Compiling Recurrences over Dense and Sparse Arrays

SHIV SUNDRAM, Stanford University, USA
MUHAMMAD USMAN TARIQ, Stanford University, USA
FREDRIK KJOLSTAD, Stanford University, USA

We present a framework for compiling recurrence equations into native code. In our framework, users specify a system of recurrences, the types of data structures that store inputs and outputs, and scheduling commands for optimization. Our compiler then lowers these specifications into native code that respects the dependencies in the recurrence equations. Our compiler can generate code over both sparse and dense data structures, and determines if the recurrence system is solvable with the provided scheduling primitives. We evaluate the performance and correctness of the generated code on several recurrences, from domains as diverse as dense and sparse matrix solvers, dynamic programming, graph problems, and sparse tensor algebra. We demonstrate that the generated code has competitive performance to hand-optimized implementations in libraries. However, these handwritten libraries target specific recurrences, specific data structures, and specific optimizations. Our system, on the other hand, automatically generates implementations from recurrences, data formats, and schedules, giving our system more generality than library approaches.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Source code generation**.

Additional Key Words and Phrases: recurrences, sparse tensor algebra, linear algebra, dynamic programming

ACM Reference Format:
Shiv Sundram, Muhammad Usman Tariq, and Fredrik Kjolstad. 2024. Compiling Recurrences over Dense and Sparse Arrays. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 103 (April 2024), 26 pages. https://doi.org/10.1145/3649820

1 INTRODUCTION

Recurrences compute a value of a sequence based on previous values in the same sequence. Recurrences are used to express dynamic programs, graph analysis, linear solvers, and matrix factorization. Algorithms as diverse as sequence alignment, Dijkstra's algorithm, triangular matrix solves, and even Cholesky factorization can all be expressed as recurrence equations. A simple example of a recurrence is a running sum S of an array A where $S_i = S_{i-1} + A_i$. The computed value at each i depends on previously computed values. Due to this dependency, there is a limited number of ways that values can be computed. For example, one cannot compute the values backwards with i stepping from n down to 0.

Several optimized library implementations of these operations exist. However, recurrence implementations form a large design space across recurrences (e.g., the Viterbi and Floyd-Warshall algorithms), the choice of data structures (e.g., dense and sparse arrays/tensors), and different optimizations. Where a library implementation does not exist, programmers must write and hand-optimize their own kernels. Programmers are thus faced with the challenge of implementing and optimizing a complex algorithm across diverse data structures while respecting dependencies.

Authors' addresses: Shiv Sundram, shiv1@stanford.edu, Stanford University, P.O. Box 1212, Stanford, California, USA; Muhammad Usman Tariq, Stanford University, Stanford, USA; Fredrik Kjolstad, Stanford University, Stanford, USA, kjostad@stanford.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/4-ART103
https://doi.org/10.1145/3649820

Table 1. Features of different recurrences. A ✔denotes the recurrence contains that feature.

Algorithm	Type	Recurrence Dependencies	Multiple Equations	Sparsity	Masks	Timestep Variables
Prefix Sum	dynamic	✔				
Cholesky	direct solver	✔	✔	✔	✔	
Triangular Solve	direct solver	✔		✔	✔	
Fused Tri Solve	direct solver	✔	✔	✔	✔	
QR	direct solver	✔	✔	✔	✔	
LU	direct solver	✔	✔	✔	✔	
Floyd Warshall	dynamic/graph	✔				✔
Viterbi	dynamic/graph	✔		✔		✔
Needleman-Wunsch	dynamic	✔				
Gauss-Seidel	iterative solver	✔				✔
SpMV	tensor algebra			✔	✔	
SDDMM	tensor algebra			✔	✔	

Table 1 highlights the complexity of the space by listing several recurrence equations across several domains. Recurrence dependencies, multiple interleaved equations, and sparsity combine to make it hard to implement and optimize recurrences.

Existing approaches to general recurrence programming systems do not capture this full design space. Prior work on systems for solving general recurrence equations, like the Dyna language [Eisner et al. 2004], use dynamic runtime analysis to track output dependencies and determine the order in that to calculate outputs. These approaches are thus not as efficient as handwritten code. Furthermore, the Dyna approach does not generalize across data structures and also does not support optimization decisions like controlling the loop order. And other approaches to compiling numerical code to dense and sparse tensors, such as TACO [Kjolstad et al. 2017], SparseTIR [Ye et al. 2023], and MLIR SparseTensor [Bik et al. 2022], do not support recurrences.

We describe how to generate imperative code from recurrence equations, along with separate descriptions of data structures and loop nest ordering. Based on these specifications, our compiler generates bespoke code for CPUs. The major contributions of this paper are:

- A simple recurrence language that extends tensor index notation (Section 4).
- A simple dependency checking model (Section 5 and Section 6).
- An algorithm that lowers recurrences to loop nests that iterate over abstract arrays (Section 6).
- Optimizations for recurrence programs, including loop fusion and auto-parallelization (Section 8).

To evaluate these ideas, we developed a compiler for recurrences called RECUMA (RECUrrence computation MAchine). RECUMA lowers recurrences to loop nests over abstract arrays, checking dependencies and optimizing loop order during construction. It then uses ideas from prior work on sparse tensor algebra compilation [Kjolstad et al. 2019, 2017] to lower loop nests over abstract arrays to imperative C code over dense and sparse data structures, although we have extended these ideas to support triangular loops. We demonstrate RECUMA's ability to generate code with different loop nest orders as well as to generate code over different types of dense and sparse data structures, and show that these choices affect performance. Moreover, RECUMA includes a set of program optimization strategies that can be shared across different recurrence problems. We show that the performance of the generated code is competitive with handwritten implementations from existing libraries, including CXSparse [Davis 2006a,b] for sparse matrix decompositions, Parasail [Daily 2016] for sequence-alignment, Boost [Boost 2002] for Floyd-Warshall, and PolyBench [Pouchet 2016] for Gauss-Seidel. Since tensor algebra equations are recurrence equations without dependencies, we also show that tensor algebra kernels generated by RECUMA have performance competitive with those generated by TACO [Kjolstad et al. 2017].

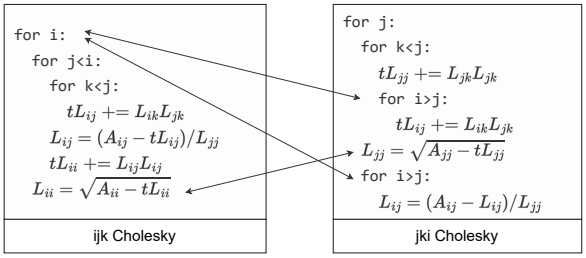

Fig. 1. Pseudocode for two Cholesky implementations with different loop ordering (ijk and jki). The arrows highlight three of the differences between the variants. Loops with no bounds implicitly iterate from [0,N).

2 CHOLESKY DECOMPOSITION EXAMPLE

To illustrate the requirements of a recurrence compiler and the complexity it must capture, we use the Cholesky decomposition as a running example. The Cholesky decomposition is used to solve a linear system involving a symmetric positive definite matrix. For any symmetric positive definite matrix A, the decomposition computes a lower triangular matrix L such that $LL^T = A$. The matrix L can then be used to solve linear systems of the form $Ax = b$ (to compute x) through a sequence of two triangular solves. The Cholesky decomposition is expressible as a system of two mutually dependent recurrence equations that together define the matrix L, where previously computed values of L are used to compute subsequent entries:

$$L_{ij} = \frac{A_{ij} - \sum_k L_{ik}L_{jk}}{L_{jj}} \;:\; k < j < i \qquad\qquad L_{ii} = \sqrt{A_{ii} - \sum_j L_{ij}L_{ij}} \;:\; j < i$$

Each recurrence equation expresses the values of the components of L as a function of both A and L itself. The recurrences have constraints on the index variables that describe what components of the result L that the equation computes and what components of the operands that it uses to compute them. The constraints specify that the first recurrence calculates only the lower triangular elements of the output (L_{ij} where $j < i$), while the second recurrence calculates the diagonal entries (L_{ii}). The upper triangular elements are not calculated by these equations and thus default to zeros. Computed values in the equations depend on other values computed in the same equation as well as values computed in the other equation. That is, diagonal components rely on the values of non-diagonal components, and vice versa. The equations are therefore inter-dependent and their computation must be interleaved. Any valid Cholesky decomposition implementation must respect both intra-recurrence and inter-recurrence dependencies.

Implementing a Cholesky decomposition with different loop orderings leads to significantly different code structure. This is in contrast to basic linear algebra expressions (e.g., matrix multiplication) in which rearranging the loop headers is sufficient to implement different loop orderings. To illustrate, the left of Figure 1 shows pseudocode for the commonly-used up-looking Cholesky decomposition. We denote this variant as an ijk Cholesky decomposition following the nesting order of the loops. The ijk loop ordering, also known as the Cholesky–Banachiewicz algorithm, is useful when the matrices A and L are stored row-major, to ensure efficient data structure accesses. If A and L are column-major, however, it is better to use a loop nest with a jki loop ordering, shown on the right of Figure 1. Notice the drastic differences between the two forms:

Loop Headers The i and j loop bounds are different due to the recurrence constraints. In the first loop nest i iterates from 0 to N, while in the second loop nest it iterates from $j + 1$ to N.

```
for(int i=0; i<N; i++) {
    //not shown: load sparse row i of A into a dense array tmpAij and init the dense
         temporary array tmpLij to build the output row i of L
    double Lii = 0;
    for(int j=0; j<i; j++) {
        double tLij = 0;
        for (int p_k_Ljk = L1_pos[j]; p_k_Ljk< L1_pos[j+1]; p_k_Ljk++) {
            double Ljk = L_vals[p_k_Ljk];
            double Lik = tmpLij_vals[Ljk_crd];
            tLij += Lik * Ljk;
        }
        double Aij = tmpAij_vals[j];
        double Ljj = L_vals[L1_pos[j]-1];
        double Lij = (Aij - tLij) / Ljj;
        tmpLij_vals[j] = Lij;
        Lii += Lij*Lij;
    }
    double Aii = tmpAij_vals[i];
    tmpLij_vals[i] = Lii = sqrt(Aii-Lii);
    //not shown: compress tmpLij into new row of output L, set L_1crd and L1_pos
}
```

Fig. 2. Sparse ijk Cholesky Factorization

Diagonal Elements The calculations of the diagonal elements L_{ii} in the first loop nest are reformulated as calculations of L_{jj} in the second nest.

Number of Loops The two implementations have different numbers of for loops. Unlike the ijk version, the jki version has two loops over i, the first of is which is the inner most loop and performs a summation. The second i loop is a middle loop and scales this summation by the inverse of L_{jj}. These two i loops cannot be fused.

Therefore, unlike a matrix multiply kernel where an ijk loop nest can be converted into a jki loop nest by simply permuting the order of the loop headers, switching the loop iteration order in a recurrence system may require more changes. Moreover, there are six possible Cholesky loop orderings corresponding to the different permutations of i, j, and k, all of which are useful in different scenarios and which have notable differences from each other. Despite these differences, the six variants share a common recurrence equation formulation. Similar recurrences systems can be defined for the LU decomposition, QR decomposition, triangular solve, as well as dynamic programming and graph problems outside linear algebra (e.g., Table 1).

Finally, when a large portion of the entries of A are zeros, we need to generate implementations of the six Cholesky variants that use sparse data structures. Figure 2 shows an implementation of the simpler ijk Cholesky variant where the matrices are stored in compressed sparse row (CSR) data structures. This code is more complicated than the dense version but the above principles still apply. Moreover, the pseudocode in Figure 1 still applies as long as we view the matrices as logical and abstract arrays for which we have not yet introduced physical data structures. In this paper we describe how to compile recurrences to code that operate on both dense and sparse data structures.

3 OVERVIEW

Our recurrence compiler takes in a declarative system of recurrence equations, dense/sparse data structure descriptions, and a schedule describing optimizations such as loop ordering. The compiler generates an imperative C implementation that computes the recurrence sequence. The compiler ensures that dependencies are observed, that the code adheres to the user-specified loop ordering, and that the code operates on the user-specified data structures.

Figure 3 shows the process of compiling declarative recurrence expressions to an imperative C implementation through two lowering passes. These passes move from the recurrences equations

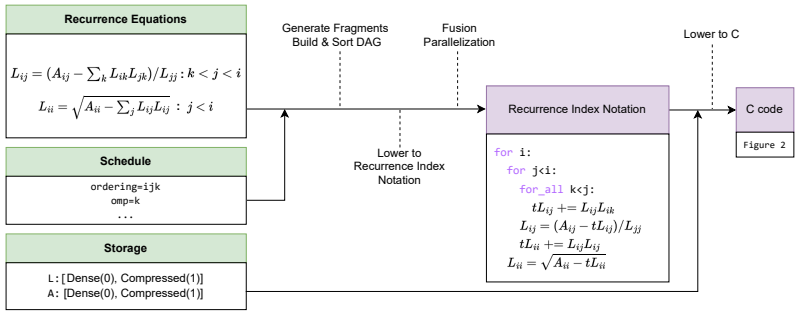

Fig. 3. An overview of the recurrence compilation workflow. Green boxes are user-provided inputs to the compiler. Purple boxes are intermediate and final implementations produced by the lowering passes.

to an imperative IR we call recurrence index notation (RIN), and finally to imperative C code. The figure illustrates the process for a dense Cholesky decomposition with an ijk loop ordering.

The output of the first lowering stage is in recurrence index notation, an imperative IR that describes a program's loop structure and placement of compute statements. All data accesses in RIN are to logical/abstract arrays. This allows the compiler to reason about the desired program's loops, computations, and dependencies without regard for how the tensors are stored. Therefore, loop reordering, dependency analysis, placement of compute statements within loop nests, and low-level program optimizations (e.g., loop fusion and auto-parallelization) can be carried out without the need to manage low-level constructs related to sparse code, such as while loops, if statements, and indirect memory accesses.

Recurrences are lowered to RIN by a new lowering algorithm. In this process, each recurrence equation (or a sub-expression of the equation) is placed as a statement into a location in an imperative loop nest. These expressions are greedily placed into the first location of a loop nest where the expression's dependencies are satisfied. The core component of this algorithm is its ability to reason about, for each iteration and location in the loop nest, what outputs have already been computed and what needs to be computed next. If it is not possible to generate a program that respects both the desired loop order and the dependencies of the recurrences, then the compiler reports an error. Once the RIN is fully formed, it is possible to apply user specified optimizations to it. Loop fusion, however, happens automatically in the RIN generation stage.

The final pass lowers RIN into C code. We use standard techniques utilized by tensor algebra compilers like TACO, COMET [Tian et al. 2021], and MLIR SparseTensor [Bik et al. 2022] for generating sparse loops, augmented to support triangular loop bounds. The final lowering pass is the only pass that requires information about the data structures in which the tensors are stored. Deferring the data structure selection to after loop optimization is an important design point that drastically simplifies the dependency handling, RIN generation, and loop optimization. The final code can use dense arrays or sparse data structures, which can be used to represent graphs, sparse matrices, and sparse arrays.

4 RECURRENCE LANGUAGE

Our language for recurrences, which we used in the Cholesky example, extends the common mathematical notation used to describe recurrence equations. The language consists of a set of equations that describe how different parts of a tensor are computed. Tensors are indexed by index variables, and scalar expressions can be summed over reduction indices. The language thus extends

the tensor index notation [Ricci and Levi-Civita 1900] that is used by many tensor algebra compilers. For example, in both our recurrence language and in tensor index notation, matrix multiplication is expressed as $A_{ij} = \sum_k B_{ik}C_{kj}$. However, the recurrence language has four features that extends tensor index notation:

Recurrences The core feature of a recurrence equation is that components of the result array can be computed based on previously computed components of the same array, thus introducing dependencies. The same array (i.e., tensor) may thus appear on both the left and right-hand sides of a recurrence equation.

Constraints Constraints bound the domain of one index variable in terms of another index variable. Constraints can be equalities or inequalities of index variables (e.g., $j = i$ or $j < i$).

Multiple equations A recurrence problem may consist of multiple recurrence equations that compute disjoint parts of the result array. Furthermore, these recurrences can be mutually dependent; result components computed in one recurrence equation may be used to calculate components in another equation, and vice versa.

Timestep Variables Timestep variables can be used to describe iterative computation in time.

The first three features were notably used in the Cholesky equations in Section 2. The fourth feature, timestep variables, is used in algorithms that have to iteratively compute a result, such as the Floyd-Warshall algorithm. Each program in our language is thus a set of recurrence equations, in which each equation has its own constraints over index variables.

4.1 Expressing Recurrence Equations

The grammar of the recurrence language is given in Figure 4. A *recurrence* consists of a recurrence equation that connects indexed elements of a result tensor to expressions required to compute them. Each tensor element is accessed by a *tensorAccess*, which denotes an element indexed by a sequence of *index* variables, which may optionally be offset by constants. A single tensorAccess forms the left-hand side of a recurrence. A recurrence with an optional list of constraints is a *constrainedRecurrence*, a set of which makes a program.

The subscripts of a tensorAccess are the index variables used to access the corresponding tensor element. The index variables in a tensorAccess subscript may each be offset by an integer constant. This is illustrated by the Fibonacci equation $F_i = F_{i-1} + F_{i-2}$, where the indexing expressions on the right-hand side are $i - 1$ and $i - 2$. While our indexing expressions do not include the full spectrum of affine indexing expressions, they are sufficient to express many important recurrences.

iVar ::= string

index :: = iVar '+' constantInt
| constantInt

tensor ::= string

tensorAccess ::= $tensor_{index^+}^{index^*}$

expr ::= tensorAccess
| $\sum_{iVar}$ expr
| constant
| '−' expr
| expr '+' expr
| expr expr
| ...

recurrence ::= tensorAccess '=' expr

constraint ::= iVar '<' index
| iVar '<=' index
| iVar '=' index

constraints :: = constraint (',' constraint)*

constrainedRecurrence ::= recurrence ':' constraints

program ::= constrainedRecurrence (',' constrainedRecurrence)*

Fig. 4. Grammar for the recurrence language

A recurrence equation's right-hand side consists of scalar arithmetic operations and summations, which operate on tensorAccesses and constants. Supported binary operations include addition, subtraction, multiplication, min, max, but can be expanded with other linear or non-linear [Henry et al. 2021] operations as needed. Similarly, unary operations include square roots and negations. Each recurrence equation is assigned a set of constraints defined over index variables. The language also includes an operator denoting a summation across an index variable for a given expression. Such reductions can, however, be built on top of any commutative and associative binary operation.

Finally, some recurrences represent iterative algorithms that iterate over a timestep k, and this variable may be used as an indexing variable when accessing an element in the output data structure. Accordingly, a tensorAccess includes an optional superscript that may contain a timestep variable. The Floyd-Warshall all-pairs shortest paths algorithm, for example, is defined as $D_{ij}^{k} = min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$, in which k denotes the current timestep. D_{ij}^{k} denotes the distance between nodes i and j at the kth timestep. Separating timestep variables from other index variables is useful; the algorithm does not require the output data structure D to store the distances from every intermediate timestep k, since the algorithm only requires knowing the last $k-1$th timestep to make progress. D can thus be implemented as a two-dimensional (instead of three-dimensional) tensor containing D_{ij}^{k}. Timestep variables give the programming system enough information to instantiate these optimizations, while simultaneously being intuitive to the programmer.

4.2 Denoting Schedule and Storage

A schedule tells a compiler how to optimize code [Ragan-Kelley et al. 2013] and can also be applied to compilers for sparse operations [Senanayake et al. 2020]. The main component of a schedule is an ordered list of index variables denoting loop ordering. As shown, a Cholesky decomposition can be implemented as a triply nested loop. If the user wants to generate a program in which the outer loop iterates over j, the middle over k, and the inner over i, then the user will simply provide a jki loop ordering to the compiler.

The final component is the description of the data structures of the result and operand tensors. Like in prior work on sparse tensor algebra compilation [Chou et al. 2018; Kjolstad et al. 2017], each tensor can have multiple dimensions and each dimension must have a level format (Dense or Compressed). The Dense keyword states that in a particular dimension, each element is stored in the data structure, including zero entries. The Compressed keyword denotes that for a particular dimension, only nonzero elements are stored. These level formats can be composed to form common sparse formats like compressed sparse rows (CSR) or compressed sparse columns (CSC). To generate a Cholesky decomposition over a CSR matrix, the matrix's associated storage type is [Dense(0), Compressed(1)]. The 0 before 1 indicates that rows are stored before columns. In a column-major matrix, dimension 1 thus is stored before 0.

By separating specifications of equations, constraints, schedules, and storage info, the user can then concisely express a variety of recurrences over dense tensors, sparse tensors, and graphs, which are ultimately either dense or sparse matrices. For a Cholesky decomposition using CSR matrices, the recurrences, storage, and loop ordering are expressed to RECUMA in Python as so:

```
rec1 = "L(i,j) = (A(i,j)-Sum{k}(L(i,k)*L(j,k)))/L(j,j) : [k<j,j<i]"
rec2 = "L(i,j) = sqrt(A(i,j)-Sum{k}(L(i,k)*L(j,k))) : [k<j,j=i]"
schedule["ordering"] = "ijk"
storage["A"] = Storage([Dense(0), Compressed(1)])
storage["L"] = Storage([Dense(0), Compressed(1)])
program = Program([rec1, rec2], schedule, storage) #generate C program
```

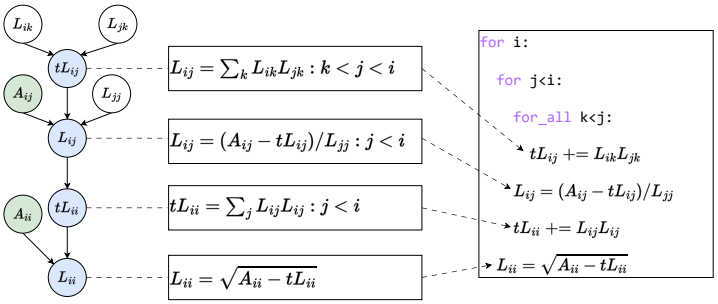

Fig. 5. A dependency graph orders the placement of recurrences in the abstract loops of the ijk Cholesky. Green nodes are inputs, blue nodes are left-hand sides of fragments, white nodes are outputs that are also dependencies, implicitly calculated by blue nodes in previous iterations.

5 RECURRENCE DEPENDENCIES

In recurrence equations, there are dependencies between computed values. For example, in the prefix sum $S_i = A_i + S_{i-1}$, the ith element depends on all previously computed elements. The values, therefore, cannot be calculated in an arbitrary order. When a user provides a loop ordering, it imposes a constraint on the order in which to calculate outputs. By 1) identifying these dependencies in the recurrence equations, 2) representing the dependencies in a dependency graph, and 3) reasoning about how loop ordering affects dependency management, we can construct the foundation of an algorithm for checking dependencies while lowering recurrences to imperative code.

5.1 Dependency Graphs

In a recurrence, a dependency of an element is any previous element in the recurrence sequence needed to calculate it. The elements calculated by a recurrence equation directly depend on all the elements accessed in the operands on the equation's right-hand side. As long as no such dependencies are inverted (meaning all values are calculated before their use) then a loop order correctly computes the results. In the first equation of the Cholesky example, $L_{ij} = (A_{ij} - \sum_k L_{ik}L_{jk})/L_{jj}$, the result L_{ij} depends on A_{ij}, L_{ik}, L_{jk}, and L_{jj}. We model these dependencies with a directed acyclic graph in which each tensorAccess expression is a node and each dependency is an edge. Figure 5 shows an example dependency graph for the Cholesky factorization.

Before identifying dependencies in a recurrence system, recurrences are broken into smaller, fragment recurrences. A fragment is a recurrence in which a summation expression (including the inner expression) is placed in a separate recurrence from the expression encapsulating the full summation. In our Cholesky example, the two input recurrences are thus expanded to the four recurrences shown in the center column of Figure 5. A fragment for a summation calculates a partial value of the final output, and we name the partial value by taking the final output's tensorAccess name and preceding it with a t (for temporary). The subsequent code generation stage places the fragment recurrences into the loop instead of the original recurrences. This allows subexpressions from different input recurrences to be interleaved within a single fused loop nest.

For each fragment recurrence, we draw an edge in the dependency graph from each operand to the result. Figure 5 shows how the dependency graph (left) is used to order the fragment recurrences (center), which are then placed into the imperative loop nests of the RIN (right).

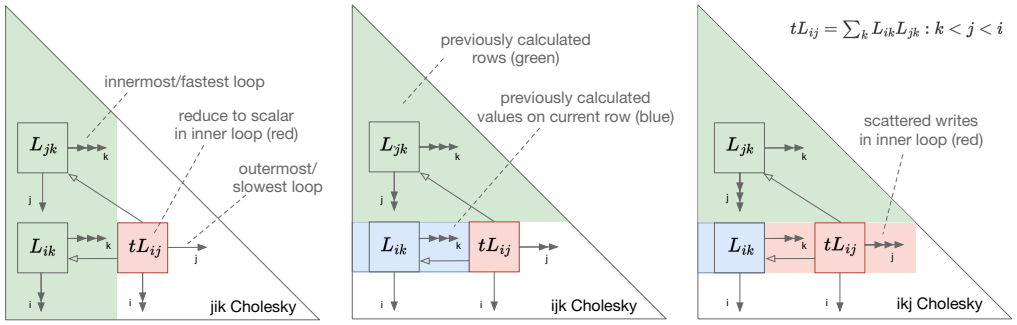

Fig. 6. Comparing dataflow and dependencies of jik, ijk, and ikj Cholesky decompositions. Solid arrows illustrate the direction a tensorAccess moves along as the corresponding loop iteration variable increases. Calculating tL_{ij} requires that L_{ik} and L_{jk} are calculated. Arrows with an empty head denote dependencies. For a jik program, when calculating tL_{ij}, all previous columns must already be calculated (green). For an ijk program, all previous rows (green) and all previous entries in the same row (blue) must already be calculated. In ikj Cholesky, the inner loop (triple head arrows) notably iterates over both rows and columns while scattering writes (red) across a row. tL_{ij} is a partially computed value of L_{ij}, which then becomes read as L_{ik} and L_{jk}.

5.2 Dependencies and Loop Ordering

Any code generation algorithm for recurrences must manage dependencies so that the final code does not violate them. The treatment of dependencies depends both on the recurrence equations and on the user-specified loop ordering. For some recurrences, certain loop orderings cannot be made to respect dependencies and are therefore illegal. And in general, different loop orders will have different loop-carried dependencies.

Data structure decisions, however, do not affect the loop dependencies we describe here. It is true that accessing a data structure in the wrong order carries a high overhead, but it does not affect correctness. The reason is that both dense arrays and sparse data structure can be randomly accessed instead of traversed. But where a dense array can be accessed in constant time, a sparse data structure may require a search. For example, randomly accessing an row-major CSR sparse data structure in column-major order requires an $O(\log n)$ search if it is ordered and an $O(n)$ search if it is unordered. Our compiler techniques can generate such searches and can therefore generate code for any RIN loop nest that passes the dependency checks. Alternatively, a compiler can report an error before generating C code if a data structure is traversed in the wrong direction, which is a simple check that is orthogonal to the dependencies we treat in this paper.

The effect of different loop orders on dependency handling is exemplified by comparing three different loop orders of the running Cholesky example. Figure 6 visualizes the dependencies of the (fragment) recurrence equation $tL_{ij} = \sum_k L_{ik}L_{jk}$ in three different loop orders of Cholesky decomposition. Here, tL_{ij} is a partially computed value of L_{ij}, which then becomes accessed as L_{ik} and L_{jk} in subsequent iterations. The left figure shows a jik Cholesky in which the output L is calculated column-by-column. The dependency graph shows that calculating tL_{ij} requires having already calculated the two dependencies L_{ik} and L_{jk}. In the jik loop order, results are calculated one column at a time, such that the jth iteration of the outer loop computes the jth column. And the Cholesky constraints state that $k < j$, so we know that column k, which includes both dependencies, is already calculated by the time we get to calculating column j. In the visual language of the figure, the fact that both dependencies are computed when we get to L_{ij} is shown by them being situated in the green area of previously computed columns.

The middle figure shows an ijk Cholesky in which the output L is calculated one row at a time, such that the ith iteration of the outer loop computes the ith row. The dependencies of tL_{ij} are still L_{ik} and L_{jk}, which are in rows i and j respectively. When calculating tL_{ij} in row i, the constraint $j < i$ only guarantees that L_{jk}, in row j, is already calculated. The other dependency, L_{ik}, is not satisfied by this constraint as it lies in the row that we are currently trying to calculate. However, we can satisfy the dependency L_{ik} by assuming that we calculate the elements of row i in increasing order. Thus, because $k < j$, we know that the kth element in the current row has been calculated by the time we are calculating the jth element.

The right figure shows an ikj Cholesky in which L is computed row-by-row. Like in jik and ijk Cholesky, results depend on both values from previously computed rows and previously computed elements on the same row. However, because the j loop, which indexes into the result, is nested inside the k loop, the ikj Cholesky scatters results down the part of the i row after the current value of k.

The dependencies have implications for parallelization. For example, they imply that the outer loop j in the jik program has a loop carried dependency. That is, when calculating an entry in column j, we must have already calculated all previous columns until column j. Meanwhile, both the outer loop i and the middle loop j in the ijk program have a loop carried dependency. That is, when calculating an entry in row i we must have calculated all previous rows up to i. Additionally, when calculating the jth entry of tL_{ij} within a row, we must have already calculated all previous entries in the row until the jth one. A loop with a loop carried dependency cannot in general be trivially parallelized.[1] In ijk Cholesky, there is a loop carried dependency in the outer and middle loop, so only the inner loop can be parallelized. In jik Cholesky, only the outer loop contains a loop carried dependency, meaning both the middle and inner loop can be parallelized.

The lowering algorithm from recurrence equations to RIN, which we describe in Section 6, manages recurrence dependencies with inductive assumptions that formalize the above intuition. The lowering algorithm incrementally constructs a proof by induction during lowering by making assumptions about what is computed at each lowering step. For example, for jik Cholesky, the lowering algorithm makes an assumption that a statement over c is true for all $c < j$, and then uses this assumption to prove that the same statement is true for j. If both this inductive hypothesis and the base case are true, then the statement is always true. Similarly, if we can calculate the jth column of L by assuming that all columns $c < j$ are already calculated, then we can always satisfy the dependencies necessary to calculate every entry in L.

6 RECURRENCE EQUATION LOWERING

Recurrence equation lowering lowers recurrences into the imperative recurrence index notation (RIN) IR. This IR represents a program in which the recurrence results are iteratively calculated in an abstract loop nest where physical data structures have not yet been specified. The lowering algorithm generates the user-defined loop order, thus simplifying dependency checking and avoiding the complexity of loop reordering in the face of fragmented loops such as those of the jki Cholesky in Figure 1. Recurrence lowering is based on three principles that form the steps of the algorithm:

(1) Recurrences can be decomposed into smaller fragments that put summation statements in the their own recurrence.
(2) RIN can then be generated by placing the fragment recurrences into a loop nest one at a time, such that dependencies between fragments are not violated. Dependencies can be modeled with a dependency graph that encodes the order in which to place statements.

[1] Exceptions include tree-based parallelization schemes for reductions and prefix sums.

```
#1) generates minimal fragements 2) sorts fragments 3) places each fragment into RIN
     program, making inductive assumptions when necessary
def lower(input_recurrencs, loop_ordering):
    fragment_recurrences = generateFragments(input_recurrences)
    sorted_fragments = topologicalSort(fragment_recurrences)
    assumes rin_program = {},{} #no inductive assumptions yet, RIN initially empty

    for statement in sorted_fragments:
        tryPlacing(statement, rin_program)

        #try making inductive assumption, if that lets us place statement
        while statement.notPlaced() and canMakeAssumption(loop_ordering, assumes):
            assumes = makeAssumption(loop_ordering, assumes)
            tryPlacing(statement, rin_program, assumes)

        if statement.notPlaced():
            return fail loop_ordering is impossible
    return rin_program

#greedily place statement into first location where dependencies are satistfied
def tryPlacing(statement, program, assumptions):
    #function defined by Equation 2
    location = PlacementLocation(statement, statement.dependencies, rin_program)
    if location:
        location.place(statement) #inserts readiness statement
```

Fig. 7. Pseudocode for RIN generation. Not shown is 1. the process for generating empty loop headers at a potential location for placing a fragment and 2. the check that index variables used in a fragment are in scope at the potential location. Both of these happen implicitly when iterating over valid locations in the program

(3) When determining where to place a statement, what results have already been calculated can be inductively determined at any location and any iteration of the nested loop. The inductive assumptions are necessary when a statement depends on results computed by the same statement in earlier loop iterations.

The lowering algorithm follows these steps to generate an RIN program and terminates once all fragments have been placed. It reports an error if the loop order violates a dependency. Pseudocode for the placement algorithm is shown in Figure 7 and described in the following sections.

6.1 Recurrence Index Notation

The recurrence index notation (RIN) consists of abstract loop nests that iterate over index variables to compute result values in scalar statements. The loops are abstract in that they iterate over and compute with abstract tensors, meaning specific physical data structures have not yet been introduced. This abstraction impacts scalar compute statements and loop bounds. The scalar statements access the logical tensors through tensor accesses. And the loop bounds are given as set expressions that describe how to co-iterate over the relevant tensors. Moreover, the loop bounds contain constraints such as $i < j$ to express triangular iteration spaces. Like in prior work [Kjolstad et al. 2017], accesses to and iteration over physical data structures are introduced only when generating low-level imperative code (Section 7).

Two examples of RIN loop nests were given in Figure 1 and the grammar is shown in Figure 8. The if statements and while loops often found in sparse codes are absent in RIN; all loops are for or forall loops (in which a for loop has loop-carried dependencies, and a forall does not). Thus, a recurrence program that requests dense data structures and an otherwise equivalent recurrence program that requests sparse data structures will have the exact same RIN. Finally, the set expressions and ranges of the loop bounds are left implicit and are inferred from tensors and statements in bodies of the loops during low-level code generation. For example, the range of a loop is inferred from the tensor

assign ::= tensorAccess '=' expr
| tensorAccess '+=' expr

expr ::= expr '+' expr
| expr expr
| '-' expr
| ...

tensorAccess ::= $\text{tensor}^{\text{index}^*}_{\text{index}^+}$

index ::= iVar
| iVar '+' constantInt
| constantInt

stmt ::= assign
| for
| forall

for ::= 'for' loopHeader ':' stmt+

forall ::= 'forall' loopHeader ':' stmt+

loopHeader ::= iVar '<' index ':'
| iVar '>' index ':'
| index '<' iVar '<' index ':'
| iVar ':'

RIN ::= stmt+

Fig. 8. Grammar for Recurrence Index Notation

dimensions it indexes. The set expression for iterating over a statement containing a multiplication expression is the intersection of its operands' iteration spaces [Kjolstad et al. 2017].

6.2 Generate Fragment Equations

Recurrence fragments are generated by traversing the recurrence bottom-up and placing each summation operation in its own recurrence. Because fragment recurrences do not compute final results, they instead store their partial sums to temporaries. To disambiguate a temporary from a result, the temporary is denoted by the corresponding tensor name L being preceded by the letter t (e.g., tL). The temporary is a partial computation of a value that will eventually reside in the output tensor. The Cholesky decomposition, for example, will decompose each of its two input equations into two fragments, since each equation contains one summation. The four resulting recurrences for the Cholesky decompositions are shown in Figure 5.

6.3 Determine Placement Order

The placement algorithm only places a fragment into the loop at a location where the fragment's dependencies are already calculated. There is thus a limited set of correct orderings in which fragments can be placed into the imperative loop, and these orderings can be inferred from the directed acyclic dependency graph over all fragments' tensorAccesses. This graph is generated right before the placement stage.

In a dependency graph, a directed edge $A \rightarrow B$ (meaning tensor access B depends on the values from tensor access A) exists if there exists a recurrence equation with B as the result and A as an operand. As described in Section 5.1, the tensor accesses across all fragments form a directed acyclic graph, in which each tensor access is a node and each dependency is a directed edge.

Placement order is then determined by a topological sort of the fragments. For example, in an ijk Cholesky decomposition, the first placed equation is $tL_{ij} = \sum_k L_{ik}L_{jk}$ because of its position in the dependency graph in Figure 5; of the four nodes referring to the fragment results, it is the only one in the dependency graph not dependent on any other. tL_{ii}, on the other hand, cannot be placed first. As shown in Figure 5, it depends on L_{ij}, which itself is dependent on tL_{ij}.

6.4 Greedily Place Equations into an RIN Loop Nest

The placement algorithm starts with an empty loop nest created from the recurrence's index variables and a user-specified loop order. The algorithm proceeds by iterating through the dependency graph in topological order and placing each fragment into a location within the RIN.

As fragments are placed into the loop, the compiler keeps track of what outputs are already computed at each location in the loop via internal state. This state is a table that maps program locations to outputs, indicating that a certain subsection of the output can be considered computed at this location in the RIN. If a compute statement fully calculating an output is present at a certain

location, the compiler tracks that this output is ready directly after this location. This state is modified whenever fragments are placed and whenever inductive assumptions are made. Inductive assumptions preemptively enforce that an output will be calculated by a certain program location, even if the corresponding compute statements have not yet been inserted into the RIN. The compiler tracks these assumptions with the same mechanism, and modifies that table accordingly. While this state is not embedded within RIN, we annotate Figure 9 with readiness markers that illustrate readiness information internal to the compiler. In the Cholesky example, for instance, a marker illustrating that L_{ii} is considered calculated after the marker's location would be $ready\ L(i,i)$.

Let R be a collection of locations in a RIN loop nest, ordered according to program order. At any particular location $p \in R$, the set of outputs and temporaries that have already been computed at p can be calculated with the following equation. Here, $ready$ refers to the internal table the compiler keeps about which output is considered calculated at location loc. Thus, $ready(loc)$ records the set of all outputs considered computed by the RIN at line loc.

$$readyAtLocation(p) = \{ \bigcup_{loc \in R}^{p} ready(loc) \} \tag{1}$$

Therefore, the set of already calculated outputs at point p is the set union of what is considered ready at each program location until p, inclusively. In other words, it is determined by collecting all readiness info from an in-order traversal over RIN's AST, terminating at location p.

For a statement s with a set of dependencies D, the placement algorithm chooses a program point l in the set of program points R in the RIN in which to place s. Assuming program points in R are ordered according to the program order, this location can be determined using Equation 1 calculating $readyAtLocation(p)$.

$$placementLocation(s, D, R) = \min_{\forall p \in R} p \quad s.t. \;\; D \subset readyAtLocation(p) \tag{2}$$

In other words, each statement is greedily placed into the first location in the RIN loop nest where the statement's dependencies are computed. When a statement is placed in the loop nest, the compiler internally records that the statement's result has been computed at the statement program point following the statement.

We show several steps of the placement process for ijk Cholesky in Figure 9. For each step, green readiness markers show the lowering algorithm's record of what values have been computed at each program point. The placement algorithm terminates when all fragments have been placed, in which case the IR is fully formed, or when it it not possible to place a statement anywhere. In the latter case, the given loop-ordering is illegal and the compiler returns an error.

A statement may depend on a value in the output calculated by itself in an earlier loop iteration. This is a direct result of having dependencies amongst outputs within a recurrence. Due to this cyclic dependency, placing this statement into the loop is not possible without a prescient guarantee about what outputs have been calculated for each iteration and location in the loop. These guarantees are made in the form of inductive assumptions. When these assumptions are made, additional readiness info is recorded by the compiler. This is shown in stage b of Figure 9, in which green readiness markers encoding these guarantees are placed into the loop nest.

An assumption guarantees that for a given loop iteration variable i, the subsequence of output elements that can be considered as already calculated at the start of iteration i is a function of i. In an ijk Cholesky decomposition, this could mean that at the beginning of iteration i of the outer loop, all rows of the output L up to row i (exclusively) are already calculated. Each assumption can be thought of as a strong inductive hypothesis. In this hypothesis, let A_I be a tensor access from

```
forall i
  forall j<i
    forall k<j
```

(a) No Cholesky statements can be placed unless inductive assumptions have been made

```
for i
  ready L(:i,:)
  for j<i
    ready L(i,:j)
    forall k<j
      tL_ij += L_ik L_jk
    ready tL(i,j)
```

(b) Inductive assumptions made over i and j, allowing placement of $tL(i, j)$

```
for i
  ready L(:i,:)
  for j<i
    ready L(i,:j)
    forall k<j
      tL_ij += L_ik L_jk
    ready tL(i,j)
    L_ij = (A_ij - tL_ij)/L_jj
    ready L(i,j)
```

(c) Place $L(i, j)$

```
for i
  ready L(:i,:)
  for j<i
    ready L(i,:j)
    forall k<j
      tL_ij += L_ik L_jk
    ready tL(i,j)
    L_ij = (A_ij - tL_ij)/L_jj
    ready L(i,j)
    tL_ii += L_ij Lij
    ready tL(i,i)
  L_ii = sqrt(A_ii - tL_ii)
```

(d) Place remaining fragments

Fig. 9. Steps of the placement algorithm for generating RIN of an ijk Cholesky decomposition. Green text show readiness markers that are not part of IR but internal to the compiler

a fragment's left-hand side, in which A is a tensor and I is an ordered list of indices. Let v be an index variable such that $v \in I$. Let J be another list of index variables, and let i represent the *index of an index variable* in J or I. Elements in J or I can use an indexing colon, where a single colon : represents all potential indices in a dimension, and $: v$ represents all potential indices less than v.

If an inductive assumption is made over v, then at the beginning of iteration v in the loop over v, all outputs encapsulated by A_J have been computed. Here, J is defined as:

$$J_i = \begin{cases} :v & \text{if } I_i = v \\ I_i & \text{if an assumption already exists over a parent loop over } I_i \\ : & \text{otherwise} \end{cases}$$

The strong inductive hypothesis guarantees that at the beginning of loop iteration v, all slices of A until the vth slice are already computed. As with strong induction, this hypothesis can only become true if all statements computing values in the vth slice of A are placed within iteration v of this loop, such that the vth iteration fully calculates the vth slice. In RIN, this corresponds to converting the for loop over v to a forall loop. This conversion implies three things:

(1) The guarantee that all slices of A until slice v are calculated before iteration v of that loop.
(2) The restriction that the current loop over v must be the last loop over v that writes to slice v of A. Computations writing to this slice can now only exist within this loop over v, or the aforementioned guarantee will be broken, rendering the inductive hypothesis false.
(3) The introduction of a loop-carried dependency in this loop with respect to A, which will affect whether the loop can be parallelized

For example, as shown in Figure 6, in an ijk Cholesky decomposition computing L_{ij}, an assumption over i would indicate that all rows of L up to row i (shown in green) are already computed at the start of iteration i in the loop over i. Using the formula for J_i, we now consider $L(: i\ ,\ :)$ as ready

in this loop. The algorithm then records this readiness info internally, which we visualize in stage b of Figure 9 by placing a readiness marker $ready\ L(:i\ ,\ :)$ in the first location within the loop over i. For the current fragment we are trying to place tL_{ij} += $L_{ik}L_{jk}$, the guarantee that all rows up to i are calculated satisfies the dependency over L_{jk}, an element in row j. The fragment's constraint guarantees that $j < i$, so L_{jk} can now be considered as already computed. The algorithm is then forced to place all statements calculating row i of L within this loop iteration i. This is shown by the RIN in the top row of Figure 9, which illustrates what happens during this assumption.

Figure 9 also illustrates how after an initial assumption is made about i in the ijk Cholesky RIN, an additional, nested assumption is made over the middle loop j. The formula for J_i shows that this new assumption inherits the outer assumption over i. The algorithm then records the new readiness information corresponding to this assumption, illustrated by placing the marker $ready\ L(i\ ,\ :j)$ at the beginning of iteration j for loop j. The readiness marker means that at the beginning of iteration j in the loop over j, all elements in row i until the jth element are computed. This is visualized by the middle diagram in Figure 6. For the fragment we are trying to place, tL_{ij} += $L_{ik}L_{jk}$, the constraint guarantees that $k < j$, so the dependency L_{ik} is now considered computed at this program point. As the other dependency L_{jk} was satisfied by the first assumption over i, both dependencies are now satisfied in the loop over j, allowing us to place the statement as shown in stage b of Figure 9.

A similar sequence of events will occur for all other loop orderings. In a jik Cholesky ordering where the outer loop iterates over j, an assumption over j would guarantee that at the beginning of iteration j in loop j, all *columns* of output L up to column j are already computed. This assumption gives a guarantee about columns because j indexes the column dimension in L_{ij}.

If the algorithm succeeds, then the inductive assumption is justified. We can make assumptions over index variables in the order given by the loop ordering, until the number of assumptions made matches the number of dimensions of the highest dimensional output. In this case, we cannot make more than the two assumptions over loops over i and j, as L is only two dimensional. Finally, if the input recurrences contain no recurrence dependencies amongst outputs, then inductive assumptions are not required. Tensor algebra expressions (e.g., matrix multiply), for example, lack dependencies, as computation of outputs is embarrassingly parallel, so the placement algorithm will never make an inductive assumption.

6.5 Variable Substitutions

Intuitively, a recurrence statement that uses an index variable i should be placed in a loop over i so that the variable is in scope. For recurrences, however, this is actually too stringent a restriction that at best will suppress loop fusion and at worst will prevent correct code generation. In many cases, two index variables with different names (e.g. i, j) may be logically interchangeable, meaning a recurrence statement that uses i can be rewritten in terms of j, allowing it to be placed within a loop over j. For example, assume the user defines two recurrences, an identity recurrence $X_i = Y_i : i < N$ defined over i and another identity recurrence $Y_j = Z_j : j < N$ defined over j. For a given loop ordering of the single variable i, a naive greedy placement algorithm would generate a loop over i, note the 2nd fragment is defined solely over j, and naively deduce this fragment defined over j cannot be placed in the i loop. However, the 2nd recurrence be safely rewritten as $Y_i = Z_i : i < N$, as i's iteration space of $[0, N)$ in the current RIN loop is equivalent to j's iteration space in the original recurrence. In this case the two variables are considered isomorphic, and the 2nd recurrence should be rewritten so it can be placed within the the i loop.

A recurrence lowering algorithm should thus determine if and when variables can be interchanged. Our algorithm can substitute index variables in fragments' tensorAccesses if the new variable v is *isomorphic* to an existing variable u. For a particular fragment, two variables u and v

```
for i:
  ready S(:i)
  forall k<i:
    tS_i += S_k
  ready tS(i)
  S_i = sqrt(tS_i)
```

Fig. 10. RIN for *ik* ordering

```
for k:
  ready S(:k)
  ready tS(:k+1)
  S_k = sqrt(tS_k)
  ready S(k)
  forall i>k:
    tS_i += S_k
```

Fig. 11. RIN for *ki* ordering

are isomorphic at a program location p if v's lower and upper bound at location p are the same as u's upper and lower bounds in the original fragment recurrence, as determined by the constraints. To determine when variables should be interchanged, let v be the ith variable in the loop ordering, u be the jth variable, and F be the fragment (using u) we are attempting to place in RIN location p, where scope(p) is the list of all index variables in scope at p.

$$\begin{aligned} S_u = \{\text{iteration space of } u \text{ in fragment}\}\ S_v = \{\text{iteration space of } v \text{ in scope}(p)\} \\ u \cong v \iff inf(S_u) = inf(S_v) \wedge sup(S_u) = sup(S_v) \\ (i < j) \wedge (u \cong v) \rightarrow \text{replace } u \text{ with } v \text{ in } F \end{aligned} \tag{3}$$

Therefore, variable u only need be replaced with isomorphic variable v if v comes before u in the given loop ordering. This allows fragments using u to be placed directly into a loop over v if it comes before a loop over u or if a loop over u does not yet exist. These substitutions *normalize* a set of fragments with respect to a schedule and can be done during greedy placement itself. Alternatively, this normalization can be executed once after fragment generation and before the dependency graph is generated, eliminating the need to rewrite fragments while placing them.

In the ijk Cholesky decomposition, this can occur when placing the fragment recurrence for $L_{jj} = \sqrt{tL_{jj}}$. Because the fragment calculates a diagonal value where $j = i$, and because the outer loop iterates over i (the first loop in the ordering), it makes sense to redefine L_{jj} in terms of i. This allows the equivalent fragment $L_{ii} = \sqrt{tL_{ii}}$ to be placed in the outer loop over i.

This substitution is useful in when a summation variable comes before a non-summation variable in the given loop ordering. Consider the recurrence $S_i = sqrt(\sum_k S_k)$, where k is a summation variable and $k < i$. The code in Listings 10 and 11 show RIN for both ik and ki loop orderings.

In the left ik listing, an inductive assumption is made over i. Once the loop over k completes, tS_i is considered fully computed, and the algorithm can then place the statement $S_i = \sqrt{tS_i}$.

In the right ki listing, an inductive assumption is made over k. Because $i > k$, for outer loop iteration k, the inner loop over i writes to all tS_i where $i > k$. The inductive assumption over k implies that iteration k will be the last iteration in which the inner loop writes to $tS_{i=k+1}$, implying tS_{k+1} is ready once the inner loop completes. This is equivalent to saying tS_k is ready at the beginning of the k loop, and we can thus immediately place $S_k = \sqrt{tS_k}$ there. To handle this, the algorithm makes the substitution $i \rightarrow k$ to rewrite the fragment into $S_k = \sqrt{tS_k}$.

This above case happens in the kji, kij, and ikj Cholesky decompositions. The transformation shown in the above listings is also included as its own transformation in the Symbolic Fractal Analysis [Menon and Pingali 2004] compiler framework, which has been used to change loop orderings of the Cholesky decomposition and triangular solve. However, the transformation requires an existing imperative Cholesky code as input. With variable substitutions and RIN, a hardcoded transformation is not necessary, as the above transformation happens automatically.

7 C CODE GENERATION

Our RECUMA recurrence compiler is written in Python, and the toolchain includes a Python based front-end for defining the recurrence equations, constraints, and a schedule. The aforementioned fragment generation, DAG generation, and greedy lowering occur in this pipeline, ending in the generation of the RIN intermediate representation. The RIN is then lowered to C code. This final phase generates code that iterates over dense and sparse data structures.

Our RIN to C code generation phase uses the code generation ideas from the TACO compiler. TACO is a tensor algebra compiler that compiles tensor algebra expressions, where tensors' data structures are separately specified, into loops that co-iterate over the different (dense and sparse) data structures to compute the result.

We discussed the differences between recurrence equations, as supported by RECUMA, and tensor index notation, as supported by TACO, in Section 4. These differences—recurrences, constraints, multiple equations, and timestepping variables—primarily affect the algorithm that generates RIN from recurrence equations, which we described in preceding sections. The primary exception is the constraints, which result in triangular loop bounds in the RIN. The C code generation phase in RECUMA can therefore use the same general approach as TACO, with a straight-forward modification to support triangular loop bounds. We therefore refer the reader to the TACO body of work for how to compile high-level loops over abstract tensors to loops over concrete sparse and dense data structures [Kjølstad 2020].

However, we discuss various optimizations that RECUMA performs that are particular to recurrences in Section 8. Some of these optimizations are performed during C code generation. Moreover, in RECUMA, for any given recurrence and schedule, the choice of data structure does not affect the correctness of the generated code, even if the the algorithm involves data access patterns for which the data-structure is a poor fit. RECUMA guarantees that the generated C-code is always correct, and will do so at the cost of the generated code's performance when necessary. If the loop order requires a tensor to be accessed in a different way from the way it is stored in memory (e.g., accessing a CSR matrix in column-major order), then RECUMA will insert a $O(\log n)$ binary search within the CSR data structure.

8 OPTIMIZATIONS

RECUMA supports a mix of general program optimizations and domain-specific optimizations targeted towards families of similar recurrences. General techniques like loop fusion and loop parallelization are automatic byproducts of the lowering process from recurrence equations to recurrence index notation. Moreover, the marching pointers and mask optimizations can also be specified by the user as part of their schedule. Other optimizations, such as tiling and wavefront parallelism, are not currently supported by RECUMA. We believe, however, that they can be incorporated in future work as loop transformations on the recurrence index notation.

8.1 Parallelization

The distinction between a for loop and forall loop in the recurrence index notation informs auto-parallelization. If a loop is a forall and operates solely over dense data structures, the corresponding C loop is parallelized with an OpenMP pragma. To illustrate, we use the Viterbi algorithm [Viterbi 1967], a dynamic program for determining the most likely path taken in a hidden Markov process. It is often also treated as a graph problem, requiring sparse data structures, when the set of states are sparsely connected with edges that represent transition probabilities. The algorithm calculates the most likely sequence of hidden states encountered given a sequence of observations. The associated recurrence utilizes three matrices. Given a transition probability matrix A and an

emission probability matrix B, the output matrix V can be calculated with the equation below on the right. In this case, using a jki or jik loop ordering will result in the outer loop over j being a for loop, and the inner loops being forall. This is shown in the following RIN, which implements a jik loop ordering of the Viterbi recurrence:

```
for j:
  forall i:
    forall k:
      V_ij = max(V_{k,j-1} A_ki B_ij V_ij)
```

$$V_{ij} = \max_k V_{k,j-1} A_{ki} B_{ij} \tag{4}$$

The index variable j is expressible as a timestep variable, but is here used as a normal index variable because knowing intermediate states is useful in Viterbi path problems. Performance results of parallelization are shown in the evaluation. Enabling parallelization for the loop over i can be done with the following command: `schedule["omp"] ="i"`.

8.2 Loop Fusion

As described in Section 6, the lowering algorithm greedily places each compute statement into the *first* location of a RIN program where the statement's dependencies are satisfied. This implies that the algorithm will always try to fuse the statement into an existing loop before attempting to place it in a subsequent loop. In other words, loop fusion happens automatically in this framework. Furthermore, the algorithm's ability to handle multiple recurrences implies that statements from separately defined recurrences will, when possible, be placed within a single fused loop. This is highly beneficial for performance, as a RECUMA program will never incur the cost of looping over an array twice when one loop will suffice. This is especially important for memory-bound kernels in which the costs of accessing large arrays dominates a kernel's execution time.

To illustrate, we show how two sparse triangular solves will be fused together automatically. Given a lower-triangular matrix L and an input vector B, the triangular solve will attempt to solve $Lx = b$ with the following recurrence: $X_i = (B_i - \sum_j L_{ij} X_j)/L_{ii} \ : \ j < i$.

It is common to have to solve the same matrix system L for multiple right-hand side B vectors. In a situation that requires performing solves for two such B vectors, the user can provide the following recurrence equations that solve for X and Y:

$$X_i = \left(B1_i - \sum_j L_{ij} X_j\right)/L_{ii} \ : j < i \qquad Y_i = \left(B2_i - \sum_j L_{ij} Y_j\right)/L_{ii} \ : j < i$$

Usually, each triangular solve would require a separate doubly-nested loop over i and j. However, if the user provides both equations and an ij loop-ordering, RECUMA will generate a single ij doubly-nested loop that calculates both X and Y. This performance benefits of using a fused kernel in this example are demonstrated with experiments discussed in Section 9.4.

Two recurrences need not be identical (like in the case above) for fusion to occur; fusion can occur between completely different recurrences. To illustrate, the Cholesky decomposition $A = LL^t$, is commonly followed by two triangular solves to solve for x in the equation $Ax = LL^t x = b$. The first triangular solve can be fused into the outer Cholesky loop. To generate such a fused kernel for an ijk loop nest, the user need only provide the recurrence equations for both the triangular solve and the Cholesky decomposition. The placement algorithm will then generate a loop nest in which the triangular solve to calculate X_i occurs in the outer-loop over i from the ijk Cholesky loop nest.

8.3 Masks

Masks are used in sparse problems to represent a tensor's sparsity pattern. In a breadth-first search, which is expressible as a recurrence, masks can be used to denote the subset of nodes in a graph that are of interest to the user, such that the BFS now only needs to iterate over the relevant data.

In various matrix solvers, including the triangular solve and Cholesky, LU, and QR decompositions, they are employed similarly, such that optimized kernels need only iterate over the entries of the matrix that are known to be non-zero. The process of precalculating this nonzero pattern is commonly referred to as symbolic factorization and is used as a preprocessing step to the numerical factorization in which the nonzero values are calculated.

Generation of the mask is not within the scope of this work, as the underlying process is not necessarily based on solving recurrence. The user hence provides a pre-calculated mask to RECUMA, which is often justified due to many sparse problems consisting of repeatedly solving different matrix problems with the same underlying sparsity pattern. Depending on the sparsity pattern, generating and iterating over masks does not always improves performance. It is thus an optimization parameter for sparse recurrences, and we briefly describe the significance of masks for relevant problems.

In a triangular solve, symbolic analysis can be done with a data structure known as an elimination graph [Gilbert 1994]. In this process, given a sparse b-vector, the nonzero elements of b are used as roots to a depth-first search of the sparse matrix L, in which a nonzero at location L_{ij} represents an edge between nodes i and j. The DFS emits a topologically sorted mask denoting the locations of nonzeroes in X, even though X's values have not yet been calculated. For a Cholesky decomposition, a data structure called the elimination tree [Liu 1986] can generate a mask of L in linear time.

The user can optionally provide a mask, along with an indication of whether the mask is CSC or CSR. When possible, the compiler will then lower any appropriate loops in the RIN to iterate over the mask, thus avoiding needles computations whose output is zero. In cases, where the loops need to iterate over both rows and columns, it may be useful to provide both CSC and CSR masks, which is possible. The primitive for doing so is shown below:

```
storage["L"] = Storage([Dense(0), Compressed(1)]) #L stored in CSR
storage["L"].addMask(SparseMask([Dense(0), Compressed(1)])  #CSR Mask
storage["L"].addMask(SparseMask([Dense(1), Compressed(0)])  #CSC Mask
```

8.4 Marching Pointers

When using a sparse data structure best suited to efficiently iterating over a row (e.g., CSR), it is often useful to use an auxiliary data structure that also lets us efficiently iterate over a column (and vice versa). This is useful in cases when the loop ordering and recurrence require efficient iteration over both a row and a column. Figure 6 shows that for ikj Cholesky, the inner loop iterations over j (shown with triple head arrows) requires we iterate over a column when accessing L_{jk}, so it is best to use a CSC data structure. However, because the outer loop is over i, we wish to build the output L_{ij} row-by-row, in which the inner loop also iterates over the ith row. It thus is useful to be able to efficiently iterate over the ith row, while being able to efficiently iterate over columns. This can be done with expensive binary-searches, but more efficiently implemented by keeping a 1D vector M of *marching pointers*, such that for outer iteration i and inner loop iteration j, $M(j)$ contains an offset to output element L_{ij} in the CSC matrix L, allowing us to easily iterate over row i as if row i was row major. At the end of outer loop iteration i, each offset is incremented - these increments only happen in the outer loop, so they are infrequent and do not negatively affect performance. Using marching pointers is easily accomplished with the primitive:

```
storage["L"] = Storage([Dense(1), Compressed(0)]) #L stored in CSR
storage["L"].addMarchers(0,i) #marchers along dimension 0 (rows), updated on ith iteration
```

9 EVALUATION

We evaluate the performance of recurrence implementations generated by RECUMA against hand-optimized implementations in popular libraries (Section 9.1). Furthermore, to demonstrate the

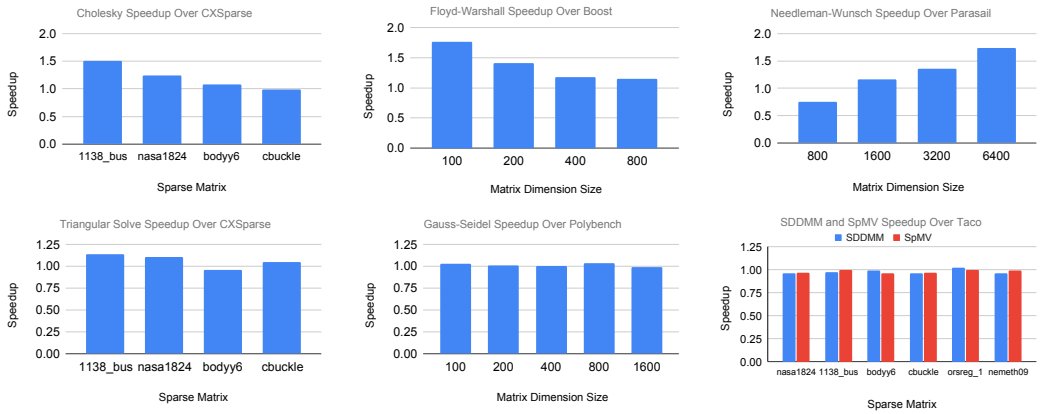

Fig. 12. Speedup of RECUMA-generated kernels over libraries. All times are averaged over 100 trials, except SpMv (500 trials), and Gauss Seidel (1000 trials for N=[100,200,400]). Sparse matrices for ji tri solve and ikj Cholesky were chosen via stratified random sampling of SPD matrices from SuiteSparse from the following intervals over #nonzeros [1-5K], [5K-50K], [50K-500K], and [500K-1M]. SDDMM and SpMV matrices include the first four SPD systems used in tri. solve and Cholesky and two additional random non-SPD systems.

importance of a system that supports different data formats and optimizations, we also also perform experiments that demonstrate the performance implications of data formats (Section 9.2), different loop orderings (Section 9.3), fusion (Section 9.4), and parallelism (Section 9.5).

9.1 Comparison with Existing Libraries

We evaluate the performance of RECUMA-generated implementations of sparse Cholesky decomposition, sparse triangular solve, sparse SDDMM, sparse SpMV, dense Floyd-Warshall, dense Needleman-Wunsch, and dense Gauss-Seidel. SDDMM and SpMv are sparse tensor algebra routines and are thus compared TACO-generated implementations. The remaining kernels are compared against popular libraries. Unless noted, all performance metrics were aggregated over 100 trials and conducted on a 40-core Intel Xeon CPU. Graphs showing speedup of RECUMA kernels over existing library kernels are shown in Figure 12.

Triangular Solve. We evaluate a RECUMA kernel for performing a ji triangular solve with CSC data structures against CXSparse, whose triangular solve uses the same schedule and data structures. Figure 12 shows that their performance is comparable for the matrices tested.

Cholesky Decomposition. We generate a RECUMA kernel of an ikj Cholesky using CSC data structures with marching pointers, and compare it against a CXSparse kernel using the same ordering and data structures (including marching pointers). Both implementations use a mask, with the main difference being the mask is calculated in the CXSparse Cholesky loop nest, while it is calculated before hand in RECUMA. However, mask calculation performance is almost inconsequential, as it is an $O(n^2)$ algorithm while Cholesky is $O(n^3)$. Speedup plots are shown in Figure 12.

Floyd-Warshall. We generate a Floyd-Warshall RECUMA kernel and benchmark it against a 2D implementation from the Boost graph library. The algorithm is inherently dense, so the connectivity and sparse structure of the underlying matrix is irrelevant to the runtime; only the number graph nodes N matters. The RECUMA vs Boost speedup graph is shown in Figure 12, in which

RECUMA code outperforms Boost. Boost uses unnecessary C++ features and abstractions that affect performance, while RECUMA's output is simple C code that operates on a C array.

Needleman-Wunsch. The Needleman-Wunsch algorithm is commonly used to align two strings of genomics data. The algorithm operates on a dense 2D matrix, where each cell depends on its left, upper, and upper-left neighbor. We evaluate code generated by RECUMA against the Parasail library. As shown in Figure 12, for small sizes Parasail outperforms RECUMA. As we increase the problem size, RECUMA consistently performs Parasail. During compile time, Parasail generates a large suite of kernels utilizing different vectorization strategies. We show Parasail's best performing kernel in Figure 12. We surmise that Parasail's vectorized code is not optimized for the target processor, otherwise Parasail would likely demonstrate superior performance.

Gauss-Seidel. We generate a dense Gauss-Seidel iterative solver, representing an in-place 5-pt stencil. We compare its performance to the Gauss-Seidel solver in the PolyBench benchmarking suite, modified slightly to perform the same computation as our RECUMA kernel. Speedup plots from Figure 12 show that performance of the RECUMA and PolyBench kernels are nearly identical.

Tensor Algebra. We evaluate two tensor algebra kernels: sparse matrix-vector multiply (SpMv) and sampled dense times dense matrix multiply (SDDMM). We evaluate C code generated by RECUMA against C code generated by TACO. The TACO kernels were generated without any additional TACO scheduling primitives. As shown in the speedup plots in Figure 12 RECUMA expectedly exhibits nearly equivalent performance to that of TACO for these two kernels.

9.2 Data Formats

The user specifies the format of the kernels' data. Choosing formats amenable to a program's data access patterns ensures good memory access locality and thus improves performance.

Many recurrences, like the Viterbi equation, are used in both dense and sparse settings. We evaluate RECUMA's ability to generate both dense and sparse variants by changing user specification of the data-structures. Figure 13 shows performance of RECUMA's generated Viterbi kernels when tested with both sparse and dense matrix formats while varying the sparsity of the data. As expected, a sparse format leads to significantly better performance as sparsity increases.

Standard libraries, on the other hand, are tied to one or only a few data formats. CXSparse, for example, provides handwritten sparse matrix solver kernels, but requires input data be in a CSC format. Users who have their input data in alternative format incur a performance penalty from converting their data to the expected form. In a situation where input data is given to the users in a CSR format, we show in Table 2 the cost of converting the matrix from CSR to CSC, the time to calculate CXSparse's CSC triangular solve, as well as the time to calculate a RECUMA CSR triangular solve, in which no format conversion is necessary. The conversion routines are optimized

Table 2. The ratio of the format conversion time against compute time can be significant. None of the kernels shown here use elimination graphs, which are not always useful. Each matrix's Cholesky factor L is used to do the tri. solve, as the original matrices are SPD but not lower triangular.

Triangular Solve and Format Conversion Runtimes (microseconds)					
Matrix	nonzeros	CXSparse-ji CSC tri solve	Eigen CSR->CSC conversion	RECUMA-ij CSR tri solve	RECUMA-ji CSC tri solve
1138_bus	4054	47.371	33.824	51.25	30.767
nasa1824	30280	201.878	114.519	181	141.614
bodyy6	134208	21306	418.2	20277.2	18753.9
cbuckle	676515	3291.39	1969.58	3635.19	2946.36

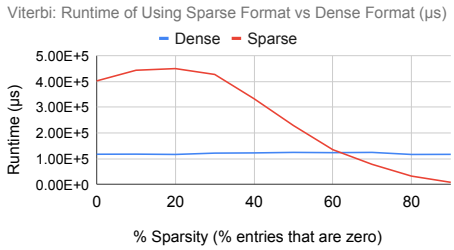

Fig. 13. N=1600 RECUMA Viterbi performance w/ random sparsity pattern

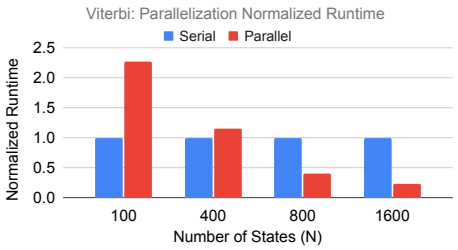

Fig. 14. Viterbi RECUMA normalized runtime of using parallel over serial loops on dense format

kernels from Eigen. For several problem sizes, these conversions consume a nontrivial portion of runtime when compared to the time it takes to run the triangular solve. RECUMA's generality avoids this problem entirely, as a custom kernel can be generated to fit the input data format.

9.3 Loop Ordering

The choice of loop ordering will influence a program's data access patterns and performance. RECUMA's flexibily allows a user to select a loop ordering amenable to a given input data format. The effect of loop schedules on sparse codes, in which data access patterns can be irregular, is harder to predict than in dense codes. It is thus useful to generate and try different variants. Table 3 compares ijk and ikj Cholesky kernels, which each perform better on different matrices. For the triangular solve, Table 2 shows that the ji schedule always performs best on our examples.

9.4 Loop Fusion

Loop fusion occurs automatically in RECUMA due to the greedy nature of the RIN generation algorithm. Fusing loops avoids the performance penalties of iterating over the same memory in different loops. To demonstrate the benefits of fusion, we use RECUMA to generate a fused kernel

Table 3. Comparison of the ikj Cholesky of RECUMA and the CXSparse Cholesky which also uses an ikj strategy. Matrices were chosen with stratified random sampling from SuiteSparse Matrix collection of SPD matrices to ensure proper spread of nonzero values

Cholesky Runtimes (microseconds)				
Matrix	nonzeros	CXSparse ikj perf	RECUMA ikj perf	RECUMA ijk perf
1138_bus	4054	1976.41	1310.05	3098.5
nasa_10824	39208	12582.9	10123.3	21415.1
bodyy6	134208	6415100	5916360	5765950
cbuckle	676515	441425	447210	448646

Table 4. Performance comparison of a fused RECUMA kernel to perform an ij sparse triangular solve for two right-handsides vs calling two separate RECUMA sparse triangular solve kernels.

Fused Triangular Solve vs Unfused Triangular Solves (microseconds)			
Matrix	Fused runtime	Unfused runtime	Speedup
1138_bus	55.85	107.4	1.92x
nasa_10824	302.83	585.32	1.93x
bodyy6	20539.6	40672.4	1.98x
cbuckle	3478.72	6675.27	1.92x

that computes two sparse ij triangular solves for the same matrix L but different B right-hand side vectors. We evaluate the fused performance against that of invoking a separate RECUMA kernel for each triangular solve. The triangular solve is often dominated by the cost of iterating over the matrix L. The fused kernel iterates over L once, so it performs *nearly* twice as fast as executing two individual triangular solves, as shown in Table 4.

9.5 Parallelization

Figure 14 displays performance of the RECUMA code when the compiler performs parallelization for the i loop in a jik Viterbi kernel. As expected, parallelization does not help for very small problems where overheads of spawning threads hurt performance but does improve performance as problem size increases.

10 RELATED WORK

Other domains have captured large program design spaces with compilers and DSLs for image processing [Ragan-Kelley et al. 2013], tensor algebra [Kjolstad et al. 2017], and sparse array programming [Henry et al. 2021]. The Symbolic Fractal Analysis [Menon and Pingali 2004] framework has been used to change certain loop orderings of a given dense Cholesky program, but it does not generate an imperative programs from declarative recurrences. The use of inductive proofs to generate code for different ijk forms of a blocked LU decomposition was explored in the FLAME project [Gunnels et al. 2001], though it is different in that it operates on recurrences over blocks to generate a program that calculates the decomposition with BLAS calls. Ortega [1988a,b] discussed the performance and design of different ijk forms of the Cholesky and LU decompositions. Sympiler [Cheshmi et al. 2017] is a compilation framework that can explore the optimization space of sparse Cholesky and triangular solves, generating native code specific to a sparse matrix's sparsity pattern. For certain loop orderings of matrix decompositions, the multifrontal method [Liu 1992] can compute the decomposition as a matrix assembly problem. The CHOLMOD library targets a supernodal [Ng and Peyton 1993] form of the sparse Cholesky decomposition, in which subproblems are computed with dense BLAS kernels. The supernodal algorithm uses a one-dimensional tiling of the Cholesky decomposition. We leave loop tiling and supernodal optimizations as future work.

Dynamic programming was formalized by Bellman [2010] to address problems in optimization. Gaussian elimination was presented as a type of dynamic program by Lehman [1960]. The process of converting a loop nest that recalculates overlapping subproblems into an asymptotically better dynamic program is addressed in the framework of simplifying reductions [Gautam and Rajopadhye 2006; Yang et al. 2021]. Huang [2008] showed that the Viterbi equation, CYK parsing, and certain graph problems are all similar dynamic programming algorithms over different semirings.

Parasail [Daily 2016] can explore the optimization spaces of alignment algorithms and Halide has support for optimizing in-place IIR filters [Chaurasia et al. 2015]. Parasail explores different vectorization strategies, while other sequence alignment implementations utilize wavefront parallelism. We leave vectorization of RECUMA kernels and wavefront loop skewing as future work. GraphBLAS [Kepner et al. 2016] is a general and performant library that draws upon dualities between linear algebra and graphs to implement graph algorithms based on recurrence equations. It does not, however, handle dense dynamic programs or solvers. Bellman's GAP [Sauthoff et al. 2013] is another DSL for certain dynamic programs, but like Parasail, is limited to alignment algorithms.

The polyhedral model [Feautrier 1991; Lamport 1974] applies to general programs and was originally applied to finite difference method recurrences [Karp et al. 1967]. The sparse polyhedral model extends the polyhedral model with some support for sparse data structures [Strout et al. 2018]. Our system provides a simple and effective alternative dependency management to that of the polyhedral model while generalizing across sparse data structures. Dyna [Eisner et al. 2004] is

a DSL that accepts general recurrences and solves them bottom-up. However, it does not generate a static loop nest. It computes outputs by placing every recursive subproblem (i.e., every output element) into a queue, thus requiring expensive runtime analysis to determine which subproblems to solve next. Dyna has similarities to logic programming languages [Kowalski and Clark 2003] like PROLOG, which can solve such subproblems bottom-up but must handle dependencies at runtime. Dyna can no longer be compiled (as verified by the author), so its exact performance is unknown.

11 CONCLUSION

We have described a compiler for a language of general recurrences that can express programs across several domains, including dynamic programs, direct matrix solvers, graph algorithms, and tensor algebra. The compiler lowers recurrences into imperative loop nests that iterate over dense or sparse data structures. Our compiler controls loop ordering and uses induction to manage the dependencies inherent to recurrences. We showed how fusion can be applied to recurrences over sparse data structures, letting us generate code competitive with handwritten libraries.

Observing that these problems share a theoretical foundation, we envision a shared ecosystem in which optimizations developed for one recurrence algorithm can be easily and readily applied to similar algorithms in other fields, allowing for improvements in program performance and developer productivity to be shared across the respective domains.

12 ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their valuable feedback, comments, and insights on improving this manuscript. We wish to thank Scott Kovach for valuable discussions on recursive computations and for extensive comments on an early draft of the paper. We also thank Olivia Hsu, Rohan Yadav, Nathan Zhang, Matthew Sotoudeh, Manya Bansal, AJ Root, Rubens Lacouture, Bobby Yan, James Dong, and Alexander Rucker for their comments on an early draft. This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work was in part supported by the National Science Foundation under Grant CCF-2216964. Shiv Sundram was supported by an NSF Graduate Research Fellowship.

13 DATA-AVAILABILITY STATEMENT

Performance results were generated with a publicly available artifact [Sundram et al. 2024] containing all benchmarking code and scripts. Instructions for reproducibility are available via an archived version of the artifact on Zenodo. Benchmarking results may vary based on the hardware used.

REFERENCES

Richard E Bellman. 2010. *Dynamic programming*. Princeton university press.

Aart J.C. Bik, Bixia Zheng, Fredrik Kjolstad, Nicolas Vasilache, Penporn Koanantakool, and Tatiana Shpeisman. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Transactions on Architecture and Code Optimization* (2022).

Boost. 2002. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., USA.

Gaurav Chaurasia, Jonathan Ragan-Kelley, Sylvain Paris, George Drettakis, and Fredo Durand. 2015. Compiling high performance recursive filters. In *Proceedings of the 7th conference on high-performance graphics*. 85–94.

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (oct 2018), 30 pages. https://doi.org/10.1145/3276493

Jeffrey A. Daily. 2016. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics* 17 (2 2016). https://doi.org/10.1186/s12859-016-0930-z

T. A. Davis. 2006a. *CXSparse: a Concise eXtended Sparse Matrix Package.* https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/dev/CXSparse

T. A. Davis. 2006b. *Direct Methods for Sparse Linear Systems.* SIAM, Philadelphia, PA.

Jason Eisner, Eric Goldlust, and Noah A. Smith. 2004. Dyna: a declarative language for implementing dynamic programs. In *Annual Meeting of the Association for Computational Linguistics*.

Paul Feautrier. 1991. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

Gautam and S. Rajopadhye. 2006. Simplifying Reductions. *SIGPLAN Not.* 41, 1 (jan 2006), 30–41.

John R Gilbert. 1994. Predicting structure in sparse matrix computations. *SIAM J. Matrix Anal. Appl.* 15, 1 (1994), 62–79.

John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.* 27, 4 (dec 2001), 422–455. https://doi.org/10.1145/504210.504213

Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

Liang Huang. 2008. Advanced dynamic programming in semiring and hypergraph frameworks. *Coling 2008: Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications-Tutorial notes* (2008), 1–18.

Richard M Karp, Raymond E Miller, and Shmuel Winograd. 1967. The Organization of Computations for Uniform Recurrence Equations. *J. ACM* 14, 3 (1967), 563–590.

Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 180–192.

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 77:1–77:29.

Fredrik Berg Kjølstad. 2020. *Sparse tensor algebra compilation.* Ph. D. Dissertation. Massachusetts Institute of Technology.

Robert Kowalski and Keith L Clark. 2003. Logic programming. In *Encyclopedia of Computer Science*. 1017–1031.

Leslie Lamport. 1974. The Parallel Execution of DO Loops. *Commun. ACM* 17, 2 (1974), 83–93.

R Sherman Lehman. 1960. *DYNAMIC PROGRAMMING AND GAUSSIAN ELIMINATION.* Technical Report. RAND CORP SANTA MONICA CALIF.

Joseph W Liu. 1986. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Transactions on Mathematical Software (TOMS)* 12, 2 (1986), 127–148.

Joseph W. H. Liu. 1992. The Multifrontal Method for Sparse Matrix Solution: Theory and Practice. *SIAM Rev.* 34 (1992), 82–109.

Vijay Menon and Keshav Pingali. 2004. Look left, look right, look left again: An application of fractal symbolic analysis to linear algebra code restructuring. *International Journal of Parallel Programming* 32 (2004), 501–523.

Esmond Ng and Barry W. Peyton. 1993. A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors. *SIAM Journal on Scientific Computing* 14, 4 (1993), 761–769. https://doi.org/10.1137/0914048 arXiv:https://doi.org/10.1137/0914048

James M. Ortega. 1988a. The ijk forms of factorization methods I. Vector computers. *Parallel Comput.* 7 (1988), 135–147.

James M. Ortega. 1988b. The ijk forms of factorization methods II. Vector computers. *Parallel Comput.* 7 (1988), 149–162.

Louis-Noel Pouchet. 2016. *PolyBench.* http://web.cs.ucla.edu/~pouchet/software/polybench/

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530.

MMG Ricci and Tullio Levi-Civita. 1900. Méthodes de calcul différentiel absolu et leurs applications. *Math. Ann.* 54, 1-2 (1900), 125–201.

Georg Sauthoff, Mathias Möhl, Stefan Janssen, and Robert Giegerich. 2013. Bellman's GAP—a language and compiler for dynamic programming in sequence analysis. *Bioinformatics* 29, 5 (01 2013), 551–560. arXiv:https://academic.oup.com/bioinformatics/article-pdf/29/5/551/16919063/btt022.pdf

Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (nov 2020), 30 pages. https://doi.org/10.1145/3428226

Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

Shiv Sundram, Muhammad Usman Tariq, and Fredrik Kjolstad. 2024. Artifact for OOPSLA 2024 Paper: Compiling Recurrences over Dense and Sparse Arrays (version 1). https://doi.org/10.5281/zenodo.10774458

Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High Performance Sparse Tensor Algebra Compiler in MLIR. (12 2021).

A. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* 13, 2 (1967), 260–269. https://doi.org/10.1109/TIT.1967.1054010

Cambridge Yang, Eric Atkinson, and Michael Carbin. 2021. Simplifying Dependent Reductions in the Polyhedral Model. *Proc. ACM Program. Lang.* 5, POPL, Article 20 (jan 2021), 33 pages.

Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 660–678.

Received 21-OCT-2023; accepted 2024-02-24