CS 526
Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526
DEPENDENCE TRANSFORMS

The slides adapted from Vikram Adve
Motivation

Memory hierarchy optimizations
Goal 1: Improving reuse of data values within loop nest
Goal 2: Exploit reuse to reduce cache, TLB misses

Tiling
Goal 1: Exploit temporal reuse when data size > cache size
Goal 2: In parallel loops, reduce synchronization overhead

Software Prefetching
Goal: Prefetch predictable accesses k iterations ahead

Software Pipelining
Goal: Extract ILP from multiple consecutive iterations

Automatic parallelization Also, auto-vectorization
Goal 1: Enhance parallelism
Goal 2: Convert scalar loop to explicitly parallel
Goal 3: Improve performance of parallel code
Reordering Transformation

**Definition.** Legal Transformation preserves the meaning of that program, i.e., all externally visible outputs are identical to the original program, and in identical order.

• We consider two programs equivalent (i.e., the transformation preserving the program meaning) if on the same inputs both the original and transformed programs, after being executed, produce the same outputs.

**Theorem.** A reordering transformation that preserves all data dependences in a program is a legal transformation.

*For discussion, see Allen and Kennedy book.*
Dependence Distance

**Dependence Distance:** If there is a dependence from statement S1 on iteration \( I \) and statement S2 on iteration \( I' \) then the corresponding dependence distance vector is

\[
d_{I,I'} = [I'_1 - I_1, \ldots, I'_k - I_k]
\]

*Note: Computing distance vectors is harder than testing dependence*
Dependence Distance

Direction Vector: For a distance vector of the form \( d_{I,I'} = [I'_1 - I_1, ..., I'_k - I_k] \) the corresponding direction vector is \( \delta_{I,I'} = [\delta_1, ..., \delta_k, ..., \delta_m] \), where

\[
\delta_k = \begin{cases} 
- , & \text{if } I'_k - I_k < 0 \\
+ , & \text{if } I'_k - I_k > 0 \\
= , & \text{if } I'_k - I_k = 0 \\
* , & \text{if sign } +, -, = 
\end{cases}
\]

Note: \( I < J \) iff the leftmost non-’=’ entry in \( \delta(I,J) \) is ’+’.

- We use the property of lexicographical ordering
**Loop-Carried Dependence**

Statement S2 has a loop carried dependence on statement S1 iff S1 references location M on iteration \( I \), S2 references M on iteration \( I' \) and \( d(I,I') > 0 \).

```latex
do i = 1 to N
    A(i+1) = B(i)
    B(i+1) = A(i)
enddo
```

**Level** of loop-carried dependence is the leftmost non-“=“ sign in the direction vector

- Forward dependence: S1 appears before S2 in the loop body
- Backward dependence: S2 appears before S1 in the loop body
## Reordering Transformations

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preprocessing transformations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop normalization</td>
<td>Make loops canonical</td>
<td>Simplify, improve dep. analysis</td>
</tr>
<tr>
<td>Ind. var. substitution</td>
<td>Identify aux. induction vars</td>
<td>Improve dependence information</td>
</tr>
<tr>
<td>Scalar expansion</td>
<td>Replace scalar with array</td>
<td>Eliminate spurious dependences</td>
</tr>
<tr>
<td>Scalar/array privatization</td>
<td>Treat var. as iteration-private</td>
<td>Eliminate spurious dependences</td>
</tr>
<tr>
<td>Variable renaming</td>
<td>Use multiple copies of vars</td>
<td>Eliminate anti- and output-dependences</td>
</tr>
<tr>
<td>Reduction recognition</td>
<td>Recognize reductions</td>
<td>Ignore special-case dependences</td>
</tr>
<tr>
<td><strong>Reordering transformations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop interchange</td>
<td>Change loop nesting order</td>
<td>Cache, parallelism, vectorization</td>
</tr>
<tr>
<td>Loop strip-mining</td>
<td>Make 2 nested loops</td>
<td>”</td>
</tr>
<tr>
<td>Loop skewing</td>
<td>Change wavefront loop to parallel</td>
<td>Improve loop parallelism</td>
</tr>
<tr>
<td>Loop reversal</td>
<td>Run loop backwards</td>
<td>Reduce array storage</td>
</tr>
<tr>
<td>Index set splitting</td>
<td>Break loop by index space</td>
<td>Remove some deps.</td>
</tr>
<tr>
<td>Loop distribution</td>
<td>Break loop by statements</td>
<td>Simplify parallelization, vectorization</td>
</tr>
<tr>
<td>Loop alignment</td>
<td>Change carried to indep.</td>
<td>Simplify parallelization, vectorization</td>
</tr>
<tr>
<td>Loop fusion</td>
<td>Join loops by statements</td>
<td>Improve cache reuse</td>
</tr>
</tbody>
</table>
Math Intermezzo: Unimodular Matrix

A matrix $T$ is unimodular iff it is a square integer matrix with determinant $\pm 1$ or $-1$.

These properties will help us compose transformations:

- Product of two unimodular matrices is also unimodular
- Its inverse is also unimodular

For each integer vector $x$, a unimodular matrix $T$ maps it into a unique vector $y = Tx$. 

\[
\begin{bmatrix}
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{bmatrix}
\]
A transformation is called \textit{unimodular} if the matrix $T$ is unimodular (i.e., square integer matrix with determinant $+1$ or $-1$).

\begin{align*}
\text{Loop interchange: } T &= \begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}, \vec{t} = \vec{0} \\
\text{Loop reversal: } T &= [-1], \vec{t} = (U_1 - 1)
\end{align*}

Legality of the transformation: $T \cdot \vec{t} \geq 0$
Examples of Unimodular Transformations

**Interchange**

\[
\text{for } i=2 \text{ to } N \\
\quad \text{for } j=2 \text{ to } M-1 \\
\quad \quad A[i,j] = A[i,j]*2 \\
\quad \quad A[i,j] = A[i,j]*2 \\
\quad \text{end for} \\
\text{end for}
\]

Transform matrix

\[
\begin{bmatrix}
  i' \\
  j'
\end{bmatrix} = \begin{bmatrix}
  0 & 1 \\
  1 & 0
\end{bmatrix} \begin{bmatrix}
  i \\
  j
\end{bmatrix}
\]

**Reversal**

\[
\text{for } k=1 \text{ to } L \\
\text{endfor}
\]

\[
[k'] = [-1][k] + L
\]

**Skew**

\[
\text{for } i=2 \text{ to } N \\
\quad \text{for } j=2 \text{ to } N \\
\quad \text{end for} \\
\text{end for}
\]

\[
\begin{bmatrix}
  i' \\
  j'
\end{bmatrix} = \begin{bmatrix}
  1 & 0 \\
  1 & 1
\end{bmatrix} \begin{bmatrix}
  i \\
  j
\end{bmatrix}
\]
Legality of Unimodular Transformations

A transformed loop nest is equivalent to the original if it preserves all dependencies. A transformation between these two nets is legal if the nests are equivalent.

Let D be the set of distance vectors of a loop nest. A unimodular transformation T is legal if and only if

$$\forall d \in D \ . \ T \cdot d \geq 0$$

Proof sketch (from Banerjee, Unimodular Transformations 2011):
Consider loop body S of the original nest and S’ of the transformed one. Two iterations S(I) and S(I’) in the original nest become S’(TI) and S’(TI’) in the transformed. S’(TI) precedes S’(TI’) iff T · I’ ≥ T · I.

“if part”: For each d, assume T · d ≥ 0. Consider that a statement S(I’) in iteration I’ depend on the statement S(I). Because d = I’ − I is the distance vector in the original loop, we get T · I’ − T · I = T(I’ − I) ≥ 0. With this we get that all dependencies are preserved in the transformed loop, i.e. the two loop nests are equivalent.

“only-if part”: Assume the transformation is legal. Let d=I’-I denote a distance in the original loop (and the statement in the iteration ‘I’ depends on the one in iteration I). By hypothesis (the loop nests are equivalent), T · I’ ≥ T · I, so then T · I’ − T · I ≥ 0 and so T · (I’−I) = T · d ≥ 0.
Loop Interchange

**Informal Definition:** Change nesting order of loops in a **perfect loop nest**, with no other changes.

```
for i=2 to N
    for j=2 to M-1
    end for
end for
```

```
for j=2 to M-1
    for i=2 to N
    end for
end for
```
Uses of Loop Interchange

1. Move independent loop innermost
2. Move independent loop outermost
3. Make accesses stride-1 in memory
4. Loop tiling (combine with strip-mining)
5. Unroll-and-jam (combine with unrolling)
Loop Interchange

Direction Vectors and Loop Interchange:
If $\delta$ is a direction vector of a particular dependence $S_1 \rightarrow S_2$ in a loop nest and the order of loops in the loop nest is permuted, then the same permutation can be applied to $\delta$ to obtain the new direction vector for the conflicting instances of $S_1$ and $S_2$.

Direction Matrix: A matrix where each row is the direction vector of a single dependence, i.e.,
each row $\leftrightarrow$ a dependence
each column $\leftrightarrow$ a loop
Loop Interchange Properties

Legality: A permutation of the loops in a perfect nest is legal iff the direction matrix, after the permutation is applied, has no “-” direction as the leftmost non-“=” direction in any row

• Recall, for legality the vector after transformation should be lexicographically greater than “0” vector.

• Some more intuition: To preserve the dependencies, consider the cases before transformation of (=,=) [independent], (=,+) and (+,=) [the dependence is still carried but by the outer (resp. inner loops)], (+,+), (+,-) [Dependence is still carried]. But (+,-) is illegal since the antidependence turns into a true dependence.

Profitability: machine-dependent:

1. vector machines
2. parallel machines
3. caches with single outstanding loads
4. caches with multiple outstanding loads
Direction Matrix

Direction Matrix:
each row ↔ a dependence
each column ↔ a loop

for i = 2 to N
    for j = 2 to M-1
        Sp: \( A[i,j] = B[i-1,j-1] \)
    endfor
endfor

\[
\begin{align*}
\text{Sp} \rightarrow \text{Sq}: & \quad A[i,j]/A[i,j] = = \\
\text{Sp} \rightarrow \text{Sq}: & \quad A[i,j]/A[i-1,j] = + \\
\text{Sq} \rightarrow \text{Sp}: & \quad B[i,j]/B[i-1,j-1] = + \\
\end{align*}
\]
Direction Matrix (Illegal)

Direction Matrix:
each row ↔ a dependence
each column ↔ a loop

for i = 2 to N
  for j = 2 to M-1
    Sp: A[i,j] = B[i-1,j-1]
  endfor
endfor

Sp→Sq: A[i,j]/A[i,j] = =
Sp→Sq: A[i,j]/A[i-1,j+1] + -
Sq→Sp: B[i,j]/B[i-1,j-1] + +
Applying Loop Interchange

1. **Single ’+’ entry**: a “serial loop”
   - Move loop outermost for vectorization
   - Move loop innermost for parallelization

2. **Multiple ’+’ entries**: Outermost one carries dependence
   - Loop carrying the dependence *changes* after permutation!
   - May still benefit by moving carried-dependences to the outermost loop
Example

for $i = 1$ to $n$
    for $j = 1$ to $m$
        end for
    end for
end for

for $i = 1$ to $n$
    for $j = 1$ to $m$ // vectorize
    end for
end for

**parallel** for $j = 1$ to $m$
    for $i = 1$ to $n$
    end for
end for
Loop Reversal

Informal Definition: Reverse the order of execution of the iterations of a loop

for $i=2$ to $N$
    for $j=2$ to $M-1$
        for $k=1$ to $L$
        endfor
    endfor
endfor

for $i=2$ to $N$
    for $j=2$ to $M-1$
        for $k=L$ to $1$ step $-1$
        endfor
    endfor
endfor
Legality of Loop Reversal

The loop that is reversed should not carry dependence

Recall, *Legality*: the vector after transformation should be lexicographically greater than “0” vector.

E.g., \((1, -1) > (0,0)\) but \((-1, 1) < (0,0)\)

In our case, two dependencies:

\[
\begin{align*}
(1) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} + & = + > 0 \\
(2) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} - & = + > 0
\end{align*}
\]
Uses of Loop Reversal

Convert a ’-’ to a ’+’ in a direction vector to enable other transformations, e.g., loop interchange.

Scalarize a vector statement (e.g., in Fortran 90) by ensuring that values are read before being written.

- Scalarized code:

  \[
  \text{for } i = 64 \text{ to } 2 \text{ step } -1 \\
  A[i] = A[i-1] \times e \\
  \text{endfor}
  \]
Loop Skewing

**Informal Definition:** Increase dependence distance by n by substituting loop index \( j \) with \( jj = j + n \* i \).

E.g., For \( n = 1 \), we use \( jj = j + 1 \)

\[
\begin{align*}
\text{for } i=2 \text{ to } N & \quad \text{for } i=2 \text{ to } N \\
\quad \text{for } j=2 \text{ to } N & \quad \text{for } jj=i+2 \text{ to } i+N \\
\quad \quad \quad \quad A[i,j] = A[i-1,j] & \quad \quad \quad A[i,jj-i] = A[i-1,jj-i] \\
\quad \quad \quad \quad + A[i,j-1] & \quad + A[i,jj-i-1] \\
\quad \end{align*}
\]

- Improve parallelism by converting ‘=’ to ‘+’ in a direction vector
- Improve vectorization in a similar way
- (Rarely) Could be used to *simplify* index expressions
for $I_1 := 0$ to $5$ do
  for $I_2 := 0$ to $6$ do

$D = \{(0, 1), (1, 0), (1, -1)\}$.

for $I'_1 := 0$ to $5$ do
  for $I'_2 := I'_1$ to $6 + I'_1$ do

$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$

$D' = TD = \{(0, 1), (1, 1), (1, 0)\}$
Loop Strip Mining

**Informal Definition** Convert a single loop into two nested loops for a specified “block size”

(*Always safe.*)

for i=1 to N
  A[i] = x + B[i] * 2
end for

for ii=1 to N step B
  for i=ii to min(ii+B-1, N)
    A[i] = x + B[i] * 2
  end for
end for
Loop Strip Mining Applications

- **Loop tiling:** *strip-mine* and then *interchange* multiple uses. Can be useful for increasing cache locality or blocking parallel loops;

```
for j=1 to N
  for ii=1 to N step B
    for i=ii to min(ii+B-1, N)
      A[i][j] = x + B[i][j]
```

- **Prefetching:** strip-mine by cache line size; prefetch once per outer iteration

- **Instruction scheduling:** strip-mine and then unroll inner loop

```
for ii=1 to N step B
  for j=1 to N
    for i=ii to min(ii+B-1, N)
      A[i][j] = x + B[i][j]
```
Tiling Example

\[
\text{for } I'_1 := 0 \text{ to } 5 \text{ do } \\
\text{for } I'_2 := I'_1 \text{ to } 6 + I'_1 \text{ do } \\
A[I'_2 - I'_1 + 1] := 1/3 \times (A[I'_2 - I'_1] \\
+ A[I'_2 - I'_1 + 1] + A[I'_2 - I'_1 + 2]); \\
\]

\[
T' = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}
\]

\[
D' = TD = \{(0, 1), (1, 1), (1, 0)\}
\]

\[
\text{for } II'_1 := 0 \text{ to } 5 \text{ by } 2 \text{ do } \\
\text{for } II'_2 := 0 \text{ to } 11 \text{ by } 2 \text{ do } \\
\text{for } I'_1 := II'_1 \text{ to } \min(5, II'_1 + 1) \text{ do } \\
\text{for } I'_2 := \max(I'_1, II'_2) \text{ to } \min(6 + II'_1, II'_2 + 1) \text{ do } \\
a[I'_2 + 1] := 1/3 \times (a[I'_2] + a[I'_2 + 1] + a[I'_2 + 2]);
\]
Loop Distribution

Informal Definition: Convert a loop nest containing two or more statements into two or more distinct loop nests so that each statement appears in only a single resulting loop nest.

for i = 2 to N
S1: A[i] = B[i] + C[i]
S2: D[i] = A[i] * 2.0
end for

for i = 2 to N
S1: A[i] = B[i] + C[i]
end for

for i = 2 to N
S2: D[i] = A[i] * 2.0
end for
Loop Distribution Applications

• Create perfect loops nests for other transformations like loop interchange
• Convert a loop-carried dependence within a loop into a loop-independent dependence crossing two loops:

```
for i=2 to N
S1:   A[i] = B[i] + C[i]
S2:   D[i] = A[i-1] * 2.0
end for
```
Maximal Loop Distribution

- Identify the SCCs of the data dependence graph, to group statements in an SCC in a single loop nest.
- Sort the SCCs using a topological sort on the dependence graph.
- Generate distinct loop nests, one for each SCC, in sorted order.
- If we have control dependence between a statement $S_1$ is one SCC and the statement $S_2$ in another SCC, create an array ‘flags’ that contains the Boolean conditions, populate it in the first SCC that induce dependence and use them in the second SCC.

Reminder:
- **Strongly connected graph**: a directed graph in which there is a path between all pairs of vertices.
- **Strongly connected component (SCC)** is a maximal strongly connected subgraph.
Loop Fusion

**Informal Definition:** Merge two or more distinct (perhaps non-adjacent) loops with identical loop bounds into a single loop.

```plaintext
for i=1 to N
    A[i] = i*i
end for

for i=1 to N
    B[i] = A[i] + 1
end for
```
Loop Fusion

for i=1 to M
    for j=1,N-1
        A[j,i] = i*i + j*j
    end for
    for j=1 to N
        B[j,i] = A[j,i] + i + j
    end for
end for

for i=1 to M
    for j=1 to N-1
        A[j,i] = i*i + j*j
        B[j,i] = A[j,i] + i + j
    end for
    // peel last iteration:
    j=N
    B[j,i] = A[j,i] + i + j
end for
Loop Fusion Motivation

- Increase cache reuse (if same array accessed in two loops) Fundamental optimization for array languages (e.g., Fortran 90, HPF, MATLAB, APL)

  Example in F90:
  
  \[
  \]

- Increase granularity of parallelism (work per iteration) Important for shared-memory parallelism (the model with parallel loop and barriers)
Legality of Loop Fusion

Fusion-Preventing Dependence: A loop-independent dependence from S1 to S2 in different loops is fusion-preventing if fusing the two loops causes the dependence to become a loop-carried dependence from S2 to S1.

Legality of Loop Fusion: Two loops can be fused if all three conditions are satisfied:
1. Both have identical bounds (*transform loops if needed*)
2. There is no fusion-preventing dependence between them.
3. There is no path of loop-independent dependences between them that contains a loop or statement that is not being fused with them.
Loop Fusion: Illegal Cases

for i=1 to M
    for j=2 to N
        A[j,i] = B[j-1,i] * 2
    end for

for j=2 to N
end for

for i=1 to M
    for j=2 to N
        t[j] = B[j-1,i]
    end for

for j=2 to N
    A[j,i] = t[j] * 2
end for

Create temporary array to make fusion possible
Loop Alignment

**Informal Definition:** Eliminate a carried dependence by increasing the number of iterations and executing statements on different subsets of the iterations

*(Always safe)*

```
for i=2 to N
    A[i] = B[i] + C[i]
    D[i] = A[i-1] * 2.0
end for
```

```
i = 1
D[i+1] = A[i] * 2
```

```
for i=2 to N-1
    A[i] = B[i] + C[i]
    D[i+1] = A[i] * 2.0
end for
```

```
i = N
A[i] = B[i] + C[i]
```
More details:

Optimizing Compilers for Modern Architectures

Allen and Kennedy

Academic Press
IV  Dependence Analysis and Transformations

We will analyze the Successive Over Relaxation (SOR) kernel for solving partial differential equations:

```c
  t = 0;
  for (i=1; i < N-1; i++)
    for (j=1; j < N-1; j++) {
      S2:  A[i,j] = 0.25 * t;
    }
```

19. **[5 points]**: Construct the dependence graph for the code fragment below. Mark each edge as a true, anti, or output dependence in the usual way. (Do not show input dependences.) Label each dependence edge with its direction vector. If a particular direction vector entry has multiple values (e.g., + and =), show that as multiple dependence edges with separate direction vectors.

20. **[5 points]**: Suggest a transformation that will reduce the number of dependences. Write down the transformed loop nest and its dependence graph.

Remove the temporary variable \( t \), since its value is used only once. We remove antidependence and output dependence on the temporary variable (recall the discussion in class).
Bonus
Polyhedral Compilation

Brief Introduction to Polyhedral Compilation Techniques:

Basic polyhedral concepts in program analysis
Iteration spaces; array references
Dependence analysis
Loop transformations: representation
Loop transformations: code generation
Polyhedra

**k-tuple:** A point in $\mathbb{Z}^k$, e.g., $(1, -4, 3)$ or $J = (i_1, i_2, \ldots, i_k)$

**Tuple set:** A set of tuple points $(0,1,2), (2,3) \ldots$

**Tagged tuple set:** A set of tuple points $A(1,2), C(3)$
- Can be represented as a tuple, where e.g., $\text{map}(A)=0, \text{map}(C)=2$

**Polyhedron:** A tuple set defined by affine inequalities

**General:** \[ \{(i_1, i_2, \ldots, i_k) : A \cdot \vec{i} \leq \vec{U}\} \]
- e.g. \[ \{(i_1, i_2) : L_1 \leq i_1 < U_1 \land L_2 \leq i_2 < U_2\} \]
- Focus on convex polyhedral
- Integer polyhedron: all in/out points are integers
- Integer hull: set of integer points that bounds rational polyhedron
Tuple Relations

Tuple relation (or relation or mapping:) A mapping from tuple sets to tuple sets, e.g.,
\{(i, j) \rightarrow (ii, jj) : 0 \leq i < N \land 0 \leq j < N \land ii = i \land jj = i + j - 1\}

A relation, R, “applied” to a tuple set, S, yields a new tuple set, R(S).
E.g., S = \{(i) : 0 \leq i \leq N\}, R = \{(i) \rightarrow (ii) : 0 \leq i \leq N \land ii = 2i + 1\},
results in R(S) = \{(ii) : \exists k : ii = 2k + 1 \land 1 \leq ii \leq 2N + 1\}. 
Dependence Analysis Steps

1. Extract model from the code
   - Affine iteration spaces as Polyhedra
   - Array references as polyhedral mappings

2. Dependence analysis:
   - Turn into polyhedral satisfaction problem

3. Transformations:
   - Permutations/transformations on the model, specified by tuple relations
   - Generate code from the model (original code and the transformed iteration spaces)
Affine **Iteration Spaces as Polyhedra**

for $i_1 = L_1$ to $U_1$
    $S_1$
    for $i_2 = L_2$ to $U_2$
        $S_2$
        \[ \cdots \]
        for $i_k = L_k$ to $U_k$
            $S_k$
        end for
    end for
end for

Every statement in the program has an associated iteration space, describing the enclosing loops:

\[
L = \{(i_1, i_2, \ldots, i_k) : L_1 \leq i_1 < U_1 \\
\quad \land \quad L_2 \leq i_2 < U_2 \\
\quad \land \quad L_k \leq i_k < U_k\}
\]

- For polyhedral analysis, $L_i, U_i$ must be affine functions of index variables $(i)$, loop-invariant program variables and constants.
Array References as Polyhedral Mappings

for \( i_1 = L_1 \) to \( U_1 \)
  \( S_1 \)
  for \( i_2 = L_2 \) to \( U_2 \)
    \( S_2 \)
      \ldots
      for \( i_k = L_k \) to \( U_k \)
        \( A[i_1, \ldots, i_k] = \ldots \)
      end for
    end for
  \ldots
end for

Every array reference in the program is a mapping from the iteration space (of the statement) to array elements. E.g.,

\[
L \rightarrow A : \{ (\vec{i}, \vec{a}) : \vec{i} \in L \\
\quad \land \ a_1 = f_1(\vec{i}) \ldots \\
\quad \land \ a_r = f_r(\vec{i}) \}
\]

• For polyhedral analysis, \( f_i \), must be affine functions of index variables \( (i) \), loop-invariant program variables and constants.
Checking for Data Dependence

There is a data dependence between
\[ A(f_1(\vec{i}), f_2(\vec{i}), \ldots, f_r(\vec{i})) \] and \[ A(g_1(\vec{i}), g_2(\vec{i}), \ldots, g_r(\vec{i})) \]

iff the following polyhedron contains integer points:

\[
\{(i_1, i_2, \ldots, i_r, j_1, j_2, \ldots, j_r) : \vec{i} \in L \land \vec{j} \in L \land \\
f_1(\vec{i}) = g_1(\vec{j}) \land \ldots \land f_r(\vec{i}) = g_r(\vec{j})\}
\]
Program Transformations

Program transformations as polyhedral mappings: Many program transformations can be represented as a mapping (for each original program statement) from its iteration space in the original program to its iteration space in the transformed program.

Loop reordering transformations: a transformation on a perfect loop nest that reorders the loop iteration space but does not modify the relative order of statements within the innermost loop (sometimes called an atomic block).

\[ L \rightarrow L : \{ (\bar{i}) \rightarrow (\bar{ii}) : \bar{i} \in L \wedge i_{i_1} = \varphi_1(\bar{i}) \wedge \ldots \wedge i_{i_k} = \varphi_k(\bar{i}) \} \]
Loop Transformations and Matrices

Alternate representation for loop transformations – as a matrix: 
$$\Phi(\vec{i}) = T \cdot \vec{i} + \vec{t}$$

- The transformation is affine iff $T$ is a constant matrix and $\vec{t}$ is a parametric vector consisting of loop-invariant program variables and constants.
- Each column in the matrix product represents a single input loop. Each row in the matrix product represents a single output loop.
- The transformation is called *unimodular* if $T$ is unimodular (i.e., square integer matrix with determinant $+1$ or $-1$)
Loop Transformations and Matrices

A transformation is called *unimodular* if the matrix $T$ is unimodular (i.e., square integer matrix with determinant $+1$ or $-1$)

Loop interchange: $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $\vec{t} = \vec{0}$

Loop reversal: $T = [-1]$ , $\vec{t} = (U_1 - 1)$

Legality of the transformation: $T \cdot \vec{t} > 0$
Example Transformations

Loop reversal: $\Phi = \{(i) \rightarrow (ii) : L_1 \leq i \leq U_1 \land ii = U_1 - i + 1\}$

```
    do i = L_1 to U_1
       A(i) = B(i) + C(i)
    enddo

    do ii = U_1 to L_1 by -1
       A(ii) = B(ii) + C(ii)
    enddo
```
Example Transformations

Loop reversal: \( \Phi = \{(i) \rightarrow (ii) : L_1 \leq i \leq U_1 \land ii = U_1 - i + 1\} \)

\[
\begin{align*}
do & \ i = L_1 \ to \ U_1 \\
\quad & \ A(i) = B(i) + C(i) \\
\quad & \ enddo
\end{align*}
\[
\begin{align*}
do & \ ii = U_1 \ to \ L_1 \ by \ -1 \\
\quad & \ A(ii) = B(ii) + C(ii) \\
\quad & \ enddo
\end{align*}
\]

Loop interchange: \( \Phi = \{(i, j) \rightarrow (jj, ii) : L_1 \leq i \leq U_1 \land L_2 \leq j \leq U_2 \land ii = i \land jj = j\} \)

\[
\begin{align*}
do & \ i = L_1 \ to \ U_1 \\
\quad & \ do \ j = L_2 \ to \ U_2 \\
\quad & \quad \ A(i, j) = B(i+j, i-j) + 1 \\
\quad & \ enddo \\
\quad & \ enddo
\end{align*}
\[
\begin{align*}
do & \ ii = L_1 \ to \ U_1 \\
\quad & \ do \ jj = L_2 \ to \ U_2 \\
\quad & \quad \ A(ii, jj) = B(ii+jj, ii-jj) + 1 \\
\quad & \ enddo \\
\quad & \ enddo
\end{align*}
\]
Imperfect Loop Nests

**General approach:** Add an extra ("sequencing") dimension in the iteration space to enforce ordering on individual statements:

```plaintext
for i = L_1 to U_1
    S1(i)  \hspace{1cm} L(S1) = \{(i, 0, j): L1 \leq i \leq U1 \land j = L2\}
    for j = L_2 to U_2
        S2(i,j) \hspace{1cm} L(S2) = \{(i, 1, j): L1 \leq i \leq U1 \land L2 \leq j \leq U2\}
    end for
end for
S3(i) \hspace{1cm} L(S3) = \{(i, 2, j): L1 \leq i \leq U1 \land j = U2\}
end for
```
Pros and Cons

Pros:

• Principled representation
• Fine-grained optimization and analysis using mathematical programming
• Simplify loop transformations

Cons:

• In general, NP-complete problem: boils down to Integer programming
• Memory consuming, especially for irregular nests with control flow
References

Courses/Lectures:
• Louis-Noël Pouchet course:
  http://web.cse.ohio-state.edu/~pouchet/#lectures
• Pollylabs video and written tutorials:
  http://www.pollylabs.org/education.html

Tools: GCC Graphite, URUK, Omega, Loop…

Polly (LLVM):
• Tool: http://polly.llvm.org
• Interactive playground: http://playground.pollylabs.org/