Data Dependence in Arrays

15-411/15-611 Compiler Design

Seth Copen Goldstein

April 9, 2026

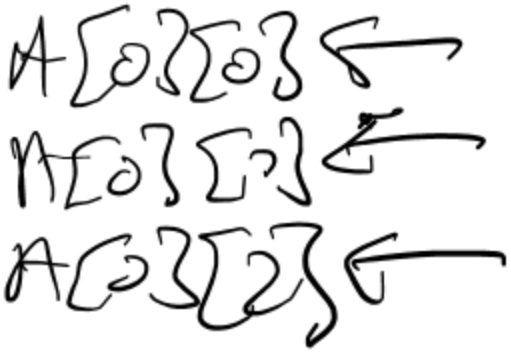
Common loop optimizations

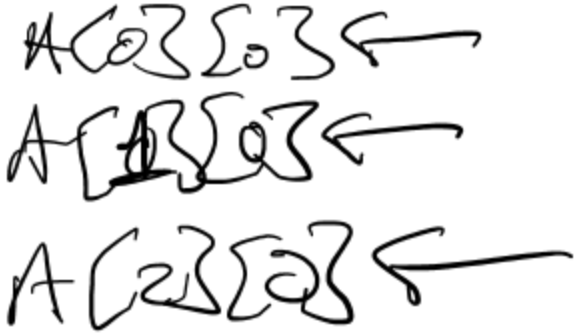
- Hoisting of loop-invariant computations
  - pre-compute before entering the loop
- Elimination of induction variables
  - change p=i*w+b to p=b,p+=w, when w,b invariant
- Loop unrolling
  - to to improve scheduling of the loop body
- Software pipelining
  - To improve scheduling of the loop body
- **Loop permutation**
  - **to improve cache memory performance**
- **Parallelization/Vectorization**

Requires understanding data dependencies

```
for (j=0; j<M; j++) {
    for (i=0; i<N; i++) {
        A[i][j] = A[i][j]+C;
    }
}
```

- Is there an opportunity for optimization?





Loop Interchange

```
for (j=0; j<M; j++) {
    for (i=0; i<N; i++) {
      A[i][j] = A[i][j]+C;
    }
}
```

- Can we swap inner and outer loop?
- Is there an advantage?

```
for (i=0; i<N; i++) {
    A[i] = A[i]+C;
}
```

- Is there an opportunity for optimization?

Loop Parallelization

```
for (i=0; i<N; i++) {
    A[i] = A[i]+C;
}
```

- No dependencies,
  can vectorize or parallelize

Data-Dependent Loop Transformations

- Goals:
  - Improving Locality
  - Automatic Vectorization
- Key Ideas:
  - Locality
  - Iteration Spaces
  - Data Dependence
  - Unimodular Transformations
  - Other Transformations

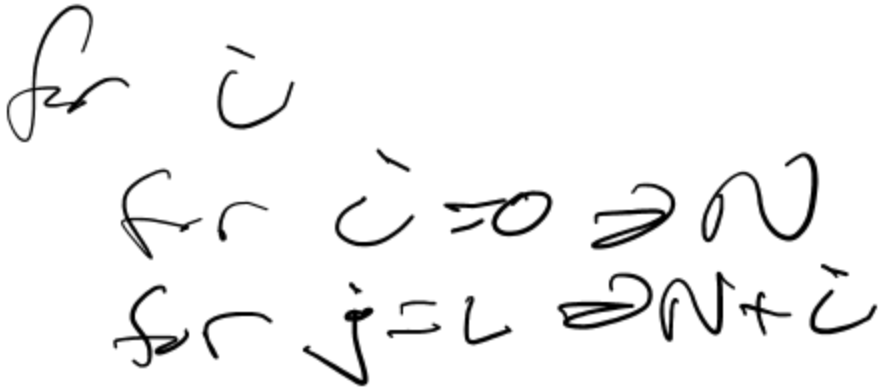
Loop Transformation Theory

Example Loop Transformations

- Loop Interchange
- Cache Blocking
- Skewing
- Loop Reversal
- …

Can improve locality

Can enable above



Dependencies in Loops

- Loop independent data dependence occurs between accesses in the same loop iteration.
- Loop-carried data dependence occurs between accesses across different loop iterations.
- There is data dependence between

  access a at iteration i-k and

  access b at iteration i when:
  - a and b access the same memory location
  - There is a path from a to b
  - Either a or b is a write

Defining Dependencies

- Flow Dependence
- Anti-Dependence
- Output Dependence

W → R δ^f

R → W δ^a

W → W δ^o

true

false

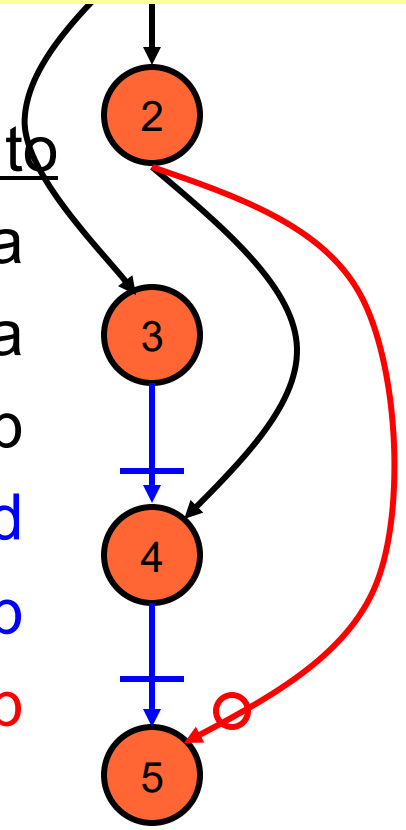
```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

Example Dependencies

```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

These are scalar dependencies. The same idea holds for memory accesses.

source	type	target	due to
S1	δ^f	S2	a
S1	δ^f	S3	a
S2	δ^f	S4	b
S3	δ^a	S4	d
S4	δ^a	S5	b
S2	δ^o	S5	b

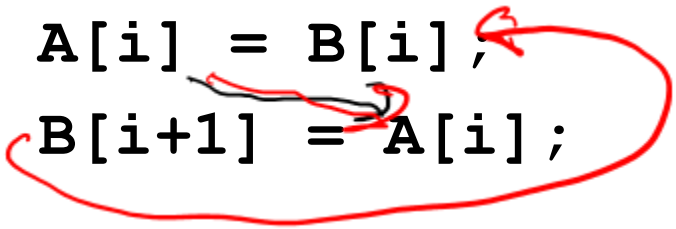


What can we do with this information?
What are anti- and flow- called “false” dependences?

Data Dependence in Loops

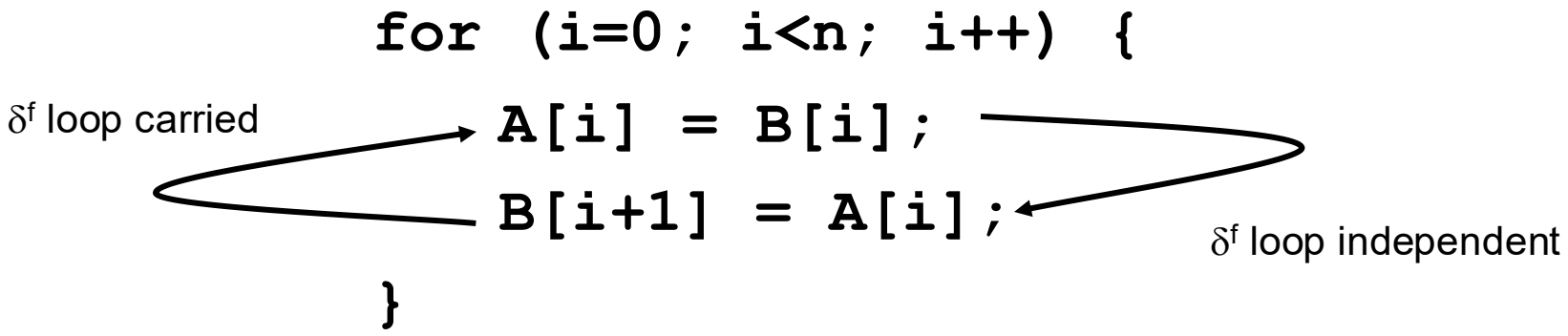
- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```



Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

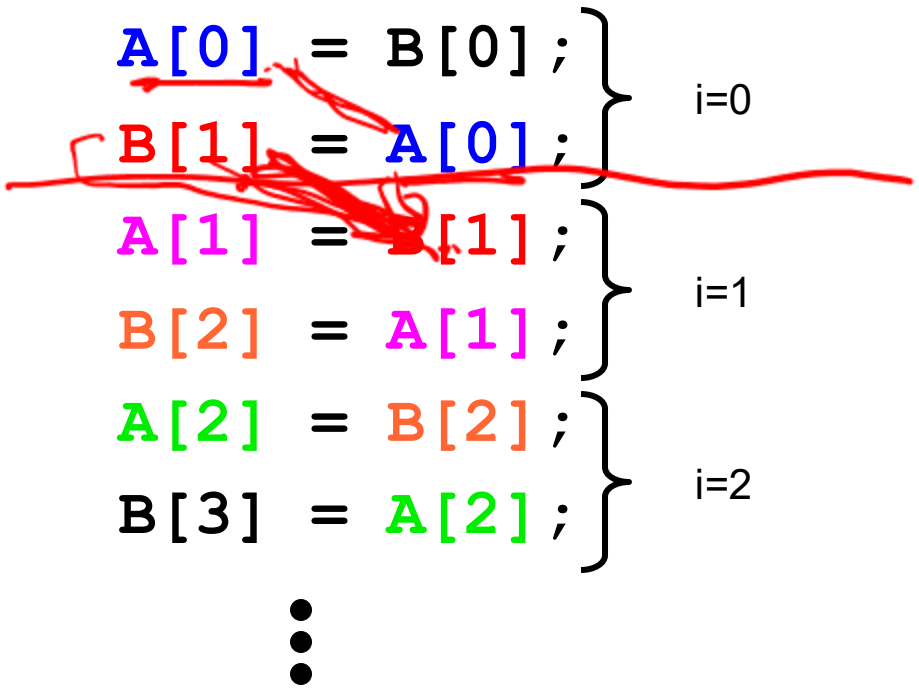


Unroll Loop to Find Dependencies

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

δ^f loop carried

δ^f loop independent



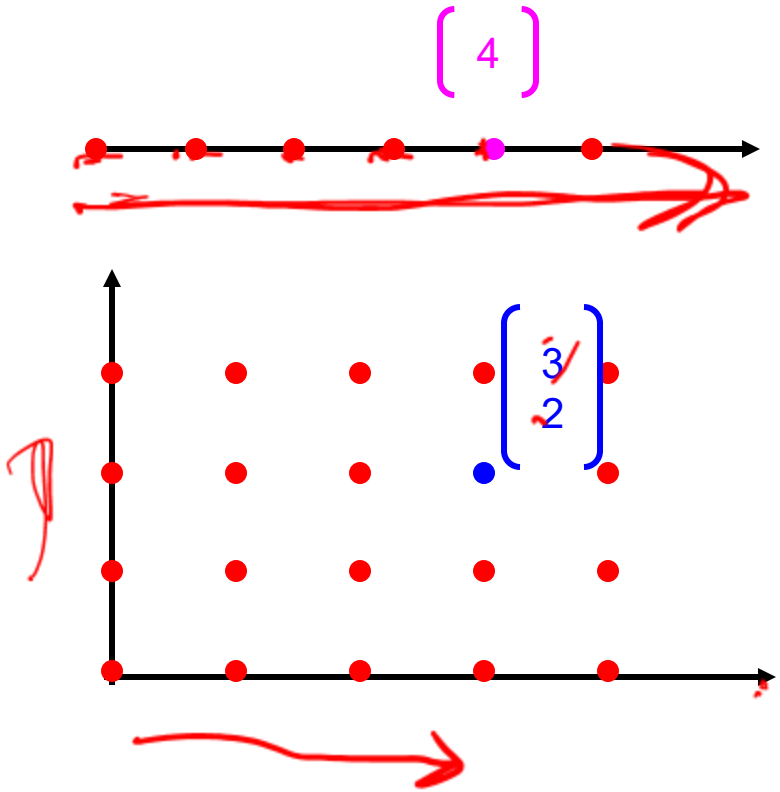
Distance/Direction of the dependence is also important.

Iteration Space

Every iteration generates a point in an n-dimensional space, where n is the depth of the loop nest.

```
for (i=0; i<n; i++) {
    •••
}
```

```
for (i=0; i<n; i++)
    for (j=0; j<4; j++) {
        •••
}
```



Distance Vector

```
for (i=0; i<n; i++) {
     A[i] = B[i];
     B[i+1] = A[i];
}
```

```
A[0] = B[0];   }
B[1] = A[0];   }  i=0
A[1] = B[1];   }
B[2] = A[1];   }  i=1
A[2] = B[2];   }
B[3] = A[2];   }  i=2
    ⋮
```

Distance vector is the difference between the target and source iterations.

$d = I_t - I_s$

Exactly the distance of the dependence, i.e.,

$I_s + d = I_t$

Example of Distance Vectors

```
for (i=0; i<n; i++)
   for (j=0; j<m; j++){
      A[i,j] =    ;
          = A[i,j];
      B[i,j+1] =    ;
          = B[i,j];
      C[i+1,j] =    ;
          = C[i,j+1] ;
   }
```

j

i

$A_{0,2}$= =$A_{0,2}$ $B_{0,3}$= =$B_{0,2}$ $C_{1,2}$= =$C_{0,3}$	$A_{1,2}$= =$A_{1,2}$ $B_{1,3}$= =$B_{1,2}$ $C_{2,2}$= =$C_{1,3}$	$A_{2,2}$= =$A_{2,2}$ $B_{2,3}$= =$B_{2,2}$ $C_{3,2}$= =$C_{2,3}$
$A_{0,1}$= =$A_{0,1}$ $B_{0,2}$= =$B_{0,1}$ $C_{1,1}$= =$C_{0,2}$	$A_{1,1}$= =$A_{1,1}$ $B_{1,2}$= =$B_{1,1}$ $C_{2,1}$= =$C_{1,2}$	$A_{2,1}$= =$A_{2,1}$ $B_{2,2}$= =$B_{2,1}$ $C_{3,1}$= =$C_{2,2}$
$A_{0,0}$= =$A_{0,0}$ $B_{0,1}$= =$B_{0,0}$ $C_{1,0}$= =$C_{0,1}$	$A_{1,0}$= =$A_{1,0}$ $B_{1,1}$= =$B_{1,0}$ $C_{2,0}$= =$C_{1,1}$	$A_{2,0}$= =$A_{2,0}$ $B_{2,1}$= =$B_{2,0}$ $C_{3,0}$= =$C_{2,1}$

Example of Distance Vectors

```
for (i=0; i<n; i++)
   for (j=0; j<m; j++){
      A[i,j] =    ;
          = A[i,j];
      B[i,j+1] =    ;
          = B[i,j];
      C[i+1,j] =    ;
          = C[i,j+1] ;
   }
```

$A_{0,2}$= $B_{0,3}$= $C_{1,2}$=	=$A_{0,2}$ =$B_{0,2}$ =$C_{0,3}$	$A_{1,2}$= $B_{1,3}$= $C_{2,2}$=	=$A_{1,2}$ =$B_{1,2}$ =$C_{1,3}$	$A_{2,2}$= $B_{2,3}$= $C_{3,2}$=	=$A_{2,2}$ =$B_{2,2}$ =$C_{2,3}$
$A_{0,1}$= $B_{0,2}$= $C_{1,1}$=	=$A_{0,1}$ =$B_{0,1}$ =$C_{0,2}$	$A_{1,1}$= $B_{1,2}$= $C_{2,1}$=	=$A_{1,1}$ =$B_{1,1}$ =$C_{1,2}$	$A_{2,1}$= $B_{2,2}$= $C_{3,1}$=	=$A_{2,1}$ =$B_{2,1}$ =$C_{2,2}$
$A_{0,0}$= $B_{0,1}$= $C_{1,0}$=	=$A_{0,0}$ =$B_{0,0}$ =$C_{0,1}$	$A_{1,0}$= $B_{1,1}$= $C_{2,0}$=	=$A_{1,0}$ =$B_{1,0}$ =$C_{1,1}$	$A_{2,0}$= $B_{2,1}$= $C_{3,0}$=	=$A_{2,0}$ =$B_{2,0}$ =$C_{2,1}$

j

i

A yields: $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$

B yields: $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$

C yields: $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$

Data Dependences

Loop carried: between two statements instances in two different iterations of a loop.

Loop independent: between two statements instances in the same loop iteration.

Lexically forward: the source comes before the target .

Lexically backward: otherwise.

The right-hand side of an assignment is considered to precede the left-hand side.

Lexicographic Order
Example of vectors

Consider the vectors **a** and **b** below :

$$\mathbf{a} = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

We say that **a** is lexicographically less than **b** at level 3, $\mathbf{a} \prec_3 \mathbf{b}$, or simply that $\mathbf{a} \prec \mathbf{b}$.

Both **a** and **b** are lexicographically positive because $\mathbf{0} \prec \mathbf{a}$, and $\mathbf{0} \prec \mathbf{b}$.

Dependence Vectors

- Dependence vector in an n-nested loop is denoted as a vector: **d**=(d_1, d_2, …, d_n).
- Each d_i is a possibly infinite range of ints in $\left[d_i^{\min}, d_i^{\max}\right]$, where

$$d_i^{\min} \in Z \cup \{-\infty\}, d_i^{\max} \in Z \cup \{\infty\} \text{ and } d_i^{\min} \leq d_i^{\max}$$

- So, a single dep vector represents a set of distance vectors.
- A distance vector defines a distance in the iteration space.
- A dependence vector is a distance vector if each d_i is a singleton.

Other defs

- Common ranges in dependence vectors
  - [1, ∞] as + or >
  - [- ∞, -1] as – or <
  - [- ∞, ∞] as ± or *

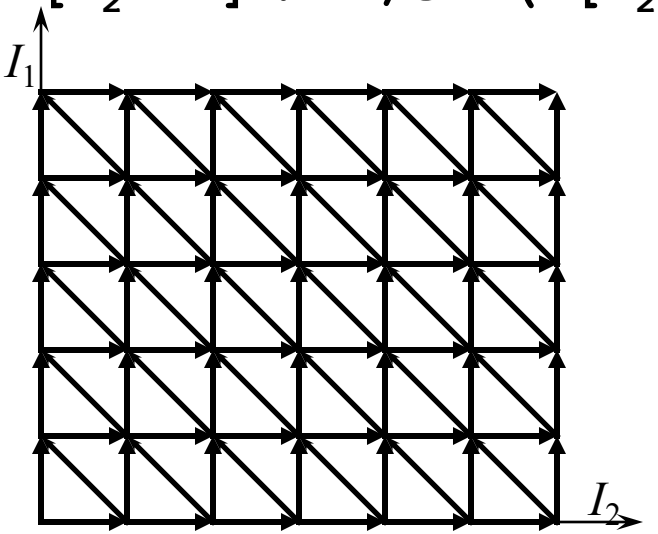
- A distance vector is the difference between the target and source iterations (for a dependent ref), e.g., $\mathbf{d} = \mathbf{I}_t - \mathbf{I}_s$

Examples

```
for I1 := 1 to n
   for I2 := 1 to n
      for I3 := 1 to n
         C[I1,I3] += A[I1,I2] * B[I2,I3]
```

(0,1,0)

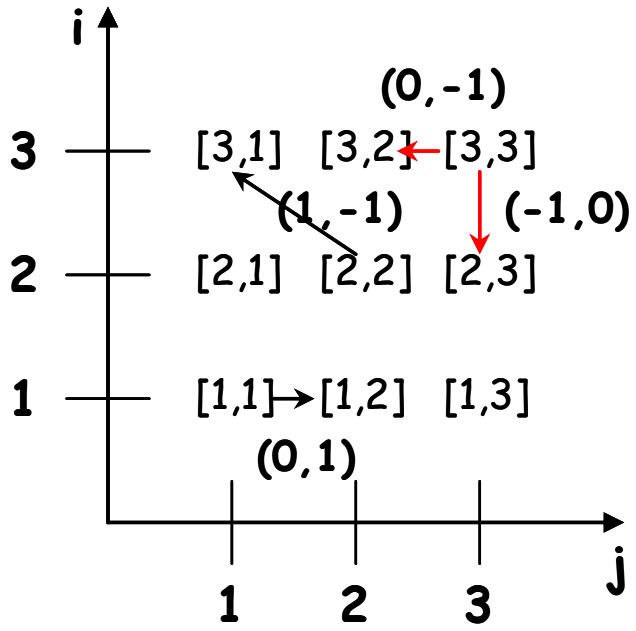
```
for I1 := 0 to 5
  for I2 := 0 to 6
    A[I2 + 1] := 1/3 * (A[I2] + A[I2 + 1] + A[I2 + 2])
```



D={(0,1),(1,0),(1-1)}

Plausible Dependence vectors

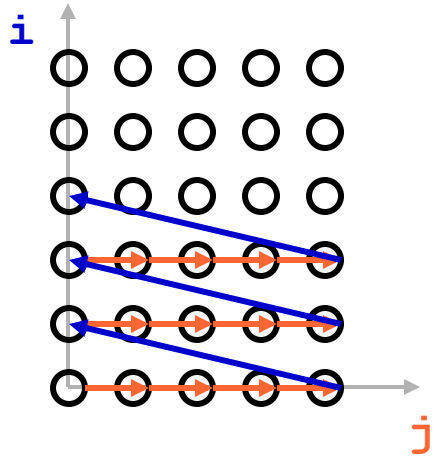
- A dependence vector is plausible iff it is lexicographically non-negative.
- All sequential programs have plausible dependence vectors. Why?
- Plausible: (1,-1)
- implausible (-1,0)



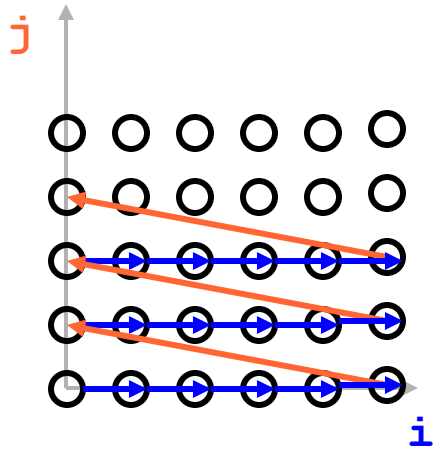
Loop Transforms

- A loop transformation changes the order in which iterations in the iteration space are visited.
- For example, Loop Interchange

```
for i := 0 to n
  for j := 0 to m
    body
```




```
for j := 0 to m
  for i := 0 to n
    body
```



Unimodular Transforms

- Interchange
  permute nesting order
- Reversal
  reverse order of iterations
- Skewing
  scale iterations by an outer loop index

Interchange

- Change order of loops
- For some permutation p of 1 … n

```
for I1 := …
    for I2 := …
      …
        for In := …
            body
```

```
for Ip(1) := …
    for Ip(2) := …
      …
        for Ip(n) := …
            body
```

- When is this legal?

Transform and matrix notation

- If dependences are vectors in iter space, then transforms can be represented as matrix transforms
- E.g., for a 2-deep loop, interchange is:

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} p_2 \\ p_1 \end{bmatrix}$$

- Since, T is a linear transform, T**d** is transformed dependence:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} d_2 \\ d_1 \end{bmatrix}$$

Reversal

- Reversal of i^{th} loop reverses its traversal, so it can be represented as:

Reversal

- Reversal of ith loop reverses its traversal, so it can be represented as:
  Diagonal matrix with ith element = -1.

- For 2 deep loop, reversal of outermost is:

$$T = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} -p_1 \\ p2 \end{bmatrix}$$

Skewing

- Skew loop I_j by a factor f w.r.t. loop I_i maps

$$(p_1,\ldots,p_i,\ldots,p_j,\ldots) \qquad (p_1,\ldots,p_i,\ldots,p_j + fp_i,\ldots)$$

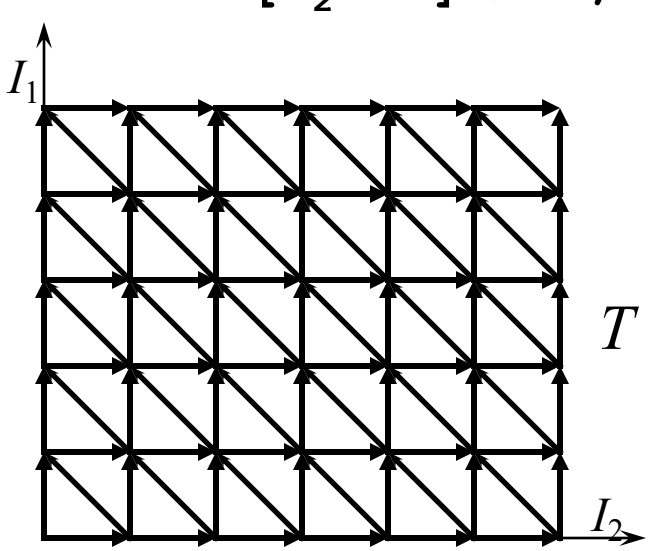
- Example for 2D

$$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 + p_1 \end{bmatrix}$$

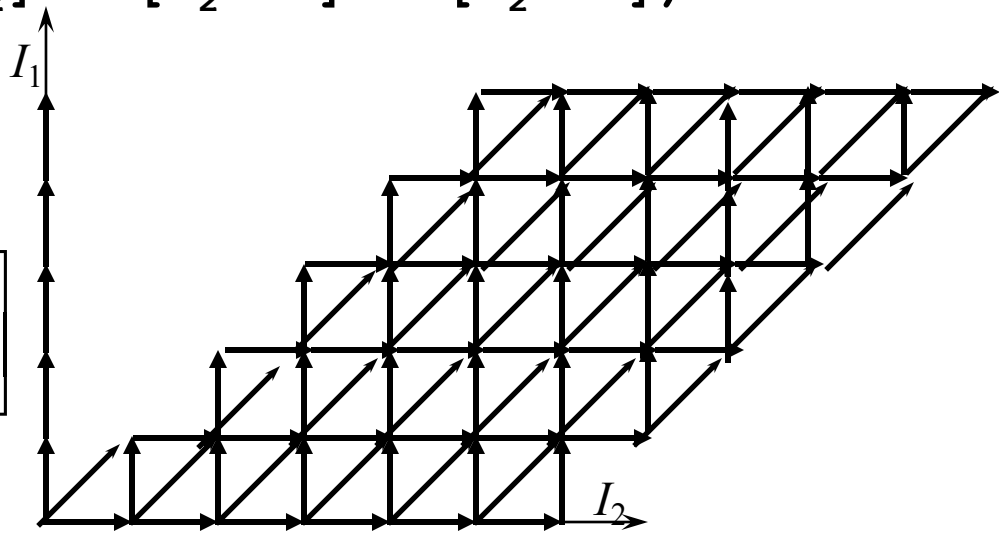
Loop Skewing Example

```
for I1 := 0 to 5
  for I2 := 0 to 6
    A[I2 + 1] := 1/3 * (A[I2] + A[I2 + 1] + A[I2 + 2])
```

D={(0,1),(1,0),(1-1)}



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




```
for I1 := 0 to 5
  for I2 := I1 to 6+I1
    A[I2-I1+1] := 1/3 * (A[I2-I1] + A[I2-I1+ 1] + A[I2-I1+ 2])
```

D={(0,1),(1,1),(1,0)}

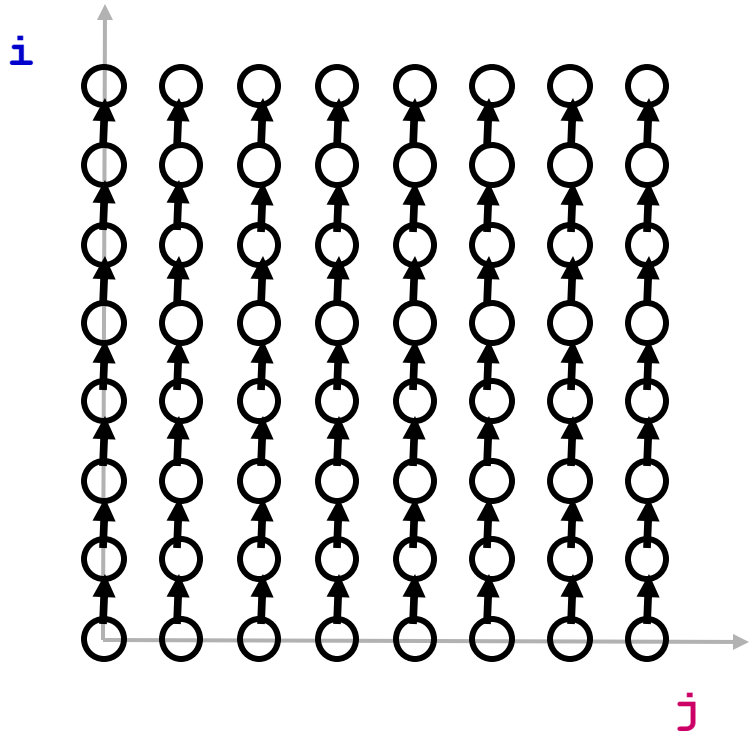
But...is the transform legal?

- Distance/direction vectors give a partial order among points in the iteration space
- A loop transform changes the order in which 'points' are visited
- The new visit order must respect the dependence partial order!

But...is the transform legal?

- Loop reversal ok?
- Loop interchange ok?

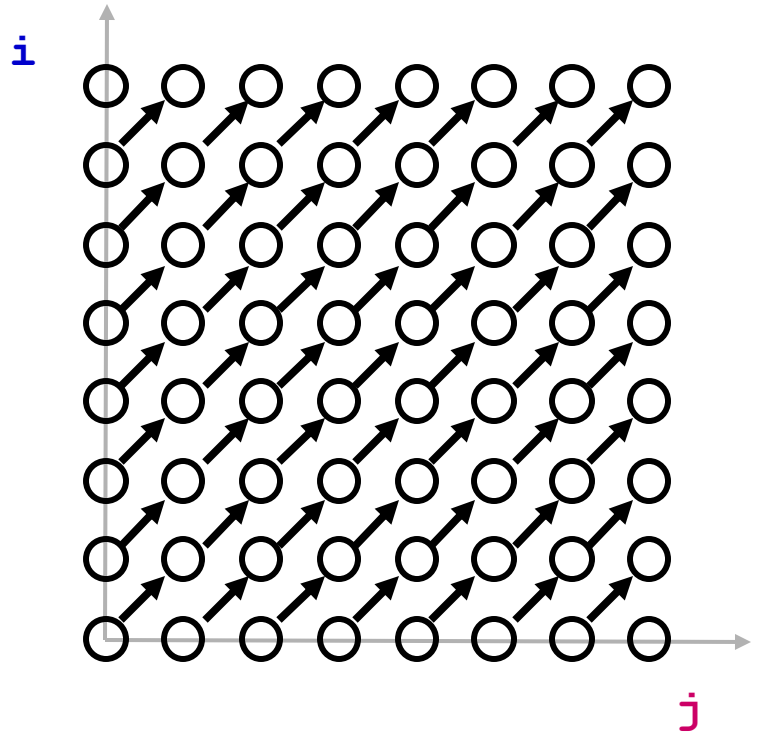
```
for i = 0 to N-1
    for j = 0 to N-1
       A[i+1][j] += A[i][j];
```



But...is the transform legal?

- Loop reversal ok?
- Loop interchange ok?

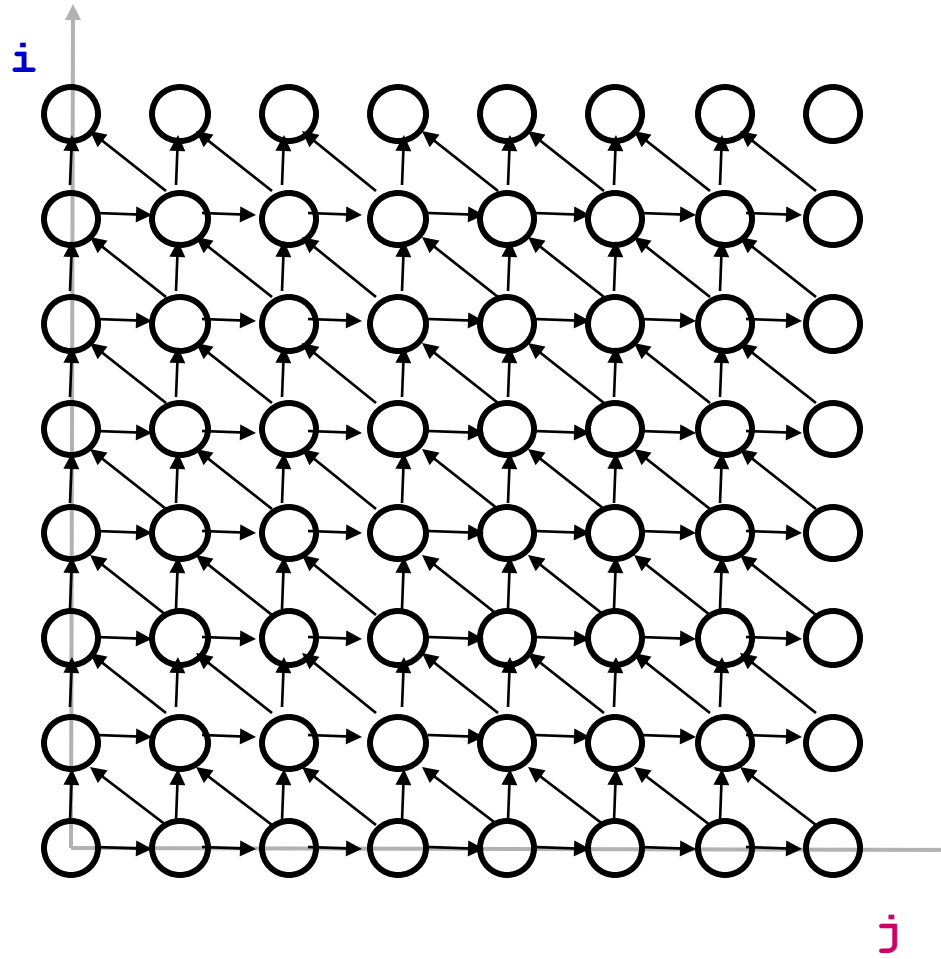
```
for i = 0 to N-1
    for j = 0 to N-1
       A[i+1][j+1] += A[i][j];
```



But...is the transform legal?

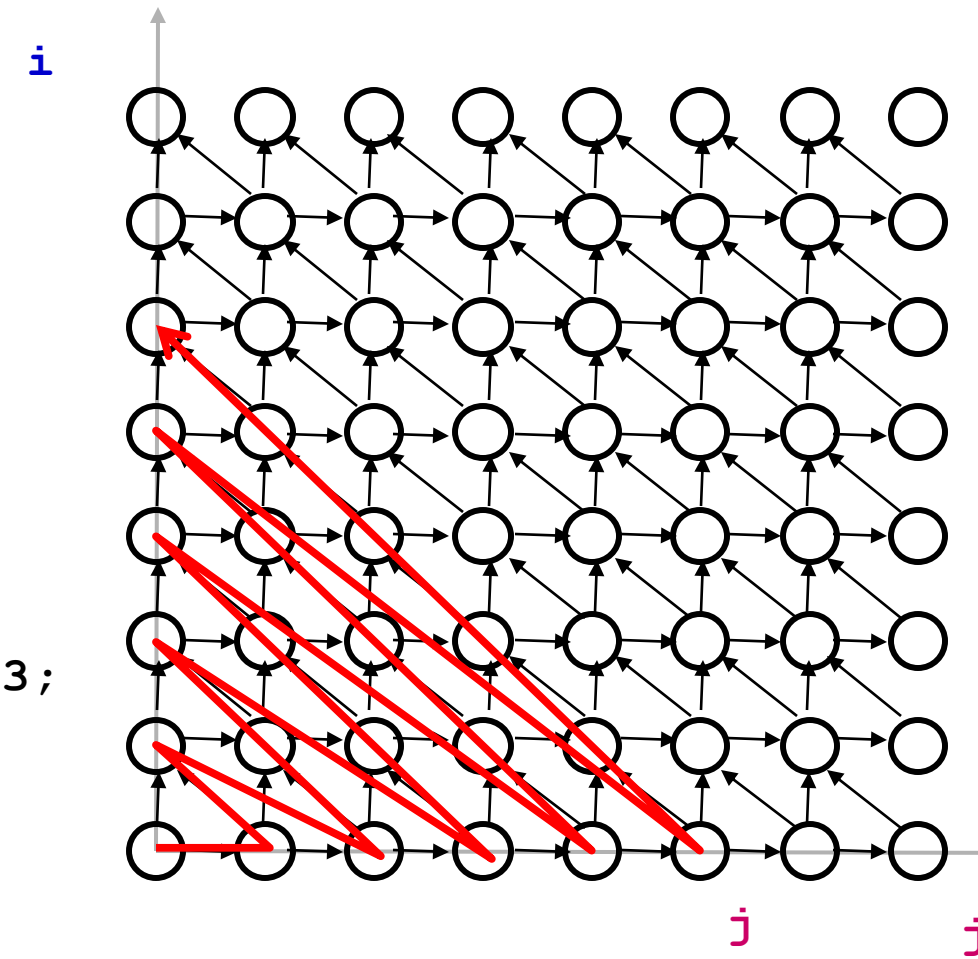
- What other visit order is legal here?

```
for i = 0 to TS
  for j = 0 to N-2
    A[j+1] =
      (A[j] + A[j+1] + A[j+2])/3;
```



But...is the transform legal?

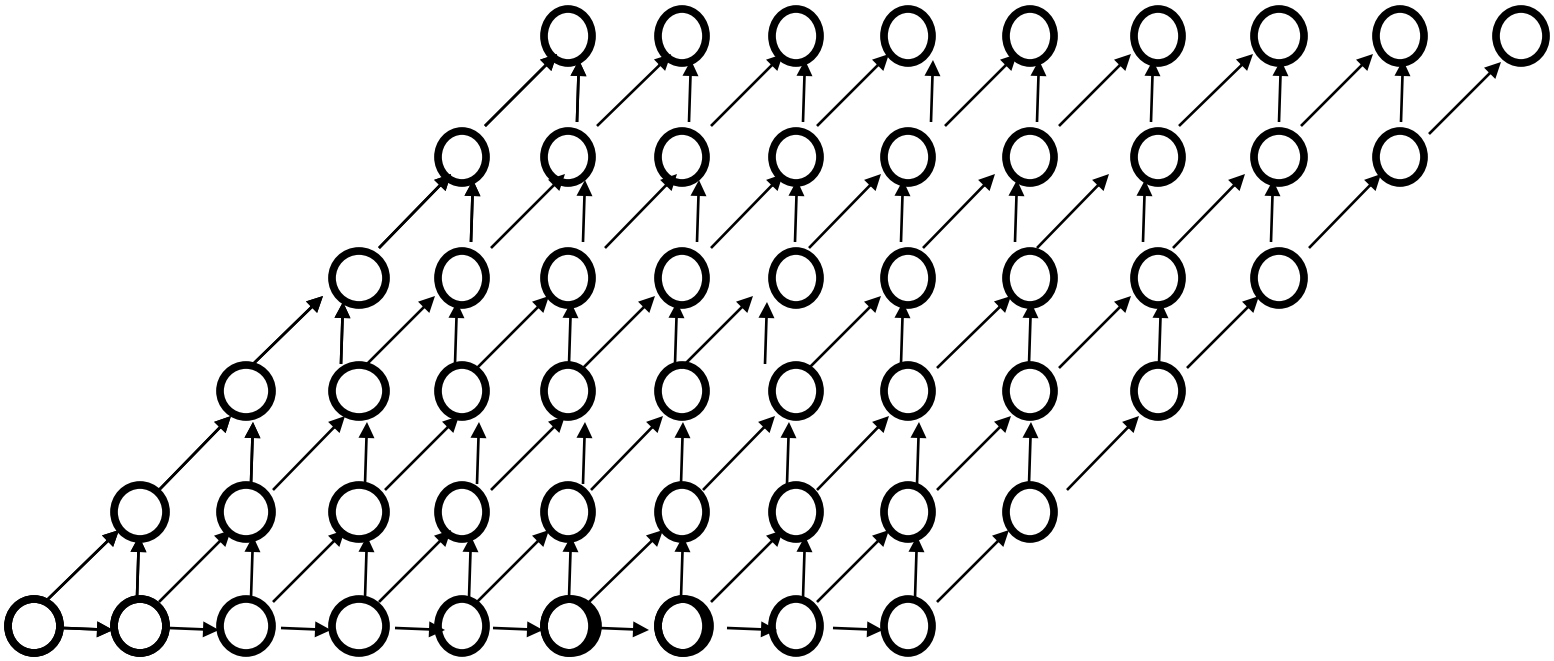
- What other visit order is legal here?




```
for i = 0 to TS
    for j = 0 to N-2
      A[j+1] =
        (A[j] + A[j+1] + A[j+2])/3;
```

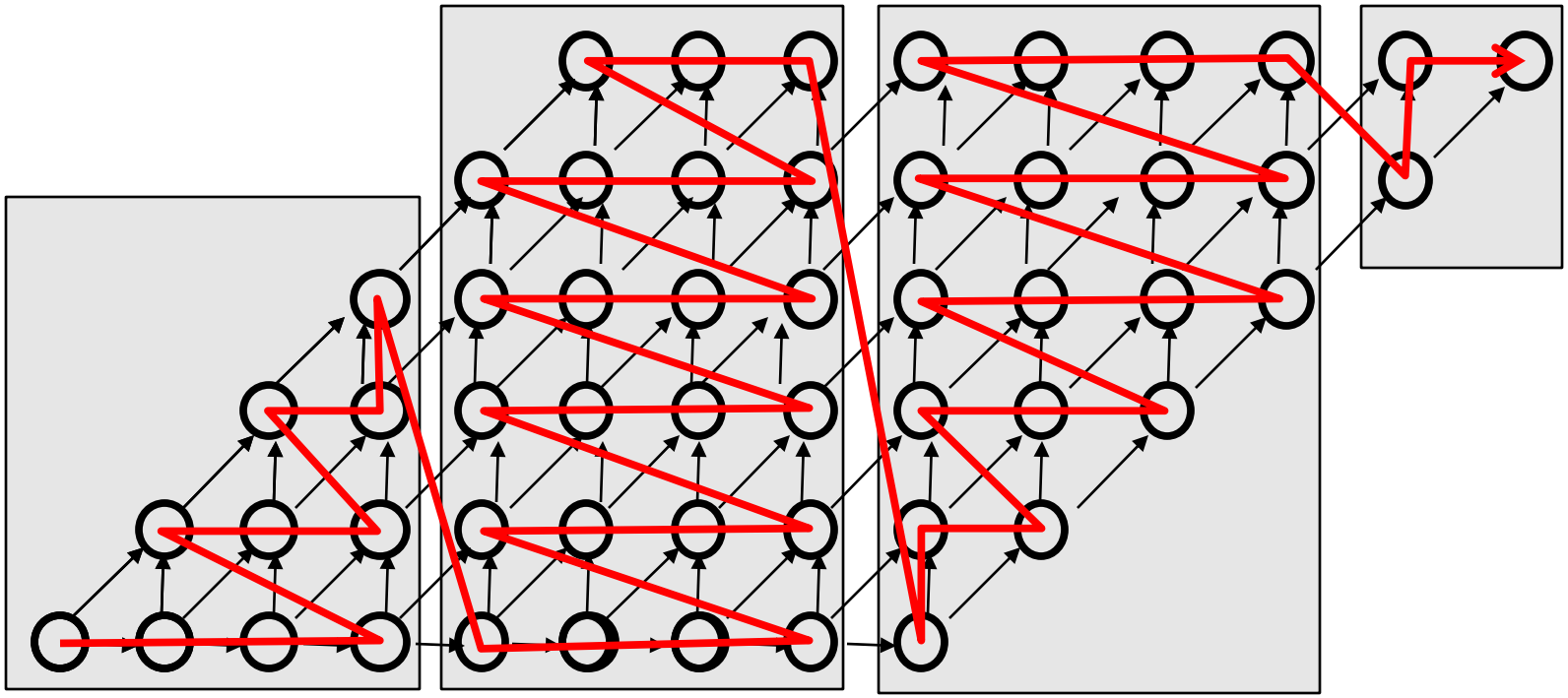
But...is the transform legal?

- Skewing...



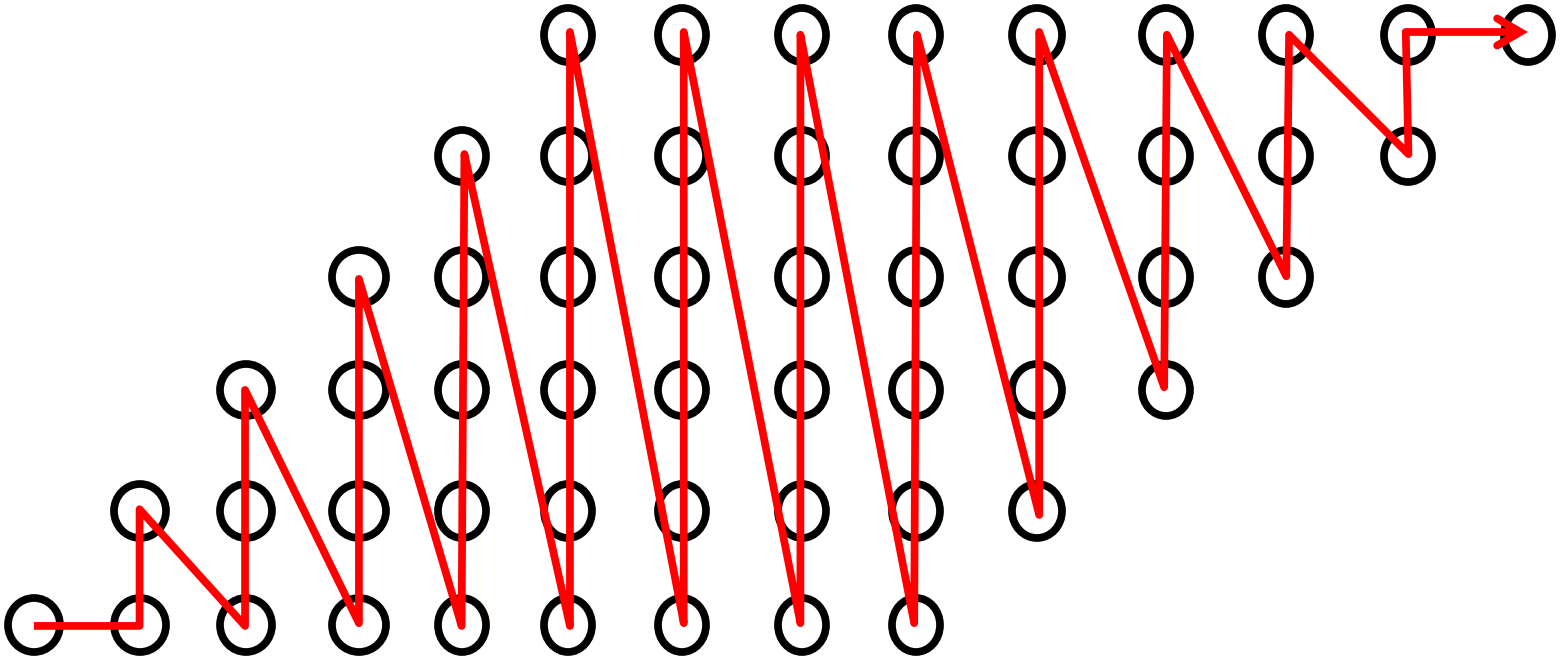
But...is the transform legal?

- Skewing...now we can block



But...is the transform legal?

- Skewing...now we can loop interchange



Unimodular transformations

- Express loop transformation as a matrix multiplication
- Check if any dependence is violated by multiplying the distance vector by the matrix – if the resulting vector is still lexicographically positive, then the involved iterations are visited in an order that respects the dependence.

Reversal

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

Interchange

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Skew

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

"A Data Locality Optimizing Algorithm", M.E.Wolf and M.Lam

Finding Data Dependences

The General Problem

```
DO i1 = L1, U1
  DO i2 = L2, U2
                ...
        DO in = Ln, Un
 S1               A(f1(i1,...,in),...,fm(i1,...,in)) = ...
 S2               ... = A(g1(i1,...,in),...,gm(i1,...,in))
        ENDDO
        ...
  ENDDO
ENDDO
```

A dependence exists from S1 to S2 if:

– There exist α and β such that

- $\alpha < \beta$ (control flow requirement)
- $f_i(\alpha) = g_i(\beta)$ for all i, $1 \le i \le m$ (common access requirement)
- [at least one is a write]

Example

```
for (i=0; i<N; i++) {
    A[i] = A[i-1]+C;
}
```

Example

```
for (i=0; i<N; i++) {
    A[i] = A[i-1]+C;
}
```

- At iteration i+1 we read what was written at iteration i
- Distance vector: (1)
- Dependence vector: (<)

General Solver?

- Looking for an interger solution to:

$$f_i(\alpha) = g_i(\beta) \text{ for all } i,\ 1 \leq i \leq m$$

- N-deep loop nest
- M subscripts per array reference
- General case, too hard
- Restrict to linear functions of loop-indices
- System of linear equations
  (2n variables and m equations)

Basics: Conservative Testing

- Consider only linear subscript expressions
- Finding integer solutions to system of linear Diophantine Equations is NP-Complete
- Most common approximation is Conservative Testing, i.e., See if you can assert

  "No dependence exists between two subscripted references of the same array"
- Never incorrect, may be less than optimal

Basics: Indices and Subscripts

Index: Index variable for some loop surrounding a pair of references

Subscript: A PAIR of subscript positions in a pair of array references

For Example:

```
A[I][j] = A[I][k] + C
```

`<I,I>` is the first subscript

`<j,k>` is the second subscript

Basics: Complexity

A subscript is said to be

- ZIV if it contains no index
  zero index variable
- SIV if it contains only one index
  single index variable
- MIV if it contains more than one index
  multiple index variable

For Example:

```
A[5][I+1][j] = A[1][I][k] + C
```

First subscript is ZIV

Second subscript is SIV

Third subscript is MIV

Basics: Separability

- A subscript is separable if its indices do not occur in other subscripts
- If two different subscripts contain the same index they are coupled

For Example:

```
A[I+1][j] = A[k][j] + C
```

Both subscripts are separable

```
A[I][j][j] = A[I][j][k] + C
```

Second and third subscripts are coupled

Basics:Coupled Subscript Groups

- Why are they important?

  Coupling can cause imprecision in dependence testing

```
   for (i=0; i<= 100; i++) {
S1    A[i+1][i] = B[i] + C
S2    D[i] = A[i][i] * E
   ENDDO
```

Dependence Testing: Overview

- Partition subscripts of a pair of array references into separable and coupled groups
- Classify each subscript as ZIV, SIV or MIV
  - Reason for classification is to reduce complexity of the tests.
- For each separable subscript apply single subscript test. Continue until prove independence.
- Deal with coupled groups
- If independent, done
- Otherwise, merge all direction vectors computed in the previous steps into a single set of direction vectors

Classify as ZIV/SIV/MIV

- Easy step
- Just count the number of different indices in a subscript

Applying Single Subscript Tests

- ZIV Test
- SIV Test
  - Strong SIV Test
  - Weak SIV Test
    - Weak-zero SIV
    - Weak Crossing SIV
- SIV Tests in Complex Iteration Spaces

ZIV Test

```
  for (i=0; i<N; i++) {
S       A[e1] = A[e2] + B[j]
  }
```

e1,e2 are constants or loop invariant symbols

If (e1-e2)!=0 No Dependence exists

Strong SIV Test

- Strong SIV subscripts are of the form

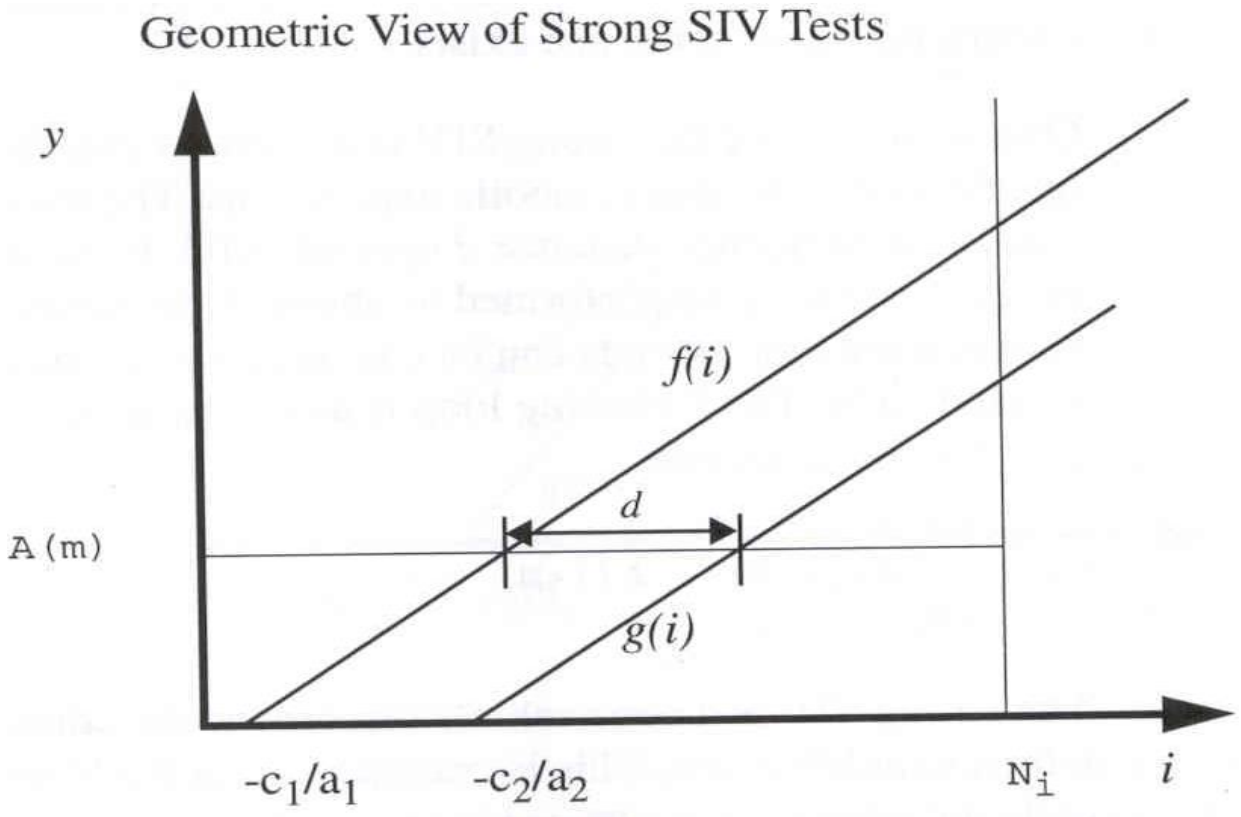
$$\langle ai + c_1, ai + c_2 \rangle$$

- For example the following are strong SIV subscripts

$$\langle i + 1, i \rangle$$

$$\langle 4i + 2, 4i + 4 \rangle$$

Strong SIV Test $\langle ai + c_1, ai + c_2 \rangle$



$$d = i' - i = \frac{c_1 - c_2}{a}$$

Dependence exists if: $|d| \leq U - L$

Strong SIV Test Example

```
for (k=0; k<100, k++) {
   for (j=0; j<100; j++) {
        A[j+1][k] = ...
              ... = A[j][k] + 32
   }
}
```

General test

- The GCD test can give conservative answer to “dependent” or “independent”
- For each pair of subscripts:
  - Write to A[ai+c1]
  - Read from A[bi+c2]
- Want to see if there is a solution to

  $a\alpha + c1 = b\beta + c2$
- Looking for $a\alpha - b\beta = c2 - c1$, i.e.,
- Does GCD(α, β) divide (c2-c1)?

Using GCD test

```
for (k=0; k<100, k++) {
   A[2*k+10] = ...
               ... = A[2*k+1]
   }
}
```

Hierarchy of Tests

- GCD – existence without bound checking
- Strong SIV for subscript pairs of form:

  $i+c_1$, $i+c_2$
- Banerjee Test: uses bounds to check for existence
- Delta Test: for coupled subscript pairs
- Omega Test: uses ILP to find exact distance if it exists

Loops & Array Dependence

- Test for independence
- If independent, then can:
  - Parallelize
  - Rearrange for locality
  - Many others
- If dependent and distance vectors can possibly rearrange loop to do above
- See Wolff & Lam