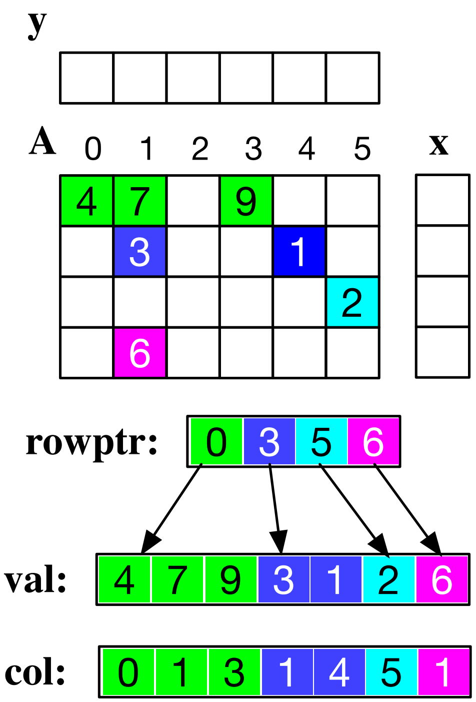
Loop Transformation Frameworks for Sparse Codes and Program Synthesis Opportunities

Michelle Mills Strout (University of Arizona),
Mary Hall (University of Utah), and
Catherine Olschanowsky (Boise State University)

THE UNIVERSITY OF ARIZONA

Sparse Codes are Hard to Optimize and Transform



y = A*x

```
// Dense matrix vector mult.
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++)
    y[i] +=  A[i][j] * x[j];
  }
}
```

```
// sparse matrix vector mult. (SpMV)
for (i=0; i<n; i++) {
  for(k=rowptr[i];k<rowptr[i+1];k++){
    y[i] += val[k]*x[col[k]];
  }
}
```

- Indirect accesses are slow
- Many different sparse formats
- Which sparse format is ideal depends on: algorithm, sparse structure, AND computation

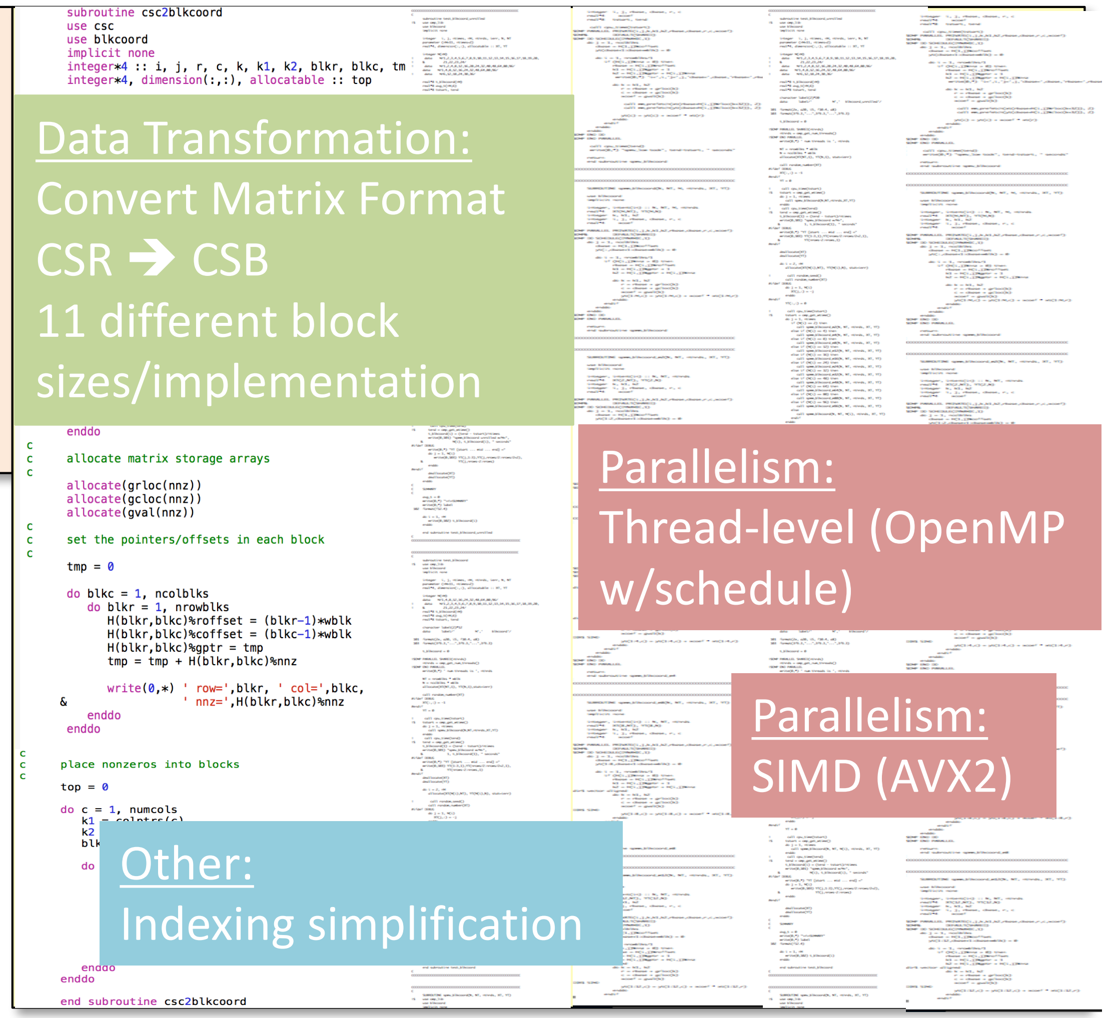
Current Approaches

- Developing new sparse formats and optimizations: HiCOO, sparse tiling, wavefront parallelization, ...
- Code generation from a DSL
  - Bernoulli compiler work
  - TACO work generates efficient implementations given a sparse tensor formats and a tensor expression
- Transforming existing code
  - Sparse Polyhedral Framework
  - CHiLL-I/E, scripting compiler for specifying inspector-executor transformations

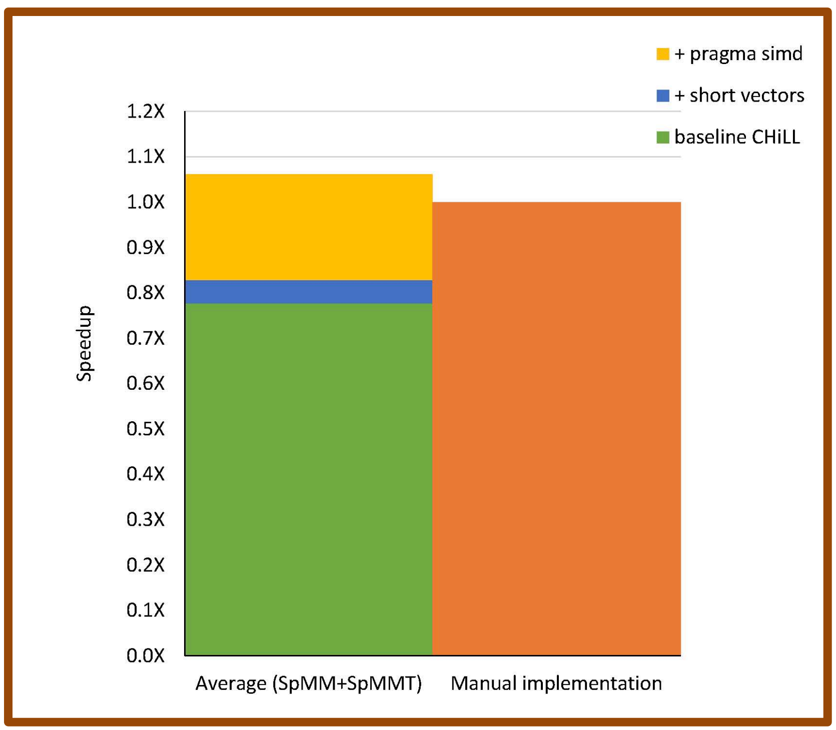
Transformation Example: SpMM from LOBPCG (NUCLEI)

```
/* SpMM from LOBCG on symmetric matrix */
for( i =0; i < n ; i ++) {
  for ( j = index [ i ]; j < index [ i +1]; j ++)
   for( k =0; k < m ; k ++);
      y [ i ][ k ]+= A [ j ]* x [ col [ j ]][ k ];
   /* transposed computation exploiting symmetry*/
   for ( j = index [ i ]; j < index [ i +1]; j ++)
      for( k =0; k < m ; k ++)
         y [ col [ j ]][ k ]+= A [ j ]* x [ i ][ k ];
}
```

Code A: Multiple SpMV computations (SpMM), 7 lines of code

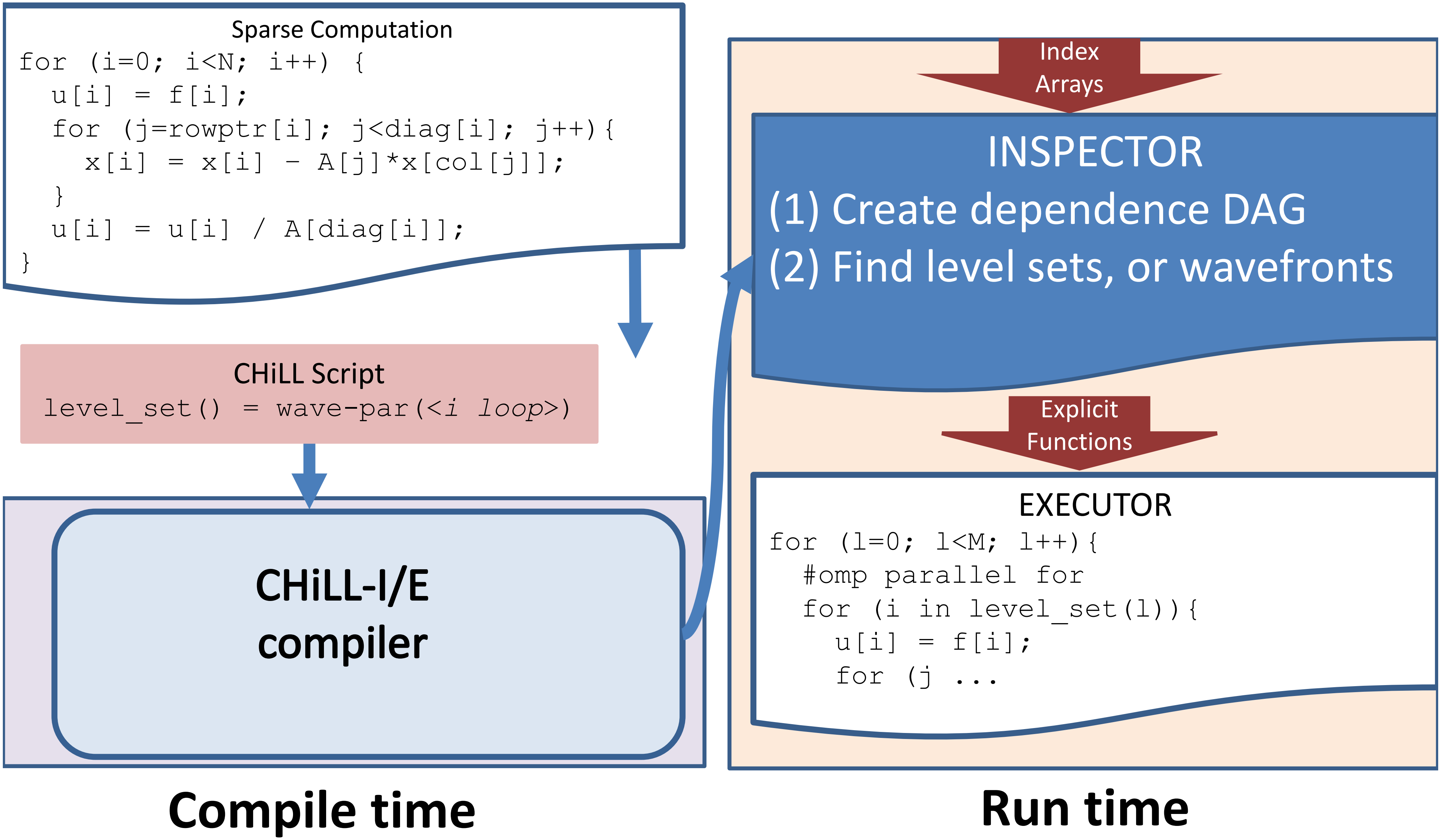


Code B: Manually-optimized SpMM from LOBCG, 2109 lines of code



Take-away message: Compiler-optimized Code A faster than manual Code B!

CHiLL-I/E: Inspector-Executor Transformations



Opportunities to Leverage Synthesis Tools?

- Constraint-solving-based synthesis techniques
  - Polyhedral model uses Farkas lemma to derive scheduling constraints from data dependences
  - Sparse Polyhedral Framework can produce constraints for the uninterpreted functions the inspector must produce at runtime

- Run-time realization of uninterpreted functions
  - Could be synthesized to specialize for usage
  - Data structure synthesis tools like Cozy

Deriving constraints for uninterpreted functions

- Constraint-based data dependence analysis
- Transformations introduce new uninterpreted functions and modify data dependences
- Convert data dependence relations into constraints on uninterpreted functions

7

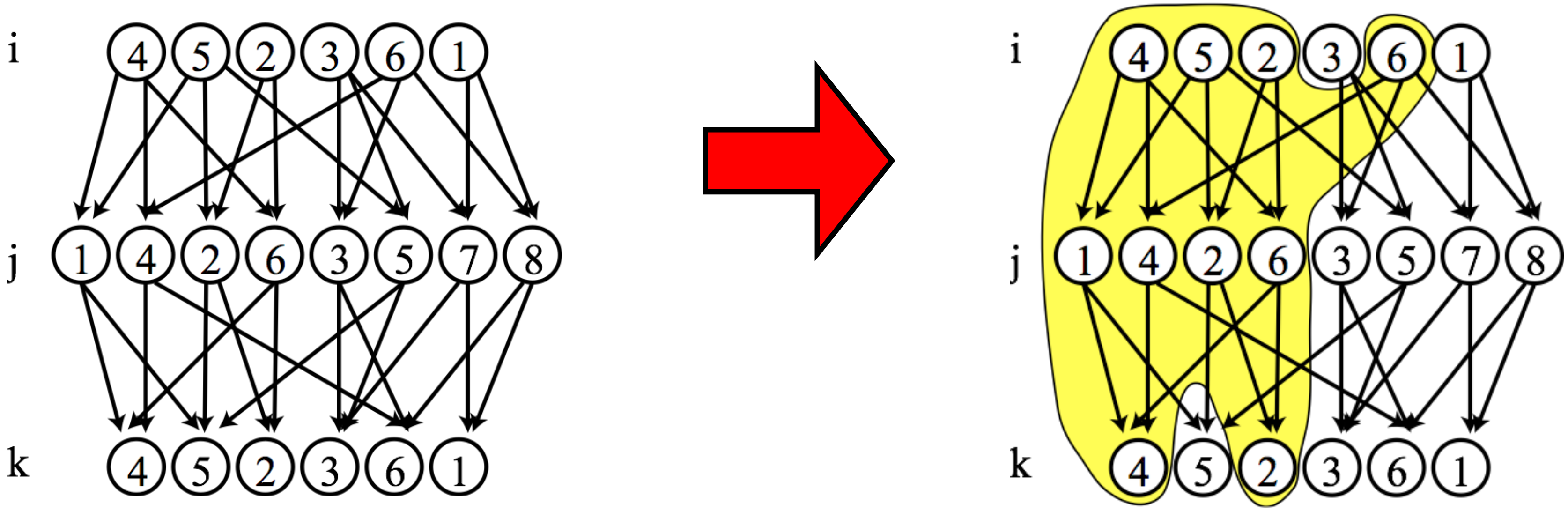
Constraint-Based Data Dependence Analysis of Sparse Computation

```
for (int j=0; j<n; j++){
  x[j] = x[j] / Lx[colPtr[j]];
  for(int p=colPtr[j]+1; p<colPtr[j+1]; p++){
    x[row[p]] = x[[row[p]] - Lx[p] * x[j];
```

$$\{[j,p] \rightarrow [j',p'] : \overbrace{j = j' \wedge p < p'}^{lexicographical\ Ordering} \wedge \overbrace{row(p) = j'}^{Array\ Access\ Equality} \wedge$$

$$\underbrace{0 \leq j, j' < n \wedge colPtr(j) < p < colPtr(j+1) \wedge colPtr(j') < p' < colPtr(j'+1)\}}_{Loop\ Bounds}$$

Example Transformation Introducing an Uninterpreted Function



$$T_{F_1 \to F_2} = \{[s,0,i] \to [s,0,t,0,i] \mid t = \Theta(0,i)\}$$
$$\cup \ \{[s,1,j] \to [s,0,t,1,j] \mid t = \Theta(1,j)\} \cdots$$

$$F_1 = \{[s,0,i]\} \cup \{[s,1,j]\} \cup \{[s,2,k]\}$$

$$F_2 = \{[s,0,t,0,i] \mid t = \Theta(0,i)\} \cup \{ [s,0,t,1,j] \mid t = \Theta(1,j)\} \cdots$$

Transformed Dependences Need to be Lexicographically Non-Negative

$$D_{I_0 \to J_0} = \{[s,0,i] \to [s,1,j] \quad | \quad i = l(j) \vee \; i = r(j)\}$$

$$\begin{aligned} T_{F_1 \to F_2} \quad = & \quad \{[s,0,i] \to [s,0,t,0,i] \mid t = \Theta(0,i)\} \\ \cup & \quad \{[s,1,i] \to [s,0,t,1,j] \mid t = \Theta(1,j)\} \cdots \end{aligned}$$

$$\begin{aligned} D_{I_0 \to J_0} = \{[s,0,t_1,0,i] \to [s,0,t_2,1,j] \quad | \quad & (t_1 = \Theta(0,i) \wedge t_2 = \Theta(1,j) \wedge i = l(j)) \\ & \vee (t_1 = \Theta(0,i) \wedge t_2 = \Theta(1,j) \wedge i = r(j))\} \end{aligned}$$

Constraints Derived from Dependence

$$D_{I_0 \to J_0} = \{[s,0,t_1,0,i] \to [s,0,t_2,1,j] \quad | \quad (t_1 = \Theta(0,i) \wedge t_2 = \Theta(1,j) \wedge i = l(j)) \\ \vee (t_1 = \Theta(0,i) \wedge t_2 = \Theta(1,j) \wedge i = r(j))\}$$

⇓

$$\forall s, t_1, t_2, i, j \ : \ (i = l(j) \vee i = r(j)) \Rightarrow \Theta(0,i) \leq \Theta(1,j)$$

If iteration i must be executed before iteration j, then iteration i must be in the same or earlier tile than j.

Summary: Synthesis and Transformed Sparse Codes

- Use dependence analysis of original code and inspector-executor transformations to create constraints
- Remains to be seen how these constraints can be used to synthesize inspector code
- Use data structure synthesis to generate specialized implementations of run-time realizations of uninterpreted functions (not discussed)