Transformations and Dependencies

Organization of lecture:
- Using ILP to generate transformed code for loop permutation
- What is a dependence?
- Dependence abstractions (summarizes): distance/direction
- Computing dependence abstractions using ILP
- How to avoid calling the ILP calculator:
  - ZIV-SIV subscripts and separability
  - GCD test
  - Caching of results

Recall:
- Polyhedral algebra tools for
  - determining emptiness of convex polyhedra
  - enumerating integers in such a polyhedron.
- Central ideas:
  - reduction of matrices to echelon form by unimodular column operations,
  - Fourier-Motzkin elimination

Let us use these tools to determine (i) legality of permutation and (ii) generation of transformed code.

Code Generation for Transformed Loop Nest

Two problems: (1) Loop bounds (2) Change of variables in body

(1) New bounds:
- Original bounds: $A \cdot \mathbf{L} \leq b$ where $A$ is in echelon form
- Transformations: $\mathbf{L'} = T \cdot \mathbf{L}$
- Note: for loop permutation, $T$ is a permutation matrix
  - $T^{-1}$ is inverse of integer matrix
- So bounds on $\mathbf{L'}$ can be written as $A \cdot T^{-1} \mathbf{L'} \leq b$
- Perform Fourier-Motzkin elimination on this system of inequalities to obtain bounds on $\mathbf{L'}$.

(2) Change of variables:
- $\mathbf{L'} = T^{-1} \mathbf{L}$
- Replace old variables by new using this formula.

Example:

Fourier-Motzkin elimination
When is permutation legal?

**Position so far:** If there is a dependence between iterations, then permutation is illegal.

**DO I = 1, 100
DO J = 1, 100
X(I,J) = ... X(G(I,J))...**

Is there a flow dependence between different iterations?

\[
\begin{align*}
1 & \leq Iw, Jw, Ir, Jr \leq 100 \\
(Iw, Jw) & \prec (Ir, Jr) \\
2Ir & = 2Jr - 1 \\
Jw & = Jr - 1
\end{align*}
\]

ILP decision problem: Is there an integer in union of two convex polyhedra?

No \(\Rightarrow\) permutation is legal.

Key points:

- Loop bounds determination in transformed code is mechanical.
- Polyhedral algebra technology can handle very general bounds with max's in lower bounds and min's in upper bounds.
- No need for pattern matching etc for triangular bounds and the like.

Permuation is legal only if dependence does not exist; too simplistic.

**Example:**

**DO I = 1, 100
DO J = 1, 100
X(I,J) = ... X(G(I,J))...**

Only dependence is flow dependence:

\[
\begin{align*}
1 & \leq Iw, Jw, Ir, Jr \leq 100 \\
(Iw, Jw) & \prec (Ir, Jr) \\
Iw & = Ir - 1 \\
Jw & = Jr - 1
\end{align*}
\]

ILP problem has solution: for example \((Iw = 1, Jw = 1, Ir = 2, Jr = 2)\)

Dependence exists but loop interchange is legal!

Consider single loop case first:

**DO I = 1, 100
X(I) = ... X(G)...**

Flow dependence between iterations:

- Iteration 1 writes to X(1) which is read by iteration 3.
- Iteration 2 writes to X(2) which is read by iteration 5.
- ...
- Iteration 49 writes to X(49) which is read by iteration 99.

If we ignore the array locations and just think about dependence between iterations, we can draw this geometrically as follows:

 Dependence arrows always go forward in iteration space (eg. there cannot be a dependence from iteration 5 to iteration 2)

Point: Existence of dependence is a very "coarse" criterion to determine if interchange is legal.

Additional information about dependence may let us conclude that a transformation is legal.

To get a handle on all this, let us first define dependence precisely.
Intuitively, dependence arrows tell us constraints on transformations.

Suppose a transformed program does iteration 2 before iteration 1. OK!
Transformed program does iteration 3 before iteration 1. Illegal!

Formal view of a dependence relation between points in the iteration space:
DO I = 1, 500
X(2I+1) = ... X(2I+4)...
Flow dependence = \{(2w, 2hw + 1) | 1 \leq 2w \leq 49\}
(Note: this is a convex set)

In the spirit of dependence, we will often write this as follows:
Flow dependence = \{(w, 2hw + 1) | 1 \leq 2w \leq 49\}

2D loop nest
DO 10 I = 1,100
DO 10 J = 1,100
10 X(I,J) = X(I-1,J+4) + 1
Dependence relation of the form (I1, J1) \rightarrow (I2, J2)
Picture in iteration space:

Legal and illegal dependence arrows:

If (A \rightarrow B) is a dependence arrow, then A must be
lexicographically less than or equal to B.

Dependence relation can be computed using ILP calculator
DO 10 I = 1,100
DO 10 J = 1,100
10 X(I,J) = X(I-1, J+4) + 1
Flow dependence constraints: (Iw, Jw) \rightarrow (Iw, Jw)
1 \leq Iw, Jw, Jw, Jw \leq 100
1 \leq Iw, Jw \rightarrow (Iw, Jw)
Jw = Iw - 1
Jw = Jw + 1

Use ILP calculator to determine the following relations:
D = \{(h, Jw) \rightarrow (hw + 1, Jw - 1) | 1 \leq hw \leq 99 \land (2 \leq Jw \leq 100)\}

If we have the full dependence relation, can we determine when
permutation is legal?
Let us look at geometric picture to understand when permutation is legal.

Intuitively, if an iteration is dependent on an iteration in its "upper
left hand corner", permutation is illegal. How do we express this
formally?
Legality of permutation can be framed as an ILP problem.

10 I = 1, 100
10 J = 1, 100
10 \( X(I,J) = \mathbb{I}(I,J) \mathbb{P}(I,J) + 1 \)

Permutation is illegal if there exist iterations \((I_1, J_1), (I_2, J_2)\) in source program such that

- \((I_1, J_1) \rightarrow (I_2, J_2) \in D\) (dependent iterations)
- \((I_2, J_2) \rightarrow (I_1, J_1)\) (iterations done in wrong order in transformed program)

This can obviously be phrased as an ILP problem and solved.

One solution: \((I_1, J_1) = (1, 2), (I_2, J_2) = (3, 4)\).
Interchange is illegal.

Problems with using full dependence sets:

- Expensive (time/space) to compute full relations
- Need to solve ILP problems again to determine legality of permutation
- Symbolic loop bounds (‘\(N\)’) require parameterized sets (‘\(N\)’ is unbounded variable in definition of dependence set)

Dependence abstractions: summary of dependence set \(D\)

- less information than full set of tuples in \(D\)
- more information than non-sensitivities of \(D\)
- intuitively, “as much as is needed for transformations of interest”

Distance/Direction: Summarize dependence relation

Look at dependence relation from earlier slide
\[
\{(1,2) \rightarrow (2,3), (3,4) \rightarrow (2,2), (2,2) \rightarrow (3,4)\}
\]

Distance between dependent iterations = \((3, 1)\). That is, \((I_0, J_0) \rightarrow (I_1, J_1)\) in dependence relation, implies
\[
I_1 - I_0 = 1, 
J_1 - J_0 = -1
\]
We will say that the distance vector is \((3, 1)\).
Note: From distance vector, we can easily recover the full relation.
In this case, distance vector is an exact summary of relation.

Distance vectors are an approximation of a dependence:
(intuitively, we know the arrows but we do not know their sources.)
Example: \(D = \{(1w, 2w+1) \mid 1 \leq 1w \leq 49\}\)
Distance vectors: \(\{(2), (3), (4), \ldots, (49)\}\)
Distance vectors can obviously never be negative (if -1 was a distance vector for some dependency then an iteration \(I_1\) that depends on iteration \(I_1 + 1\) which is impossible.)

General picture

Permutation is coordinate transformation: \(L' = P \cdot L\) where \(P\) is a permutation matrix.

Conditions for legality of transformation:
For each dependence \(D\) in loop nest, check that there do not exist iterations \(L'_1\) and \(L'_2\) such that
\[
(L'_1 \rightarrow L'_2) \in D \\
P(L'_1) = P(L'_2)
\]

First condition: dependent iterations
Second condition: iterations are done in wrong order in transformed program.
Legality of permutation can be determined by solving a bunch of ILP problems.
Computing distance vectors for a dependence

\[
\begin{align*}
\text{DO } & I = 1, 100 \\
X \text{DIM} & = \ldots \text{XCD} \\
\text{Flow dependence:} \\
1 & \leq Iw < Ir \leq 100 \\
2fw+1 & = Ir \\
\text{Flow dependence } & = \{(Iw, 2fw + 1) \mid 1 \leq Iw \leq 100 \}
\end{align*}
\]

Computing distance vectors without computing dependence set:
Introduce a new variable \( \Delta = Ir - Iw \) and project onto \( \Delta \)

\[
\begin{align*}
1 & \leq Iw \leq 100 \\
2fw+1 & = Ir \\
\Delta & = Ir - Iw
\end{align*}
\]

Solution: \( \Delta = \{d \mid 2 \leq d \leq 50 \} \)

Example 2D loop nest:

\[
\begin{align*}
\text{DO } & I = 1, 100 \\
\text{DO } & J = 1, 100 \\
X \text{DIM} & = X(I+1, J+1) + 1 \\
\text{Flow dependence constraints:} \\
(Iw, Jw) & < (Ir, Jw) \\
Ir & = Iw + 1 \\
Jr & = Jw + 1 \\
\text{Distance vector:} & = (Ir - Iw, Jr - Jw) \\
\text{Solution:} & = (1, -1)
\end{align*}
\]

General approach to computing distance vectors:
Set of distance vectors generated from a dependence is itself a polyhedral set.
Computing distance vectors without computing dependence set:
To the linear system representing the existence of the dependence, add new variables corresponding to the entries in the distance vector and project onto these variables.

Reality check:
In general, dependence is some complicated convex set.
In general, distance vectors of a dependence are also some complicated convex set.

What is the point of “summarizing” one complicated set by another equally complicated set?!!!
Answer: We use distance vector summary of a dependence only when dependence can be summarized by a single distance vector (called a uniform dependence).

How do we summarize dependence when we do not have a uniform dependence? Answer: use direction vectors.

Digression: When is a dependence a uniform dependence?
That is, when can a dependence be summarized by a single distance vector?

Conjecture: subscripts are of the following form

\[
\begin{align*}
\text{DD } & I \\
\text{DD } & J \\
X(I+a, J+b) & = \ldots X(D+c, D+d) \\
\text{Check:} & \text{flow dependence equations are} \\
Iw + a & = Ir + c \\
Jw + b & = Jr + d \\
\text{So distance vector is} & = (a - c, b - d) \\
\text{Let us introduce some terminology to make the conjecture precise.}
\end{align*}
\]

ZIV-SIV-MIV Subscripts
Consider equalities for following dependence problem:

\[
\begin{align*}
\text{DD } & I \\
\text{DD } & J \\
\text{DD } & K \\
X(I+1, J+1, K) & = \ldots X(D+1, D+1, D+1) \\
\text{Subscripts in 1st dimension of A do not involve loop variables} \\
\Rightarrow & \text{subscripts called Zero Index Variable (ZIV) subscripts} \\
\text{Subscripts in 2nd dimension of A involve only one loop variable (I)} \\
\Rightarrow & \text{subscripts called Single Index Variable (SIV) subscripts} \\
\text{Subscripts in 3rd dimension of A involve many loop variables (IJK)} \\
\Rightarrow & \text{subscripts called Multiple Index Variable (MIV) subscripts}
\end{align*}
\]
Separable SIV Subscript

\begin{align*}
\text{DO } I & \\
\text{DO } J & \\
\text{DO } K & \\
A_{I,J,K} &= \ldots A_{I,J,K} + c
\end{align*}

Subscripts in both the first and second dimensions are SIV.

However, index variable in first subscript (I) does not appear in any other dimension

$\Rightarrow$ separable SIV subscripts

Second subscript is also SIV, but its index variable J appears in 3rd dimension as well

$\Rightarrow$ coupled SIV subscripts

Conjecture: Consider the flow dependence in following program

\begin{align*}
\text{DO } I & \\
\text{DO } J & \\
X_{I,J,K} &= \ldots X_{I,J,K}
\end{align*}

Conjecture: If flow dependence exists, it can be summarized by a distance vector iff each subscript is a separable SIV subscript.

This conjecture is false.

Another example:

\begin{align*}
\text{DO } I & \\
\text{DO } J & \\
X_{I,J,K} &= \ldots X_{I,J,K}
\end{align*}

Here, subscripts are MIV, and the subscripts of reads and writes look quite different.

Dependence equations:

\begin{align*}
I_o - I_o &= I_o - I_o \\
J_o + J_o &= 2J_o
\end{align*}

Easy to verify that dependence distance is (1,1).

Modified conjecture: Consider the program

\begin{align*}
\text{DO } I & \\
X_{A[I] + a} &= \ldots X_{A[I] + b}
\end{align*}

Here, I is a vector, A and B are matrices, etc.

If $A = B$, columns of $A$ (and of $B$) are linearly independent and dependence exists, then dependence is uniform dependence.

Proof: Equality system is $A + I_o = a = B + I_o = b$.

\begin{align*}
A + I_o + a &= B + I_o + b \\
B = (I_o - I_o) &= a = (a_{i=0}) \text{ (since } A = B) \\
B + \Delta &= a = (a_{i=0}) \text{ (dependence vector)}
\end{align*}

Since null space of B contains only the 0 vector, the equation has a unique solution if it has a solution at all.

Two points:

- You must check inequalities to make sure dependence actually exists.

\begin{align*}
\text{DO } I & = 1, 100 \\
X_{1,100} &= \ldots X_{1}
\end{align*}

- It is incorrect to conclude that distance vector is (100) since no dependence exists.

- As we will see later, separable SIV subscripts are very common; MIV is very rare.

End of discussion.
**Direction vectors Examples:**

**Example:**

\[ \begin{align*}
10 & \text{I } 1 = 1, 100 \\
10 & \text{XGT}\text{+1} = \text{XCD} + 1 \\
\end{align*} \]

Flow dependence equation: \( 2L_0 + 1 = L_r \).

Dependence relation: \( \{(1 \rightarrow 3), (2 \rightarrow 5), (3 \rightarrow 7), \ldots \} \).

No fixed distance between dependent iterations.

But all distances are \( >0 \), so use **direction vector** instead.

**How to calculate direction:** \( (+) \) direction = some distance in range \([1, \infty)\).

Intuition: \( (+) \) direction is **not** in any direction, but a distance.

In general, direction = \((+, 0), (\neg, \bot), \) or \((\neg, \neg)\).

Also written by some authors as \((\neg, \neg), (\neg, \bot), \) or \((\neg, \bot)\).

Direction vectors are not exact.

(eg) If we try to recover dependence relation from direction \((+),\) we get bigger relation than \((1):\)

\( \{(1 \rightarrow 2), (1 \rightarrow 3), \ldots (1 \rightarrow 100), (2 \rightarrow 3), (2 \rightarrow 4), \ldots \} \)

**Directions for Nested Loops**

Assume loop nest is \((L_1, L_2)\).

If \((I_1, I_2) \rightarrow \{(L_1, L_2)\}\) is dependence relation, then

Distance = \((L_2 - I_2, L_2 - I_1)\)

Direction = \((\text{sign}(L_2 - I_1), \text{sign}(L_2 - I_1))\)

**How to compute Directions:** Use IP engine

**Example:**

\[ \begin{align*}
10 & \text{I } 1 = 1, 100 \\
10 & \text{XCD} = \ldots \\
10 & = \ldots \text{XCD} \ldots \\
\end{align*} \]

Focus on flow dependence:

\( f(L_0) = g(L_r) \)

\( 1 \leq L_0 \leq 100 \)

\( 1 \leq L_r \leq 100 \)

First, use inequalities above to test if dependence exists in any direction (called \((\neg)\) direction).

If IP engine says there are no solutions, no dependence.

Otherwise, determine the direction(s) of dependence.

Test for direction \((+, 0)\): add inequality \( L_0 < L_r \)

Test for direction \((0, \neg)\): add inequality \( L_0 = L_r \)

In a single loop, direction \((\neg, \neg)\) cannot occur.

**Computing Directions:** Nested Loops

Same idea as single loop: **hierarchical testing**

- Figure 1: Hierarchical Testing for Nested Loop

**Key ideas:**

1. Define direction vectors top-down.

(eg) No dependence in \((+, 0)\) direction

\[ \Rightarrow \] no need to do more tests.

2. Do not test for impossible directions like \((\neg, 0)\).

**Big hairy example:** Compute dependences for following program

\[ \begin{align*}
10 & \text{I } 1, N \\
10 & \text{J } 1, N \\
10 & \text{XCD} = \ldots \text{XCD} \ldots \\
\end{align*} \]

It is also possible to compute direction vectors by projecting on the variables in the \( \Delta \), the loop difference vector.

Similar to what we did for distance vectors.

Left as an exercise for you.
Linear system for anti-dependence:

\[ \begin{align*}
I_u &= I_i \\
J_u &= J_i \\
1 &\leq I_u, I_i, J_u, J_i \leq N \\
(I_u, J_u) &\leq (I_i, J_i) \\
\Delta 1 &= (I_u - I_i) \\
\Delta 2 &= (J_u - J_i)
\end{align*} \]

Projecting onto \( \Delta 1 \) and \( \Delta 2 \), we get

\[ \begin{align*}
\Delta 1 &= 0 \\
0 &\leq \Delta 2 \leq (N - 1)
\end{align*} \]

So directions for anti-dependence are

\[ \begin{align*}
0 &\quad \text{and} \quad 0 \\
0 &\quad +
\end{align*} \]

Similarly, you can compute direction for flow dependence

\[ 0 \]

and also show that no output dependence exists.

Dependence matrix for a loop nest

Matrix containing all dependence distance/direction vectors for all dependences of loop nest.

In our example, the dependence matrix is

\[ \begin{align*}
0 &\quad 0 \\
0 &\quad +
\end{align*} \]

Correctness of general permutation

Transformation matrix: \( T \)

Dependence matrix: \( D \)

Matrix in which each column is a distance/direction vector

Legality: \( T \cdot D > 0 \)

Dependence matrix of transformed program: \( T \cdot D \)
Examples:

DO I = 1,N
    DO J = 1,N
        X(I,J) = X(I,J-1)+D...
    END DO J
END DO I

Distance vector = (1,1) => permutation is legal
Dependence vector of transformed program = (1,1)

DO I = 1,N
    DO J = 1,N
        X(I,J) = X(I,J-1)+D...
    END DO J
END DO I

Distance vector = (1,1) => permutation is not legal

Remarks on dependence abstractions

A good dependence abstraction for a transformation should have the following properties:

- Easy to compute
- Easy to test for legality
- Easy to determine dependence abstractions for transformed program

Direction vectors are a good dependence abstraction for permutation.

Engineering a dependence analyzer

In principle, we can use IP engine to compute all directions.

Reality: most subscripts and loop bounds are simple!

Engineering a dependence analyzer:

First check for simple cases.

Call IP engine for more complex cases.

Important optimization: splitting of linear systems

In practice, many dependence computations can be decomposed into two or more smaller, independent problems.

DO 10 I
    DO 10 J
    DO 10 K
    A(I,J,K) = ...A(I,J,K) + c

I occurs only in first subscript and bounds on I are independent of other variables => inequalities/equalities for (I,J,K) for example can be separated from rest of system and solved separately.

Separable SIV subscript: Simple, precise tests exist.

DO 10 J
    DO 10 I
    DO 10 K
    X(aI + b,...) =...X(cI + d,...)

Equation for flow dependence: a*I_0 + b = c*I_r + d.

Strong SIV subscript: a = c

=> I_r = I_0 = (b-d)/a

If a divides (b-d), and quotient is within loop bounds of I_0, there is a dependence, and we have Ith component of the direction/distance vector.

Otherwise, no need to check other dimensions - no dependence exists!

In benchmarks, roughly 37% of subscripts are strong SIV!
Another important case:

\[ \text{DD 10 I} \]
\[ 10 \ X(0I + b, \ldots, d) = cX(cI + d, \ldots) \ldots \]

Weak SIV subscript: Either \( a \) or \( c \) is 0.

Say \( c = 0 \Rightarrow I_a = (d-b)/a \) and \( I_r > I_w \). 

If \( a \) divides \((d-b)/a\) and quotient is within loop bounds, then dependence exists with all iterations beyond \( I_a \).

Important loop transformation: Index set splitting

It may be worth eliminating dependence by performing iterations \( I_n/(d-b)/a = 1 \) in one loop, iteration \((d-b)/a\) by itself and then the remaining iterations in another loop.

General SIV Test. Equation: \( a \cdot I_w + b = c \cdot I_r + d \) (1)

We can use column operations to reduce to echelon form etc.

But usually, \( a \) and \( c \) are small integers (\( \text{mag} < 5 \)). Exploit this.

Build a table indexed by \((a, c)\) pairs for \( a \) and \( c \) between 1 and 5.

Two entries in each table position:

(i) \( \gcd(a, c) \)

(ii) one solution \((I_w, I_r) = (s, t)\) to eqn \( a \cdot I_w + c = \gcd(a, c) \).

Given Equation (1), if \( a \) and \( c \) are between 1 and 5,

(i) if \( \gcd(a, c) \) does not divide \((d-b)/a\), no solution.

(ii) otherwise, one solution is \((s, t) = (d-b)/\gcd(a, c)\).

(iii) General solution:

\[ (I_w, I_r) = n \cdot (s, t) + (d-b)/\gcd(a, c) \]

\( n \) is parameter.

Case when \( a \) or \( c \) in Equation (1) are \( -c \): minor modification of this procedure.

Implementation notes:

(I) Check for ZIV/separable SIV first before calling IP engine.

(II) In hierarchical testing for directions, solution to equalities should be done only once.

(III) Output of equality solver may be useful to determine distances and to eliminate some directions from consideration.

(e.g.) \( \text{DD 10 I} \)
\[ \text{DD 10 J} \]
\[ A(J) = A(J+1) + 1 \]

Flow dependence, \( I_w = I_r + 1 \Rightarrow \text{distance}(J) = -1 \)

Direction vector cannot be \((0,-1)\). So only possibility is \((1,-1)\): test only for this.

(VI) Negative loop step sizes: Loop normalization

\( \text{DD 10 I} = 10, \ldots, 1 \)
\[ \text{10 } \ldots \]

If we use \( I \) to index into iteration space, dependence distances become negative!

Solution: Use trip counts \((0, I, \ldots)\) to index loop iterations.

\[ \text{DD 10 I} = 1, \ldots, S \]
\[ x(O) = X(2I+S) \ldots \]

Flow dependence, from trip \( n_w \) to \( n_r \):

\[ I + n_w \cdot s = 2I + n_r \cdot s \Rightarrow I \]

Distance vector: \([n_r - n_w]\)

Loop normalization: Transform all loops so low index is 0 and step size is 1. We are doing it implicitly.

(VII) Imperfectly nested loops

Distance/direction not adequate for imperfectly nested loops.

Imperfectly nested loops: triangular solve/Cholesky/LU

\( \text{DD 10 I} = 1, \ldots, N \)
\[ \text{DD 20 J = 1, I = 1} \]
\[ 20 X(J) = B(J) + L(J, I) \cdot X(J) \]
\[ 10 X(O) = B(D)/L(G, I) \]

What is the analog of distance/direction vectors for imperfectly nested loops?
One approach: Compute distance/direction only for common loops.
Not adequate for many applications like imperfect loop interchange.

```fortran
C row triangular solve
DO 10 I = 1,N
   DO 20 J = 1, I-1
      20 BCD = B(I) - LG(J) * XCD
   10 X(I) = BCD / LG(I)
```

```fortran
C column triangular solve
DO 10 I = 1,N
   X(I) = BCD / LG(I)
   DO 20 J = I+1, N
      20 BCD = B(J) - LG(J) * XCD
```

What is a good dependence abstraction for imperfectly nested loops?
Some tests for a good dependence abstraction for imperfectly nested loops:
- Easy to see that both versions of triangular solve are legal
- Easy to see that all six versions of Cholesky factorization are legal
- Easy to determine dependence abstraction for transformed program

Conclusions
Traditional position: exact dependence testing (using IP engine) is too expensive.
Recent experience:
(i) exact dependence testing is OK provided we check for easy cases (2IV, strong SIV, weak SIV)
(ii) IP engine is called for 3-4% of tests for direction vectors
(iii) Cost of exact dependence testing: 3-6% of compile time