The Sparse Polyhedral Framework:

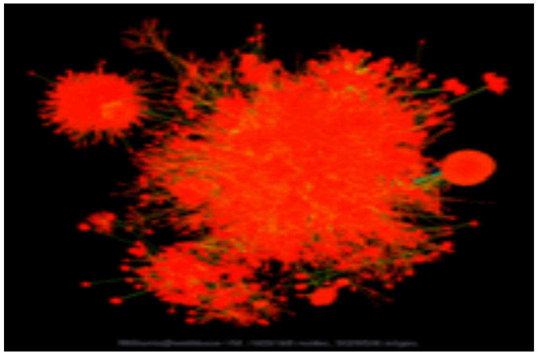
Composing compiler-generated inspector-executor code

Prof. Michelle Mills **Strout** (University of Arizona)
Prof. Mary Hall (University of Utah)
Dr. Catherine Olschanowsky (Boise State University)

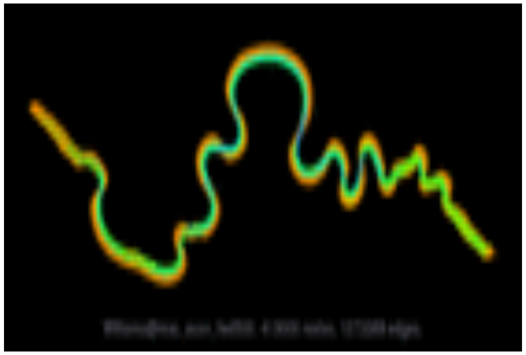
THE UNIVERSITY OF ARIZONA

Sparse Computations

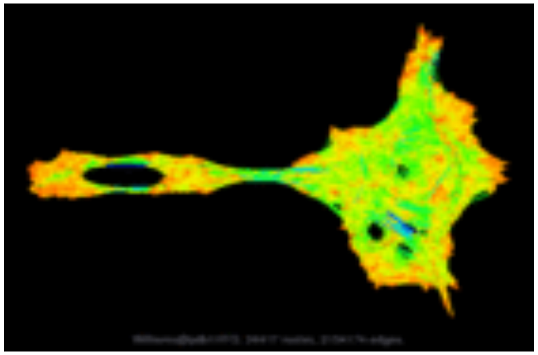
- Sparse matrices appear frequently in large linear systems of equations
- Sparse matrices have ***diverse*** applications (examples shown below)



Network Theory
(Web connectivity)



Epidemiology
(2D Markov model of epidemic)



Biology
(Protein structure)

Importance of Sparse Computations

- Other Application Areas
  - Partial differential equation solvers (PDEs)
  - Finite element analysis
  - Molecular dynamics simulations
  - Big graph analytics

- Example algorithms
  - Sparse matrix vector multiply (SpMV)
  - Sparse triangular solve
  - Matrix powers kernel
  - QR decomposition
  - Cholesky
  - LU-decomposition

Challenges Remain

- **Problem:** Sparse matrix codes optimized per
  - Sparse matrix format (specialized per algorithm)
  - Architecture
  - Parallelization strategy

- **Goal:** Compiler-optimized sparse codes

- **Today:** Sparse Polyhedral Framework for
  - Optimizing sparse computations with compiler
  - Composing inspector-executor transformations

Background: Sparse Matrix Formats

- Compressed Sparse Row (CSR) Representation
  - Default sparse storage format in many sparse matrix applications
  - Column dimension explicitly represented using an ***index array***
  - Row dimension implicit

$$A = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & c & d & 0 \\ e & 0 & f & g \end{bmatrix}$$

```
rowptr: [ 0   1   2       4         7 ]

A:      [ a   b   c   d   e   f   g ]
col:    [ 0   1   1   2   0   2   3 ]

diag:   [ 0   1   3       6 ]
```

Background: Dense Triangular Solver

$$A\vec{u} = \vec{f}$$

- Lower-Triangular, Forward Solve
- Rows cannot be processed in parallel
- `u[0]` has to be computed before `u[1]`
- `u[0]` and `u[1]` have to be computed before `u[2]`…
- Outer `i` loop cannot be parallelized

Dense Matrix

a	0	0	0
b	c	0	0
d	e	f	0
h	i	j	k

```
for (i = 0; i < n; i++){
  u[i] = f[i];
  for (j = 0; j < i; j++) {
    u[i] -= A[i][j]*u[j];
  }
  u[i] /= A[i][i];
}
```

Background: Sparse Triangular Solver

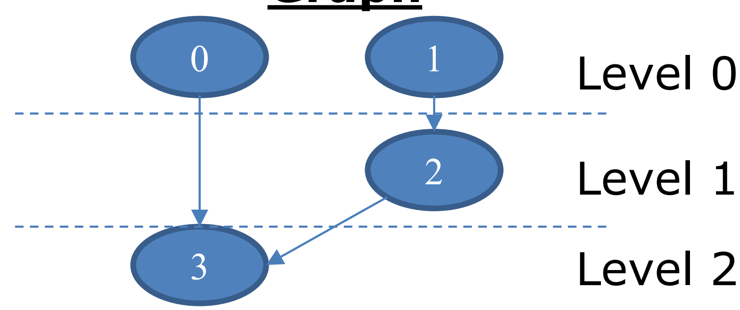
Sparse Matrix

a	0	0	0
0	b	0	0
0	c	d	0
e	0	f	g

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

- Sparse Lower-Triangular, Forward Solve
- Some rows can be processed in parallel
  - Eg. Rows 0 and 1

Dependence/Task Graph

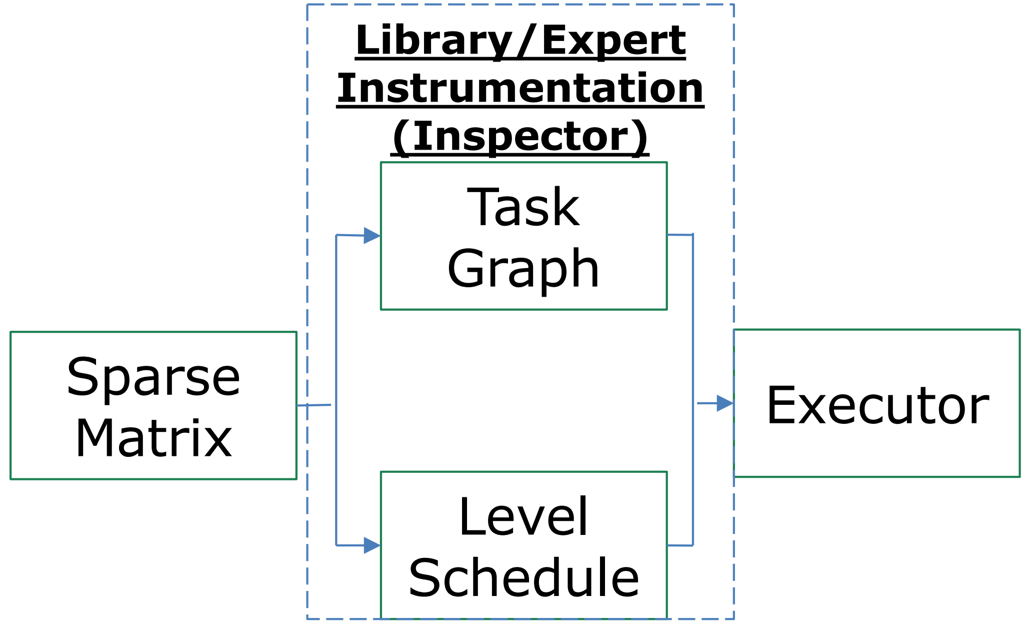


- Parallel wavefront scheduled computation (`i` loop partially parallel)

Exploiting Parallelism in Sparse Triangular Solve

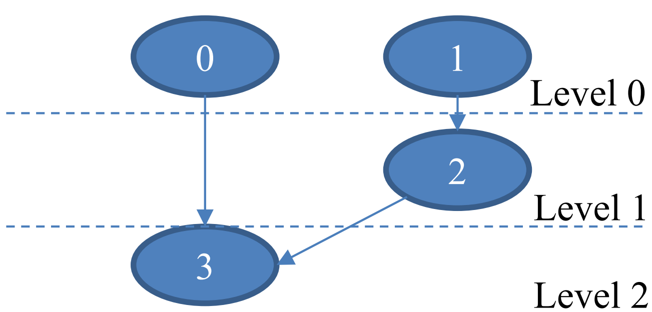
- Currently level-by-level parallelization done via libraries or expert programmers

- At ***runtime*** construct task graph for use by parallel code




```
// Executor
for(v=0;v<num_levels;v++)
  par-for (i=level[v];
             i<level[v+1];i++) {
    row = p[i];
    u[row] = f[row];
    for (j=index[row];
          j<diag[row]; i++){
      u[row] -= A[j]*u[col[j]];
      u[row] /= A[diag[row]];
}}}
```

Task Graph



Challenges for the Compiler

- Array accesses and loop bounds are non-affine
- Impossible for compiler to determine dependences precisely
- `i` loop marked as sequential

Original Sparse Triangular Solve Code

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

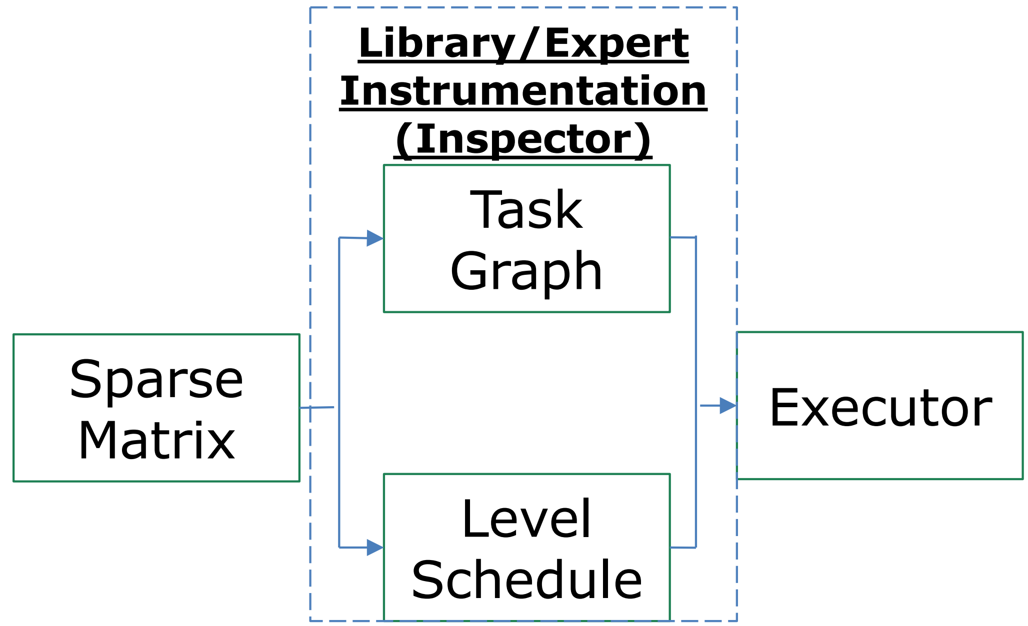
Another kind of code and transformations

- **What:** Inspector-Executor Transformations
- **Background:** Loop Transformation Frameworks
- **How:** Sparse Polyhedral Framework (SPF)
- Optimizations using SPF
  - Array access simplification examples
  - Inspector fusion examples

(Recall) Exploiting Parallelism in Sparse Triangular Solve

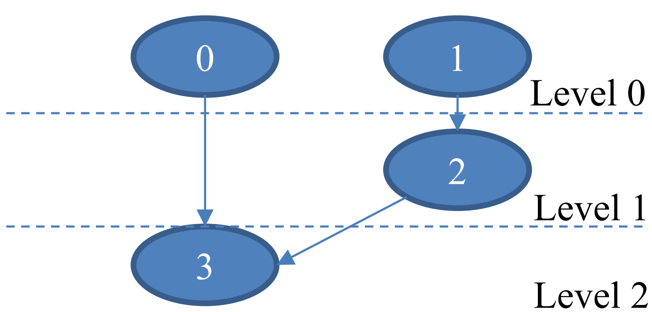
- Currently level-by-level parallelization done via libraries or expert programmers

- At ***runtime*** construct task graph and for use by parallel code




```
// Executor
for(v=0;v<num_levels;v++)
  par-for (i=level[v];
           i<level[v+1];i++) {
    row = p[i];
    u[row] = f[row];
    for (j=index[row];
         j<diag[row]; i++){
      u[row] -= A[j]*u[col[j]];
      u[row] /= A[diag[row]];
}}}
```

Task Graph



Inspector-Executor Tranformation

Traverses index array

Generates data reordering function σ

Reorder data and updates index array

Inspector

```
for i=0,7
  ... r[i] ...
for j=0,7
  sigma[j] = ...

for j=0,7
  Z'[sigma[j]]=Z[j]
  r'[j]=sigma[r[j]]
```

Original Code

```
for i=0,7
  Y[i] = Z[r[i]]
```

Executor

```
for i=0,7
  Y[i] = Z'[r'[i]]
```

Inspector-Executor Related Work

- Wavefront parallelism [Mirchandaney 88] [Rauchwerger 98] [Zhuang 09] [Park 14]
- Distributed memory parallelism [Saltz 91] [Basumallik 09] [Ravishankar 12]
- Data and iteration reordering of parallel and reduction loops for improved data locality [Ding 99] [Mitchell 99] [Mellor-Crummey 01] [Han 06]
- Sparse tiling for aggregating across loops [Douglas 00] (Strout 01) [Mohiyuddin 09]

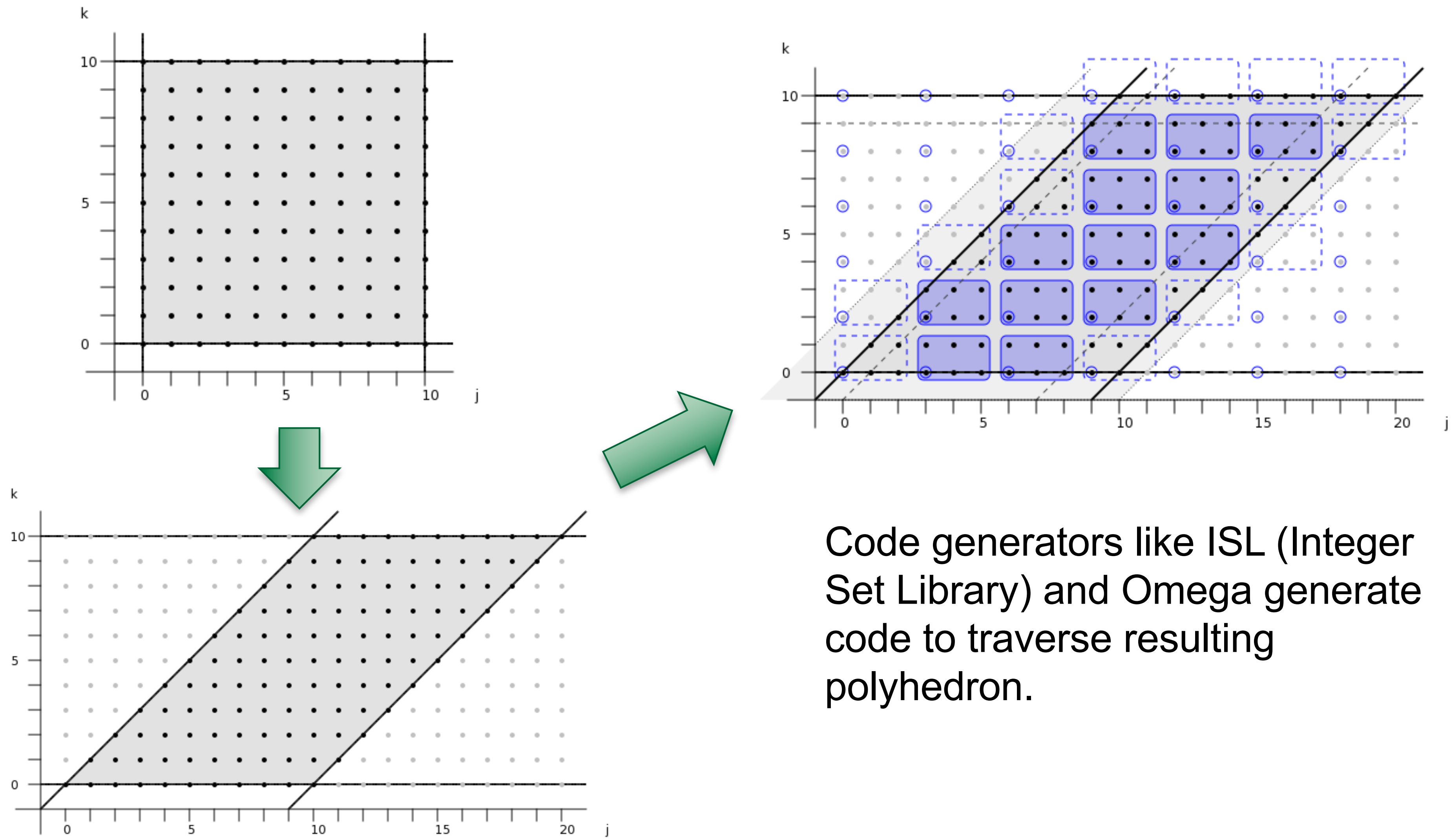
Problem: Inspector-Executor (I/E) Transformations Not Mainstream Yet

- ***Loop transformation frameworks*** being used in general purpose compilers (GCC, LLVM, ...)
- Enables the ***composition*** and autotuning of loop transformations (loop fusion, tiling, ...)
- ***Only*** individual and hard-coded compositions of I/E in research compilers
- Composition with typical loop transformations
  - CHiLL scriptable compiler at University of Utah
  - Other work [Ravishankar PPoPP 2015]

Solution: Sparse Polyhedral Framework (SPF)

- Loop transformation framework built on the polyhedral model
- Uses ***uninterpreted functions*** to represent index arrays
- Enables the ***composition of inspector-executor transformations***
- Exposes opportunities for compiler to
  - ***Simplify*** indirect array accesses and
  - ***Optimize*** inspector-executor code

Polyhedral Model for Specifying Loop Transformations



Code generators like ISL (Integer Set Library) and Omega generate code to traverse resulting polyhedron.

Picture done with tiling visualizer by Jon Roelofs

Polyhedral Loop Transformation Framework

```
// Loop Nest
for (i=0; i<N; i++){
  for (j=0; j<N; j++){
    D[i+j][j] = ...
  }
}
```

- Loops represented as sets of integer tuples

$$\{[i,j] \mid 0 \leq i < N \wedge 0 \leq j < N\}$$

- Array accesses with functions

$$\{[i,j] \rightarrow [v,w] \mid v = i + j \wedge w = j\}$$

- Transformations with functions

Transformation Framework: Loop Fusion Example

```
// Original Loop Nests
for (j=1; j<N; j++) {
  B[j] = ... ;
}
for (k=0; k<N-1; k++) {
  C[k] = ... B[k+1] ... ;
}
```

Iteration space

$$I = \{[0,j,0] \mid 1 \le j < N\} \cup \{[1,k,0] \mid 0 \le k < N-1\}$$

Loop fusion transformation

$$T = \{[0,j,0] \rightarrow [0,j',0] \mid j' = j\} \cup \{[1,k,0] \rightarrow [0,j',1] \mid j' = k+1\}$$

Old Iterators as Function of New Iterators

$$j = j'$$
$$k = j' - 1$$

```
// After loop fusion
for (j'=1; j'<N; j'++) {
  B[j'] = ... ;
  C[j'-1] = ... B[j'] ... ;
}
```

Rewrite array index expressions

Sparse Polyhedral Framework (SPF)

- Loop transformation framework built on the polyhedral model
- Uses ***uninterpreted functions*** to represent index arrays
- Enables the ***composition of inspector-executor transformations***
- Exposes opportunities for compiler to
  - ***Simplify*** indirect array accesses and
  - ***Optimize*** inspector-executor code

SPF: Uninterpreted Functions Represent Index Arrays

```
// SpMV for CSR
for (i=0; i<n; i++){
  for (k=rowptr[i]; k<rowptr[i+1]; k++){
    y[i] += a[k]*x[col[k]];
  }
}
```

Iteration space

$$I = \{[i,k] | 0 \leq i < n \wedge rowptr(i) \leq k < rowptr(i+1)\}$$

SPF: Representing *Inspector-Executor Transformations* with Uninterpreted Functions

```
// SpMV for CSR
// (Compressed Sparse Row)
for (i=0; i<n; i++){
  for (k=rowptr[i];
       k<rowptr[i+1]; k++){
     y[i] += a[k]*x[col[k]];
}}
```

Coalesce Transformation

$$
\begin{aligned}
T &= \{[i,k] \rightarrow [k'] \mid k' = c(i,k) \\
&\quad \wedge 0 \leq k' < NNZ\} \\
NNZ &= count(I) \\
c &= order(I)
\end{aligned}
$$

```
// Inspector code
NNZ   = count( rowptr )
c     = order( rowptr )
c_inv = inverse( c )

// Executor code
// SpMV for COO (Coordinate Storage)
for (k'=0; k'<NNZ; k'++) {
  y[c_inv[k'][0]] +=
     a[c_inv[k'][1]]*x[col[c_inv[k'][1]]];
}
```

Old Iterators as Function of New Iterator

$$
\begin{aligned}
i &= c^{-1}(k')[0] \\
k &= c^{-1}(k')[1]
\end{aligned}
$$

SPF: Inspector Dependence Graphs

```
// SpMV for CSR
// (Compressed Sparse Row)
for (i=0; i<n; i++){
  for (k=rowptr[i];
       k<rowptr[i+1]; k++){
     y[i] += a[k]*x[col[k]];
}}
```

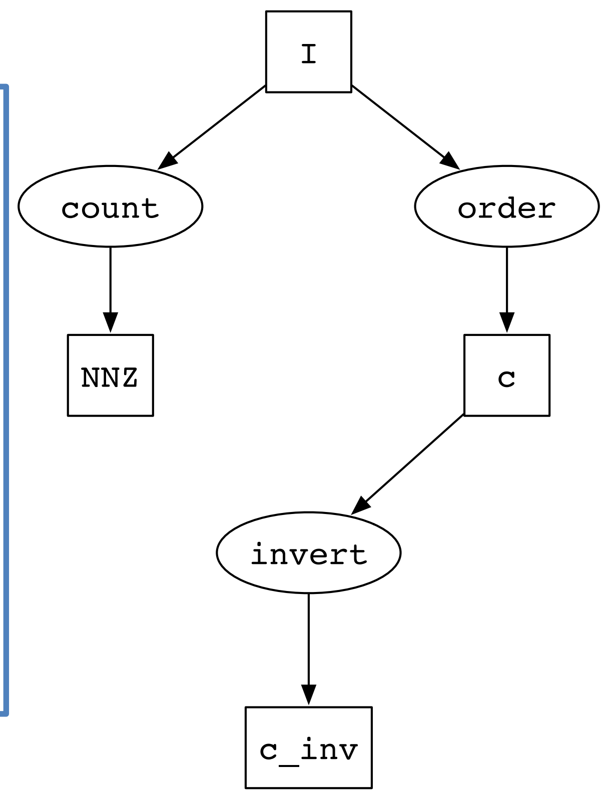
```
// Inspector code
NNZ   = count( rowptr )
c     = order( rowptr )
c_inv = inverse( c )

// Executor code
// SpMV for COO (Coordinate Storage)
for (k'=0; k'<NZZ; k'++) {
  y[c_inv[k'][0]] +=
    a[c_inv[k'][1]]*x[col[c_inv[k'][1]]];
}
```

Coalesce Transformation

$$\begin{aligned} T &= \{[i,k] \rightarrow [k'] \mid k' = c(i,k) \\ &\quad \wedge 0 \leq k' < NNZ\} \\ NNZ &= count(I) \\ c &= order(I) \end{aligned}$$

Inspector Dependence Graph (IDG)



Sparse Polyhedral Framework (SPF)

- Loop transformation framework built on the polyhedral model
- Uses ***uninterpreted functions*** to represent index arrays
- Enables the ***composition of inspector-executor transformations***
- Exposes opportunities for compiler to
  - ***Simplify*** indirect array accesses and
  - ***Optimize*** inspector-executor code

Composing Transformations in SPF

$$\vec{y} = A\vec{x}$$

```
// SpMV for Coordinate Storage
for (k=0; k<NNZ; k++) {
  y[row[k]] += val[k]*x[col[k]];
}
```

Data reordering transformation

$$R_{Y \to Y'} = \{[i] \to [i'] | i' = \sigma(i)\}$$
$$\sigma = dataReordHeuristic(A_{K \to Y})$$

```
// Inspector
sigma     = dataReordHeuristic(row);
y'        = reorderArray( y, sigma );

// Executor
for (k=0; k<NNZ; k++) {
  y'[sigma[row[k]]] += val[k]*x[col[k]];
}
```

Computation reordering transformation

$$T_{K \to K'} = \{[k] \to [k'] | k' = \delta(k)\}$$
$$\delta = iterReordHeuristic(A_{K \to Y'})$$

```
// Inspector
sigma     = dataReordHeuristic(row);
y'        = reorderArray( y, sigma );
delta     = iterReordHeuristic( sigma, row );
delta_inv = invert(delta);

// Executor
for (k'=0; k'<NNZ; k'++) {
  y'[sigma[row[delta_inv[k']]]]
    += val[delta_inv[k']]*x[col[delta_inv[k']]];
}
```

Data space index domains

$$Y = \{[i] \quad | \quad 0 \le i < N\}$$
$$V = \{[i] \quad | \quad 0 \le i < NNZ\}$$
$$X = \{[i] \quad | \quad 0 \le i < N\}$$

Iteration space

$$K = \{[k] \quad | \quad 0 \le k < NNZ\}$$

Data access functions

$$A_{K \to Y} = \{[k] \to [i] \quad | \quad i = row(k)\}$$
$$A_{K \to V} = \{[k] \to [i] \quad | \quad i = k\}$$
$$A_{K \to X} = \{[k] \to [i] \quad | \quad i = col(k)\}$$

Modified data space and access function

$$Y' = \{[i'] | 0 \le i' < N\}$$
$$A_{K \to Y'} = \{[k] \to [i'] | i' = \sigma(row(k))\}$$

Modified iteration space and access functions

$$K' = \{[k'] | 0 \le k' < NNZ\}$$
$$A_{K' \to Y'} = \{[k'] \to [i'] | i' = \sigma(row(\delta^{-1}(k')))\}$$
$$A_{K' \to V} = \{[k'] \to [i] | i = \delta^{-1}(k')\}$$
$$A_{K' \to X} = \{[k'] \to [i] | i = col(\delta^{-1}(k'))\}$$

Another SPF Example
MOLDYN: molecular dynamics benchmark

```
for s=1,T
  for i=1,n
    ... = ...Z[i]
  endfor

  for j=1,m
    Z[l[j]] = ...
    Z[r[j]] = ...
  endfor

  for k=1,n
    ... += Z[k]
  endfor
endfor
```

Access Relation for i loop

$$A_{I_0 \to Z_0} = \{[i] \to [i]\}$$

Access Relation for j loop

$$A_{J_0 \to Z_0} = \{[j] \to [i] | i = l(j) \vee i = r(j)\}$$

Data Dependences
between i and j loop

$$D_{I_0 \to J_0} = \{[i] \to [j] \mid (i = l(j)) \vee (i = r(j))\}$$

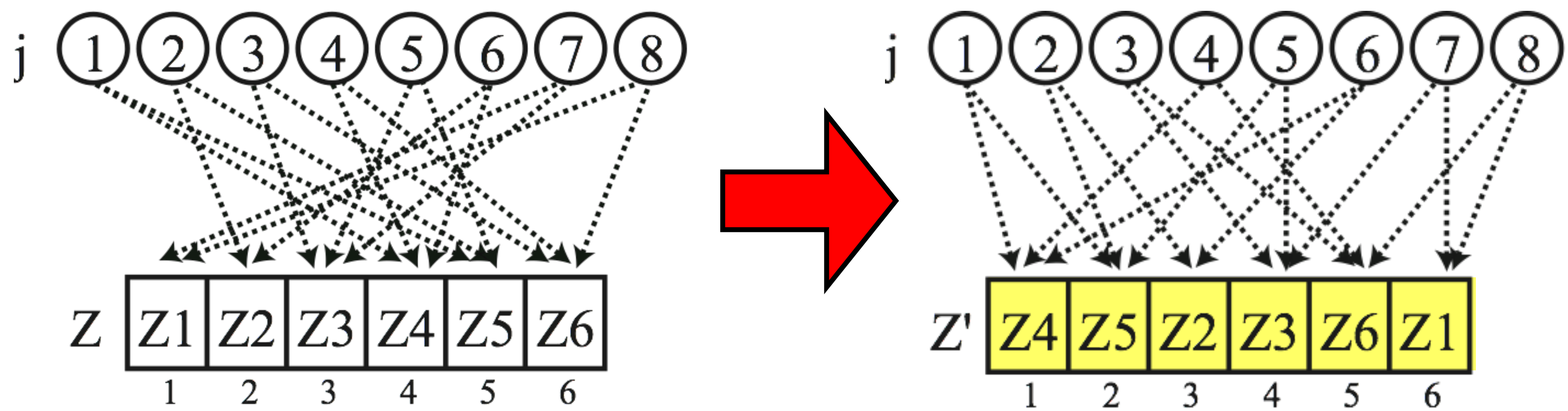
Data Permutation Reordering (Equations are the compile-time abstraction)

$$R_{Z_0 \to Z_1} = T_{I_0 \to I_1} = \{[i] \to [\sigma(i)]\}$$

CPACK reordering heuristic [Ding & Kennedy 99]

$$A_{J_0 \to Z_0} = \{[j] \to [i] | i = l(j) \vee i = r(j)\}$$

$$A_{J_0 \to Z_1} = \{[j] \to [i] \quad | \quad i = \sigma(l(j)) \vee i = \sigma(r(j))\}$$

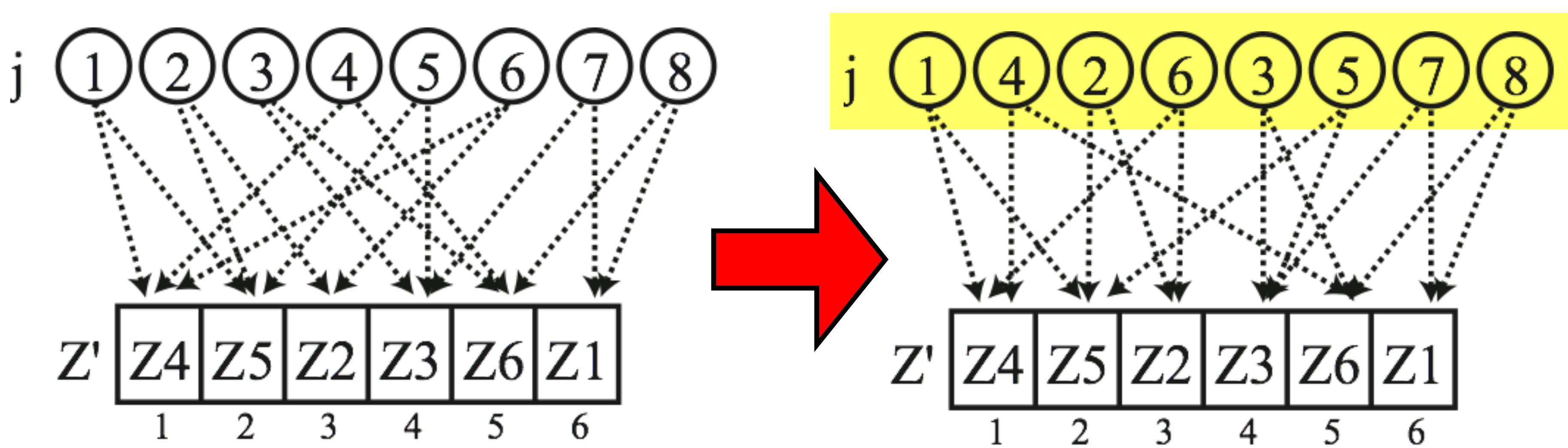


Iteration Permutation Reordering

$$T_{J_0 \to J_1} = \{[j] \to [x] \quad | \quad x = \delta(j)\}$$

$$A_{J_0 \to Z_1} = \{[j] \to [i] \quad | \quad i = \sigma(l(j)) \vee i = \sigma(r(j))\}$$

$$A_{J_1 \to Z_1} = \{[j] \to [i] \quad | \quad i = \sigma(l(\delta^{-1}(j))) \vee i = \sigma(r(\delta^{-1}(j)))\}$$



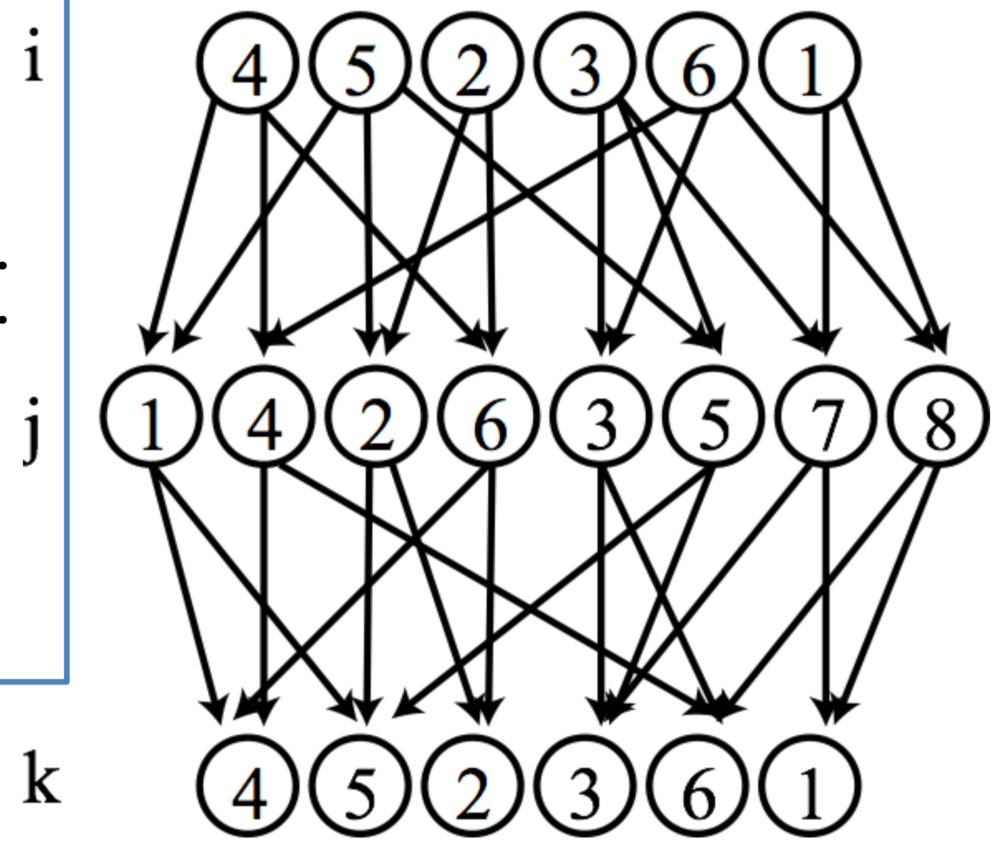
Dependences Between Loops after other transformations

$$D_{I_1 \to J_1} = \{[0,i] \to [1,j] \quad | \quad i = \sigma(l(\delta^{-1}(j))) \\ \vee i = \sigma(r(\delta^{-1}(j)))\}$$

```
for s=1,T
  for i=1,n
    ... = ...Z'[i]
  endfor

  for j'=1,m
    Z'[sigma[l[delta_inv[j]]]] = ...
    Z'[sigma[r[delta_inv[j]]]] = ...
  endfor

  for k=1,n
    ... += Z[k]
  endfor
endfor
```

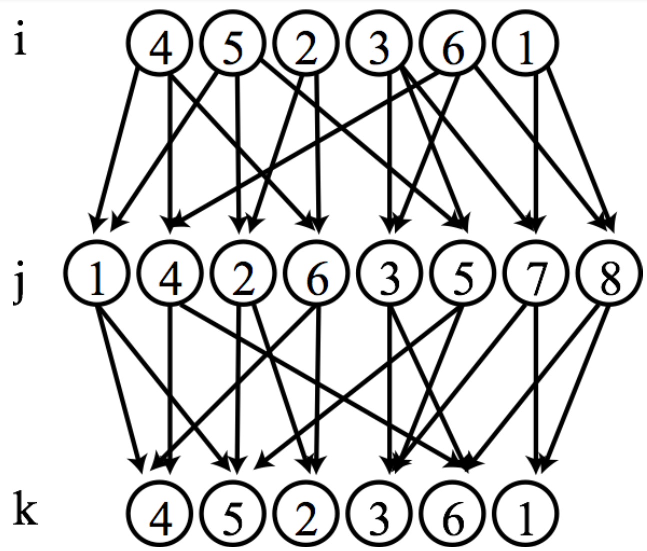


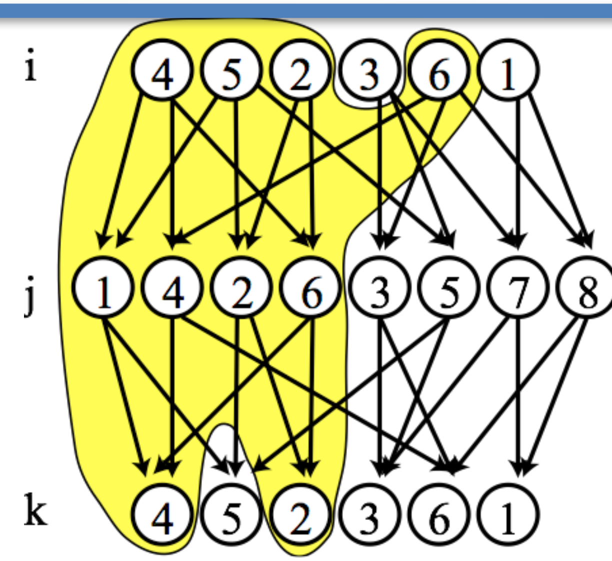
Full Sparse Tiling (FST)

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

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

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





Key Insights in SPF

- The inspectors ***traverse*** the data mappings and/or the data dependences
- We can ***express*** how the data mappings and data dependences will be reordered
- Subsequent inspectors ***traverse the new*** data mappings and data dependences
- Use polyhedral code generator (ISL) for outer loops and deal with sparsity in inner loops and access relations

Sparse Polyhedral Framework (SPF)

- Loop transformation framework built on the polyhedral model
- Uses ***uninterpreted functions*** to represent index arrays
- Enables the ***composition of inspector-executor transformations***
- Exposes opportunities for compiler to
  - ***Simplify*** indirect array accesses and
  - ***Optimize*** inspector-executor code

Simplifications at Compile Time

- Simplifying array accesses
  - Know that `delta_inv[]` is the inverse of `delta[]`
  - `delta[delta_inv[i]]` becomes `i`

- Reducing inspector loop depth
  - Relevant to inspectors that iterate over dependences
  - Dependence inspector in worst case can be 2X the depth of original loop
  - Use monotonicity information and relationships such as `rowptr(i) <= diagptr(i)`
  - Find iterator as function of another iterator `(i=col[j])`
  - Can remove one level of nesting in inspector

(Recall) SPF: Inspector Dependence Graphs

```
// SpMV for CSR
// (Compressed Sparse Row)
for (i=0; i<n; i++){
  for (k=rowptr[i];
       k<rowptr[i+1]; k++){
     y[i] += a[k]*x[col[k]];
}}
```

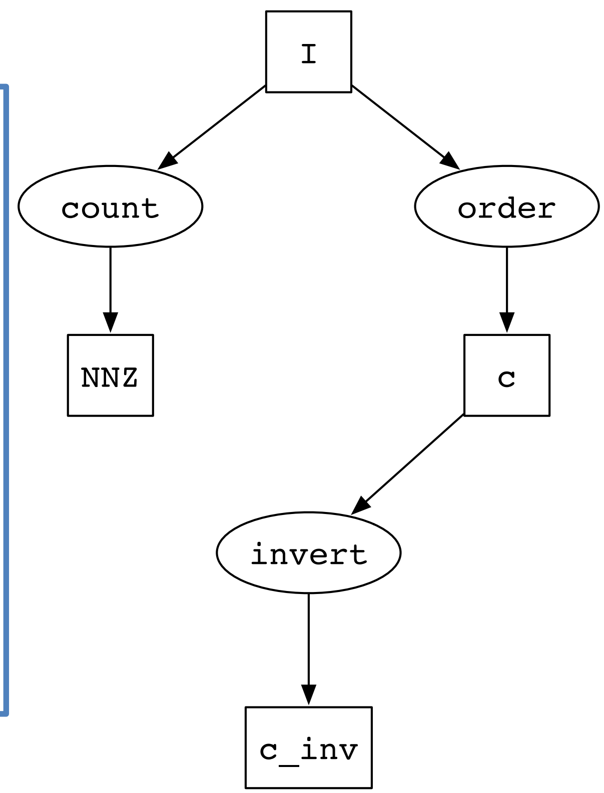
```
// Inspector code
NNZ   = count( rowptr )
c     = order( rowptr )
c_inv = inverse( c )

// Executor code
// SpMV for COO (Coordinate Storage)
for (k'=0; k'<NNZ; k'++) {
  y[c_inv[k'][0]] +=
    a[c_inv[k'][1]]*x[col[c_inv[k'][1]]];
}
```

Coalesce Transformation

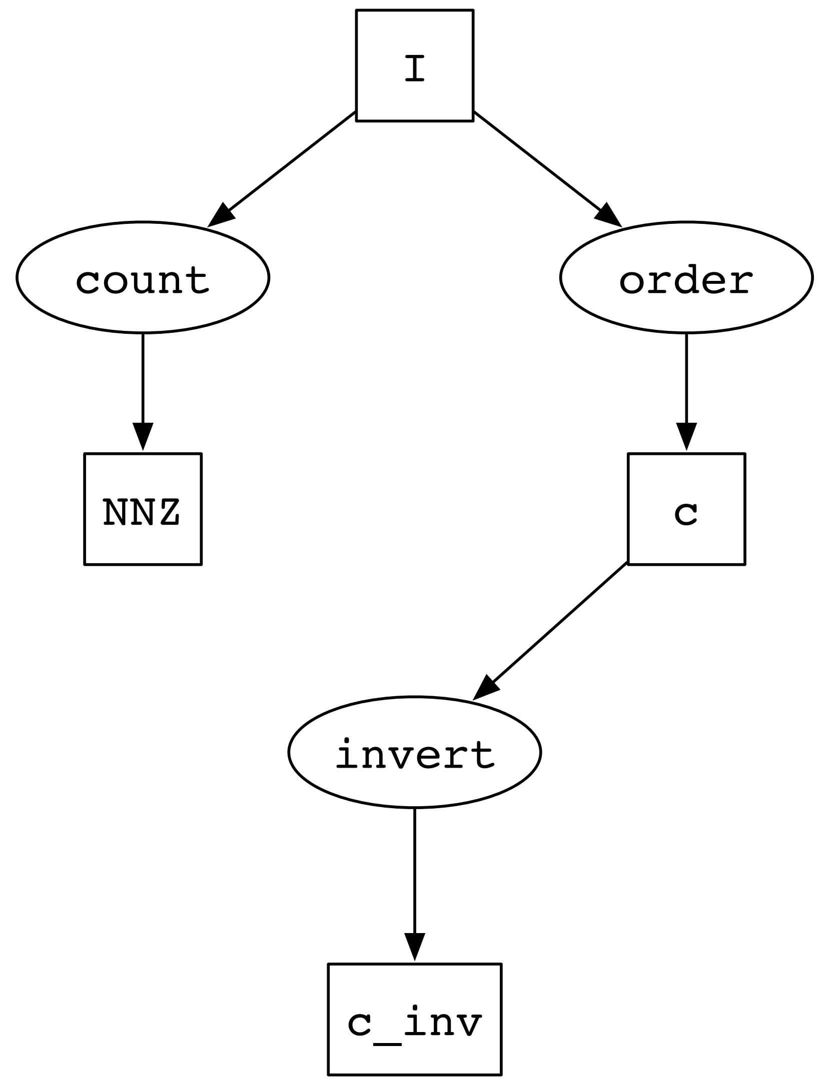
$$\begin{aligned} T &= \{[i,k] \rightarrow [k'] \mid k' = c(i,k) \\ &\quad \wedge 0 \leq k' < NNZ\} \\ NNZ &= count(I) \\ c &= order(I) \end{aligned}$$

Inspector Dependence Graph (IDG)

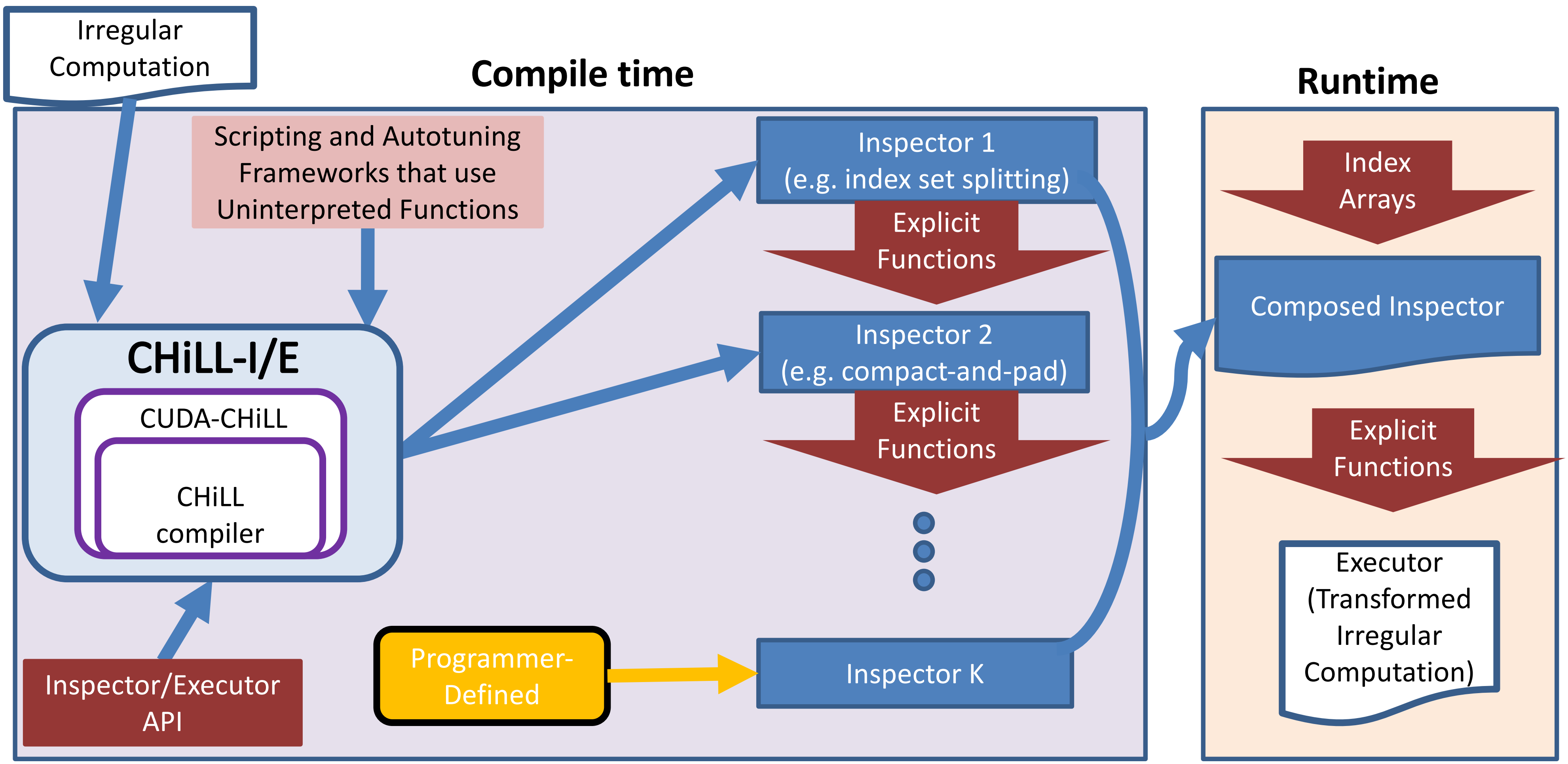


Fusing Inspectors

- Count and order iterate over same iteration set
- Executor only uses `c_inv`
- ***Can fuse count, order, and invert*** into a single loop that produces `NNZ` and `c_inv`



Goal: Inspector-Executor Transformations Composed by Compiler



Example: Sparse Triangular Solve

Irregular Computation

```
for (i=0; i<N; i++) {
  for (j=idx[i]; j<idx[i+1]; j++){
    x[i] = x[i] - A[j]*x[col[j]]; }}
```

Compile time

Scripting

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

Dependencies and Scheduling using Uninterpreted Functions

```
Deps = {[i]->[i']: i<i' and
       i=col(j) and idx(i)<=j<idx(i+1) }

Sched = {[i,j]->[l,i,j]: i in
                          level_set(l) }
(l sequential, i parallel, j sequential)
```

CHiLL-I/E compiler

Runtime

Index Arrays

Inspector X

```
CHiLLIE::func part-par(EF col, EF idx){
  CHiLLIE::func level_set;
  // BFS traversal of Deps doing gets.
  ... idx(i) ... col(j) ...
  // Place appropriate i's in each set.
  ... level_set(l).insert(i) ...
  return level_set; }
```

Explicit Functions

Executor

```
for (l=0; l<M; l++){
#omp parallel for
 for (i in level_set(l)){
   for (j=index[i]; j<index[i+1]; j++){
    x[i] = x[i] - A[j]*x[col[j]]; }}}
```

PLDI 2019 Artifact

- Source code: https://github.com/CompOpt4Apps/Artifact-DataDepSimplify
- Docker container
  - `docker pull kingmahdi/pldi19_artifact`
  - `docker run -it --rm kingmahdi/pldi19_artifact`
  - `more slist.txt`
  - `...Artifact-DataDepSimplify# ./simplification slist.txt`
  - `more data/gs_csr.c`
  - `more data/gs_csr.json`
  - `more results/gs_csr.out`
  - `./codegen short_codeGenlist.txt`
  - `more performanceEval/src/fs_csc_inspector.hh`

Sparse Polyhedral Framework (SPF) Summary

- ***Inspector-Executor transformations*** enable the parallelization and optimization of sparse code
- ***Uninterpreted functions*** can represent the composition of such transformations
- Bijectivity, monotonicity, and other ***information*** about uninterpreted functions enables I/E simplification
- ***Inspector Dependence Graph (IDG)*** enables optimizations such as inspector fusion

Collaborators and Funding



- Mary Hall (University of Utah)
- Catherine Olschanowsky (Boise State Univ.)

- Anand Venkat (Intel, PhD in 2016)
- Mahdi Soltan Mohammadi (Univ. of Arizona)
- Jeanne Ferrante (UC, San Diego, emeritus)
- Larry Carter (UC, San Diego, emeritus)

The presented research was partially supported by a National Science Foundation a National Science Foundation grant CCF-1564074 (CHiLL-I/E).