Introduction to Loop Transformations

Key Concepts:
- **Perfectly nested loops**: Loops nest in which all assignment statements occur in the body of innermost loop.
  
  ```
  DO J = 1, M
  DO I = 1, N
  YCD = YCD + A(I,J)*X(J)
  ENDDO
  ENDDO
  ```

- **Imperfectly nested loops**: Loops nest in which some assignment statements occur within some but not all loops of loop nest.
  
  ```
  DO K = 1, N
  DO J = 1, M
  DO I = 1, N
  A(I,J,K) = A(I,J,K) / X(I,J)
  ENDDO
  ENDDO
  ENDDO
  ```

Our focus for now: perfectly nested loops

CoG of lecture:
- We have seen two key transformations of perfectly nested loops:
  - Loop body enhancements: permutation and tiling.
  - There are other loop transformations that we will discuss in class.
- Powerful way of thinking of perfectly nested loop execution and transformations:
  - Loop body instances ↔ iteration space of loop
  - Loop transformation ↔ change of basis for iteration space

Iteration Space of a Perfectly Nested Loop
Each iteration of a loop nest with n loops can be viewed as an integer point in an n-dimensional space.

Iteration space of loop: all points in n-dimensional space corresponding to loop iterations

```
DO I = 1, N
DO J = 1, M
S
```

Execution order = lexicographic order on iteration space:
(0, 1) ≤ (1, 2) ≤ ... ≤ (1, M) ≤ (2, 1) ≤ (2, 2) ≤ ... ≤ (N, M)
**Loop permutation - Linear transformation on iteration space**

```
DO  I = 1, N
DO  J = 1, M
    S
```

**Locality enhancement:**
Loop permutation brings iterations that touch the same cache line "closer" together, so probability of cache hits is increased.

**Subtle issue 1:** Loop permutation may be illegal in some loop nests

```
DO  I = 2, N
DO  J = 1, M
```

Assume that array has 1's stored everywhere before loop begins. After loop permutation:

```
DO  I = 1, N
DO  J = 2, M
```

Transformed loop will produce different values (A[I,J] for example).

```
=> permutation is illegal for this loop.
```

**Subtle issue 2:** Generating code for transformed loop nest may be non-trivial.

Example: triangular loop bounds (triangular solve/Cholesky)

```
FOR I = 1, N
    FOR J = 1, I-1
        S
```

Here, inner loop bounds are functions of outer loop index. Just exchanging the two loops will not generate correct bounds.

**Question:** How do we determine when loop permutation is legal? How do we generate loop bounds for transformed loop nest?
Two problems:

(i) Are there integer solutions?
(ii) Enumerate all integer solutions.

Most problems regarding correctness of transformations and code generation can be reduced to these problems.

Intuition about systems of linear inequalities:

Equality: line (2D), plane (3D), hyperplane (> 3D)
Inequality: half-plane (2D), half-space (> 2D)

Region described by inequality is convex
(if two points are in region, all points in between them are in region)
Let us formulate correctness of loop-termination as an ILP problem.

**Invariants:** If all iterations of a loop nest are independent, then
termination is certainly legal.

This is stronger than we need, but it is a good starting point.

What does independent mean?

Let us look at dependences.

---

**Conservative Approximation:**

- Real programs: imprecise information to read for safe approximation
  - "When you are not sure whether a dependence exists, you must assume it does."

```
Example:
procedure f(X(j));
begin
  X(i) := 10;
  X(j) := 5;
end;
```

**Question:** Is there an output dependence from the first assignment to the second?

**Answer:** If i = j, there is a dependence; otherwise, not.

```
=> Unless we know from interprocedural analysis that the parameters i and j are always distinct,
we must play it safe and insert the dependence.
```

**Key notion:** Allowing a program to execute may refer to the same location (like X(i)) and X(j).

May-dependence vs must-dependence: More precise analysis may eliminate may-dependences.

---

**Dependences in loops:**

```
FOR 10 I = 1, N
  X(I) := Y(I)...
1.0 = ...X(g(I))...
```

- **Conditions for flow dependence** from iteration I_0 to I_2:
  - 1 ≤ I_0 ≤ I_2 ≤ N (write before read)
  - f(I_0) = g(I_2) (same array location)

- **Conditions for anti-dependence** from iteration I_2 to I_0:
  - 1 ≤ I_2 < I_0 ≤ N (read before write)
  - f(I_0) = g(I_2) (same array location)

- **Conditions for output dependence** from iteration I_0 to I_2:
  - 1 ≤ I_0 < I_2 ≤ N (write in program order)
  - f(I_0) = f(I_2) (same array location)

---

**Dependences in nested loops:**

```
FOR 10 I = 1, 100
  FOR 10 J = 1, 100
    X(I,J) := Y(I,J)...
10 = ...X(g(I,J), Y(I,J))...
```

- **Conditions for flow dependence** from iteration (I_0,J_0) to (I_2,J_2):
  - 1 ≤ I_0 ≤ 100
  - 1 ≤ J_0 ≤ 100
  - 1 ≤ I_2 ≤ 3
  - 1 ≤ J_2 ≤ 200
  - (I_1,J_1) ≤ (I_2,J_2)
  - f(I_1,J_1) = f(I_2,J_2)
  - g(I_1,J_1) = g(I_2,J_2)

**Result:** The above inequalities order iterations of nested loops.
Anti and output dependencies can be defined analogously.

Army subscripts are affine functions of loop variables:

dependence testing can be formulated as a set of ILP problems

ILP Formulation:

\[
\begin{align*}
1 & \leq \text{lw} < \text{lr} \leq 100 \\
2\text{lw} & = \text{lr} + 1
\end{align*}
\]

which can be written as:

\[
\begin{align*}
1 & \leq \text{lw} \\
\text{lw} & \leq \text{lr} - 1 \\
\text{lr} & \leq 100 \\
2\text{lw} & \leq 2\text{lr} + 1 \\
2\text{lr} + 1 & \leq 2\text{lw}
\end{align*}
\]

The system:

\[
\begin{bmatrix}
-1 & 0 \\
1 & -1 \\
0 & 1 \\
2 & 2
\end{bmatrix}
\begin{bmatrix}
\text{lw} \\
\text{lr}
\end{bmatrix}
\leq
\begin{bmatrix}
-1 \\
100 \\
1 \\
-1
\end{bmatrix}
\]

ILP Formulation for Nested Loops:

\[
\begin{align*}
\text{FOR I = 1, 100} \\
\text{FOR J = 1, 100} \\
\text{X(I,J)} & = \ldots \text{X}(I,J+1) \\
\end{align*}
\]

Is there a flow dependence between different iterations?

\[
\begin{align*}
1 & \leq \text{lw} \leq 100 \\
1 & \leq \text{lr} \leq 100 \\
1 & \leq \text{lw} \leq 100 \\
1 & \leq \text{lr} \leq 100 \\
(\text{lw}, \text{lr}) & < (\text{lw}', \text{lr}') \text{(lexicographic order)} \\
\text{lr} - 1 & = \text{lw} \\
\text{lr} + 1 & = \text{lw}
\end{align*}
\]

\[
(\text{lw}, \text{lr}) < (\text{lw}', \text{lr}') \text{ is equivalent to} \\
\text{lw} < \text{lr} \text{ OR } ((\text{lw} = \text{lr}) \text{ AND } (\text{lw} < \text{lr}'))
\]

We end up with two systems of inequalities:

\[
\begin{align*}
1 & \leq \text{lw} \leq 100 \\
1 & \leq \text{lr} \leq 100 \\
1 & \leq \text{lw} \leq 100 \\
1 & \leq \text{lr} \leq 100 \\
\text{lw} & < \text{lr} \\
\text{lw} + 1 & = \text{lw} \\
\text{lr} = 1 & = \text{lw} \\
\text{lr} + 1 & = \text{lw}
\end{align*}
\]

Convert lexicographic order \(<\) into integer equations/inequalities.

Dependence edata if either system has a solution.
We can actually handle fairly complex bounds involving min’s and max’s.

```
FOR I = 1, 100
FOR J = max(0, L2(I)) , min(100, L2(I))
X(I, J) = .AX(I-1, J+1)
```

```
\[ \text{max}(L1(I), L2(I)) \leq J \leq \text{min}(U1(I), U2(I)) \]
```

Min’s and max’s in loop bounds may seem weird, but actually they describe general polyhedral iteration spaces!

![Diagram of iteration space]

For a given I, the J co-ordinate of a point in the iteration space of the loop nest satisfies max(L1(I), L2(I)) ≤ J ≤ min(U1(I), U2(I))

More important cases in practice: variables in upper/lower bounds

```
FOR I = 1, N
    FOR J = 1 , N-1
        \[ \text{max}(0, L2(I)) , \text{min}(100, L2(I)) \]
```

Solutions: Treat N as though it was an unknown in system of equations.

```
1 \leq \text{Iu} \leq N
1 \leq Jw \leq N - 1
```

This is equivalent to saying if there is a solution, for any value of N.

Is there an integer solution to system \( \text{Ax} \leq b \)?

Older solution technique: Fourier-Motzkin elimination.

Inclusion: "Gaussian elimination for inequalities"

More modern techniques exist, but all known solutions require time exponential in the number of inequalities.

Anything you can do to reduce number of inequalities is good.

Equations should not be converted blindly into inequalities but handled separately.

---

Summary

Problem of determining if a dependence exists between two iterations of a perfectly nested loop can be framed as ILP problem of the form:

\[ \text{max}(L1(I), L2(I)) \leq J \leq \text{min}(U1(I), U2(I)) \]

Is there an integer solution to system \( \text{Ax} \leq b \)?

How do we solve this decision problem?
Thm: The linear Diophantine equation \( a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = c \) has integer solutions if \( \text{gcd}(a_1, a_2, \ldots, a_n) \) divides \( c \).

Proof:

We prove only the IF case by induction, the proof in the other direction is trivial.

Induction is on \((\text{min(smallest coefficient, number of variables)})\).

Base case:
- If \( \# \text{ of variables} = 1 \), then equation is \( a_1 x_1 = c \) which has integer solutions if \( a_1 \) divides \( c \).
- If \( \text{smallest coefficient} = 1 \), then \( \gcd(a_1, a_2, \ldots, a_n) = 1 \) which divides \( c \).

Wlog, assume that \( a_1 = 1 \), and observe that the equation has solutions of the form \( (\text{c - a_2 t_2 - a_3 t_3 -\ldots- a_n t_n, t_2, t_3, \ldots, t_n}) \).

Inductive case:

Suppose smallest coefficient is \( a_1 \), and let \( t = x_1 + \text{floor}(a_2/a_1) x_2 + \ldots + \text{floor}(a_n/a_1) x_n \).

In terms of this variable, the equation can be rewritten as
\[
(a_1) t + (a_2 \mod a_1) x_2 + \ldots + (a_n \mod a_1) x_n = c
\]
where we assume that all terms with zero coefficient have been deleted.

Observe that \( (1) \) has integer solutions if original equation does too.

Now \( \gcd(a_1, a_2, \ldots, a_n) = \gcd(a_1, (a_2 \mod a_1), \ldots,(a_n \mod a_1)) \).

If \( a_1 \) is the smallest co-efficient in \( (1) \), we are left with \( 1 \) variable base case. Otherwise, the size of the smallest co-efficient has decreased, so we have made progress in the induction.

It is useful to consider solution process in matrix-theoretic terms.

We can write single equation as
\[
(3 \ 5 \ 8)(x \ y \ z)^T = 6
\]

It is hard to read off solution from this, but for special matrices, it is easy.
\[
(2 \ 0 \ 1)(a \ b \ t) = 8
\]

Solution is \( a = 4, b = t \)

Key concept: column echelon form:
"lower triangular form for underdetermined systems"
Systems of Diophantine Equations:

Key idea: use integer Gaussian elimination

Example:
\[
\begin{pmatrix}
2 & 3 & 4 \\
1 & -1 & 2
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
=
\begin{pmatrix}
5 \\
5
\end{pmatrix}
\]

It is not easy to determine if this Diophantine system has solutions.

Easy special case: lower triangular matrix

\[
\begin{pmatrix}
1 & 0 & 0 \\
2 & 5 & 0
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
=
\begin{pmatrix}
5 \\
5
\end{pmatrix}
\]

\[
x = 5, \quad z = \text{arbitrary integer}
\]

Question: Can we convert general integer matrix into equivalent lower triangular system?

INTEGER GAUSSIAN ELIMINATION

Example:
\[
\begin{pmatrix}
2 & 3 & 4 \\
x & -1 & 2
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
=
\begin{pmatrix}
5 \\
5
\end{pmatrix}
\]

Integer Gaussian Elimination

Find matrices \(U_1, U_2, \ldots, U_k\) such that

\[
A*U_1*U_2*\ldots*U_k \text{ is lower triangular (say } L)\]

Solve \(Lx' = b\) (easy)

Compute \(x = (U_1*U_2*\ldots*U_k)x'\)

Proof:

\[
(A*U_1*U_2*\ldots*U_k)x' = b \\
\Rightarrow A(U_1*U_2*\ldots*U_k)x' = b \\
\Rightarrow x = (U_1*U_2*\ldots*U_k)x'
\]

Caution: Not all column operations preserve integer solutions.

\[
\begin{pmatrix}
2 & 3 \\
0 & 4
\end{pmatrix}
\begin{pmatrix}
x \\
y
\end{pmatrix}
=
\begin{pmatrix}
5 \\
5
\end{pmatrix}
\]

Solution: \(x = -8, \ y = 7\)

which has no integer solutions!

Intuition: With some column operations, recovering solution of original system requires solving lower triangular system using rationals.

Question: Can we stay purely in the integer domain?

One solution: Use only unimodular column operations

Unimodular Column Operations:

(a) Interchange two columns

Check

Let \(x, y\) satisfy first eqn.

Let \(x', y'\) satisfy second eqn.

\[
x' = y, \quad y' = x
\]

(b) Negate a column

Check

\[
x' = -x, \quad y' = y
\]

(c) Add an integer multiple of one column to another

Check

\[
x = x' + n y'
\]

Facts:

1. The three unimodular column operations
   - interchanging two columns
   - negating a column
   - adding an integer multiple of one column to another

   preserve integer solutions, as do sequences of these operations.

2. Unimodular column operations can be used to reduce a matrix \(A\) into lower triangular form.

3. A unimodular matrix has integer entries and a determinant of +1 or -1.

4. The product of two unimodular matrices is also unimodular.
Algorithm: Given a system of Diophantine equations $Ax = b$

1. Use unimodular column operations to reduce matrix $A$ to lower triangular form $L$.
2. If $Lx' = b$ has integer solutions, so does the original system.
3. If explicit form of solutions is desired, let $U$ be the product of unimodular matrices corresponding to the column operations.
   
   \[ x = U x' \]  
   where $x'$ is the solution of the system $Lx' = b$

Detail: Instead of lower triangular matrix, you should compute ‘column echelon form’ of matrix.

**Column echelon form:** Let $i_j$ be the row containing the first non-zero in column $j$.

(i) $r(i_{j+1}) > r_j$ if column $j$ is not entirely zero.

(ii) Column $(j+1)$ is zero if column $j$ is.

\[
\begin{pmatrix}
  x & 0 & 0 \\
  0 & x & x \\
  x & x & x
\end{pmatrix}
\]

is lower triangular but not column echelon.

Point: writing down the solution for this system requires additional work with the last equation (1 equation, 2 variables). This work is precisely what is required to produce the column echelon form.

Note: Even in regular Gaussian elimination, we want column echelon form rather than lower triangular form when we have under-determined systems.