

Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis

Kazem Cheshmi
Rutgers University
Piscataway, NJ, US
kazem.ch@rutgers.edu

Michelle Mills Strout
University of Arizona
Tucson, AZ, US
mstrout@cs.arizona.edu

Shoaib Kamil
Adobe Research
New York, NY, US
kamil@adobe.com

Maryam Mehri Dehnavi
Rutgers University
Piscataway, NJ, US
maryam.mehri@rutgers.edu

ABSTRACT

Sympiler is a domain-specific code generator that optimizes sparse matrix computations by decoupling the symbolic analysis phase from the numerical manipulation stage in sparse codes. The computation patterns in sparse numerical methods are guided by the input sparsity structure and the sparse algorithm itself. In many real-world simulations, the sparsity pattern changes little or not at all. Sympiler takes advantage of these properties to symbolically analyze sparse codes at compile time and to apply inspector-guided transformations that enable applying low-level transformations to sparse codes. As a result, the Sympiler-generated code outperforms highly-optimized matrix factorization codes from commonly-used specialized libraries, obtaining average speedups over Eigen and CHOLMOD of 3.8 \times and 1.5 \times respectively.

CCS CONCEPTS

• **Software and its engineering** \rightarrow **Source code generation**; • **Computing methodologies** \rightarrow *Parallel programming languages*;

KEYWORDS

Matrix computations, sparse methods, loop transformations, domain-specific compilation

ACM Reference format:

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017*, 13 pages.
DOI: 10.1145/3126908.3126936

1 INTRODUCTION

Sparse matrix computations are at the heart of many scientific applications and data analytics codes. The performance and efficient

memory usage of these codes depends heavily on their use of specialized sparse matrix data structures that only store the nonzero entries. However, such compaction is done using index arrays that result in indirect array accesses. Due to these indirect array accesses, it is difficult to apply conventional compiler optimizations such as tiling and vectorization even for static index array operations like sparse matrix vector multiply. A static index array does not change during the algorithm; for more complex operations with dynamic index arrays such as matrix factorization and decomposition, the nonzero structure is modified during the computation, making conventional compiler optimization approaches even more difficult to apply.

The most common approach to accelerating sparse matrix computations is to identify a specialized library that provides a manually-tuned implementation of the specific sparse matrix routine. A large number of sparse libraries are available (e.g., SuperLU [22], MUMPS [2], CHOLMOD [11], KLU [20], UMFPACK [15]) for different numerical kernels, supported architectures, and specific kinds of matrices. While hand-written specialized libraries can provide high performance, they must be manually ported to new architectures and may stagnate as architectural advances continue. Alternatively, compilers can be used to optimize code while providing architecture portability. However, indirect accesses and the resulting complex dependence structure run into limitations of compile-time loop transformation frameworks.

Compiler loop transformation frameworks such as those based on the polyhedral model use algebraic representations of loop nests to transform code and successfully generate highly-efficient dense matrix kernels [5, 10, 40, 55, 67, 69]. However, such frameworks are limited when dealing with non-affine loop bounds and/or array subscripts, both of which arise in sparse codes. Recent work has extended polyhedral methods to effectively operate on kernels with static index arrays. These work build run-time *inspectors* to examine the nonzero structure and use *executors* to transform code execution [65, 68, 70–72]. However, these techniques are limited to transforming sparse kernels with static index arrays. Sympiler addresses these limitations by performing *symbolic analysis* at compile time to specialize the code for the nonzero pattern whereas the inspector-executor approaches can only reorder data and schedules. Symbolic analysis is a term from the numerical computing community; it uses the nonzero pattern of the sparse matrix to analyze computation patterns. Information from symbolic analysis can be

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

SC17, Denver, CO, USA

© 2017 ACM. 978-1-4503-5114-0/17/11...\$15.00

DOI: 10.1145/3126908.3126936

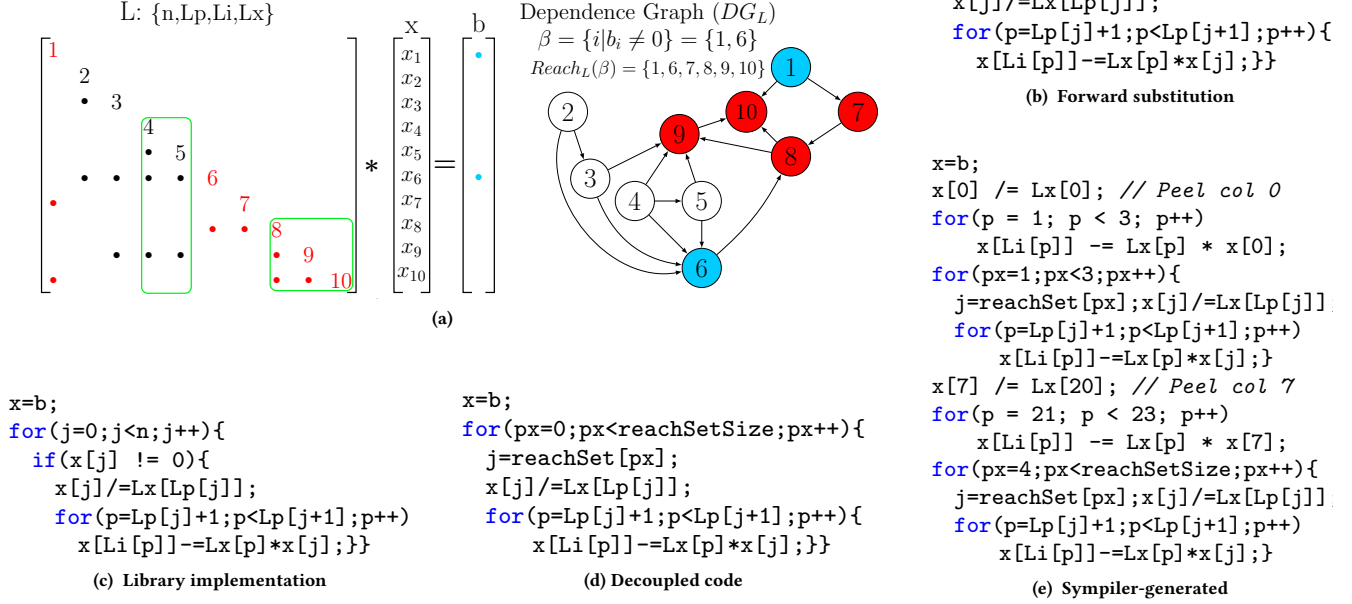


Figure 1: Four different codes for solving the linear system in (a). In all four code variants, matrix L is stored in compressed sparse column (CSC) format, with $\{n, Lp, Li, Lx\}$ representing {matrix order, column pointer, row index, nonzeros} respectively. The dependence graph DG_L is the adjacency graph of matrix L ; vertices of DG_L correspond to columns of L and its edges show dependencies between columns in triangular solve. Vertices corresponding to nonzero columns are colored in blue and columns that participate in the computation because of the dependence structure are in red. Boxes around columns show supernodes of different sizes. (b) is a forward substitution algorithm. (c) is a library implementation that skips iterations when the corresponding entry in x is zero. (d) is the decoupled code that uses the symbolic information provided by the $reachSet$, which is computed by performing a depth-first search on DG_L . (e) is the Sympiler-generated code which peels iterations corresponding to the columns inside the reach-set with more than 2 nonzeros.

used to make subsequent numeric manipulation faster, and can be reused as long as the matrix nonzero structure remains constant.

For a number of sparse matrix methods such as LU and Cholesky, it is well known that viewing their computations as a graph (e.g., elimination tree, dependence graph, or quotient graph) and applying a method-dependent graph algorithm yields information about dependencies that can then be used to more efficiently compute the numerical method [14]. Most high-performance sparse matrix computation libraries utilize symbolic information, but couple this symbolic analysis with numeric computation, further making it difficult for compilers to optimize such codes.

This work presents Sympiler, a domain-specific code generator which produces high-performance sparse matrix code. Sympiler fully decouples symbolic analysis from numeric computation and transforms the sparse code using symbolic information. The symbolic information is obtained using a symbolic inspector. Inspector-guided transformations, such as variable-sized blocking, are then applied resulting in performance equivalent to hand-tuned libraries. But Sympiler goes further than existing numerical libraries by generating code for a specific matrix nonzero structure. Because the matrix structure often arises from properties of the underlying physical system that the matrix represents, in many cases the same structure reoccurs multiple times, with different values of nonzeros. Thus, Sympiler-generated code can combine inspector-guided and

low-level transformations to produce even more efficient code. The transformations applied by Sympiler improve the performance of sparse matrix codes through applying single-core optimizations such as vectorization and increasing data locality, which should extend to improve performance on shared and distributed memory systems.

1.1 Motivating Scenario

The sparse triangular solve takes a lower triangular matrix L and a right-hand side (RHS) vector b and solves the linear equation $Lx = b$ for x . It is a fundamental building block in many numerical algorithms such as factorization [14, 43], direct system solvers [13], and rank update methods [18], where the RHS vector is often sparse. A naïve implementation visits every column of the matrix L to propagate the contributions of its corresponding x value to the rest of x (see Figure 1b). However, when b is sparse the solution vector is also sparse which can reduce the iteration space of the sparse triangular solve. The reduced iteration space is proportional to the number of nonzero values in x . To benefit from this property, the nonzero pattern of x has to be computed. Based on a theorem from Gilbert and Peierls [31], the dependence graph $DG_L = (V, E)$ for matrix L with nodes $V = \{1, \dots, n\}$ and edges $E = \{(j, i) | L_{ij} \neq 0\}$ can be used to compute the nonzero pattern of x , where n is the matrix rank and numerical cancellation is neglected. The nonzero

indices in x are given by $Reach_L(\beta)$ which is the set of all nodes reachable from any node in $\beta = \{i | b_i \neq 0\}$ and is computed by performing a depth-first search on the directed graph DG_L starting with β . An example dependence graph is illustrated in Figure 1a. The blue colored nodes correspond to the set β and the final $reach-set$ $Reach_L(\beta)$ contains all the colored nodes.

Figure 1 shows four different implementations of sparse triangular solve. All solvers shown in Figure 1 assume the input matrix L is stored in a compressed sparse column (CSC) storage format. While the naive implementation in Figure 1b traverses all columns, the typical library implementation shown in Figure 1c skips iterations when the corresponding value in x is zero.

The implementation in Figure 1d shows a decoupled code that uses the symbolic information provided by the precomputed reach-set. This decoupling simplifies numerical manipulation and reduces the run-time complexity from $O(|b| + n + f)$ in Figure 1c to $O(|b| + f)$ in Figure 1d, where f is the number of floating point operations and $|b|$ is the number of nonzeros in b . Sympiler goes further by building the reach-set at compile time and using it to generate code specialized for the specific matrix structure and the RHS. The Sympiler-generated code is shown in Figure 1e, where the code only iterates over reached columns and peels iterations where the number of nonzeros in a column is greater than some threshold (in the figure this threshold is 2). These peeled loops can be further transformed with vectorization to speed up execution. This shows the power of fully decoupling the symbolic analysis phase from the code that manipulates numeric values; the compiler aggressively applies conventional optimizations using the reach-set to guide the transformations. On matrices from the Florida Sparse Matrix Collection [19], the Sympiler-generated code shows speedups between 8.4× to 19× with an average of 13.6× compared to the forward solve code (Figure 1b) and from 1.2× to 1.7× with an average of 1.3× compared to the library-equivalent code (Figure 1c).

1.2 Static Sparsity Patterns

Sympiler takes advantage of the fundamental concept that the structure of sparse matrices in scientific codes is dictated by the physical domain and as such does not change in many applications. This structure often arises from the physical topology of the underlying system, the discretization, and the governing equations. These remain unchanged for long periods in simulations across many domains. Some examples include: (i) solving nonlinear time-dependent differential equations, where the Jacobian matrix has a static sparsity pattern for each time point while the numerical values change (example domains include fluid dynamics [45] and electromagnetics [24, 47]); (ii) design problems where values or parameters are chosen to maximize some measure of performance; (examples include computer animation [7, 56]); (iii) domains where the sparse matrix is assembled from a physical topology such as power system modeling; (iv) controlling rigid multibody movements in robotics applications where a sequence of linear systems needs to be solved for a static input [52]; and (v) simulations where the sparse stiffness matrix is assembled using a discretized mesh and governing equations and remains static for long periods (e.g. aerospace and electromagnetic simulations).

1.3 Contributions

This work describes Sympiler, a sparsity-aware code generator for sparse matrix algorithms that leverages symbolic information to generate fast code for a specific matrix structure. Major contributions of this paper are:

- A novel approach for building *compile-time symbolic inspectors* that obtain information about a sparse matrix for use during compilation.
- *Inspector-guided transformations* that leverage compile-time information to transform sparse matrix code for specific algorithms.
- Implementations of symbolic inspectors and inspector-guided transformations for two algorithms, namely the sparse triangular solve and the sparse Cholesky factorization.
- A demonstration of the performance impact of our code generator, showing that Sympiler-generated code outperforms state-of-the-art libraries for triangular solve and Cholesky factorization by up to 1.7× and 6.3× respectively.

2 SYMPILER: A SYMBOLIC-ENABLED CODE GENERATOR

Sympiler generates efficient sparse kernels by tailoring sparse code to specific matrix sparsity structures. By decoupling the symbolic analysis phase, Sympiler uses information from symbolic analysis to guide code generation for the numerical manipulation phase of the kernel. In this section, we describe the overall structure of the Sympiler code generator, as well as the domain-specific transformations enabled by leveraging information from the symbolic inspector.

2.1 Sympiler Overview

Sympiler currently supports sparse triangular solve and Cholesky factorization. Given one of these numerical methods and an input matrix stored using the compressed sparse column (CSC) format, Sympiler utilizes a method-specific *symbolic inspector* to obtain information about the matrix. Sample code that uses Sympiler is shown in Figure 2a, specifying the input matrix and numerical method. The numerical solver is internally represented using a domain-specific abstract syntax tree (AST) which is annotated with potential transformations. The annotated information is used to apply domain-specific optimizations while lowering the code for the numerical method. In addition, the lowered code is annotated with additional low-level transformations (such as unrolling) when applicable based on domain- and matrix-specific information. Finally, the annotated code is further lowered to apply low-level optimizations and output to C source code.

2.2 Symbolic Inspector

Different numerical algorithms can make use of symbolic information in different ways. Prior work has described run-time graph traversal strategies for various numerical methods [12, 14, 44, 53]. The compile-time inspectors in Sympiler are based on these strategies. For each class of numerical algorithms with the same symbolic analysis approach, Sympiler uses a specific symbolic inspector to

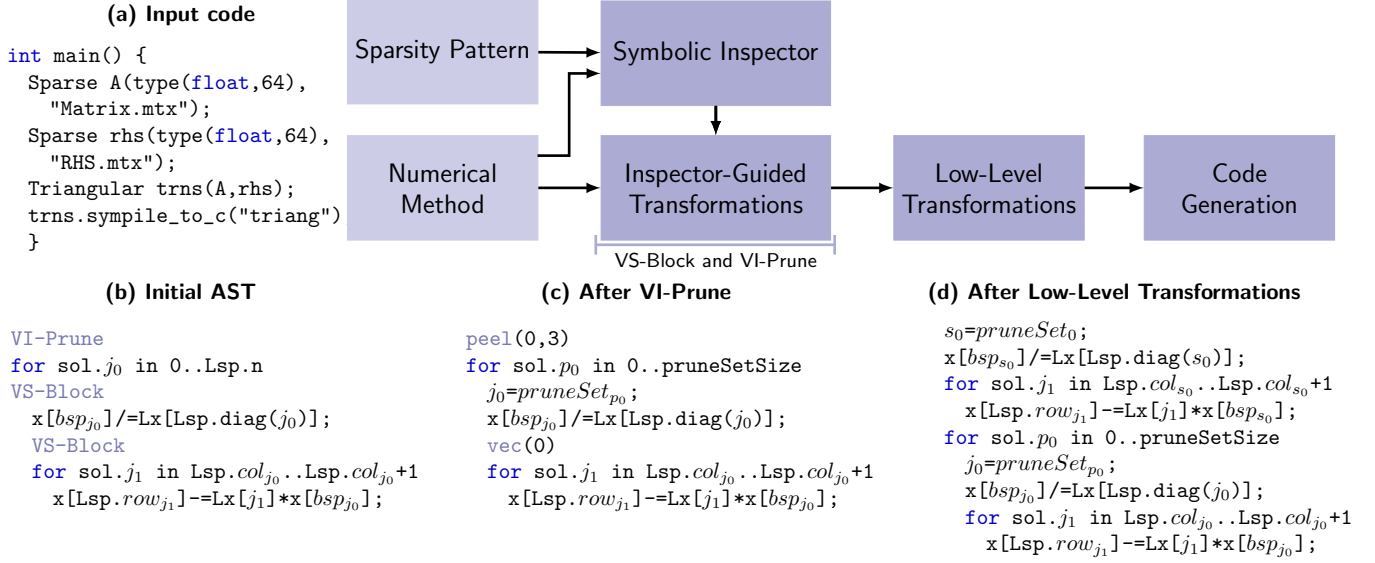


Figure 2: Sympiler lowers a functional representation of a sparse kernel to imperative code using the inspection sets. It constructs a set of loop nests and annotates them with domain-specific information that is later used in inspector-guided transformations. The inspector-guided transformations use the lowered code and inspection sets as input and apply transformations. Inspector-guided transformations also provide hints for low-level transformations by annotating the code. For instance, the transformation steps for the code in Figure 1 are: (a) Sympiler input code describing input matrices as well as the numerical method; (b) The initial AST with annotations showing where the VI-Prune and VS-Block transformations apply; (c) The transformed code after VI-Prune which has used the pruneSet to add low-level transformation hints such as peeling iterations 0 and 3; (d) The final code where hinted low-level transformations are applied (peeling is only shown for iteration zero).

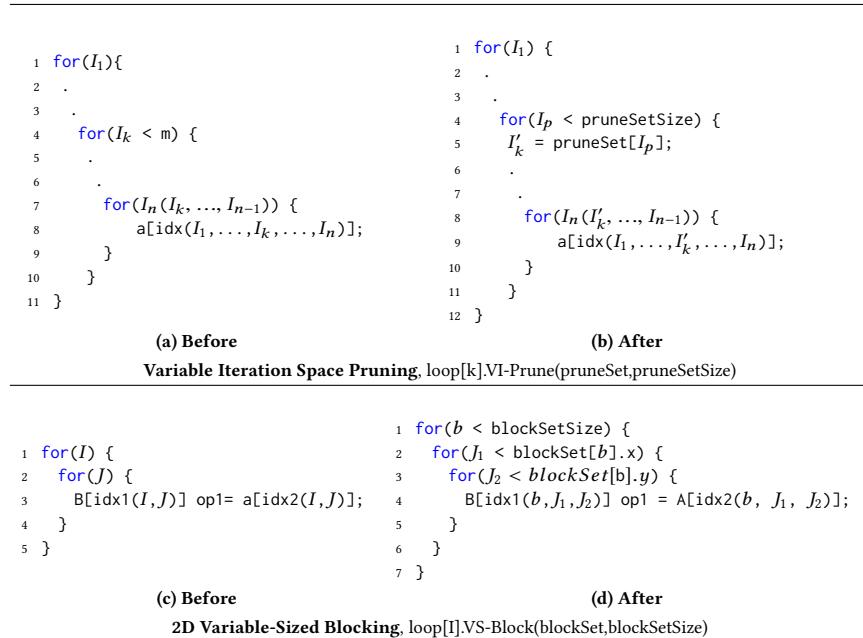


Figure 3: The inspector-guided transformations. Top: The loop over I_k with iteration space m in (a) transforms to a loop over I_p with iteration space $pruneSetSize$ in (b). Any use of the original loop index I_k is replaced with its corresponding value from $pruneSet$ i.e., I'_k . Bottom: The two nested loops in (c) are transformed into loops over variable-sized blocks in (d).

obtain information about the sparsity structure of the input matrix and stores it in an algorithm-specific way to be used in the transformation stages.

We classify symbolic inspectors based on the numerical method as well as the transformations enabled by the obtained information. For each combination of algorithm and transformation, the symbolic inspector creates an *inspection graph* from the given sparsity pattern and traverses it during inspection using a specific *inspection strategy*. By running the inspector on the inspection graph *inspection sets* are generated. Inspection sets are used to guide transformations in Sympiler.

For our motivating example, triangular solve, the *reach-set* can be used to prune loop iterations that perform work made unnecessary due to the sparseness of the matrix or the right hand side. In this case, the inspection set is the reach-set and the inspection strategy is to perform a depth-first search over the inspection graph, which is the directed dependency graph DG_L of the triangular matrix. For the example linear system shown in Figure 1, the symbolic inspector generates the reach-set $\{6, 1, 7, 8, 9, 10\}$.

2.3 Inspector-guided Transformations

The initial lowered code along with inspection sets obtained by the symbolic inspector go through a series of passes that further transform the code. Sympiler currently supports two transformations guided by inspection sets, namely *Variable Iteration Space Pruning* and *2D Variable-Sized Blocking*, which can be applied independently or jointly depending on the input sparsity. As shown in Figure 2b, the code is internally annotated with information showing where inspector-guided transformations may be applied. The symbolic inspector provides the required information to the transformation phases, which decide whether to transform the code based on the inspection sets. Given the inspection set and annotated code, transformations occur as illustrated in Figure 3.

2.3.1 Variable Iteration Space Pruning. Variable Iteration Space Pruning (VI-Prune) prunes the iteration space of a loop using information about the sparse computation. The iteration space for sparse codes can be considerably smaller than that of dense codes, since only iterations with nonzeros are computed. The inspection stage of Sympiler generates an inspection set that enables transforming the unoptimized sparse code to a code with a reduced iteration space.

Given this inspection set, VI-Prune is applied at a particular loop-level to transform the code from Figure 3a to Figure 3b. In the figure, the transformation is applied to the k^{th} loop nest in line 4. In the transformed code the iteration space is pruned to `pruneSetSize`, which is the inspection set size. In addition to the new loop, all references to I_k are replaced by its corresponding value from the inspection set, `pruneSet[Ip]`. Furthermore, the transformation phase uses the inspection set information to annotate certain loops with low-level optimizations. These low-level transformations are applied in subsequent stages of code generation and are guided by tunable thresholds to generate faster code.

In our running triangular solve example, the VI-Prune transformation elides unnecessary iterations due to zeros in the right hand side. In addition, depending on the number of iterations the loops will run (which is known thanks to the symbolic inspector), loops

are annotated with directives to unroll and/or vectorize during code generation.

2.3.2 2D Variable-Sized Blocking. 2D Variable-Sized Blocking (VS-Block) converts a sparse code to a set of non-uniform dense sub-kernels. In contrast to the conventional approach of blocking/tiling dense codes, where the input and computations are blocked into smaller uniform sub-kernels, the unstructured computations and inputs in sparse kernels make blocking optimizations challenging. The symbolic inspector identifies sub-kernels with similar structure in the sparse matrix methods and sparse inputs to find “blockable” sets that are not necessarily of the same size or consecutively located. These blocks are similar to the concept of *supernodes* [43] in sparse libraries. VS-Block must deal with a number of challenges:

- The block sizes are variable in a sparse kernel.
- Because of using compressed storage formats, the block elements may not be in consecutive memory locations.
- The type of numerical method used may have to change after applying this transformation. For example, to apply VS-Block to sparse Cholesky code, a dense Cholesky factorization has to be applied to the diagonal segment of the blocks and the off-diagonal segments need to be updated with dense triangular solves.

To address the first challenge, the symbolic inspector provides an inspection set which specifies the size of each block. For the second challenge, the transformed code allocates temporary block storage and copies data as needed prior to operating on the block. Finally, to deal with the last challenge, the synthesized loops/instructions in the lowering phase will contain information about the block location in the matrix and the correct operation is chosen for each loop/instruction when applying the transformation. Similar to VI-Prune, VS-Block also annotates loops with low-level transformations such as tiling for the code generation phase. By leveraging specific information about the matrix when applying the transformation, Sympiler is able to apply VS-Block to sparse numerical methods.

An off-diagonal version of the VS-Block transformation is shown in Figures 3c and 3d. A new outer loop is created. This outer loop provides block information to the inner loops using the *blockSet*. The inner loop in Figure 3c is transformed to two nested loops (lines 2–6) that iterate over the block specified by the outer loop. In line 3 of Figure 3c, the vector a is operated on and the result is stored in matrix B . After the VS-Block transformation, this vector operation is converted to a matrix-matrix operation as shown in line 4 of Figure 3d. Indices J_1 and J_2 are used to access a particular block in matrix A . Examples of applying VS-Block to triangular solve and Cholesky factorization are provided in Section 3.

2.4 Enabled Conventional Low-level Transformations

While applying inspector-guided transformations, the original loop nests are transformed into new loops with potentially different iteration spaces, enabling the application of conventional low-level transformations. Based on the applied inspector-guided transformations as well as the properties of the input matrix and the right-hand side vectors, the code is annotated with transformation directives.

```

1 for(column j = 0 to n){
2   f = A(:,j)
3   PruneSet = The sparsity pattern of row j
4   for(every row r in PruneSet){ // Update
5     f -= L(j:n,r) * L(j,r);
6   }
7   L(k,k) = sqrt(f(k)); // Diagonal
8   for(off-diagonal elements in f){ // Off-diagonal
9     L(k+1:n,k) = f(k+1:n) / L(k,k);
10  }
11 }

```

Figure 4: Pseudo-code of left-looking Cholesky.

An example of these annotations is shown in Figure 2c where loop peeling is annotated within the VI-Pruned code. To decide when to add these annotations, the inspector-guided transformations use sparsity-related parameters such as the average block size. Following lists sources that enable Low-level transformations:

- (1) Symbolic information provides dependency information at compile time allowing Sympiler to apply more transformations such as peeling based on the reach-set in Figure 1;
- (2) Inspector-guided transformations remove some of the indirect memory accesses and annotate the code with potential conventional transformations;
- (3) Sparsity-specific code generation provides Sympiler with information such as loop boundaries at compile time. As a result, several customized transformations are applied such as vectorization of loops with iteration counts greater than a threshold.

Figure 1e demonstrates the process in which iterations in the triangular solve code after VI-Prune are peeled. In this example, the inspection set used for VI-Prune is the reach-set $\{1, 6, 8, 9, 10\}$. Because the reach-set is created in topological order, iteration ordering dependencies are met and thus code correctness is guaranteed after loop peeling. As shown in Figure 2c, the transformed code after VI-Prune is annotated with the enabled peeling transformation based on the number of nonzeros in the columns (the *column count*). The two selected iterations with column count greater than two are peeled. The peeled iterations are either replaced with a specialized kernel or another transformation such as vectorization is applied to them.

3 CASE STUDIES

Sympiler currently supports two important sparse matrix computations, namely the triangular solve and Cholesky factorization. This section discusses some of the graph theory and algorithms used in Sympiler’s symbolic inspector to extract inspection sets for these two matrix methods. The run-time complexity of the symbolic inspector is also presented to evaluate inspection overheads. Finally, we demonstrate the process of applying the VI-Prune and VS-Block transformations using the inspection sets. Sympiler’s extension to other matrix methods is also discussed.

Table 1 shows a classification of the inspection graphs, inspection strategies, and resulting inspection sets for the two studied

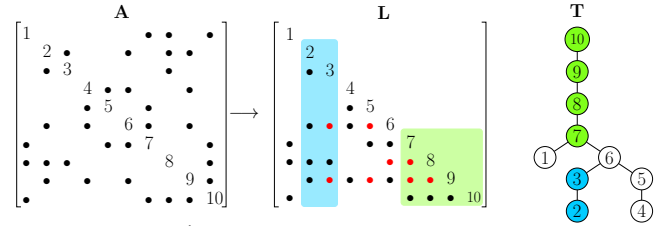


Figure 5: An example matrix A and its L factor from Cholesky factorization. The corresponding elimination tree (T) of A is also shown. Nodes in T and columns in L highlighted with the same color belong to the same supernode. The red nonzeros in L are fill-ins.

numerical algorithms in Sympiler. As shown in Table 1, the symbolic inspector performs a set of known inspection methods and generates sets which include symbolic information. The last column of Table 1 shows the list of low-level transformations enabled by each inspector-guided transformation.

3.1 Sparse Triangular Solve

Theory: The symbolic inspector traverses the dependency graph DG_L using depth-first search (DFS) to determine the inspection set for the VI-Prune transformation, which in this case is the reach-set from DG_L and the right-hand side vector. The graph DG_L is also used to detect blocks with similar sparsity patterns, also known as supernodes, in sparse triangular solve. The block-set, which contains columns of L grouped into supernodes, is identified by inspecting DG_L using a node equivalence method. The node equivalence algorithm initially assumes nodes v_i and v_j are equivalent and compares their outgoing edges. If the outgoing edges point to the same destination nodes then the two nodes are equal and are merged.

Inspector-guided Transformations: Using the reach-set, VI-Prune limits the iteration space of the loops in triangular solve to only those that operate on relevant nonzeros. The VS-Block transformation changes the loops in triangular solve to apply blocking as shown in Figure 2b. The diagonal block of each column-block, which is a small triangular solve, is solved first and its solution replaces the off-diagonal segment of the matrix.

Symbolic Inspection: The time complexity of DFS on graph DG_L is proportional to the number of edges traversed and the number of nonzeros in the RHS of the system. The time complexity for the node equivalence algorithm is proportional to the number of nonzeros in L . We provide overheads for these methods for the tested matrices in Section 4.3.

3.2 Cholesky Factorization

Cholesky factorization is commonly used in direct solvers and is used to precondition iterative solvers. The algorithm factors a Hermitian positive definite matrix A into LL^T , where matrix L is a lower triangular matrix. Figure 5 shows an example matrix A and the corresponding L matrix after factorization.

Theory: The elimination tree (etree) [17] is one of the most important graph structures used in the symbolic analysis of sparse

Table 1: Inspection and transformation elements in Sympiler for triangular solve and Cholesky. DG: dependency graph, SP (RHS): sparsity patterns of the right-hand side vector, DFS: depth-first search, SP(A): sparsity patterns of the coefficient A , SP(L_j): sparsity patterns of the j^{th} row of L , unroll: loop unrolling, peel: loop peeling, dist: loop distribution, tile: loop tiling.

Transformations	Triangular Solve			Cholesky			Enabled Low-level
	Inspection Graph	Inspection Strategy	Inspection Set	Inspection Graph	Inspection Strategy	Inspection Set	
VI-Prune	DG + SP(RHS)	DFS	Prune-set (reach-set)	etree + SP(A)	Single-node up-traversal	Prune-set (SP(L_j))	dist, unroll, peel, vectorization
VS-Block	DG	Node equivalence	Block-set (supernodes)	etree + ColCount(L)	Up-traversal	Block-set (supernodes)	tile, unroll, peel, vectorization

factorization algorithms. Figure 5 shows the corresponding elimination tree for factorizing matrix A . The etree of A is a spanning tree of $G^+(A)$ satisfying $parent[j] = \min\{i > j : L_{ij} \neq 0\}$ where $G^+(A)$ is the graph of $L + L^T$. The filled graph or $G^+(A)$ results at the end of the elimination process and includes all edges of the original matrix A as well as the fill-in edges. Detailed discussions of the theory behind the elimination tree, the elimination process, and the filled graph can be found in [14, 53].

Figure 4 shows the pseudo-code of the left-looking sparse Cholesky, which is performed in two phases of *update* (lines 3–6) and *column factorization* (lines 7–10). The update phase gathers the contributions from the already factorized columns on the left. The column factorization phase calculates the square root of the diagonal element and applies it to the off-diagonal elements.

To find the prune-set that enables the VI-Prune transformation, the row sparsity pattern of L has to be computed. Figure 4 shows how this information is used to prune the iteration space of the update phase in the Cholesky algorithm. Since L is stored in column compressed format, the etree and the sparsity pattern of A are used to determine the L row sparsity pattern. A non-optimal method for finding the row sparsity pattern of row i in L is that for each nonzero A_{ij} the etree of A is traversed upwards from node j until node i is reached or a marked node is found. The row-count of i is the visited nodes in this subtree. Sympiler uses a similar but more optimized approach from [14] to find row sparsity patterns.

Supernodes used in VS-Block for Cholesky are found with the L sparsity pattern and the etree. The sparsity pattern of L is different from A because of fill-ins created during factorization. However, the elimination tree T along with the sparsity pattern of A are used to find the sparsity pattern of L prior to factorization. As a result, memory for L is allocated ahead of time to eliminate the need for dynamic memory allocation. To create the supernodes, the fill-in pattern should be first determined. Equation (1) is based on a theorem from [29] and computes the sparsity pattern of column j in L , L_j , where $T(s)$ is the parent of node s in T and “\” means exclusion. The theorem states that the nonzero pattern of L_j is the union of the nonzero patterns of the children of j in the etree and the nonzero pattern of column j in A .

$$L_j = A_j \cup \{j\} \cup \left(\bigcup_{s=T(s)} L_s \setminus \{s\} \right) \quad (1)$$

When the sparsity pattern of L is obtained, the following rule is used to merge columns to create basic supernodes: when the number

of nonzeros in two adjacent columns j and $j - 1$, regardless of the diagonal entry in $j - 1$ are equal and $j - 1$ is the only child of j in T , the two columns are merged.

Inspector-guided transformations: The VI-Prune transformation is applied to the update phase of Cholesky. With the row sparsity pattern information when factorizing column i , Sympiler only iterates over dependent columns instead of all columns smaller than i . The VS-Block transformation can be applied to both the update and the column factorization phases. Therefore, the outer loop in the Cholesky algorithm in Figure 4 is converted to a new loop that iterates over the block-set. All references to the column j in the inner loops will be changed to the $blockSet[j]$. For the diagonal part of the column factorization, a dense Cholesky needs to be computed instead of the square root in the non-supernodal version. The resulting factor from the diagonal elements applies to the off-diagonal rows through a sequence of dense triangular solves. VS-Block also converts the update phase from vector operations to matrix operations.

Symbolic Inspection: The computational complexity for building the etree in sympiler is nearly $O(|A|)$. The run-time complexity for finding the sparsity pattern of row i is proportional to the number of nonzeros in row i of A . The method is executed for all columns which results in a run-time of nearly $O(|A|)$. The inspection overhead for finding the block-set for VS-Block includes the sparsity detection which is done in nearly $O(|A| + 2n)$ and the supernode detection which has a run-time complexity of $O(n)$ [14].

3.3 Other Matrix Methods

The inspection graphs and inspection strategies supported in the current version of Sympiler are used in a large class of commonly-used sparse matrix computations. The applications of the elimination tree go beyond the Cholesky factorization method and extend to some of the most commonly used sparse matrix routines in scientific applications such as LU, QR, orthogonal factorization methods [46], and incomplete and factorized sparse approximate inverse preconditioner computations [38]. Inspection of the dependency graph and proposed inspection strategies that extract reach-sets and supernodes from the dependency graph are the fundamental symbolic analyses required to optimize algorithms such as rank update/downdate methods [18], incomplete LU(0) [49], incomplete Cholesky preconditioners, and up-looking implementations of factorization algorithms. Thus, Sympiler with the current set of symbolic inspectors can be made to support many of these

Table 2: Matrix set: The matrices are sorted based on the number of nonzeros in the original matrix; *nnz* refers to number of nonzeros, *n* is the rank of the matrix.

Problem ID	Name	n (10^3)	nnz (A) (10^6)
1	cbuckle	13.7	0.677
2	Pres_Poisson	14.8	0.716
3	gyro	17.4	1.02
4	gyro_k	17.4	1.02
5	Dubcova2	65.0	1.03
6	msc23052	23.1	1.14
7	thermomech_dM	204	1.42
8	Dubcova3	147	3.64
9	parabolic_fem	526	3.67
10	ecology2	1000	5.00
11	tmt_sym	727	5.08

matrix methods. We plan to extend to an even larger class of matrix methods and to support more optimization techniques.

4 EXPERIMENTAL RESULTS

We evaluate Sympiler by comparing the performance to two state-of-the-art libraries, namely Eigen [33] and CHOLMOD [11], for the Cholesky factorization method and the sparse triangular solve algorithm. Section 4.1 discusses the experimental setup and experimental methodology. In Section 4.2 we demonstrate that the transformations enabled by Sympiler generate highly-optimized codes for sparse matrix algorithms compared to state-of-the-art libraries. Although symbolic analysis is performed only once at compile time for a fixed sparsity pattern in Sympiler, we analyze the cost of the symbolic inspector in Section 4.3 and compare it with symbolic costs in Eigen and CHOLMOD.

4.1 Methodology

We selected a set of symmetric positive definite matrices from [19], which are listed in Table 2. The matrices originate from different domains and vary in size. All matrices have real numbers and are in double precision. The testbed architecture is a 3.30GHz Intel®Core™i7-5820K processor with L1, L2, and L3 cache sizes of 32KB, 256KB, and 15MB respectively with turbo-boost disabled. OpenBLAS.0.2.19 [73] is used for dense BLAS (Basic Linear Algebra Subprogram) routines when needed. All Sympiler-generated codes are compiled with GCC v.5.4.0 using the -O3 option. Each experiment is executed 5 times and the median is reported.

We compare the performance of the Sympiler-generated code with CHOLMOD [11] as a specialized library for Cholesky factorization and with Eigen [33] as a general numerical library. CHOLMOD provides one of the fastest implementations of Cholesky factorization on single-core architectures [32]. Eigen supports a wide range of sparse and dense operations including sparse triangular solve and Cholesky. Thus, Cholesky factorization results are compared with both Eigen and CHOLMOD while results for triangular solve are compared to Eigen. Both libraries are installed and executed using the recommended default configuration. For the Cholesky factorization both libraries support the more commonly used left-looking

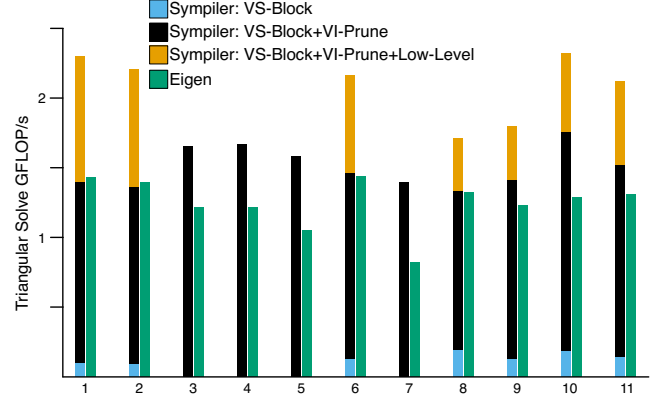


Figure 6: Sympiler’s performance compared to Eigen for triangular solve. The stacked-bars show the performance of the Sympiler (numeric) code with VS-Block and VI-Prune. The effects of VS-Block, VI-Prune, and low-level transformations on Sympiler’s performance are shown separately.

(supernodal) algorithm which is also the algorithm used by Sympiler. Sympiler applies one or both of the inspector-guided transformations as well as some of the enabled low-level transformations. For low-level transformations, Sympiler currently supports unrolling, scalar replacement, and loop distribution. For direct comparison of different implementations, a constant floating point operation (FLOP) count is used across all implementations.

4.2 Performance of Generated Code

This section shows how the combination of the introduced transformations and the decoupling strategy enable Sympiler to outperform two state-of-the-art libraries for sparse Cholesky and sparse triangular solve.

Triangular solve: Figure 6 shows the performance of Sympiler-generated code compared to the Eigen library for a sparse triangular solve with a sparse RHS. The nonzero fill-in of the RHS in our experiments is selected to be less than 5%. The sparse triangular system solver is often used as a sub-kernel in algorithms such as left-looking LU [14] and Cholesky rank update methods [18] or as a solver after matrix factorizations. Thus, typically the sparsity of the RHS in sparse triangular systems is close to the sparsity of the columns of a sparse matrix. For the tested problems, the number of nonzeros for all columns of L is less than 5%.

The average improvement of Sympiler-generated code, which we refer to as Sympiler (numeric), over the Eigen library is 1.49×. Eigen implements the approach demonstrated in Figure 1c, where symbolic analysis is not decoupled from the numerical code. However, the Sympiler-generated code only manipulates numerical values which leads to higher performance. Figure 6 also shows the effect of each transformation on the overall performance of the Sympiler-generated code. In the current version of Sympiler the symbolic inspector is designed to generate sets so that VS-Block can be applied before VI-Prune. Our experiments show that this ordering often leads to better performance mainly because Sympiler supports

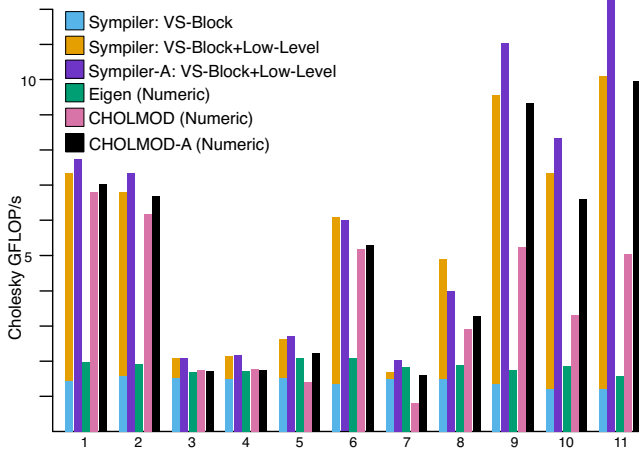


Figure 7: The performance of Sympiler (numeric) for Cholesky compared to CHOLMOD (numeric) and Eigen (numeric). The stacked-bar shows the performance of the Sympiler-generated code. The effect of VS-Block and low-level transformations are shown separately. The VI-Prune transformation is already applied to the baseline code so it is not shown here. Sympiler-A and CHOLMOD-A refer to versions with node amalgamation.

supernodes with a full diagonal block. As support for more transformations are added to Sympiler, we will enable it to automatically decide the best transformation ordering. Whenever applicable, vectorization and peeling transformations are applied after VS-Block and VI-Prune. Peeling leads to higher performance if applied after VS-Block where iterations related to single-column supernodes are peeled. Vectorization is always applied after VS-Block and does not improve performance if only VI-Prune is applied.

Matrices 3, 4, 5, and 7 do not benefit from the VS-Block transformation so their Sympiler run-times in Figure 6 are only for VI-Prune. Since small supernodes often do not lead to better performance, Sympiler does not apply the VS-Block transformation if the average size of the participating supernodes is smaller than a threshold. This parameter is currently hand-tuned and is set to 160. VS-Block is not applied to matrices 3, 4, 5, and 7 since the average supernode size is too small and thus does not improve performance. Also, since these matrices have a small column count vectorization does not payoff.

Cholesky: We compare the numerical manipulation code of Eigen and CHOLMOD for Cholesky factorization with the Sympiler-generated code. The results for CHOLMOD and Eigen in Figure 7 refer to the numerical code performance in floating point operations per second (FLOP/s). Eigen and CHOLMOD both execute parts of the symbolic analysis only once if the user explicitly indicates that the same sparse matrix is used for subsequent executions. However, even with such an input from the user, none of the libraries fully decouple the symbolic information from the numerical code. This is because they can not afford to have a separate implementation for each sparsity pattern and also do not implement sparsity-specific optimizations. For fairness, when using Eigen and CHOLMOD we explicitly tell the library that the sparsity is fixed and thus report

only the time related to the library’s numerical code (which still contains some symbolic analysis).

As shown in Figure 7, for Cholesky factorization Sympiler performs up to 1.3 \times , 2.3 \times , and 6.3 \times better than CHOLMOD with node amalgamation [26], CHOLMOD without node amalgamation, and Eigen respectively. Eigen uses the left-looking non-supernodal algorithm and thus its performance does not scale well with large matrices. CHOLMOD benefits from supernodes and performs well for large matrices with large supernodes. However, CHOLMOD does not perform well for some small matrices and large matrices with small supernodes. Node amalgamation merges small supernodes to increase their size for better performance and is implemented in both CHOLMOD and Sympiler. Sympiler provides the highest performance for almost all tested matrix types which demonstrates the effectiveness of sparsity-specific code generation.

The application of kernel-specific and aggressive optimizations when generating code for dense sub-kernels enables Sympiler to generate fast code for any sparsity pattern. Since BLAS routines are not well-optimized for small dense kernels they often do not perform well for the small blocks produced by applying VS-Block to sparse codes [63]. Therefore, libraries such as CHOLMOD do not perform well for matrices with small supernodes. Sympiler has the luxury to generate code for its dense sub-kernels; instead of being handicapped by the performance of BLAS routines, it generates specialized and highly-efficient codes for small dense sub-kernels. If the average column-count for a matrix is above a tuned threshold, Sympiler will call BLAS routines [73] instead. Since the column-count directly specifies the number of dense triangular solves, which is the most important dense sub-kernel in Cholesky, the average column-count is used to decide when to switch to BLAS routines [73]. For example, the average column-count of matrices 3, 4, 6, and 8 is less than the column-count threshold.

Decoupling the prune-set calculation from the numerical manipulation phase also improves the performance of the Sympiler-generated code. As discussed in subsection 3.2, the sparse Cholesky implementation needs the row sparsity pattern of L . The elimination tree of A and the upper triangular part of A are both used in CHOLMOD and Eigen to find the row sparsity pattern. Since A is symmetric with only the lower part stored, both libraries compute the transpose of A in the numerical code to access the upper triangular elements. Through fully decoupling symbolic analysis from the numerical code, Sympiler has the L row sparsity information in the prune-set ahead of time. Therefore, both the reach function and the matrix transpose operations are removed from the numeric code.

4.3 Symbolic Analysis Time

All symbolic analysis is performed at compile time in Sympiler and its generated code only manipulates numerical values. Since symbolic analysis is performed once for a specific sparsity pattern, its overheads amortize with repeat executions of the numerical code. However, as demonstrated in Figures 8 and 9 even if the numerical code is executed only once, which is not common in scientific applications, the accumulated symbolic+numeric time of Sympiler is close to Eigen for the triangular solve and faster than both Eigen and CHOLMOD for Cholesky.

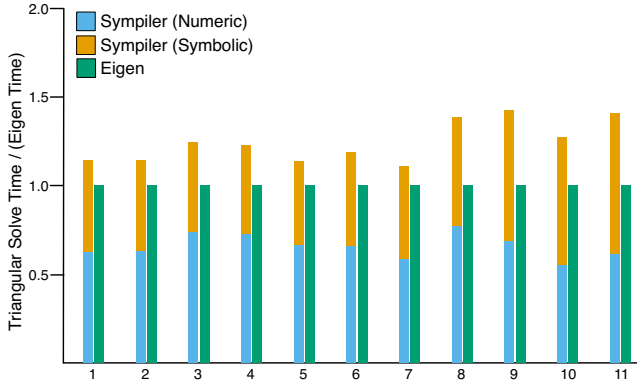


Figure 8: Sparse triangular solve symbolic+numeric time for Sympiler and Eigen’s normalized over the Eigen time.

Triangular solve: Figure 8 shows the time Sympiler spends to do symbolic analysis at compile time, Sympiler (symbolic), for sparse triangular solve, normalized over Eigen’s run-time. No symbolic time is available for Eigen since as discussed, Eigen uses the code in Figure 1c for its triangular solve implementation. Sympiler’s numeric plus symbolic time is on average $1.27\times$ slower than the Eigen code. In addition, depending on the matrix, code generation and compilation in Sympiler costs between $6\text{--}197\times$ more than the numeric solve. It is important to note that since the sparsity structure of the matrix in triangular solve does not change in many applications, the overhead of the symbolic inspector and compilation is only paid once. For example, in preconditioned iterative solvers a triangular system must be solved per iteration and often the iterative solver must execute thousands of iterations [8, 41, 50] until convergence since the systems in scientific applications are not necessarily well-conditioned.

Cholesky: Sparse libraries perform symbolic analysis ahead of time which can be re-used for same sparsity patterns and improves the performance of their numerical executions. We compare the analysis time of the libraries with Sympiler’s symbolic inspection time. Figure 9 provides the symbolic analysis and numeric manipulation times for both libraries normalized over Eigen time. The time spent by Sympiler to perform symbolic analysis is referred to as Sympiler symbolic. CHOLMOD (symbolic) and Eigen (symbolic) refer to the partially decoupled symbolic code that is only executed once if the user indicates that sparsity remains static. In nearly all cases Sympiler’s accumulated time is better than the other two libraries. Code generation and compilation, which are not shown in the chart, cost at most $0.3\times$ the cost of numeric factorization. Also, similar to the triangular solve example, a matrix with a fixed sparsity pattern must be factorized many times in scientific applications. For example, in Newton-Raphson (NR) solvers for nonlinear systems of equations, a Jacobian matrix is factorized in each iteration and the NR solvers require tens or hundreds of iterations to converge [21, 51].

5 RELATED WORK

Compilers for general languages are hampered by optimization methods that either cannot optimize sparse codes or only

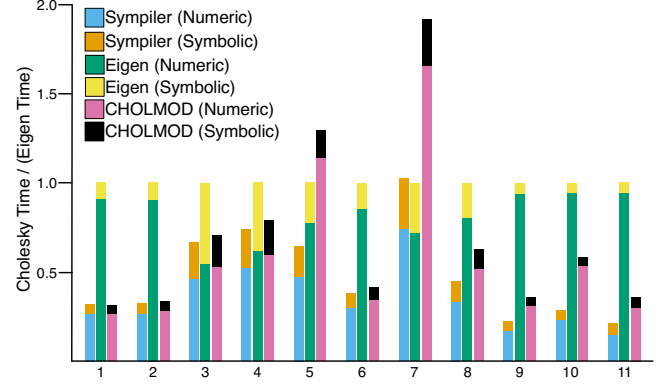


Figure 9: Symbolic+numeric time for Sympiler, CHOLMOD, and Eigen for the Cholesky algorithm. All times are normalized over the Eigen’s accumulated symbolic+numeric time.

apply conservative transformations that do not lead to high performance. This is due to the indirection required to index and loop over nonzero elements of sparse data structures. Polyhedral methods are limited when dealing with non-affine loop nests or subscripts [5, 10, 40, 55, 67, 69] common in sparse computations.

To make it possible for compilers to apply more aggressive loop and data transformations to sparse codes, recent work [65, 68, 70–72] has developed compile-time techniques for automatically creating *inspectors* and *executors* for use at run-time. These techniques use an inspector to analyze index arrays in sparse codes at run-time and an executor that uses this run-time information to execute code with specific optimizations. These inspector-executor techniques are limited in that they only apply to sparse codes with static index arrays; such codes require the matrix structure to not change during the computation. The aforementioned approach performs well for methods such as sparse incomplete LU (0) and Gauss-Seidel methods where additional nonzeros/fill-ins are not introduced during computation. However, in a large class of sparse matrix methods, such as direct solvers including Cholesky, LU, and QR decompositions, index arrays dynamically change during computation since the algorithm itself introduces fill-ins. In addition, the indirections and dependencies in sparse direct solvers are tightly coupled with the algorithm, making it difficult to apply inspector-executor techniques. Partial evaluation techniques [39] specialize code by using partial inputs and the program to generate code that works well for all the remaining inputs. These methods have not been successfully extended to support sparse matrix methods [36].

Domain-specific compilers integrate domain knowledge into the compilation process, improving the compiler’s ability to transform and optimize specific kinds of computations. Such an approach has been used successfully for stencil computations [37, 57, 66], signal processing [54], dense linear algebra [34, 64], matrix assembly and mesh analysis [1, 48], simulation [9, 42], and sparse operations [16, 58]. Though the simulations and sparse compilers use some knowledge of matrix structure to optimize operations, they do not build specialized matrix solvers.

Specialized Libraries are the typical approach for sparse direct solvers. These libraries differ in (1) which numerical methods are

implemented, (2) the implementation strategy or variant of the solver, (3) the type of the platform supported, and (4) whether the algorithm is specialized for specific applications.

Each numerical method is suitable for different classes of matrices; for example, Cholesky factorization requires the matrix be symmetric (or Hermitian) positive definite. Libraries such as SuperLU [22], KLU [20], UMFPACK [12], and Eigen [33] provide optimized implementations for LU decomposition methods. The Cholesky factorization is available through libraries such as Eigen [33], CSparse [14], CHOLMOD [11], MUMPS [2–4], and PARDISO [59, 60]. QR factorization is implemented in SPARSPAK [28, 30], SPL00ES [6], Eigen [33], and CSparse [14]. The optimizations and algorithm variants used to implement sparse matrix methods differ between libraries. For example, LU decomposition can be implemented using multifrontal methods [12, 15, 35], left-looking [20, 22, 27, 28], right-looking [25, 44, 61], and up-looking [13, 62] methods. Libraries are developed to support different platforms such as sequential implementations [11, 14, 20], shared memory [15, 23, 59], and distributed memory [3, 23]. Finally, some libraries are designed to perform well on matrices arising from a specific domain. For example, KLU [20] works best for circuit simulation problems. In contrast, SuperLU-MT applies optimizations with the assumption that the input matrix structure leads to large supernodes; such a strategy is a poor fit for circuit simulation problems.

6 FUTURE WORK AND CONCLUSION

In this paper we demonstrated how decoupling symbolic analysis from numerical manipulation can enable the generation of domain-specific highly-optimized sparse codes with static sparsity patterns. Sympiler, the proposed domain-specific code generator, takes the sparse matrix pattern and the sparse matrix algorithm as inputs to perform symbolic analysis at compile time. It then uses the information from symbolic analysis to apply a number of inspector-guided and low-level transformations to the sparse code. The Sympiler-generated code outperforms two state-of-the-art sparse libraries, Eigen and CHOLMOD, for the sparse Cholesky and the sparse triangular solve algorithms.

We believe Sympiler has the potential to support a large class of sparse matrix methods, accommodate adaptive changes to the input sparsity pattern, and generate optimized code for parallel and distributed hardware platforms. So far, our implementation has only scratched the surface of what kinds applications and optimizations are possible. In future work, we will explore new inspector-guided transformations to support code generation for parallel and heterogeneous platforms. Although static sparsity is common and important enough to warrant exploring code specialized for specific sparsity patterns, adaptive updates to the input sparsity can also be supported by Sympiler and will be explored in future work. While domain experts are often aware of changes in sparsity, Sympiler will be extended to detect incorrect input for code generated for a specific sparsity pattern. Finally, we plan to extend Sympiler to enable cross-kernel optimization and optimize whole applications at once rather than only the solver.

7 ACKNOWLEDGEMENTS

This work is supported by the U.S. National Science Foundation (NSF) Award Numbers CCF-1657175 and CCF-1564074. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by the National Science Foundation grant number ACI-1548562.

REFERENCES

- [1] Martin S Alnæs, Anders Logg, Kristian B Ølgaard, Marie E Rognes, and Garth N Wells. 2014. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software (TOMS)* 40, 2 (2014), 9.
- [2] Patrick R Amestoy, Iain S Duff, and J-Y L'Excellent. 2000. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer methods in applied mechanics and engineering* 184, 2 (2000), 501–520.
- [3] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.* 23, 1 (2001), 15–41.
- [4] Patrick R Amestoy, Abdou Guermouche, Jean-Yves L'Excellent, and Stéphane Pralet. 2006. Hybrid scheduling for the parallel solution of linear systems. *Parallel computing* 32, 2 (2006), 136–156.
- [5] Corinne Ancourt and François Irigoin. 1991. Scanning polyhedra with DO loops. In *ACM Sigplan Notices*, Vol. 26. ACM, 39–50.
- [6] Cleve Ashcraft and Roger G Grimes. 1999. SPOLES: An Object-Oriented Sparse Matrix Library. In *PPSC*.
- [7] Yunfei Bai, M. Danny Kaufman, C. Karen Liu, and Jovan Popović. 2016. Artist-dynamics for 2D animation. In *ACM Transactions on Graphics (SIGGRAPH 2016)*.
- [8] Michele Benzi, Jane K Cullum, and Miroslav Tuma. 2000. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing* 22, 4 (2000), 1318–1332.
- [9] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for Physical Simulation on CPUs and GPUs. *ACM Trans. Graph.* 35, 2, Article 21 (May 2016), 12 pages. DOI: <https://doi.org/10.1145/2892632>
- [10] Chun Chen. 2012. Polyhedra scanning revisited. *ACM SIGPLAN Notices* 47, 6 (2012), 499–508.
- [11] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)* 35, 3 (2008), 22.
- [12] Timothy A Davis. 2004. Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)* 30, 2 (2004), 196–199.
- [13] Timothy A Davis. 2005. Algorithm 849: A concise sparse Cholesky factorization package. *ACM Transactions on Mathematical Software (TOMS)* 31, 4 (2005), 587–591.
- [14] Timothy A Davis. 2006. *Direct methods for sparse linear systems*. Vol. 2. Siam.
- [15] Timothy A Davis. 2011. Algorithm 915, SuiteSparseQR: Multifrontal multi-threaded rank-revealing sparse QR factorization. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 8.
- [16] Timothy A Davis. 2013. Algorithm 930: FACTORIZE: An object-oriented linear system solver for MATLAB. *ACM Transactions on Mathematical Software (TOMS)* 39, 4 (2013), 28.
- [17] Timothy A Davis and William W Hager. 2005. Row modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.* 26, 3 (2005), 621–639.
- [18] Timothy A Davis and William W Hager. 2009. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Transactions on Mathematical Software (TOMS)* 35, 4 (2009), 27.
- [19] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.
- [20] Timothy A Davis and Ekanathan Palamadai Natarajan. 2010. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software (TOMS)* 37, 3 (2010), 36.
- [21] Ailson P de Moura and Adriano Aron F de Moura. 2013. Newton–Raphson power flow with constant matrices: a comparison with decoupled power flow methods. *International Journal of Electrical Power & Energy Systems* 46 (2013), 108–114.
- [22] James W Demmel, Stanley C Eisenstat, John R Gilbert, Xiaoye S Li, and Joseph WH Liu. 1999. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.* 20, 3 (1999), 720–755.
- [23] James W Demmel, John R Gilbert, and Xiaoye S Li. 1999. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Anal. Appl.* 20, 4 (1999), 915–952.
- [24] Richard C Dorf. 2006. *Electronics, power electronics, optoelectronics, microwaves, electromagnetics, and radar*. CRC press.

- [25] Iain S Duff, Nick IM Gould, John K Reid, Jennifer A Scott, and Kathryn Turner. 1991. The factorization of sparse symmetric indefinite matrices. *IMA J. Numer. Anal.* 11, 2 (1991), 181–204.
- [26] Iain S Duff and John K Reid. 1983. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software (TOMS)* 9, 3 (1983), 302–325.
- [27] Iain S Duff and John Ker Reid. 1996. The design of MA48: a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Transactions on Mathematical Software (TOMS)* 22, 2 (1996), 187–226.
- [28] Alan George and Joseph WH Liu. 1979. The design of a user interface for a sparse matrix package. *ACM Transactions on Mathematical Software (TOMS)* 5, 2 (1979), 139–162.
- [29] Alan George and Joseph W. Liu. 1981. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference.
- [30] Alan George and Joseph W Liu. 1981. Computer solution of large sparse positive definite. (1981).
- [31] John R Gilbert and Tim Peierls. 1988. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.* 9, 5 (1988), 862–874.
- [32] Nicholas IM Gould, Jennifer A Scott, and Yifan Hu. 2007. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software (TOMS)* 33, 2 (2007), 10.
- [33] Gaël Guennebaud and Benoit Jacob. 2010. Eigen. URL: <http://eigen.tuxfamily.org> (2010).
- [34] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. 2001. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)* 27, 4 (2001), 422–455.
- [35] Anshul Gupta, George Karypis, and Vipin Kumar. 1997. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems* 8, 5 (1997), 502–520.
- [36] Fred G Gustavson, Werner Liniger, and R Willoughby. 1970. Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations. *Journal of the ACM (JACM)* 17, 1 (1970), 87–109.
- [37] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 311–320. DOI: <https://doi.org/10.1145/2304576.2304619>
- [38] Carlo Janna, Massimiliano Ferronato, and Giuseppe Gambolati. 2015. The use of supernodes in factored sparse approximate inverse preconditioning. *SIAM Journal on Scientific Computing* 37, 1 (2015), C72–C94.
- [39] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Peter Sestoft.
- [40] Wayne Kelly. 1998. Optimization within a unified transformation framework. (1998).
- [41] David S Kershaw. 1978. The incomplete Cholesky conjugate gradient method for the iterative solution of systems of linear equations. *J. Comput. Phys.* 26, 1 (1978), 43–65.
- [42] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 20.
- [43] Xiaoye S Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)* 31, 3 (2005), 302–325.
- [44] Xiaoye S Li and James W Demmel. 2003. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software (TOMS)* 29, 2 (2003), 110–140.
- [45] Douglas K Lilly. 1965. On the computational stability of numerical solutions of time-dependent non-linear geophysical fluid dynamics problems. *Mon. Wea. Rev.* 93, 1 (1965), 11–26.
- [46] Joseph W. H. Liu. 1990. The Role of Elimination Trees in Sparse Factorization. *SIAM J. Matrix Anal. Appl.* 11, 1 (Jan. 1990), 134–172. DOI: <https://doi.org/10.1137/0611010>
- [47] José A Zevallos Luna, Alexandre Siligaris, Cédric Pujol, and Laurent Dussopt. 2013. A packaged 60 GHz low-power transceiver with integrated antennas for short-range communications.. In *Radio and Wireless Symposium*. 355–357.
- [48] Fabio Luporini, David A Ham, and Paul HJ Kelly. 2016. An algorithm for the optimization of finite element integration loops. *arXiv preprint arXiv:1604.05872* (2016).
- [49] Maxim Naumov. 2012. Parallel incomplete-LU and Cholesky factorization in the preconditioned iterative methods on the GPU. *NVIDIA Technical Report NVR-2012-003* (2012).
- [50] M Papadrakakis and N Bitoulas. 1993. Accuracy and effectiveness of preconditioned conjugate gradient algorithms for large and ill-conditioned problems. *Computer methods in applied mechanics and engineering* 109, 3-4 (1993), 219–232.
- [51] Roger P Pawlowski, John N Shadid, Joseph P Simonis, and Homer F Walker. 2006. Globalization techniques for Newton–Krylov methods and applications to the fully coupled solution of the Navier–Stokes equations. *SIAM review* 48, 4 (2006), 700–721.
- [52] Lukas Polok, Viorela Ila, Marek Solony, Pavel Smrz, and Pavel Zemcik. 2013. Incremental Block Cholesky Factorization for Nonlinear Least Squares in Robotics.. In *Robotics: Science and Systems*.
- [53] Alex Pothen and Sivan Toledo. 2004. Elimination Structures in Scientific Computing. (2004).
- [54] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93, 2 (2005), 232–275.
- [55] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. 2000. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming* 28, 5 (2000), 469–498.
- [56] Michael Rabinovich, Roi Poranne, Daniele Panozzo, and Olga Sorkine-Hornung. 2017. Scalable locally injective mappings. *ACM Transactions on Graphics (TOG)* 36, 2 (2017), 16.
- [57] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.
- [58] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A Anderson, and Mikhail Smelyanskiy. 2016. Sparso: Context-driven optimizations of sparse linear algebra. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 247–259.
- [59] Olaf Schenk and Klaus Gärtner. 2004. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems* 20, 3 (2004), 475–487.
- [60] Olaf Schenk, Klaus Gärtner, and Wolfgang Fichtner. 2000. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT Numerical Mathematics* 40, 1 (2000), 158–176.
- [61] Kai Shen, Tao Yang, and Xiangmin Jiao. 2000. S+: Efficient 2D sparse LU factorization on parallel machines. *SIAM J. Matrix Anal. Appl.* 22, 1 (2000), 282–305.
- [62] Andrew H Sherman. 1978. Algorithms for sparse Gaussian elimination with partial pivoting. *ACM Transactions on Mathematical Software (TOMS)* 4, 4 (1978), 330–338.
- [63] Jaewook Shin, Mary W Hall, Jacqueline Chame, Chun Chen, Paul F Fischer, and Paul D Hovland. 2010. Speeding up Nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 253–262.
- [64] Daniele G Spampinato and Markus Püschel. 2014. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 23.
- [65] Michelle Mills Strout, Alan LaMille, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An approach for code generation in the sparse polyhedral framework. *Parallel Comput.* 53 (2016), 32–57.
- [66] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. 2011. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 117–128.
- [67] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. 2009. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 1–12.
- [68] Harmen LA Van Der Spek and Harry AG Wijshoff. 2010. Sublimation: expanding data structures to enable data instance specific optimizations. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 106–120.
- [69] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. 2006. Polyhedral code generation in the real world. In *International Conference on Compiler Construction*. Springer, 185–201.
- [70] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 521–532.
- [71] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 41.
- [72] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 185.
- [73] Z Xianyi. 2016. OpenBLAS: an optimized BLAS library. (2016).

A ARTIFACT DESCRIPTION: SYMPILER: TRANSFORMING SPARSE MATRIX CODES BY DECOUPLING SYMBOLIC ANALYSIS

A.1 Abstract

This artifact provides information to reproduce results shown in the paper: “Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis”. We explain how to compile and run Sympiler and also provide instructions to build the CHOLMOD and Eigen libraries used for comparison. For convenience, a script for evaluating each algorithm in the paper is provided.

A.2 Description

A.2.1 Check-list (artifact meta-information).

- **Algorithm:** The sparse Cholesky factorization and the triangular solver with a sparse right-hand-side
- **Program:** C++ code
- **Compilation:** GCC v.5.4.0 with -O3 flag.
- **Binary:** OpenBLAS v.0.2.19
- **Data set:** Publicly available matrix market files
- **Run-time environment:** Ubuntu 16.04
- **Hardware:** Intel®Core™i7-5820K
- **Output:** The L-factor for the Cholesky algorithm and the solution vector x for the triangular solver.
- **Experiment workflow:** Install OpenBlas, CHOLMOD, and Eigen. Compile Sympiler and execute the test scripts.
- **Experiment customization:** Some parameters need to be changed in CHOLMOD.
- **Publicly available:** yes.

A.2.2 *How software can be obtained (if available).* Sympiler’s code will be available in Github. We will provide a link when accepted. To maintain anonymity is not ready for public release.

A.2.3 *Hardware dependencies.* There is not hardware dependency for Sympiler. It compiles and runs on any x86 processor. However, the reported results are obtained on an Intel®Core™i7-5820K.

A.2.4 *Software dependencies.* The current version of Sympiler relies on BLAS for evaluating dense sub-blocks in Cholesky.

A.2.5 *Datasets.* The current version of Sympiler only supports input matrices in the *matrix market* format. All matrices selected for evaluation are publicly available from the Florida Sparse Matrix Repository. The matrices can be downloaded from the UF website: www.cise.ufl.edu/research/sparse/matrices/

For the sparse triangular system examples, a matrix from the UF repository is first factorized and the resulting factor L is stored in a local repository to be used in the triangular solve experiments.

A.3 Installation

The steps to installing Sympiler and then the setup instructions for Eigen and CHOLMOD are presented here.

Sympiler. Since the current version of Sympiler calls some BLAS routines, the user needs to modify the `BLAS_LIB` variable in `CMakeLists.txt` and set it to the installed OpenBLAS library path. The following shows the remaining steps for installing Sympiler.

```
$ git clone <https://github.com/sympiler/sympiler.git> Sympiler
$ cd Sympiler;
```

```
$ mkdir build; cd build;
$ cmake ..
$ make
```

Libraries. We have prepared a simple project for evaluating Cholesky and the triangular solver using Eigen and CHOLMOD libraries. After installing CHOLMOD and Eigen, the paths have to be set via `CHOL_LIB` and `Eigen_LIB` in `libTest/CMakeLists.txt`. The following steps will prepare the test environment for evaluation.

```
$ cd libTest;
$ mkdir build; cd build;
$ cmake ..
$ make
```

Similar to the paper, the results for running CHOLMOD with and without node amalgamation are generated.

A.4 Experiment workflow

Sympiler. We have prepared a script which generates the performance results for each algorithm evaluated in the paper. For the triangular solver, the user can run:

```
$ cd /path/to/Sympiler/build/
$ ./triangular.sh /path/to/triangular/input/matrix.mtx
```

The *triangular* script first generates a code for the given matrix and stores it in the *symGen* folder. Then it compiles and runs the generated code for the input matrix. For the Cholesky factorization, the user can run a separate script as follows:

```
$ ./cholesky.sh /path/to/input/matrix.mtx
```

Libraries. To compare Sympiler’s performance with the two libraries, two binary files one for each algorithm in *libTest* are provided. These files are built from the source code available in the repository. For triangular solve evaluation, the following sequence of commands will show the execution time of Eigen and Sympiler. Functions are provided within the code that verify the correctness of the resulting vector x .

```
$ cd /path/to/Sympiler/libTest/build/
$ ./libTriangular /path/to/triangular/input/matrix.mtx
```

For the Cholesky evaluations, the following commands show the performance of both Eigen and CholMod libraries for a given input matrix. A built-in function compares the L factor to check correctness.

```
$ cd /path/to/Sympiler/libTest/build/
$ ./libCholesky /path/to/input/matrix.mtx
```

A.5 Evaluation and expected results

The expected results include the factor L for Cholesky and the solution vector x for the triangular solver. The performance results include the numerical manipulation and the symbolic analysis time which are also presented in FLOPS.

A.6 Experiment customization

The experiments can be easily ran for other *matrix market* matrices.

A.7 Notes

An updated installation guide will be provided in the Sympiler repository.