ReACT: Redundancy-Aware Code Generation for Tensor Expressions

Tong Zhou
tz@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Ruiqin Tian'
ruiqin.tian@horizon.ai
Horizon Robotics
Shanghai, China

Rizwan A. Ashraf
rizwan.ashraf@pnnl.gov
Pacific Northwest National Laboratory
Richland, Washington, USA

Roberto Gioiosa
roberto.gioiosa@pnnl.gov
Pacific Northwest National Laboratory
Richland, Washington, USA

Gokcen Kestor
gokcen.kestor@pnnl.gov
Pacific Northwest National Laboratory
Richland, Washington, USA

Vivek Sarkar
vsarkar@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

High-level programming models for tensor computations are becoming increasingly popular in many domains such as machine learning and data science. The index notation is one such model that is widely adopted for expressing a wide range of tensor computations algorithmically and also as input to programming systems. In programming systems, sparse tensors can be specified as type annotations, and a compiler can be employed to perform code generation for the specified tensor expressions and sparse formats. Different code generation strategies and optimization decisions can have a significant impact on the performance of the generated code. However, the code generation strategies used by current state-of-the-art tensor compilers can result in redundant computations being present in the output code. In this work, we identify four common types of redundancies that can occur when generating code for compound expressions, and introduce new techniques that can avoid these redundancies. Empirical evaluation on real-world compound kernels, such as Sampled Dense Dense Matrix Multiplication (SDDMM), Graph Neural Network (GNN) and Matricized-Tensor Times Khatri-Rao Product (MTTKRP) shows that our generated code with redundancy elimination can result in performance improvements of 1.1× to 25× relative to a state-of-the-art Tensor Algebra Compiler (TACO) and up to 101× relative to library approaches such as the SciPy sparse.

CCS CONCEPTS

• Software and its engineering → Source code generation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PACT 2022, October 10–12, 2022, Chicago, IL. © 2022 Association for Computing Machinery. ACM ISBN 978-1-4503-9415-5/22/04. . . $15.00
https://doi.org/10.1145/XXXX.XXXX

1 INTRODUCTION

Tensor computations are commonly found in many domains, such as machine learning, data science and scientific computing. Tensors are a generalization of matrices to an arbitrary number of dimensions, which are often large and sparse in real world, such as social networks, user-item rating matrix, etc [23]. Sparse tensors are used to represent a multi-factor or multi-relational dataset, and have found numerous applications in data analysis and mining [20, 28], health care [5], natural language processing [9], machine learning [22], and social network analytics [40], among many others.

Programs in these domains typically require a significant amount of compute power and need to consume a large volume of input data, so their performance is often critical. The traditional approach of performing sparse tensor computation is to use hand-optimized libraries, such as SciPy (sparse), Intel MKL, and cuSparse. This process not only requires a significant amount of manual work to optimize different kinds of kernels on different architectures, but can also reduce locality in compound tensor expressions due to the need to store intermediate results across library calls. A compiler-based approach will employ a compiler to automatically generate data structures and code for tensor expressions. A state-of-the-art sparse tensor compiler is the TACO compiler [18], which automatically generates fused C code for compound tensor expressions from a high-level Einstein-like notation. In its programming model, sparse and dense tensor operations are expressed uniformly by the notation but with different type annotations. A sparse tensor operand will have a sparse type, with each tensor dimension specified as dense or sparse. By default, the TACO algorithm attempts to maximally fuse all operations present in a compound expression. Our observation is that this approach can be problematic for both completeness and performance reasons.
The completeness issue is that maximum fusion will not lead to
correct code for many compound expressions, when the operators
do not satisfy distributive law. For example, division is not distrib-
utive over addition in the following expression: \( A_{ij} = B_{ij} / (C_{ik} \times D_{kj}) \)
(a matrix multiplication followed by an element-wise division),
therefore the division operation and the matrix multiplication can-
not be fused into one loop. In such cases, the TACO system reports
that such expressions are unsupported and no code will be gener-
ated. The performance issue is that redundant computations can
be introduced when always fusing all operations to the maximum
degree. However, not fusing at all, i.e. making a function call for
each individual operation, can lead to both unnecessary memory
loads and stores and redundant computations too.

In this work, we propose ReACT: a novel code generation tech-
nique to support a class of compound sparse tensor operations, and
fuses the operations in a way that results in fewer redundancies
relative to state-of-the-art code generation approaches. Specifically,
this paper makes the following contributions:

- We identify four common types of redundancies that can
appear in either maximally fused or unfused code for a se-
quence of dense or sparse tensor operations. With these
redundancy analysis in hand, we can analyze the redu-
dancy types of maximally fused and unfused code given a
computation pattern.
- We introduce a redundancy-aware fusion algorithm that
aims to obtain the benefits of generating fused code while
avoiding generation of redundant computations.
- We also introduce additional memory optimizations that
further improve the performance of the generated code by
reducing memory footprint.
- We evaluate our approach on a set of commonly used sparse
tensor kernels from real world applications. The performance
results show that our approach achieved up to orders of mag-
nitude performance improvement relative to a state-of-the-
art tensor algebra compiler (TACO) and the SciPy. sparse
library.

2 BACKGROUND AND MOTIVATION

This section gives an overview of the programming model used in
this work, and a motivating example.

2.1 Index Notation

Index notation or Einstein notation is a concise, expressive and
widely used programming model to express dense and sparse ten-
sor computation. It is widely adopted and supported in many com-
mon programming models to express tensor operations, such as
the numpy, einsum API in NumPy [4], PyTorch (torch.einsum)
and TensorFlow (tf.einsum). It’s also the input language in deep
learning frameworks such as Tensor Comprehension [39], and in
sparse tensor compilers such as TACO [18] and COMET [24, 38].
All these libraries and frameworks implement some variant of the
original Einstein notation to expand the expressiveness. In this
work, we adopt a consistent Einstein notation semantic as used
by numpy.einsum and the state-of-the-art compilers [18, 24]. We
refer the reader to the numpy.einsum API page for a more compre-
prehensive description of the notation. The approach proposed in this
work can work with at least the following operations:

- Element-wise addition, subtraction, multiplication and di-
vision of two tensors (or a tensor and a constant), such as
\( A_{ij} = B_{ij} + C_{ij} \).
- Unary element-wise operation on a tensor, such as sigmoid,
ReLU and tanh etc.
- Reduction operations on a tensor, such as row-wise reduc-
tion: \( A_i = \text{sum}(B_{ij}) \).
- Tensor contractions such as \( A_{ij} = B_{ikl} \times C_{klj} \).

Note that a summation or contraction index is implied if an
index variable appears on the right-hand-side tensors but not on
the left-hand-side tensor. Besides, a custom function can be specified
to express other types of reduction other than summation, such as
\( A_i = \max(B_{ij}) \). It is also possible to express operations like
MTTKRP (Matricized tensor times Khatri-Rao product) as \( A_{ir} = B_{ijk} \times D_{tr} \times C_{ktr} \), where index variable \( r \) and \( k \) are summed (\( r \) is not
a summation index although it appears in both input tensors). The
sparse formats of each tensor are specified as type annotations, and
a compiler will automatically generate code from the expression in
the backend.

Also, note that for some operation sequences, the order of evalu-
ation can result in different asymptotic time complexities, such as a
chain of matrix multiplications [24]. In this work, we assume such
order is already determined and do not attempt to reorder matrix
multiplications, etc.

2.2 Motivating Example

We use the SDDMM (Sampled Dense-Dense Matrix Multiplica-
tion) kernel that is commonly used in machine learning domain to
motivate our approach. SDDMM is essentially a dense matrix mul-
tiplication followed by a sparse-dense element-wise multiplication.
A library based approach will compute this operation sequence by
evaluating the two operations individually (Listing 1, annotated
with “Library approach”). With no fusion, this approach naturally
results in redundant memory loads and stores. Furthermore, it also
introduces redundant multiplications since \( B \) is sparse. The sparsity
of \( B \) acts like a filter, so that only the part of the matrix multiplica-
tion results are used that correspond to the position of the non-zeros
in \( B \). A compiler-based approach that employs maximum fusion
(such as TACO) can eliminate these two redundancies by fusing the
two operations together, as shown in Listing 1. However, a closer
look at the generated code reveals another type of redundancy. In
line 17, there are two multiplications (\( B \times C \times D \)), if we hoist \( B \) out
of the innermost loop, and first sum up \( C \times D \), and then multiply
the sum with \( B \), we will get the same result but with fewer multi-
plcations and memory loads. This is essentially taking advantage
of distributive law to reduce multiplications. The simplest case is
that \( (A + B) \times C \) saves one multiplication than \( A \times C + B \times C \) while
computing the same results. The optimization with hoisting \( B \) out
is a multi-dimensional version of the same kind of optimization,
where the sum is stored in an array instead of a scalar (due to \( k \) is

\[^{1}\text{The} \times \text{sign in tensor contractions can be omitted and we use .} \times \text{to represent element-
wise multiplications} \]
not the innermost loop). Our redundancy-aware approach removes this kind of redundancy, shown in "ReACT" in Listing 1.

Listing 1: C-like implementation for SDDMM $A_{ij} = B_{ij} \odot (C_{ij} \otimes D_{ij})$ assuming $B$ has a CSR format. The library-based approach produces redundant computation (computes dead values) because $B$ is sparse, not every element in $T[i, j]$ in the first loop will be used. TACO’s generated code also introduces redundant computation because when the summation along the $k$ dimension $C[i,j] \otimes D[k,j]$ is first completed and then multiplied with $B$, the number of multiplications can be reduced.

```c
// Library approach
for (jT = 0; jT < NJ; jT++)
    T[jT] += C[i,k] * D[k, B.cols[jB]];

// A state-of-the-art sparse tensor compiler (TACO)
A = sparse_dense_matmul(B, matmul(C, D))

// Maximally fused code generation
for (i = 0; i < NI; ++i)
    A[i] = sum(D[i,j])
A[i] = B[i,j] * C[i]

// Redundancy-Eliminated code generation
double t = new double[NJ];
for (i = 0; i < NI; ++i)
    for (k = 0; k < NK; ++k)
        for (jB = B.rowptrs[i]; jB < B.rowptrs[i+1]; jB++)
            T[jT++] = C[i,k] * D[k, B.cols[jB]];
```

3 REDUNDANCY CATEGORY

We first identify four types of redundancies that can be introduced during maximally fused or unfused code generation of a sequence of tensor operations.

3.1 Reduction Redundancy

We motivate reduction redundancy by using a simple expression $a = c \ast \text{sum}(b)$. This expression contains a summation along the $i$ dimension of $B$ followed a scalar multiplication. To avoid creating temporary variable, code generation strategy that employs maximal fusion would fuse the multiplication and the summation into one loop, which results in $O(NI)$ (the number of iterations in the $i$ loop) multiplications and summations. While first doing the summation and then a scalar multiplication would only require $O(1)$ multiplication. We refer to the kind of redundancy present in maximal fusion as reduction redundancy. Listing 2 shows a more realistic example (first broadcasting and pointwise-multiplying a matrix with a column vector and then reducing along the column dimension). The temporary tensor can also be multi-dimensional depending on the loop order, i.e. the result of the reduction does not have to be a scalar. In fact, SDDMM is one example that can exhibit this type of redundancy, and the optimized code is shown in "ReACT" in Listing 1.

More generally, for any expression that contains an element-wise operation $\odot$ and a reduction operation $\oplus$, this optimization is applicable to any semiring ($\mathbb{K}, \odot, \oplus$) where $\odot$ is distributive over $\oplus$. Evaluating the reduction (with the $\oplus$ operator) before the $\odot$ operation reduces the number of $\oplus$ operations performed.

Listing 2: Different code generations for $A_i = \sum_j B_{ij} C_{ij}$. The maximally fused code generation performs one more multiplication in the innermost loop. And the library approaches require storing a temporary tensor $D$.

```c
// Library approach 1
D[i,j] = B[i,j] \odot C[i]
A[i] = sum(D[i,j])

// Library approach 2
D[i] = sum(D[i,j])
A[i] = D[i] \odot C[i]

// Maximally fused code generation
for (int i = 0; i < NI; i++)
    for (int j = 0; j < NJ; j++)
        A[i] = B[i,j] \odot C[i];

// Redundancy-Eliminated code generation
double t = 0;
for (int i = 0; i < NI; i++)
    for (int j = 0; j < NJ; j++)
        t = B[i,j];
A[i] = t \times C[i];
```

3.2 Loop-Invariant Redundancy

Loop-invariant redundancy is another type of redundant computation that can be present in maximally fused code, which arises when the computation of some intermediate result is independent (invariant) of a loop level. Listing 3 shows two examples. In general, the intermediate results can be independent of any loop level. In a $n$-level nested loop, suppose the loop indices are $i_1, i_2, \ldots, i_k, \ldots, i_n$, if a subexpression is dependent of all loops from $i_1$ to $i_k$, but independent of $i_{k+1}$. Then the intermediate tensor needs to have indices that are after $k+1$ and the subexpression depends on as its dimensions, to achieve maximum redundancy elimination without changing the loop order. One example is that suppose we have a loop nest with order $i - k - j - l$ ($i$ is the outermost loop), and we have a subexpression that is only dependent on $i$ and $j$. To eliminate the redundancy while keeping the loop order, the dimension of the intermediate tensor needs to be the same as the $j$ dimension. Similar to reduction redundancy, there is also a tradeoff between the loop order and the dimension of the intermediate result. But this type of redundancy does not require distributive law.

Although reduction redundancy alone and loop-invariant redundancy alone do not change the asymptotic time complexity (the depth of the deepest loop nest), the combination of them does increase the depth of the loop nest, and thus increases asymptotic time complexity. Listing 4 shows the Graph Neural Network kernel...
3.3 Load-Store Redundancy

Load-store redundancy can occur when fusible code is not fused, leading to redundant loads and stores. After fusion, the redundant loads and stores can be eliminated, and the size of the intermediate storage can typically be reduced as well. Fusion requires more considerations when fusing sparse loops with dense loops, or fusing sparse loops together.

3.4 Dead-Value Redundancy

Dead-value redundancy arises when some values in a tensor are computed but not effectively used later. Listing 1 shows SDDMM for example. In the “Library-based” version, all elements in \( T[i,j] \) are computed in the first loop, but only those that correspond to the nonzero coordinates in \( B \) are used. The sparser \( B \) is, the more redundancy this code introduces.

More generally, consider a dataflow graph constructed from a tensor program, as long as the values computed by an operator may not be fully used by its consumers, this type of redundancies can happen. Another example is \( A[i,j] = \text{where}(B[i,j] > 0, C[i,j] + D[i,j], C[i,j] - D[i,j]) \). Here the where operator selects data from two arrays depending on the condition. As we can see, for every \((i,j)\) either \( C[i,j] + D[i,j] \) or \( C[i,j] - D[i,j] \) is used but not both.

Dead-value redundancy is closely related to high-level (array-oriented) programming models, and cannot be discovered by traditional dead code elimination for scalar values. The difference is that although a tensor as a pointer is used, but that does not mean all its values are used.

4 REDUNDANCY-AWARE CODE GENERATION

Our key contribution is that when generating code for a compound tensor expression, instead of always fusing all operators to the maximal degree or not fusing them at all, we propose a redundancy-aware fusion-based code generation. In our workflow, the input is a straight-line code sequence consisting of (dense or sparse) tensor operations. If an expression contains multiple tensor operations, we will first decompose them into individual operations (and introduce temporary tensors), analogous to three-address code in compiler intermediate languages. After the pre-processing, the input is a sequence of tensor operations.

The key steps are as follows, and are described in detail later:

1. Convert the operations into the index tree representation. Each initial tree represents an individual tensor operation. Also construct the dependence graph between the operations.
2. Perform sparse iteration space propagation.
3. Perform trie-like fusion.
4. Perform memory optimizations.
5. Generate code from the final (transformed) tree.

Note that step 2, 3 and 4 serve as optimizations and will be the main focus of this paper. When they are disabled, the generated code will still work but will be less efficient. Step 2 refines loop ranges so as to skip the iterations that compute the unused values. Step 3 improves data locality while avoiding introducing redundant computation. Step 4 is a memory optimization that reduces memory footprint and also improves locality. All these optimizations can be performed on our intermediate representation. Figure 1 shows the input program, the intermediate representation after each step of transformation and final pseudocode generated for SDDMM.

4.1 Convert To Intermediate Representation

We use two forms of intermediate representation: index tree and data dependence graph. An index tree is a tree representation of a loop nest. A data dependence graph represents the data dependences between the operations. Similar representations have been used in [29] as loop structure graph and loop level dependence graph. All transformations we introduce in this work are performed using the index tree and the data dependence graph.
4.1.1 Index Tree. An index tree is a tree to represent the structure of a loop nest. There are two types of nodes: index node and compute node. Each index node represents a loop level, and the leaf nodes are always compute nodes to represent compute statements. The parent-child relation in the tree corresponds to the nesting structure of the loop.

4.1.2 Dependence Graph. The data dependence graph is a directed acyclic graph (DAG) that represents the data dependence between the compute nodes. A dependence edge is labeled with its dependence type. For the purpose of fusion, input dependence is also included (fusing two nodes with input dependence can also improve locality). During fusion, some edges may be marked as fusion-preventing. Section 4.3 has more detail on how the data dependence graph is used during fusion.

4.1.3 Break Compound Expressions. The first step of the compilation pipeline is to convert the expressions to the intermediate representation (IR). This process requires breaking a compound expression (or a sequence of operations) into individual operations (as listed in Section 2), introducing intermediate tensors to store temporary results and then creating index trees for each operation and building the dependence graph among them. The indices of the intermediate tensors will need to be inferred from the operator. The sparse formats of the intermediate tensors will be determined later in the pipeline.

For example, SDDMM $A_{ij} = B_{ij} \cdot (C_{ik} \cdot D_{kj})$ will be converted into two individual operations: $T_{ij} = C_{ik} \cdot D_{kj}; A_{ij} = B_{ij} \cdot T_{ij}$. The constructed trees and their dependencies are shown in Figure 1b.

4.2 Sparse Iteration Spaces Propagation

The dead-value redundancy described in section 3 can be eliminated by sparse iteration space propagation. The iteration space of a tensor operation is a set consisting of all possible loop iterations for the operation. A loop that iterates over a sparse (compressed) dimension of a tensor has a sparse iteration space, in the sense that the coordinates visited by the loop are not contiguous. Listing 5 shows an example where the $i$ loop has a dense iteration space and the $j$ loop has a sparse iteration space. We also refer to the iteration space of a tensor dimension as dense or sparse depending on whether the format of the dimension is dense or sparse.

Listing 5: An example of iteration space. The $i$ loop has a dense iteration space and the $j$ loop has a sparse iteration space because it iterates over a sparse/compressed dimension (getting the coordinates using indirect memory access).

```c
for (i = 0; i < NI; ++i) {
    int row = i;
    for (j = row_start; j < row_end; j++) {
        T[jT] += C[i,k] * D[k, B.cols[j]];
    }
}
```

Note that the concept of iteration space depends on the storage format of a tensor rather than the actual sparsity of the tensor. For example, if we store a sparse tensor in dense format, the iteration space of the tensor would be dense.
We first give an overview of computing the iteration space for a compound expression (we refer the reader to [18] for more details on this part). Then we describe how the iteration spaces can be computed for individual operations connected by dependence edges and the benefits of doing so (our contribution).

In sparse tensor algebra, an addition or subtraction implies iterating over the union of the nonzeros in the operands, denoted by Boolean operator ∨ (using TACO’s notation). Multiplication or division implies iterating over the intersection of the nonzeros in the operands, denoted by Boolean operator ∧. For example, the iteration space of operation $A_i = B_i + C_i$ is denoted as $B_i ⊕ C_i$, and the iteration space of operation $A_i = B_i * C_i$ is denoted as $B_i ∧ C_i$. Note that each dimension has its own iteration space and using these rules we can construct iteration spaces for any tensor algebra expression. A dense iteration space is a true term in this notation and can be used to simplify the Boolean expressions. For SDDMM as a compound expression ($A_{ij} = B_{ij} * (C_{ik} D_{kj})$), the iteration space for each dimension can be represented as $B_i ∧ C_i$ for dimension $i$, $C_k ∧ D_k$ for dimension $k$ and $B_j ∧ D_j$ for dimension $j$ respectively. Note that the computation of iteration space does not require maximum fusion or the operators to satisfy distributive law. For example, for a variant of SDDMM: $A_{ij} = B_{ij}/(C_{ik} D_{kj})$, its iteration spaces can be computed the same way while division is not distributive over addition. In other words, the computation of iteration space is concerned about what values are used instead of what operators are applied on the values.

Our observation is that computing the iteration spaces for individual operations and fusing them later makes it possible to create a loop structure that avoids both dead-value redundancy and reduction redundancy and supports expressions such as, $A_{ij} = B_{ij}/(C_{ik} D_{kj})$ by fusing only the $i$ level of the two operations. Computing the iteration spaces for individual operations is conceptually similar to computing that of a compound expression by following the dependence edges. The difference is that once an iteration space is computed for an index variable, all index nodes that have the variable will get the same iteration space (propagation). For operation sequence, such as $T_{ij} = C_{ik} D_{kj}; A_{ij} = B_{ij} * T_{ij}$, we will follow the dependence edges and represent the iteration space for the index variables in both operation. The resulting iteration spaces for the two operations are shown in Figure 1c (only for $j$ nodes since both $i$ nodes and the $k$ node have totally dense space).

If there are multiple uses of a tensor, the iteration space originated from each use will need to be unionised. Listing 6 shows such an example where tensor $C$ is used by two other operations. In such case, the operation that computes $C$ will have an iteration space of $A_i ∧ (D_i ∨ F_i)$ for dimension $i$, and $B_j ∧ (D_j ∨ F_j)$ for dimension $j$. We call this step sparse iteration space propagation as the sparse iteration space of one operation could be propagated to another statement via the dependence edges. We also call the Boolean expressions iteration masks when the dense dimension are omitted.

4.3 Trie-Like Operation Fusion

Tensor operations typically consist of multiple layers of loops. When fusing two tensor operations together, there can be multiple possible fusing depth depending on the loop level at which the two operators are fused. Listing 7 shows two possible loop structures for SDDMM that both produce correct code. Version 1 fuses the two operations into the innermost level while version 2 will not fuse the multiplication with $B_{i,j}$ into the $i$-k- $j$ loop. Note, that version 1 is also correct because multiplication is distributive over addition.

Our key observation is that always fusing the operators to the maximum degree (all operators in the innermost loop) not only requires operators to satisfy distributive law and also introduce multiple types of redundant computations, as shown in section 3. Instead, our fusion policy represents each individual operation’s loop structure as a tree, and fuses them in a prefix tree (trie)-like style [19], i.e. only fusing the common prefix nodes. Only fusing the prefix loops and not any deeper avoids certain redundancies shown in section 3. As a result, the fusion process is equivalent to inserting each original tree into a trie. Figures 1c and 1d show the trees for the two operations in SDDMM before fusion and after the fusion. Now the $k$ dimension will first be summed, before multiplying with $B_{i,j}$, which eliminates reduction redundancy.

Furthermore, for expressions such as $A_{ij} = B_{ij}/(C_{ik} * D_{kj})$ ($B$ is sparse), it’s not possible to fuse the division and the multiplication both into the innermost level. For expressions like this, TACO will report the expression is unsupported. On the other hand, a library-based approach will not be able to eliminate the dead-value redundancy. In contrast, our approach would naturally support such expressions with all types of redundancies eliminated. The resulting code will be similar to the one in Listing 1 (with the only difference that the multiplication in line 33 will be replaced by a
division for this example). This code will be free of dead-value redundancy, has good locality, and produces correct results. An alternative code generation is to make the loop order $i - j - k$ ($k$ is innermost), which produces a strided memory access to $D$ when no tiling, but requires only a scalar variable to store the reduction results.

4.3.1 Fusion Legality. We now determine what operations can be legally fused. Fusion transformation needs to preserve the data dependencies imposed by the original program. Two operations cannot be fused if fusion causes

- (rule 1) a loop-independent dependence to become a loop-carried dependence in the reversed direction (fusion-preventing edges).
- (rule 2) a cycle in the dependence graph.

Listing 8 shows a softmax-like example where fusing two loops would improve the locality of loading from array $B$ but is prevented by rule 1. Besides, there are other trivial fusion preventing conditions such as incompatible loop ranges. An example is shown in Listing 9. We show examples with dense loops for ease of understanding, but the same legality check still applies to sparse loops. The legality check is performed uniformly on the intermediate representation regardless of dense or sparse loops.

**Listing 8:** Fusing the two loops is not legal because after fusion a loop-independent dependence (on $a$) would be a loop-carried dependence in the reverse direction.

**Listing 9:** In this example, operation 1 and 2 cannot be fused because they have incompatible loop ranges (one is N and another one is 0). As a result, operations 1 and 3 cannot be fused because fusing them without fusing operation 2 will create a dependence cycle.

```
for (i = 0; i < N; i++) // Operation 1
    a := A[i] + B[i];
for (i = 0; i < N; i++) // Operation 2
    c := b / a;
for (i = 0; i < N; i++) // Operation 3
    D[i] := c * A[i];
```

The problem of finding fusible operations is modeled as determining a solution of grouping nodes in an annotated dependence graph, where some edges are marked as fusion-preventing. In our index notation programming model, rule 1 is always satisfied, because the programming model cannot express such type of loop-carried dependence. So the only fusion-preventing edges are the edges that connect two operations with incompatible loop ranges. The goal of the fusion algorithm is to try to discover fusible operations as much as possible while not creating any dependence cycles and not fusing two nodes connected with a fusion preventing edge.

Algorithm 1 shows the algorithm to discover fusible groups (legal to fuse the operations in a group). The algorithm visits each node in the dependence graph in topological order. When visiting each node, it determines whether the node can be fused with its children. A node is fused with a child node if there is no other path from the node to the child, and the edge connecting them is not fusion-preventing. The algorithm will not create any dependence cycle because the only way to create a cycle when fusing two nodes is when there is another path between the nodes and some edge on the path is fusion-preventing. When two nodes are fused into the same group, the edges into and out from them are added to the group. The time complexity of Algorithm 1 is $O(V + E)$ if the in-degree and the out-degree of all nodes are bounded by a constant.

**Algorithm 1** Algorithm to compute fusion (fusible) groups.

```
function FindFusionGroups(G)
    nodes := get nodes in G in topological sort
    for node in nodes do
        g := getGroupHead(node)
        for (edge, child) in getOutEdges(g) do
            if edge is fusion-preventing then
                continue
            if there exists other path from node to child then
                continue
            addNodeToGroup(g, child)
        findFusionGroups(n, n)
    function GetGroupHead(n)
        if n does not belong to any group then
            return n
        return n’s group head
    function AddNodeToGroup(g, n)
        record n’s group head as g
        move all n’s out edges to that of g
        redirect all n’s edges in to g
```

Many tensor operations are expressed as multi-level loop nests. The legality check is conditioned at each loop level, when fuse in Algorithm 2 is called recursively. At each loop level, $\text{findFusionGroups}$ returns a set of index nodes (inner loops) that can be legally fused.

4.3.2 Trie-Like Fusion Algorithm. Algorithm 2 shows the algorithm for trie-like fusion once the fusible groups are determined. To simplify the algorithm, a root index node is created to include all initial index trees as its children, which semantically represents the program that contains all the operations. Given a root node, function $\text{fuse}$ tries to fuse its children nodes that 1) belong to the same fusion group (safe to fuse); 2) have the same iteration space. At each loop level ($\text{fuse call}$), before the actual fusion, we build a dependence graph for the children nodes with edge annotations (such as fusion-preventing). Then calling $\text{FindFusionGroups}$ will

\[ \text{It’s possible to fuse two loops with different iteration spaces but in this work we do not experiment with that.} \]
return fuseable groups. When a set of nodes are fused, the first node is chosen as “host node”, and will inherit all other nodes’ children (if any). Then fuse is recursively called on the host nodes.

Algorithm 2 Trie-Like fusion algorithm.

```plaintext
1: function Fuse(root)
2:     children := root.getChildren()
3:     create dependence graph \(G'\) for children and mark corresponding edges as fusion-preventing
4:     for group in FindFusionGroups(\(G'\)) do
5:         host := get first node in group
6:         for node in other nodes in group except host do
7:             if host and node have same iteration space then
8:                 move all node’s children to host
9:                 remove node
10:             end if
11:     end for
12:     Fuse(host)
```

4.3.3 Input-Sensitive Fusion. Though in the general case trie-like fusion results in more efficient code, for some special type of inputs, trie-like fusion may actually introduce more memory accesses than maximal fusion. Take Listing 1 as an example, if \(\mathbb{N} K\) is 1, then “ReACT” by using the temporary array \(T\). In general, when reduction is performed on a very short array, this issue can happen. During our evaluation, we did notice a few inputs of such case for MTTKRP. If the compiler knows such input information at compile-time, it can perform maximal fusion instead of trie-like fusion for the selected indices (dimensions) for which the loop iteration number tends to be very small, such as just 1. We defer finer-grain and more adaptive fusion decisions to future work.

4.4 Memory Optimization

Fusion also creates opportunities to reuse the memory buffers. For example, it is possible to reduce \(T_{ij}\) to a one-dimensional tensor \(T_j\) in Figure 1d and reuse its memory buffer for every \(i\) iterations. This also requires its memory to be reset before each \(i\) iteration starts since \(T_j\) needs to accumulate sum across the \(K\) dimensions. This is a generalization of what is known as scalar replacement [15]. The idea is that a loop independent true dependence between an array store and an array load (or an input dependence between two array loads), the array reference can be replaced by a scalar to reduce memory reference and enhance register usage.

This generalized form of scalar replacement to multi-dimensional tensors can be conveniently applied on the index tree IR. In general, given a pair of compute nodes where one consumes the value (intermediate tensor) computed by the other one, if the two nodes share a common index node ancestor \(id\), then this dependence is independent of loop level \(id\), thus the dimension of the intermediate tensor can be reduced. The transformation algorithm works by comparing the paths from the root node to the statement that defines a tensor, and the statements that use it. All indices that appear in the longest common prefix of the paths can be eliminated from the dimensions of the tensor. The algorithm is shown in Algorithm 3. \(T\) is an index tree after fusion. Helper function getPathFromRoot(\(n\)) returns the path consisting of nodes from the root to \(n\). Helper function getLongestCommonPrefix(\(paths\)) returns the longest common ancestor nodes of the \(paths\).

Algorithm 3 Algorithm to reduce dimensions of the intermediate tensors.

```plaintext
1: function ReduceDim(T)
2:     for def in T do
3:         paths := new Vector()
4:         defPath := getPathFromRoot(def)
5:         append defPath to paths
6:         for use in def.getUsers() do
7:             usePath := getPathFromRoot(use)
8:             append usePath to paths
9:         end for
10:         lastCommonAncestor := get last element in nodes
11:         prepend a compute node to lastCommonAncestor to reset def
12:     end for
13:     for n in nodes do
14:         eliminate dimension n.label from def and def.getUsers()
15:     end for
16: function getPathFromRoot(node)
17:     path := new Vector()
18:     while node is not NULL do
19:         append node at the front of path
20:         node := node.getParent()
21:     return path
```

We also perform a second kind of memory optimization which we call intermediate tensor sparsification whereby the intermediate tensor can use a compressed representation. Note that in the pre-processing step, a compound expression is broken into individual unit expressions by introducing intermediate tensors, which by default use dense storage. The idea of this optimization is that if all references (including both definition and usage) to an intermediate tensor have the same sparse iteration mask, this tensor can be stored and accessed using a compressed format to reduce memory footprint. Figure 1e shows the trees after memory optimizations for SDDMM.

4.5 Code Generation from Index Tree IR

We briefly describe how to generate code from the final index tree IR in this section. There are approximately three pieces in the process:

- Generate the loop structure reflected by the index tree.
- Generate loop bounds and indexing using indirect memory access for loops with sparse iteration space (for dense loops this is trivial).
- Generate the compute statements in the correct order that observes data dependences.

Step 1 can be done by traversing the final index tree in depth-first order and generating the loops accordingly (the parent loop is generated before the child loop). Step 3 can be done by traversing the dependence graph in topological sort. Step 2 is solved by past work, and we refer the interested reader to [18] [38] for more details. The final generated code is shown in Figure 1f. Note, we assume the sparse matrices \(A\) and \(B\) use CSR format without loss of generality.
When the outermost loop does not carry dependence, it’s straightforward to generate parallel code by emitting OpenMP pragmas. Algorithm 4 shows the procedure of adding OpenMP pragmas to the outermost loop, which should provide sufficient parallelism. Note that the allocation, de-allocation and the accesses to the temporary buffers are thread-private.

Algorithm 4 Algorithm to add OpenMP pragmas.

1: function genOpenMPPragmas(tree)
2:    generate pointer declarations.
3:    generate "#pragma omp parallel"
4:    generate code block start "{"
5:    generate loops as described in Section 4.5
6:    let i be the index of the outermost loop
7:    if i is not a summation index then
8:        annotate the i loop with "#pragma omp parallel for".
9:    generate code block end "}".

5 EVALUATION

5.1 Methodology and Kernels

To evaluate the technique described in this paper, we compare it with a state-of-the-art tensor algebra compiler, TACO [31], and a popular sparse tensor algebra library, SciPy’s sparse [2], using real-world matrices and tensors as inputs. SciPy uses Intel MKL library routines for dense matrix operations, and its own C++ implementation for sparse matrix operations. The matrices and tensors used in our evaluation are obtained from the SuiteSparse Matrix Collection [11] and the FROSTTT Tensor Collection [32]. All experiments are run on a 16-core Intel(R) Xeon(R) 2.20GHz CPU running GCC 10.3.0 (using -O3). The L1, L2 and L3 data cache sizes are 256KB, 2MB and 55 MB respectively and the memory capacity is 128GB. We run each configuration at least 10 times in a row and report the mean of the execution times. For shorter runs, we run them more than 10 times so that the total runtime is around 1 second. The variations of the running times are usually very small (coefficient of variation is < 5%) so the performance graphs only include mean times.

Table 1: The list of kernels evaluated in this work. The index notation representation is also shown along with the sparse/dense dimensions in each tensor. (CSR indicates that the first dimension is dense and the second dimension is sparse; all dimensions are sparse for CSF.)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Index Notation</th>
<th>Sparse</th>
<th>Dense</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDDMM</td>
<td>$A_{ij} = B_{ij} + (C_{ik} + D_{kj})$</td>
<td>B (CSR)</td>
<td>NK=16/32/64</td>
</tr>
<tr>
<td>MTTKRP</td>
<td>$A'<em>{ir} = (X</em>{ijk} + C_{kj}) + B_{ij}$</td>
<td>X (CSR)</td>
<td>r=16/32</td>
</tr>
<tr>
<td>GNN-kernel1</td>
<td>$Z_{ij} = A_{ik} + X_{kh} + W_{ijh}$</td>
<td>A (CSR)</td>
<td>NH=128/256; NJ=16</td>
</tr>
<tr>
<td>GNN-kernel2</td>
<td>$Z_{ij} = A_{ik} + (X_{ik} + Y_{kh})$</td>
<td>A (CSR)</td>
<td>NK=16/32/64; NJ=NK</td>
</tr>
</tbody>
</table>

Table 1 summarizes the kernels we evaluate, their index notations, and the sparse dimensions in each tensor. SDDMM (Sampled Dense Dense Matrix Multiplication) is a common kernel used in data science and machine learning [21]. MTTKRP (Matricized Tensor Times Khatri-Rao Product) is the key kernel in the alternating least squares (ALS) method for Canonical Polyadic Decomposition (CPD) [33]. We used two common kernels in Graph Neural Network (GNN) [30]: one is a sparse-dense matrix multiplication (SpMM) followed by a dense matrix multiplication (MM) and the other one is SDDMM followed by an SpMM [27]. In Table 1, MTTKRP is in order-3 formulation and the evaluation order of GNN-kernel1 starts with multiplying $B$ and $C^4$. The evaluation order is the same for all configurations for all comparisons. Column “Sparse Tensor” identifies the sparse tensor in each kernel. We used CSR format for the sparse matrices, and CSF format for the sparse tensors. Column “Dense Dim Sizes” shows the dense dimension sizes that can be configured (independent of different inputs), such as the $k$ dimension in SDDMM. We choose dimension sizes that are realistic for each kernel. For all four kernels, only the sparse tensor is read from input files, all other dense matrices are randomly generated according to the configurable dimension sizes. For example, for SDDMM we generate a random matrix of dimension $NI \times NK$ where $NI$ is determined by the input sparse matrix ($B$’s first dimension size) and $NK$ is configured as 16/32/64 respectively, to test multiple sizes. All the results below are shown in normalized runtime relative to ReACT. Table 3 summarizes the sparse tensors that are used as inputs to the kernels.

Redundancy Analysis. Table 2 shows the theoretical performance analysis (analytical prediction) based on the redundancy analysis described in section 3 for the 4 kernels. The performance analyses are normalized to ReACT (so ReACT has an entry 1). Entry AS stands for “asymptotically slower” (relative to ReACT), and CS stands for “slower by a small constant” (relative to ReACT). Here we use the term “asymptotically slower” in a loose sense to indicate more significant slowdown compared to “CS”, and as some factor (such as shape size, sparsity etc) increases, the slowdown will become more significant. For SDDMM, TACO’s maximally fused code naturally eliminates dead-value redundancy but contains reduction redundancy, resulting in one more memory load and floating point multiplication in the innermost loop, and thus is categorized as “CS”, and the slowdown constant factor should be a small number. However SciPy’s code will contain dead-value redundancy due to no fusion at all, making it asymptotically slower with respect to the sparsity of matrix $B_{ij}$. For GNN-kernel1, TACO is asymptotically slower than ReACT, while SciPy is only slower by a small constant factor. This is because TACO’s code contains both reduction redundancy and loop-invariant redundancy, and the combination of these two results in code that has higher asymptotic time complexity. SciPy, on the other hand, does not have such redundancies due to no fusion at all and contains only load-store redundancy which falls into category “CS”. For GNN-kernel2, both TACO and SciPy are asymptotically slower because TACO’s code contains both reduction redundancy and loop-invariant redundancy and SciPy’s code contains dead-value redundancy. Finally, for MTTKRP, TACO’s code contains only reduction redundancy so “CS” is what is predicted by the theoretical performance analysis; SciPy is not faster.
applicable to MTTKRP since it does not support higher-order sparse tensor operations.

Table 2: Theoretical performance analysis relatively to ReACT. AS stands for "asymptotically slower", and CS stands for "slower by a small constant" (both relative to ReACT).

<table>
<thead>
<tr>
<th>Kernels</th>
<th>ReACT</th>
<th>TACO</th>
<th>SciPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDDMM</td>
<td>1</td>
<td>CS</td>
<td>AS</td>
</tr>
<tr>
<td>GNN-kernel1</td>
<td>1</td>
<td>AS</td>
<td>CS</td>
</tr>
<tr>
<td>GNN-kernel2</td>
<td>1</td>
<td>AS</td>
<td>AS</td>
</tr>
<tr>
<td>MTTKRP</td>
<td>1</td>
<td>CS</td>
<td>N/A</td>
</tr>
</tbody>
</table>

5.2 Evaluation with Manual Implementation

For this evaluation, we compare the sequential and parallel performance of code obtained from ReACT, TACO, and the SciPy library for the four kernels listed in Table 1. The code generated for ReACT is through a manual effort that mimics the code generation algorithm presented earlier. No additional tweaking or optimization is performed on the manually generated code. We expect the manually generated code to be equivalent to the one that will be implemented automatically in the future for ReACT. The performance results for ReACT were obtained using code generated by the TACO web tool [1]. Although we ran the experiments with multiple shapes (16, 32, etc.) for each kernel, the performance trends are similar across the tested shapes as shown in Table 1. Thus, for brevity, we only include the performance results using the largest shape tested.

Multi-Core Performance. Figure 2 shows the parallel performance comparison for the four kernels. For SDDMM, ReACT is 1.48x faster than TACO on average, due to the elimination of reduction redundancy (performing fewer multiplications and memory loads in the innermost loop) and orders of magnitude faster than SciPy, due to the elimination of dead-value redundancy (not computing unused values). For GNN-kernel1, ReACT is an order of magnitude faster than both TACO and SciPy, but for different reasons. TACO’s generated code for GNN-kernel1 has higher time complexity (containing both reduction redundancy and loop-invariant redundancy) than the code generated by ReACT, thus much slower. SciPy is also slower than ReACT because it does not utilize multiple cores for the SpMM computation, which accounts for most of its execution time. For GNN-kernel2, ReACT is faster than TACO for the same reason as GNN-kernel1, and faster than SciPy for the same reason as SDDMM. For MTTKRP, SciPy does not support expression of higher order sparse tensors so it is not included (a dense NumPy-array based implementation would simply run out of memory for all our inputs). In the MTTKRP performance graph, the first four inputs are 3D tensors and the last three are 4D tensors. On average, ReACT achieved 1.1x speedup over TACO due to the elimination of reduction redundancy. The performance improvement for MTTKRP is not as significant as for SDDMM because these inputs tend not to have many elements along the reduction dimensions (the amount of reduction redundancy is limited, especially for the 3D inputs).

Table 3: Summary of real-world matrices and higher-order tensors used in our evaluations.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>#Nonzeros</th>
<th>Density</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit</td>
<td>170,980 × 170,980</td>
<td>958,936</td>
<td>3 × 10⁻³</td>
<td>Circuit Simulation Problem</td>
</tr>
<tr>
<td>mac-econ</td>
<td>206,500 × 206,500</td>
<td>1,273,389</td>
<td>9 × 10⁻³</td>
<td>Economic Problem</td>
</tr>
<tr>
<td>cop20k-A</td>
<td>121,192 × 121,192</td>
<td>2,624,331</td>
<td>2 × 10⁻⁴</td>
<td>2D/3D Problem</td>
</tr>
<tr>
<td>pwtk</td>
<td>217,918 × 217,918</td>
<td>11,524,432</td>
<td>2 × 10⁻⁴</td>
<td>Structural Problem</td>
</tr>
<tr>
<td>shipscl</td>
<td>140,874 × 140,874</td>
<td>3,568,176</td>
<td>2 × 10⁻⁴</td>
<td>Structural Problem</td>
</tr>
<tr>
<td>consph</td>
<td>83,334 × 83,334</td>
<td>6,010,480</td>
<td>9 × 10⁻⁴</td>
<td>2D/3D Problem</td>
</tr>
<tr>
<td>rma10</td>
<td>46,835 × 46,835</td>
<td>2,529,092</td>
<td>1 × 10⁻³</td>
<td>Computational Fluid Dynam-ics Problem</td>
</tr>
<tr>
<td>cant</td>
<td>62,451 × 62,451</td>
<td>4,007,383</td>
<td>1 × 10⁻³</td>
<td>2D/3D Problem</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>36,417 × 36,417</td>
<td>4,344,765</td>
<td>3 × 10⁻⁴</td>
<td>Weighted Undirected Graph</td>
</tr>
<tr>
<td>bcstlc17</td>
<td>10,974 × 10,974</td>
<td>428,650</td>
<td>4 × 10⁻⁴</td>
<td>Structural Problem</td>
</tr>
</tbody>
</table>

Table 4: Summary (geometric mean) of single-core normalized execution times for all kernels.

<table>
<thead>
<tr>
<th>Kernels</th>
<th>ReACT</th>
<th>TACO</th>
<th>SciPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDDMM</td>
<td>1</td>
<td>1.80</td>
<td>66.24</td>
</tr>
<tr>
<td>GNN-kernel1</td>
<td>1</td>
<td>10.44</td>
<td>1.29</td>
</tr>
<tr>
<td>GNN-kernel2</td>
<td>1</td>
<td>19.24</td>
<td>46.34</td>
</tr>
<tr>
<td>MTTKRP</td>
<td>1</td>
<td>1.08</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Single-Core Performance. The single-core performance results for SDDMM and MTTKRP show similar trends as the parallel results, as shown in table 4. One interesting observation we’d like to note is the single-core performance of GNN-kernel1, for which more detailed performance results are shown in Figure 3. Comparing the performance of ReACT and SciPy, the trend is that as the input size increases (from left to right), ReACT’s speedup relative to SciPy becomes more significant. This is likely because when the input size gets larger, SciPy would need to store a larger temporary tensor, resulting in a larger memory footprint and lower performance relative to ReACT. For smaller inputs, however, ReACT could be slower than SciPy despite performing more fusion. This is due to the fact that the current ReACT version of GNN-kernel1 does not
Input matrices
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
Normalized Execution Time
25.4 34.5 105.7 65.9 116.7
ReACT
TACO
SciPy
(a) SDDMM (NK=64)
Input matrices
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
Normalized Execution Time
7.7 7.6 7.0 7.2 6.6 6.7 4.4 5.2 4.0 4.2 3.4 5.4 6.2 6.8 5.3 5.9 3.2 4.3 10.6 9.9
ReACT
TACO
SciPy
(b) GNN-kernel1 (NH=256, NJ=16)
Input matrices
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
Normalized Execution Time
22.0 25.8 25.8 21.5 24.4 23.3 25.1 6.3 12.3 8.6 31.7 44.1 96.7 60.1 101.3
ReACT
TACO
SciPy
(c) GNN-kernel2 (NK=NJ=64)
Input matrices
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
Normalized Execution Time
ReACT
TACO
SciPy
(d) MTTKRP (R=32)

Figure 2: Parallel performance comparison of ReACT, TACO and SciPy with 16 threads. The x-axis shows different real-world input matrices (sorted by density from high to low) and tensors. The y-axis shows normalized execution time (lower is better).

have a performant implementation of the dense GEMM part (e.g., no loop tiling, etc.), while the SciPy version invokes high-performance MKL routines for dense GEMM.

Impact of Memory Optimizations. To quantify the impact of the memory optimizations, we also tested the performance of the ReACT code with fusion only (no memory optimizations) for applicable kernels. Note that such fusion-only implementations would still be able to reuse data across the operators in a more timely manner than totally unfused code, but the memory footprint will significantly increase. Figure 3 shows the performance results of ReACT without memory optimizations (the ReACT-no-mem-opt bar) relative to ReACT with full optimizations (the ReACT bar) and SciPy. As the input size increases, the slowdown of ReACT-no-mem-opt becomes more significant due to its larger memory footprint. For SDDMM, without the memory optimizations, the ReACT code will essentially have to store the large temporary tensor created by dense GEMM, causing it to either run out of memory, like SciPy, or incur > 10x slowdown compared to the ReACT version with full optimizations. We observe that although the memory optimizations are only made possible by fusion, fusion alone without memory optimizations may not be sufficient to achieve performance improvements.

6 RELATED WORK

This work falls in the category of code generation and compiler optimization for high-level programming models. Therefore, the following discussion does not include vendor libraries or kernel-specific optimization techniques.

Sparse and Dense Tensor Compilers. The Tensor Contraction Engine [8, 14] is an early effort to perform optimized code generation (synthesis) from tensor contraction expressions, targeted at quantum chemistry. It also performs fusion but does not handle sparse tensors or avoid sparse-specific redundancies. More recent efforts on optimizing dense linear algebra computations include [34], which uses a polyhedral representation and also performs redundancy elimination. The TACO compiler [18] introduced a general code generation technique from index notation for tensor expressions, which specifies each dimension as dense or sparse.
Figure 3: Single-Core normalized execution time for GNN-kernel1 with ReACT, ReACT without memory optimizations (ReACT-no-mem-opt) and SciPy (lower is better).

TACO’s scheduling language can be used to manually specify optimizations. In a follow-on work [17], a concrete index notation was proposed as an IR to split an expression into multiple subexpressions and store temporary results. COMET [24, 38] is a MLIR-based compiler infrastructure for dense and sparse tensor algebra computations. It uses a dimension-wise sparse storage scheme and a code generation algorithm similar to TACO, but has more performance portability since it builds upon MLIR.

Redundancy Elimination in Compilers. Traditional redundancy elimination techniques for scalar programs include dead code elimination, common subexpression elimination, partial redundancy elimination and loop invariant code motion [6]. GLORE [12] uses the LER notation to perform more general loop redundancy elimination (such as in a while loop) than traditional compiler techniques. None of these approaches considers redundancies associated with different fusion depths for sparse tensor expressions or eliminates dead values that are part of a non-dead tensor.

Loop and Operator Fusion. Fusion has been studied extensively for loops [13, 16] and, more recently, in the context of kernel [26] and operator fusion [10] [25] [37]. However, fusion in the context of redundancy elimination for sparse operations has not been addressed by past work, e.g., classical loop fusion will not be able to fuse SDDMM in a way that eliminates redundant computations. Deep learning compilers and frameworks such as TensorFlow XLA [37], PyTorch JIT and TVM [10] can fuse dense element-wise/reduction operators, but they are not able to fuse dense operators with sparse operators or fuse multiple sparse operators together. A major challenge in fusing sparse operators arises from their irregular loop iteration spaces. A key contribution of this work is to show how fusion can enable redundancy elimination in the presence of sparse operators.

Compiler Optimizations for Sparse Code. In general, programs with sparse tensor operations are difficult to analyze and optimize. The Sparse Polyhedral Framework (SPF) [36][35] was introduced to integrate inspector-executor-based transformations into the polyhedral model. However, there are no general optimization techniques in SPF that can optimize across kernels, as in our work. Other related work includes [7], which focuses on identifying sets of regular sub-computations by mining for regular sub-regions in an irregular data structure.

7 CONCLUSIONS

In this work, we identify four common types of redundancies that can occur when generating code for a sequence of dense/sparse tensor operations, and introduce ReACT, a redundancy-aware code generation technique (trie-like fusion + memory footprint reduction) that can avoid these redundancies. Empirical evaluation on real-world applications such as SDDMM, GNN, and MTTKRP showed that our generated code with redundancy elimination resulted in 1.1x to orders-of-magnitude performance improvements relative to a state-of-the-art tensor algebra compiler (TACO) and library approaches such as scipy.sparse.

8 FUTURE WORK

More Adaptive Fusion. Due to its use of intermediate buffers, trie-based fusion may not always produce optimal performance when the cost of storing and loading intermediate results exceeds the benefit of redundancy elimination. Section 4.3.3 presents a simple static strategy, but it can be further improved. In addition, when multiple legal fusion choices are possible, the fusion algorithm presented in this work will essentially select a fusion choice at random, while it might be possible to develop a more sophisticated cost model to help select the best fusion choice.

Other Optimizations. Our current code generation does not include loop tiling or loop reordering, both of which can have a major impact on the performance. Developing suitable tiling and reordering strategies can be challenging in the presence of sparse tensors and different kind of hardware accelerators. Some of these topics are interesting candidates for future work.

Implementation in MLIR-based COMET compiler: As of this publication, the techniques proposed herein have been mostly implemented within the open-source COMET compiler [3]. We expect users to be able to use COMET to access the proposed redundancy reduction techniques from this paper for their algorithms and use cases. COMET also provides the opportunity for tiling and reordering, among other optimizations that can be used to deliver additional performance improvements.

9 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. The research described in this paper is part of the Data Model Convergence Initiative at Pacific Northwest National Laboratory. It was conducted under the Laboratory Directed Research and Development Program at PNNL, a multi-program national laboratory operated by Battelle for the U.S. Department of Energy (DOE). This work was also supported through DOE Office of Advanced Scientific Computing Research as part of the Center for Artificial Intelligence-focused Architectures and Algorithms. PNNL is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.