Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis

Kazem Cheshmi[1], Shoaib Kamil[2], Michelle Strout[3], Maryam Mehri Dehnavi[1]

Rutgers University[1], Adobe Research[2], University of Arizona[3]

OUTLINE

- Overview
- Sympiler: A code generator for optimizing sparse matrix methods
  - Sympiler internals (input, inspection, transformation, and code generation) with the triangular system solver example
  - Sympiler for Cholesky factorization
- Conclusion

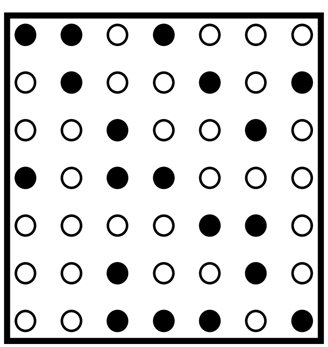
OUTLINE

- Overview
- Sympiler: A code generator for optimizing sparse matrix methods
  - Sympiler internals (input, inspection, transformation, and code generation) with the triangular system solver example
  - Sympiler for Cholesky factorizationil
- Conclusion

Linear Systems of Equations

Many simulations in scientific problems require solving a **linear system of equations**, $Ax = b$.

NXN matrix often sparse



Sparse A

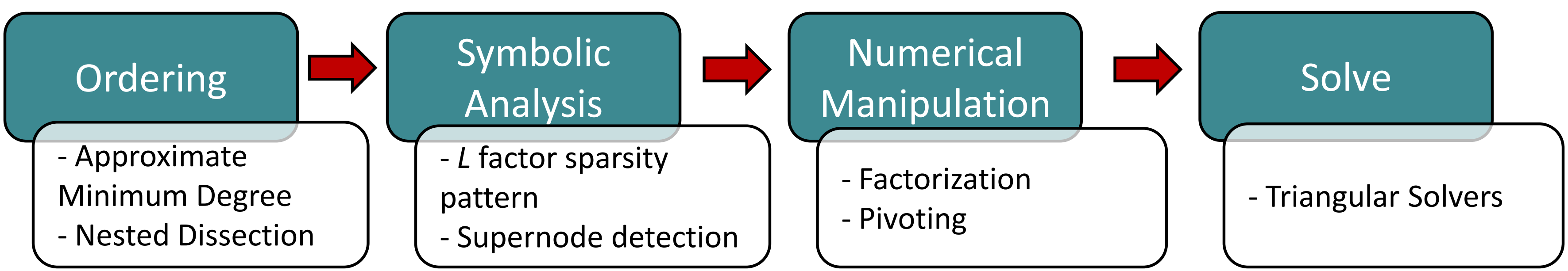
NX1 column vector of unknown

Represents whatever is "driving" the problem, e.g. boundary conditions, remote sources, etc.

Symbolic Analysis in Sparse Matrix Solvers

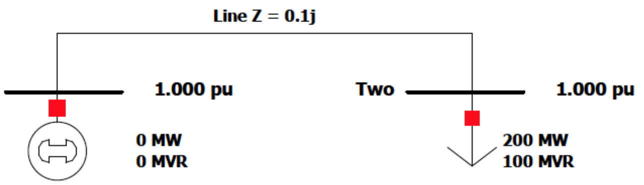
- Linear system solver classification:
  - Direct solvers
  - Iterative solvers

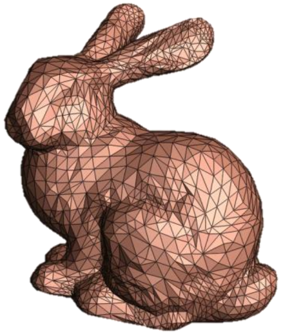
- Steps to factorizing a sparse matrix and solving a linear system using a direct method: Cholesky ($A=LL^T$), triangular solver ($Lx=b$)



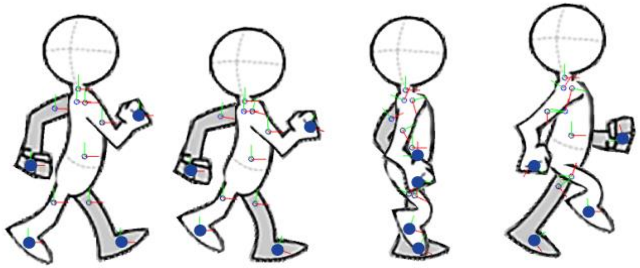
Static Sparsity Patterns

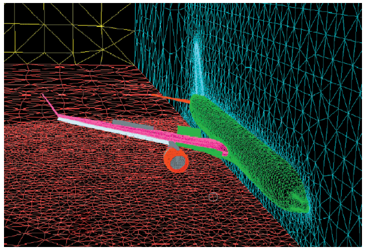
The **sparsity patterns** in many applications such as power modeling, animation, and circuit simulation often remain **static** for a period of time since the structure arises from the physical topology of the underlying system, the discretization, and the governing equations.





Sympiler generates optimized code for a static sparsity pattern by doing symbolic analysis at compile-time.





Library vs. Sympiler

Codes for Sparse Triangular Solve

Library

```
int top=Reach(Lp,Li,x,stack);
for (px=top; px>0; px--) {
  j=stack[px];
  x[j]/=Lx[Lp[j]]
  p=Lp[j]+1;
  for (; p<Lp[j+1]; p++)
    x[Li[p]]-=Lx[p]*x[j];
  }
```

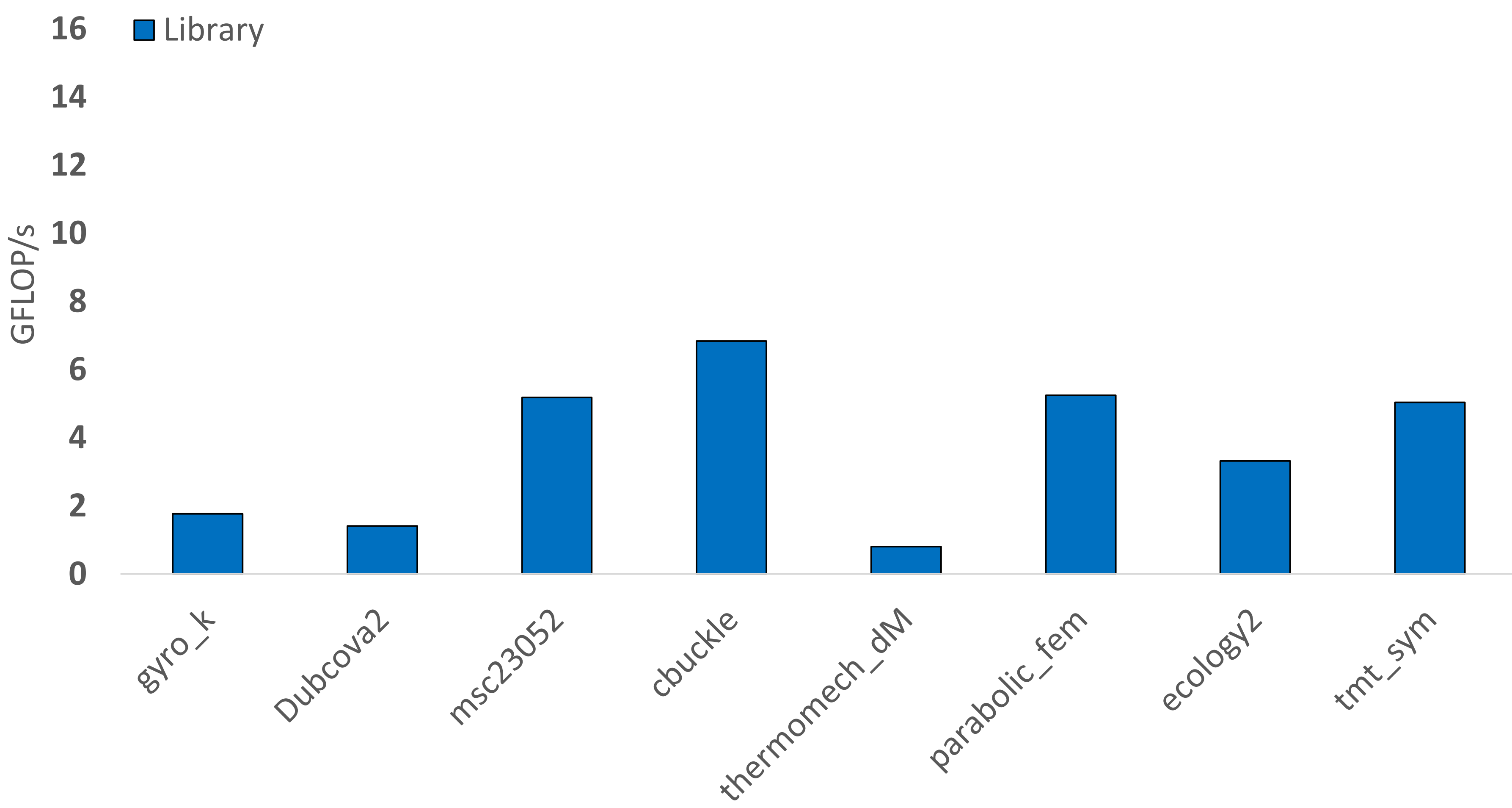
Symbolic analysis coupled with the numerical code

Sympiler

```
x=b;
x[0] /= Lx[0]; // Peel
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
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];
}
```

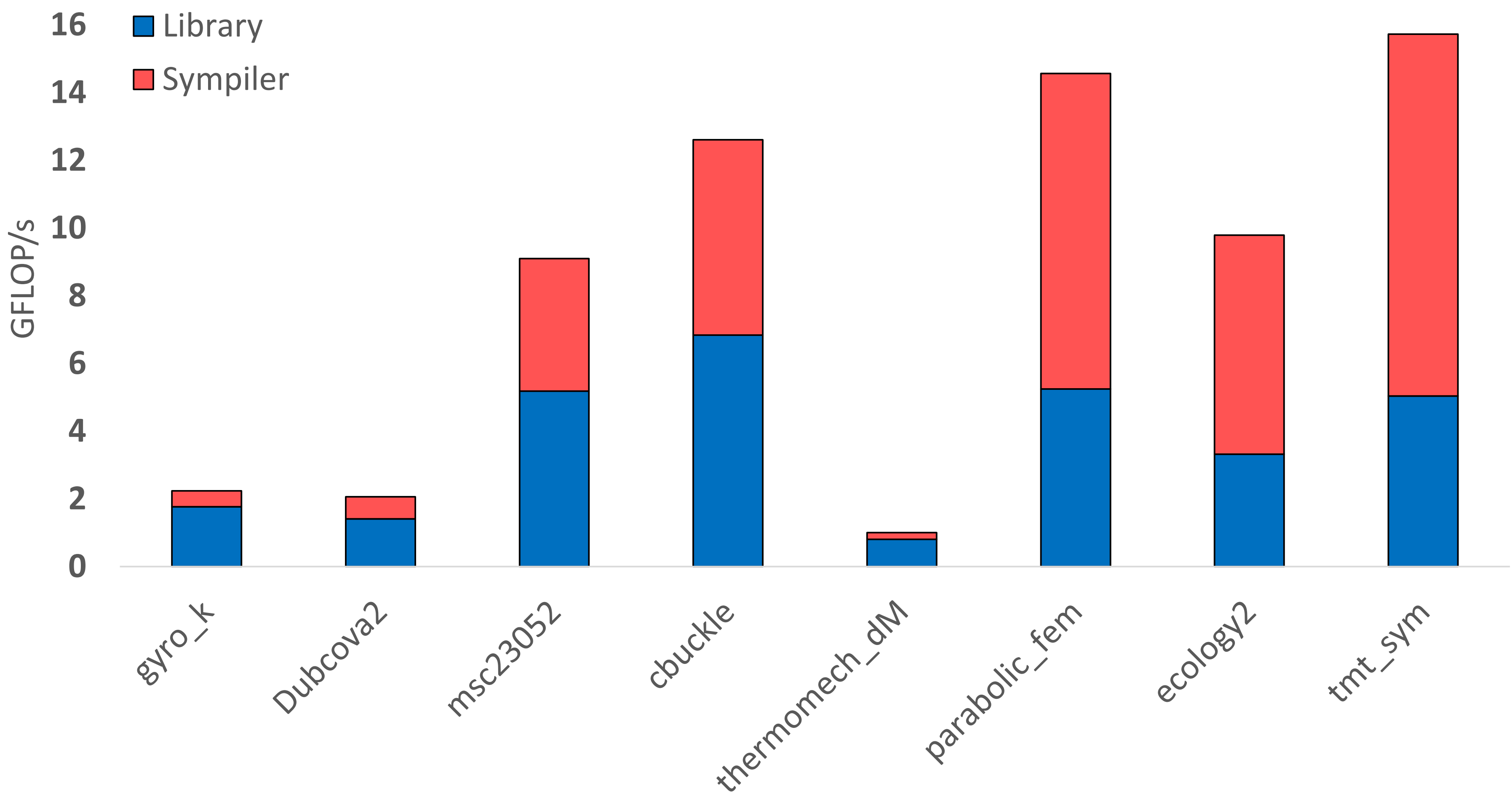
Sympiler-optimized numerical code

Cholesky Numeric Performance: Standard Library



The blue bar shows the performance for Eigen’s numeric code for selected benchmarks.

CHOLESKY NUMERIC PERFORMANCE: SYMPILER



The red bar shows Sympiler's added performance by generating optimized code for a static sparsity.

OUTLINE

- Overview
- Sympiler: A code generator for optimizing sparse matrix methods
  - Sympiler internals (input, inspection, transformation, and code generation) with the triangular system solver example
  - Sympiler for Cholesky factorization
- Conclusion

SYMPILER: A DOMAIN-SPECIFIC COMPILER FOR SPARSE DIRECT SOLVERS

Sympiler is a domain-specific compiler for generating high-performance code for direct sparse solvers

- The user provides the sparsity and type of solver as inputs.
- Sympiler decouples symbolic information from numerical computation at compile-time to generate optimized code.

Example: Find the solution to x, $Lx = b$ where L is sparse lower triangular matrix.

L: {n,Lp,Li,Lx}

$$\begin{bmatrix} 1 &&&&&&&&& \\ & 2 &&&&&&&& \\ & \bullet & 3 &&&&&&& \\ &&& 4 &&&&&& \\ &&& \bullet & 5 &&&&& \\ & \bullet & \bullet & \bullet & \bullet & 6 &&&& \\ \bullet &&&&&& 7 &&& \\ &&&&& \bullet & \bullet & 8 && \\ && \bullet & \bullet & \bullet &&& \bullet & 9 & \\ \bullet &&&&&&& \bullet & \bullet & 10 \end{bmatrix} * \overset{\text{x}}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{bmatrix}} = \overset{\text{b}}{\begin{bmatrix} \bullet \\ \\ \\ \\ \\ \bullet \\ \\ \\ \\ \\ \end{bmatrix}}$$

Solving a Sparse Triangular System

Solve *Lx=b* with *L* unit lower triangular; *L, x, b* are sparse.

$x = b$
for $j = 0$ to $n-1$ **do**
  if $x_j \neq 0$
    for each $i > j$ **for** which $l_{ij} \neq 0$ **do**
      $x_i = x_i - l_{ij} x_j$

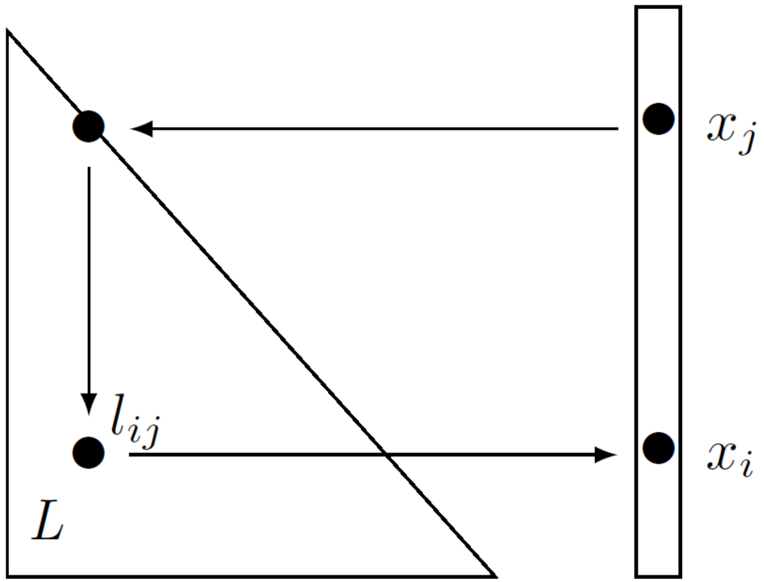
if $b_i \neq 0$ then $x_i \neq 0$

if $x_j \neq 0$ and $\exists i (l_{ij} \neq 0)$
then $x_i \neq 0$

start with pattern β of b

graph L: edge (j, i) if $l_{ij} \neq 0$

$\mathcal{X} = Reach_L(\beta)$



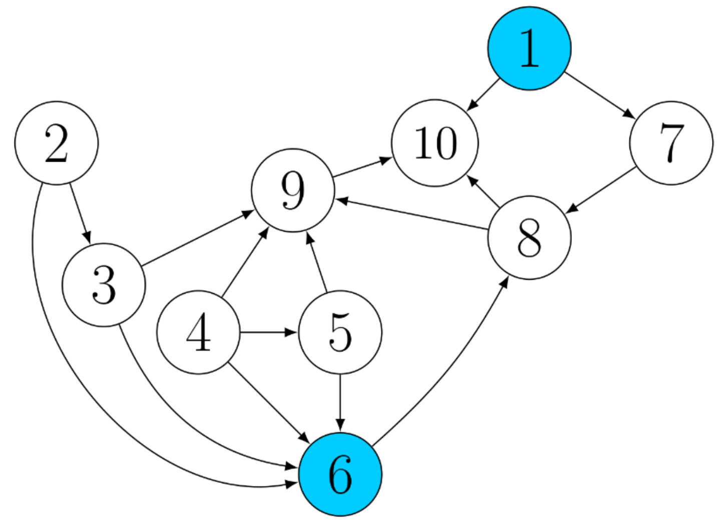
Adapted from "Direct Methods for Sparse Linear Systems"; Tim Davis; SIAM, 2006.

Symbolic Analysis in Sparse Triangular System Solver

L: {n,Lp,Li,Lx}

$$\begin{bmatrix}
1 &&&&&&&&& \\
& 2 &&&&&&&& \\
& \bullet & 3 &&&&&&& \\
&&& 4 &&&&&& \\
&&& \bullet & 5 &&&&& \\
& \bullet & \bullet & \bullet & \bullet & 6 &&&& \\
\bullet &&&&&& 7 &&& \\
&&&&& \bullet & \bullet & 8 && \\
&& \bullet & \bullet & \bullet &&& \bullet & 9 & \\
\bullet &&&&&&& \bullet & \bullet & 10
\end{bmatrix} *
\overset{\text{x}}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{bmatrix}} =
\overset{\text{b}}{\begin{bmatrix} \bullet \\ \\ \\ \\ \\ \bullet \\ \\ \\ \\ \\ \end{bmatrix}}$$

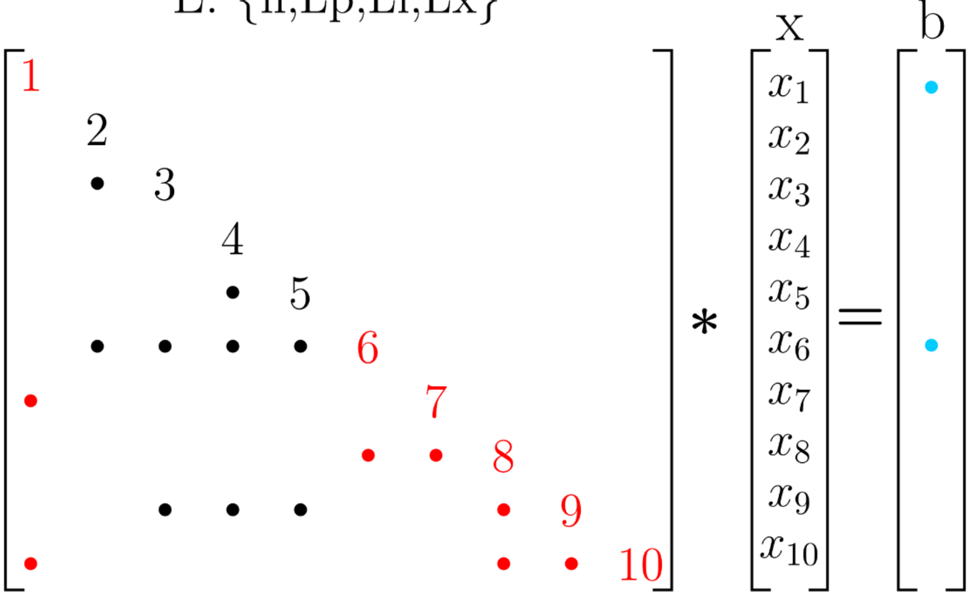
Dependence Graph (DG_L)



Symbolic Analysis in Sparse Triangular System Solver

Depth First Search (DFS)

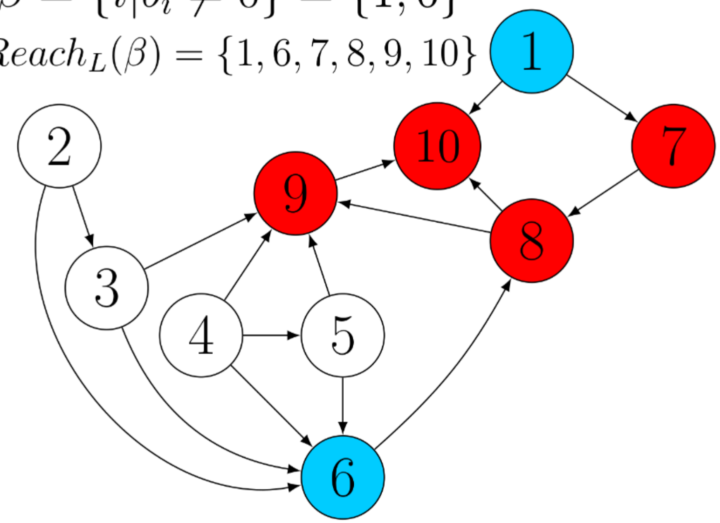
L: {n,Lp,Li,Lx}



Dependence Graph (DG_L)

$\beta = \{i | b_i \neq 0\} = \{1, 6\}$

$Reach_L(\beta) = \{1, 6, 7, 8, 9, 10\}$



Transforming The Sparse Code

Standard Code

```
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];
 }
```

Transforming The Sparse Code

Standard Code

```
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];
 }
```

Reachset = {6,1,7,8,9, 10}

Symbolically-Guided Code

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

Transforming The Sparse Code

Symbolically-Guided Code

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

TRANSFORMING THE SPARSE CODE

Symbolically-Guided Code

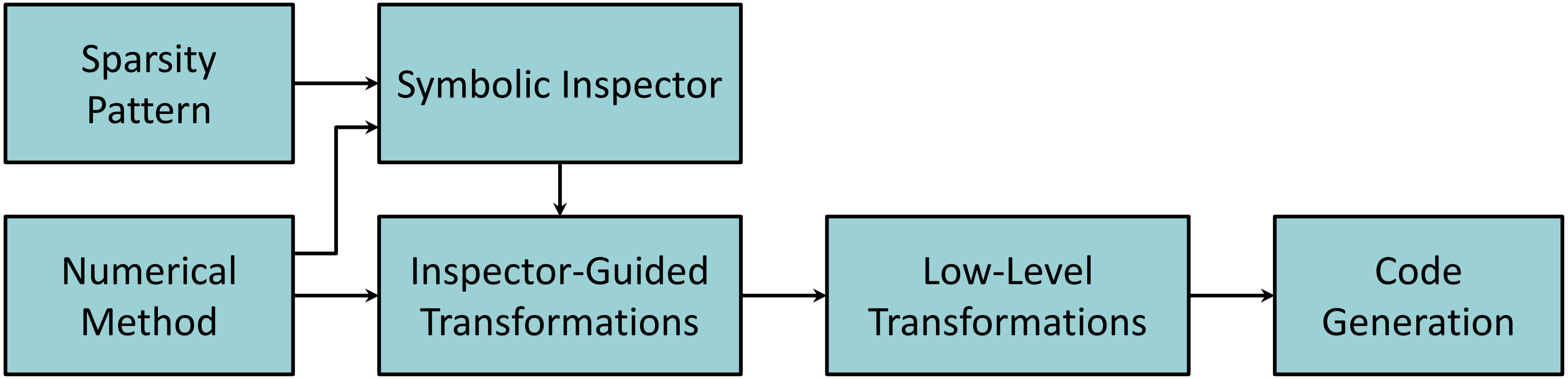
```
for (px=0; px<RSsize; px++) {
  j=reachset[px];
  x[j]/=Lx[Lp[j]]
  p=Lp[j]+1;
  for (; p<Lp[j+1]; p++)
    x[Li[p]]-=Lx[p]*x[j];
  }
```

Peel

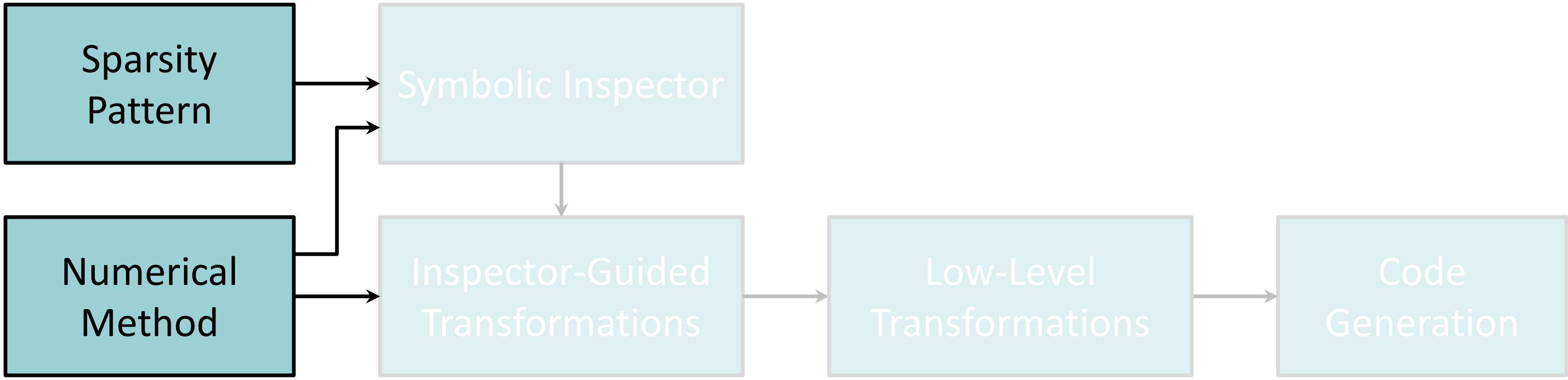
Optimized Code

```
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];
}
```

SYMPILER INTERNALS



INPUTS TO SYMPILER

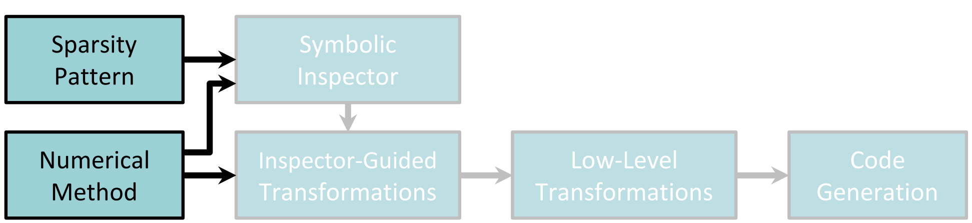


An Example Input Representation of the Sparse Triangular System Solver

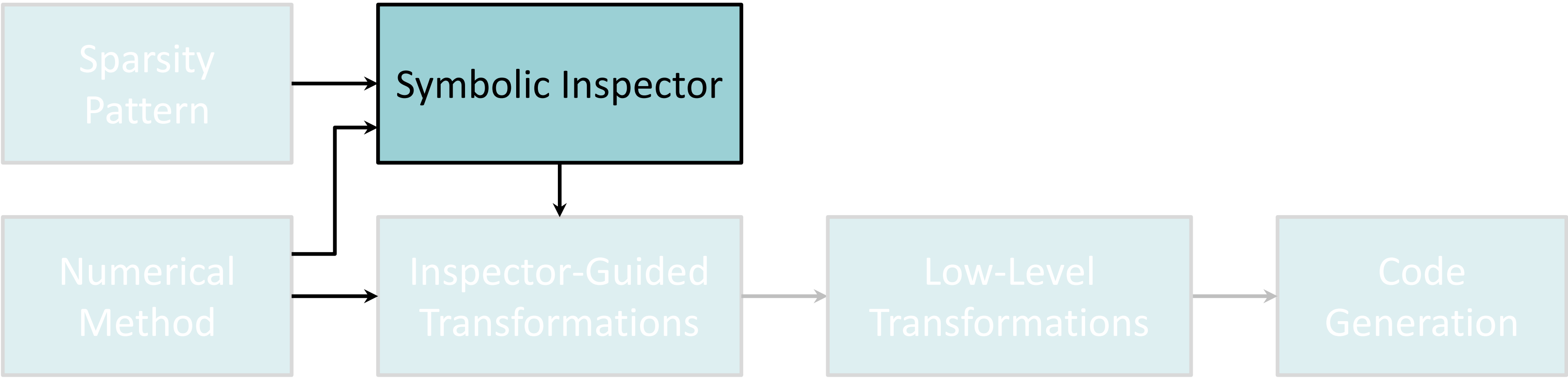
```
int main() {
 Sparse L(Float(64),“L.mtx");
 Sparse rhs(Float(64),"RHS.mtx");

 Triangular trns(L,rhs);
 trns.sympile_to_c("triang");
}
```

Input sparsity patterns

Numerical method

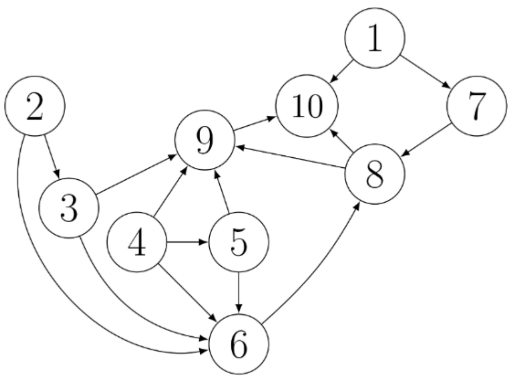


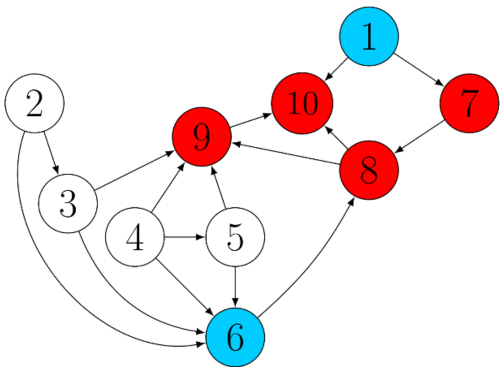
Symbolic Inspection



The Symbolic Inspector

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**.



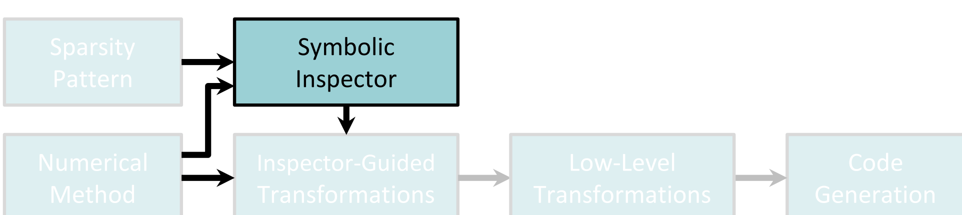


$$Reach_L(\beta) = \{1, 6,7,9,10\}$$

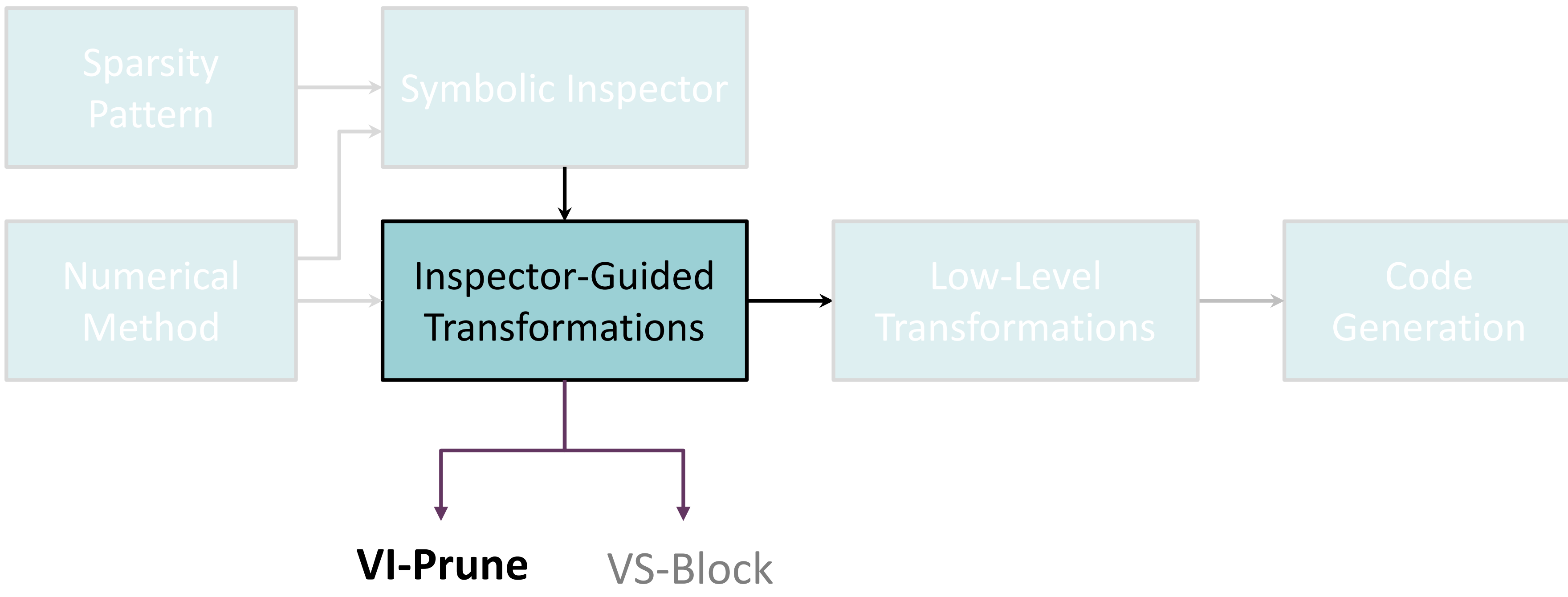
Inspection Graph: DG_L

Inspection Strategy: DFS

Inspection Set: Reachset



INSPECTOR-GUIDED TRANSFORMATIONS



Inspector-Guided Transformations: VI-Prune

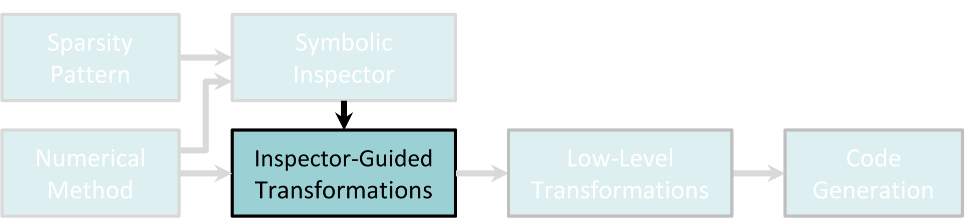
Variable Iteration Space Pruning (VI-Prune) prunes the iteration space of a loop using information about the sparse computation.

```
for(I_1){
 …
   for(I_k < m) {
     …
      for(I_n(I_k,..., I_{n-1})) {

          a[idx(I_1,...,I_k,...,I_n)];
      }
    }
}
```

Original code

```
for(I_1){
…
   for(I_p < pruneSetSize) {
        I'_k = pruneSet[I_p];
     …
      for(I_n(I'_k,..., I_{n-1})) {
          a[idx(I_1,...,I'_k,...,I_n)];
      }
    }
}
```

Transformed with VI-Prune

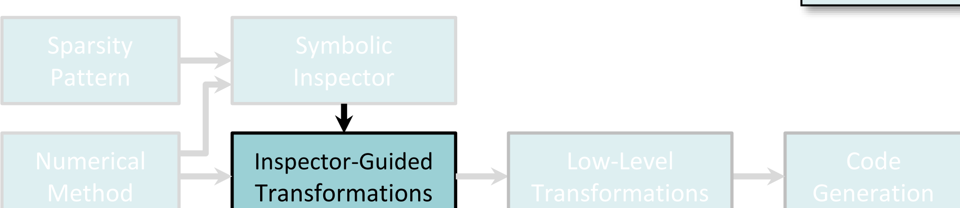


The Initial Abstract Syntax Tree (AST) with Annotations

```
int main() {
 Sparse L(Float(64),“L.mtx");
 Sparse rhs(Float(64),"RHS.mtx");

 Triangular trns(L,rhs);
 trns.sympile_to_c("triang");
}
```

```
VI-Prune
for sol.j0 in 0..Lsp.n
 VS-Block
x[bspj0] /= Lx[Lsp.diagj0)];
   VS-Block
    for sol.j1 in Lsp.colj0..Lsp.colj0+1
   x[Lsp.rowj1] -= Lx[j1]*x[bspj0];
```

Annotations for **Inspector-Guided transformations**



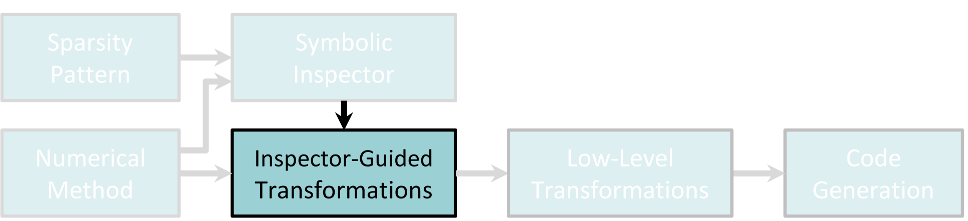
VI-Prune Transformation for Triangular Solve

```
VI-Prune
for sol.j0 in 0..Lsp.n
 VS-Block
 x[bspj0] /= Lx[Lsp.diagj0)];
 VS-Block
 for sol.j1 in Lsp.colj0..Lsp.colj0+1
   x[Lsp.rowj1] -= Lx[j1]*x[bspj0];
```

VI-Prune transformation

```
peel(0,3)
for sol.p0 in 0..pruneSetSize
j0=pruneSetp0;
x[bspj0]/=Lx[Lsp.diag(j0)];
vec(0,3)
for sol.j1 in Lsp.colj0..Lsp.colj0+1
  x[Lsp.rowj1]-=Lx[j1]*x[bspj0];
```

```
s0=pruneSet0;
x[bsps0]/=Lx[Lsp.diag(s0)];
for sol.j1 in Lsp.cols0..Lsp.cols0+1
x[Lsp.rowj1]-=Lx[j1]*x[bsps0];
for sol.p0 in 0..pruneSetSize
j0=pruneSetp0;
x[bspj0]/=Lx[Lsp.diag(j0)];
for sol.j1 in Lsp.colj0..Lsp.colj0+1
  x[Lsp.rowj1]-=Lx[j1]*x[bspj0];
```



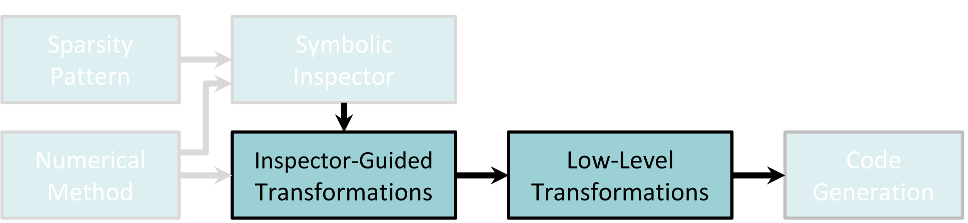
Low-Level Transformation for Triangular Solve

```
VI-Prune
for sol.j0 in 0..Lsp.n
 VS-Block
 x[bspj0] /= Lx[Lsp.diagj0)];
 VS-Block
 for sol.j1 in Lsp.colj0..Lsp.colj0+1
   x[Lsp.rowj1] -= Lx[j1]*x[bspj0];
```

```
peel(0,3)
for sol.p0 in 0..pruneSetSize
j0=pruneSetp0;
x[bspj0]/=Lx[Lsp.diag(j0)];
vec(0,3)
for sol.j1 in Lsp.colj0..Lsp.colj0+1
  x[Lsp.rowj1]-=Lx[j1]*x[bspj0];
```

Low-level transformation: Peel(0,3)

```
s0=pruneSet0;
x[bsps0]/=Lx[Lsp.diag(s0)];
for sol.j1 in Lsp.cols0..Lsp.cols0+1
x[Lsp.rowj1]-=Lx[j1]*x[bsps0];
for sol.p0 in 0..pruneSetSize
j0=pruneSetp0;
x[bspj0]/=Lx[Lsp.diag(j0)];
for sol.j1 in Lsp.colj0..Lsp.colj0+1
  x[Lsp.rowj1]-=Lx[j1]*x[bspj0];…
```

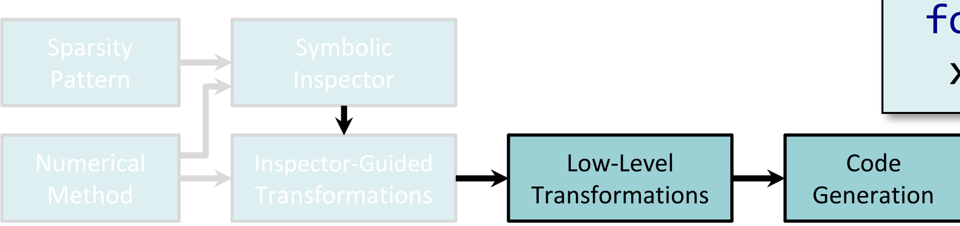


Code-Generation for Triangular Solve

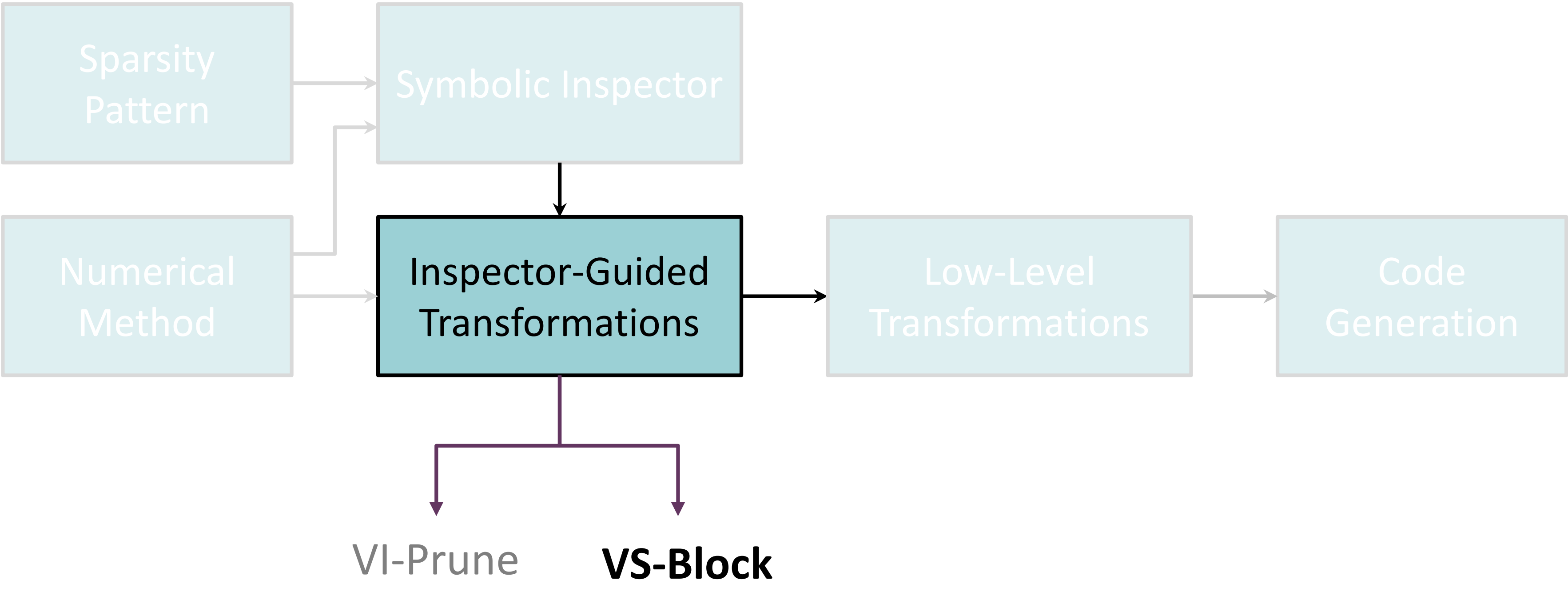
```
s_0=pruneSet_0;
x[bsp_s0]/=Lx[Lsp.diag(s_0)];
for sol.j_1 in Lsp.col_s0..Lsp.col_s0+1
 x[Lsp.row_j1]-=Lx[j_1]*x[bsp_s0];
for sol.p_0 in 0..pruneSetSize
j_0=pruneSet_p0;
 x[bsp_j0]/=Lx[Lsp.diag(j_0)];
 for sol.j_1 in Lsp.col_j0..Lsp.col_j0+1
   x[Lsp.row_j1]-=Lx[j_1]*x[bsp_j0];…
```

Final *C* code generation

```
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];}
```



INSPECTOR-GUIDED TRANSFORMATIONS



INSPECTOR-GUIDED TRANSFORMATIONS: VS-BLOCK

2D Variable-Sized Blocking (VS-Block) converts a sparse code to a set of non-uniform dense subkernels. VS-Block identifies supernodes.

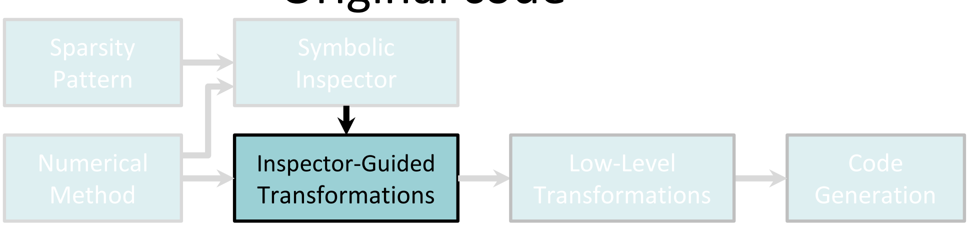
The unstructured computations and inputs in sparse kernels make blocking optimizations challenging: variable block sizes, algorithm change for diagonal operations, and the block elements may not be in consecutive locations.

```
for(I) {
  for(J) {

   B[idx1(I,J)] op1= a[idx2(I,J)];
  }
}
```

Original code

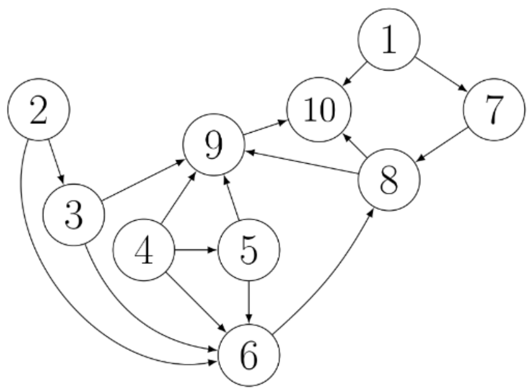
```
for(b < blockSetSize) {
 for(J1 < blockSet[b].x) {
   for(J2 < blockSet[b].y) {
     B[idx1(b,J1,J2)] op1 =
        A[idx2(b,J1,J2)];
    }
  }
}
```

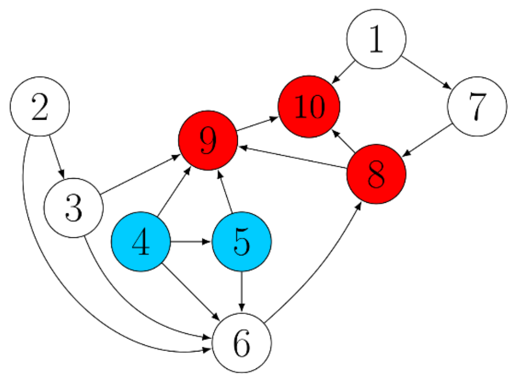
Transformed with VS-Block

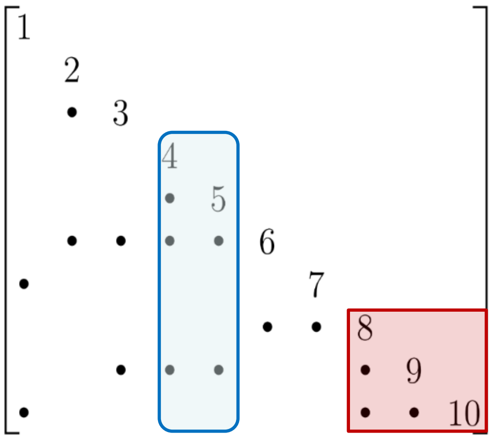


VS-Block for Triangular Solver

The Symbolic inspector detects supernodes (columns with similar sparsity patterns) and creates dense subkernels.



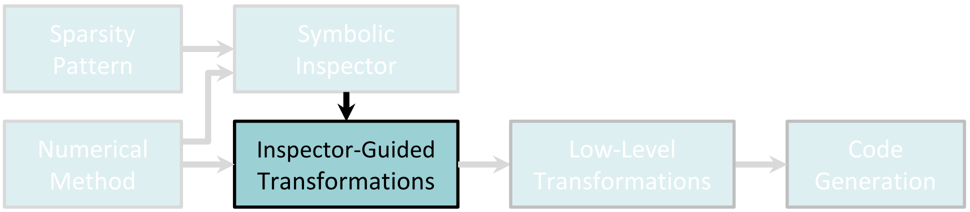




Inspection Graph: DG_L

Inspection Strategy: Node equivalence

Inspection Set: Blockset



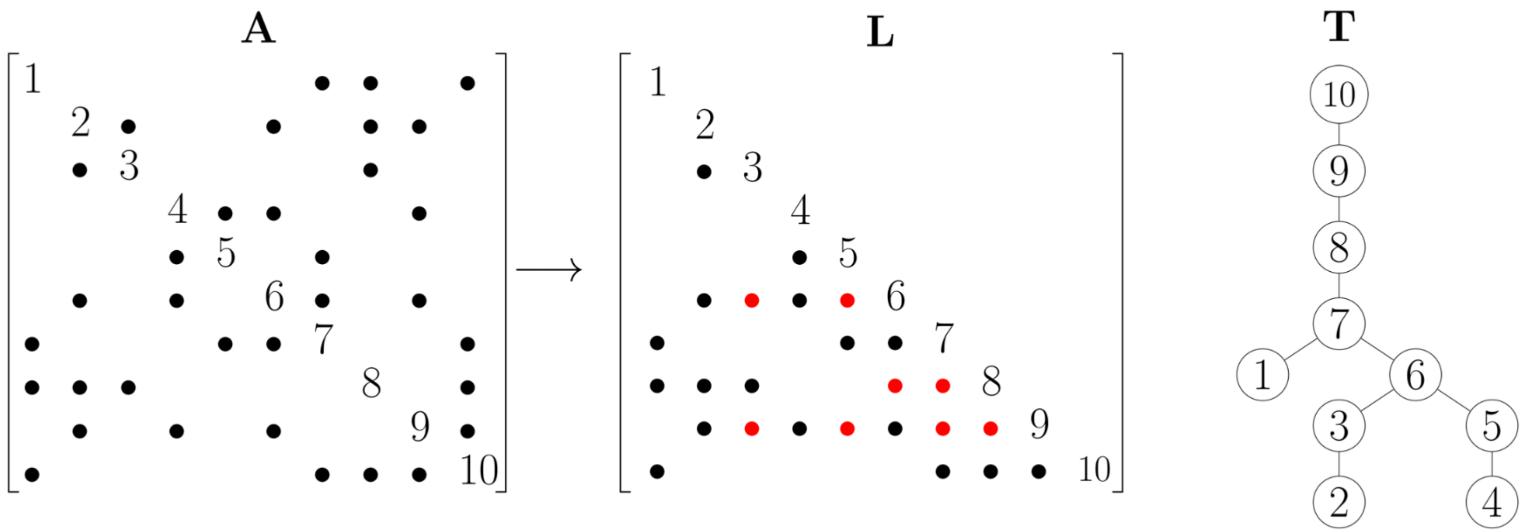
OUTLINE

- Overview
- Sympiler: A code generator for optimizing sparse matrix methods
  - Sympiler internals (input, inspection, transformation, codegen) with the triangular system solver example
  - Sympiler for Cholesky factorization
- Conclusion

CASE STUDY: CHOLESKY FACTORIZATION

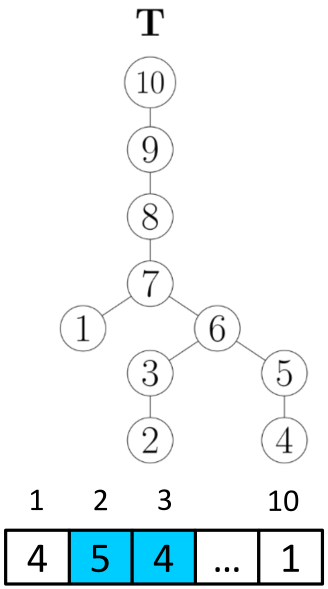
Cholesky factorization is commonly used in direct solvers and is used to precondition iterative solvers.

The elimination tree (T) is one of the most important graph structures used in the symbolic analysis of sparse factorization algorithms.

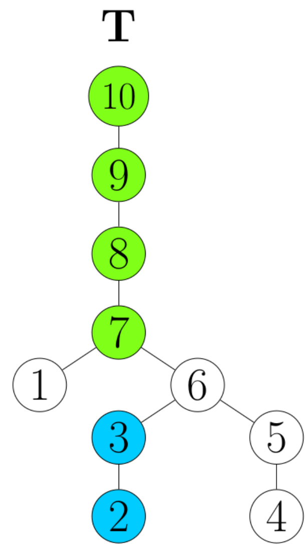


Symbolic Inspection for VS-Block in Cholesky

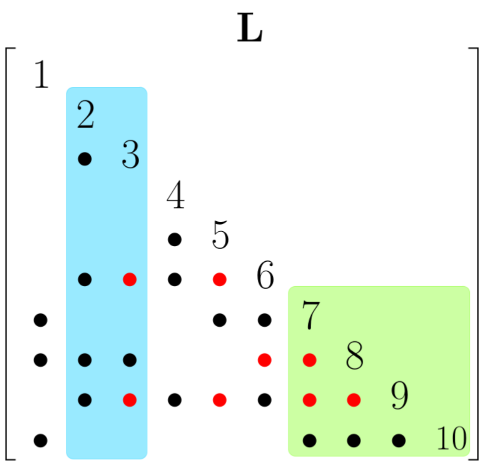
Supernodes in VS-Block for Cholesky are found using the L sparsity pattern and the elimination tree.



Inspection Graph: Elimination tree and column count



Inspection Strategy: Uptraversal



Inspection Set: Blockset

The VS-Block Transformation in Cholesky

```
for(column j = 0 to n){
 f = A(:,j);
 // Update
 for(r=0 to j-1 && L(j,1)!=0){
  f -= L(j:n,r) * L(j,r);
 }
 // Diagonal
 L(k,k) = sqrt(f(k));
 // Off-diagonal
 for(off-diagonal elements in f){
  L(k+1:n,k) = f(k+1:n) / L(k,k);
 }
}
```

VS-Block

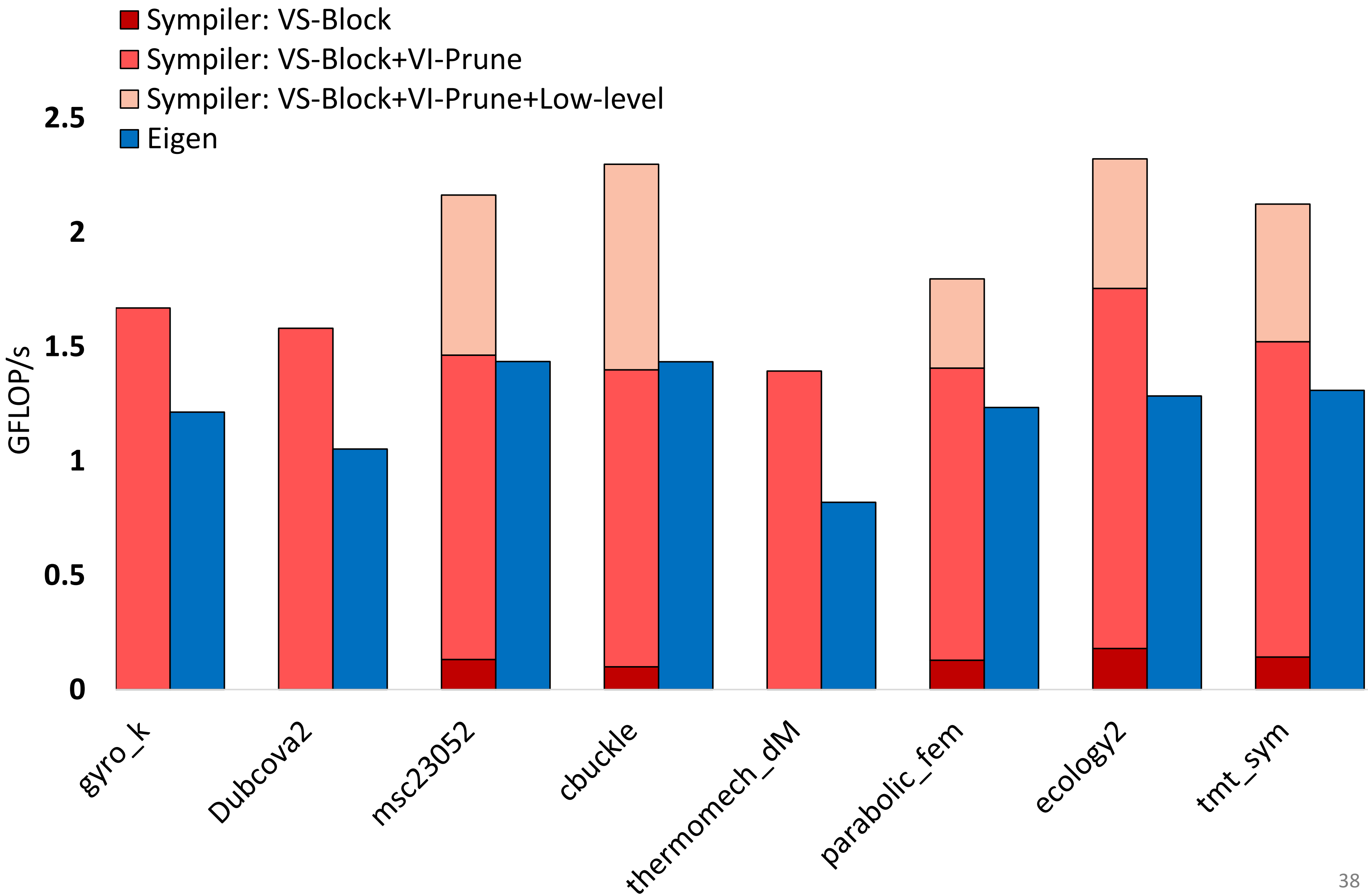
```
for(column j = 1 to blockSetNo){
lb = blockSet[j-1];
ub = blockSet[j];
f = A(:,lb:ub)
// Update
for(block r=0 to j-1 &&
L(j,r)!=0){
 f -= L(lb:n,r) * L'(j,r);
}
// Diagonal
L(lb:ub,lb:ub) =
Cholesky_dense(f(lb:ub));
// Off-diagonal
for(off-diagonal elements in f){
L(ub+1:n,lb:ub) =
triangular_dense(f(u+1:n,lb:ub),
 L(lb:ub,lb:ub));
}
}
```

Experimental Setup

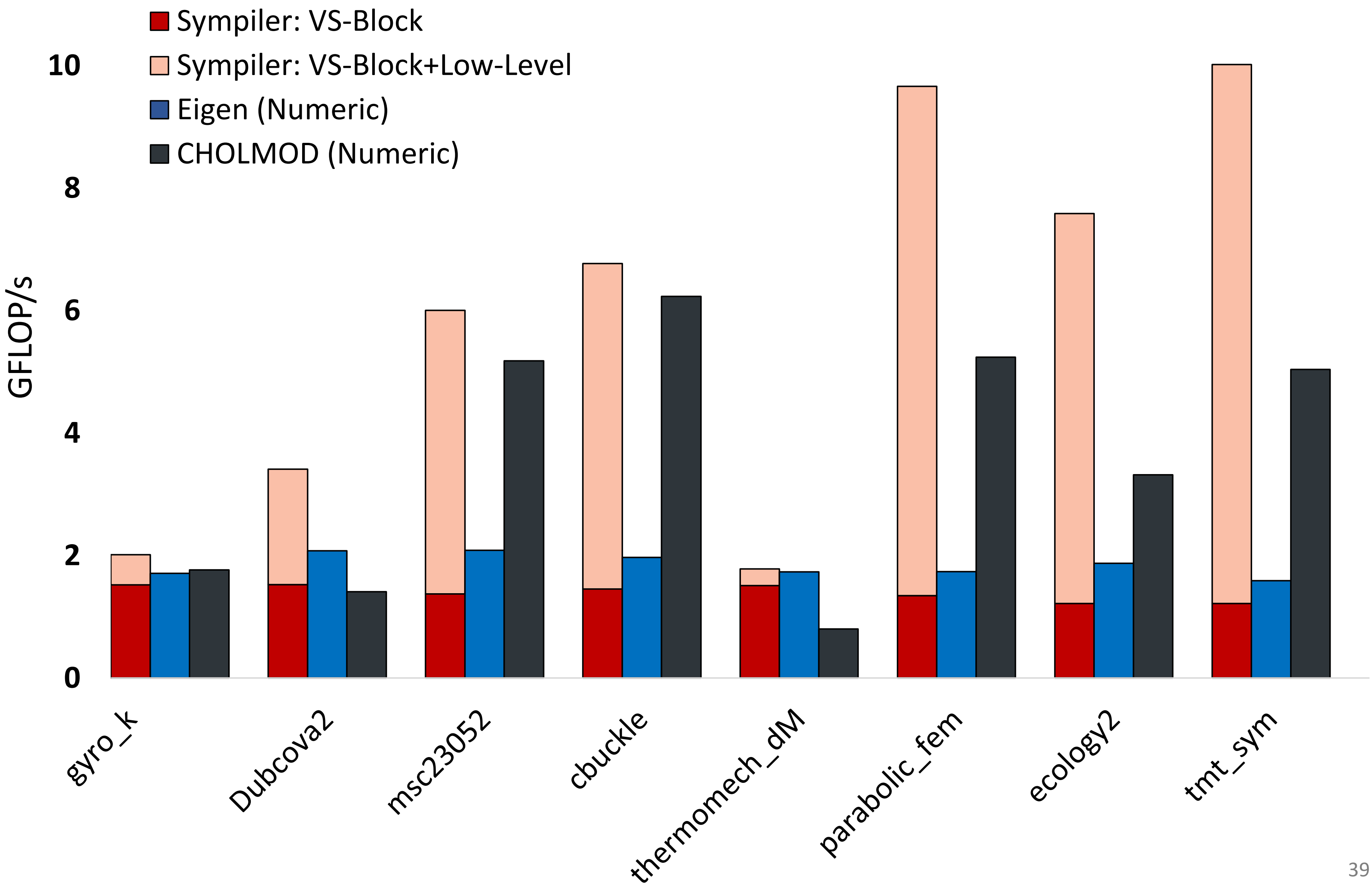
Numeric and symbolic times are compared separately where applicable. **Target processor**: Intel Core i7-5820K; **Benchmarks**: University of Florida repository

Name	Application	Order (10^3)	Non-zeros (10^6)
gyro_k	duplicate model reduction problem	17.4	1.02
Dubcova2	2D/3D problem	65.0	1.03
msc23052	structural problem	23.1	1.14
cbuckle	shell buckling	13.7	0.677
thermomech_dM	thermal problem	204	1.42
parabolic_fem	computational fluid dynamics problem	526	3.67
ecology2	circuitscape	1000	5.00
tmt_sym	electromagnetics	727	5.08

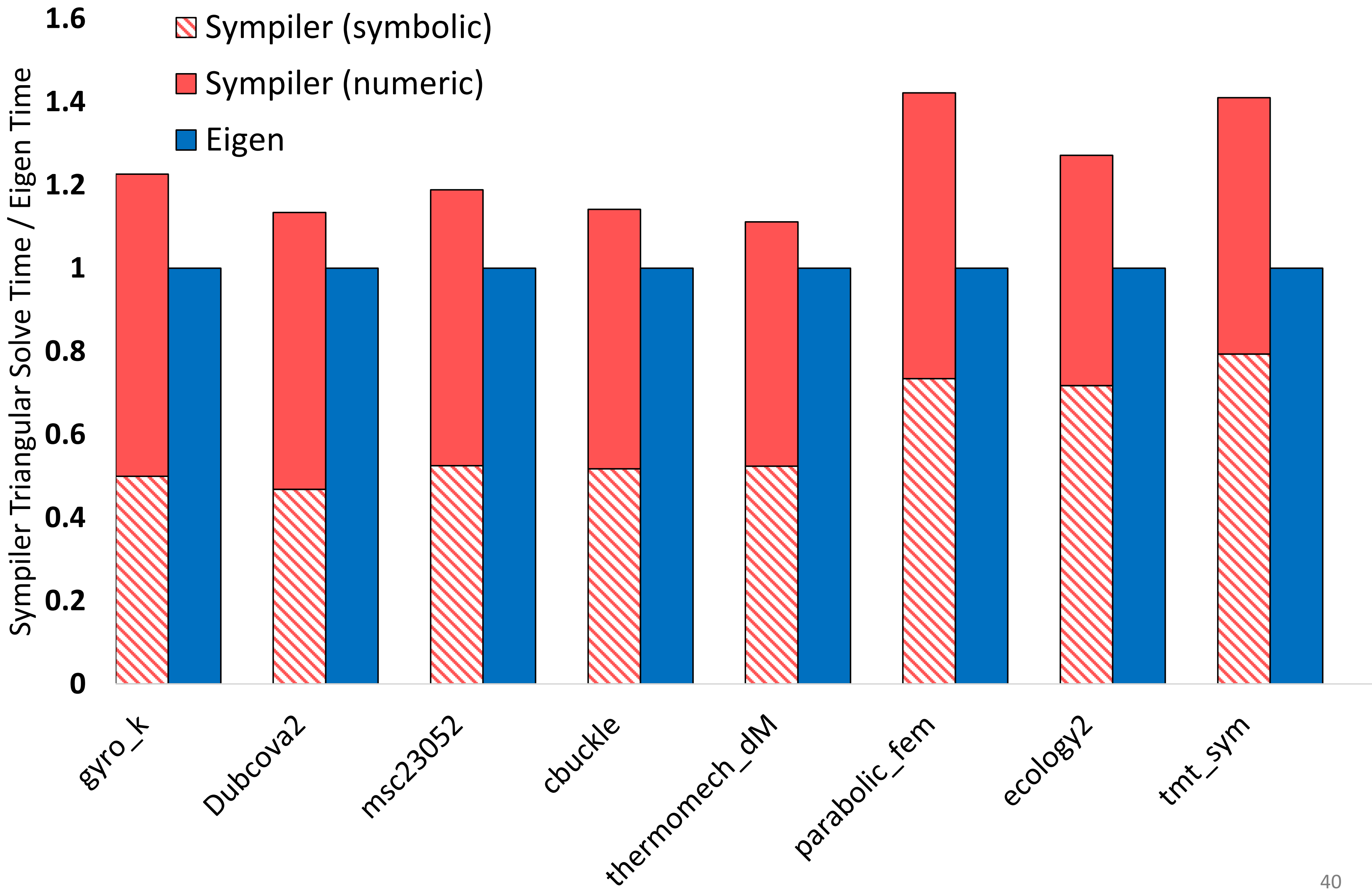
Triangular Solver Performance



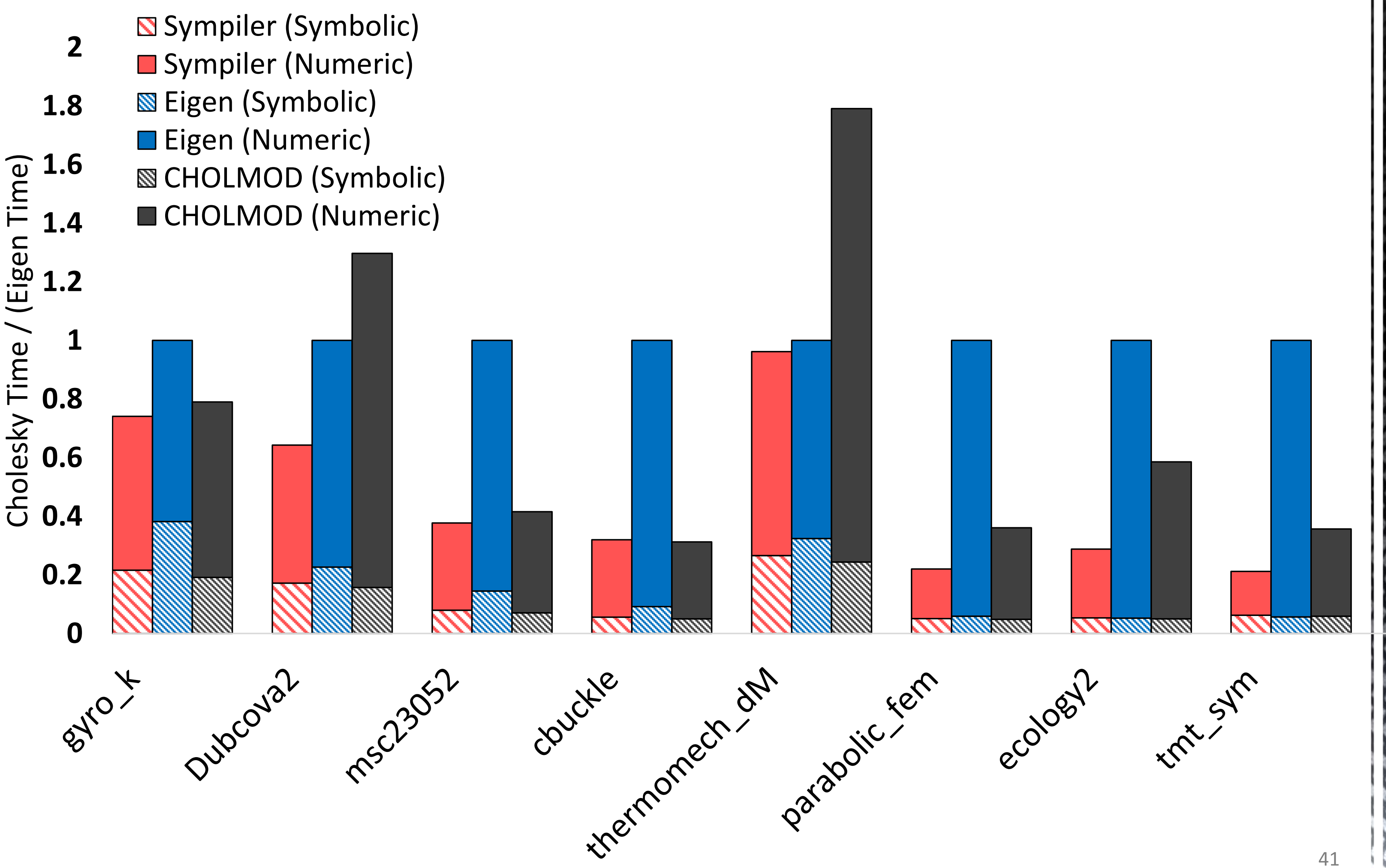
Cholesky Performance



SYMBOLIC ANALYSIS COST: TRIANGULAR SOLVER



SYMBOLIC ANALYSIS COST: CHOLESKY



Outline

- Overview
- Sympiler: A code generator for optimizing sparse matrix methods
  - Sympiler internals (input, inspection, transformation, codegen) with the triangular system solver example
  - Sympiler for Cholesky factorization
- Conclusion

Conclusion

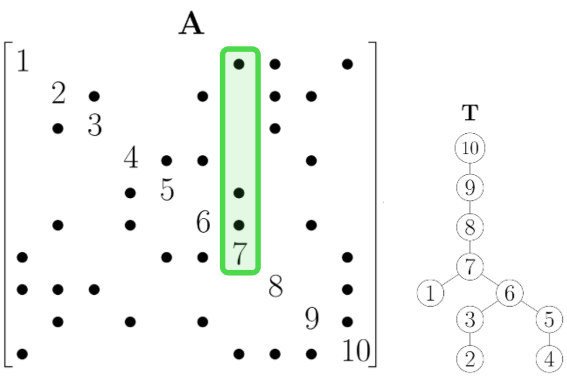
- Sympiler is a domain-specific code generator for transforming sparse matrix codes.
- It uses the information from symbolic analysis to apply a number of inspector-guided and low-level transformations.
- The Sympiler-generated code outperforms two state-of-the-art sparse libraries, Eigen and CHOLMOD.
- Sympiler source code is publicly available from:

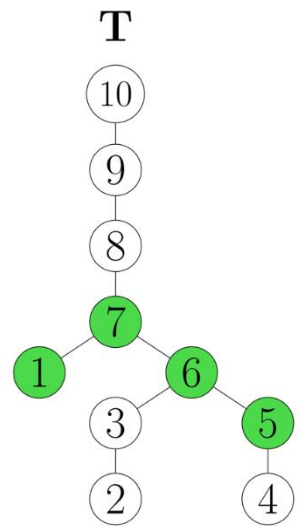
https://github.com/sympiler/sympiler

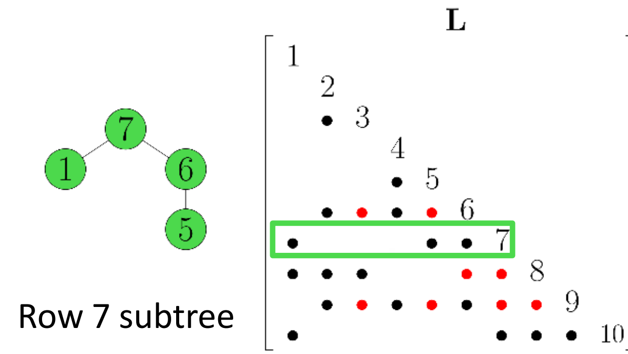
Thank you

Symbolic Inspection for VI-Prune in Cholesky

The Symbolic inspector uses the elimination tree and upper part of A to find row sparsity patterns to create the prune-set.







Inspection Graph: Elimination tree and upper A

Inspection Strategy: Single-node uptraversal

Inspection Set: Pruneset

The VI-Prune Transformation in Cholesky

```
for(column j = 0 to n){
 f = A(:,j)
 // Update
 for(r=0 to j-1 && L(j,1)!=0){
  f -= L(j:n,r) * L(j,r);
 }
 // Diagonal
 L(k,k) = sqrt(f(k));
 // Off-diagonal
 for(off-diagonal elements in f){
  L(k+1:n,k) = f(k+1:n) / L(k,k);
 }
}
```

VI-Prune

```
for(column j = 0 to n){
 f = A(:,j)
// Update
 for(every r in PruneSet){
  f -= L(j:n,r) * L(j,r);
 }
 // Diagonal
 L(k,k) = sqrt(f(k));
 // Off-diagonal
 for(off-diagonal elements in f){
L(k+1:n,k) = f(k+1:n) / L(k,k);
 }}
```