Dependences and Transformations

Overview

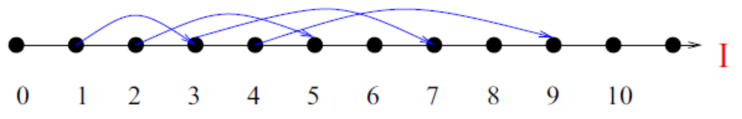
- Dependence
  - Binary relation between iteration space points
  - Can be computed using ILP calculator
- Dependence abstractions
  - Distance vectors
  - Direction vectors
  - Dependence matrix
- Unimodular transformations
  - Permutation, skewing, reversal and compositions
  - Representation using unimodular matrices
- Synthesizing unimodular transformations
  - Converting a loop into a fully permutable loop nest

Dependence

Consider single loop case first:

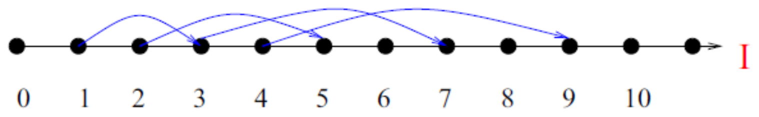
```
DO I = 1, 100
  X(2I+1) = ....X(I)...


Flow dependences between iterations:
   Iteration 1 writes to X(3) which is read by iteration 3.
   Iteration 2 writes to X(5) which is read by iteration 5.
   ....
   Iteration 49 writes to X(99) 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)

Intuitively, dependence arrows tell us constraints on transformations.



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

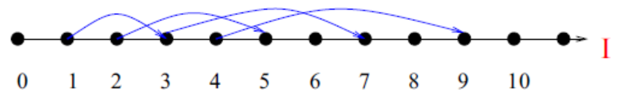
Transformed program does iteration 3 before iteration 1. Illegal!

Formal view of dependence

Formal view of a dependence: relation between points in the iteration space.

```
DO I = 1, 100
  X(2I+1) = ....X(I)...
```

Flow dependence $= \{(Iw, 2Iw + 1) | 1 \leq Iw \leq 49\}$
(Note: this is a convex set)



In the spirit of dependence, we will often write this as follows:

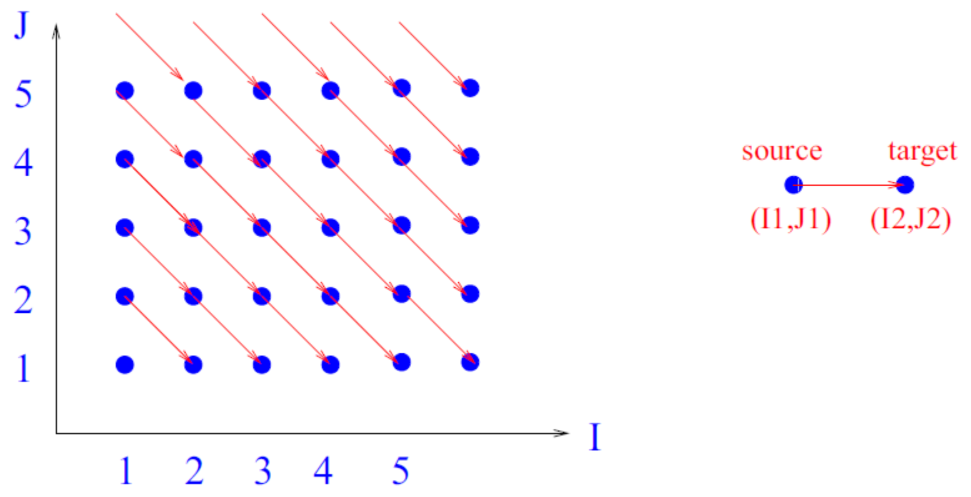
Flow dependence $= \{(Iw \rightarrow 2Iw + 1) | 1 \leq Iw \leq 49\}$

2D loop nest

```
DO 10 I = 1,100
  DO 10 J = 1,100
  10 X(I,J) = X(I-1,J+1) + 1
```

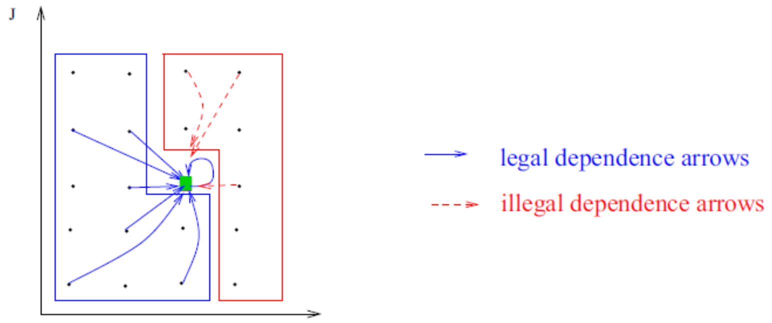
Dependence: relation of the form $(I_1, J_1) \rightarrow (I_2, J_2)$.

Picture in iteration space:



Dependence arrows are lexicographically positive

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+1) + 1
```

Flow dependence constraints: $(I_w, J_w) \rightarrow (I_r, J_r)$

- $1 \leq Iw, Ir, Jw, Jr \leq 100$
- $(I_w, J_w) \prec (I_r, J_r)$
- $I_w = I_r - 1$
- $J_w = J_r + 1$

Use ILP calculator to determine the following relation:

$$D = \{(Iw, Jw) \rightarrow (Iw + 1, Jw - 1) | (1 \leq Iw \leq 99) \wedge (2 \leq Jw \leq 100)\}$$

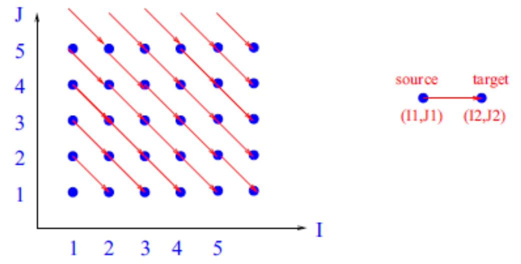
Dependence abstractions

- In practice, working with the full dependence relation for a loop nest is expensive and difficult
- Usually, we use an abstraction of dependence relation
  - Summary information about dependence
  - Summary is an over-approximation of actual dependence relation
- Two abstractions are popular
  - Distance vectors
  - Direction vectors
  - Dependence matrix: collection of distance/direction vectors

Distance/direction: Summarize dependence relation

Look at dependence relation from earlier slides:

$$\{(1,2) \rightarrow (2,1), (1,3) \rightarrow (2,2), ..(2,2) \rightarrow (3,1)...\}$$



Difference between dependent iterations $= (1, -1)$. That is,

$$(I_w, J_w) \rightarrow (I_r, J_r) \in \text{dependence relation, implies}$$

$$I_r - I_w = 1$$

$$J_r - J_w = -1$$

We will say that the *distance vector* is $(1, -1)$.

Note: From distance vector, we can easily recover the full relation.

In this case, distance vector is an *exact* summary of relation.

Computing distance vectors for a dependence

```
DO I = 1, 100
  X(2I+1) = ....X(I)...
```

Flow dependence:

$$\begin{aligned} 1 &\leq Iw < Ir \leq 100 \\ 2Iw + 1 &= Ir \end{aligned}$$

Flow dependence $= \{(Iw, 2Iw + 1) | 1 \leq Iw \leq 49\}$

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

$$\begin{aligned} 1 &\leq Iw < Ir \leq 100 \\ 2Iw + 1 &= Ir \\ \Delta &= Ir - Iw \end{aligned}$$

Solution: $\Delta = \{d | 2 \leq d \leq 50\}$

Example:2D loop nest

```
DO 10 I = 1,100
  DO 10 J = 1,100
  10 X(I,J) = X(I-1,J+1) + 1
```

Flow dependence constraints: $(I_w, J_w) \rightarrow (I_r, J_r)$
Distance vector: $(\Delta_1, \Delta_2) = (I_r - I_w, J_r - J_w)$

- $1 \leq Iw, Ir, Jw, Jr \leq 100$
- $(I_w, J_w) \prec (I_r, J_r)$
- $I_w = I_r - 1$
- $J_w = J_r + 1$
- $(\Delta_1, \Delta_2) = (I_r - I_w, J_r - J_w)$

Solution: $(\Delta_1, \Delta_2) = (1, -1)$

Direction vectors **Example:**

```
DO 10 I = 1,100
 10 X(2I+1) = X(I) + 1
```

Flow dependence equation: $2I_w + 1 = I_r$.

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

No fixed distance between dependent iterations!

But all distances are +ve, so use *direction vector* instead.

Here, direction = (+).

Intuition: (+) direction = some distances in range $[1, \infty)$

In general, direction = (+) or (0) or (-).

Also written by some authors as (<), (=), or (>).

Direction vectors are not exact.

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

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

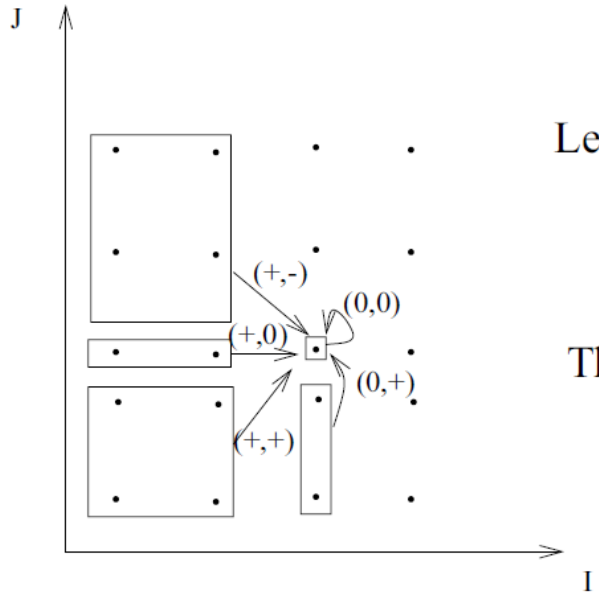
Directions for Nested Loops

Assume loop nest is (I,J).

If $(I_1, J_1) \rightarrow (I_2, J_2) \in$ dependence relation, then

$$\text{Distance} = (I_2 - I_1, J_2 - J_1)$$

$$\text{Direction} = (sign(I_2 - I_1), sign(J_2 - J_1))$$



Legal direction vectors:

(+,+) (0,+)

(+,-) (0,0)

(+,0)

The following direction vectors cannot exist:

(0,-) (-,+)

(-,0)

(-,-)

Valid dependence vectors are lexicographically positive.

How to compute Directions: Use IP engine

```
    DO 10 I = 1, 100
      X(f(I)) = ...
   10         = ...X(g(I))..
```

Focus on flow dependences:

$f(I_w) = g(I_r)$

$1 \leq I_w \leq 100$

$1 \leq I_r \leq 100$

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

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

Otherwise, determine the direction(s) of dependence.

Test for direction (+): add inequality $I_w < I_r$

Test for direction (0): add inequality $I_w = I_r$

In a single loop, direction (−) cannot occur.

Computing Directions: Nested Loops

Same idea as single loop: *hierarchical testing*

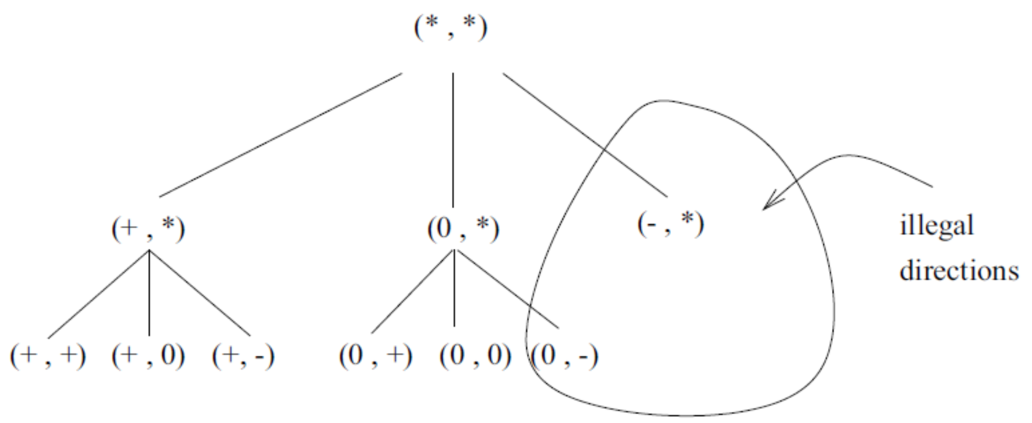


Figure 1: Hierarchical Testing for Nested Loop

Key ideas:

(1) Refine direction vectors top down.

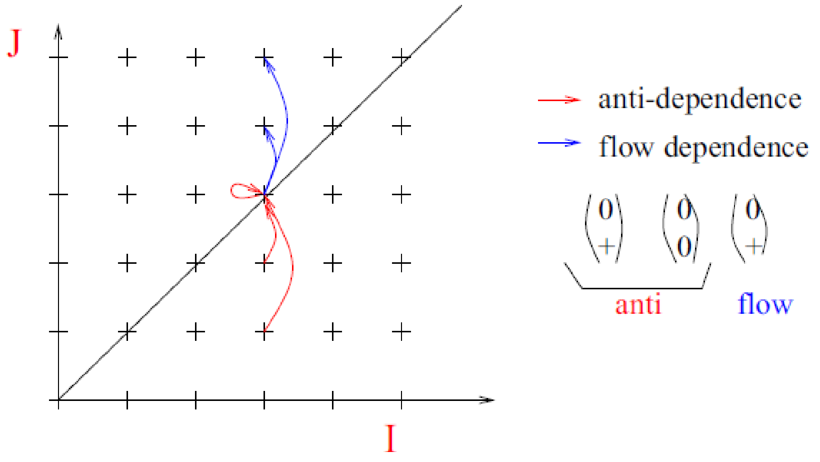
(eg),no dependence in $(*, *)$ direction

$\Rightarrow$ no need to do more tests.

(2) Do not test for impossible directions like $(-, *)$.

Example

```
DO I = 1,N
  DO J = 1,N
    X(I,J) = ...X(I,I)...
```



Linear system for anti-dependence:

$$I_w = I_r$$

$$J_w = I_r$$

$$1 \leq I_w, I_r, J_w, J_r \leq N$$

$$(I_r, J_r) \preceq (I_w, J_w)$$

$$\Delta 1 = (I_w - I_r)$$

$$\Delta 2 = (J_w - J_r)$$

Projecting onto $\Delta 1$ and $\Delta 2$, we get

$$\Delta 1 = 0$$

$$0 \leq \Delta 2 \leq (N-1)$$

So directions for anti-dependence are

```
 0    and  0
 0         +
```

Dependence matrix

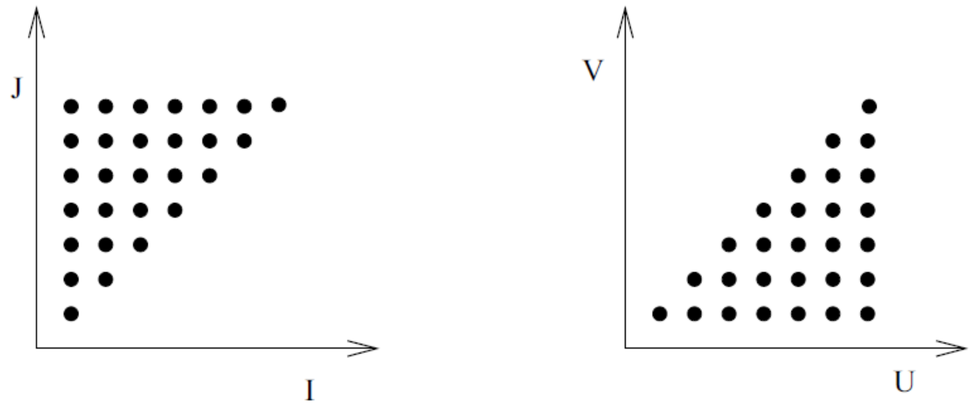
Dependence matrix for a loop nest

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

In our example, the dependence matrix is

$$\begin{pmatrix} 0 & 0 \\ 0 & + \end{pmatrix}$$

Using dependence matrices



```
DO I = 1, N
  DO J = I,N
  ..........
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

```
DO U = 1, N
  DO V = 1,U
  .........
```

$$\begin{bmatrix} I1 \\ J1 \end{bmatrix} \rightarrow \begin{bmatrix} I2 \\ J2 \end{bmatrix} \qquad T\begin{bmatrix} I1 \\ J1 \end{bmatrix} \qquad T\begin{bmatrix} I2 \\ J2 \end{bmatrix}$$

Dependence distance $= \begin{bmatrix} I2 - I1 \\ J2 - J1 \end{bmatrix}$

Distance between iterations =

$$T\begin{bmatrix} I2 \\ J2 \end{bmatrix} - T\begin{bmatrix} I1 \\ J1 \end{bmatrix} = T\begin{bmatrix} I2 - I1 \\ J2 - J1 \end{bmatrix} = \begin{bmatrix} J2 - J1 \\ I2 - I1 \end{bmatrix}$$

Check for legality: interchange positions in distance/direction vector & check for lex +ve

If transformation P is legal and original dependence matrix is D, new dependence matrix is T*D.

Using dependence matrices

Correctness of general permutation

Transformation matrix: T

Dependence matrix: D

Matrix in which each column is a distance/direction vector

Legality: $T.D \succ 0$

Dependence matrix of transformed program: $T.D$

Conclusions

Traditional position: exact dependence testing (using IP engine) is too expensive

Recent experience:

(i) exact dependence testing is OK provided we first check for easy cases (ZIV,strong SIV, weak SIV)

(ii) IP engine is called for 3-4% of tests for direction vectors

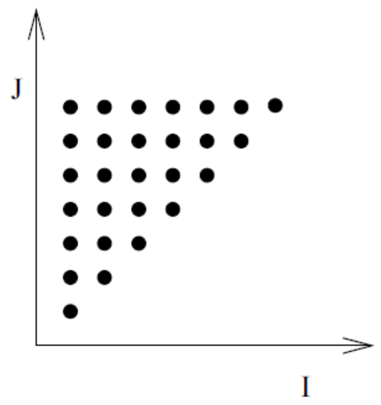
(iii) Cost of exact dependence testing: 3-5% of compile time

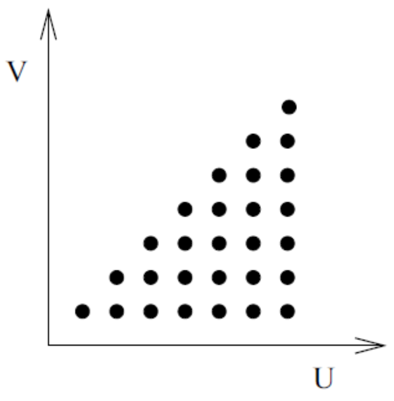
Unimodular transformations

Overview

- Unimodular transformations
  - Can be represented by unimodular matrix
    - Integer matrix with determinant of 1 or -1
    - Integer equivalent of orthogonal matrix in numerical linear algebra
  - Permutation, skewing, reversal
  - Compositions of these transformations
- Synthesizing unimodular transformations for locality
  - Making a loop nest fully permutable to enable tiling

Loop permutation





```
DO  I =  1, N
   DO J = I,N
   ..........
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

```
DO U = 1, N
   DO V = 1,U
   .........
```

$$\begin{bmatrix} I1 \\ J1 \end{bmatrix} \rightarrow \begin{bmatrix} I2 \\ J2 \end{bmatrix} \qquad T\begin{bmatrix} I1 \\ J1 \end{bmatrix} \qquad T\begin{bmatrix} I2 \\ J2 \end{bmatrix}$$

Dependence distance $= \begin{bmatrix} I2 - I1 \\ J2 - J1 \end{bmatrix}$

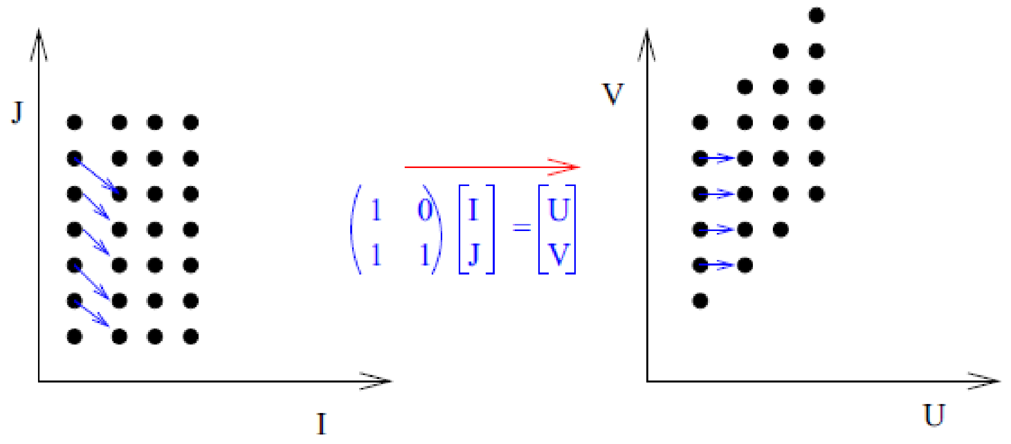
Distance between iterations =

$$T\begin{bmatrix} I2 \\ J2 \end{bmatrix} - T\begin{bmatrix} I1 \\ J1 \end{bmatrix} = T\begin{bmatrix} I2 - I1 \\ J2 - J1 \end{bmatrix} = \begin{bmatrix} J2 - J1 \\ I2 - I1 \end{bmatrix}$$

Check for legality: interchange positions in distance/direction vector & check for lex +ve

If transformation P is legal and original dependence matrix is D, new dependence matrix is T*D.

Loop Skewing: a linear loop transformation



$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

Skewing of inner loop by outer loop: $\begin{pmatrix} 1 & 0 \\ k & 1 \end{pmatrix}$ (k is some fixed integer)

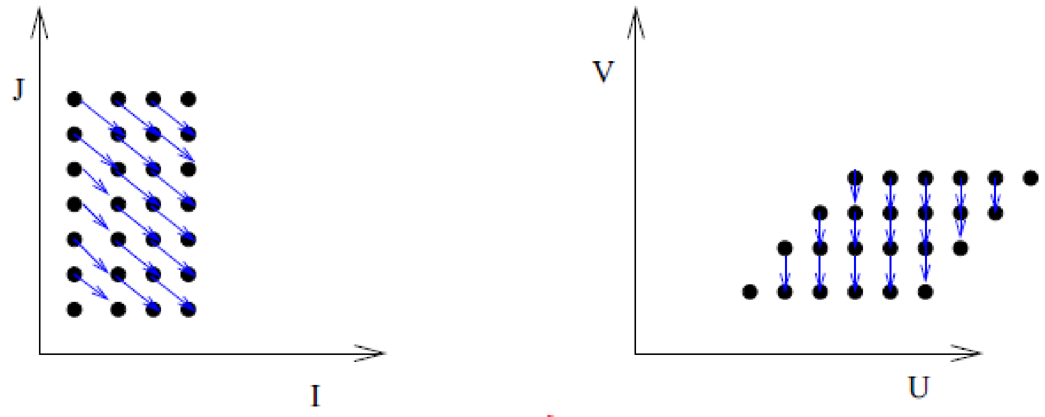
Skewing of inner loop by an outer loop: always legal

New dependence vectors: compute T*D

In this example, $D = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ $T*D = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

This skewing has changed dependence vector but it has not brought dependent iterations closer together....

Skewing outer loop by inner loop



$$\xrightarrow{\hphantom{xxxx}}$$

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

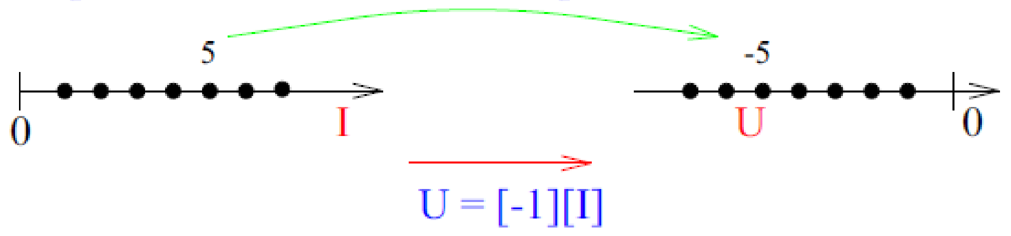
Outer loop skewing: $\begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix}$

Skewing of outer loop by inner loop: not necessarily legal

In this example, $D = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ $T*D = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$ incorrect

Dependent iterations are closer together (good) but program is illegal (bad).
How do we fix this??

Loop Reversal:a linear loop transformation



```
DO  I =  1, N
  X(I) = I+2
```

```
DO U =  -N,-1
  X(-U) = -U +2
```

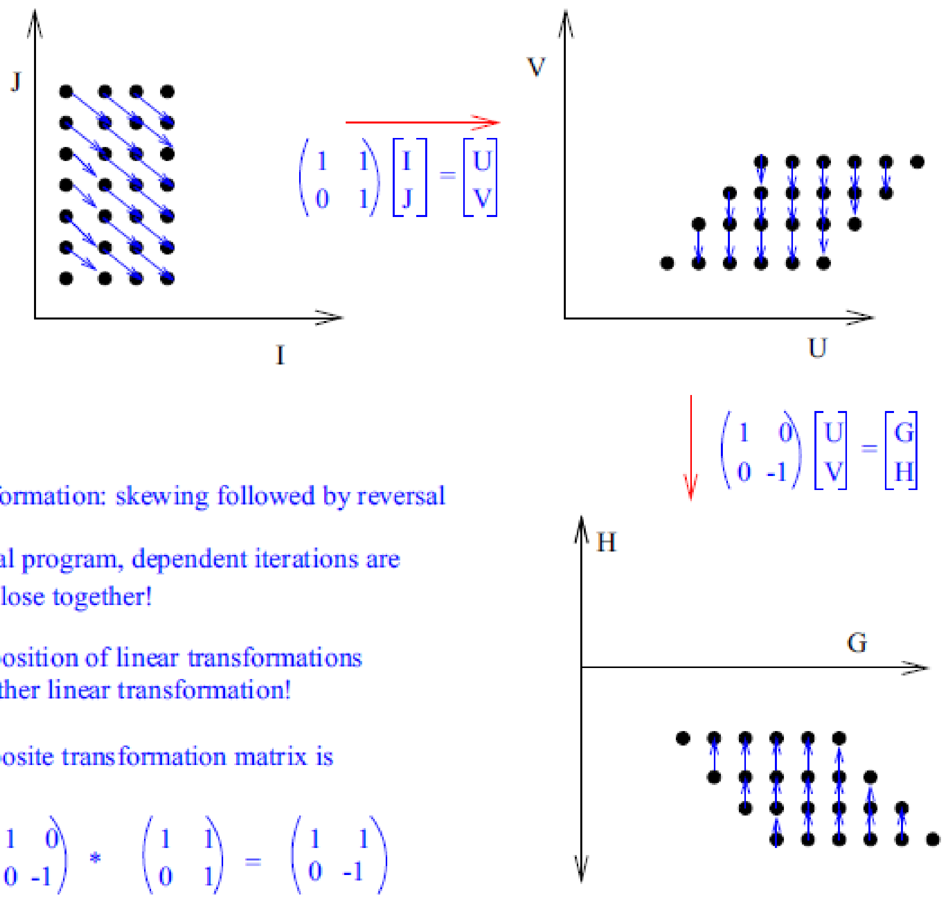
Transformation matrix = [-1]

Another example: 2-D loop, reverse inner loop $\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix}$

Legality of loop reversal: Apply transformation matrix to all dependences & verify lex +ve

Code generation: easy

Need for composite transformations



$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} G \\ H \end{bmatrix}$$

Transformation: skewing followed by reversal

In final program, dependent iterations are close together!

Composition of linear transformations = another linear transformation!

Composite transformation matrix is

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}$$

How do we synthesize this composite transformation??

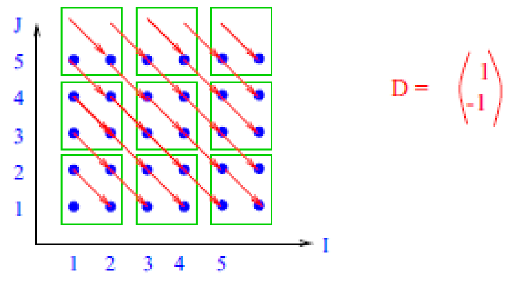
Some facts about permutation/reversal/skewing

- Transformation matrices for permutation/reversal/skewing are unimodular.
- Any composition of these transformations can be represented by a unimodular matrix.
- Any unimodular matrix can be decomposed into product of permutation/reversal/skewing matrices.
- Legality of composite transformation T: check that $T.D \succ 0$. (Proof: $T_3 * (T_2 * (T_1 * D)) = (T_3 * T_2 * T_1) * D$.)
- Code generation algorithm:
  - Original bounds: $A * \underline{I} \leq b$
  - Transformation: $\underline{U} = T * \underline{I}$
  - New bounds: compute from $A * T^{-1}\underline{U} \leq b$

Synthesizing composite transformations using matrix-based approaches

- Rather than reason about sequences of transformations, we can reason about the single matrix that represents the composite transformation.
- Enabling abstraction: dependence matrix

In general, tiling is not legal.



Tiling is illegal!

Tiling is legal if loops are fully permutable (all permutations of loops are legal).

Tiling is legal if all entries in dependence matrix are non-negative.

- Can we always convert a perfectly nested loop into a fully permutable loop nest?
- When we can, how do we do it?

Theorem: If all dependence vectors are distance vectors, we can convert entire loop nest into a fully permutable loop nest.

Example: wavefront

Dependence matrix is $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

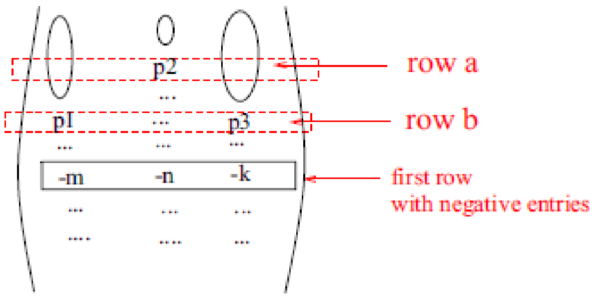
Dependence matrix of transformed program must have all positive entries.

So first row of transformation can be (1 0).

Second row of transformation (m 1) (for any $m > 0$).

General idea: skew inner loops by outer loops sufficiently to make all negative entries non-negative.

Transformation to make first row with negative entries into row with non-negative entries



(a) for each negative entry in the first row with negative entries,
find the first positive number in the corresponding column
assume the rows for these positive entries are a,b etc as shown above

(b) skew the row with negative entries by appropriate multiples of
rows a,b....
For our example, multiple of row a = ceiling(n/p2)
multiple of row b = ceiling(max(m/p1,k/p3))

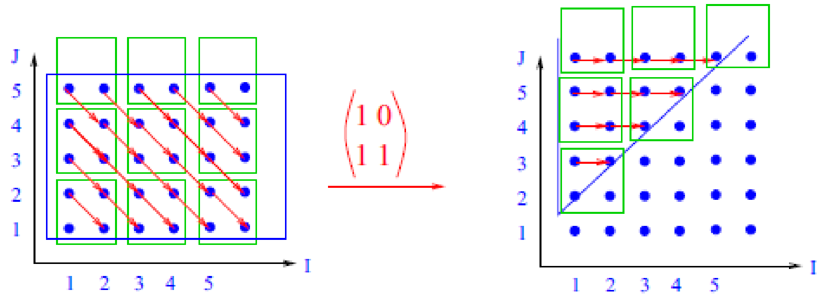
Transformation:
$$\begin{pmatrix} I \\ 0\ 0\ ..0\ \text{ceiling}(n/p2)\ 0\ 0\ \text{ceiling}(\max(m/p1,k/p3))0...0 \\ I \end{pmatrix}$$

General algorithm for making loop nest fully permutable:

If all entries in dependence matrix are non-negative, done.
Otherwise,

1. Apply algorithm on previous slide to first row with non-negative entries.
2. Generate new dependence matrix.
3. If no negative entries, done.
4. Otherwise, go step (1).

Result of tiling transformed wavefront



Original loop Tiled fully permutable loop

Tiling generates a 4-deep loop nest.

Not as nice as height reduction solution, but it will work fine for locality enhancement except at tile boundaries (but boundary points small compared to number of interior points).

What happens with direction vectors?

In general, we cannot make loop nest fully permutable.

Example: $D = \begin{pmatrix} + \\ - \\ + \end{pmatrix}$

Best we can do is to make some of the loops fully permutable.

We try to make outermost loops fully permutable, so we would interchange the second and third loops, and then tile the first two loops only.

Summary

- Dependence relation
  - Binary relation between points in iteration space
  - Can be computed using ILP calculator
- Dependence abstractions
  - Summary of dependence relation
    - Not as accurate but easier to compute and use
  - Distance/direction vectors
    - Put them together in dependence matrix
- Unimodular transformations
  - Can be represented using unimodular matrix
    - permutation, skewing, reversal, compositions of these
  - Synthesize unimodular transformations using dependence matrix as driver
    - Making a loop nest fully permutable