access Elimination

Subsequently, given that several assignments are repeated within a conditional block, we apply the distributive assignment grouping transform and replace the repeated assignments with one assignment that doubles the right-hand side in Listing 5.3.

```
1  function mttkrp(C, A, B)
2      for l=_, j=_, k=_, i=_
3          A = A[i, k, l], B_ij = B[i, j], B_kj = B[k, j], B_lj = B[l, j]
4          if i <= k && k <= l
5              if i != k && k != l
6                  C[i, j] += 2 * A * B_kj * B_lj
7                  C[k, j] += 2 * A * B_ij * B_lj
8                  C[l, j] += 2 * A * B_ij * B_kj
9              if i == k && k != l
10                 C[i, j] += 2 * A * B_kj * B_lj
11                 C[l, j] += A * B_ij * B_kj
12             if i != k && k == l
```

```
13                C[i, j] += A * B_kj * B_lj
14                C[k, j] += 2 * A * B_ij * B_lj
15            if i == k && k == l
16                C[i, j] += A * B_kj * B_lj
```

Listing 5.3: MTTKRP: Distributive Assignment Grouping

Given that there are no pairs of assignments with swapped indices accessing C with equivalent right-hand sides, we can identify that there is no visible output symmetry in this kernel. Thus, we skip the output symmetry transform and move on to the consolidating conditional blocks transform. It is possible that in Listing 5.3, the blocks given by 9-11 and 12-14 have equivalent assignments because the have the same number of assignments. By swapping equivalent indices, we can actually rewrite the three assignments in both blocks to be

```
C[i, j] += A * B_kj * B_lj
C[l, j] += A * B_ij * B_kj
C[k, j] += A * B_ij * B_lj.
```

Thus, we can combine the conditional blocks on lines 9-11 and 12-14 to give us Listing 5.4.

```
function mttkrp(C, A, B)
    for l=_, j=_, k=_, i=_
        A = A[i, k, l], B_ij = B[i, j], B_kj = B[k, j], B_lj = B[l, j]
        if i <= k && k <= l
            if i != k && k != l
                C[i, j] += 2 * A * B_kj * B_lj
                C[k, j] += 2 * A * B_ij * B_lj
                C[l, j] += 2 * A * B_ij * B_kj
            if (i == k && k != l) || (i != k && k == l)
                C[i, j] += A * B_kj * B_lj
                C[l, j] += A * B_ij * B_kj
                C[k, j] += A * B_ij * B_lj
            if i == k && k == l
                C[i, j] += A * B_kj * B_lj
```

Listing 5.4: MTTKRP: Consolidate Conditional Blocks

We observe that j is consistently the second index used to access both B and C. We can make the loop order concordant by transposing both B and C and moving the j loop to be innermost loop since A is not accessed with j, as shown in Listing 5.5. We just need to perform an additional post-processing step now of transposing the output tensor back after this kernel.

```
function mttkrp(C, A, B)
    for l=_, k=_, i=_, j=_
        A = A[i, k, l], B_ji = B_T[j, i], B_jk = B_T[j, k], B_jl = B[j, l]
        if i <= k && k <= l
            if i != k && k != l
                C_T[j, i] += 2 * A * B_jk * B_jl
                C_T[j, k] += 2 * A * B_ji * B_jl
                C_T[j, l] += 2 * A * B_ji * B_jk
            if (i == k && k != l) || (i != k && k == l)
                C_T[j, i] += A * B_jk * B_jl
                C_T[j, l] += A * B_ji * B_jk
                C_T[j, k] += A * B_ji * B_jl
            if i == k && k == l
                C_T[j, i] += A * B_jk * B_jl
```

Listing 5.5: MTTKRP: Concordize Tensors

Lastly, we separate the conditional blocks handling the diagonals of A into a new loop nest as depicted in Listing 5.6. To avoid doubling accesses to A from doing so, we additionally construct two separate tensors, `A_nondiag` and `A_diag` which hold all the elements in the canonical triangle of A that are not on any diagonal and that are on some diagonal, respectively.

```
function mttkrp(C, A_nondiag, A_diag, B)
    for l=_, k=_, i=_, j=_
        A = A_nondiag[i, k, l], B_ji = B_T[j, i], B_jk = B_T[j, k], B_jl = B[j, l]
        if i <= k && k <= l
            if i != k && k != l
                C_T[j, i] += 2 * A * B_jk * B_jl
                C_T[j, k] += 2 * A * B_ji * B_jl
                C_T[j, l] += 2 * A * B_ji * B_jk
    for l=_, k=_, i=_, j=_
        A = A_diag[i, k, l], B_ji = B_T[j, i], B_jk = B_T[j, k], B_jl = B[j, l]
        if i <= k && k <= l
            if (i == k && k != l) || (i != k && k == l)
                C_T[j, i] += A * B_jk * B_jl
                C_T[j, l] += A * B_ji * B_jk
                C_T[j, k] += A * B_ji * B_jl
            if i == k && k == l
                C_T[j, i] += A * B_jk * B_jl
```

Listing 5.6: MTTKRP: Separate Loop Nests

Chapter 6

Evaluation

6.1 Implementation

I developed a compiler[1] in Julia to automate the process of applying the strategies described in the prior section. Provided an assignment and a list of symmetric tensors, the compiler outputs a kernel that exploits symmetry. The compiler uses RewriteTools, the same rewriting package used by Finch. RewriteTools is a utility for term rewriting, that provides a language for finding subexpressions that satisfy specific conditions and applying predefined transformations on the matches [2]. We use this library to define a set of simplification rules and identify specific control structures, einsums, and operations to which these rules are applied. The compiler takes a Finch program expression as input and outputs executable Finch code.

The entry point to the compiler is a function called `symmetrize` with the following method signature

```
function symmetrize(program, symmetries, loop_order)
```

where `program` is a `@finch_program` instance consisting of an `ASSIGN` statement—i.e. a statement of the form `TENSOR[INDEX...]  <CALL> = EXPR` in Finch. `symmetries` consists of a dictionary of `TENSOR` symbols mapped to a set of sets of `INDEX` symbols representing the symmetry partitions of each symmetric tensor. `loop_order` consists of a list of all the `INDEX` symbols that present themselves in `program`, in the order in which they should be iterated through, from outermost loop to innermost loop. The output is a `@finch_program` instance consisting of the optimized kernel to compute the `program` input.

For instance, the call to the compiler given by Listing 6.1 will result in the output given by Listing 6.2.

```
C = :C; A = :A; B = Bx
i = index(:i); j = index(:j); k = index(:k)
prgm = @finch_program C[i, j] += A[i, k] * B[k, j]
```

[1]This compiler is currently standalone, but it can be integrated into Finch—or any library—as an extension in the future.

```
optimized_executable = symmetrize(prgm, {A: {{i, j}}}, [j, i])
```

Listing 6.1: Sample call to compiler.

```
Finch program: for j = virtual(Dimensionless),
    k = virtual(Dimensionless),
    i = virtual(Dimensionless)
  let A_ik = A[i, k]
    begin
      if <(i, k)
        begin
          C[i, j] <<+>>= *(B[k, j], A_ik)
        end
      end
      if <=(i, k)
        begin
          C[k, j] <<+>>= *(A_ik, B[i, j])
        end
      end
    end
  end
end
```

Listing 6.2: The result of printing the output of the call to the compiler in Listing 6.1.

The compiler generates code in two phases that parallels the methodology described in Chapter 5: in the *symmetrization* phase, the compiler first identifies the set of permutable indices and generates all the possible equivalence groups with these indices. Then, it constructs a conditional block for each equivalence group where the condition is the boolean expression representing the equivalence group and the body is a set of assignments. These assignments are produced by applying the unique permutations possible under that equivalence group to the original input assignment expression.

In the *optimization* phase, the compiler performs the transforms delineated in Section 5.1.2. In the implementation, each transform from Table 5.1 has been mapped into a rewrite rule that is then applied if applicable in this phase. For instance, the Distributive Assignment Grouping transform is formulated as the rule shown in Listing 6.3.

```
Fixpoint(Rewrite(Postwalk(@rule block(~s1..., assign(~lhs, +, ~rhs), ~s2..., assign(~lhs,
     +, ~rhs), ~s3...) => block(s1..., assign(lhs, +, call(*, 2, rhs)), s2..., s3...))))(
    ex)
```

Listing 6.3: The rewrite rule corresponding to the Distributive Assignment Grouping transform.

This rule identifies assignment statements in a Finch program that have the same left hand and right hand sides and combines them into one assignment statement with the same left hand side and the right hand side multiplied by two via a post-order traversal over the nodes in the Finch program given by `ex`. This rule is repeatedly applied until no further changes occur. A series of rewrite rules like this, albeit many more complex, are applied to the symmetrized code to optimize it.

This compiler implementation is publicly available online at https://github.com/radha-patel/symmetry-compiler.

6.2 Results

In this section, we showcase the performance speedups obtained by applying the methodology described in Section 5 to Finch kernels via the compiler we implemented. We specifically focus on the six kernels introduced in Section 2.2 and restated in Table 6.1 as they encompass all the different symmetry types and strategies to take advantage of symmetry that have been discussed in this thesis. The assignment expressions given in Table 6.1 were used as input to the compiler to generate the optimized kernels. However, these assignment expressions are not necessarily the assignment used in the naive version of each of these kernels. To focus on the performance improvements from reducing memory reads and filtering redundant computations specifically, we transpose tensors as needed to ensure that the loop order is concordant with the access order.

Kernel	Assignment Expression	Symmetric Tensors
SSYMV	$y[i] = A[i,j] * x[j]$	A - $\{i, j\}$
SYPRD	$y[] = x[i] * A[i,j] * x[j]$	A - $\{i, j\}$
SSYMM	$C[i,j] = A[i,j] * B[i,j]$	A - $\{i, j\}$
SSYRK	$C[i,j] = A[i,k] * A[k,j]$	C - $\{i, j\}$
TTM	$C[i,j,l] = A[i,k,l] * B[k,j]$	A - $\{i, k, j\}$
MTTKRP	$C[i,j] = A[i,k,l] * B[k,j] * B[l,j]$	A - $\{i, k, l\}$
4D MTTKRP	$C[i,j] = A[i,k,l,m] * B[k,j] * B[l,j] * B[m,j]$	A - $\{i, k, l, m\}$

Table 6.1: Evaluation Kernels

All experiments were run on a single core of a 12-core 2-socket Intel Xeon E5-2695 v2 running at 2.40GHz 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 the symmetric compiler. 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.

We used a variety of datasets in our benchmarks, including both existing tensors that are already symmetric or have been symmetrized, and randomly generated tensors. The matrices we use (Table 6.2) are derived from the matrix benchmark suite used by Vuduc et. all [45] and downloaded from the SuiteSparse matrix repository. Given that there does not currently exist a database of symmetric tensors, we generated uniformly distributed symmetric random sparse tensors of varying sizes and sparsities via an Erdős–Rényi distribution.

We envision a similar capacity for performance enhancements by applying the given techniques to kernels generated by any library.

Group/Name	Application Area	Dimension	Nonzeros	Symmetric*
ATandT/onetone2	Harmonic balance method	36057	227628	
Boeing/bcsstk35	Stiff matrix automobile frame	30237	1450163	✓
Boeing/crystk02	FEM Crystal free vibration	13965	968583	✓
Boeing/crystk03	FEM Crystal free vibration	24696	1751178	✓
Boeing/ct20stif	CT20 Engine block	52329	2698463	✓
Brethour/coater2	Simulation of coating flows	9540	207308	
Cote/vibrobox	Vibroacoustic problem	12328	342828	✓
FIDAP/ex11	3D steady flow caculation	16614	1096948	
Goodwin/goodwin	Fluid mechanics problem	7320	324784	
Goodwin/rim	FEM fluid mechanics problem	22560	1014951	
Grund/bayer02	Chemical process simulation	13935	63679	
Grund/bayer10	Chemical process simulation	13436	94926	
Hamm/memplus	Circuit Simulation	17758	126150	
HB/gemat11	Power flow	4929	33185	
HB/lnsp3937	Fluid flow modeling	3937	25407	
HB/orani678	Economic modeling	2529	90185	
HB/saylr4	Oil reservoir modeling	3564	22316	✓
HB/sherman3	Oil reservoir modeling	5005	20033	
HB/sherman5	Oil reservoir modeling	3312	20793	
Mallya/lhr10	Light hydrocarbon recovery	10672	232633	
Mulvey/finan512	Financial portfolio optimization	74752	596992	✓
Nasa/nasasrb	Shuttle rocket booster	54870	2677324	✓
Simon/olafu	Accuracy problem	16146	1015156	✓
Simon/raefsky3	Fluid structure interaction	21200	1488768	
Simon/raefsky4	Buckling problem	19779	1328611	✓
Simon/venkat01	Flow simulation	62424	1717792	
Shyy/shyy161	Viscous flow calculation	76480	329762	
Wang/wang3	Semiconductor device simulation	26064	177168	
Zitney/rdist1	Chemical process separation	4134	94408	

Table 6.2: Symmetric Matrix Benchmark Suite

* Matrices that are not originally symmetric are symmetrized by computing $A + A^T$ when represented by a symmetric tensor. Note that the nonzero count reflects that before symmetrization.

6.2.1 SSYMV

The sparse symmetric matrix vector kernel is given by

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

The tensor formats we utilized are described in Figure 6.1. We used the matrices from the matrix benchmark suite for A (Table 6.2), symmetrized when not symmetric, and randomly generated a vector for x.

Tensor	Level Format	Dimension
A	`Dense(SparseList(Element(0.0)))`	$n \times n$
x	`Dense(Element(0.0))`	n
y	`Dense(Element(0.0))`	n

Figure 6.1: SSYMV Tensor Formats

If tensor A has sparsity p, the runtime of the naive SSYMV kernel is $O(2n^2p)$. 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.

The optimized kernel generated by the symmetric compiler performed an average of **1.36** times faster than the naive kernel (Listing A.1), with a maximal speedup of **1.85**, as shown in Figure 6.2.

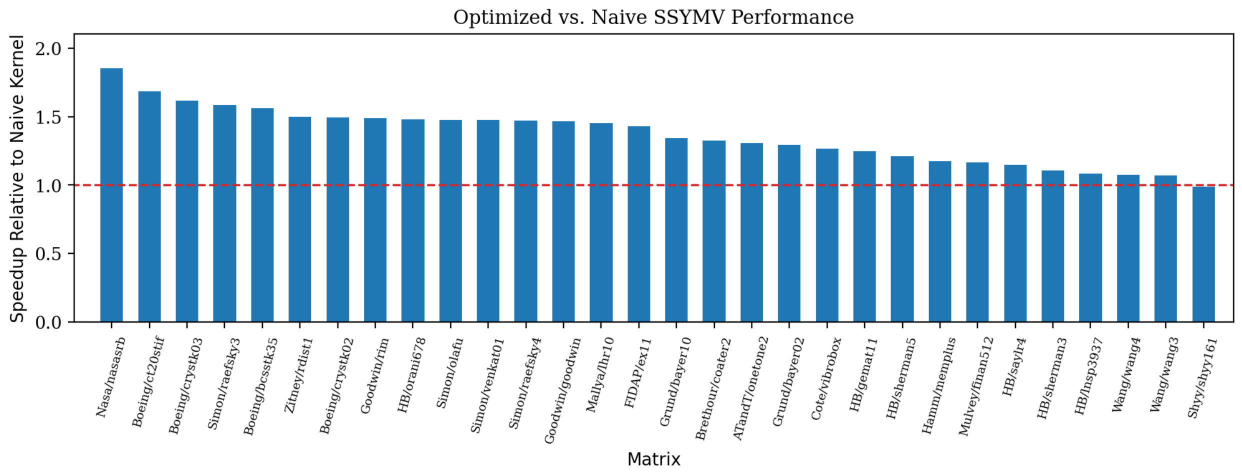


Figure 6.2: Performance of the optimized SSYMV kernel normalized to the naive SSYMV kernel.

6.2.2 SYPRD

The symmetric triple product kernel is given by

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

The tensor formats we utilized are described in Figure 6.3. We used the matrices from the matrix benchmark suite for A, symmetrized when not symmetric, and randomly generated a vector for x.

Tensor	Level Format	Dimension
A	`Dense(SparseList(Element(0.0)))`	$n \times n$
x	`Dense(Element(0.0))`	n
y	`Scalar(0.0)`	-

Figure 6.3: SYPRD Tensor Formats

If tensor A has sparsity p, the runtime of the naive SYPRD kernel is also $O(2n^2p)$. The optimized kernel accesses only $\frac{1}{2}$ of the values of A and only performs $\frac{1}{2}$ of the computations because we have $\{\{i, j\}\}$ invisible symmetry in C. Approximating the expression for large n, we can therefore expect a speedup of 2x.

The optimized kernel generated by the symmetric compiler performed an average of 1.04 times faster than the naive kernel (Listing A.2), with a maximal speedup of **1.77**. As seen in Figure 6.4, for some matrices, the optimized kernel performed well, achieving the expected speedup, but for matrices, there was nearly a 2x slowdown. In particularly, we we get a speedup for sparsities above 0.04% (the sparsity of Grund/bayer02). Zitney/rdist1 which is the leftmost bar of the plot has sparsity 0.5%, while Shy/shyy161 which is the rightmost bar of the plot has sparsity 0.005%. We suspect the slight slowdowns to be a result of control flow overhead from having separate branches to handle the diagonals and non-diagonals of A, and that being magnified when the density decreases.

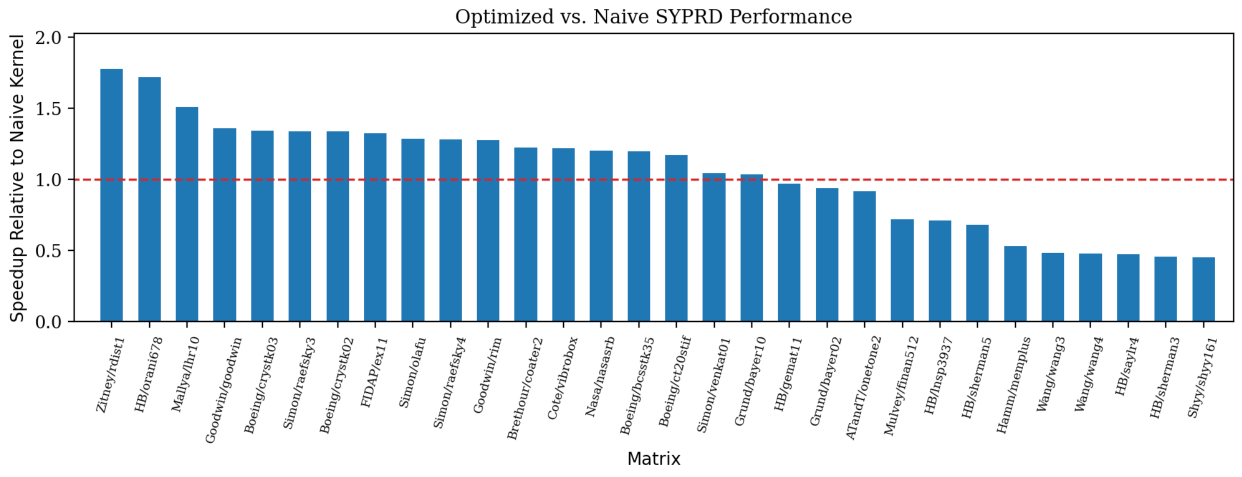


Figure 6.4: Performance of the optimized SYPRD kernel normalized to the naive SSYMV kernel.

6.2.3 SSYMM

The sparse symmetric matrix-matrix kernel is given by

$$C[i, j] = A[i, k] * B[k, j].$$

The tensor formats we utilized are described in Figure 6.5. We used the matrices from the matrix benchmark suite for A (Table 6.2), symmetrized when not symmetric, and randomly generated a matrix for B.

Tensor	Level Format	Dimension
A	`Dense(SparseList(Element(0.0)))`	$n \times n$
B	`Dense(Dense(Element(0.0)))`	$n \times r$
C	`Dense(Dense(Element(0.0)))`	$n \times r$

Figure 6.5: SSYMM Tensor Formats

If tensor A has sparsity p, the runtime of the naive SSYMM kernel is $O(n^2 rp)$. The optimized kernel accesses only $\frac{1}{2}$ of the values of A, but performs all computations. Because this kernel is compute-bound and we don't save on compute, we do not expect much speedup, which our results confirm (Figure 6.6). We suspect the slight slowdowns here too to be a result of control flow overhead from having separate branches to handle the diagonals and non-diagonals of A.

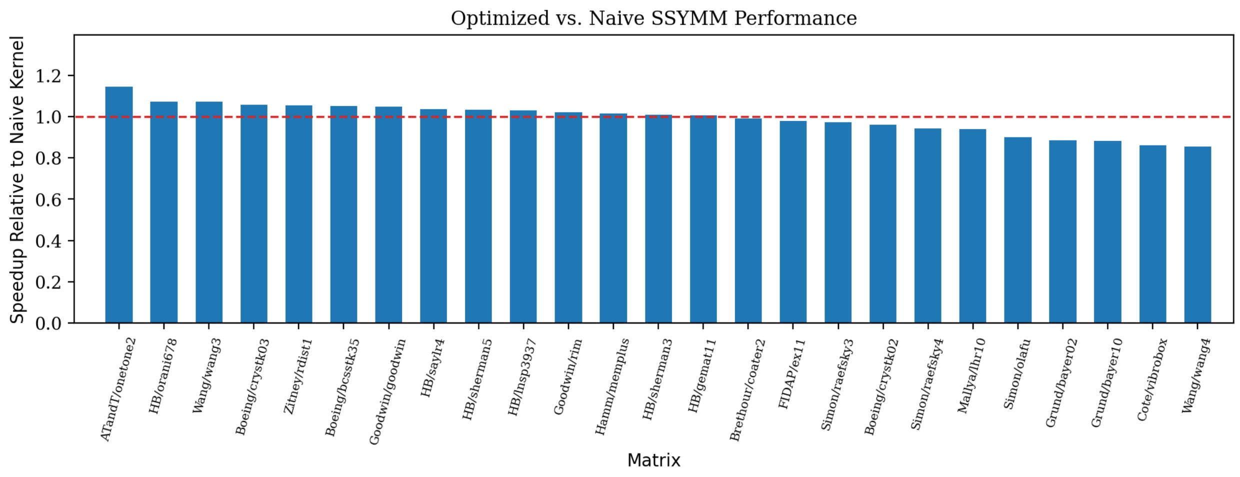


Figure 6.6: SSYMM Performance

6.2.4 SSYRK

The sparse symmetric rank-k update kernel is given by

$$C[i,j] = A[i,k] * A[k,j].$$

The tensor formats we utilized are described in Figure 6.7. We used the matrices from the matrix benchmark suite for A (Table 6.2). Note that A is not symmetric, but by nature of the computation, C is symmetric.

Tensor	Level Format	Dimension
A	`Dense(SparseList(Element(0.0)))`	$n \times n$
C	`Dense(Dense(Element(0.0)))`	$n \times n$

Figure 6.7: SSYRK Tensor Formats

If tensor A has sparsity p, the runtime of the naive SSYRK kernel is $O(n^3 p)$. 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 2.

The optimized kernel generated by the symmetric compiler performed an average of **1.14** times faster than the naive kernel (Listing A.4), with a maximal speedup of **1.76** times.

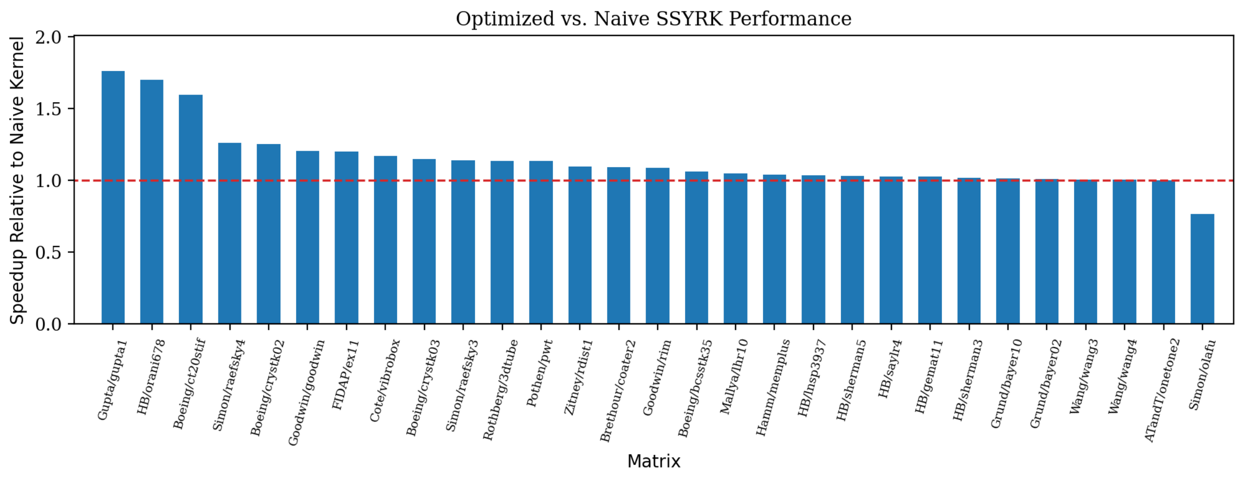


Figure 6.8: SSYRK Performance

6.2.5 TTM

The tensor times matrix kernel is given by

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

The tensor formats we utilized are described in Figure 6.9. We randomly generated symmetric tensors with a range of sizes and sparsities for A and randomly generated a dense tensor for B.

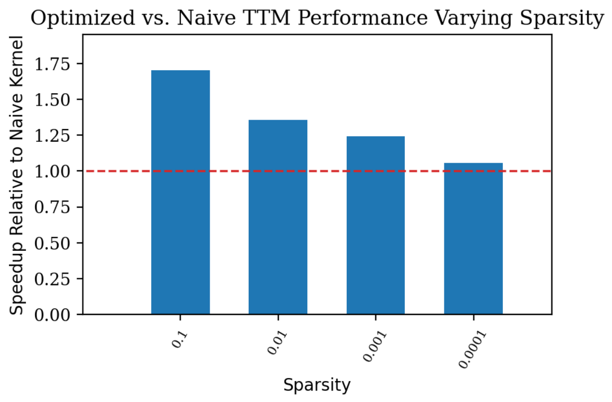
Tensor	Level Format	Dimension
A	`Dense(SparseList(SparseList(Element(0.0))))`	$n \times n \times n$
B	`Dense(Dense(Element(0.0)))`	$n \times r$
C	`Dense(Dense(Dense(Element(0.0))))`	$r \times n \times n$

Figure 6.9: TTM Tensor Formats

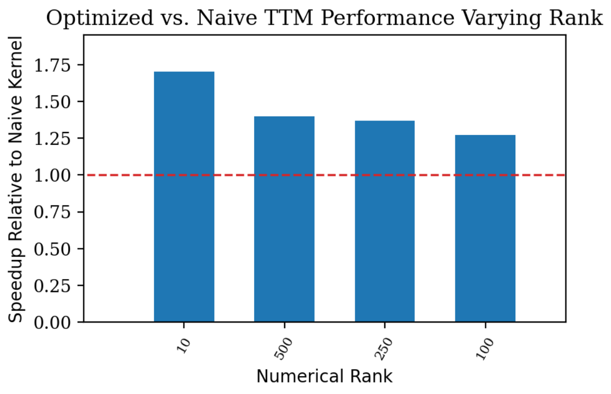
The naive and optimized kernels we benchmarked with are provided in the Appendix (Listing A.5). Notably, the optimized kernel our compiler implementation generates consists of only three assignments because of the transform to group assignments across branches.

If tensor A has sparsity p, the runtime of the naive TTM kernel is $O(n^3rp)$. 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. This theoretical speedup is maximized for high density and low rank; we find that with rank $r = 1$ (i.e. a tensor times vector computation), sparsity $p = 1$, and $n = 500$, we achieve a maximal speedup of **2.27** times.

Figure 6.10 demonstrates the effects of varying sparsity and rank. We achieve a greater speedup with greater density of tensor A (Figure 6.10a), likely because this increases memory pressure, enabling us to take greater advantage of the savings resulting from performing only $\frac{1}{6}$ of memory accesses of A. While theoretically lower rank should correspond to a greater speedup, we find that the results are not consistent (Figure 6.10b), potentially because of greater control flow overhead with lower rank.



(a)



(b)

Figure 6.10: Performance of the optimized TTM kernel normalized to the naive TTM kernel and with varying sparsity and varying rank. In plot (a), we set dimension $n = 500$ and numerical rank $r = 10$. In plot (b), we set dimension $n = 500$ and sparsity $p = 0.1$.

6.2.6 MTTKRP

The matricized tensor times Khatri-Rao product kernel is given by

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

The tensor formats we utilize are described in Figure 6.11. We randomly generated symmetric tensors with a range of sizes and sparsities for A and randomly generated a dense tensor for B.

The naive and optimized kernels we benchmarked with are provided in the Appendix (Listing A.6). The optimized kernel our compiler implementation generates consists of two loop nests, one that handles the nondiagonals and another to handle the diagonals to simplify control flow logic. Our compiler skips the grouping assignments across branches transform (which is performed in TTM) because it increases the number of conditional blocks due to the invisible symmetry in C, which causes factors of 2 to emerge in some assignments, but not others.

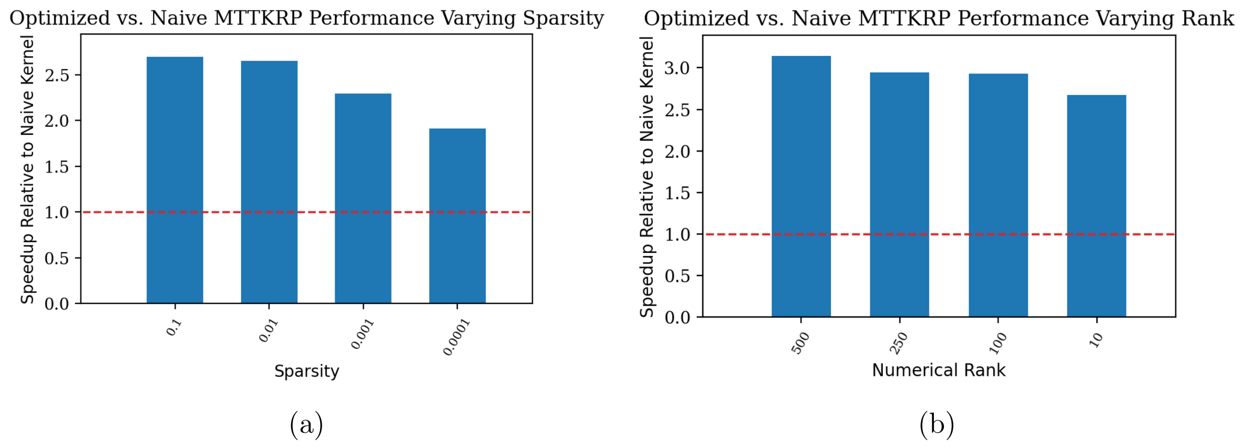
Tensor	Level Format	Dimension
A	`Dense(SparseList(SparseList(Element(0.0))))`	$n \times n \times n$
B	`Dense(Dense(Element(0.0)))`	$n \times r$
C	`Dense(Dense(Element(0.0)))`	$r \times n$

Figure 6.11: MTTKRP Tensor Formats

If tensor A has sparsity p, the runtime of the naive MTTKRP kernel is $O(n^3rp)$. 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. Thus, we expect a speedup of at least 2x. This theoretical speedup is also maximized for high density and low rank; we find that with rank $r = 1$, sparsity $p = 1$, and $n = 500$, we achieve a maximal speedup of **3.17** times.

Figure 6.12 demonstrates the effects of varying sparsity and rank. As expected, we achieve a greater speedup with greater density of tensor A (Figure 6.12a). Theoretically, lower rank should correspond to a greater speedup, but MTTKRP exhibits relatively consist speedups across different ranks (Figure 6.12b). This is also likely because the cost of reading A outweighs the cost of reading B or writing to C even when the rank is high. Interestingly, the speedup is highest when the rank is equivalent to n; this may be attributed to control flow overhead from having many branches, which proportionally has less of an impact when there are more values in the dense matrix.

In an order-3 tensor of dimension 500, 0.04% of the coordinates are diagonals. We find that the time that the loop nest to handle diagonals takes compared to the loop nest that handles the non-diagonals is not proportional to this distribution; regardless, the time to handle diagonals is still relatively insignificant, ranging from a minimum of 1% of time for rank 500 and maximum of 4% for sparsity 0.0001 (Figures 6.12c-6.12d).



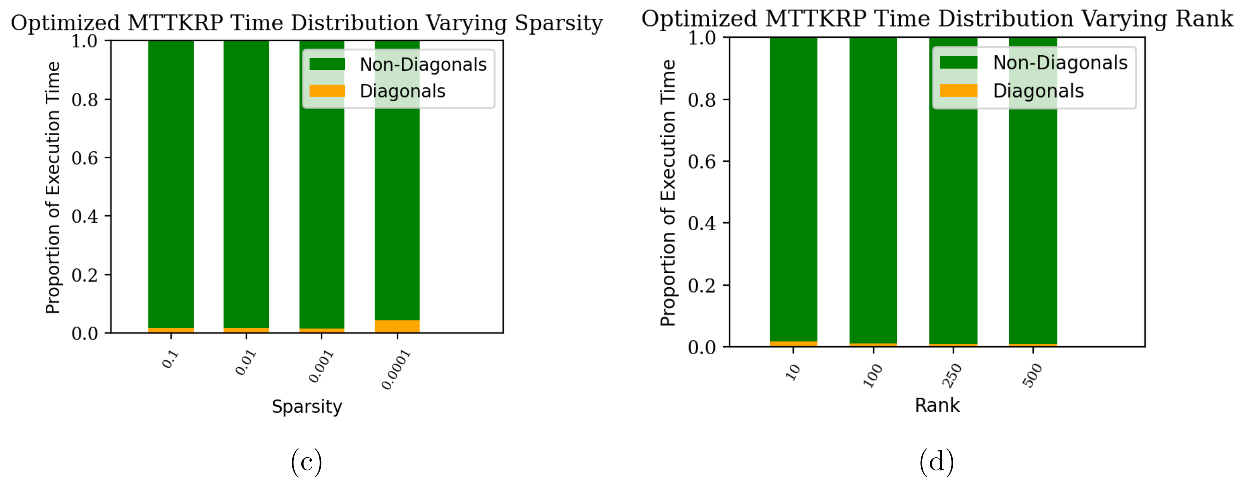


Figure 6.12: Plots (a) and (b) depict the performance of the optimized MTTKRP kernel normalized to the naive MTTKRP kernel and with varying sparsity and varying rank. Plots (c) and (d) showcase that handling the non-diagonals take the bulk of the execution time; the time to handle the diagonals (i.e. the base case) is near negligible. In plots (a) and (c), we set dimension $n = 500$ and numerical rank $r = 10$. In plots (b) and (d), we set dimension $n = 500$ and sparsity $p = 0.1$.

Let us now take a look at a 4-dimensional MTTKRP given by

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

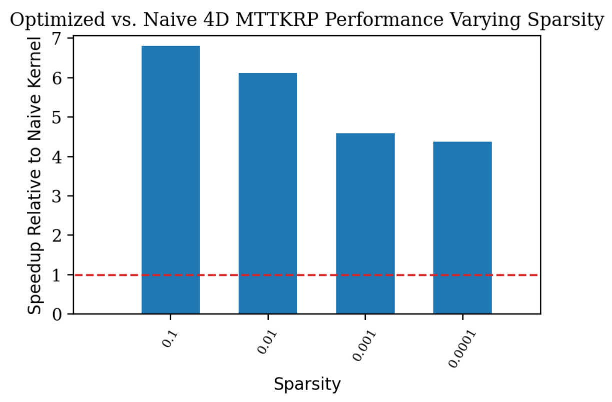
The tensor formats are equivalent to the 3-dimensional case, except we add another `SparseList` level to A to get `Dense(SparseList(SparseList(SparseList(Element(0.0)))))`.

The optimized 4-dimensional MTTKRP kernel (Listing A.7) also consists of two loops nests, one for the nondiagonals and one for the diagonals. While the loop nest for the nondiagonals remains relatively simple, we have 4 separate branches in the loop nest for the diagonals (as opposed to 2 for 3-dimensional MTTKRP).

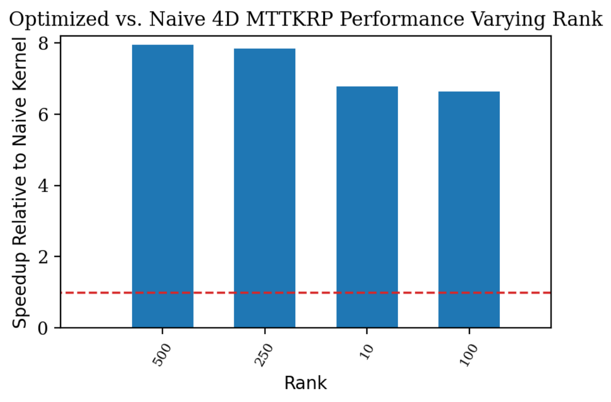
The optimized 4-dimensional MTTKRP kernel accesses only $\frac{1}{4!} = \frac{1}{24}$ of the values of A and performs $\frac{1}{3!} = \frac{1}{6}$ of the computations because we have $\{\{k, l, m\}\}$ invisible symmetry in C. Thus, we expect a speedup of at least 6x.

We expect this theoretical speedup to be maximized for high density and low rank. We achieve a speedup of **7.82** times with rank $r = 1$, sparsity $p = 1$, and $n = 100$; however, we find that the speedups are greater for greater rank. We obtain a maximal speedup of **7.94** times with sparsity 0.1 and rank 500 (Figure 6.13b). For this same case, we obtain a speedup of **10.17** times if we consider only the nondiagonal values, which suggests potential benefits from register blocking from having four assignments in the loop nest that handles the nondiagonals of the optimized kernel that all use the same memory read from A. We hypothesize that the benefits of register blocking are more pronounced when rank is higher. As expected, the performance improves as sparsity increases, with a speedup of **4.37** times for sparsity 0.0001 and (Figure 6.13b) and a speedup of **6.81** times for sparsity 0.1.

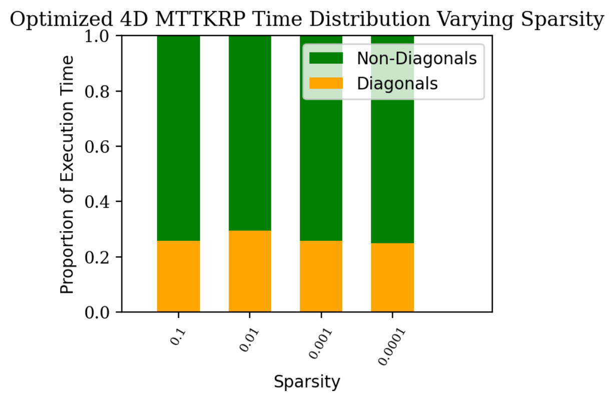
In an order-4 tensor of dimension 100, 6% of the coordinates are on diagonals. Since the number of branches to handle the diagonals increase with more dimensions of A, we expect there to be significantly greater control flow overhead. Thus, is not surprising that an average of 25.3% of the execution time of the optimized kernel in our benchmarks is from handling diagonals (Figures 6.13c-6.13d); this percent is relatively consist across sparsities and ranks.



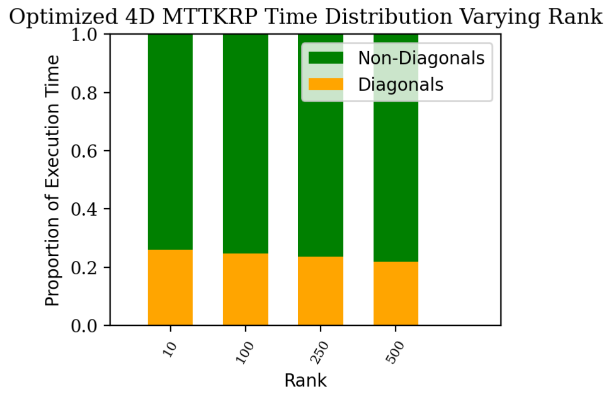
(a)



(b)



(c)



(d)

Figure 6.13: Plots (a) and (b) depict the performance of the optimized 4-dimensional MTTKRP kernel normalized to the naive 4-dimensional MTTKRP kernel and with varying sparsity and varying rank. Plots (c) and (d) showcase that handling the non-diagonals take the bulk of the execution time; the time to handle the diagonals (i.e. the base case) is near negligible. In plots (a) and (c), we set dimension $n = 100$ and numerical rank $r = 10$. In plots (b) and (d), we set dimension $n = 100$ and sparsity $p = 0.1$.

We expect the performance benefits of using a symmetric kernel to perform n-dimensional MTTKRP to scale proportionally to $n!$, albeit with increasing control flow costs from handling the nondiagonals. However, if the diagonals are all zeroed, the factorial growth is evident; for instance, we find a **54.56** times speedup with 5-dimensional MTTKRP. Evidently, as the the complexity of the kernel increases, it becomes more and more beneficial to optimize for symmetry.

Chapter 7

Conclusion

7.1 Summary

In this thesis, 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.

7.2 Future Work

While this work provides a strong foundation for exploiting symmetry in tensor kernels, several avenues for future research remain:

7.2.1 Generalizing to More Types of Symmetry

Our current methodology focuses on fully and partially symmetric tensors using the conventional definition of symmetry where the values at coordinates with permutations of indices in the same part of a symmetry partition are the same. We can expand and adapt our methodology to encompass other forms of symmetry like antisymmetry, block symmetry, or cyclic symmetry that commonly arise in physics, mathematics, chemistry, and machine learning.

7.2.2 Integration with Existing Systems

Integrating our symmetric compiler directly or implementing systematic approach we propose to exploit symmetry within Finch or existing numerical libraries such as LAPACK would significantly enhance the performance of these widely used tools. Furthermore, embedding our framework within larger systems, such a distributed tensor computation frameworks, would demonstrate its scalability and utility in real-world applications.

7.2.3 Exploring Parallelization Opportunities

The optimized code generated by the compiler I implemented is single-threaded and the naive kernels we compared it to were also single-threaded. A natural next step is exploring parallel computing techniques, such as multi-threading on shared-memory systems or distributed computing across clusters, especially for kernels involving high-dimensional tensors. This may just involve changes to the compiler implementation, or also to the methodology itself in order to effectively load balance whilst maintaining correctness.

7.2.4 Improving Compiler Heuristics and Transformations

Our methodology relies on several basic heuristics when determining which transforms to apply: for instance, we apply the transform to group assignments across branches if it results in fewer conditional blocks. Using a more thorough heuristic that considers more factors (i.e. the specific assignments across conditional blocks, dimensions in tensors, types of symmetry identified so far, etc.) could improve the efficiency of the transforms. It would also be worthwhile to explore more transformations that could be made to simplify the code to handle non-diagonals, which becomes quite complex when working with high-dimensional tensors.

7.2.5 Implementing Data Formats Tailored to Symmetry

Data formats tailored to symmetry that provide concordant accesses when iterating over different triangles of a tensor could significantly improve performance when the output tensor is symmetric. Then, we could experiment with algorithms that utilize a combination of an input-oriented and output-oriented approach—as opposed to just the input-oriented approach we propose in this thesis—since both would result in in-order memory accesses. Such data formats could also eliminate the need for or simplify extra post-processing steps like replicating the canonical triangle of a tensor to the noncanonical triangles.

7.3 Final Remarks

In conclusion, this thesis addressed a significant gap in the literature on symmetric tensors by providing a generalizable framework for generating symmetry-aware code. Our work demonstrates the potential for substantial performance improvements in tensor computations by effectively leveraging symmetry, highlighting the importance of symmetry-aware optimizations for high-performance computing applications.

Appendix A

Code listing

```
@finch_kernel function ssymv_ref(y, A, x)
    y .= 0
    for j = _, i = _
        y[i] += A[i, j] * x[j]
    end
    return y
end

@finch_kernel function ssymv_opt(y, A, x, temp)
    y .= 0
    for j = _
        temp .= 0
        for i = _
            let A_ij = A[i, j]
                if i <= j
                    y[i] += x[j] * A_ij
                end
                if i < j
                    temp[] += A_ij * x[i]
                end
            end
        end
        y[j] += temp[]
    end
    return y
end
```

Listing A.1: SSYMV Naive and Optimized Kernels

```
@finch_kernel function syprd_ref(y, A, x)
    y .= 0
    for j=_, i=_
        y[] += x[i] * A[i, j] * x[j]
    end
    return y
end

@finch_kernel function syprd_opt(y, A, x)
```

```
    y .= 0
    for j=_, i=_
        let x_i = x[i], A_ij = A[i, j], x_j = x[j]
            if i < j
                y[] += 2 * A_ij * x_i * x_j
            end
            if i == j
                y[] += A_ij * x_i * x_j
            end
        end
    end
    return y
end
```

Listing A.2: SYPRD Naive and Optimized Kernels

```
@finch_kernel function symm_ref(C, A, B)
    C_T .= 0
    for k=_, i=_, j=_
        C_T[j, i] += A[i, k] * B_T[j, k]
    end
    return C_T
end

@finch_kernel function ssymm_opt(C_T, A, B_T)
    C_T .= 0
    for k=_, i=_, j=_
        let A_ik = A[i, k]
            if i <= k
                C_T[j, i] += A_ik * B_T[j, k]
            end
            if i < k
                C_T[j, k] += A_ik * B_T[j, i]
            end
        end
    end
    return C_T
end
```

Listing A.3: SSYMM: Optimized Kernel

```
@finch_kernel function ssyrk_ref(C, A)
    C .= 0
    for k=_, j=_, i=_
        C[i, j] += A[i, k] * A[j, k]
    end
    return C
end

@finch_kernel function ssyrk_opt(C, A)
    C .= 0
    for k=_, j=_, i=_
        if i <= j
            C[i, j] += A[i, k] * A[j, k]
        end
    end
    return C
end
```

Listing A.4: SSYRK: Optimized Kernel

```
@finch_kernel function ttm_ref(C, A, B)
    C .= 0
    for l=_, j=_, k=_, i=_
        C[i, j, l] += A[k, j, l] * B_T[i, k]
    end
    return C
end

@finch_kernel ttm_opt(C, A, B_T)
    C .= 0
    for l=_, k=_, j=_, i=_
        let jk_leq = (j <= k), kl_leq = (k <= l)
            let A_jkl = A[j, k, l]
                if jk_leq && kl_leq
                    C[i, j, k] += A_jkl * B_T[i, l]
                end
                if j < k && kl_leq
                    C[i, k, l] += A_jkl * B_T[i, j]
                end
                if jk_leq && k < l
                    C[i, j, l] += A_jkl * B_T[i, k]
                end
            end
        end
    end
    return C
end
```

Listing A.5: TTM: Optimized Kernel

```
@finch_kernel function mttkrp_ref(C_T, A, B_T)
    C_T .= 0
    for l=_, k=_, i=_, j=_
        C_T[j, i] += A[i, k, l] * B_T[j, l] * B_T[j, k]
    end
    return C_T
end

@finch_kernel mttkrp_finch_opt_1(C_T, A_nondiag, B_T)
    C_T .= 0
    for l=_, k=_, i=_, j=_
        if i < k && k < l
            let B_ij = B_T[j, i], A_ikl = A_nondiag[i, k, l], B_kj = B_T[j, k], B_lj = B_T
                [j, l]
                C_T[j, l] += 2 * B_kj * B_ij * A_ikl
                C_T[j, k] += 2 * B_lj * B_ij * A_ikl
                C_T[j, i] += 2 * B_kj * B_lj * A_ikl
            end
        end
    end
    return C_T
end

@finch_kernel function mttkrp_finch_opt_2(C_T, A_diag, B_T)
    C_T .= 0
    for l=_, k=_, i=_, j=_
        if identity(i) <= identity(k) && identity(k) <= identity(l)
            let ik_eq = (i == k), kl_eq = (k == l)
                let B_ij = B_T[j, i], A_ikl = A_diag[i, k, l], B_kj = B_T[j, k], B_lj = B_T
                    [j, l]
                    if (ik_eq && !kl_eq) || (!ik_eq && kl_eq)
                        C_T[j, i] += B_kj * B_lj * A_ikl
                        C_T[j, i] += B_lj * B_ij * A_ikl
                        C_T[j, i] += B_kj * B_ij * A_ikl
                    end
                    if ik_eq && kl_eq
                        C_T[j, i] += B_kj * B_lj * A_ikl
                    end
                end
            end
        end
    end
    return C_T
end
```

Listing A.6: MTTKRP: Naive and Optimized Kernels

```
@finch_kernel function mttkrp_ref(C_T, A, B_T)
    C_T .= 0
    for m=_, l=_, k=_, i=_, j=_
        C_T[j, i] += A[i, k, l, m] * B_T[j, l] * B_T[j, k] * B_T[j, m]
    end
    return C_T
end

@finch_kernel function mttkrp_opt_1(C_T, A_nondiag, B_T)
    C_T .= 0
    for m=_, l=_, k=_, i=_, j=_
        if i < k && k < l && l < m
            let A_iklm = A_nondiag[i, k, l, m], B_T_jl = B_T[j, l], B_T_jk = B_T[j, k],
                B_T_ji = B_T[j, i], B_T_jm = B_T[j, m]
                C_T[j, m] += 6 * B_T_jl * B_T_jk * B_T_ji * A_iklm
                C_T[j, l] += 6 * B_T_jk * B_T_ji * A_iklm * B_T_jm
                C_T[j, k] += 6 * B_T_jl * B_T_ji * A_iklm * B_T_jm
                C_T[j, i] += 6 * B_T_jl * B_T_jk * A_iklm * B_T_jm
            end
        end
    end
    return C_T
end

```

```
@finch_kernel function mttkrp_opt_2(C_T, A_diag, B_T)
    C_T .= 0
    for m=_, l=_, k=_, i=_, j=_
        if identity(i) <= identity(k) && identity(k) <= identity(l) && identity(l) <=
            identity(m)
            let ik_eq = (i == k), kl_eq = (k == l), lm_eq = (l == m)
                let A_iklm = A_diag[i, k, l, m], B_T_jl = B_T[j, l], B_T_jk = B_T[j, k],
                    B_T_ji = B_T[j, i], B_T_jm = B_T[j, m]
                    if (ik_eq && !kl_eq && !lm_eq) || (!ik_eq && kl_eq && !lm_eq) || (!
                        ik_eq && !kl_eq && lm_eq)
                        C_T[j, m] += 3 * B_T_jl * B_T_jk * B_T_ji * A_iklm
                        C_T[j, l] += 3 * B_T_jk * B_T_ji * A_iklm * B_T_jm
                        C_T[j, k] += 3 * B_T_jl * B_T_ji * A_iklm * B_T_jm
                        C_T[j, i] += 3 * B_T_jl * B_T_jk * A_iklm * B_T_jm
                    end
                    if (ik_eq && !kl_eq && lm_eq)
                        C_T[j, m] += 3 * B_T_jl * B_T_jk * B_T_ji * A_iklm
                        C_T[j, k] += 3 * B_T_jl * B_T_ji * A_iklm * B_T_jm
                    end
                    if (ik_eq && kl_eq && !lm_eq) || (!ik_eq && kl_eq && lm_eq)
                        C_T[j, m] += B_T_jl * B_T_jk * B_T_ji * A_iklm
                        C_T[j, l] += B_T_jk * B_T_ji * A_iklm * B_T_jm
                        C_T[j, k] += B_T_jl * B_T_ji * A_iklm * B_T_jm
                        C_T[j, i] += B_T_jl * B_T_jk * A_iklm * B_T_jm
                    end
                    if ik_eq && kl_eq && lm_eq
                        C_T[j, m] += B_T_jl * B_T_jk * B_T_ji * A_iklm
                    end
                end
            end
        end
    end
    return C_T
end
```

Listing A.7: 4-D MTTKRP: Optimized Kernel

```
@finch_kernel function mttkrp_ref(C_T, A, B_T)
    C_T .= 0
    for n=_, m=_, l=_, k=_, i=_, j=_
        C_T[j, i] += A[i, k, l, m, n] * B_T[j, l] * B_T[j, k] * B_T[j, m] * B_T[j, n]
    end
    return C_T
end

@finch_kernel function mttkrp_opt_1(C_T, A_nondiag, B_T)
    C_T .= 0
    for n=_, m=_, l=_, k=_, i=_, j=_
        if i < k && k < l && l < m && m < n
            let A_iklmn = A_nondiag[i, k, l, m, n], B_T_jl = B_T[j, l], B_T_jk = B_T[j, k
                ], B_T_ji = B_T[j, i], B_T_jm = B_T[j, m], B_T_jn = B_T[j, n]
                C_T[j, i] += 24 * B_T_jl * A_iklmn * B_T_jk * B_T_jm * B_T_jn
                C_T[j, l] += 24 * A_iklmn * B_T_jk * B_T_ji * B_T_jm * B_T_jn
                C_T[j, k] += 24 * B_T_jl * A_iklmn * B_T_ji * B_T_jm * B_T_jn
                C_T[j, n] += 24 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jm
                C_T[j, m] += 24 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jn
            end
        end
    end
    return C_T
end

@finch_kernel function mttkrp_opt_2(C_T, A_diag, B_T)
    C_T .= 0
    for m=_, l=_, k=_, i=_, j=_
        if identity(i) <= identity(k) && identity(k) <= identity(l) && identity(l) <=
            identity(m) && identity(m) <= identity(n)
            let ik_eq = (i == k), mn_eq = (m == n), kl_eq = (k == l), lm_eq = (l == m)
                let A_iklmn = A_nondiag[i, k, l, m, n], B_T_jl = B_T[j, l], B_T_jk = B_T[j,
                     k], B_T_ji = B_T[j, i], B_T_jm = B_T[j, m], B_T_jn = B_T[j, n]
                    if (ik_eq && !kl_eq && !lm_eq && !mn_eq) || (!ik_eq && kl_eq && !lm_eq
                        && !mn_eq) || (!ik_eq && !kl_eq && lm_eq && !mn_eq) || (!ik_eq && !
                        kl_eq && !lm_eq && mn_eq)
                        C_T[j, i] += 12 * B_T_jl * A_iklmn * B_T_jk * B_T_jm * B_T_jn
                        C_T[j, l] += 12 * A_iklmn * B_T_jk * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, k] += 12 * B_T_jl * A_iklmn * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, n] += 12 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jm
                        C_T[j, m] += 12 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jn
                    end
                    if (ik_eq && !kl_eq && lm_eq && !mn_eq) || (!ik_eq && kl_eq && !lm_eq
                        && mn_eq) || (ik_eq && !kl_eq && !lm_eq && mn_eq)
                        C_T[j, i] += 6 * B_T_jl * A_iklmn * B_T_jk * B_T_jm * B_T_jn
                        C_T[j, l] += 6 * A_iklmn * B_T_jk * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, k] += 6 * B_T_jl * A_iklmn * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, n] += 6 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jm
                        C_T[j, m] += 6 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jn
                    end
                    if (ik_eq && kl_eq && !lm_eq && !mn_eq) || (!ik_eq && kl_eq && lm_eq &&
                         !mn_eq) || (!ik_eq && !kl_eq && lm_eq && mn_eq)
                        C_T[j, i] += 4 * B_T_jl * A_iklmn * B_T_jk * B_T_jm * B_T_jn
                        C_T[j, l] += 4 * A_iklmn * B_T_jk * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, k] += 4 * B_T_jl * A_iklmn * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, n] += 4 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jm
                        C_T[j, m] += 4 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jn
                    end
                    if (ik_eq && !kl_eq && lm_eq && mn_eq) || (ik_eq && kl_eq && !lm_eq &&
                        mn_eq)
                        C_T[j, i] += 2 * B_T_jl * A_iklmn * B_T_jk * B_T_jm * B_T_jn
                        C_T[j, l] += 2 * A_iklmn * B_T_jk * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, k] += 2 * B_T_jl * A_iklmn * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, n] += 2 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jm
                        C_T[j, m] += 2 * B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jn
                    end
                    if (ik_eq && kl_eq && lm_eq && !mn_eq) || (!ik_eq && kl_eq && lm_eq &&
                        mn_eq)
                        C_T[j, i] += B_T_jl * A_iklmn * B_T_jk * B_T_jm * B_T_jn
                        C_T[j, l] += A_iklmn * B_T_jk * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, k] += B_T_jl * A_iklmn * B_T_ji * B_T_jm * B_T_jn
                        C_T[j, n] += B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jm
                        C_T[j, m] += B_T_jl * A_iklmn * B_T_jk * B_T_ji * B_T_jn
                    end
                    if ik_eq && kl_eq && lm_eq && mn_eq
                        C_T[j, i] += B_T_jl * A_iklmn * B_T_jk * B_T_jm * B_T_jn
                    end
                end
            end
        end
    end
    return C_T
end
```

Listing A.8: 5-D MTTKRP: Optimized Kernel

Bibliography

[1] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. Finch: Sparse and structured array programming with control flow. *arXiv preprint arXiv:2404.16730*, 2024.

[2] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, pages 41–54, New York, NY, USA, February 2023. Association for Computing Machinery.

[3] George EP Box. Some theorems on quadratic forms applied in the study of analysis of variance problems, i. effect of inequality of variance in the one-way classification. *The annals of mathematical statistics*, pages 290–302, 1954.

[4] Jonathon Cai, Muthu Baskaran, Benoît Meister, and Richard Lethin. Optimization of symmetric tensor computations. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2015.

[5] Pierre Comon, Gene Golub, Lek-Heng Lim, and Bernard Mourrain. Symmetric tensors and symmetric tensor rank. *SIAM Journal on Matrix Analysis and Applications*, 30(3):1254–1279, 2008.

[6] Gábor Csányi and TA Arias. Tensor product expansions for correlation in quantum many-body systems. *Physical Review B*, 61(11):7348, 2000.

[7] Yujia Deng, Xiwei Tang, and Annie Qu. Correlation tensor decomposition and its application in spatial imaging data. *Journal of the American Statistical Association*, 118(541):440–456, 2023.

[8] Ernesto Dufrechou, Pablo Ezzatti, and Enrique S Quintana-Ortí. Selecting optimal spmv realizations for gpus via machine learning. *The International Journal of High Performance Computing Applications*, 35(3):254–267, 2021.

[9] Purnima Ghale and Harley T Johnson. A sparse matrix–vector multiplication based algorithm for accurate density matrix computations on systems of millions of atoms. *Computer Physics Communications*, 227:17–26, 2018.

[10] Colin R Goodall. *Computation Using the QR Decomposition*, chapter 13. Elsevier, 1993.

[11] José Henrique de M Goulart, Maxime Boizard, Rémy Boyer, Gérard Favier, and Pierre Comon. Tensor cp decomposition with structured factor matrices: Algorithms and performance. *IEEE Journal of Selected Topics in Signal Processing*, 10(4):757–769, 2015.

[12] Walter Greiner and Berndt Müller. *Quantum mechanics: symmetries*. Springer Science & Business Media, 2012.

[13] PR He and, WJ Zhang, Q Li and, and FX Wu. A new method for detection of graph isomorphism based on the quadratic form. *J. Mech. Des.*, 125(3):640–642, 2003.

[14] John D Head and Michael C Zerner. A broyden—fletcher—goldfarb—shanno optimization procedure for molecular geometries. *Chemical physics letters*, 122(3):264–270, 1985.

[15] Howard H. Hu. *Fluid Mechanics*, chapter Chapter 10 - Computational Fluid Dynamics, pages 421–272. Academic Press, 2 edition, 2012.

[16] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[17] Jakub Kurzak, Mark Gates, Ali Charara, Asim YarKhan, and Jack Dongarra. SLATE working note 12: Implementing matrix inversions. Technical Report ICL-UT-19-04, Innovative Computing Laboratory, University of Tennessee, June 2019. revision 07-2019.

[18] Pai-Wei Lai, Huaijian Zhang, Samyam Rajbhandari, Edward Valeev, Karol Kowalski, and P Sadayappan. Effective utilization of tensor symmetry in operation optimization of tensor contraction expressions. *Procedia Computer Science*, 9:412–421, 2012.

[19] Lapack — linear algebra package.

[20] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[21] Jiajia Li, Jee Choi, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. Model-driven sparse cp decomposition for higher-order tensors. In *2017 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 1048–1057. IEEE, 2017.

[22] Chunlei Liu, Roland Bammer, Burak Acar, and Michael E Moseley. Characterizing non-gaussian diffusion by using generalized diffusion tensors. *Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine*, 51(5):924–937, 2004.

[23] Haiping Lu, Konstantinos N Plataniotis, and Anastasios N Venetsanopoulos. A survey of multilinear subspace learning for tensor data. *Pattern Recognition*, 44(7):1540–1551, 2011.

[24] JOHANNES SEBASTIAN Mueller-Roemer and André Stork. Gpu-based polynomial finite element matrix assembly for simplex meshes. In *Computer Graphics Forum*, volume 37, pages 443–454. Wiley Online Library, 2018.

[25] John L Nazareth. Conjugate gradient method. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(3):348–353, 2009.

[26] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. A sparse matrix vector multiply accelerator for support vector machine. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 109–116. IEEE, 2015.

[27] James M Ortega and James M Ortega. Quadratic forms and optimization. *Matrix Theory: A Second Course*, pages 143–173, 1987.

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

[29] Szilard Pafka and Imre Kondor. Noisy covariance matrices and portfolio optimization. *The European Physical Journal B-Condensed Matter and Complex Systems*, 27:277–280, 2002.

[30] Qiyuan Pang and Haizhao Yang. A distributed block chebyshev-davidson algorithm for parallel spectral clustering. *Journal of Scientific Computing*, 98(3):1–24, 2024.

[31] Anh Huy Phan and Andrzej Cichocki. Tensor decompositions for feature extraction and classification of high dimensional datasets. *Nonlinear theory and its applications, IEICE*, 1(1):37–68, 2010.

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

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

[34] Jessica Shi, Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. An attempt to generate code for symmetric tensor computations. *arXiv preprint arXiv:2110.00186*, 2021.

[35] Sharana Kumar Shivanand, Bojana Rosić, and Hermann G Matthies. Stochastic modelling of symmetric positive definite material tensors. *Journal of Computational Physics*, page 112883, 2024.

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

[37] Daniel GA Smith, Lori A Burns, Andrew C Simmonett, Robert M Parrish, Matthew C Schieber, Raimondas Galvelis, Peter Kraus, Holger Kruse, Roberto Di Remigio, Asem Alenaizan, et al. Psi4 1.4: Open-source software for high-throughput quantum chemistry. *The Journal of chemical physics*, 152(18), 2020.

[38] E. Solomonik. Efficient algorithms for symmetric tensor contraction.

[39] Edgar Solomonik, Jeff Hammond, and James Demmel. A preliminary analysis of cyclops tensor framework. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-29*, 2012.

[40] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 813–824. IEEE, 2013.

[41] Christopher Stover and Eric W. Weisstein. Einstein summation. MathWorld–A Wolfram Web Resource, 2023.

[42] Dane Taylor, Sean A Myers, Aaron Clauset, Mason A Porter, and Peter J Mucha. Eigenvector-based centrality measures for temporal networks. *Multiscale Modeling & Simulation*, 15(1):537–574, 2017.

[43] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus Johannes Jacobus Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Aprà, Theresa L Windus, et al. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[44] Bart Vandereycken and Stefan Vandewalle. A riemannian optimization approach for computing low-rank solutions of lyapunov equations. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2553–2579, 2010.

[45] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 26–26, November 2002. ISSN: 1063-9535.

[46] Praveen Yadav and Krishnan Suresh. Large scale finite element analysis via assembly-free deflated conjugate gradient. *Journal of Computing and Information Science in Engineering*, 14(4):041008, 2014.

[47] Allen Y Yang, Kun Huang, Shankar Rao, Wei Hong, and Yi Ma. Symmetry-based 3-d reconstruction from perspective images. *Computer Vision and Image Understanding*, 99(2):210–240, 2005.