Abstractions for Specifying Sparse Matrix Data Transformations

Payal Nandy Mary Hall	Eddie C. Davis Catherine Olschanowsky	Mahdi S Mohammadi, Wei He Michelle Strout
University of Utah	Boise State University	University of Arizona

Motivation

- The polyhedral model is suitable for ***affine***
  - loop bounds, array access expressions and transformations
- Polyhedral model unsuitable for sparse matrix & unstructured mesh computations (***non-affine)***
  - Array accesses of the form A[B[i]]
  - Loop bounds of the form index[i] ≤ j < index[i+1]
- Key Observation
  - ***Compiler generated code for run time inspector & executor***
  - ***Run time inspection***
    - can reveal mapping of iterations to array indices
    - Potentially change iteration or data space

Related Work

Inspector/Executor

Mirchandaney, Saltz et al., 1988
Rauchwerger, 1998
Basumallik and Eigenmann, 2006
Ravishankar et al., 2012

Polyhedral Support for Indirection

Pugh and Wonnacott, 1994

Frameworks for Sparse Computations

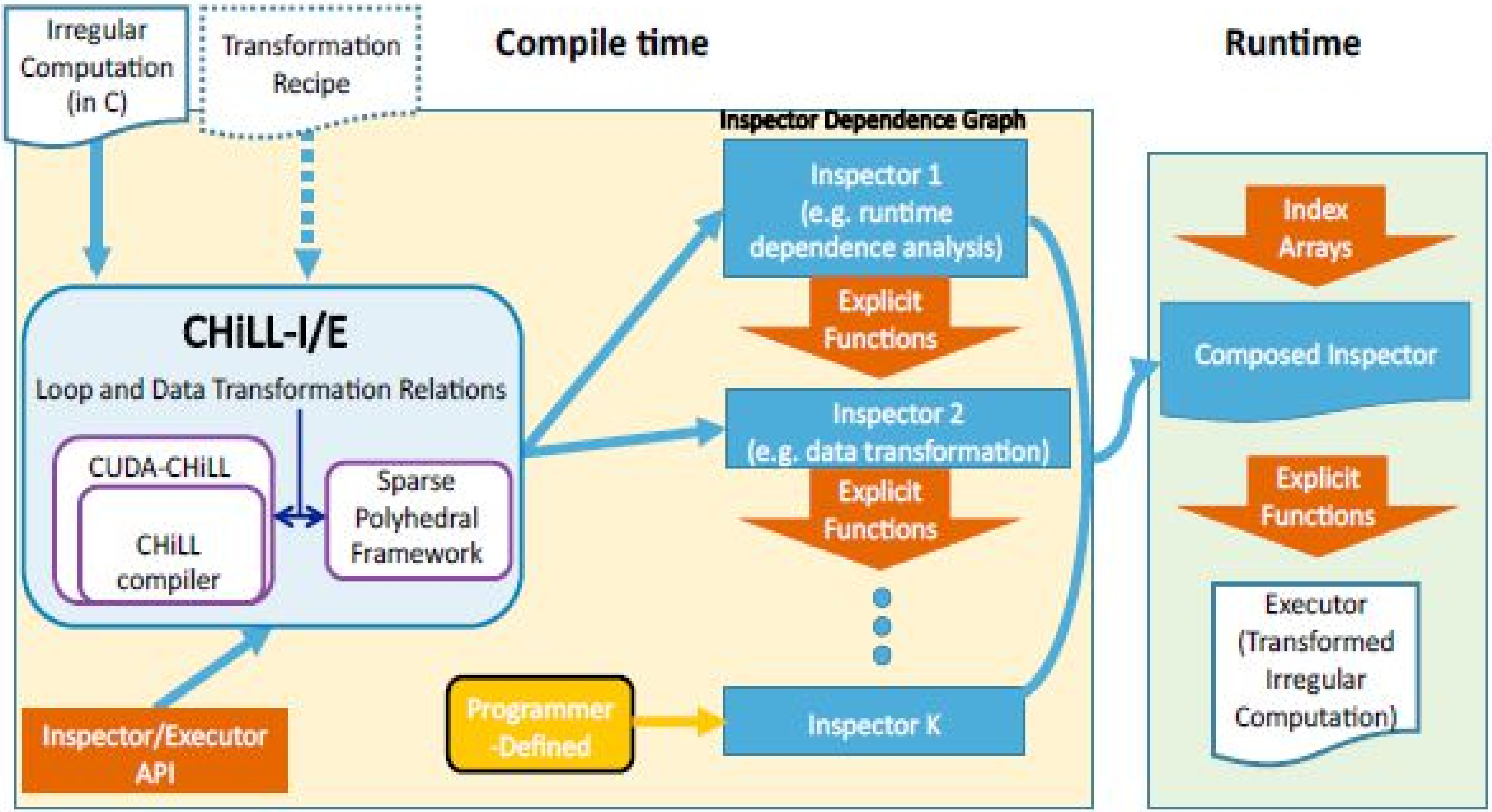
SIPR: Shpeisman, 1999
Bernoulli: Mateev, 2001

Data Transformations

Bik, 1996
Ding and Kennedy, 1999
Mellor-Crummey et al., 2001
Gilad et al., 2010
van derSpek, 2011

Prior work did not integrate all of these, and mostly did not expand data with zero-valued elements.

CHiLL-I/E - Vision



Foundation – Sparse Polyhedral Framework

- 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

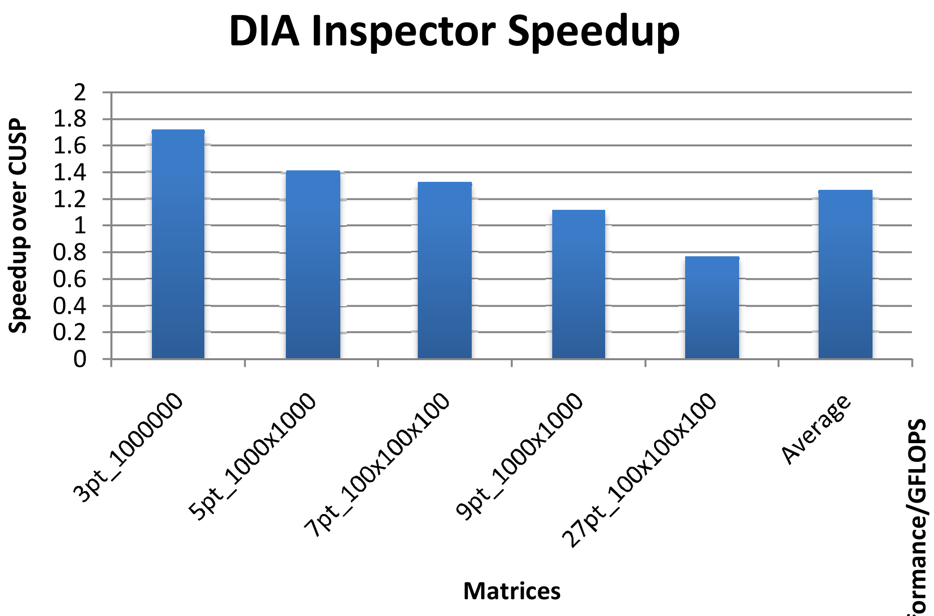
Foundation – CHiLL Compiler Framework

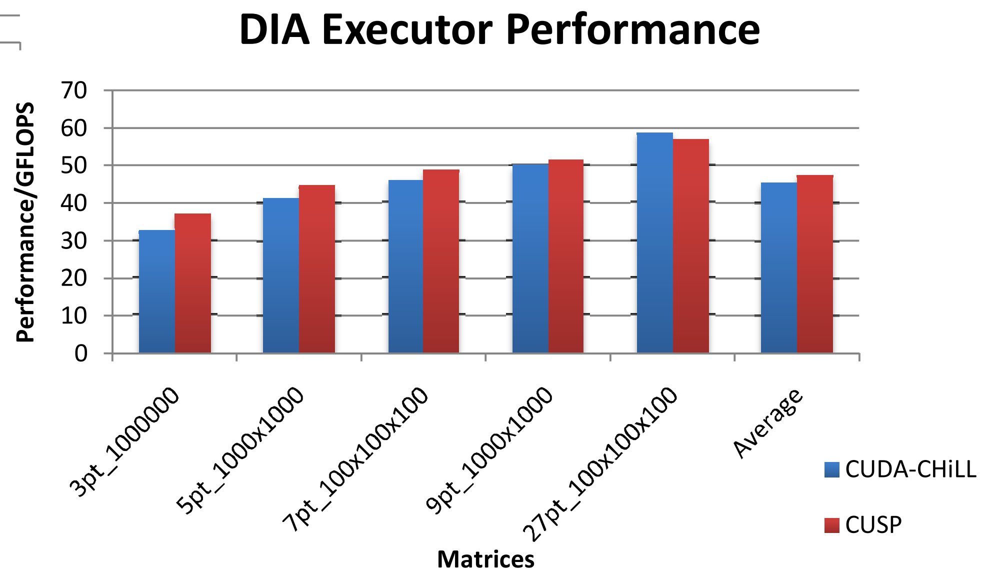
- Runtime data & iteration reordering transformations for non-affine loop bounds and array access
  - Make-dense
  - Compact, compact-and-pad
- Composable with polyhedral transformations
  - Tile, skew, permute
- Integration with user-specified Inspectors
- Automatically generated Inspector/Executors
  - Inspectors optimized for making less passes over data
  - Optimized executors performed comparable to runtime libraries

[CGO '14], [PLDI '15] [SC '16] [IPDPS '16] [LCPC '16]

Prior Research Performance Indicators

Performance of Compiler generated Inspectors and Executors competitive with CUSP





[PLDI'15]

Contribution

- Derive abstractions for Sparse Matrix ***Data Transformations***
  - Focus on transformations that modify data representation
- Extend Sparse Polyhedral Framework to Support data transformations
  - Modify data representation to reflect structure of input matrix
  - Expand iteration space to match new data representation
- Generalize representation of Inspector/executor transformations
  - Goal: automatically compose them

Abstractions

Transformation Relations

- Include uninterpreted functions
- Inc ludes non-affine transformations
- Composable with existing transformations

Inspector Dependence Graph

- Derived from Transformation relations
- Data flow representation of Inspector functionality

Automatic Generation of optimized Inspector/ Executor

- Compiler walks IDG to generate Inspector
- Inspector instantiates explicit functions for Executor

Sparse Matrix-Vector Multiply (SpMV)

Begin with Compressed Sparse Row (CSR) format

A: [1 5 7 2 3 6 4]
index: [0 2 4 6 7]
col: [0 1 0 1 2 3 3]

Compressed Sparse Row (CSR)

Non-affine loop bounds

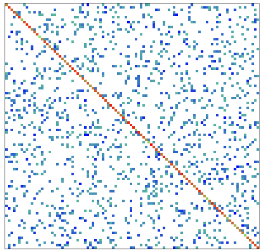
```
for (i=0; i < n; i++)
    for (j=index[i]; j<index[i+1]; j++)
        y[i]+=A[j]*x[col[j]];
```

Non-affine subscript

Sparse Matrix Formats

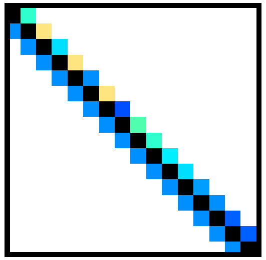
Iteration Space Transformation

Data & Iteration Space Transformation



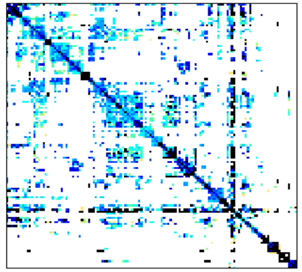
A: [1 5 7 2 3 6 4]
row: [0 0 1 1 2 2 3]
col: [0 1 0 1 2 3 3]

COO



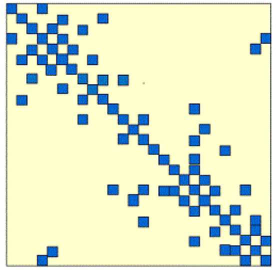
1	5	0	0
7	2	0	0
0	0	3	6
0	0	0	4

DIA



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

BCSR



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

ELL

Moldyn (molecular dynamics) – Data + Iteration Reordering

CSR to COO

Transformation Relations

$T_{coalesce}$ = {[i,j] → [k] | k = c(i,j) 0 ≤ k < NNZ

$I_{exec} = T_{coalesce}$ (I)

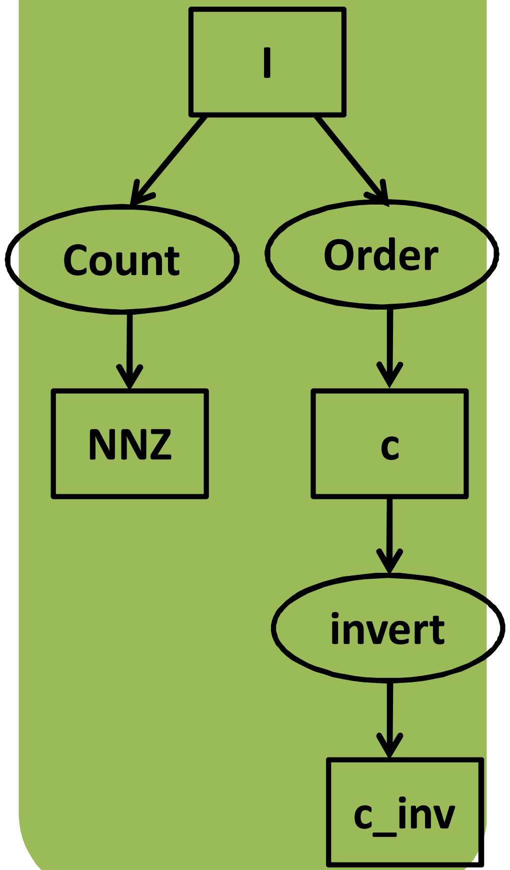
Generate Inspector

NNZ = count (I)

c = order (I)

c_inv = invert(c)

IDG



Inspector

```
struct access_relation c;
for (i=0; i<=n-1; i++)
   for (j=index[i]; j<=index[i+1]-1;
                                    j++)
         c.create_mapping(i,j);
```

Executor

```
for (k = 0; k < NNZ; k++)
    y[c_inv[k][0]]  += A[c_inv[k][1]]*
                      x[col[c_inv[k][1]]];
```

Enabling Data Transformations

make-dense

```
for (i=0; i < n; i++)
    for (j=index[i]; j<index[i+1]; j++)
        y[i]+=A[j]*x[col[j]];
```

```
for (i=0; i < n; i++)
  for(k=0; k <n; k++)
    for (j=index[i]; j<index[i+1]; j++)
      if(k== col[j])
          y[i]+=A[j]*x[k]
```

Guard Condition

CSR to DIA: Transformations

Dense Matrix

$$\begin{pmatrix} 1 & 5 & 0 & 0 \\ 7 & 2 & 0 & 0 \\ 0 & 0 & 3 & 6 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

CSR Format

$$A \begin{pmatrix} 1\ 5\ 7\ 2\ 3\ 6\ 4 \end{pmatrix}$$

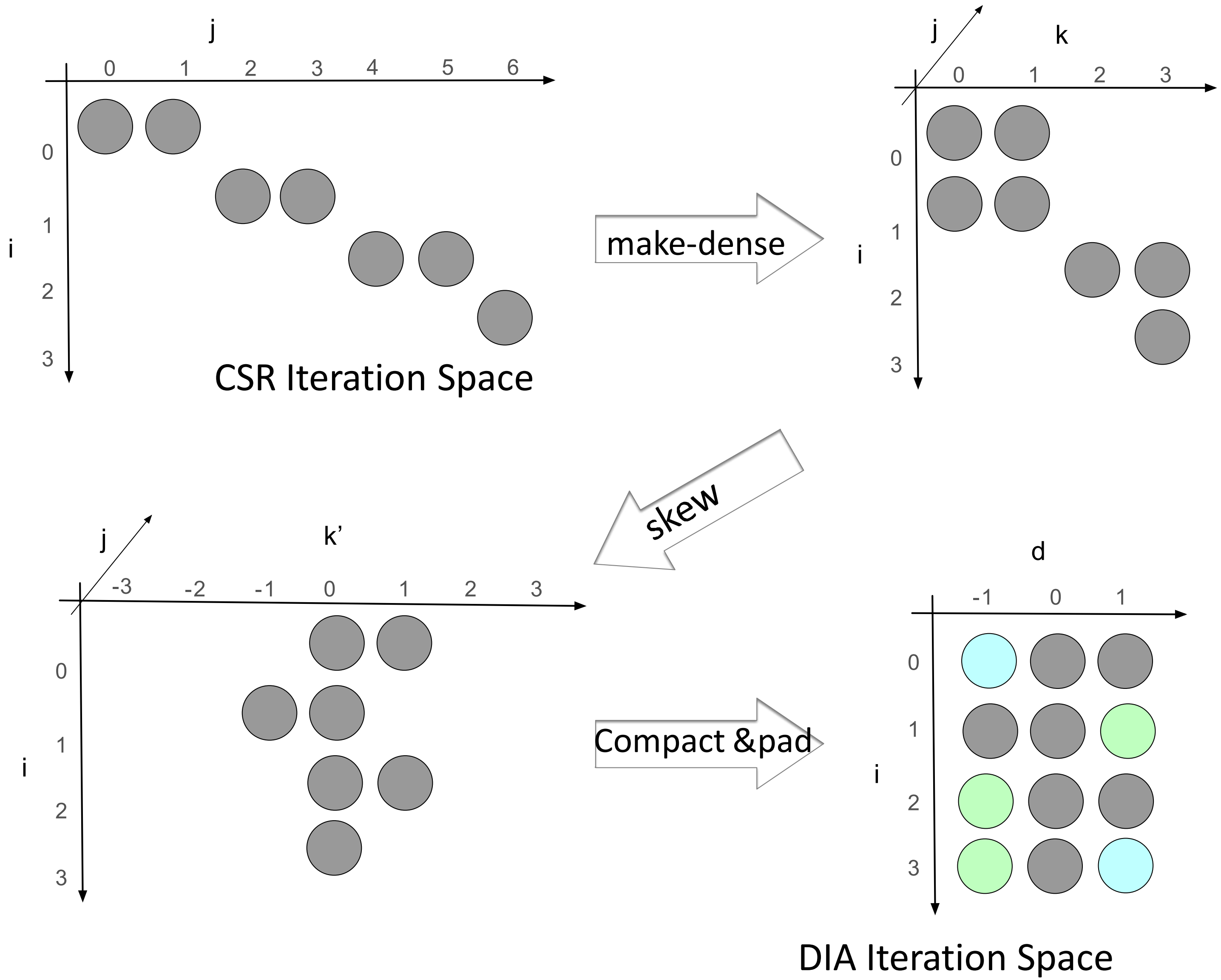
$$\text{index} \begin{pmatrix} 0\ 2\ 4\ 6\ 7 \end{pmatrix}$$

$$\text{col} \begin{pmatrix} 0\ 1\ 0\ 1\ 2\ 3\ 3 \end{pmatrix}$$

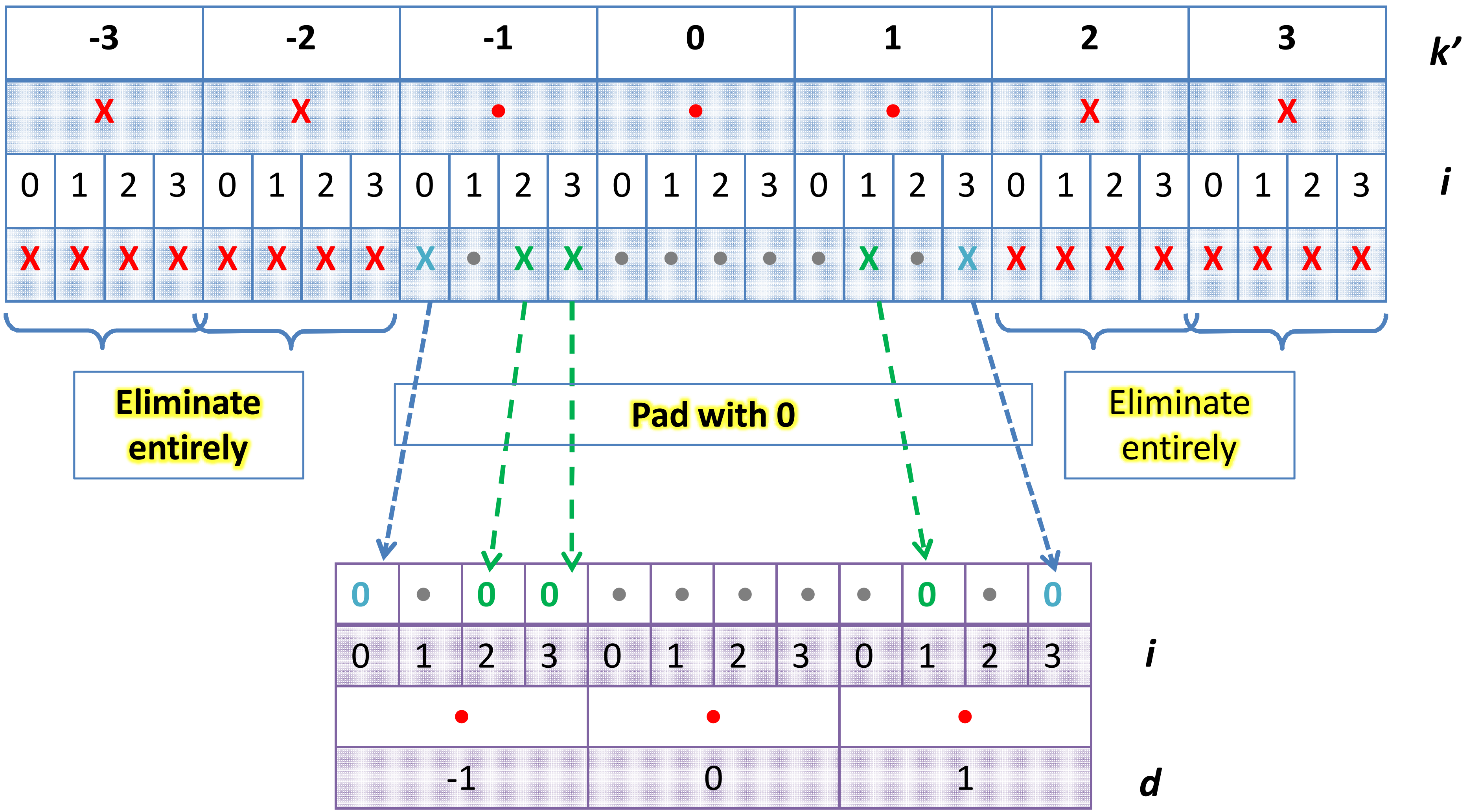
DIA Format

$$A' \begin{pmatrix} 0 & 1 & 5 \\ 7 & 2 & 0 \\ 0 & 3 & 6 \\ 0 & 4 & 0 \end{pmatrix}$$

$$\text{offsets} \begin{pmatrix} -1\ 0\ 1 \end{pmatrix}$$



Compact-and-pad



CSR to DIA

Transformation Relations

$T_{make\text{-}dense} = \{[i,j] \rightarrow [i,k,j] \mid 0 \leq k < N \wedge k = col(j)\}$

$T_{skew} = \{[i,k,j] \rightarrow [i, k',j] \mid k' = k-i\}$

$T_{compact\text{-}and\text{-}pad} = \{[k'.i,j] \rightarrow [i;d] \mid 0 \leq d < ND \wedge$
$k' = col(j) - i \wedge c(d) = k'$

$Iexec = T_{compact\text{-}and\text{-}pad}(T_{skew}(T_{make\text{-}dense}(I)))$

Generate Inspector

$D\_set = \{[k'] \mid \exists j, k' = col(j)-i \wedge$
$index(i) \leq j < index(i+1)\}\}$

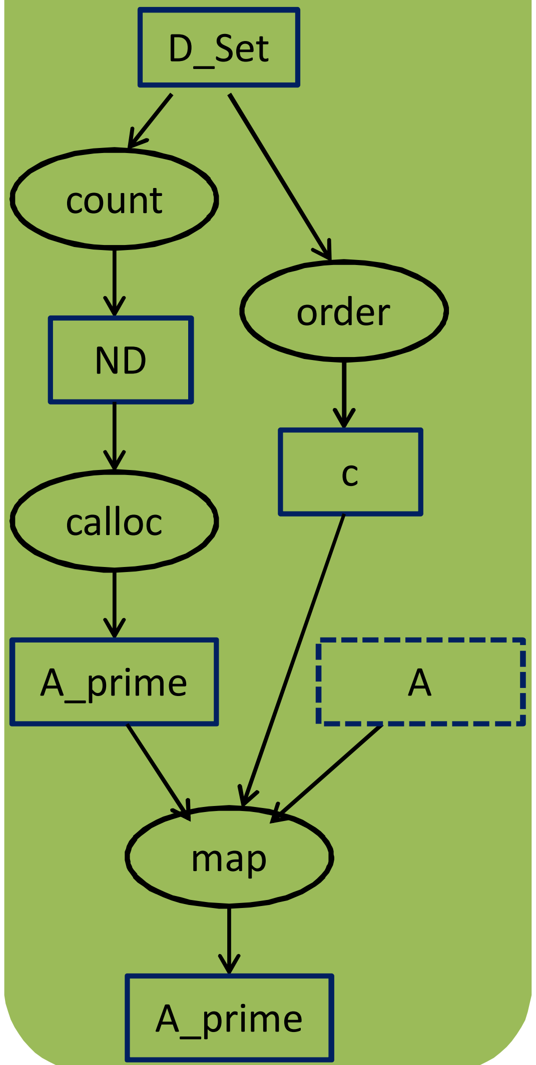
$ND = count(D\_set)$

$C = order(D\_set)$

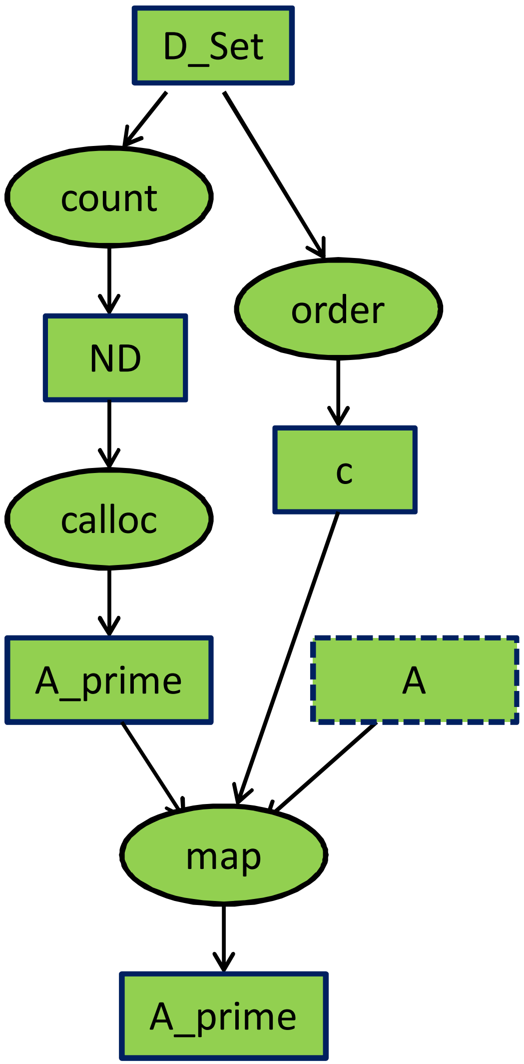
$A\_prime = calloc(N*ND*sizeof(datatype))$

*map: $R_{A \rightarrow A\_prime} = \{[j] \rightarrow [i,d] \mid 0 \leq d < ND \; \exists$
$k', k' = col(j)-i \wedge c(d)=k' \}$*

IDG



CSR to DIA



<u>Inspector Code for DIA</u>

```
ND = 0; D_set = emptyset;
 for(i = 0; i<N; i++)
  for(j = index[i]; j < index[i+1]; j++) {
    k_prime = col(j)-i;
    if (!marked[k_prime])
            D_set = D_set U <k_prime,ND++>;
 }
 A_prime = calloc(N*ND*sizeof(datatype));
 c = calloc(ND*sizeof(indextype));
 for(i = 0; i<N; i++)
  for(j = index[i]; j < index[i+1]; j++) {
    k_prime = col(j)-i;
    d = lookup(k_prime,D_set);
    c[d] = k_prime;
    A_prime[i][d] = A[j];
  }
```

Executor Code

```
for (i=0; i < N; i++)
   for(d=0; d<ND; d++)
      y[i] += A[i][d]*x[i+c[d]];
```

Future Work - Optimizing the IDG

- Minimize inspector passes over input data
- Extend IDG to support fusion of Inspectors
- Additional optimizations
  - Dynamic data structures (e.g. linked lists) to eliminate sweeps to calculate size of data representation
  - Integrate existing inspector library functions

Conclusion

- Abstractions for data transformations in sparse matrix & unstructured mesh computations
- Approach
  - Transformation Relations
  - Inspector Dependence Graph
  - Compiler generated optimized Inspector/Executor code
- Vision: Create a framework to compose complex transformation sequences for inspectors and executors