Abstractions for Specifying Sparse Matrix Data Transformations

Payal Nandy
Mary Hall
University of Utah

Eddie C Davis
Catherine Olschanowsky
Boise State University

Mahdi S Mohammadi, Wei He
Michelle Strout
University of Arizona

MOTIVATION

- The polyhedral model is suitable for ***affine transformations***
  - 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]

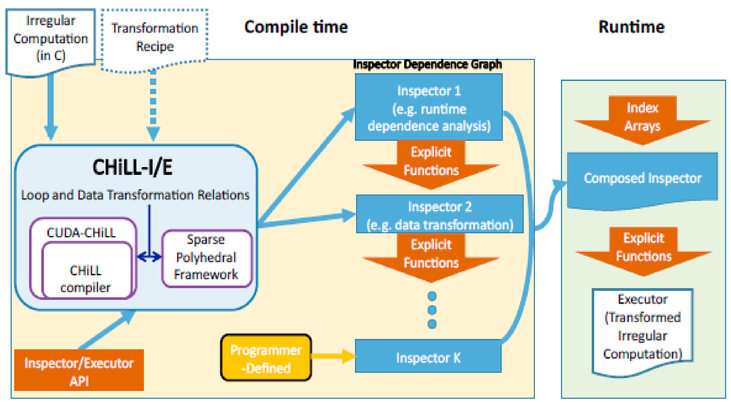
Key Observation

Compiler generated code for run time inspector & executor

Inspector
- Collects information that is only available at runtime
- Widely used in optimizing irregular computations, where information about data dependences, loop bounds, data structures, and memory access patterns
- Used to guide code transformation, parallelization, and data layout

Executor
- Optimized version of original computation created at compile time.
- Uses optimized data layout and iteration spaces generated by Inspector

CHiLL–I/E VISION



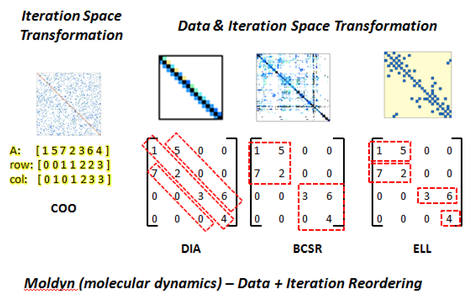
CONTRIBUTION

*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

Sparse Matrix Formats



Moldyn (molecular dynamics) – Data + Iteration Reordering

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)

Sparse Matrix Vector Multiply

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

APPROACH

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

CHiLL Compiler Framework

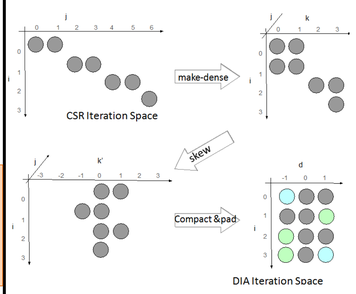
- Runtime data & iteration reordering transformations for non-affine loop bounds and array access
  - Make-dense
  - Compact, compact-and-pad
- Composable with polyhedral transformations
  - tile, skew, permute
- Integration with user-specified Inspectors
- Automatically generated Inspector/Executors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

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

EXAMPLE – CSR to DIA



Transformation Relations

$T_{make\text{-}dense}$ = {[i,j] →[i,k,j] | 0 ≤ k < N ^ k = col(j)

T_{skew} = {[i,k,j] →[i, k',j] | k' = k-i}

$T_{compact\text{-}and\text{-}pad}$ = {[k',i,j] →[i;d] | 0 ≤ d < ND ^ k' = col(j) - i ^ c(d) = k'

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

Generate Inspector

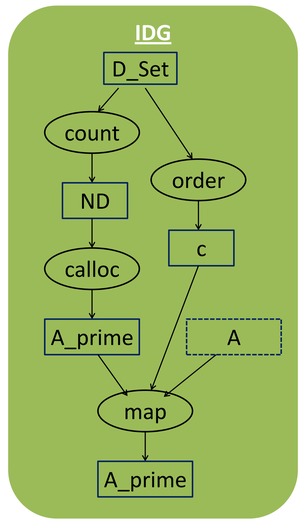
D_set = {[k'] | ∃ j, k' = col(j)-i ^ index(i) ≤ j < index(i+1)}

ND = count(D_set)

C = order(D_set)

*A_prime = calloc(N*ND*sizeof(datatype))*

map: $R_{A \to A\_prime}$ = {[j] → [i,d] | 0 ≤ d <ND ∃ k', k' = col(j)-i ^ c(d)=k' }



Executor Code

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

Inspector Code for DIA

```
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];
   }
```

CONCLUSION

- Abstractions for data transformations in sparse matrix & unstructured mesh computations
- Vision: Create a framework to compose complex transformation sequences for inspectors and executors
- Minimize inspector passes over input data
- Extend IDG to support fusion of Inspectors
- Integrate existing inspector library functions