

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)



Sparse Codes are Hard to Optimize and Transform

y

A

0	1	2	3	4	5
4	7	9			
	3		1		
				2	
		6			

x

$y = A^*x$

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

rowptr:

val:

col:

```
// 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

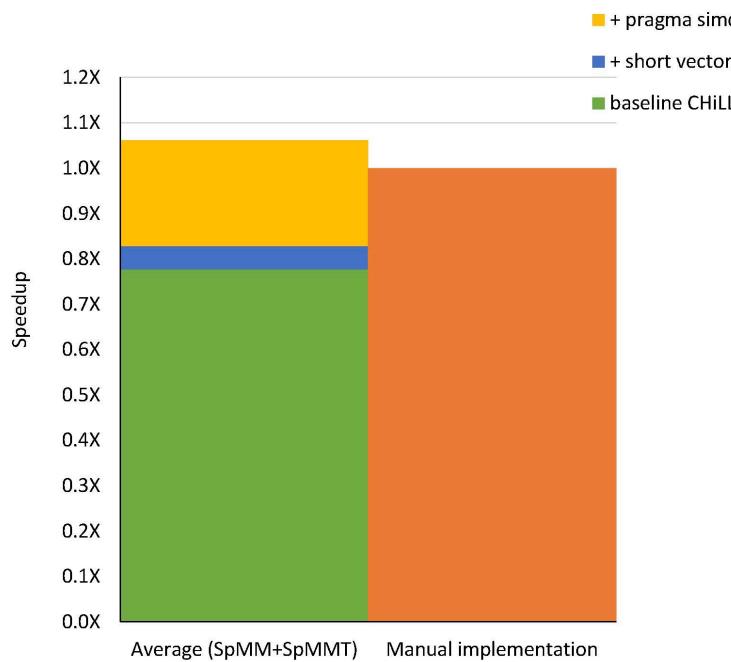
- Developing new sparse formats and optimizations: HiCOO, sparse tiling, waveform parallelization, ...
- Code generation from a DSL
 - Bernoulli compiler work
 - TACO work generates efficient implementations given a sparse tensor formats and a tensor expression
- Transforming existing code
 - Sparse 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



```
subroutine csc2blkcoord
use csc
use blkcoord
implicit none
integer*4 :: i, j, r, c, k, k1, k2, blkr, blkc, tm
integer*4, dimension(:,:,:), allocatable :: top
```

Data Transformation:
Convert Matrix Format
CSR → CSB
11 different block sizes/implementation

```
enddo
c      allocate matrix storage arrays
c      allocate(gloc(nnz))
c      allocate(gcol(nnz))
c      allocate(gval(nnz))
c      set the pointers/offsets in each block
c      tmp = 0
do blkc = 1, ncolblk
    do blkr = 1, nrowblk
        H(blkr,blkc)%roffset = (blkr-1)*wblk
        H(blkr,blkc)%coffset = (blkc-1)*wblk
        H(blkr,blkc)%gptr = tmp
        tmp = tmp + H(blkr,blkc)%nnz
        write(0,*)
        & ' row=' , blkr, ' col=' , blkc,
        & ' nnz=' , H(blkr,blkc)%nnz
    enddo
enddo
c      place nonzeros into blocks
top = 0
do c = 1, numcols
    k1 = colstarts(c)
    k2 = colends(c)
    blkc = c
    do
        enddo
    enddo
end subroutine csc2blkcoord
```

Other:
Indexing simplification

A screenshot of a large, complex Fortran code editor. The interface shows multiple windows of code, likely representing different parts of the manually-optimized SpMM implementation. The code is written in Fortran 90/95 style with various subroutines and modules.

Parallelism:
Thread-level (OpenMP w/schedule)

Parallelism:
SIMD (AVX2)

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

Sparse Computation

```
for (i=0; i<N; i++) {  
    u[i] = f[i];  
    for (j=rowptr[i]; j<diag[i]; j++) {  
        x[i] = x[i] - A[j]*x[col[j]];  
    }  
    u[i] = u[i] / A[diag[i]];  
}
```

CHiLL Script

```
level_set() = wave-par(<i loop>)
```

CHiLL-I/E
compiler

Compile time

Index
Arrays

INSPECTOR

- (1) Create dependence DAG
- (2) Find level sets, or wavefronts

Explicit
Functions

EXECUTOR

```
for (l=0; l<M; l++) {  
    #omp parallel for  
    for (i in level_set(l)) {  
        u[i] = f[i];  
        for (j ...
```

Run time

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



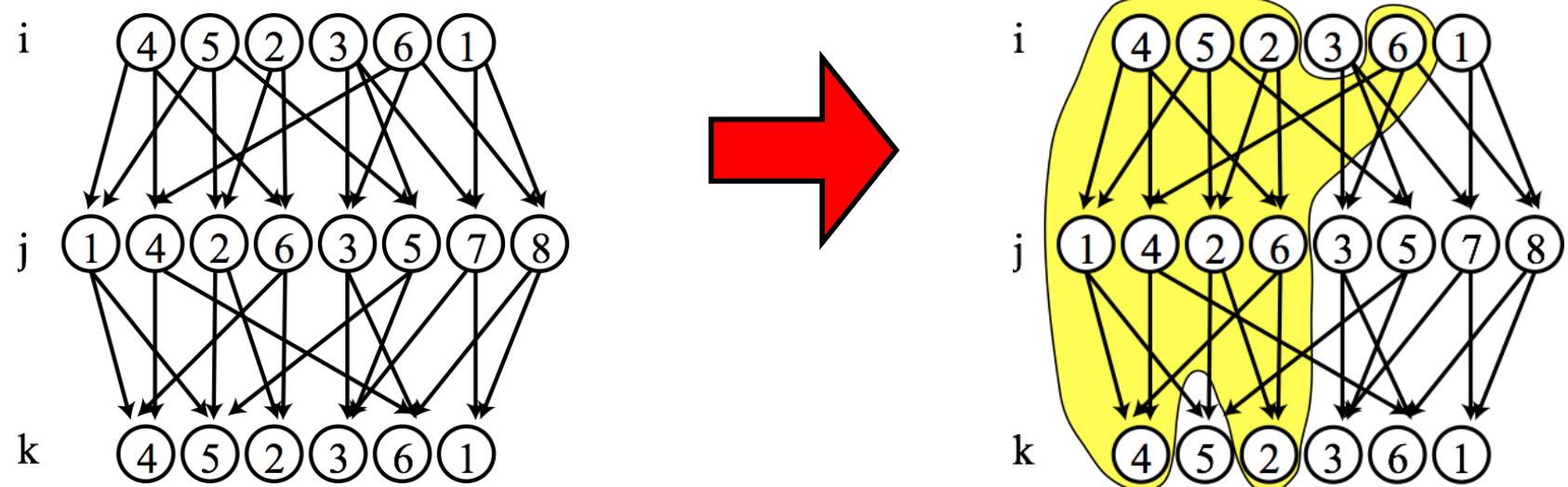
Constraint-Based Data Dependence Analysis of Sparse Computation

```
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];
```

$$\underbrace{\{[j, p] \rightarrow [j', p'] : \overbrace{j = j' \wedge p < p'}^{\text{lexicographical Ordering}} \wedge \overbrace{\text{row}(p) = j'}^{\text{Array Access Equality}} \wedge}_{\text{Loop Bounds}} \\ 0 \leq j, j' < n \wedge \underbrace{\text{colPtr}(j) < p < \text{colPtr}(j + 1) \wedge \text{colPtr}(j') < p' < \text{colPtr}(j' + 1)}_{\text{Loop Bounds}}\}$$



Example Transformation Introducing an Uninterpreted Function



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

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



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

Transformed Dependences Need to be Lexicographically Non-Negative

$$D_{I_0 \rightarrow J_0} = \{[s, 0, i] \rightarrow [s, 1, j] \mid i = l(j) \vee i = r(j)\}$$



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



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



Constraints Derived from Dependence

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



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

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)

