RESEARCH-ARTICLE

REPTILE: Performant Tiling of Recurrences

MUHAMMAD USMAN TARIQ, Stanford University, Stanford, CA, United States

SHIV SUNDRAM, Stanford University, Stanford, CA, United States

FREDRIK BERG KJØLSTAD, Stanford University, Stanford, CA, United States

Published: 09 October 2025
Accepted: 12 August 2025
Received: 25 March 2025

REPTILE: Performant Tiling of Recurrences

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

We introduce REPTILE, a compiler that performs tiling optimizations for programs expressed as mathematical recurrence equations. REPTILE recursively decomposes a recurrence program into a set of unique tiles and then simplifies each into a different set of recurrences. Given declarative user specifications of recurrence equations, optimizations, and optional mappings of recurrence subexpressions to external libraries calls, REPTILE generates C code that composes compiler-generated loops with calls to external hand-optimized libraries. We show that for direct linear solvers expressible as recurrence equations, the generated C code matches and often exceeds the performance of standard hand-optimized libraries. We evaluate REPTILE's generated C code against hand-optimized implementations of linear solvers in Intel MKL, as well as two non-solver recurrences from bioinformatics: Needleman-Wunsch and Smith-Waterman. When the user provides good tiling specifications, REPTILE achieves parity with MKL, achieving between 0.79–1.27x speedup for the LU decomposition, 0.97–1.21x speedup for the Cholesky decomposition, 1.61x–2.72x for lower triangular matrix inversion, 1.01–1.14x speedup for triangular solve with multiple right-hand sides, and 1.14–1.73x speedup over handwritten implementations of the bioinformatics recurrences.

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

Additional Key Words and Phrases: recurrences, tiling, code generation, linear solvers, BLAS/LAPACK

ACM Reference Format:
Muhammad Usman Tariq, Shiv Sundram, and Fredrik Kjolstad. 2025. REPTILE: Performant Tiling of Recurrences. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 296 (October 2025), 27 pages. https://doi.org/10.1145/3763074

1 Introduction

Recurrence equations are ubiquitous in engineering, optimization, and scientific computing. These equations describe how to compute a value of a sequence based on previous values in the same sequence. They are particularly useful for expressing compute-bound linear solvers, such as the Cholesky decomposition, LU decomposition, and triangular solve.

Recent work by Sundram et al. [2024] showed how to compile recurrences to native code. A recurrence compiler allows users to separately specify mathematical recurrence equations and performance optimizations, which the compiler then lowers to imperative loop nests. The generality of such compilers allows for rapid prototyping, in which users can quickly generate programs for arbitrary recurrences and quickly experiment with different performance optimizations. For most memory-bound recurrences, recurrence compilers can generate code that is competitive with handwritten libraries. However, for recurrences that are compute bound and have exploitable reuse, such as direct linear solvers, the generated code does not match the performance of the hand-optimized routines found in libraries such as BLAS and LAPACK.

Authors' Contact Information: Muhammad Usman Tariq, Stanford University, Stanford, USA, usman25@stanford.edu; Shiv Sundram, Stanford University, Stanford, USA, shiv1@stanford.edu; Fredrik Kjolstad, Stanford University, Stanford, USA, kjolstad@stanford.edu.

ACM 2475-1421/2025/10-ART296
https://doi.org/10.1145/3763074

While a recurrence compiler achieves reasonable performance for a general set of problems, BLAS and LAPACK libraries achieve high performance for a limited class of algorithms that includes solvers. Performant solvers are the result of significant hand tuning and optimization, particularly as they employ loop tiling to utilize on-chip caches and specialize tiles to different types of computations. The effort to hand-optimize solvers makes it difficult to write new variants or to tune them for new machines and architectures with different levels of parallelism or different cache hierarchies. Thus, solver libraries are often optimized for a specific architecture and specific problem sizes, and they often need to be re-tuned, or even partially rewritten, to adapt to different architectures and inputs. Moreover, each solver recurrence can have multiple valid loop orderings with different performance tradeoffs, can be implemented with different tiling decisions, can call different hand-optimized subroutines for tiles, and can have fused or unfused loops. Since the space of optimizations is large, and depends on the input size and target architecture, it is impractical to hand-write kernels for every combination, which motivates a compiler approach.

When direct solvers expressible as recurrences are tiled, each tile can potentially be specialized and thus be computed with a different hand-optimized library subroutine. For example, if a Cholesky factorization is tiled column-wise, it gets split into four tiles, where one computes a recursive Cholesky, one a triangular solve, one a symmetric ranked update, and one a matmul. Each of these has existing library routines that can be used to accelerate the computation. Frameworks like FLAME [Gunnels et al. 2001], PLASMA [Bosilca et al. 2011], and ReLAPACK [Peise and Bientinesi 2016] utilize this recursive decomposition, but do not provide the ability to generate code for an arbitrary recurrence solver. In addition, they do not compose calls to external library routines with generated code, and do not let users specify a desired loop ordering. To attain high performance across architectures, it is important to have control over how a recurrence-based solver recursively decomposes into other routines under different loop orderings, to decide how deep to unroll this recursion, and to control how to map the resulting linear algebra routines to external functions.

We present a compiler technique that tiles user-provided recurrence equations and generates efficient C code. Given a specification of what dimensions to tile and the tile sizes, our compiler automatically determines what must be computed in different tiles, enforces the recurrence's dependencies, and gives the user the opportunity to map the different tiles to third-party library routines that are available. The major contributions of this paper are:

(1) A **method** that recursively decomposes an arbitrary recurrence equation into specialized tile recurrences.
(2) A **scheduling language** in which the user can specify tiling directions and sizes, along with mappings of tiles to external calls.
(3) A **C code generation algorithm** that lowers tiles to executable C code. The algorithm handles both inter-tile and intra-tile dependencies, and the generated C code implements the recurrence with the user-provided optimization, tiling, and mapping decisions.

We developed a prototype compiler called REPTILE (Recurrences with Performant Tiling). REPTILE tiles the iteration space of a system of arbitrary recurrence equations, creating unique tiles containing different recurrences. It then lowers these tiles to C code, while respecting dependencies between tiles, that matches tiles to optimized library calls as specified by the user. By mapping tiles to existing linear algebra subroutines, we show that the generated code matches or exceeds the performance of existing libraries like BLAS and LAPACK, while maintaining the ability to handle any recurrence in our recurrence language.

2 Motivating Example

To demonstrate the tiling strategy of our recurrence compiler, we use the Cholesky decomposition as a running example. The Cholesky decomposition is a commonly used matrix decomposition for solving systems of linear equations. It factors a symmetric positive-definite matrix A into a lower triangular matrix L such that $A = LL^T$. The routine has $O(n^3)$ complexity, and implementations are typically hand-optimized to get good performance. The Cholesky decomposition can be concisely expressed as the two mutually dependent recurrence equations shown in Figure 1 (left). These recurrences calculate elements of L by computing on previously calculated elements of L.

$$L_{ij} = \frac{A_{ij} - \sum_k L_{ik} L_{jk}}{L_{jj}} : k < j < i$$

$$L_{jj} = \sqrt{A_{jj} - \sum_k L_{jk} L_{jk}} : k < j$$

```
Cholesky(A, L, n):
  for j from 0 to n:
    for k from 0 to j:
      for i from j to n:
        L[i,j] += L[i,k] * L[j,k]
    L[j,j] = sqrt(A[j,j] - L[j,j])
    for i from j+1 to n:
      L[i,j] = (A[i,j] - L[i,j]) / L[j,j]
```

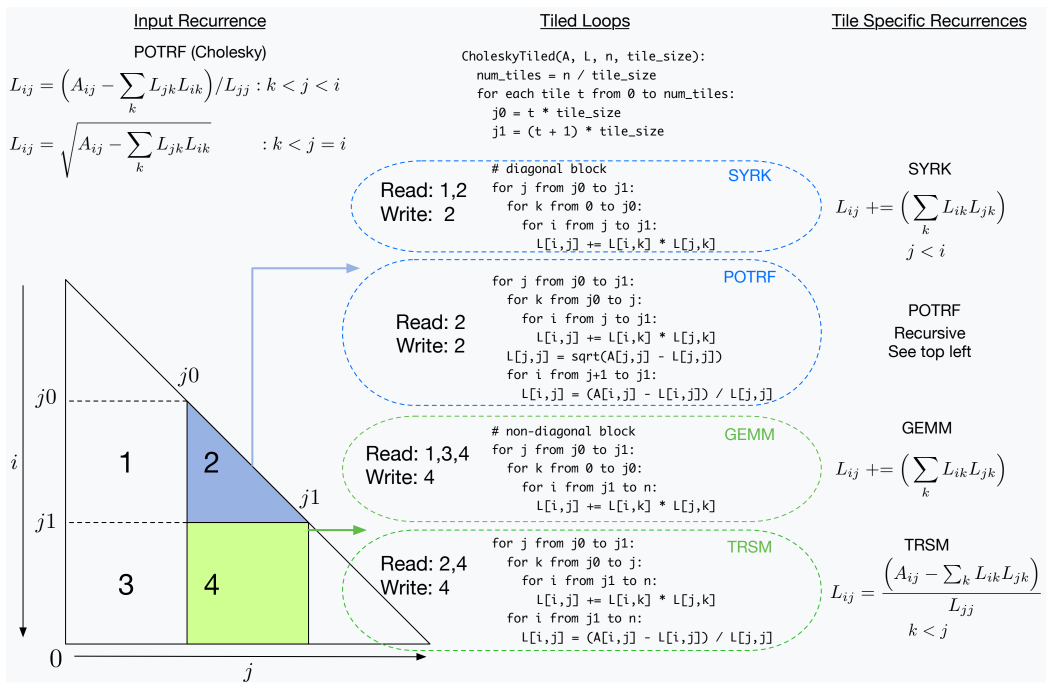
Fig. 1. The Cholesky matrix decomposition. Left: the recurrence equations for calculating Cholesky. These equations calculate elements of the output L by using previously calculated values of L lying in previous columns and rows. Right: a naive *jki* implementation of Cholesky expressed in an imperative IR. The loop nest exhibits poor memory locality and data reuse and does not match the performance of handwritten libraries.

Using the RECUMA recurrence compiler [Sundram et al. 2024], the Cholesky equations can be compiled to the loops in Figure 1. These loops use the user-defined *jki* loop order, and thus compute the matrix column by column, where each column depends on previously computed columns.

This naive implementation, however, has poor performance due to its memory access patterns. Modern computer architectures have multiple levels of caches, and efficient implementations must take advantage of this cache hierarchy. The naive implementation will iterate over full rows and columns repeatedly, leading to poor cache utilization. By the time a value is needed again, it has likely been evicted from cache, resulting in expensive memory accesses. To improve performance, it is necessary to improve data reuse by ensuring that cached data is reused before it is evicted.

The solution is tiling, where instead of computing columns sequentially, we compute multiple adjacent columns as a group, as shown in Figure 2. In this way, the program iterates over blocks of the output tensor, in which a columnn block is small enough to fit in a cache. Throughout this paper, we refer to subregions of the program's iteration space as *tiles*, and the corresponding subregions of the output tensor as *blocks*. Figure 2 shows how instead of calculating a full, individual, column at a time, the blue block (2) is fully calculated before the green block (4). This ensures that operations on a block are completed before it gets evicted from the cache, reducing the need for repeated data reads. The tiling strategy shown in Figure 2 divides the computation into groups of columns, where one of the groups spans columns from $j0$ to $j1$. Furthermore, within the group, we can see the blue and green regions, which we call the diagonal and non-diagonal block here respectively. The blue, diagonal block (2) spans rows $j0$ to $j1$ and the green, non-diagonal block (4) spans rows $j1$ to n.

The code for computing each of the diagonal and non-diagonal block is different. This is because different parts of the input recurrences' expression get evaluated to compute the blue region than to compute the green region. Furthermore, in the case of Cholesky decomposition, output blocks are dependent on entries computed in previous blocks. Therefore, we have two loop nests for the diagonal block, where each loop nest represents a tile of the program's iteration space. Here, a



$$L_{ij} = \left(A_{ij} - \sum_k L_{jk}L_{ik}\right)/L_{jj} : k < j < i$$

$$L_{ij} = \sqrt{A_{ij} - \sum_k L_{jk}L_{ik}} \quad : k < j = i$$

```
CholeskyTiled(A, L, n, tile_size):
  num_tiles = n / tile_size
  for each tile t from 0 to num_tiles:
    j0 = t * tile_size
    j1 = (t + 1) * tile_size
    # diagonal block
    for j from j0 to j1:
      for k from 0 to j0:
        for i from j to j1:
          L[i,j] += L[i,k] * L[j,k]
    for j from j0 to j1:
      for k from j0 to j:
        for i from j to j1:
          L[i,j] += L[i,k] * L[j,k]
      L[j,j] = sqrt(A[j,j] - L[j,j])
      for i from j+1 to j1:
        L[i,j] = (A[i,j] - L[i,j]) / L[j,j]
    # non-diagonal block
    for j from j0 to j1:
      for k from 0 to j0:
        for i from j1 to n:
          L[i,j] += L[i,k] * L[j,k]
    for j from j0 to j1:
      for k from j0 to j:
        for i from j1 to n:
          L[i,j] += L[i,k] * L[j,k]
      for i from j1 to n:
        L[i,j] = (A[i,j] - L[i,j]) / L[j,j]
```

$$L_{ij} \mathrel{+}= \left(\sum_k L_{ik}L_{jk}\right) \quad j < i$$

$$L_{ij} \mathrel{+}= \left(\sum_k L_{ik}L_{jk}\right)$$

$$L_{ij} = \frac{\left(A_{ij} - \sum_k L_{ik}L_{jk}\right)}{L_{jj}} \quad k < j$$

Fig. 2. Tiled Cholesky decomposition. The triangular blue block (2) now gets fully computed before the rectangular green block (4) is computed. This displays better locality than the naive solution. Each block requires two different tiles to compute, and each tile corresponds to a different linear algebra subroutine shown on the right. There is notably a recursive call to a smaller Cholesky decomposition (POTRF) to compute block 2. All four subroutines are commonly found BLAS/LAPACK subroutines. Note that j's tiling bounds $j0$ and $j1$ appear on the i axis as well. On the i axis, $j0$ and $j1$ represent the the current values of those tiling bounds, now projected onto the i axis. This projection allows easy representation of diagonal blocks.

tile is defined as a subset of the program's iteration space. The first loop nest (i.e., tile) reads from previously computed blocks and writes to the diagonal block, while the second tile finalizes the computation for the diagonal block. Each of these tiles include different code; the two loop nests that compute the diagonal block implement a SYRK and POTRF operation respectively, both of which are common linear algebra kernels. Similarly, for the non-diagonal block, we also have two loop nests, where the first one implements a GEMM and the second one implements a TRSM. Each tile is thus associated with a unique subroutine, and each of these four routines are commonly implemented linear algebra routines:

(1) **SYRK** (Symmetric Rank-K Update): This operation represents the reads from previous blocks, and multiplying the previous block by its own transpose, to make partial, symmetric updates to the diagonal block (blue triangular region 2).
(2) **POTRF** (Positive definite matrix, Triangular Factorization): This operation finalizes the computation for the diagonal block with a per-tile Cholesky decomposition.
(3) **GEMM** (Matrix Multiply): This operation represents reading previous blocks, multiplying them together, to make partial updates to the non-diagonal block (green square region 4).
(4) **TRSM** (Triangular Solve Multiple Right Hand Sides): This operation finalizes the computation for the non-diagonal block with a per block triangular solve.

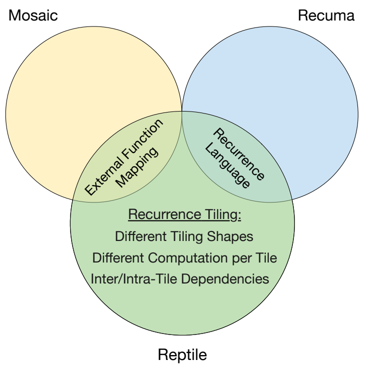

Mosaic
Recuma
External Function Mapping
Recurrence Language
Recurrence Tiling:
Different Tiling Shapes
Different Computation per Tile
Inter/Intra-Tile Dependencies
Reptile

Fig. 3. REPTILE's delta with existing tensor algebra and recurrence compilers.

ConstInt	n		
Tensor	t		
IndexVar	v		
Index	i	::=	$v + n \mid n$
TensorAccess	ta	::=	$t_{i^+}^{i^*}$
Expr	e	::=	$ta \mid \sum_v e \mid const$
			$\mid \sqrt{e} \mid e + e \mid e\ e \mid \cdots$
Recurrence	r	::=	$ta = e$
Constraint	c	::=	$v < i \mid v \leq i \mid v = i$
Constraints	cs	::=	$c \mid c, cs$
ConstrainedRec	cr	::=	$r : cs$
Program	p	::=	$cr \mid cr, p$

Fig. 4. Grammar for the recurrence language.

Thus, the tiled Cholesky decomposition code can be expressed as a series of subroutines, in which all four subroutines are used to calculate a contiguous group of columns. For this example and many other input recurrence equations, once the set of input recurrences is tiled, we can further optimize the computation by computing each resulting tile with an optimized linear algebra library kernel (if one exists). This allows us to get performance matching that of state of the art implementations. However, doing this requires knowing how to tile the recurrence while respecting the recurrence's dependencies, and it also requires knowing how to map a tile to corresponding library routine(s).

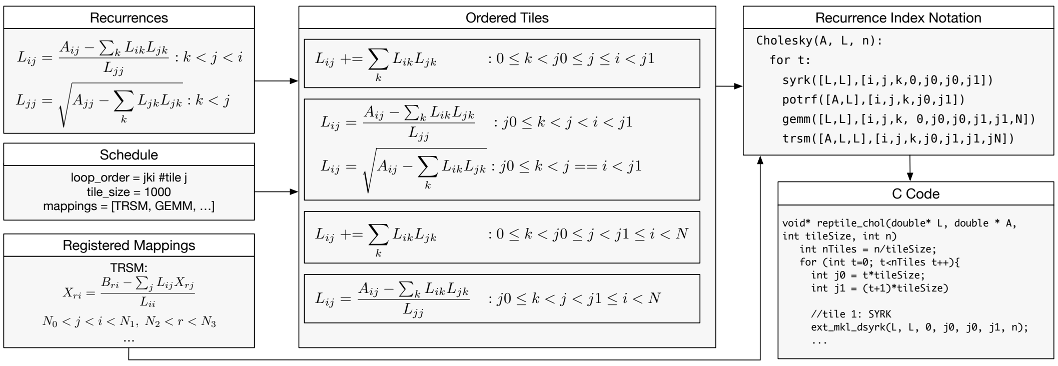
3 Background

We show how to develop a general framework that tiles, lowers, and optimizes recurrence programs with a mix of external kernels and compiler-generated loops. We build on the RECUMA recurrence language of Sundram et al. [2024], but our key contribution in REPTILE is a novel technique for tiling recurrence equations. In particular, we describe a compiler that can (1) generate tiles of different shapes, (2) identify different computations (i.e., recurrences) for different tiles, and (3) manage dependencies between tiles. In addition, we use the ideas from the Mosaic system for external plugins developed by Bansal et al. [2023] to generate code that calls out to external functions for the tiles in the code that REPTILE generates. However, in contrast to Mosaic, which is a tensor algebra compiler, we adapt this plugin system to recurrence equations.

Figure 3 shows how our approach relates to the RECUMA and Mosaic approaches. We adapt the input language of RECUMA and the mapping mechanisms of Mosaic, but introduce new mechanisms to address the problem of tiling recurrences. The grammar for the RECUMA [Sundram et al. 2024] input language of recurrences is shown in Figure 4. The recurrence language is defined by the following three features not found in tensor algebra, each of which makes tiling recurrences more complicated than tiling regular tensor algebra expressions:

Index Variable Constraints → different tile shapes Recurrence equations, unlike basic tensor algebra, support constraints that bound the domain of index variables in terms of other index variables as equalities or inequalities of index variables (e.g., $j = i$ or $j < i$). Thus, tiles in recurrences may have different shapes, such as the triangular and rectangular tiles in Figure 2, whereas tiles in tensor algebra are always rectangular.

Multiple expressions and non-distributive functions → different tile computations Recurrence programs may contain multiple interleaved recurrence expressions that compute different parts of the result, as seen in the Cholesky decomposition example. Likewise, tiles may compute different expressions, whereas tiles in tensor algebra programs compute the



$$L_{ij} = \frac{A_{ij} - \sum_k L_{ik}L_{jk}}{L_{jj}} : k < j < i$$

$$L_{jj} = \sqrt{A_{jj} - \sum_k L_{jk}L_{jk}} : k < j$$

```
loop_order = jki #tile j
tile_size = 1000
mappings = [TRSM, GEMM, ...]
```

TRSM:

$$X_{ri} = \frac{B_{ri} - \sum_j L_{ij}X_{rj}}{L_{ii}}$$

$$N_0 < j < i < N_1,\ N_2 < r < N_3$$

...

$$L_{ij} \mathrel{+}= \sum_k L_{ik}L_{jk} \quad : 0 \le k < j0 \le j \le i < j1$$

$$L_{ij} = \frac{A_{ij} - \sum_k L_{ik}L_{jk}}{L_{jj}} \quad : j0 \le k < j < i < j1$$

$$L_{ij} = \sqrt{A_{ij} - \sum_k L_{ik}L_{jk}} : j0 \le k < j == i < j1$$

$$L_{ij} \mathrel{+}= \sum_k L_{ik}L_{jk} \quad : 0 \le k < j0 \le j < j1 \le i < N$$

$$L_{ij} = \frac{A_{ij} - \sum_k L_{ik}L_{jk}}{L_{jj}} \quad : j0 \le k < j < j1 \le i < N$$

```
Cholesky(A, L, n):
  for t:
    syrk([L,L],[i,j,k,0,j0,j0,j1])
    potrf([A,L],[i,j,k,j0,j1])
    gemm([L,L],[i,j,k, 0,j0,j0,j1,j1,N])
    trsm([A,L,L],[i,j,k,j0,j1,j1,jN])
```

```
void* reptile_chol(double* L, double * A,
int tileSize, int n)
   int nTiles = n/tileSize;
   for (int t=0; t<nTiles t++){
     int j0 = t*tileSize;
     int j1 = (t+1)*tileSize)

     //tile 1: SYRK
     ext_mkl_dsyrk(L, L, 0, j0, j0, j1, n);
     ...
```

Fig. 5. Overview of REPTILE's input recurrences, mappings, simplified recurrence in tiles, and IR (which is then translated to C code). The resulting code can contain function calls, or loops with compute statements (as shown in Figure 1), or a mix of both.

same expression. Moreover, recurrences that include operations that compute on the result of a reduction must only compute the operation on the last tile.

Dependencies → tile ordering Recurrences are in-place computations that use previously computed values to compute the current value. Similarly, a tile may be dependent on previously computed tiles. Tiled recurrence programs must consider correct orders in which to compute the tiles, whereas all of tensor algebra is embarrassingly parallel. REPTILE manages these dependencies through its hierarchial dependency management system, operating both at the inter-tile and intra-tile levels.

While existing tensor algebra compiler techniques serve as an important foundation for tiling computation heavy workloads, they fundamentally require inputs to be embarrassingly parallel, compute only one expression, and have rectangular iteration domains. Recurrences, on the other hand, have these additional requirements. A tiled recurrence compiler must generate tiles of different shapes, tiles with different computations, and handle inter-tile dependencies. It must include new mechanisms to deduce which recurrences are computed in each tile. Moreover, it must identify the sub-expression of each such recurrence to compute in the tile. It therefore should take in a system of recurrences described in a recurrence language, and emit a set of tiles, in which each is associated with a unique system of *simplified* recurrences it computes. Once tiles of simplified recurrences are generated, it should be able to follow a user-provided specification that determines whether code for each tile will be compiler-generated loops or a call to an external function.

4 Overview

REPTILE is a recurrence compiler that applies a user-specified tiling strategy to recurrence equations and then compiles them to a mixture of C code and calls to optimized external library functions. Figure 5 presents an overview of REPTILE, showing the key stages of compilation from the input recurrences to the final, generated C code.

The compiler takes in two inputs: the recurrence equations expressed in a recurrence language, a schedule specifying loop order (which determines the tiling direction, e.g., tiling rows vs. columns), tile size, and external function mappings. The recurrence equations are expressed in the same recurrence language as the RECUMA system of Sundram et al. [2024], in which summations and element wise operations operate over indexed tensors and in which iteration variables can be related to each other via equality or inequality constraint that appear after a colon. One example is

```
# Define input recurrences for Cholesky
cholRec1 = Recurrence("L[i,j] = (A[i,j]-sigma(k,0,j, L[j,k]*L[i,k]))/L[j,j] : 0<=k<j<i<N")
cholRec2 = Recurrence("L[i,j] = sqrt(A[i,j]-sigma(k,0,j, L[j,k]*L[j,k])) : 0<=k<j==i<N")
recProgram = RecurrenceProgram([cholRec1, cholRec2])

# Register TRSM as an external function:
# Define its capability as a recurrence:
lib['trsm'].capability = Recurrence("X[r,i] = (B[r,i] - sigma(j,N0,i,M[i,j]*X[r,j]))/M[i,i]
              : N0<=j<i<N1, N2<=r<N3")
# Define its function signature:
lib['trsm'].func = "trsm([B,M,X], [r,i,j, N0,N1,N2,N3])" #RIN signature

# Map variables and tensors from capability to input recurrence
# Formal arguments in the signature are replaced by actual arguments in mapping below
# Using the registered signature, map B->A, M->L, X->L, and all variables and tile bounds
mapping = "trsm([A,L,L], [i,j,k, j0,j1,j1,N])"

# Assign mapping to schedule, define tilesize and loop order
recProgram.schedule.ordering = [j,i,k] # tiling direction is j, the first var
recProgram.schedule.tileSize = 1000
isValidMap = recProgram.schedule.map(mapping, lib[tri]) # apply map if valid
rin = recProgram.codeGen() # generate RIN with trsm tile mapped to trsm function
output = rin.cCodeGen() # convert RIN to C code
```

Fig. 6. REPTILE user code for column tiled Cholesky decomposition with mapped TRSM.

the equation $A_i = \sum_j A_j : j < i$, in which every element is the sum of every previous element. The Cholesky recurrences shown in Figure 1 are also expressed in this recurrence language.

These recurrence and scheduling inputs undergo two lowering passes in REPTILE. The first pass lowers the input into tiles, which are recurrences that operate on smaller and disjoint subsections of the computation's iteration space. The second pass lowers tiles to an intermediate representation of imperative loops, called Recurrence Index Notation (RIN), and external function calls. The IR is then lowered to C code.

Based on the recurrence equations and a schedule, REPTILE selects a tiling direction. For example, in the jki loop order for the Cholesky equation presented in the example section, REPTILE tiles along the columns direction. Then, in the first lowering pass, REPTILE recursively divides the original iteration space to produce an ordered list of disjoint tiles, where the computation in each tile is expressed using the same recurrence language as the input equations. Furthermore, each of the tiles covers a unique area of the original iteration space such that the union of these disjoint tiles is equal to the original iteration space.

The second lowering pass lowers the tiles to an imperative IR code (called RIN) that represents loop nests and calls to external functions, and then to executable C code. This lowering stage manages dependencies both between tiles and within tiles when placing the recurrence equations. If it is not possible to generate a program that respects the loop order and recurrence dependencies (inter-tile and intra-tile), then the compiler reports an error.

Users can also provide mappings that associate recurrence expressions with external library functions, that can match on input recurrence subexpressions. Such mappings are provided by registering an external function with an associated recurrence describing the function's computation. Users can either specify mappings to pre-defined external functions or provide their own functions. These functions may also be REPTILE-generated functions, which can be treated as external functions. Precisely, a REPTILE schedule reflects tiling at one level of the hierarchy; by having a REPTILE function map a tile to another REPTILE function, users can compositionally express multilevel tiling. The capability language used to describe each external function's computation is same as the recurrence language used to define the tiles and input recurrences. The user provides

Original Equation:

$$A_i = \sqrt{\sum_j A_j} \quad : j < i$$

Tiles: $i_0 \leq i < i_1$

1) $A_i = \sum_j A_j \quad : 0 \leq j < i_0$ *(outer expression removed)*

2) $A_i = \sqrt{\sum_j A_j} \quad : i_0 \leq j < i_1$

3) $A_i = \sqrt{\sum_j A_j} \quad : i_1 \leq j < N$ *(discarded)*

Fig. 7. Intuition: the three tiles resulting from tiling a simple equation with the constraint $j < i$. Here, we tile with respect to i, whose tiling bounds are i_0 and i_1 in each tile. The three resulting tiles have different bounds for j. In the first tile, the square root is removed because the summation does not complete. The last tile is discarded/filtered out as its bounds are inconsistent with the constraint $j < i$

tensor and variable matchings, in order to validate that the input recurrence expression and external function's registered capability expression are the same, apart from changes in variable and tensor names. If the recurrence and external function are equivalent, the generated RIN will combine tiled loops with calls to optimized external functions. This imperative, loop-based IR is then finally translated into C. This stage uses standard techniques utilized by tensor algebra compilers like Mosaic (for lowering calls to external calls) and like TACO [Kjolstad et al. 2017], COMET [Tian et al. 2021], and MLIR SparseTensor [Bik et al. 2022] for lowering loops. These capabilities allow REPTILE to leverage existing high-performance libraries when available, while still maintaining the ability to compile recurrence equations for which no mappings exist to low-level code.

A sample input for generating a Cholesky decomposition in REPTILE is shown in Figure 6. The user defines the Cholesky decomposition recurrences, a tile size of 1000, and a *jik* loop ordering (which implies that we tile over j). An external TRSM function is registered, along with its capability expressed as a recurrence. A mapping to this TRSM is provided as part of the schedule, which directs the compiler to find the tile with the capability function associated with TRSM, verify the mapping is valid, and then compute that tile with the external function.

5 Tiling

Tiling a system of recurrences involves dividing the computation into smaller pieces that can be efficiently computed. This process requires five steps:

(1) Determining the tiling direction based on the given loop order and recurrence equations (Section 5.1) (lines 1-2 in Algorithm 1).
(2) Creating initial tiles by dividing the program's iteration space along the tiling direction (Section 5.2) (lines 3-7 in Algorithm 1).
(3) Filtering tiles by intersecting them with the recurrence equations' iteration space to determine which tiles contain valid computations (Section 5.3) (lines 8-9 in Algorithm 1).
(4) Simplifying each filtered tile by updating the equations and bounds of the original recurrence(s) to reflect the intersection with the iteration space of the tile (Section 5.4) (lines 10-12 in Algorithm 1).
(5) Determining the order in which the filtered tiles will be lowered to our IR (Section 5.5) (lines 10-11 in Algorithm 1).

The following subsections explain each step. Here, we present a brief example in Figure 7 to show the main intuition behind the tiling approach. In this example, we are tiling over the i direction.

Algorithm 1 Tiling Algorithm

```
   Input: RecurrenceProgram $P$, Schedule $S$
   Output: OrderedTileList
 1: Determining Tiling Direction
 2: $tilingVar \leftarrow S.loopOrdering[0]$
 3: Creating Tiles
 4: $nonTilingVars \leftarrow all\_variables(P) \setminus \{tilingVar\} = \{var_1, var_2, ..., var_n\}$
 5: Let $v_0, v_1$ be boundary variables for $tilingVar$
 6: $R = \{\{0, v_0\}, \{v_0, v_1\}, \{v_1, N\}\}$    ▷ Three possible ranges for each non-tiling variable
 7: $tiles \leftarrow \{Tile(tilingVar : \{v_0, v_1\}, var_1 : r_1, var_2 : r_2, ..., var_n : r_n) \mid r_1, r_2, ..., r_n \in R\}$
 8: Filter Tiles
 9: $\forall$ tile $\in$ tiles,   tile.recurrences := {rec $\in$ P.recurrences | rec.iterSpace $\cap$ tile.iterSpace $\neq \emptyset$}
10: Simplify Tiles
11: $\forall$ tile $\in$ tiles,   $\forall$ rec $\in$ tile.recurrences,   rec.updateEquation()
12: $\forall$ tile $\in$ tiles,   $\forall$ rec $\in$ tile.recurrences,   rec.updateConstraints()
13: Ordering Tiles
14: $orderedTiles \leftarrow$ sort $tiles$ by bounds of $P.nonReductionVars$ then $P.reductionVars$
15: return $orderedTiles$
```

This means that in each tile, i is bounded between $i0$ and $i1$. Meanwhile, the non-tiling variable j may lie in any of the three ranges of i as shown in Figure 7, each corresponding to a tile. Notably, in Tile 1, where $j < i0$, the original summation over j does not complete, due to the summation ending prematurely at $i0$ instead of i, in which $i0 \leq i$. This leads to the simplification of the recurrence: the outer expression surrounding the summation (square root) gets removed, giving us a different mathematical operation for Tile 1 when compared to Tile 2, where the summation does complete. This is because a square root operation does not distribute over a summation, so it can only be run if the summation has completed. Meanwhile, Tile 3, where $j \geq i1$ is discarded (i.e. filtered out), since it lies completely outside of the original bounds imposed upon the equation: $j < i$.

The idea of dividing the iteration space with respect to the bounds of tiling variable leads to tiles which may be computationally different from one another. All of the evaluated linear solver recurrences (LU, TRSM, TRSV, TRTRI, POTRF) exhibit similar properties; they include constraints (e.g. $j < i$) over indexing variables, summations over these variables, and non-distributive outer expressions over the summations. These outer expression get removed depending on the tile bounds, meaning different tiles now compute different recurrences.

5.1 Determining Tiling Direction

REPTILE selects the the tiling direction based on the outermost loop variable in the user-specified loop order (line 2 of Algorithm 1). This is done to ensure that any tiling first applies to the outer most loop, as it is the loop that will benefit the most from tiling. Tiling at the outermost loop level is particularly beneficial because it results in asymptotically smaller number of tiles (for example, tiles just over columns, as opposed to both rows and columns). Since we aim to map each tile to an external function, this results in less external function call overhead and thus better performance.

Consider the Cholesky decomposition recurrence equations from Figure 1 that has a user-specified jki loop order. In this case, j is the outermost loop variable and indexes the columns of the output tensor L on the recurrence's left-hand side. This indicates the outer loop will iterate over contiguous groups (i.e., blocks) of columns, resulting in a column-by-column computation pattern. Therefore, REPTILE chooses to tile in the column direction by selecting j as the tiling variable, creating vertical blocks of columns, where each column block will be computed sequentially after the column block to the left.

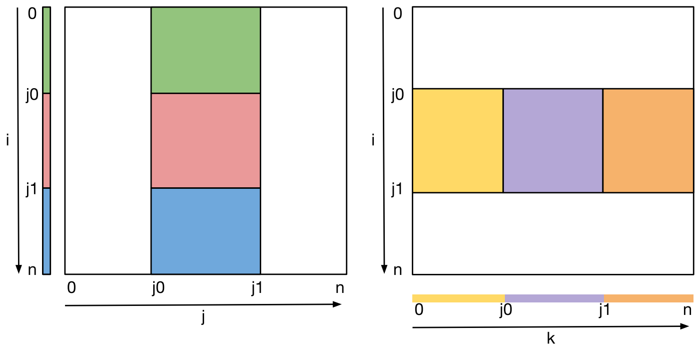


Fig. 8. On the left, we vary i (it can take on 3 different ranges as shown) with the tiling bounds of j. Here, the tiling variable j is bounded between $j0$ and $j1$. On the right we see how k varies (it can take on 3 different ranges) with respect to a particular range for the bounds of i (when i is between $j0$ and $j1$). In other words, the figure is showing a 3D iteration space via two 2D iteration spaces for the running Cholesky example.

5.2 Creating Tiles

To create tiles, we first define the boundaries for the tiling variable (Algorithm 1 line 5). For any tiling variable v, we define two boundary variables $v0$ and $v1$ that divide the iteration space into tiles of size $T_s = v1 - v0$ along the tiling direction(s). The tile size, T_s, is specified by the user in the schedule. For any v, the tiling boundaries $v0$ and $v1$ are calculated as $v0 = t \cdot T_s$ and $v1 = (t + 1) \cdot T_s$. Here t is the tile loop iteration variable ranging from 0 to $\frac{N}{T_s}$, with N being the size of the tensor. These boundaries create tiles that collectively span the entire iteration space as t increments.

We also handle non-tiling variables with respect to the boundaries of the tiling variable. Unlike a tiling variable, a non-tiling variable is not confined to the tiling bounds within a particular iteration of the outer tile loop t and can exist outside of these bounds. It is useful to view a non-tiling variable's iteration space in terms of the tiling variable's bounds. This is because the recurrences calculated by tiles can vary depending on which interval the non-tiling variables lies. In particular, each non-tiling variable can exist within one of three intervals relative to the tiling boundaries (the set R defined in line 6 of Algorithm 1):

- Before the tiling bounds of v (0 to $v0$)
- Within the tiling bounds of v ($v0$ to $v1$)
- After the tiling bounds of v ($v1$ to N)

For our running Cholesky example where j is the tiling variable, we introduce boundaries $j0$ and $j1$, calculated as $j0 = t \cdot T_s$ and $j1 = (t + 1) \cdot T_s$. The non-tiling variables i and k can each take three possible ranges relative to these boundaries:

Possible ranges for i:

- $0 \le i < j0$
- $j0 \le i < j1$
- $j1 \le i < N$

Possible ranges for k:

- $0 \le k < j0$
- $j0 \le k < j1$
- $j1 \le k < N$

The bounds for i and k each are illustrated in Figure 8, in which i and k can independently take one of these three ranges, while j is fixed within one range between $j0$ and $j1$. Figure 9 shows how the combinations of these range assignments (line 7 in Algorithm 1) results in $3 \times 3 \times 1 = 9$ tiles within each vertical slice bounded by $j0$ and $j1$. As t iterates from 0 to $\frac{N}{T_s}$, these nine tiles collectively cover the entire iteration space of the program.

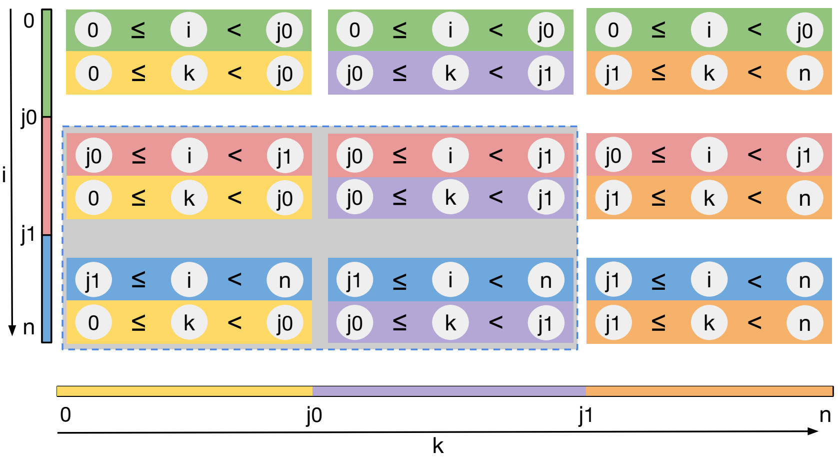


Fig. 9. Nine tiles are considered in a column tiling of a three dimensional iteration space. When tiling in the column direction, j is always in the interval between $j0$ and $j1$, but i and k may each lie before $j0$, within the $j0$ to $j1$ interval, or after $j1$. Since j is always between $j0$ and $j1$ it only has one possible interval. Given the 3 possibilities for i and k each, there are 3x3=9 program tiles total, shown above. In the Cholesky decomposition, after the filtration step where recurrences are assigned to tiles, only the four tiles within the blue-dotted box remain, and the other five tiles are not lowered to RIN.

5.3 Filter Tiles

The filtering pass shown in line 9 of Algorithm 1 discards some of these tiles, because the initial tiles may not all contain valid computations, meaning the tiles' boundaries may not be consistent with the original recurrence's constraints. We filter tiles by checking which ones intersect with the iteration space of the input recurrence equations. In particular, if a tile spans a non-empty subset of the iteration space of a recurrence, that recurrence gets assigned to that tile.

In case a tile isn't assigned any equation, it gets discarded. For example, in our running Cholesky decomposition example, tiles in the upper triangular region receive no equations because Cholesky only computes the lower triangular part. Therefore, of the nine initial tiles, only four remain in this filtering process for our Cholesky example. These four are enclosed inside the blue dotted box in Figure 9, and these four tiles emerge from two distinct regions:

(1) Where $j0 \leq i < j1$ (diagonal region):
  - One tile where k varies from 0 to $j0$, handling reads from previous tiles
  - One tile where k varies from $j0$ to $j1$, computing final output values

(2) Where $j1 \leq i < n$ (below diagonal region):
  - One tile where k varies from 0 to $j0$, handling reads from previous tiles
  - One tile where k varies from $j0$ to $j1$, computing final values

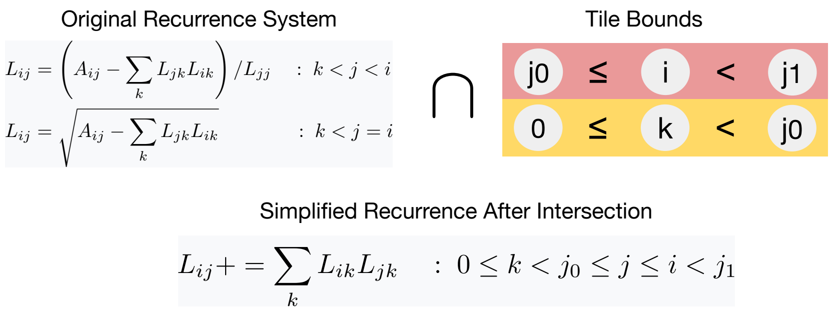
5.4 Simplify Tiles

The filtration process identifies tiles which contain valid computations. However, the creation of these tiles also leads to the imposition of tiling bounds over the bounds of the original equation (intersection of tiling bounds and bounds of the original equation). On top of this, we may also have equation simplifications via removal of outer expressions such that our tiles may end up being mathematically different from each other. We saw this in Figure 7 for the simple example of a square root over a summation, and the same idea can be extended to other examples, as seen in line 11 and 12 of Algorithm 1. Figure 10 demonstrates how intersecting the Cholesky recurrences with the provided tiling bounds results in a simplified recurrence without any outer expressions.

As stated, tile bounds can cut off summations in the recurrences, resulting in the removal of the initial recurrences' outer expressions within the tile. Figure 10 shows what the initial Cholesky recurrences change into under tiling bounds, for one of the four tiles. In particular, the summation does not complete since k is upper bounded by $j0$ inside this tile. Therefore, the outer expression including the subtraction and division/square root is removed for both the input recurrences. The resulting recurrence (with the updated equation and bounds) turns out to be an SYRK operation.

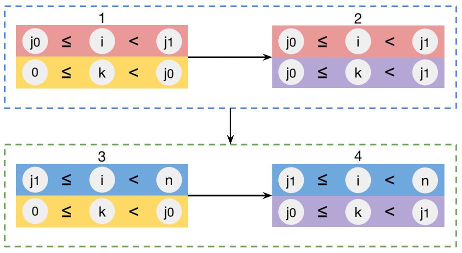
5.5 Ordering Tiles

The final step is to determine the order in which the filtered tiles will be lowered to our IR (line 11 in Algorithm 1). Since the tiling variable has fixed bounds within each tile, we order tiles based on the bounds of the non-tiling variables. In doing so, we prioritize non-reduction variables (those indexing the output tensor on the left-hand side of recurrence equations) before reduction variables (those appearing only on the right-hand side). This two-level prioritization is implemented by the sorting operation in line 11, which first sorts by non-reduction variables and then by reduction variables. This prioritization ensures the output tensor is built systematically, as non-reduction variables are the ones that index into the output tensor. Within each variable's ordering, tiles are arranged in ascending order based on their bounds. For our Cholesky example, Figures 11 and 12 show the ordering for the four tiles, and blocks of the output tensor that the four tiles write to respectively. Here, the four tiles are ordered as follows:



Original Recurrence System
$L_{ij} = \left(A_{ij} - \sum_k L_{jk}L_{ik}\right)/L_{jj} \quad : k < j < i$
$L_{ij} = \sqrt{A_{ij} - \sum_k L_{jk}L_{ik}} \quad : k < j = i$
$\cap$
Tile Bounds
j0 ≤ i < j1
0 ≤ k < j0
Simplified Recurrence After Intersection
$L_{ij} += \sum_k L_{ik}L_{jk} \quad : 0 \le k < j_0 \le j \le i < j_1$

Fig. 10. The Cholesky recurrence system is intersected with the given tiling bounds. The result is a simplified recurrence, which computes an SYRK, i.e. the part of the original recurrences corresponding to these bounds.



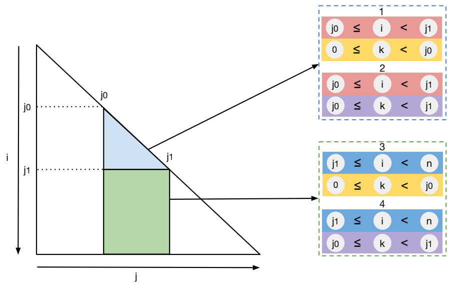


Fig. 11. Since the tiling variable j has fixed bounds within each tile, the order in which tiles will be lowered to our IR is determined based on the bounds of the non-tiling variables i and k.

Fig. 12. Tiles of the iteration space writing to blocks of the output tensor L for column-tiled Cholesky decomposition. Each block has two associated tiles, whose ordering is shown in Figure 11.

(1) First by non-reduction variable i: tiles operating on the range $j0 \leq i < j1$ come before those operating on $j1 \leq i < n$
(2) Then by reduction variable k: for tiles sharing the same i bounds, we order based on k's range ($0 \leq k < j0$ before $j0 \leq k < j1$)

6 Lowering Tiles to RIN

Tiles are lowered into an imperative IR called Recurrence Index Notation (RIN), whose grammar is shown in Figure 13. In the lowering process, the ordered tiles are lowered one by one, such that the code to compute each ordered tile is inserted into an initially empty RIN loop nest (lines 1-3 in Algorithm 2). Tiles consist of recurrence expressions with their constraints, and these expressions may be dependent on previously computed values. These previously computed values may either be in previous tiles or within the current tile itself. Therefore, we must track what has been computed at different points in the program to correctly place recurrence expressions at program points where their dependencies are satisfied. To do so, REPTILE manages dependencies at two levels: between tiles and within tiles, creating a hierarchical dependency management system (lines 2 and 5 in Algorithm 2).

REPTILE generalizes and adapts RECUMA's approach for handling dependencies in recurrence expressions during IR generation. RECUMA provides mechanisms for handling dependencies between compute statements, and REPTILE extends this to handle dependencies between tiles. RECUMA generates its IR by placing subexpressions from the user-provided recurrences into a loop nest. Code generation completes when each subexpression has been placed into the loop, at which point the loop represents the final IR for the program. RECUMA ensures a recurrence expression is only placed at a program point where its dependencies are satisfied. In particular, the expression is greedily placed in the first program point where all its dependencies are calculated.

REPTILE extends this idea hierarchically: code for a tile can only be placed into the IR once the tiles it is dependent on are computed (lines 9-10 in Algorithm 2). Within each tile, code generation is nearly identical to RECUMA's, and the placement of expressions into the loop nest for the specific tile follows RECUMA's dependency mechanism. Unlike RECUMA, in which each recurrence's expression appears in exactly one place in the final IR, the same recurrence expression for REPTILE might appear in multiple tiles. This is because in REPTILE, a recurrence may be assigned to more than one tile in the described filtration algorithm. Nevertheless, code generation of each REPTILE tile maintains its own isolated dependency analysis, while respecting cross-tile dependencies.

The dependencies are managed through two complementary mechanisms:

1. Determining what must be computed at the start of any loop iteration — whether it's the outer loop over tiles or inner loops within a tile (lines 7-8 in Algorithm 2).
2. Tracking the state of computation as we progress through the program, both at the tile level and within tiles (lines 2, 5, 13, and 16 in Algorithm 2).

The need for these two mechanisms arises from the nature of dependencies in recurrences: expressions may depend on values computed in previous iterations of a loop (requiring the first mechanism), and they may also depend on values computed within the same iteration (requiring the second mechanism). This is true both at the tile level, where a tile may depend on previous tiles or tiles in the same iteration, and within tiles, where expressions may have similar dependencies.

If the lowering process indicates that a particular tile can't be placed because its dependencies can't be satisfied, it means that tiling isn't possible for that recurrence and schedule, and REPTILE will inform the user accordingly. For example, in a tiling of jki Cholesky (column tiling), the lower block is dependent on the upper block in the output tensor. If a different recurrence had circular dependencies between a lower and upper block, column-wise tiling would be impossible.

Algorithm 2 REPTILE emits its IR tile by tile, and checks each tile's dependencies are satisfied before lowering the tile to RIN. The program maintains a complete state of what has been computed, which is formed through two complementary mechanisms: (1) Loop Iteration Readiness determines what must be computed at the start of any loop iteration at both tile level (line 7) and within tiles (line 8); and (2) Computation State Tracking updates what has been computed after placing each recurrence (line 13) and tile (line 16). Both mechanisms contribute to building the inter-tile state (lines 2, 7, 16) and intra-tile state (lines 5, 8, 13) that together determine if a tile's dependencies are satisfied (lines 9-10). If satisfied, the tile's RIN is appended to the program (line 15); otherwise, the compiler reports that tiling isn't possible (line 10).

```
    Input: OrderedTileList $orderedTiles$
    Output: RIN program
 1: $rinProgram \leftarrow$ ""
 2: $interTileState \leftarrow \{\}$                                    ▷ Tracks regions computed between tiles
 3: for $tile \in orderedTiles$ do
 4:     $tileRIN \leftarrow$ ""
 5:     $intraTileState \leftarrow \{\}$                                ▷ Tracks regions computed within a tile
 6:     for $recurrence \in tile.recurrences$ do
 7:         $interTileState.update(RegionComputedByPrevTileLoopIters(recurrence))$
 8:         $intraTileState.update(RegionComputedByPrevLoopIters(recurrence))$
 9:         if $recurrence.dependencies \not\subseteq (interTileState \cup intraTileState)$ then
10:             $Error$("Tiling not possible with given schedule: dependencies violated")
11:         end if
12:         $tileRIN.addtoLoopNest(recurrence)$
13:         $intraTileState.update(RegionComputedBy(recurrence))$
14:     end for
15:     $rinProgram.append(tileRIN)$
16:     $interTileState.update(RegionComputedBy(tile))$
17: end for
18: return $rinProgram$
```

IndexVar	v			For	for	::=	$\text{for}(i < v < i) : stmt^+$
Tensor	t			Assign	asn	::=	$t_{v^+} = e \mid t_{v^+} \mathrel{+}= e$
Index	i	::=	$v \mid v - int \mid int$	Statement	$stmt$	::=	$asn \mid for$
TensorAccess	ta	::=	t_{i^+}	RIN	rin	::=	$stmt^+$
Func	$func$	::=	$potrf([t^+], [v^+])$	Expression	e	::=	$ta \mid e + e$
			$\mid gemm([t^+], [v^+]) \mid \ldots$				$\mid e - e \mid \ldots \mid func$

Fig. 13. Grammar for Recurrence Index Notation. The IR grammar is the same as the RECUMA's RIN, with the only addition being the $func$ production to support calls to external functions.

6.1 Loop Iteration Readiness

Since recurrence expressions may be dependent on values computed in previous iterations of a loop, it is important to know what values are computed at the start of each loop iteration. This analysis applies hierarchically — both to loops over tiles and loops within tiles (lines 7-8 in Algorithm 2).

To reason about this, we see how loop variables index into the output tensor's dimensions. A similar analysis was introduced by the RECUMA compiler [Sundram et al. 2024] under the term "inductive assumptions" and we have extended it to reason about loops over tiles.

To illustrate this process, consider the lowering of the first tile for our Cholesky decomposition example. As seen in Figure 14, the recurrence expression with left hand-side L_{ij} is dependent on L_{ik} and L_{jk} which are in previous tiles. Therefore, we are unable to place the expression without

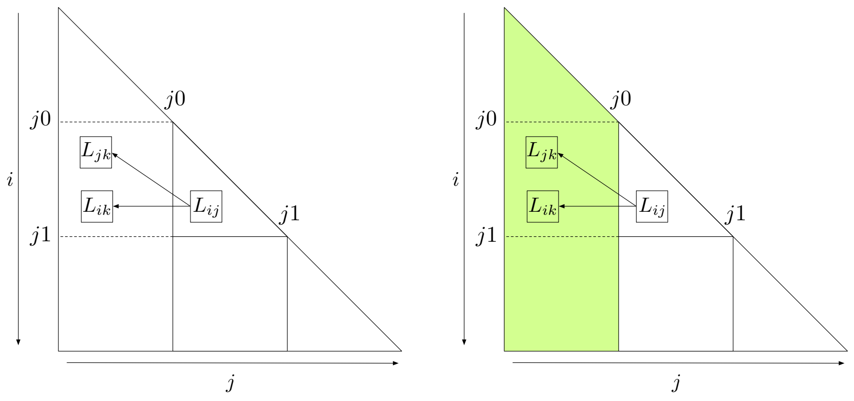


Fig. 14. For column-wise tiling of Cholesky decomposition, when computing the diagonal tile's recurrence $L_{ij} = (A_{ij} - \sum_k L_{ik}L_{jk})/L_{jj} : k < j < i$, all columns up to $j0$ (green) must be computed at the start of iteration t of the tile loop. The expression with left-hand side L_{ij} in tile 1 is dependent on L_{ik} and L_{jk}, which lie in the computed region. This allows the placement of this expression inside tile 1 to calculate L_{ij}.

```
for t in range(0, num_tiles)
  for j in range(j0, j1)
    for k in range (0, j)
      for i in range (j+1, j1)
```

(a) Tile 1 is dependent on values in previous tiles and can't be placed without determining what must be computed at the start of iteration t of the tile loop.

```
for t in range(0, num_tiles)
  computed L(:,:j0)
  for j in range(j0, j1)
    for k in range (0, j)
      for i in range (j+1, j1)
```

(b) We have determined that all columns up to $j0$ must be computed at the start of iteration t of the tile loop since the last iteration wrote to columns up to $j0$.

```
for t in range(0, num_tiles)
  computed L(:,:j0)
  tile1_code
```

(c) Tile 1 is placed since its dependencies are now satisfied. The code for computing the tile is not shown in the code above.

```
for t in range(0, num_tiles)
  computed L(:,:j0)
  tile1_code
  tile2_code
```

(d) Tile 2 is placed and the dependencies within the tile are managed similar to how RECUMA would reason about them for non-tiled code.

Fig. 15. Steps of the placement algorithm for generating RIN for tile 1 and 2 of a jki Cholesky decomposition. Green text show readiness markers that are not part of IR but internal to the compiler.

determining what has been computed at the start of iteration t of the tile loop. Since we are tiling over columns, we can determine that all columns before $j0$ must be computed at the start of iteration t ($j0$ being the lower bound for the current iteration). This is visualized on the right side of Figure 14, where the green region shows values which are assumed to be computed at the beginning of the iteration t of the tile loop (i.e., the outer most loop).

Similarly, within each tile, we need to track what's computed at the start of inner loop iterations (line 8 in Algorithm 2). This is because recurrence expressions may also be dependent on values within a tile itself which would involve determining what has been computed at the start of inner loops within a tile. For example, in a jki loop order within a tile, we might need to determine what is computed at the start of iteration j of the outer loop within this tile.

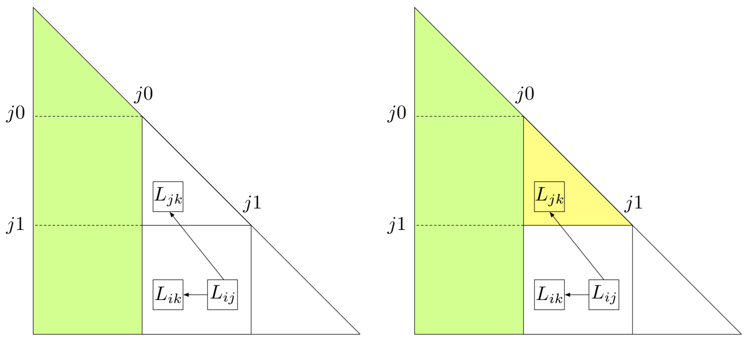


Fig. 16. Triangular block (yellow) marked as computed after placing tile 2 since no subsequent tile will write to this region of the output tensor. The recurrence expression with left-hand side L_{ij} in tile 4 is dependent on L_{jk}, which lies in the computed region. This allows the placement of this expression inside tile 4.

```
for t in range(0, num_tiles)
    computed L(:,:j0)
    tile1_code
    tile2_code
    computed L(j0:j1, j0:j)
```

(a) Yellow region of output tensor marked as computed since no future tile will modify this region. This information is recorded internally inside the IR with the "computed" statement after the code for tile 2.

```
for t in range(0, num_tiles)
    computed L(:,:j0)
    tile1_code
    tile2_code
    computed L(j0:j1, j0:j)
    tile3_code
    tile4_code
```

(b) Tile 3 and and Tile 4 placed. The existing "computed" information along with dependency analysis inside the tiles allow for the placement of these tiles.

Fig. 17. Steps of the placement algorithm for generating RIN for tile 3 and 4 of a jki Cholesky decomposition. Green text show readiness markers that are not part of IR but internal to the compiler.

More formally, for any loop l, we can determine what's computed at the start of its iteration by examining the dimension l indexes into. Let d' be this dimension, $x_{d'}$ be a variable that indexes into dimension d', $l0$ be the loop's lower bound for the current iteration, and N_i be the the bound of the tensor (or tile if a loop l is an inner loop within an individual tile) in dimension i. For the output tensor L we can specify:

- In dimension d': all values are computed for $0 \leq x_{d'} < l0$
- In all other dimensions $i \neq d'$: values are computed for $0 \leq x_i < N_i$

In the case of our Cholesky decomposition example, we have a 2-D output tensor L and we want to determine what is computed at the start of iteration t of the tile loop. Since we are tiling over columns, t indexes into the columns dimension so d' is the column dimension. The lower bound for the current iteration is $j0$ so we can determine that $L(:, : j0)$ is computed at the start of iteration t of the tile loop which means all columns of L upto $j0$ are computed at the start of iteration t of the tile loop. This reasoning applies to both the outer tile loop and to loops within tiles, allowing REPTILE to handle dependencies at both levels. The steps of the placement algorithm for lowering tile 1 and tile 2 of a jki Cholesky decomposition are shown in Figure 15.

6.2 Computation State Tracking

Recurrence expressions may depend on values computed within the current iteration of a loop, therefore we need to track what gets computed as we progress through an iteration of both the

Algorithm 3 Map Tiles to External Functions: The algorithm iterates through each tile in the RIN program (line 1), checking if a mapping exists for the tile (line 2). When a mapping is found, it verifies that both the expression and iteration space of the tile match those of the external function (line 4). If verification passes, the algorithm replaces the tile with the external function call (line 5).

```
   Input: RIN Program $rinProgram$, Mappings $mappings$
   Output: RIN Program with external function calls
1: for each $tile \in rinProgram.tiles$ do
2:     if $mappings(tile) \neq \emptyset$ then
3:         $function \leftarrow mappings(tile).function$
4:         if $function.expr \equiv tile.expr$ and $function.iterSpace \equiv tile.iterSpace$ then
5:             $rinProgram.tile \leftarrow function$
6:         end if
7:     end if
8: end for
9: return $rinProgram$
```

tile loop and inner loops within the tile. The lowering process must thus track the state of the computation at two levels: between tiles and within tiles. This hierarchical tracking is necessary because dependencies exist both between tiles (where a tile may need values from previously computed tiles) and within tiles (where expressions may depend on earlier results in the same tile).

For tile-level tracking, as tiles are placed into the loop, we identify regions of the output that become fully computed, meaning no future tile will modify those regions (line 16 in Algorithm 2). These regions are marked as "computed" when the last tile that writes to them has been placed. This tracking is crucial because future tiles may depend on values in these computed regions.

For example, in our Cholesky decomposition, after placing tile 2, we mark the triangular region between $j0$ and $j1$ as computed (as shown in yellow in Figure 16) since no subsequent tile will write to this region. This tracking enables us to later place tile 4, whose recurrence expression L_{ij} depends on L_{jk} from this computed region.

Within tiles, the tracking mechanism follows RECUMA's approach, monitoring which parts of expressions have been computed and can be used by subsequent computations in the same tile (line 13 in Algorithm 2). While we defer the details of within-tile tracking to RECUMA's formalism, the principle remains the same — tracking what's computed to ensure dependencies are satisfied. The lowering of Cholesky tiles 3 and 4 is shown in Figure 17, which demonstrates how computation state tracking enables placement of tiles dependent on previously computed tiles inside the loop.

7 Mapping Tiles to External Functions

While tiling itself is a useful optimization, in REPTILE it also serves as an intermediate step that enables the use of external library kernels to compute tiles with different subexpressions of the input recurrence equations. The user is given the ability to map these subexpressions to external functions as part of the scheduling language. As shown in Algorithm 3, REPTILE iterates through each tile in the RIN program (line 1) and verifies that the user-provided mapping is valid (lines 2-4). If so, the algorithm replaces that tile's loop nest in the generated RIN with a call to the external function (line 5). The need for a scheduling language is illustrated by the multiple ways a solver can be decomposed into different subroutines. Figure 18 shows how the Cholesky decomposition can be recursively decomposed and mapped to subroutines. Each of these subroutines (POTRF, GEMM, TRSM, SYRK) is itself a recurrence that can be further decomposed. For example, TRSM can be decomposed into a smaller TRSM and GEMM, resulting in a hierarchical tiling. The user can choose whether to map the larger and shallower TRSM directly to an external library, or to map its decomposed components (the smaller TRSM and GEMM) to external functions. For the latter case, the user can hierarchically tile the Cholesky by mapping the Cholesky's TRSM tile to a REPTILE-generated TRSM. In REPTILE, each schedule reflects tiling at one level of the hierarchy. Therefore, by linking the Cholesky's TRSM tile to a REPTILE-generated TRSM, the TRSM will be further decomposed into a smaller TRSM and GEMM. This flexibility is useful when, for example, the external library's TRSM routine is less optimized than its GEMM routine. By further decomposing the TRSM into the smaller TRSM and GEMM calls, and then mapping these two deeper calls to an external library, the resulting program spends less time in the slower external TRSM (vs the faster external GEMM) than if all four calls were mapped only at the first level of recursion. However, these additional decompositions come at the cost of making more calls to external functions, thus incurring additional overhead. In some cases, it is best not to map a tile to any function at all, if the overhead alone would degrade performance. REPTILE's scheduling language allows users to easily evaluate and benchmark all these options.

7.1 Function Capabilities and Mosaic

REPTILE, as outlined in Algorithm 3, maps tiles to external library functions (based on user specification). The mapping process is derived from the Mosaic tensor algebra compiler [Bansal et al. 2023]. In Mosaic, a user specifies what subexpressions from the input tensor algebra equation should be mapped to external functions. Each external function is pre-registered into the compiler with an associated tensor algebra equation, known as the function's capability, which mathematically describes the function's computation in the same language as the compiler's input language. During compile time, Mosaic checks that the input equation's specified subexpression and the function's capability equation have semantically equivalent ASTs. If so, Mosaic uses this function to calculate the subexpression, instead of calculating it with its own generated loops and compute statements.

Similarly, in REPTILE, both the input recurrences and function capabilities are defined in the same recurrence language. REPTILE currently includes eighteen external functions whose compute capabilities are defined in terms of the recurrence language, and users can easily add more by registering an external function with its associated function capability. Like Mosaic, REPTILE verifies the user-provided mappings are valid before generating calls to the external function. Unlike Mosaic, REPTILE handles non-rectangular iteration bounds, as iteration variables can be constrained with inequalities (e.g., $j<i$), creating triangular iteration bounds. Therefore, REPTILE also verifies that the iteration spaces of the mapped tile and function capability are equivalent. This is done by considering the iteration space inequalities as part of the input equation, such that the iteration space description becomes part of the equation's AST. This allows the AST equivalence check to also verify that the iteration spaces are equivalent.

7.2 Expression and Iteration Space Equivalence

An input recurrence and function capability may have equivalent ASTs but differently named tensors and variables. Therefore, to map a tile to an external function, the user describes how to map variable and tensor names from the function capability to the tile recurrence. This description is composed of:

(1) A tensor mapping σ_t : tensor $\rightarrow$ tensor that defines the correspondence between tensors in a tile's recurrences and the function's capability
(2) A variable mapping σ_v : iVar $\rightarrow$ iVar that matches index variables

Based on user-provided tensor and variable mappings, REPTILE verifies ASTs are equivalent to ensure that the external function calculates the desired input expression. REPTILE accomplishes this through simultaneous recursive traversal of both the tile's and function capability's ASTs, checking

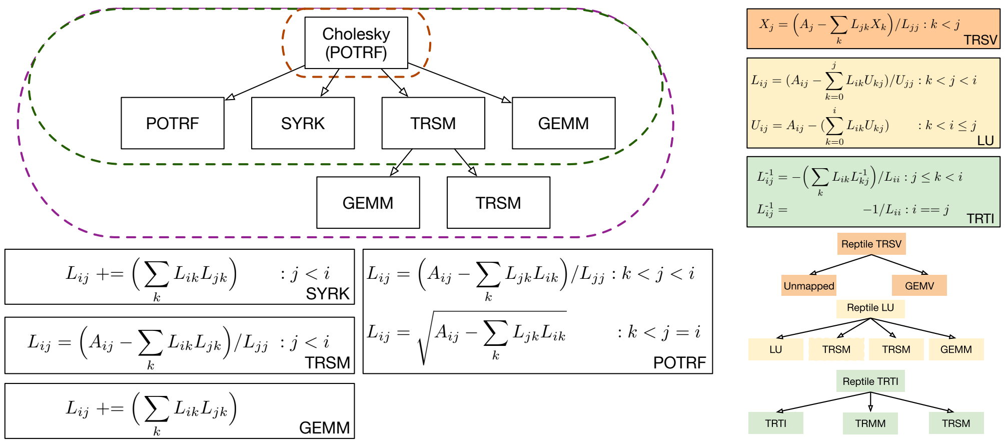

Cholesky (POTRF)
POTRF
SYRK
TRSM
GEMM
GEMM
TRSM
$L_{ij} += \left(\sum_k L_{ik}L_{jk}\right) \quad : j < i$ SYRK
$L_{ij} = \left(A_{ij} - \sum_k L_{ik}L_{jk}\right)/L_{jj} \quad : j < i$ TRSM
$L_{ij} += \left(\sum_k L_{ik}L_{jk}\right)$ GEMM
$L_{ij} = \left(A_{ij} - \sum_k L_{jk}L_{ik}\right)/L_{jj} : k < j < i$
$L_{ij} = \sqrt{A_{ij} - \sum_k L_{jk}L_{ik}} \quad : k < j = i$ POTRF
$X_j = \left(A_j - \sum_k L_{jk}X_k\right)/L_{jj} : k < j$ TRSV
$L_{ij} = (A_{ij} - \sum_{k=0}^{j} L_{ik}U_{kj})/U_{jj} : k < j < i$
$U_{ij} = A_{ij} - (\sum_{k=0}^{i} L_{ik}U_{kj}) \quad : k < i \leq j$ LU
$L^{-1}_{ij} = -\left(\sum_k L_{ik}L^{-1}_{kj}\right)/L_{ii} : j \leq k < i$
$L^{-1}_{ij} = \quad -1/L_{ii} : i == j$ TRTI
Reptile TRSV
Unmapped
GEMV
Reptile LU
LU
TRSM
TRSM
GEMM
Reptile TRTI
TRTI
TRMM
TRSM

Fig. 18. Left: Recursive decomposition of Cholesky recurrences into subroutines/tiles. Each tile can be further decomposed. We found GEMM/SYRK required no further decomposition for good performance, as multiplication routines are generally implemented well. However, further decomposition of TRSM into a TRSM and GEMM improves performance for large input sizes. We also show the subexpressions of the input Cholesky recurrences that map to each external call. Right: Decomposition of other evaluated recurrences.

the ASTs are equivalent, and that any differences in argument names to the AST operations are resolved by the tensor and variable maps. To illustrate, we observe how a user can map a tile from the Cholesky recurrences to an external TRSM kernel.

The tile mappable to a TRSM contains the following recurrence and iteration space:

$$\begin{gathered} L_{ij} = \frac{A_{ij} - \sum_k L_{jk}L_{ik}}{L_{ii}} \\ j_0 \leq k < j < j_1 \leq i < N \end{gathered} \tag{1}$$

REPTILE's external function capability for this TRSM is defined as follows:

$$\begin{gathered} X_{ri} = \frac{B_{ri} - \sum_k M_{ij}X_{rj}}{M_{ii}} \\ N_0 \leq j < i < N_1, \ N_2 \leq r < N_3 \end{gathered} \tag{2}$$

REPTILE verifies that the tile's recurrence equation and the external function capability are the same. Given the following user-provided tensor mappings σ_t and variable mappings σ_v, REPTILE applies the maps to the external function expression and then confirms the mapped external function expression is the same as the tile's recurrence expression:

$$\begin{aligned} &\sigma_t : \{X \to L, \ M \to L, \ B \to A \ \} \\ &\sigma_v : \{r \to i, \ i \to j, \ j \to k, \ N_0 \to j_0, \ N_1 \to j_1, \ N_2 \to j_1, \ N_3 \to N\} \end{aligned} \tag{3}$$

Under both of these maps in Equation 3, Equation 2 turns into Equation 1, thus verifying they represent a valid mapping. In user code, these maps are provided separately as list of arguments to a registered TRSM function, shown in Figure 19 and also shown in Figure 6.

In the code listing in Figure 19, 'lib' represents the external function registry. We note that external 'trsm' function is also registered with a function signature stored in 'lib[trsm].func', which defines all arguments to the function, specifically the relevant tensors and variables (including iteration bounds). The user-provided mapping from the tile to the function (defined in the last line in

```
# Register TRSM as an external function:
# Define its capability as a recurrence:
lib['trsm'].capability = Recurrence("X[r,i] = (B[r,i] - sigma(j,N0,i,M[i,j]*X[r,j]))/M[i,i]
               : N0<=j<i<N1, N2<=r<N3")
# Define its function signature:
lib['trsm'].func = "trsm([B,M,X], [r,i,j, N0,N1,N2,N3])"

# Using the registered signature, map B->A, M->L, X->L, and all variables and tile bounds
mapping = "trsm([A,L,L], [i,j,k, j0,j1,j1,N])" #RIN function
```

Fig. 19. User-provided schedule for mapping a triangular solve to the Cholesky decomposition.

the above code block) is itself defined like a call to this function, using the aforementioned signature, but with different arguments. In this way, the tensor and variable maps are constructed implicitly by recording how actual arguments in the mapping bind to the formal arguments defined in the registered function signature. Furthermore, this mapping exactly represents how the operation appears in the generated RIN: simply as a function call with the provided arguments, as defined in the RIN grammar in Figure 13.

8 C Code Generation

REPTILE is implemented in Python, and provides a user-friendly frontend for defining recurrence equations, constraints, and schedules as demonstrated in Figure 6. It first processes these inputs through the tiling, lowering, and mapping phases described in previous sections, ultimately generating the low-level Recurrence Index Notation (RIN) intermediate representation. This RIN is then lowered to executable C code. The RIN to C code generation phase in REPTILE builds upon the approaches used in the RECUMA compiler [Sundram et al. 2024]. This phase is a straightforward translation of the loop structures, arithmetic operations, and dependencies captured in the RIN into C code that can be directly compiled and executed. REPTILE also integrates external library calls within the generated C code. When a tile has been mapped to an external function, the RIN contains special nodes representing these function calls. During C code generation, these nodes are converted into appropriate external function calls with the correct parameters, tensor arguments, and memory layouts — each RIN function argument binds to a corresponding C function argument. Taking inspiration from Mosaic's foreign function interface [Bansal et al. 2023], REPTILE implements a framework that composes optimized MKL kernels with compiler-generated code.

9 Evaluation

We evaluate REPTILE kernels for five solvers and two genomics algorithms (Needleman-Wunsch and Smith Waterman which are discussed in section 9.6). The five solvers include Cholesky (POTRF), LU with no pivoting (GETRFNP), triangular solve (TRSV), triangular solve for multiple right-hand sides (TRSM), and lower triangular matrix inversion (TRTRI). Each of these exhibit data reuse and benefit from tiling. All solver kernels are evaluated against Intel MKL [Wang et al. 2014], and any external library calls made by REPTILE likewise call MKL kernels. For the two non-solver recurrences from bioinformatics (Needleman-Wunsch and Smith-Waterman), we evaluate their tiled and non-tiled performance against the Parasail library [Daily 2016]. REPTILE's compilation times for kernels reported in this paper are in Table 1.

All benchmarks were run on a 20-core Intel Xeon Silver CPU with a 50 MiB L3 cache, 5 MiB L2 cache, and 640 KiB L1 data cache. All REPTILE code was compiled with gcc 9.4 on Ubuntu. We leave computing LU with pivoting as future work, as it requires swapping rows. We evaluate how user-specified tiling direction, mapping choices, and tile sizes impact performance.

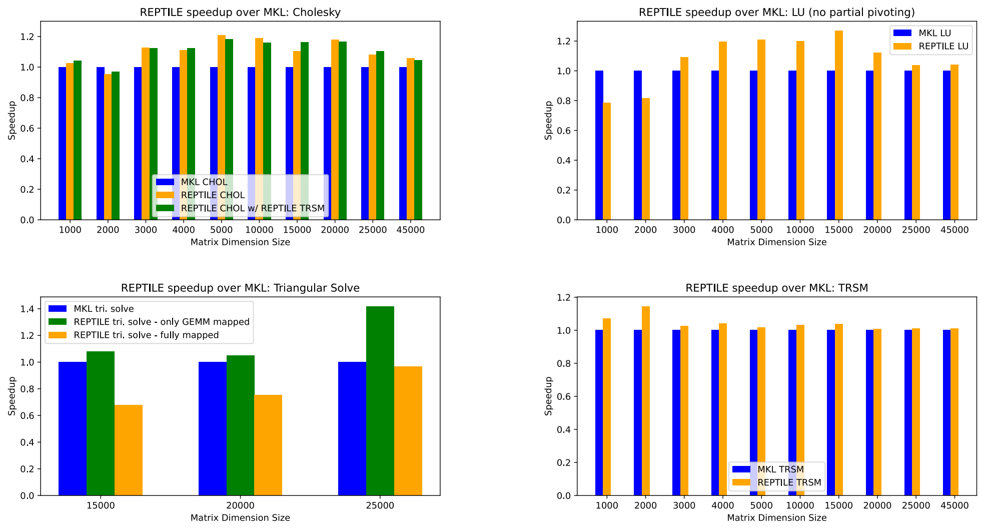


Fig. 20. Performance of REPTILE vs MKL. The Cholesky evaluation features two versions of a REPTILE Cholesky, one in which the four subroutines are each mapped to MKL calls, and one otherwise equivalent one, but where the TRSM subroutine is itself a REPTILE TRSM call, in which it is recursively decomposed into a TRSM and GEMM - the underlying GEMM and TRSM get mapped to an MKL call, illustrating a deeper recursive decomposition of the Choleksy routine. For the triangular solve (distinct from TRSM), which gets decomposed into a triangular solve and GEMM, it is better to map the GEMM subroutine to MKL and have the triangular solve subroutine be implemented with loops, as opposed to fully mapping both.

9.1 Mapping to External Handwritten Kernels

We generate REPTILE kernels for Cholesky decomposition, LU decomposition with no partial pivoting, and triangular solve with multiple right-hand sides (TRSM), and lower triangular matrix inversion (TRTRI). All of these are commonly used routines when solving systems of linear equations, and all mapping decisions are shown in Figure 18. For these four, we generated REPTILE kernels in which all computation is carried out by external functions. We evaluate these kernels for various input sizes, and for each input size, we also vary the inner REPTILE tile size. For each input, we choose the best performing tile size and show its speedup over the baseline MKL kernel in Figure 20. In all cases (except for the two smallest LU input sizes and one small Cholesky input size), we are able to find at least one tile size that makes the REPTILE kernel outperform the MKL LU routine. Notably this is accomplished by calling other MKL kernels for each tile. For the Cholesky, LU, and TRSM, we notice that as input sizes become extremely large, REPTILE's speedup over MKL approaches 1x. For Choleksy, the speedup peaks at n=5000, achieving 1.2x speedup over MKL. We postulate this is because this is around the point when data starts exceeding the capacity of the L2 cache, increasing cache misses. Meanwhile, Figure 21 shows that our TRTRI performance consistently outpaces MKL, even past n=5000, attaining up to 1.6x speedup. MKL is closed source so it is hard to pinpoint why, but TRTRI is a lesser used kernel, so it is likely just not as well optimized.

9.2 Mapping to other REPTILE kernels

We also generate a REPTILE kernel for the Cholesky decomposition, but where the TRSM external call is made to the aforementioned REPTILE-generated TRSM routine. Figure 18 shows that the Cholesky decomposition can be recursively decomposed into a smaller Cholesky, a TRSM, a SYRK, and a GEMM. In turn, the TRSM can be further decomposed into a smaller TRSM and a GEMM. First, we generated a REPTILE kernel for TRSM that calls MKL's TRSM and GEMM and then we generated a REPTILE kernel for Cholesky that used this REPTILE TRSM kernel. For some problem sizes, this resulted in better performance than the REPTILE kernel only mapped to MKL. We postulate this is because MKL's TRSM is not as well optimized as GEMM, in which case handing off and transferring multiply-add computations from the TRSM routine into the GEMM routine can improve performance depending on tile size, input size, and hardware characteristics.

All four of the subroutines can be decomposed, but decomposition of GEMM and SYRK (which is a GEMM variant) is generally not necessary as they are the most important and commonly used linear algebra subroutines, and are thus generally well tuned and well optimized.

9.3 Varying Tile Size

REPTILE lets users specify tile sizes (the length of each tile in the tiling direction), which is a critical optimization parameter. Notably, MKL (and many other BLAS/LAPACK libraries) does not let users choose tile sizes, as tile sizes are not an exposed part of the BLAS/LAPACK interface. For the TRTRI solver, the left side of Figure 21 shows that for a fixed input size and choice of tiling direction, tile size affects the performance, hitting peak performance at tilesize=512 for a column tiling, and at tilesize=1024 for a row tiling. When tile sizes get too small (e.g., 128 or smaller) and the number of tiles increases, performance expectedly degrades below MKL's. This likely due to overheads from having too many calls to external functions. Table 4 shows how cache miss rates and runtimes vary when changing the tile sizes for an unmapped Cholesky decomposition with n=4096. Here, tile sizes and cache miss rate form a U-shaped curve. Similarly, tile sizes and runtimes also make a U-shaped curve. The correlation coefficient between the miss rate and runtime is 0.97 out of 1, indicating a strong positive correlation; higher miss rates are associated with longer runtimes. The correlation coefficient for a Cholesky decomposition that maps to MKL kernels is .72. The positive but lower correlation makes sense; tile sizes still matter, but MKL has other optimization decisions hardcoded internally, which users cannot control.

Table 1. REPTILE's compilation times are small; analysis happens at the equational level, independent of matrix dimensions.

Kernel	Chol jki	Chol jki unmapped	LU kij	tri. Solve ji	TRSM jki	LU jik unmapped	TRTRI Col jik	TRTRI Row ijk	Smith Waterman	Needleman Wunsch
Time (s)	0.14	0.24	0.11	0.05	0.05	0.30	0.13	0.13	0.01	0.01

Table 2. The best tile size for each REPTILE Cholesky kernel, for each benchmarked input matrix size. In all cases, the number of column tiles is between one and five.

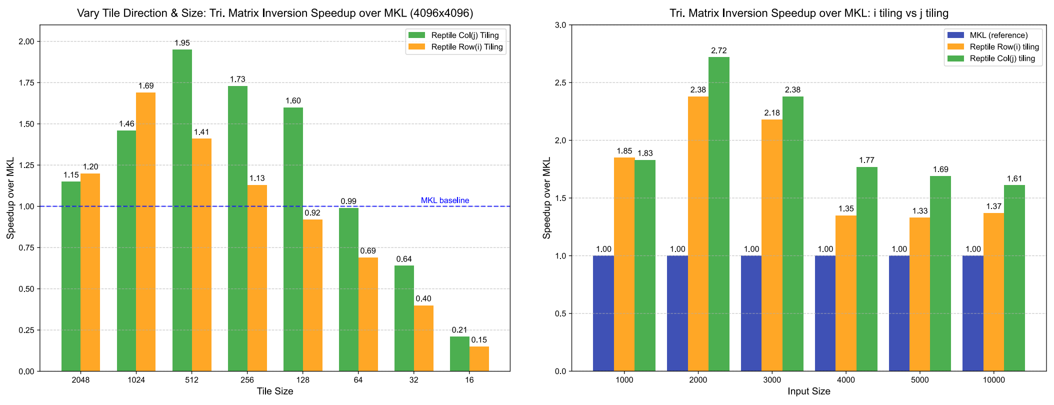
Matrix Dimension Size	1000	2000	3000	4000	5000	10000	15000	20000	25000	45000
REPTILE CHOL Best Tile Size	1000	2000	1500	800	1000	2000	3000	4000	2500	15000
REPTILE CHOL w/ TRSM Best Tile Size	1000	2000	1500	2000	2500	2000	3000	4000	12500	15000

Table 3. Unmapped REPTILE vs mapped REPTILE runtimes (s) for LU decomposition.

Matrix Size	REPTILE LU jik (unmapped)	REPTILE LU kij
1000	.4862	.0397
2000	4.09	.0332
3000	14.45	.0838
4000	52.135	.1221

Table 4. Tile size vs cache miss rate and runtime

Tile Size	**Cache Miss Rate**	**Runtime (s)**
32	9.77	2.352582
128	8.051	2.210000
256	5.815	2.123323
512	4.046	1.872756
1024	3.187	1.736880
2048	7.816	2.087795
4096 (untiled)	13.70	2.490000



(a) Varying tiling direction and size for fixed input.

(b) Scaling with input size.

Fig. 21. Triangular Matrix Inversion Speedup over MKL. (a) explores different tile sizes and tiling directions for a 4096x4096 matrix; (b) shows speedups with different matrix sizes.

For the Cholesky decomposition, for each input matrix size tested, the best tile size for our two Cholesky REPTILE kernels is shown in Table 2. For smaller inputs (n=1000 and n=2000), the best tested tile size was just the input dimension size, where there is only one column tile. We postulate this is because for these cases, the matrix is small enough to fit in the L3 cache and thus never gets evicted. For any larger inputs, the best tile size is one which results in multiple column tiles.

9.4 Choosing Tiling Direction

The tiling direction has a large impact on performance. For the TRTRI kernel, the best tiling direction depends on the input size. We generated REPTILE kernels for an i tiling and a j tiling, and compare their performance in Figure 21. Here the i tiling (i.e., row tiling) is better for smaller input sizes and larger tile sizes, whereas the j tiling is often better for larger inputs. For the largest sizes, they achieve 1.4x (row) and 1.6x (col) speedup respectively. We note that in both versions of the TRTRI kernel, they map to the same types of kernels: a TRTRI, a TRSM, and TRMM (triangular matrix multiply. However, they map to them in different ways; in each version, the kernels execute in different orders, and the TRSM and TRMM actually operate on different parts of the matrix.

Furthermore, some tiling directions may produce tiles that do not map to any commonly known kernels. For a k direction tiling of the LU decomposition, (e.g., kij or kji schedule), the tiles can be mapped to a LU, a GEMM, and two TRSM calls. For a i or j direction tiling, however, most of the tiles do not map (besides the small LU tile) to BLAS or LAPACK calls, so REPTILE will instead use its own generated loops to compute those tiles. Table 3 compares the performance of a jik schedule (j tiling, or column tiling) for LU, in which none of the tiles are mapped, and the performance of a kij schedule (where all tiles are mapped). The vast performance distance highlights the importance of choosing a good tiling direction to enable mappings.

9.5 Composing External Calls with Generated Loops

We generate a REPTILE kernel for the triangular solve, whoserecurrence is $X_i = (B_i - \sum_j L_{ij}X_j)/L_{ii} : j < i$, where B and L are inputs. The kernel is evaluated against the MKL triangular solve, and speedup results are shown in Figure 20. The triangular solve can be recursively decomposed into a smaller triangular solve and a matrix-vector multiply. We evaluated two REPTILE kernels: one in which both the smaller triangular solve and matrix-vector multiply are mapped to an MKL call, and

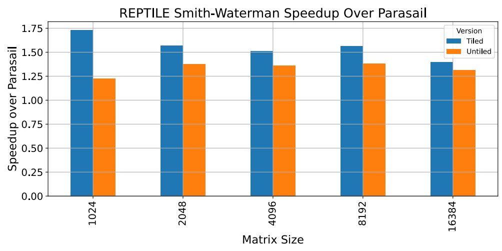


Fig. 22. REPTILE speedup: Smith-Waterman

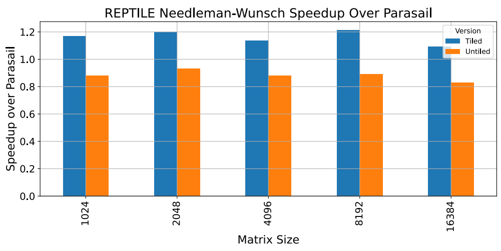


Fig. 23. REPTILE speedup: Needleman-Wunsch

one in which only the matrix-vector multiply is mapped. We find that the latter version, in which only the matrix-vector multiply is mapped, results in significantly better performance than both the MKL baseline and fully mapped REPTILE kernel. The highest speedup is 1.4x over the MKL baseline, potentially due to MKL's triangular solve kernel creating overhead and not being fully optimized. This highlights the importance of composing mapped kernels with raw compiler-generated loops.

9.6 Generating Tiled Loops without External Mapping

We generate a REPTILE kernel for the Needleman-Wunsch and Smith Waterman algorithms, two commonly used algorithms for aligning two strings of genomic sequences.

The associated recurrence for Needleman-Wunsch is

$$S_{ij} = Max(S_{i,j-1} + C1,\ S_{i-1,j} + C2, S_{i-1,j-1} + C3 * (A_i \neq B_i))$$

The associated recurrence for Smith-Waterman is

$$S_{ij} = Max(0, S_{i,j-1} + C1,\ S_{i-1,j} + C2, S_{i-1,j-1} + C3 * (A_i \neq B_j))$$

These problems are not compute bound, not linear solvers, and don't exhibit the same degree of data reuse. They still contain some data reuse, specifically the broadcasting of A_i and B_j, and can still benefit from tiling. Figure 22 and Figure 23 shows that our tiled REPTILE kernels (generated without external mappings) perform better than non-tiled kernels for both recurrences. We compare REPTILE's untiled and tiled kernels to Parasail, an optimized library [Daily 2016]. For Smith-Waterman, both REPTILE's tiled and untiled kernels outperform Parasail, whereas for Needleman-Wunsch, tiled kernels outperform Parasail, while untiled kernels are slower.

10 Related Work

The RECUMA compiler [Sundram et al. 2024] demonstrated a novel code generation algorithm for compiling recurrences to native code by managing dependencies within the equations. REPTILE is implemented in Python as a separate compiler from RECUMA, but that uses principles from RECUMA. REPTILE extends the work on recurrences by providing tiling capabilities and library mappings, while preserving the generality of a compiler. The recurrence input languages for both are the same, whereas REPTILE's scheduling language is different to allow for tiling and mapping.

Prior to RECUMA, Dyna [Eisner et al. 2004] used runtime analysis to track dependencies. While these systems offer general solutions for recurrences, they don't achieve the performance of specialized libraries. The polyhedral model [Feautrier 1991; Lamport 1974] is a well-known framework for optimizing and tiling dense loops, and it was originally applied to finite difference method recurrences [Karp et al. 1967]. TACO [Kjolstad et al. 2017; Kjølstad 2020] showed how tensor algebra expressions could be compiled to efficient code for different sparse tensor formats [Chou et al. 2018], with a scheduling language to control optimization [Kjolstad et al. 2019; Senanayake et al. 2020]. Following that, the Mosaic [Bansal et al. 2023] compiler made an extension to it by enabling mapping to external libraries. Methods for tiling loops, including ones implementing matrix decompositions have been studied by Yi and Kennedy [2002] and Carr and Kennedy [1994].

Libraries like BLAS [Dongarra et al. 1990; Lawson et al. 1979], LAPACK [Anderson et al. 1999], and ATLAS [Whaley et al. 2001] provide highly optimized implementations but only for a fixed set of operations. Certain LAPACK routines in various libraries use a recursive strategy, like the Cholesky routine in OpenBLAS [ope 2024]. However, these libraries do not let users adjust tiling sizes or directions. ScaLAPACK is a distributed memory version of LAPACK [Choi et al. 1992]. LINPACK preceded LAPACK and contained several solver implementations [Dongarra et al. 1979]. LINPACK is still used to benchmark supercomputers with a large LU decomposition[Dongarra et al. 2003]. COnfLUX and COnfCHOX are LU and Cholesky decomposition algorithms designed to minimize communication on supercomputers [Kwasniewski et al. 2021]. Similarly, minimizing communication in TRSM was studied by Wicky et al. [2017]. ParSEC is a task-based framework for distributed memory parallelism that has been used to run a Cholesky decomposition by launching a task for each of the four subroutines [Bosilca et al. 2013; Cao et al. 2024]. Different loop orderings of the Cholesky and LU decompositions were described by Ortega [1988a,b] and Menon et al. [2003]. The supernodal sparse Cholesky algorithm uses BLAS to factorize select sparse columns with similar sparsity patterns. Sympiler [Cheshmi et al. 2017, 2020] is an inspector-executor framework targeting supernodal Cholesky, triangular solve, and KKT systems. It applies low-level optimizations based on a given sparsity pattern. The Exo framework [Ikarashi et al. 2022] allows users to call external instructions, but it targets low-level instructions and doesn't replace whole sub-computations like REPTILE or Mosaic. The FLAME project [Bientinesi et al. 2005; Gunnels et al. 2001] introduced a formal methodology for deriving provably correct blocked linear algebra algorithms. While it emphasizes algorithm derivation, FLAME is not a compiler infrastructure. The BLIS [Van Zee et al. 2016; Van Zee and Van De Geijn 2015] library contains implementations of BLAS calls, underlying microkernels, and functionality for composing these subroutines as building blocks for other routines. PLASMA [Agullo et al. 2009; Bosilca et al. 2011] implements BLAS, LAPACK, and other linear algebra algorithms using optimized subroutines, but does not automatically tile. Similarly, ReLAPACK [Peise and Bientinesi 2016] provides recursive implementations of LAPACK algorithms, but these frameworks lack the generality to handle arbitrary recurrence equations.

11 Conclusion

This works presents REPTILE, a compiler that deploys a novel tiling strategy to tile user-provided recurrence equations and generates efficient code by composing compiler-generated loops with calls to external libraries. Our evaluation demonstrates that REPTILE can match or exceed the performance of hand-optimized libraries while maintaining the flexibility of a general recurrence compiler. Therefore, it helps us bridge the gap between general recurrence equation compilers, which offer flexibility but often lack performance, and optimized libraries, which provide high performance but only for specific problems. More broadly, this work lays the foundation for creating a compiler for recurrences that enables performance portability across different architectures and can hence target single node, heterogeneous, and distributed memory machines alike.

Data Availability Statement

The artifact for this paper includes the REPTILE compiler [Tariq et al. 2025]. The frontend is embedded in Python, and the compiler itself is a Python repository that compiles input recurrence equations and generates C code with externals calls to MKL. The artifact also includes scripts to reproduce the benchmarks in this paper, including the corresponding C kernels and the frontend code to generate them. Instructions for using the artifact are available via an archived version of the artifact on Zenodo. Benchmarking results may vary based on the hardware used.

Acknowledgments

We would like to thank our anonymous reviewers for their valuable feedback, comments, and insights on improving this paper. We thank James Dong, Benjamin Driscoll, Chris Gyurgyik, Olivisa Hsu, Scott Kovach, Rubens Lacouture, Devanshu Ladsaria, AJ Root, Marco Siracusa, Bala Vinaithirthan, and Rohan Yadav for their feedback on an early draft. We also wish to thank Alex Aiken for his advice and feedback on this work. 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.

References

Accessed 2024. OpenBLAS: An Optimized BLAS Library. https://www.openblas.net/. Accessed: 2024-11-15.

Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180 (08 2009), 012037. https://doi.org/10.1088/1742-6596/180/1/012037

Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK users' guide.* SIAM.

Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proc. ACM Program. Lang.* 7, PLDI, Article 122 (June 2023), 26 pages. https://doi.org/10.1145/3591236

Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. 2005. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Softw.* 31, 1 (March 2005), 1–26. https://doi.org/10.1145/1055531.1055532

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).

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. 2011. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum.* IEEE, 1432–1441.

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

Qinglei Cao, Thomas Herault, Aurelien Bouteiller, Joseph Schuchart, and George Bosilca. 2024. Evaluating PaRSEC Through Matrix Computations in Scientific Applications. In *Workshop on Asynchronous Many-Task Systems and Applications.* Springer, 22–33.

Steve Carr and Ken Kennedy. 1994. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1768–1810. https://doi.org/10.1145/197320.197366

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

Kazem Cheshmi, Danny M. Kaufman, Shoaib Kamil, and Maryam Mehri Dehnavi. 2020. NASOQ: Numerically Accurate Sparsity-Oriented QP Solver. *ACM Transactions on Graphics* 39, 4 (2020).

J. Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. 1992. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. In *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation.* 120–127. https://doi.org/10.1109/FMPC.1992.234898

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (October 2018), 30 pages.

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

J. J. Dongarra, Jermey Du Cruz, Sven Hammarling, and I. S. Duff. 1990. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 18–28. https://doi.org/10.1145/77626.77627

Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.

Jack J Dongarra, Cleve Barry Moler, James R Bunch, and Gilbert W Stewart. 1979. *LINPACK users' guide.* SIAM.

Jason Eisner, Eric Goldlust, and Noah A Smith. 2004. Dyna: A declarative language for implementing dynamic programs. In *Proc. of ACL.*

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

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

Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 703–718.

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.

Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 180–192. https://doi.org/10.1109/CGO.2019.8661185

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

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

Grzegorz Kwasniewski, Marko Kabic, Tal Ben-Nun, Alexandros Nikolaos Ziogas, Jens Eirik Saethre, André Gaillard, Timo Schneider, Maciej Besta, Anton Kozhevnikov, Joost VandeVondele, and Torsten Hoefler. 2021. On the parallel I/O optimality of linear algebra kernels: near-optimal matrix factorizations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21).* Association for Computing Machinery, New York, NY, USA, Article 70, 15 pages. https://doi.org/10.1145/3458817.3476167

Leslie Lamport. 1974. The Parallel Execution of DO Loops. *Commun. ACM* 17, 2 (1974), 83–93. http://research.microsoft.com/en-us/um/people/lamport/pubs/do-loops.pdf

Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.

Vijay Menon, Keshav Pingali, and Nikolay Mateev. 2003. Fractal symbolic analysis. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 776–813. https://doi.org/10.1145/945885.945888

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.

E Peise and P Bientinesi. 2016. Recursive algorithms for dense linear algebra: the ReLAPACK collection. ArXiv e-prints (Feb. *arXiv preprint cs.MS/1602.06763* (2016).

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

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

Muhammad Usman Tariq, Shiv Sundram, and Fredrik Kjolstad. 2025. *REPTILE artifact.* https://doi.org/10.5281/zenodo.15761691

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

Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. 2016. The BLIS Framework: Experiments in Portability. *ACM Trans. Math. Softw.* 42, 2, Article 12 (June 2016), 19 pages. https://doi.org/10.1145/2755561

Field G Van Zee and Robert A Van De Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)* 41, 3 (2015), 1–33.

Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al. 2014. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures* (2014), 167–188.

R Clint Whaley, Antoine Petitet, and Jack J Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel computing* 27, 1-2 (2001), 3–35.

Tobias Wicky, Edgar Solomonik, and Torsten Hoefler. 2017. Communication-Avoiding Parallel Algorithms for Solving Triangular Systems of Linear Equations. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 678–687. https://doi.org/10.1109/IPDPS.2017.104

Qing Yi and Ken Kennedy. 2002. *Transforming complex loop nests for locality.* Ph. D. Dissertation. USA. AAI3047379.

Received 2025-03-25; accepted 2025-08-12