CGO: G: Decoupling Symbolic from Numeric in Sparse Matrix Computations

Kazem Cheshmi
PhD Student, Rutgers University
kazem.ch@rutgers.edu

Advisor: Maryam Mehri Dehnavi

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 CHOLMOD and Eigen of 3.8× and 1.5× respectively.

1 PROBLEM AND MOTIVATIONS

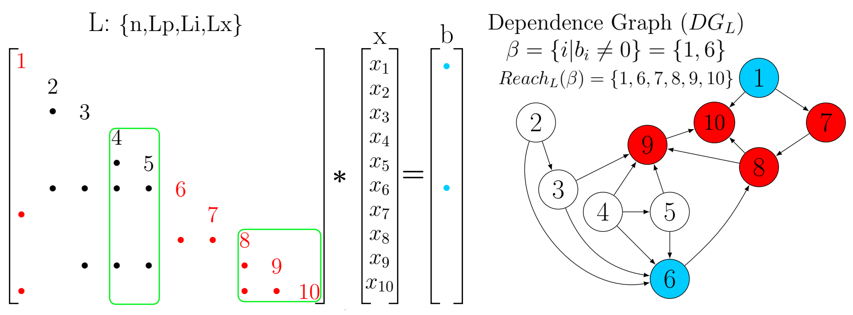
Sparse matrix computations are at the heart of many scientific applications and data analytics codes. The performance and efficient memory usage of these codes depends heavily on their use of specialized sparse matrix data structures that only store the nonzero entries. However, such compaction is done 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. Sympiler addresses these limitations by performing *symbolic analysis* at compile-time to compute fill-in structure and to remove dynamic index arrays from sparse matrix computations. Symbolic analysis is a term from the numerical computing community. It refers to phases that determine the computational patterns that only depend on the nonzero pattern and not on numerical values. Information from symbolic analysis can be used to make subsequent numeric manipulation faster, and the information 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, and quotient graph) and applying a method-dependent graph algorithm yields information about dependences that can then be used to more efficiently compute the numerical method [2]. 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, which generates high-performance sparse matrix code by fully decoupling the symbolic analysis from numeric computation and transforming code to utilize the symbolic information. After obtaining symbolic information by running a symbolic inspector, Sympiler applies inspector-guided transformations, such as variable-sized blocking, 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 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.

Motivating Scenario: 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 , direct system solvers [2, 8], and rank update methods [3], where the RHS vector is often sparse. A naïve implementation visits every column of matrix L to propagate the contributions of its corresponding x value to the rest of x (see Figure 1b). However, with a sparse b, the solution vector is also sparse, reducing the required iteration space of sparse triangular solve to be proportional to the number of nonzero values in x. Taking advantage of this property requires first determining the nonzero pattern of x. Based on a theorem from Gilbert and Peierls [5], the *dependence graph* $DG_L = (V, E)$ for matrix L with nodes $V = \{1, ..., n\}$ 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 can be computed with a depth-first search of the directed graph DG_L starting with β. An example dependence graph is illustrated in Figure 1a. The blue colored nodes correspond to set β and the final *reach-set* $Reach_L(\beta)$ contains all the colored nodes in the graph.

The implementation in Figure 1d shows a decoupled code that uses the symbolic information provided by the pre-computed reach-set. The library implementation where both symbolic analysis and numerical manipulation are mixed is shown in Figure 1c. 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 leveraging it to generate code specialized for the specific matrix structure and 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 case of the figure, this threshold is 2). These peeled loops can be further


L: {n,Lp,Li,Lx}
x
b
Dependence Graph (DG_L)
$\beta = \{i | b_i \neq 0\} = \{1, 6\}$
$Reach_L(\beta) = \{1, 6, 7, 8, 9, 10\}$
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}
1 2 3 4 5 6 7 8 9 10

(a)

```
x=b; // copy RHS to x
for(j=0;j<n;j++){
  x[j]/=Lx[Lp[j]];
  for(p=Lp[j]+1;p<Lp[j+1];p++){
    x[Li[p]]-=Lx[p]*x[j];}}
```

(b) Forward substitution

```
x=b;
for(j=0;j<n;j++){
  if(x[j] != 0){
    x[j]/=Lx[Lp[j]];
    for(p=Lp[j]+1;p<Lp[j+1];p++)
      x[Li[p]]-=Lx[p]*x[j];}}
```

(c) Library implementation

```
x=b;
for(px=0;px<reachSetSize;px++){
  j=reachSet[px];
  x[j]/=Lx[Lp[j]];
  for(p=Lp[j]+1;p<Lp[j+1];p++){
      x[Li[p]]-=Lx[p]*x[j];}}
```

(d) Decoupled code

```
x=b;
x[0] /= Lx[0]; // Peel col 0
for(p = 1; p < 3; p++)
    x[Li[p]] -= Lx[p] * x[0];
for(px=1;px<3;px++){
  j=reachSet[px];x[j]/=Lx[Lp[j]]
  for(p=Lp[j]+1;p<Lp[j+1];p++)
      x[Li[p]]-=Lx[p]*x[j];}
x[7] /= Lx[20]; // Peel col 7
for(p = 21; p < 23; p++)
    x[Li[p]] -= Lx[p] * x[7];
for(px=4;px<reachSetSize;px++){
  j=reachSet[px];x[j]/=Lx[Lp[j]]
  for(p=Lp[j]+1;p<Lp[j+1];p++)
      x[Li[p]]-=Lx[p]*x[j];}
```

(e) Sympiler-generated

Figure 1: Four different codes for solving the linear system in (a). In all four code variants, the matrix L is stored in compressed sparse column (CSC) format, with {`n,Lp,Li,Lx`} representing {matrix order, column pointer, row index, nonzeros} respectively. (b), (c), (d), and (e) show four different implementations for solving the linear system in (a).

transformed with vectorization to speed up execution. On matrices from the SuiteSparse Matrix Collection, 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).

Static Sparsity Patterns: A fundamental concept that Sympiler is built on is that the structure of sparse matrices in scientific codes is dictated by the physical domain and as such does not change in many applications. For example, in power system modeling and circuit simulation problems the sparse matrix used in the matrix computations is often a Jacobian matrix, where the structure is derived from interconnections among the power system and circuit components such as generation, transmission, and distribution resources. While the numerical values in the sparse input matrix change often, a change in the sparsity structure occurs on rare occasions with a change in circuit breakers, transmission lines, or one of the physical components. Sparse matrices in many other physical domains exhibit the same behavior and benefit from Sympiler.

2 RELATED WORK

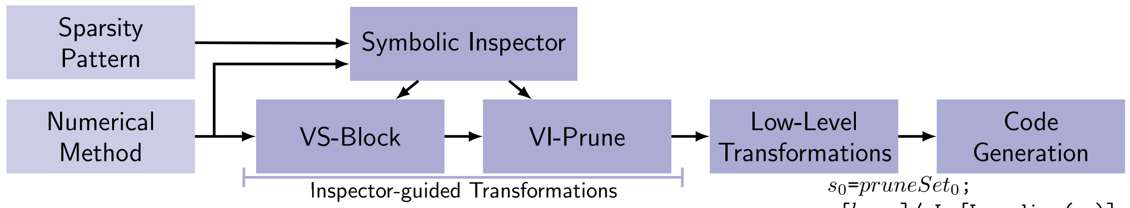
The most common approach to accelerating sparse matrix computations is to identify a specialized library that provides a manually-tuned implementation of the specific sparse matrix routine. A large number of sparse libraries are available (e.g., SuperLU [8], CHOLMOD [1], and Eigen [7]) for different 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.

Compilers can be used to optimize code while providing architecture portability. However, indirect accesses and the resulting complex dependence structure run into compile-time loop transformation framework limitations. 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 [11]. 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 by building run-time *inspectors* that examine the nonzero structure and *executors* that use this knowledge to transform code execution [10, 12]. 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 compute fill-in structure and to remove dynamic index arrays from sparse matrix computations.

3 SYMPILER: A SYMBOLIC-ENABLED CODE GENERATOR

Sympiler generates efficient sparse kernels by tailoring sparse code to specific matrix sparsity structures. In this section, we describe the overall structure of the Sympiler code generator.

Sympiler Overview: Sparse triangular solve and Cholesky factorization are currently implemented in Sympiler. 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. Code implementing the numerical solver is represented in a domain-specific abstract syntax tree (AST). The initial AST for triangular solve is shown in Figure 2a prior to any transformations. Sympiler produces the final C code by applying two series of transformations i.e. inspector-guided and low-level transformations, to this AST. An overview of the process is shown in Figure 2.



```
VI-Prune
for sol.j0 in 0..Lsp.n
VS-Block
  x[bsp_j0]/=Lx[Lsp.diag(j0)];
  VS-Block
  for sol.j1 in Lsp.col_j0..Lsp.col_j0+1
    x[Lsp.row_j1]-=Lx[j1]*x[bsp_j0];
```

(a) Initial AST

```
peel(0,3)
for sol.p0 in 0..pruneSetSize
  j0=pruneSet_p0;
  x[bsp_j0]/=Lx[Lsp.diag(j0)];
  vec(0)
  for sol.j1 in Lsp.col_j0..Lsp.col_j0+1
    x[Lsp.row_j1]-=Lx[j1]*x[bsp_j0];
```

(b) After VI-Prune

```
s0=pruneSet0;
x[bsp_s0]/=Lx[Lsp.diag(s0)];
for sol.j1 in Lsp.col_s0..Lsp.col_s0+1
  x[Lsp.row_j1]-=Lx[j1]*x[bsp_s0];
for sol.p0 in 0..pruneSetSize
  j0=pruneSet_p0;
  x[bsp_j0]/=Lx[Lsp.diag(j0)];
  for sol.j1 in Lsp.col_j0..Lsp.col_j0+1
    x[Lsp.row_j1]-=Lx[j1]*x[bsp_j0];
```

(c) After Low-Level Transformations

Figure 2: Sympiler lowers a functional representation of a sparse kernel to imperative code using the inspection sets from the symbolic inspector and through the inspector-guided and low-level transformations. For instance, the transformation steps for the code in Figure 1 are illustratedin steps a, b, and c.

Symbolic Inspector: Different numerical algorithms can make use of symbolic information in different ways, and prior work has described run-time graph traversal strategies for various numerical methods [2, 8, 9]. 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 obtain information about the sparsity structure of the input matrix and stores it in an algorithm-specific way for use during later transformation stages.

We classify the used 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*. The result of the inspection is the *inspection set*, which contains the result of running the inspector on the inspection graph. Inspection sets are used to guide the transformations in Sympiler. For our motivating example, triangular solve, the *reach-set* can be used to prune loop iterations that perform work that is unnecessary due to the sparseness of matrix or RHS. 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.

Inspector-guided Transformations: The initial lowered code along with the inspection sets obtained by the symbolic inspector are passed to a series of passes that further transform the code. Sympiler currently supports two transformations guided by the inspection sets: *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 2a, the code is 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.

Variable Iteration Space Pruning (VI-Prune) prunes the iteration space of a loop using information about the sparse computation. The inspection stage of Sympiler generates an inspection set that enables transforming the unoptimized sparse code to a code with a reduced iteration space. An example of VI-Prune is shown in Figure 1. 2D Variable-Sized Blocking (VS-Block) converts a sparse code to a set of non-uniform dense sub-kernels. The symbolic inspector identifies sub-kernels with similar structure in the sparse matrix methods and the sparse inputs to provide the VS-Block stage with "blockable" sets that are not necessarily of the same size or consecutively located. These blocks are similar to the concept of *supernodes* [8] in sparse libraries. The boxes around columns in Figure 1a show two supernodes of different sizes.

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 right-hand side vectors, the code is annotated with some transformation directives. An example of these annotations are shown in Figure 2b 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. An example is shown in Figure 1e where some of the iterations in the triangular solve code after VI-Prune can be peeled based on the number of nonzeros in their columns.

4 CASE STUDIES

This section discusses some of the graph theory and algorithms used in Sympiler's symbolic inspector to extract inspections sets for triangular solve and Cholesky factorization algorithms. Table 1 shows a classification of the inspection graphs, strategies, inspection sets, and enabled transformations for the two studied numerical algorithms in Sympiler. We also discuss extending Sympiler to other matrix methods.

Sparse Triangular Solve: The graph DG_L can also be 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, are identified by inspecting

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			
	Inspection Graph	Inspection Strategy	Inspection Set	Inspection Graph	Inspection Strategy	Inspection Set	Enabled Low-level
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(A)	Up-traversal	Block-set (supernodes)	tile, unroll, peel, vectorization

DG_L using a node equivalence method. The VS-Block transformation changes the loops shown in Figure 2a to apply VS-Blocking. The diagonal block of each column-block, which is a small triangular solve, is solved first. The solution of the diagonal components is then substituted in the off-diagonal segment of the matrix.

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. The elimination tree (etree) [2] is one of the most important graph structures used in the symbolic analysis of sparse factorization algorithms [9]. 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 fill-in edges.

To find the prune-set that enables the VI-Prune transformation, the row sparsity pattern of L has to be computed; 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 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 sparsity pattern of i is the set of the visited nodes in this subtree. Supernodes used in VS-Block for Cholesky are found with the L sparsity pattern and the etree. Equation (1) is used for computing the sparsity pattern of column j in L, L_j where $T(s)$ is the parent of node s in T and "\" means exclusion. 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 $j-1$, is equal, and $j-1$ is the only child of j in T, the two columns can be merged.

$$L_j = A_j \bigcup \{j\} \bigcup \left(\bigcup_{j=T(s)} L_s \backslash \{s\} \right) \tag{1}$$

Other Matrix Methods: The inspection graphs and inspection methods supported in the current version of Sympiler can support 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, and orthogonal factorization methods [2]. Inspection strategies of the dependency graph such as reach-set and supernode detection are the fundamental symbolic analysis required to optimize algorithms such as rank update methods[3], incomplete LU(0) and Cholesky preconditioners.

Table 2: Matrix set: The matrices are sorted based on the number of nonzeros in the original matrix.

ID	Name	nnz (A) (10^6)	ID	Name	nnz (A) (10^6)
1	cbuckle	0.677	7	thermomech_dM	1.42
2	Pres_Poisson	0.716	8	Dubcova3	3.64
3	gyro	1.02	9	parabolic_fem	3.67
4	gyro_k	1.02	10	ecology2	5.00
5	Dubcova2	1.03	11	tmt_sym	5.08
6	msc23052	1.14			

5 EXPERIMENTAL RESULTS

Methodology. We selected a set of symmetric positive definite matrices from [4] shown in Table 2. The matrices originate from different domains and are in double precision. The testbed architecture is a 3.30GHz Intel®Core™i7-5820K processor We use OpenBLAS.0.2.19 [13] 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. We compare the performance of the Sympiler-generated code with CHOLMOD [1] as a specialized library for Cholesky factorization and with Eigen [7] as a general numerical library. CHOLMOD provides one of the fastest implementations of Cholesky factorization on single-core architectures [6]. Eigen supports a wide range of sparse and dense operations including sparse triangular solve and Cholesky.

Triangular solve: Figure 3 shows the performance of Sympiler-generated code compared to the Eigen library for a sparse triangular solve with a sparse RHS. 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 3 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. Whenever applicable, the vectorization and peeling low-level transformations are also 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. Matrices 3, 4, 5, and 7 do not benefit from the VS-Block transformation so their Sympiler run-times in Figure 3 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.

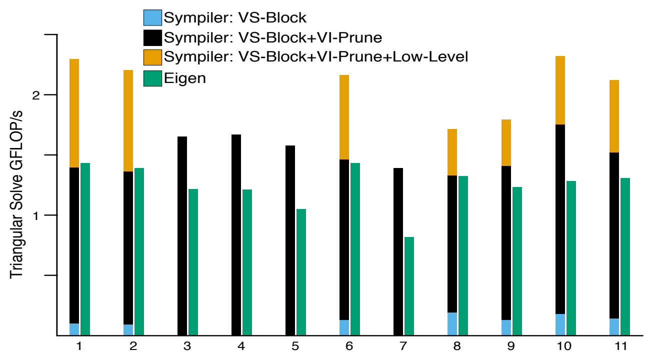


Figure 3: Sympiler's performance compared to Eigen for triangular solve. The stacked-bars show the performance of the Sympiler (numeric) code with VS-Block, VI-Prune, and low-level transformations.

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 4 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 4, for Cholesky factorization Sympiler performs up to 2.4× and 6.3× better than CHOLMOD and Eigen respectively. Eigen uses the left-looking non-supernodal approach therefore, its performance does not scale well for large matrices. CHOLMOD benefits from supernodes and thus performs well for large matrices with large supernodes. However, CHOLMOD does not perform well for some small matrices and large matrices with small supernodes. Sympiler provides the highest performance for almost all tested matrix types which demonstrates the power of sparsity-specific code generation.

The symbolic analysis is performed once for a specific sparsity pattern in Sympiler, thus its overheads amortize with repeat executions of the numerical code. However, even if the numerical code is executed only once, which is not common in scientific applications, our experiments for all tested matrices show that the accumulated symbolic+numeric time of Sympiler is close to Eigen for the triangular solve and faster than both Eigen and CHOLMOD for Cholesky. The Sympiler accumulated time is on average 1.27× slower than the Eigen triangular solve code. In Cholesky, Sympiler accumulated time is faster than CHOLMOD and Eigen on average 1.44× and 2.65× in order. Code generation and compilation cost on average 0.5× and 82× of the numeric time in order for Cholesky and triangular 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.

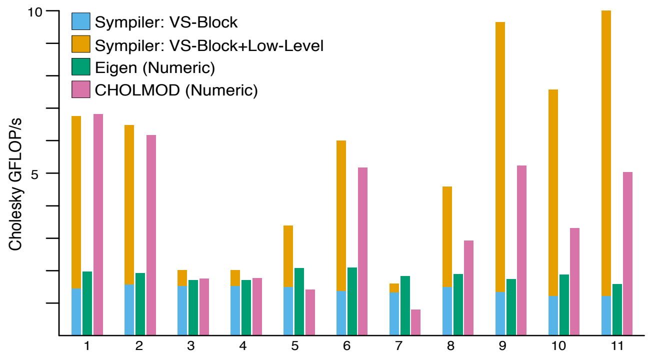


Figure 4: The performance of Sympiler (numeric) for Cholesky compared to CHOLMOD (numeric) and Eigen (numeric). The stacked-bar shows the performance of the Sympiler-generated code for each transformation.

6 CONTRIBUTIONS

In this paper we demonstrate 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, uses a novel approach for building compile-time symbolic inspectors that obtain information about a sparse matrix, to be used during compilation. It then leverage compile-time information 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.

REFERENCES

[1] 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.

[2] Timothy A Davis. 2006. *Direct methods for sparse linear systems.* Vol. 2. Siam.

[3] 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.

[4] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[5] 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.

[6] 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 TOMS* 33, 2 (2007), 10.

[7] Gaël Guennebaud and Benoit Jacob. 2010. Eigen. *URl: http://eigen. tuxfamily. org* (2010).

[8] Xiaoye S Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM TOMS* 31, 3 (2005), 302–325.

[9] Alex Pothen and Sivan Toledo. 2004. Elimination Structures in Scientific Computing. (2004).

[10] Michelle Mills Strout and et al. 2016. An approach for code generation in the sparse polyhedral framework. *Parallel Comput.* 53 (2016), 32–57.

[11] Ananta Tiwari and et al. 2009. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* IEEE, 1–12.

[12] Anand Venkat and et al. 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.

[13] Z Xianyi. 2016. OpenBLAS: an optimized BLAS library. (2016).