Locality - 2

15-411/15-611 Compiler Design

Seth Copen Goldstein

November 17, 2020

Our Goal: Increase locality

Is there locality to exploit? → Use Reuse Analysis to determine amount of possible reuse.

↓

Can we transform loop to turn reuse into locality? → Use dependence information to determine pace of possible transformations.

↓

Transform Loop using SRP → Perform unimodular transformations.

↓

Possibly introduce Tiling → turn n-deep into 2n-deep

Our Goal: Increase locality

Is there locality to exploit?

Use Reuse Analysis to determine amount of possible reuse

Key idea: Treat each iteration of loop nest as a point in an n-dimensional iteration space.

transformations.

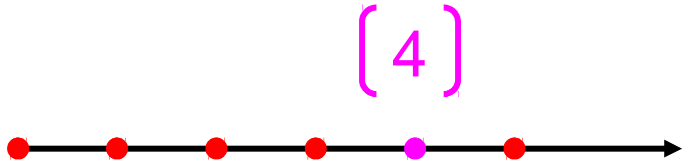
Possibly introduce Tiling

turn n-deep into 2n-deep

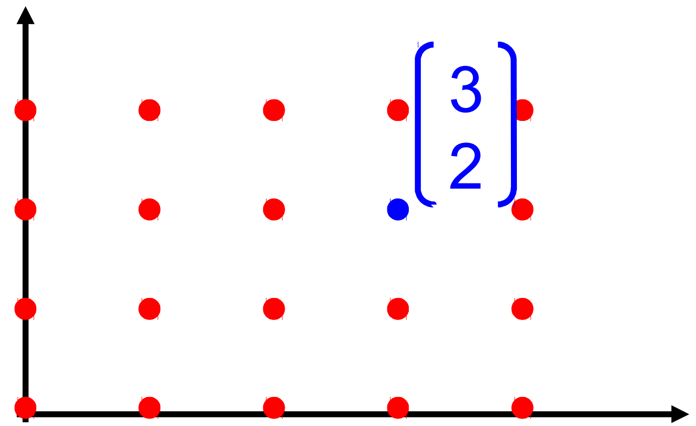
Iteration Space

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++) {
      …
}
```



Iteration Vectors

- Given a nest of n loops, the iteration vector *i* of a particular iteration of the innermost loop is a vector of integers that contains the iteration numbers for each of the loops in order of nesting level.
- Thus, the iteration vector is: $\{i_1, i_2, ..., i_n\}$
  where i_k, $1 \leq k \leq n$ represents the iteration number for the loop at nesting level k

Ordering of Iteration Vectors

- An ordering for iteration vectors
- Use an intuitive, lexicographic order
- Iteration i precedes iteration j, denoted $i < j$, iff:
  1. $i[1:n-1] < j[1:n-1]$, or
  2. $i[1:k-1] = j[1:k-1]$ and $i_k < j_k$

$$\begin{pmatrix} i_1 \\ i_2 \\ \dots \\ i_k \\ \dots \\ i_n \end{pmatrix} < \begin{pmatrix} j_1 \\ j_2 \\ \dots \\ j_k \\ \dots \\ j_n \end{pmatrix}$$

Uniformly Generated references

- f and g are indexing functions: $Z^n \rightarrow Z^d$
  - n is depth of loop nest
  - d is dimensions of array, A
- Two references A[f(**i**)] and A[g(**i**)] are uniformly generated if

$$f(\mathbf{i}) = H\mathbf{i} + c_f \text{ AND } g(\mathbf{i}) = H\mathbf{i} + c_g$$

- H is a linear transform
- c_f and c_g are constant vectors

Uniformly generated sets

for I_1 := 0 to 5
  for I_2 := 0 to 6
    $A[I_2+1] = 1/3 * (A[I_2] + A[I_2+1] + A[I_2+2])$

$A[I_2+1]$ $\begin{bmatrix} 0 & 1 \end{bmatrix} \begin{pmatrix} I_1 \\ I_2 \end{pmatrix} + \begin{bmatrix} 1 \end{bmatrix}$

$A[I_2]$ $\begin{bmatrix} 0 & 1 \end{bmatrix} \begin{pmatrix} I_1 \\ I_2 \end{pmatrix} + \begin{bmatrix} 0 \end{bmatrix}$

$A[I_2+2]$ $\begin{bmatrix} 0 & 1 \end{bmatrix} \begin{pmatrix} I_1 \\ I_2 \end{pmatrix} + \begin{bmatrix} 2 \end{bmatrix}$

Predicting Cache Behavior through "Locality Analysis"

- Definitions:
  - Reuse:
    accessing a location that has been accessed in the past
  - Locality:
    accessing a location that is now found in the cache
- Key Insights
  - Locality only occurs when there is reuse!
  - BUT, reuse does not necessarily result in locality.
  - Why not?

Steps in Locality Analysis

1. Find data reuse
   - if caches were infinitely large, we would be finished
2. Determine "localized iteration space"
   - set of inner loops where the data accessed by an iteration is expected to fit within the cache
3. Find data locality:
   - reuse $\supseteq$ localized iteration space $\supseteq$ locality

Self-Temporal

- For a reference, A[H**i**+**c**], there is self-temporal reuse between **m** and **n** when H**m**+**c**=H**n**+**c**, i.e., H(**r**)=**0**, where **r**=**m**-**n**.
- The direction of reuse is **r**.
- The self-temporal reuse vector space is: R_{ST} = Ker H
- Amount of reuse is $s^{\dim(R_{ST})}$
- There is locality if $R_{ST} \subseteq$ localized vector space.
- $R_{ST} \cap L$ = locality
- \# of mem refs = $\frac{1}{s^{\dim(R_{ST} \cap L)}}$

Example of ST reuse

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

Uniformly Generated Set:

{A[I_2], A[I_2+1], A[I_2+2]} **H =** $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$

<u>Type</u> <u>reuse space</u> <u>reuse factor</u>

Self-Temporal: Ker(H) = span{(1,0)} s

Self-Spatial

- Occurs when we access in order
- Spatial reuse occurs when only last index varies
- So, all but last row of H must be identical
- H_s := H with last row set to 0
- self-spatial reuse vector space = R_{SS}

$$R_{SS} = \ker H_S$$

- Notice, $\ker H \subseteq \ker H_s$
- If, $R_{ss} \cap L = R_{ST} \cap L$, then no additional benefit to self-spatial reuse
- $\frac{k}{ls^{\dim(R_{SS} \cap L)}}$ memory accesses/iteration

Example of SS reuse

```
for I_1 := 0 to 5
  for I_2 := 0 to 6
    A[I_2 + 1] = 1/3 * (A[I_2]+ A[I_2 + 1] + A[I_2 + 2])
```

Uniformly Generated Set:

$\{A[I_2], A[I_2+1], A[I_2+2]\}$ $H = \begin{bmatrix} 0 & 1 \end{bmatrix}$ $H_s = \begin{bmatrix} 0 & 0 \end{bmatrix}$

<u>Type</u> <u>reuse space</u> <u>reuse factor</u>

Self-Temporal: Ker(H) = span{(1,0)} s

Self-Spatial: Ker(H_s) = span{(1,0),(0,1)}l

Group Reuse

- Occurs between **different** references in a loop nest when they access
  - the same element in the reuse vector space
  - the same cache line in the reuse vector space

Example of GT reuse

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

Uniformly Generated Set:

{A[I_2], A[I_2+1] ,A[I_2+2]} **H =** $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Type	reuse space	reuse factor
Self-Temporal:	Ker(H) = span{(1,0)}	s
Self-Spatial:	Ker(H_s) = span{(1,0),(0,1)}	l
Group-Temporal:	span{(1,0),(0,1)}	3

Turning Reuse into Locality

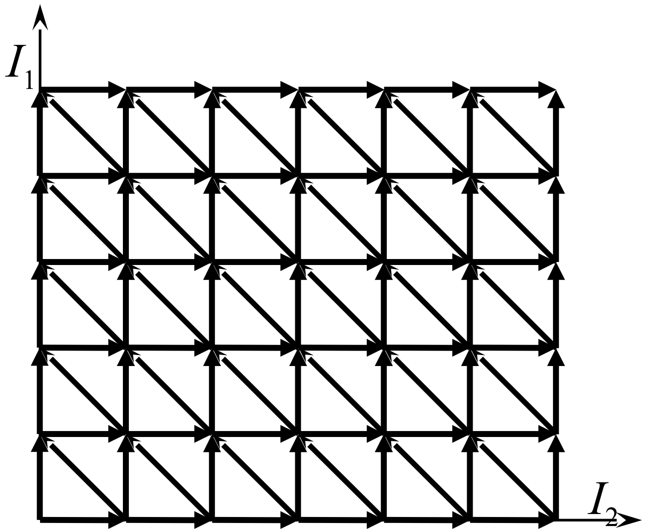
```
for I1 := 0 to 5
  for I2 := 0 to 6
    A[I2 + 1] = 1/3 * (A[I2]+ A[I2 + 1] + A[I2 + 2])
```

Type	reuse space	reuse factor
Self-Temporal:	Ker(H) = span{(1,0)}	s
Self-Spatial:	$Ker(H_s)$ = span{(1,0),(0,1)}	l
Group-Temporal:	span{(1,0),(0,1)}	3

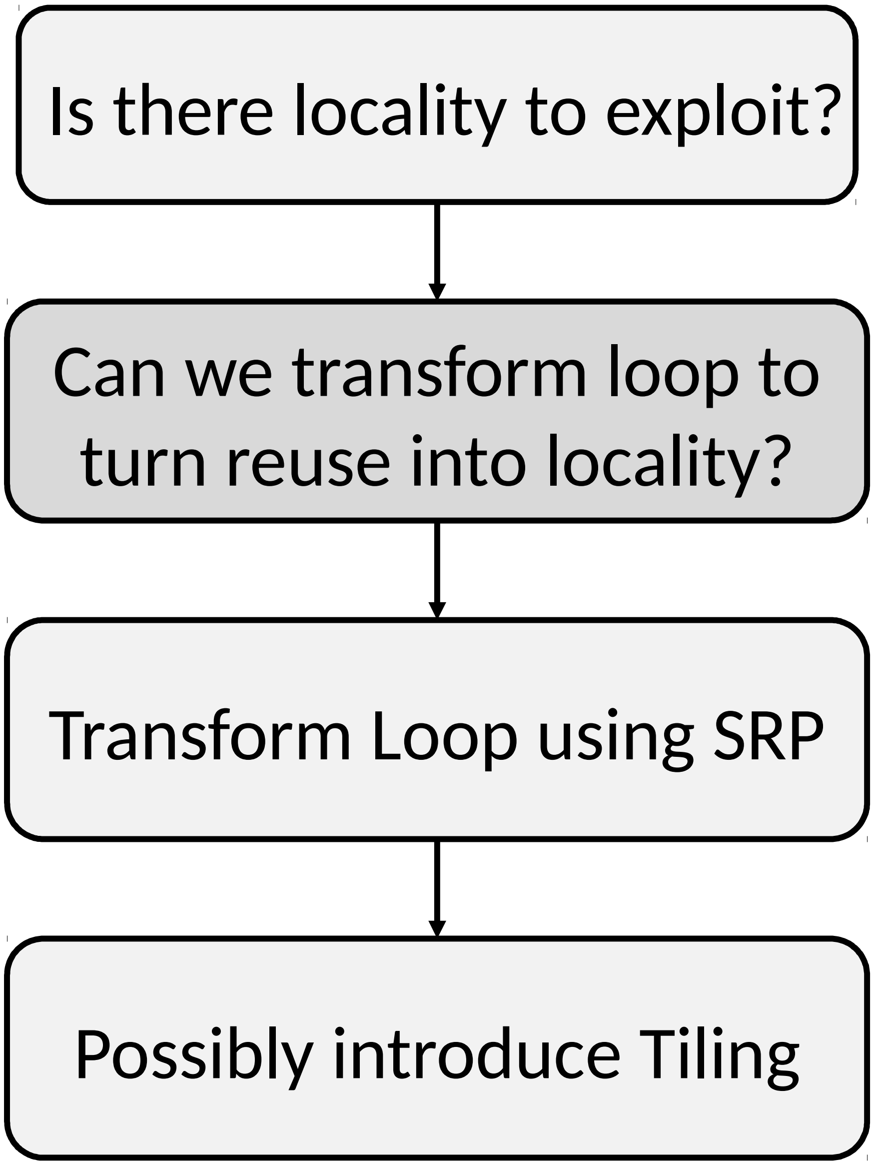
Turning Reuse into Locality

```
for I_1 := 0 to 5
  for I_2 := 0 to 6
    A[I_2 + 1] = 1/3 * (A[I_2]+ A[I_2 + 1] + A[I_2 + 2])
```

Type	reuse space	reuse factor
Self-Temporal:	Ker(H) = span{(1,0)}	s
Self-Spatial:	$Ker(H_s)$ = span{(1,0),(0,1)}	l
Group-Temporal:	span{(1,0),(0,1)}	3



Our Goal: Increase locality



Use Reuse Analysis to determine amount of possible reuse.

Use dependence information to determine pace of possible transformations.

Perform unimodular transformations.

turn n-deep into 2n-deep

Loop Dependence

- There exists a dependence from statements S_1 to statement S_2 in a common nest of loops iff there exist two iteration vectors *i* and *j* for the nest, st.

(1) (a) $i < j$ or

Loop Carried

(b) $i = j$ and there is a path from S_1 to S_2 in the body of the loop,

Loop independent

(2) statement S_1 accesses memory location *M* on iteration *i* and statement S_2 accesses location *M* on iteration *j*, and

(3) one of these accesses is a write.

Dependence Distance

- Using iteration vectors and definition of dependence we can determine the distance of a dependence:
- In n-deep loop nest if
  - S1 is source in iteration i
  - S2 is sink in iteration j
- Distance of dependence is represented with a distance vector: D
  - Vector of length n, where
  - $d_k = j_k - i_k$

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.

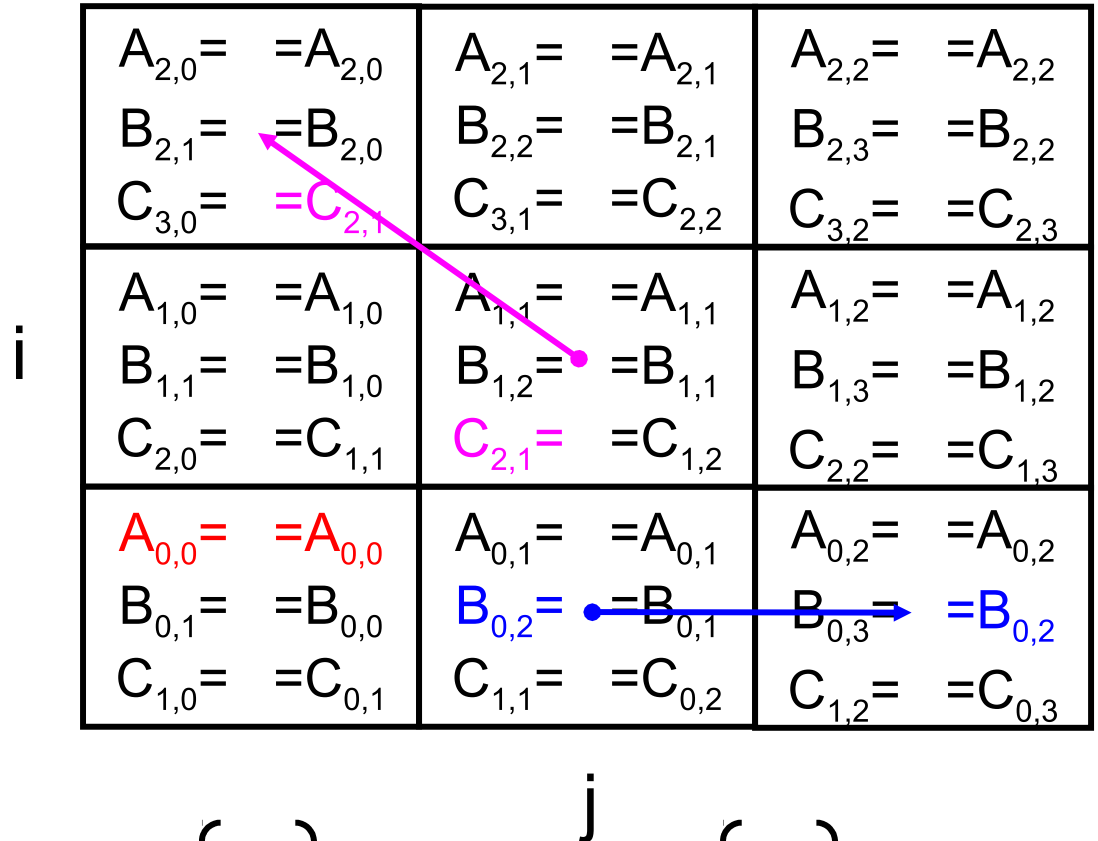
$$d = I_t - I_s$$

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] ;
   }
```



A yields: $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ B yields: $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ C yields: $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$

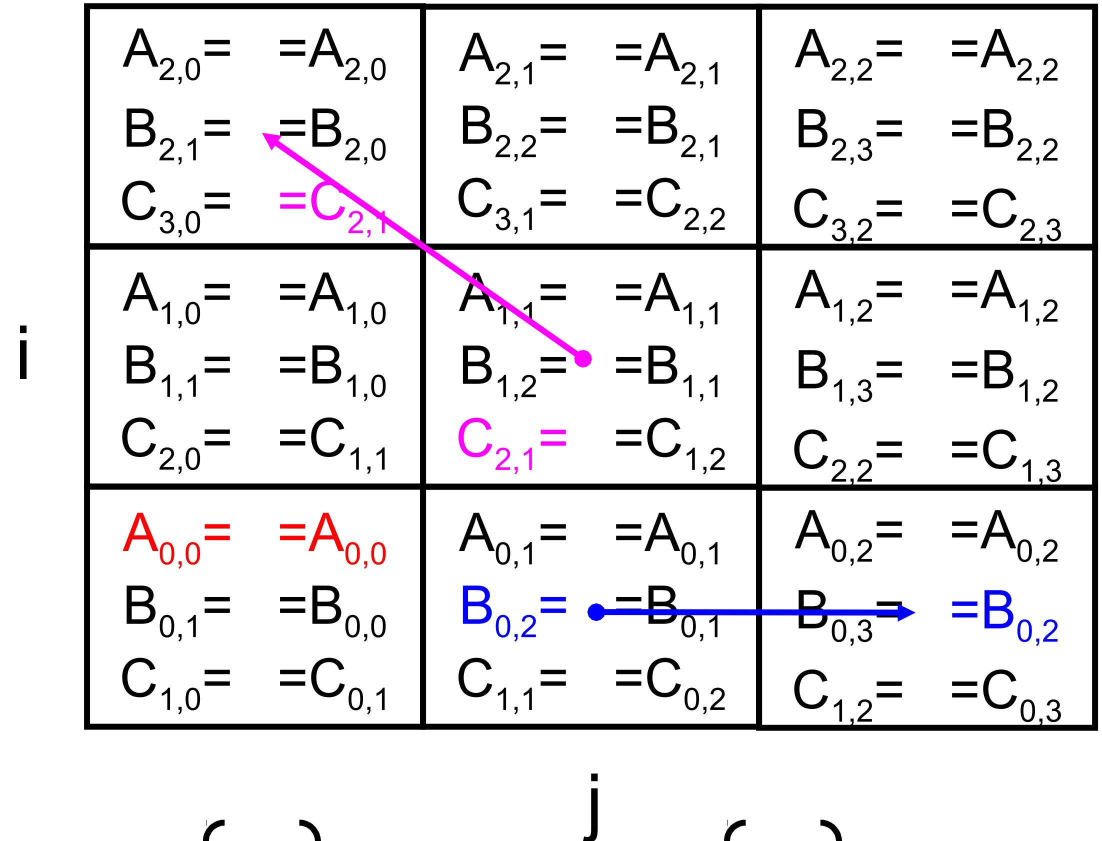
Direction Vectors

- Less precise than distance vectors, but often good enough
- In n-deep loop nest if
  - S1 is source in iteration i
  - S2 is sink in iteration j
- Distance vector: F - Vector of length n, where
  - $f_k = j_k - i_k$
- Direction vector also vector of length n, where
  - $d_k = \begin{cases} \text{“<“ if } f_k > 0, \text{ or } j_k < i_k \\ \text{“=“ if } f_k = 0, \text{ or } j_k = i_k \\ \text{“>“ if } f_k < 0, \text{ or } j_k > i_k \end{cases}$

Sometimes write ‘+’ for < and ‘-’ for >

Example of Direction Vectors

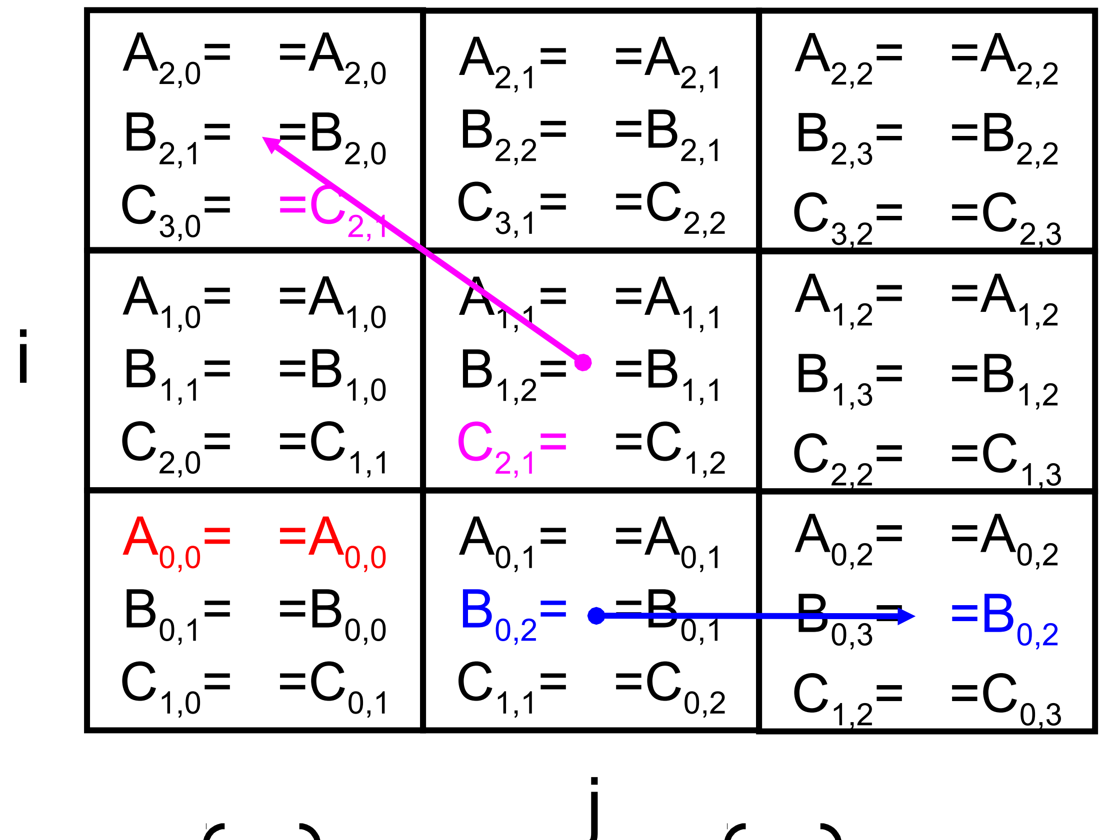
```
for (i=0; i<n; i++)
   for (j=0; j<m; j++){
      A[i,j] =    ;
           = A[i,j];
      B[i,j+1] =    ;
           = B[i,j];
      C[i+1,j] =    ;
           = C[i,j+1] ;
   }
```



A yields: $\begin{pmatrix} = \\ = \end{pmatrix}$ B yields: $\begin{pmatrix} = \\ < \end{pmatrix}$ C yields: $\begin{pmatrix} < \\ > \end{pmatrix}$

Example of Direction 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 yields: $\begin{pmatrix} = \\ = \end{pmatrix}$ B yields: $\begin{pmatrix} = \\ + \end{pmatrix}$ C yields: $\begin{pmatrix} + \\ - \end{pmatrix}$

Another Example

Example:

```
      DO I = 1, N
         DO J = 1, M
            DO K = 1, L
S1             A(I+1, J, K-1) = A(I, J, K) + 10
            ENDDO
         ENDDO
      ENDDO
```

- S_1 has a true dependence on itself.
- Distance Vector: (1, 0, -1)
- Direction Vector: (<, =, >)

Another Example

Example:

```
    DO I = 1, N
       DO J = 1, M
          DO K = 1, L
S1           A(I+1, J, K-1) = A(I, J, K) + 10
          ENDDO
       ENDDO
    ENDDO
```

- S_1 has a true dependence on itself.
- Distance Vector: (1, 0, -1)
- Direction Vector: (<, =, >)

$$\begin{pmatrix} I+1 \\ J \\ K-1 \end{pmatrix} - \begin{pmatrix} I \\ J \\ K \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$$

Note on vectors

- A dependence cannot exist if it has a direction vector whose leftmost non "=" component is not "<" as this would imply that the sink of the dependence occurs before the source.
- Likewise, the first non-zero distance in a distance vector must be postive.

The Key

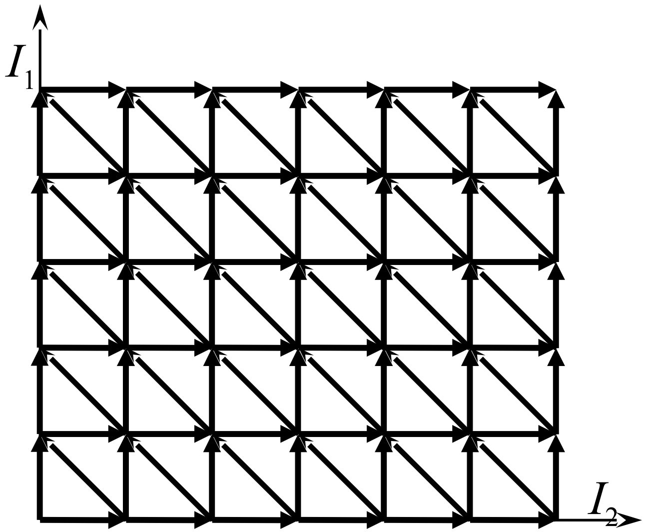
- Any reordering transformation that preserves every dependence in a program preserves the meaning of the program

- A reordering transformation may change order of execution but does not add or remove statements.

Blocking Not Currently Legal

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

Type	reuse space	reuse factor
Self-Temporal:	Ker(H) = span{(1,0)}	s
Self-Spatial:	$Ker(H_s)$ = span{(1,0),(0,1)}	l
Group-Temporal:	span{(1,0),(0,1)}	3



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)

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

```
    DO I = 1, 100
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

Step 1: Subscript Partitioning

- Partitions the subscripts into separable and minimal coupled groups
- Notations

// S is a set of m subscript pairs S_1, S_2, ...S_m each enclosed in n loops with indexes I_1, I_2, ... I_n, which is to be partitioned into separable or minimal coupled groups.

// P is an output variable, containing the set of partitions

// n_p is the number of partitions

Subscript Partitioning Algorithm

procedure *partition*(S, P, n_p)

$n_p = m$;

for i := 1 to m do $P_i = \{S_i\}$;

for i := 1 to n do begin

  k := <none>

  for each remaining partition P_j do

    if there exists s ε P_j such that s contains I_i then

      if k = < none > then $k = j$;

      else begin $P_k = P_k \cup P_j$; discard Pj; $n_p = n_p - 1$; end

end

end *partition*

Step 2: Classify as ZIV/SIV/MIV

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

Step 3: 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

```
  DO j = 1, 100
S       A(e1) = A(e2) + B(j)
  ENDDO
```

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

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

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

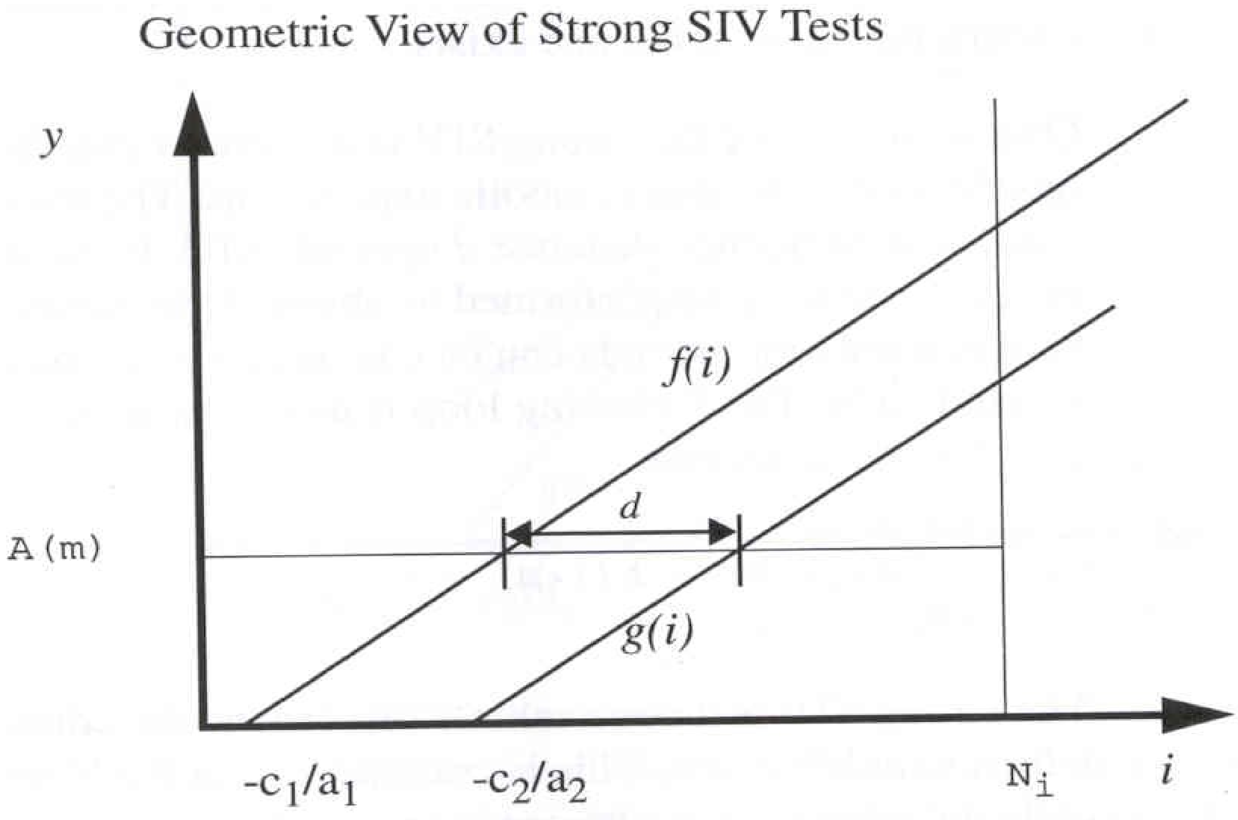
- Strong SIV subscripts are of the form
- For example the following are strong SIV subscripts

Strong SIV Test Example

```
DO k = 1, 100
  DO j = 1, 100
S1    A(j+1,k) = ...
S2       ... = A(j,k) + 32
  ENDDO
ENDDO
```

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

Strong SIV Test



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

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

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

Dependence exists if:

Weak SIV Tests

- Weak SIV subscripts are of the form

$$\langle a_1 i + c_1, a_2 i + c_2 \rangle$$

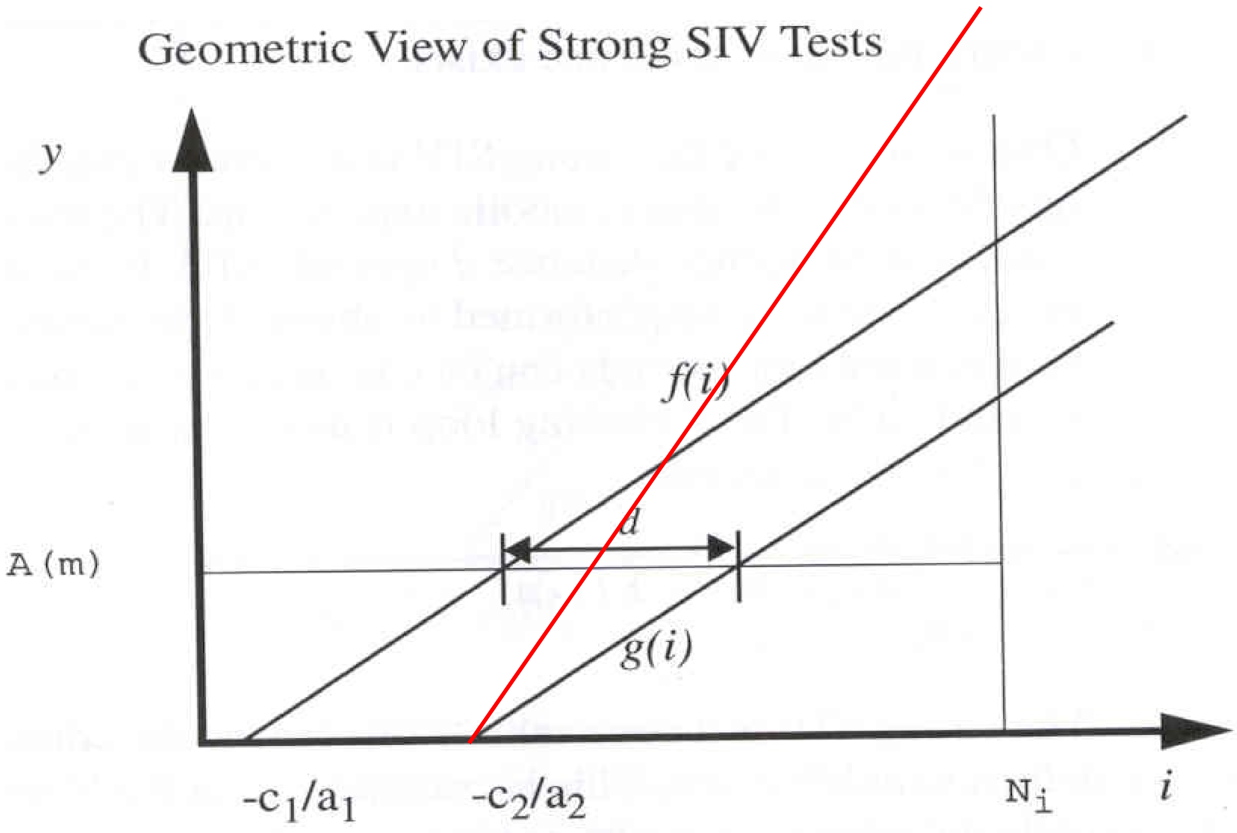
- For example the following are weak SIV subscripts

$$\langle i + 1, 5 \rangle$$

$$\langle 2i + 1, i + 5 \rangle$$

$$\langle 2i + 1, -2i \rangle$$

Geometric view of weak SIV

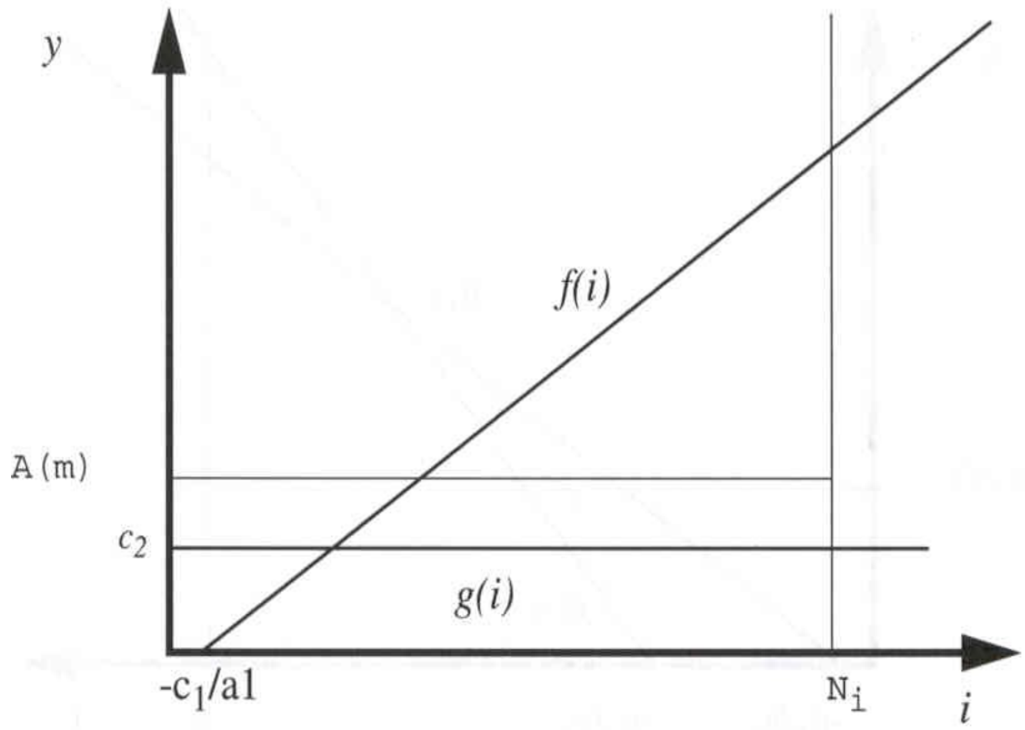


Weak-zero SIV Test

- Special case of Weak SIV where one of the coefficients of the index is zero. i.e., one of the references is always to the same location
- The test consists merely of checking whether the solution is an integer and is within loop bounds
  and, L ≤ i ≤ U

Weak-zero SIV Test

Geometric View of Weak-zero SIV Subscripts



Weak-zero SIV & Loop Peeling

```
      DO i = 1, N
S1        Y(i, N) = Y(1, N) + Y(N, N)
      ENDDO
```

subscript pairs:

Weak-zero SIV & Loop Peeling

```
      DO i = 1, N
S1        Y(i, N) = Y(1, N) + Y(N, N)
      ENDDO
```

Can be loop peeled to…

```
      Y(1, N) = Y(1, N) + Y(N, N)
      DO i = 2, N-1
S1        Y(i, N) = Y(1, N) + Y(N, N)
      ENDDO
      Y(N, N) = Y(1, N) + Y(N, N)
```

Weak-crossing SIV Test

- Special case of Weak SIV where the coefficients of the index are equal in magnitude but opposite in sign
- The test consists merely of checking whether the solution index,

  is 1. within loop bounds and is

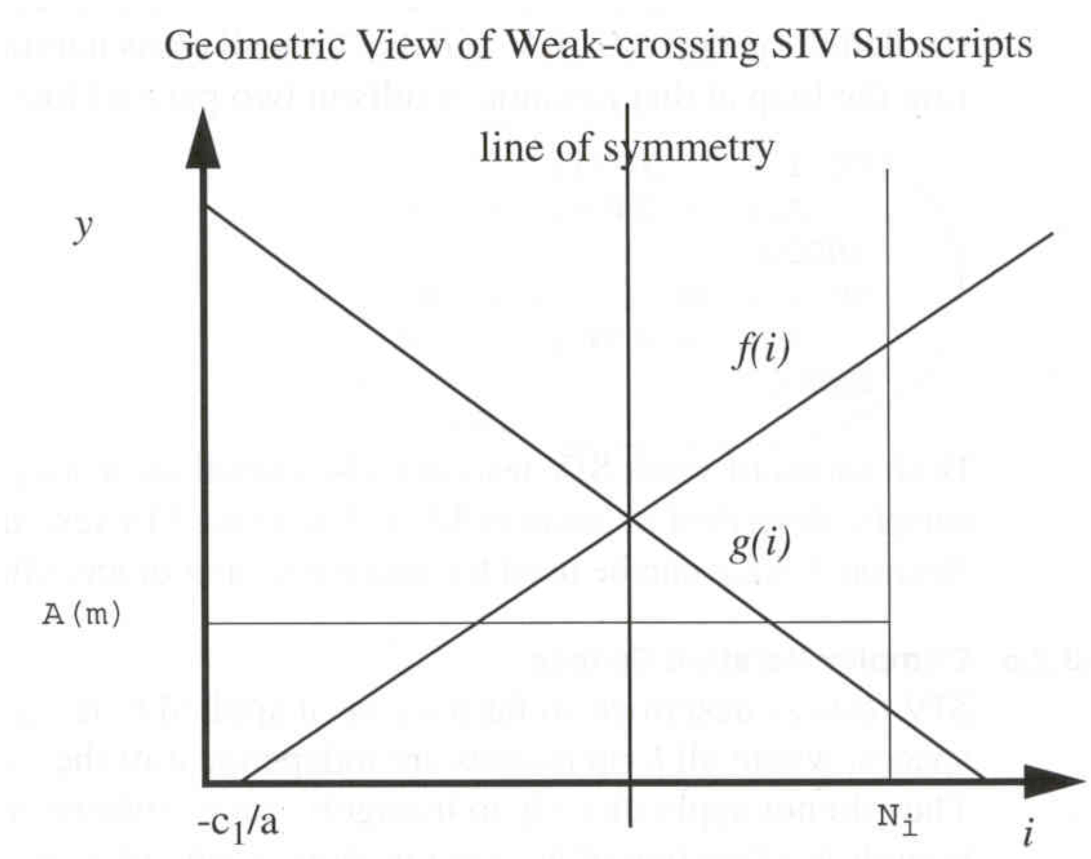
  2. either an integer or has a non-integer part equal to 1/2

- Special case of Weak SIV where the coefficients of the index are equal in magnitude but opposite in sign
- The test consists merely of checking whether the solution index, $i = \frac{c_2 - c_1}{2a}$

  is 1. within loop bounds and is

  2. either an integer or has a non-integer part equal to 1/2

Weak-crossing SIV Test
$\langle ai + c_1, -ai + c_2 \rangle$



Weak-crossing SIV & Loop Splitting

```
      DO i = 1, N
S1    A(i) = A(N-i+1) + C
      ENDDO
```

This loop can be split into…

```
      DO i = 1,(N+1)/2
         A(i) = A(N-i+1) + C
      ENDDO
      DO i = (N+1)/2 + 1, N
         A(i) = A(N-i+1) + C
      ENDDO
```

Breaking Conditions

- Consider the following example

```
      DO I = 1, L
S1          A(I + N) = A(I) + B
      ENDDO
```

- If `L<=N`, then there is no dependence from S_1 to itself
- `L<=N` is called the Breaking Condition

Using Breaking Conditions

- Using breaking conditions then can generate alternative code if it would help

```
      IF (L<=N) THEN
         A(N+1:N+L) = A(1:L) + B
      ELSE
         DO I = 1, L
S1             A(I + N) = A(I) + B
         ENDDO
      ENDIF
```

Index Set Splitting

```
DO I = 1,100
    DO J = 1, I
S1      A(J+20) = A(J) + B
    ENDDO
ENDDO
```

For values of $I < \frac{|d| - (U_0 - L_0)}{U_1 - L_1} = \frac{20 - (-1)}{1} = 21$

there is no dependence

Index Set Splitting

- This condition can be used to create a part of the loop that is independent

```
       DO I = 1,20
          DO J = 1, I
S1a          A(J+20) = A(J) + B
          ENDDO
       ENDDO
       DO I = 21,100
          DO J = 1, Ix
S1b          A(J+20) = A(J) + B
          ENDDO
       ENDDO
```

Now the inner loop for the first nest is independent.

How are we doing so far?

- Empirical study froom Goff, Kennedy, & Tseng
  - Look at how often independence and exact dependence information is found in 4 suites of fortran programs
  - Compare ZIV, SIV (strong, weak-0, weak-crossing, exact), MIV, Delta
  - Check usefulness of symbolic analysis
- ZIV used 44% of time and proves 85% of indep
- Strong-SIV used 33% of time and proves 5% (success per application 97%)
- S-SIV, 0-SIV, x-SIV used 41%
- MIV used only 5% of time
- Delta used 8% of time, proves 5% of indep
- Coupled subscripts rare (20% overall, but concentrated)

Merging Results

- After we test all subscripts we have vectors for each partition. Now we need to merge these into a set of direction vectors for the memory reference
- Since we partitioned into separable sets we can do cross-product of vectors from each partition.
- Start with a single vector = (*,*,...,*) of length depth of loop nest.
- Foreach parition, for each index involved in vector create new set from

  old vector-these_indicies x this set

Example Merge

For I

For J

S_1 A[J-1] = …

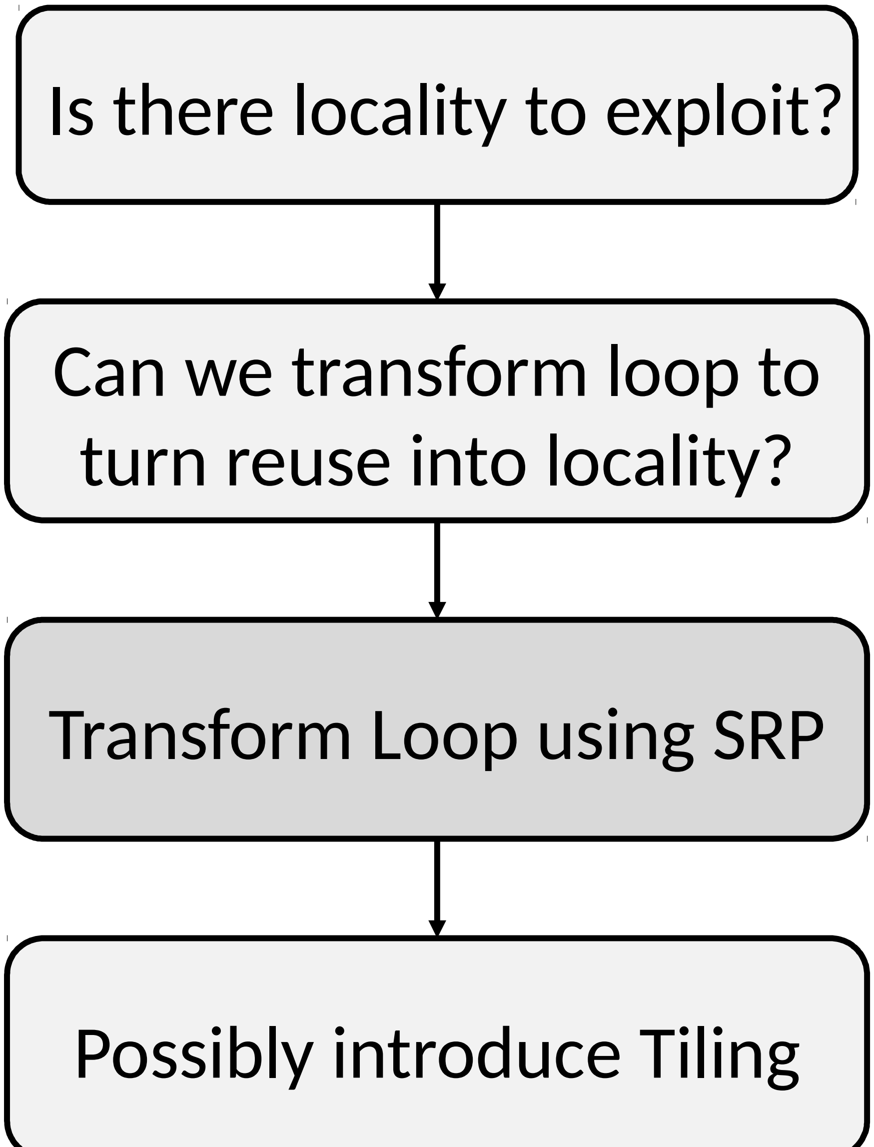
S_2 … = A[J]

For subscript in A using S_1 as source and S_2 as target: J has DV of -1

Merge -1 into (*,*) -> (*,-1). What does this mean?

- (<,-1): true dep in outer loop
- (=,-1): anti-dep from S_2 to S_1 ▭ (=,1)
- (>,-1): anti-dep from S_2 to S_1 in outer loop ▭ (<,-1)

Our Goal: Increase locality



Use Reuse Analysis to determine amount of possible reuse.

Use dependence information to determine pace of possible transformations.

Perform unimodular transformations.

turn n-deep into 2n-deep

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 I_1 := …
    for I_2 := …
     …
        for I_n := …
            body
```

→

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

- Legal if permutation on dependence vector is legal

Transform and matrix notation

- If dependences are vectors in iteration 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:
  Diagonal matrix with i[th] element = -1.

- For 2 deep loop, reversal of innermost is:

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

Skewing

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

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

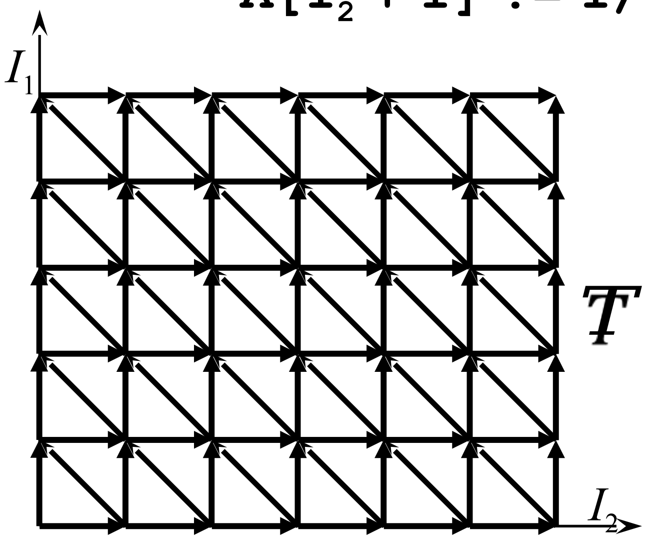
- Example for 2D

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

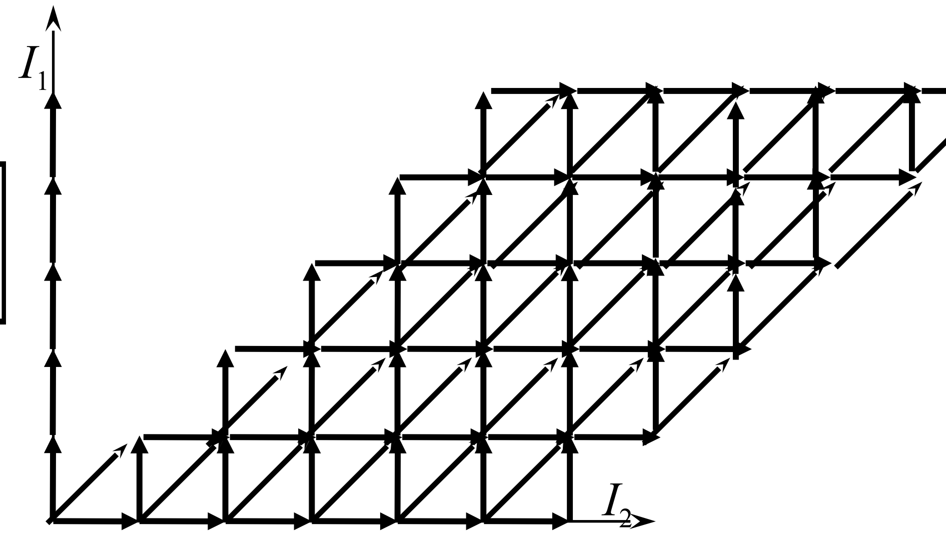
Loop Skewing Example

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

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



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




```
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)}

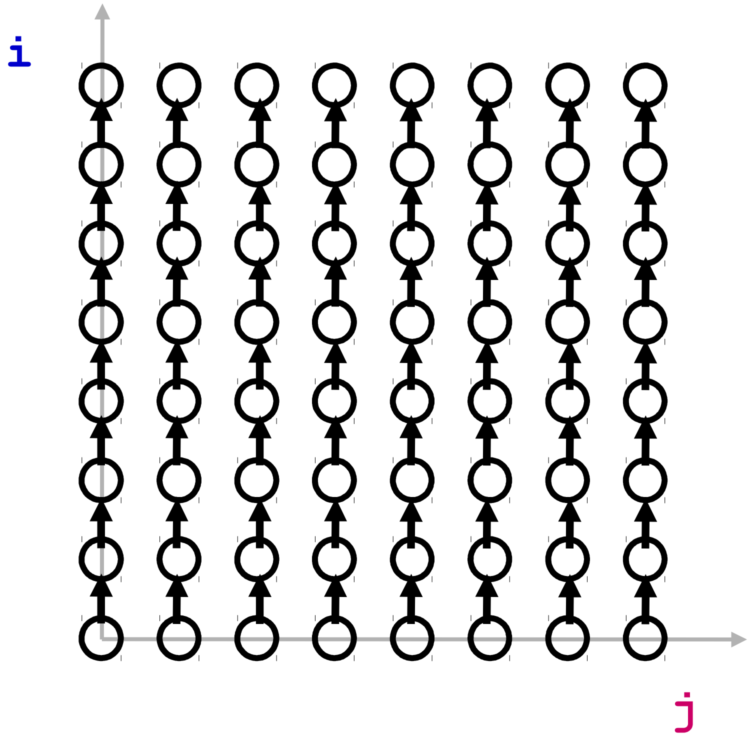
Legal Transformations

- 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?

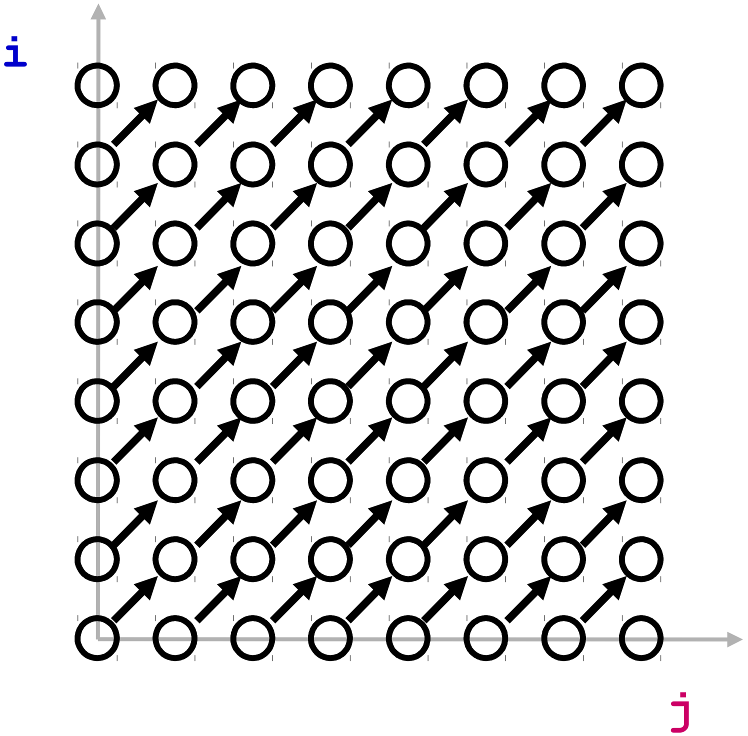
```
for i = 0 to N-1
    for j = 0 to N-1
             A[i+1][j] += A[i]
[j];
```



But...is the transform legal?

- Loop reversal ok?
- Loop interchange ok?

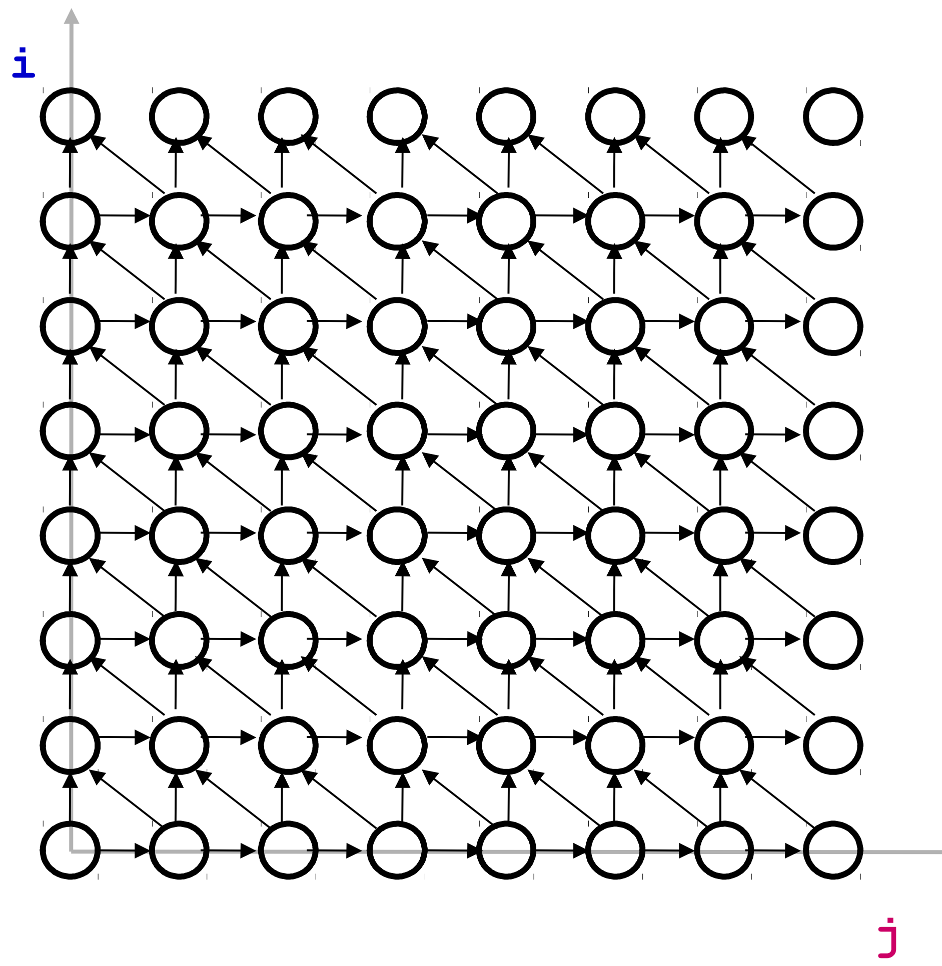
```
for i = 0 to N-1
    for j = 0 to N-1
            A[i+1][j+1] += A[i]
[j];
```



But...is the transform legal?

- What other visit order is legal here?

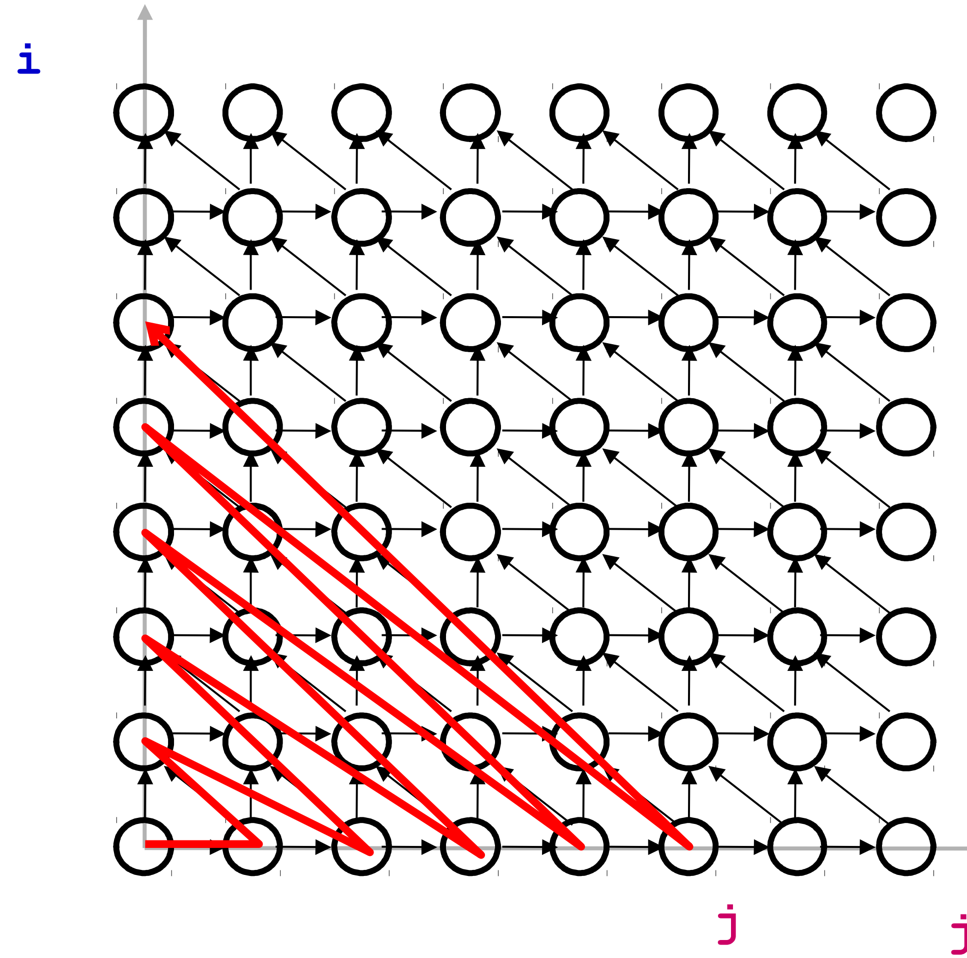
```
for i = 0 to TS
  for j = 0 to N-2
    A[j+1] =
      (A[j] + A[j+1] + A[j+2])/3;
```



But...is the transform legal?

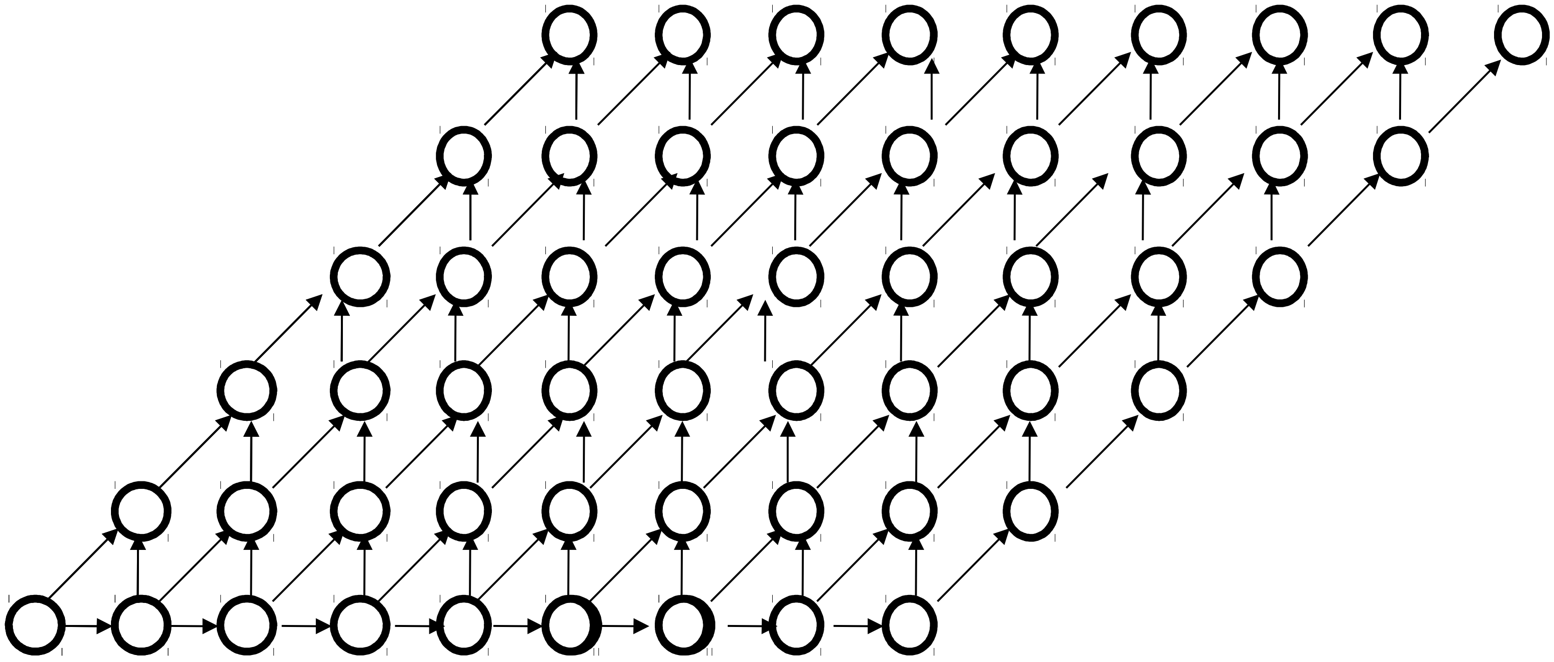
- What other visit order is legal here?

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



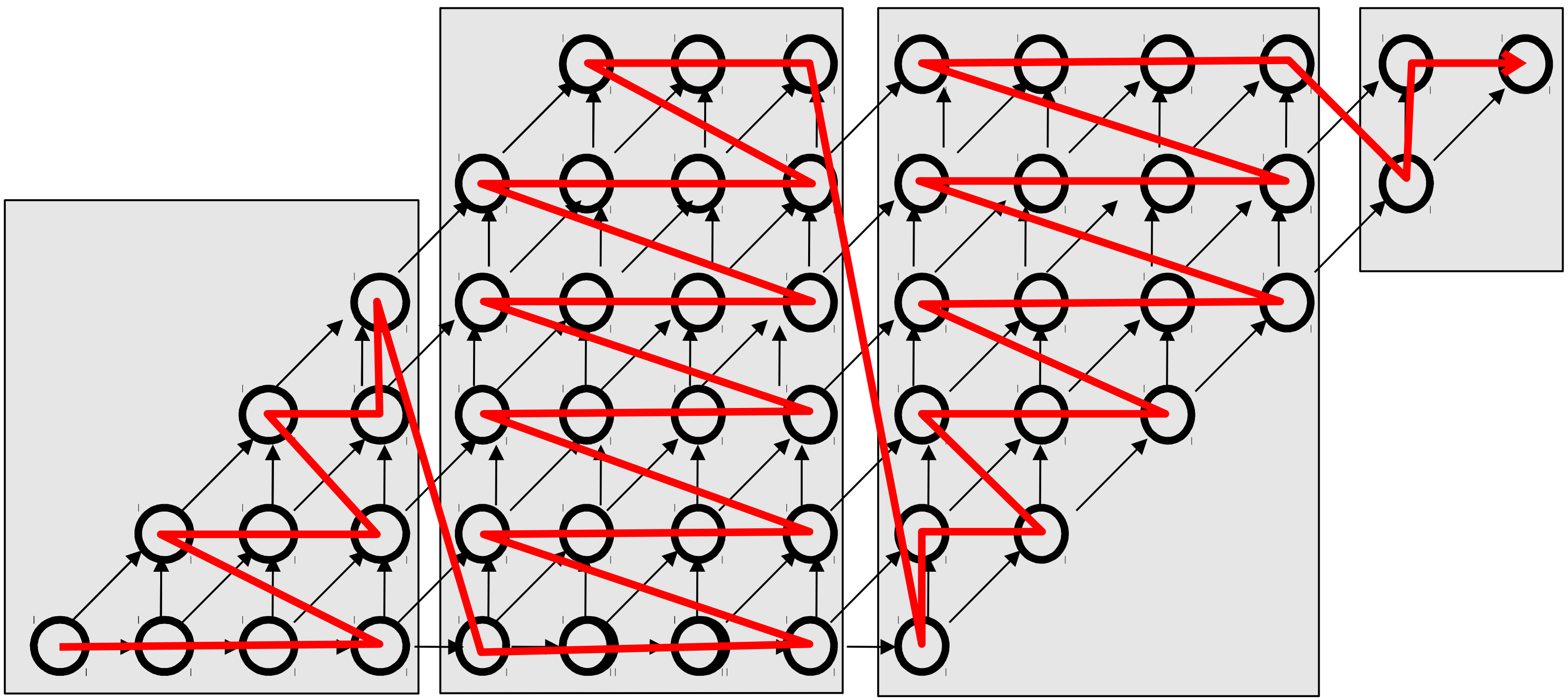
But…is the transform legal?

- Skewing…



But...is the transform legal?

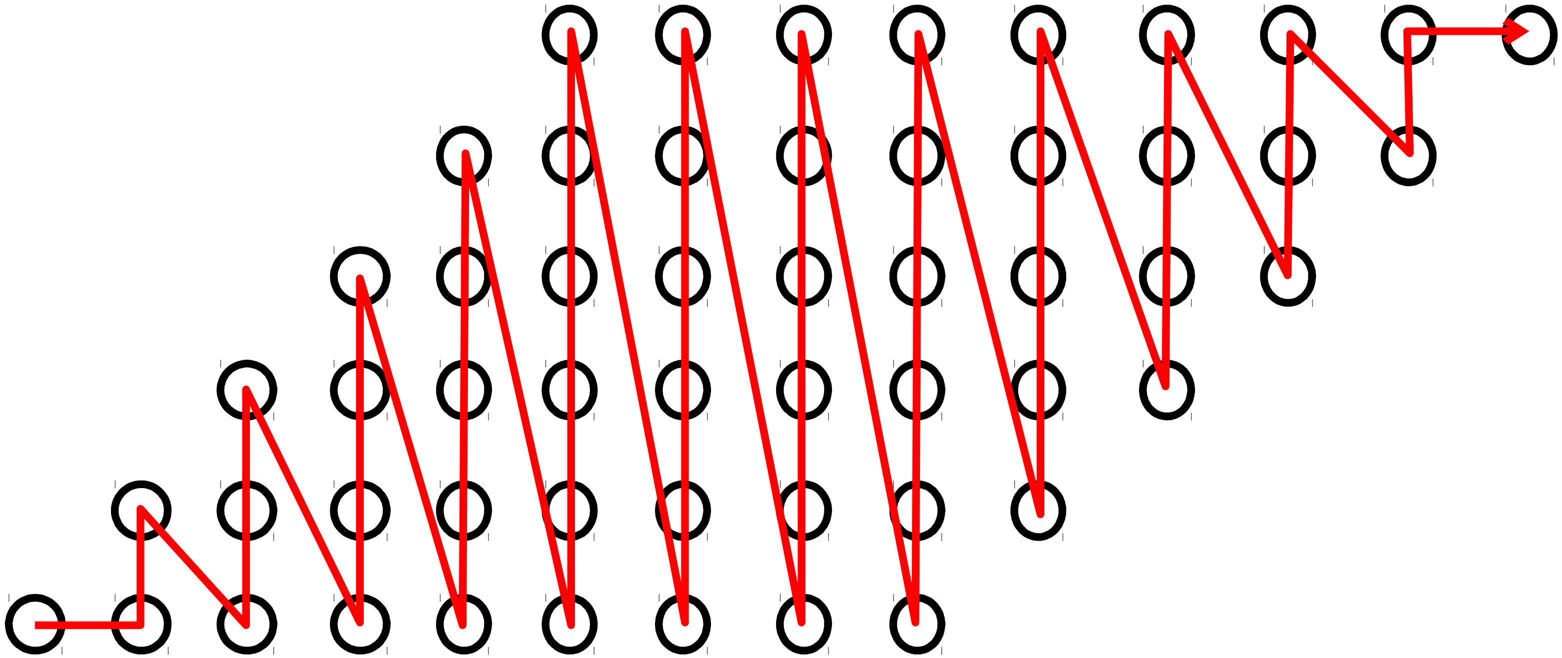
- Skewing...now we can block



We have made the inner loop, Fully Permutable

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

SRP

- Extract Dependence Information
- Extract Locality Information
- Search Possible Transformation Space for most Locality

Searching the Space

```
for I_1 := 0 to 5
  for I_2 := 0 to 6
    A[I_2 + 1] = 1/3 * (A[I_2]+ A[I_2 + 1] + A[I_2 + 2])
```

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

Uniformly Generated Set:

{A[I_2], A[I_2+1], A[I_2+2]} $H = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Original Loop:

Type	reuse space	reuse factor
Self-Temporal:	Ker(H) = span{(1,0)}	s
Self-Spatial:	Ker(H_s) = span{(1,0),(0,1)}	L
Group-Temporal:	span{(1,0),(0,1)}	3

Possible Transformations

- span{(0,1)} T= $\begin{pmatrix} 1 & 0 \\ & 1/L \\ 0 & 1 \end{pmatrix}$

- span{(1,0)} illegal

- span{(1,0),(0,1)} T= $\begin{pmatrix} 1 & 0 \\ 1/(sL) & \\ 1 & 1 \end{pmatrix}$

SRP

- Extract Dependence Information
- Extract Locality Information
- Search Possible Transformation Space for most Locality
- Transform Loop using T
  - rewrite index expressions
  - rewrite bounds
- If Neccesary, Tile