The Big Picture Problem

- Data movement is expensive …
  - in terms of execution time
  - in terms of power usage
- Data reordering and/or loop transformations can turn data reuse into data locality
- Research in the polyhedral model has led to significant automation for loop transformations that affect data locality
- However, sparse/irregular computations do not fit in the polyhedral model
Loop Transformation Example

Code generators like Cloog and Omega generate code to traverse resulting polyhedron.

Loop Transformation Frameworks

- Currently are used in some compilers to …
  - abstractly represent loops, array accesses, and data dependences in loops
  - abstract loop transformations and their effect
  - generate code for transformed loop

- Examples
  - Unimodular framework [Banerjee 90, Wolf & Lam 91]
  - Polyhedral framework [Feautrier, Pugh, Rajopadhye, Cohen, …]

- Remaining issue: Array accesses and loop bounds must be affine expressions of iterators
MOLDYN: A Computation with Indirect Memory References

- [Han & Tseng 2000]
- Extracted from CHARMM (Chemistry at HARvard Macromolecular Mechanics)
- Computes non-bonded interactions
- We use molecules from protein data bank and create interaction lists with realistic cutoff distances

Colorado State University

```plaintext
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
```

Run-time Reordering Transformations To the Rescue

- Traverses index array
- Generates data reordering function $\sigma$
- Reorder data and updates index array

Inspector

```plaintext
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

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

Executor

```plaintext
for i=0,7
  Y[i] = Z'[r'[i]]
```
Example Run-Time Reordering Transformations
implemented with Inspector/Executor Strategies

- Gather/scatter parallelization [Saltz et al. 89]
- Cache blocking [Im & Yelick 98]
- Irregular cache blocking [Douglas & Rude 00]
- Full sparse tiling (ICCS 2001)
- Run-time data and iteration permutation [Chen and Kennedy 99, Mitchell 99, …]
- Compositions of the above (PLDI 2003)
- Communication avoiding [Demmel et al 08, 09]
- Producer/consumer parallelization [Basumallik06, Ravishankar12]

Inspector/Executor Strategies show great promise BUT…

- Only a few specific transformations have been automated
- There is library support for some I/E strategies, but specializing the library for the given sparse data structures is non-trivial
- How can we automate or semi-automate the application of I/E strategies within the compiler?
Run-time Reordering Transformations

- Challenge: unable to effectively reorder data and computation at compile-time in sparse applications with indirect memory references
- Approach: run-time reordering transformations
- Vision:

Sparse Polyhedral Framework (SPF):
A new intermediate representation

- Adds uninterpreted functions to the polyhedral framework
  - Polyhedral model represents loops as sets and schedules and data accesses as relations, all with affine inequality constraints.
  - SPF adds constraints such as $j = f(k)$, where $f$ could represent a reordering like sigma from example.
  - [Pugh & Wonnacott 94] used for data dependence analysis. SPF uses to represent transformations.

- Code generation after SPF transformation results in inspector and executor code.
Recall Run-time Reordering Transformations

Example

Traverses index array
Generates data reordering function $\sigma$
Reorder data and updates index array

Original Code

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

Executor

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

Inspector

for $i=0,7$
... $r[i] ...$
for $j=0,7$
sigma[j] = ...
for $j=0,7$
$Z'[\sigma[j]] = Z[j]$
$r'[j] = \sigma[r[j]]$

Computation Specification in SPF:
Intermediate Representation for Loops

Original Code

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

- Each data array has a data space
$Y_0 = \{[y] \mid 0 \leq y \leq 7\}$
$Z_0 = \{[z] \mid 0 \leq z \leq 7\}$
- Each index array is represented with an uninterpreted function $r()$
and has a domain and range
$\{[v] \rightarrow [w] \mid 0 \leq v, w \leq 7\}$
Computation Specification in SPF

Original Code

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

- Each statement represented with …
- An iteration space set \( S0 = \{[i] \mid 0 \leq i \leq 7\} \)
- Scheduling function \( S_{S0 \rightarrow I_0} = \{[i] \rightarrow [0, i, 0]\} \)
- Access functions for each data array reference
  \[
  A_{S0 \rightarrow Y_0} = \{[i] \rightarrow [i]\}
  \]
  \[
  A_{S0 \rightarrow Z_0} = \{[i] \rightarrow [z] \mid z = r(i)\}
  \]

Computation Specification in SPF: Intermediate Representation

C for loop + pragmas

IEGenCC tool helps generate the SPF specification of algorithm.
Transformation Specification in SPF and Inspector/Executer Generator (IEGen)

Data Reordering Transformation Relation
\[ R_{Z_0 \to Z_1} = \{ [z] \to [z'] \mid z' = \sigma(z) \} \]

Inspector Dependence Graph (IDG)

Recall: Computation Specification in SPF cont …

Original Code

\[ \text{for } i=0,7 \]
\[ Y[i] = Z[r[i]] \]

- Each statement represented with …
  - An iteration space set \( S_0 = \{ [i] \mid 0 \leq i \leq 7 \} \)
  - Scheduling function \( S_{S_0 \to I_0} = \{ [i] \to [0, i, 0] \} \)
  - Access functions for each data array reference
    \[ A_{S_0 \to Y_0} = \{ [i] \to [i] \} \]
    \[ A_{S_0 \to Z_0} = \{ [i] \to [z] \mid z = r(i) \} \]
Each statement represented with …

An iteration space set \( S_0 = \{ [i] \mid 0 \leq i \leq 7 \} \)

Scheduling function \( S_{S_0 \rightarrow I_0} = \{ [i] \rightarrow [0, i, 0] \} \)

Access functions for each array reference

\( A_{S_0 \rightarrow Y_0} = \{ [i] \rightarrow [i] \} \)

\( A_{S_0 \rightarrow Z_1} = \{ [i] \rightarrow [z'] \mid z' = \sigma(r(i)) = r'(i) \} \)

Implementation Details: Transformations on Computation and Data Spaces

Reordering transformations modify the computation specification
Key Insights in SPF and IEGen

- The inspectors traverse the access functions and/or the data dependences
- We can express how the access relations and data dependences will change
- Subsequent inspectors traverse the new data mappings and data dependences
- Use polyhedral code generator (Cloog) for outer loops and deal with sparsity in inner loops and access relations
Punchline: Key Ideas in this Talk

- The Sparse Polyhedral Framework enables run-time reordering transformations.
- Existing tools do not transform indirect array accesses with enough precision.
- The PIES group is working on algorithms for doing the necessary manipulation.
- IEGenLib is an open-source library that implements the algorithms.

http://www.cs.colostate.edu/hpc/PIES

Another Short Story: Matrix Powers Kernel (with pictures)

- A is a sparse matrix
- Solving for a set of vectors \{A^0 x, A^1 x, \ldots, A^m x\}

\[
\text{for (k=1; k<=m; k++)} \{
\text{for (p=0; p<nz; p++)} \{
\quad x[k][\text{row[p]}] += a[p]*x[k-1][\text{col[p]}];
\}
\}
\]
**Villian**: Sparse Matrix Vector Multiply

- Non-zero pattern can cause poor data locality in vectors

\[ A^{k-1} \times = A^k \times \]

**Hero**: Run-time Data Reordering

- Permute the rows and columns to improve data locality of accesses to vectors
**Support:** Inspector/Executor Strategies

- Traverses index arrays
- Generates data
- Reorders data

**Inspector**

```c
for (p=0; p<nz; p++) {
    ... row[p] ...
    ... col[p] ...
}
```

```c
for (i=0; i<n; i++) {
    sigma[j] = ...
}
for (i=0; i<n; i++)
    x'[0][sigma[i]] = x[0][i];
```

**Executor**

```c
for (k=1; k<=m; k++) {
    for (p=0; p<nz; p++)
        x[k][row[p]] += a[p]*x[k-1][col[p]]; }
```

```c
for (t=1; t<=Nt; t++) {
    for (k=1; k<=m; k++) {
        for (i=0; i<n; i++) {
            for (p=0; p<nz; p++) {
                if (sigma[row[p]]==i && tile(k,i)==t))
                    x'[k][sigma[row[p]]] += a[p]*x'[k-1][sigma[col[p]]];
            }
        }
    }
}
```

---

**Sidekick: Run-time Iteration Reordering**

- Sparse tile to improve temporal data locality with a sparse tiling transformation

- Note that cleanup transformations to the code will remove nested indirect accesses and extra iterations of innermost loop
Support? Loop Transformation Frameworks

- Transformations to improve locality and expose parallelism
- Frameworks include ...
  - An Intermediate Representation (IR) of computation
  - Abstractions for loop transformations and their effect
  - Techniques for generating code for transformed loop
- Examples (Problem: focus is on affine accesses and bounds)
  - Unimodular framework [Banerjee 90, Wolf & Lam 91]
  - Polyhedral framework [Feautrier, Pugh, Rajopadhye, Cohen, …]

New Tech: Sparse Polyhedral Framework

- SPF enables run-time reordering transformations
- Example: Matrix Powers Kernel, $A^m x$

Reorder rows/cols and sparse tile

```c
for (k=1; k<=m; k++) {
    for (p=0; p<nz; p++) {
        x[k][sigma[row[p]]] += a[p] * x[k-1][col[p]];
    }
}
```
**Specific Problem:** Not Enough Precision

- Existing tools are not able to manipulate the computation representation with enough precision.
- For matrix powers kernel, after we reorder the indirect array accesses need to change:

  \[
  x[k-1][\text{col}[p]] \rightarrow x[k-1][\text{sigma}[\text{col}[p]]] \\
  x[k][\text{row}[p]] \rightarrow x[k][\text{sigma}[\text{row}[p]]]
  \]

**Plot so far: Talk Outline**

- Matrix powers kernel needs run-time reordering transformations for improved performance.
- Inspector/executor strategies implement run-time reordering transformations.
- The Sparse Polyhedral Framework enable us to transform code with indirect array accesses.
- **Next:** SPF in action and our team runs into a little snag.
\( SPF: \) Loops as Sets (IR)

\[
I = \{ [k, p] \mid 1 \leq k < m \land 0 \leq p < nz \}
\]

```
for (k=1; k<=m; k++) {
    for (p=0; p<nz; p++) {
        x[k][row[p]] += a[p]*x[k-1][col[p]];
    }
}
```

\( SPF: \) Access Functions as Relations (IR)

\[
A_{1I \to X} = \{ [k, p] \rightarrow [v, w] \mid v = k - 1 \land w = col(p) \}
\]

```
for (k=1; k<=m; k++) {
    for (p=0; p<nz; p++) {
        x[k][row[p]] += a[p]*x[k-1][col[p]];
    }
}
```
**SPF: Data Run-time Reordering Transformations as Relations**

Data reordering transformation

\[ R_{X \rightarrow X'} = \{ [k, i] \rightarrow [k, i'] \mid i' = \sigma(i) \} \]

**SPF: Iteration Run-time Reordering Transformations as Relations**

Iteration reordering transformation

\[ T_{I \rightarrow I'} = \{ [k, p] \rightarrow [t, k, i, p] \mid t = \text{tile}(k, i) \land i = \text{sigma}(\text{row}(p)) \} \]
**S*PF*: Transforming Access Function A1

**Operations Needed**
- **Apply**
  \[ X' = R_{X \rightarrow X'}(X) \]
  \[ I' = T_{I \rightarrow I'}(I) \]
- **Compose**
  \[ A_{I \rightarrow X'} = R_{X \rightarrow X'} \circ A_{I \rightarrow X} \]
  \[ A_{I' \rightarrow X'} = T_{I' \rightarrow I} \circ A_{I \rightarrow X'} \]
- **Inverse**
  \[ T_{I' \rightarrow I} = T_{I \rightarrow I'}^{-1} \]

---

**Existing Tech: Composition Problem**

- **Definition of composition**
  \[ r = r_2 \circ r_1 = (\vec{x} \rightarrow \vec{z} \in r) \iff (\exists \vec{y} \mid \vec{x} \rightarrow \vec{y} \in r_1 \land \vec{y} \rightarrow \vec{z} \in r_2) \]
  \[ \vec{x} \rightarrow \vec{y} \rightarrow \vec{z} \]

- **Need to project out all variables in vector \( \vec{y} \)**
- **Requires projecting out variables that are uninterpreted function parameters**

\[ A_{I \rightarrow X'} = R_{X \rightarrow X'} \circ A_{I \rightarrow X} \]
\[ R_{X \rightarrow X'} = \{ [k, i] \rightarrow [k, i'] \mid i' = \sigma(i) \} \]
**Insight: The Solution**

- Assume a relation is either a function or the inverse of a function.
  - Iteration and data reordering transformations are typically functions
  - Access relations are functions
- Create closed-form solutions for compose and apply given these constraints

**Solution: Avoid Projection Altogether**

- Compose when both relations are functions
  \[
  \{\vec{y} \rightarrow \vec{z} | \vec{z} = F1(\vec{y}) \land C1\} \circ \{\vec{x} \rightarrow \vec{y} | \vec{y} = F2(\vec{x}) \land C2\}
  \]
  equals
  \[
  \{\vec{x} \rightarrow \vec{z} | (\exists \vec{y} | \vec{z} = F1(\vec{y}) \land C1 \land \vec{y} = F2(\vec{x}) \land C2)\}
  \]
  which is equivalent to
  \[
  \{\vec{x} \rightarrow \vec{z} | \vec{z} = F1(F2(\vec{x})) \land C1[\vec{y}/F2(\vec{x})] \land C2[\vec{y}/F2(\vec{x})]\}\]
  Substitute \( y \) with \( F2(x) \)
IEGenLib (Inspector/Executor Generator Library)

Operations:
- define set, relation
- apply relation to set
- union sets or relations
- inverse relation
- compose relation

Inspector/Executor Generator (IEGen)

Sparse Polyhedral Framework (SPF) specification of algorithm

Explicit Relation Generators
- consecutive packing
- sparse tiling

Inspector/Executor Generator (IEGen)

Sequence of SPF specifications of transformations

Available at http://www.cs.colostate.edu/hpc/PIES
Using IEGenLib for Transformation

```python
import iegenlib
# Python code for access function for x[k-1][col[p]]
A1_I_to_X = Relation("f[k,p] -> [v,w] : v=k-1 && w=col(p)g")

# Data reordering after sparse tiling transformation
T_I_to_Iprime = Relation("f[k,p] -> [t,k,i,p] : t=tile(k,i) && i=sigma(row(p)) && 0 <= t && t < N_t && 0 <= i && i < N_r")

# Composing the two
A1_Iprime_to_X = A1_I_to_X.compose( T_I_to_Iprime.inverse() )

Output:
{ [t, k, i, p] -> [k1, i'] : i - sigma(row(p)) = 0 && i' - sigma(col(p)) = 0 && k - k1 - 1 = 0 && t - tile(k, i) = 0 && N_r - i - 1 >= 0 && N_t - t - 1 >= 0 && i >= 0 && t >= 0 }
```

And Now for Some Other Cool Stuff!

- Using full sparse tiling to improve the parallel scalability on multicores
- Separating algorithm and implementation detail specification in scientific computing applications
Parallelization using Inspector/Executor Strategies

Break computation that sweeps over mesh/sparse matrix into chunks/sparse tiles

Full Sparse Tiled Iteration Space

Task Graph

Full Sparse Tiling Helps Reduce Memory Bandwidth Demands

Speedup of FST vs Blocked Jacobi
16 cores, pwtk matrix, 2000 iterations, 30.8 GB/s Triad BW
And Now for Some Other Cool Stuff!

- Using full sparse tiling to improve the parallel scalability on multicores
- Separating algorithm and implementation detail specification in scientific computing applications

Specifying Implementation Details Orthogonally

<table>
<thead>
<tr>
<th>Source-to-Source compilation tool</th>
<th>Algorithm Specification</th>
<th>Implementation Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td>for loops (some restrictions)</td>
<td>static or dynamic, block or not, private and shared, …</td>
</tr>
<tr>
<td>CUDA</td>
<td>for loop (some restrictions)</td>
<td>unroll, vectorize, data movement, …</td>
</tr>
<tr>
<td>ORIO, POET, …</td>
<td>for loop (some restrictions)</td>
<td>unroll, tile, various loop transformations, …</td>
</tr>
</tbody>
</table>

**SAIMI project focus**

<table>
<thead>
<tr>
<th>Mesa</th>
<th>expressions</th>
<th>lookup table optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEGen</td>
<td>for loops with indirect accesses</td>
<td>inspector/executor strategies specified using Sparse Polyhedral Framework (SPF)</td>
</tr>
</tbody>
</table>
Past, Current, and Future SAIMI Work

- Small Angle X-ray Scattering
  - Up to 7x speedup due to look up table optimization
  - Created Mesa tool
  - Evaluated tool and LUT opt on 4 other applications

- Parallel Ocean Program
  - Developed the CGPOP miniapp
  - Co-Array Fortran comparison with Fortran+MPI

- GridGen
  - Separate grid specification from computation and parallelization spec

SAIMI - Separating Algorithm and Implementation via programming Model Injection

- Keep code in existing general purpose programming languages
- Annotate sub computations with pragmas to inject implementation details
- Focus on three “injectable” programming models: expressions, sparse polyhedral model, task graphs due to sparse tiling
- Show approach can be used on DOE applications (e.g., CGPOP miniapp)
The End: SPF Summary

- Sparse Polyhedral Framework (SPF) provides abstractions needed to automate performance transformation of irregular/sparse apps
- Sparse tiling is an inspector/executor strategy that …
  - turns data reuse into data locality
  - enables putting off the point at which bandwidth bound computations quit scaling
- Inspector/executor code generator (IEGen) will provide an approach for semi-automating the application of inspector/executor strategies

Contributors

- Original SPF concept developed in collaboration with Larry Carter and Jeanne Ferrante
- Chris Krieger – Task graph programming model and possibly composition of prog models
- Andy Stone – CGPOP and GridGen design and development
- John Dennis – NCAR collaborator, CGPOP
- Geri George – Project planning and management
- Catherine Olschanowsky – IEGenLib development
- Alum: Alan LaMielle – IEGen prototype
- Alum: Jon Roelofs – IEGenCC tool prototype
- Undergraduates: Mark Heim, Wes Jeannette, Ian Craig
- Christina Williams – Cool PIES icon
Data Permutation Reordering
(Equations are the compile-time abstraction)

\[ R_{Z_0 \rightarrow Z_1} = T_{t_0 \rightarrow t_1} = \{[i] \rightarrow [\sigma(i)]\} \]

CPACK reordering heuristic [Ding & Kennedy 99]

\[ A_{J_0 \rightarrow Z_0} = \{[j] \rightarrow [i] | i = l(j) \lor i = r(j)\} \]

\[ A_{J_0 \rightarrow Z_1} = \{[j] \rightarrow [i] | i = \sigma(l(j)) \lor i = \sigma(r(j))\} \]

Effect of Data Reordering on Inspector Dependence Graph (IDG)

\[ l : \text{index array} \quad r : \text{index array} \]

construct explicit relation

\[ \{[j] \rightarrow [i] | i = l(j)\} \cup \{[j] \rightarrow [i] | i = r(j)\} \]

cpack(…)

\[ \{[i] \rightarrow [v] | v = \sigma(i)\} \]

reorderArray(…)

\[ Z_0 : \text{data array} \]

\[ Z_1 : \text{data array} \]
Iteration Permutation Reordering

\[ T_{J_0 \to J_1} = \{ [j] \to [x] \mid x = \delta(j) \} \]

\[ A_{J_0 \to Z_1} = \{ [j] \to [i] \mid i = \sigma(l(j)) \lor i = \sigma(r(j)) \} \]

\[ A_{J_1 \to Z_1} = \{ [j] \to [i] \mid i = \sigma(l(\delta^{-1}(j))) \lor i = \sigma(r(\delta^{-1}(j))) \} \]

IDG After Iteration Permutation
Gauss-Seidel tech report

- Data dependence example
  \[ d_{11} = \{ [\text{iter}_1, i] \rightarrow [\text{iter}_2, i] \mid (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \land (0 \leq i < R) \} \]

- Constraint from data dependence
  \[
  \forall \text{iter}_1, \text{iter}_2, i : (1 \leq \text{iter}_1 < \text{iter}_2 \leq T) \land (0 \leq i < R) \Rightarrow \theta(\text{iter}_1, \sigma^{-1}(i)) \leq \theta(\text{iter}_2, \sigma^{-1}(i))
  \] (7)

- Showed post conditions of alg satisfied constraint

Computation and Transformation Specification in SPF

- Data and index array specifications
- Each statement represented with ...
  - An iteration space set
  - A **schedule mapping** to full iteration space
  - **Access functions** for each data array reference

- **Data dependences relations** between iterations in full iteration space

- **Transformation specification** is a sequence of data and iteration reorderings represented as integer tuple **relations**