Loop Optimization - 1

15-411/15-611 Compiler Design

Seth Copen Goldstein

March 27, 2025

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**

Requires understanding data dependencies

Data-Dependent Loop Transformations

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

Loop Transformation Theory

Plan

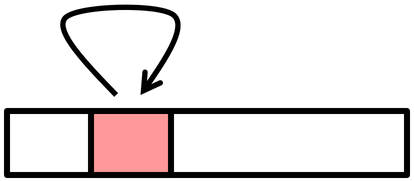
- Review Locality
- Iteration spaces
- Dependency analysis
- Transformations
  - interchange
  - reversal
  - skewing
  - Tiling
- A Data Locality Optimizing Algorithm, Wolf&Lam
- Automatic Vectorization

Today

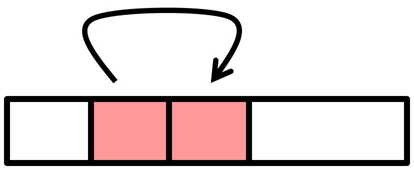
- Review Locality
- Iteration spaces
- Dependency analysis
- Transformations
  - interchange
  - reversal
  - skewing
  - Tiling
- A Data Locality Optimizing Algorithm, Wolf&Lam
- Automatic Vectorization

Recall: Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

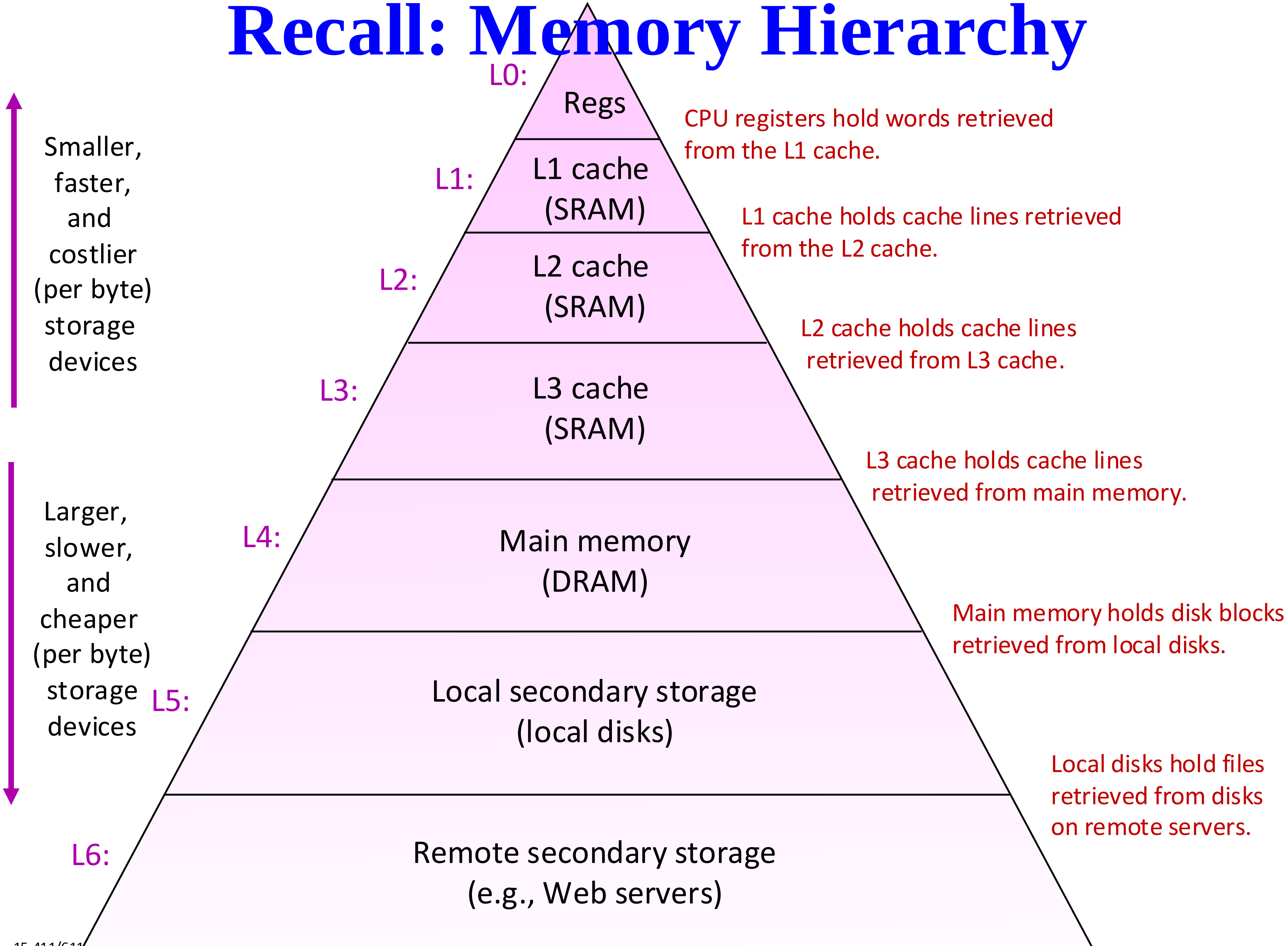


- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

Recall: Memory Hierarchy



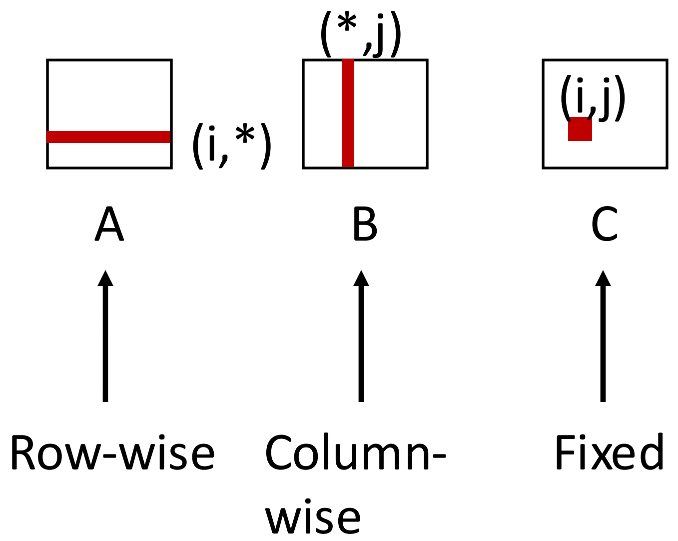
Layout of C Arrays in Memory

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - ```
    for (i = 0; i < N; i++)
      sum += a[0][i];
    ```
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- Stepping through rows in one column:
  - ```
    for (i = 0; i < n; i++)
      sum += a[i][0];
    ```
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}                                     matmult/mm.c
```

Inner loop:



Miss rate for inner loop iterations:

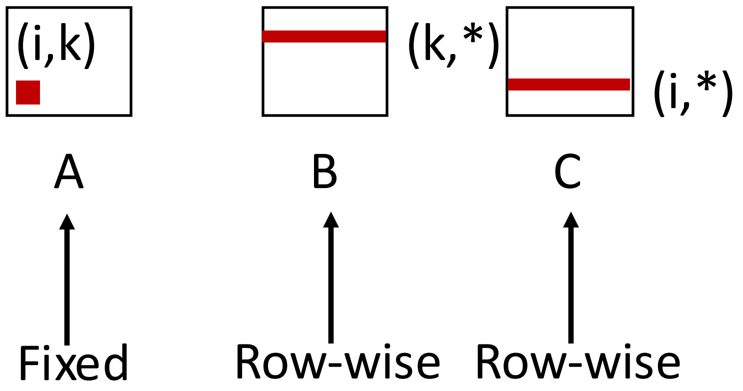
A	B	C
1/L	1.0	0.0

L = # of elements per cache line

Matrix Multiplication (`kij`)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
                        matmult/mm.c
```

Inner loop:



Miss rate for inner loop iterations:

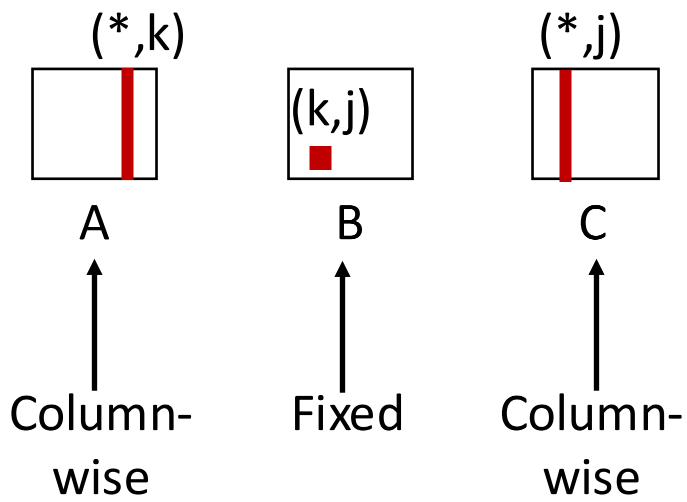
A	B	C
0.0	1/L	1/L

L = # of elements per cache line

Matrix Multiplication (`jki`)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```
matmult/mm.c

Inner loop:



<u>Miss rate for inner loop iterations:</u>

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

L = # of elements per cache line

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

`ijk` (& `jik`):
- 2 loads, 0 stores
- avg misses/iter = 1+1/L

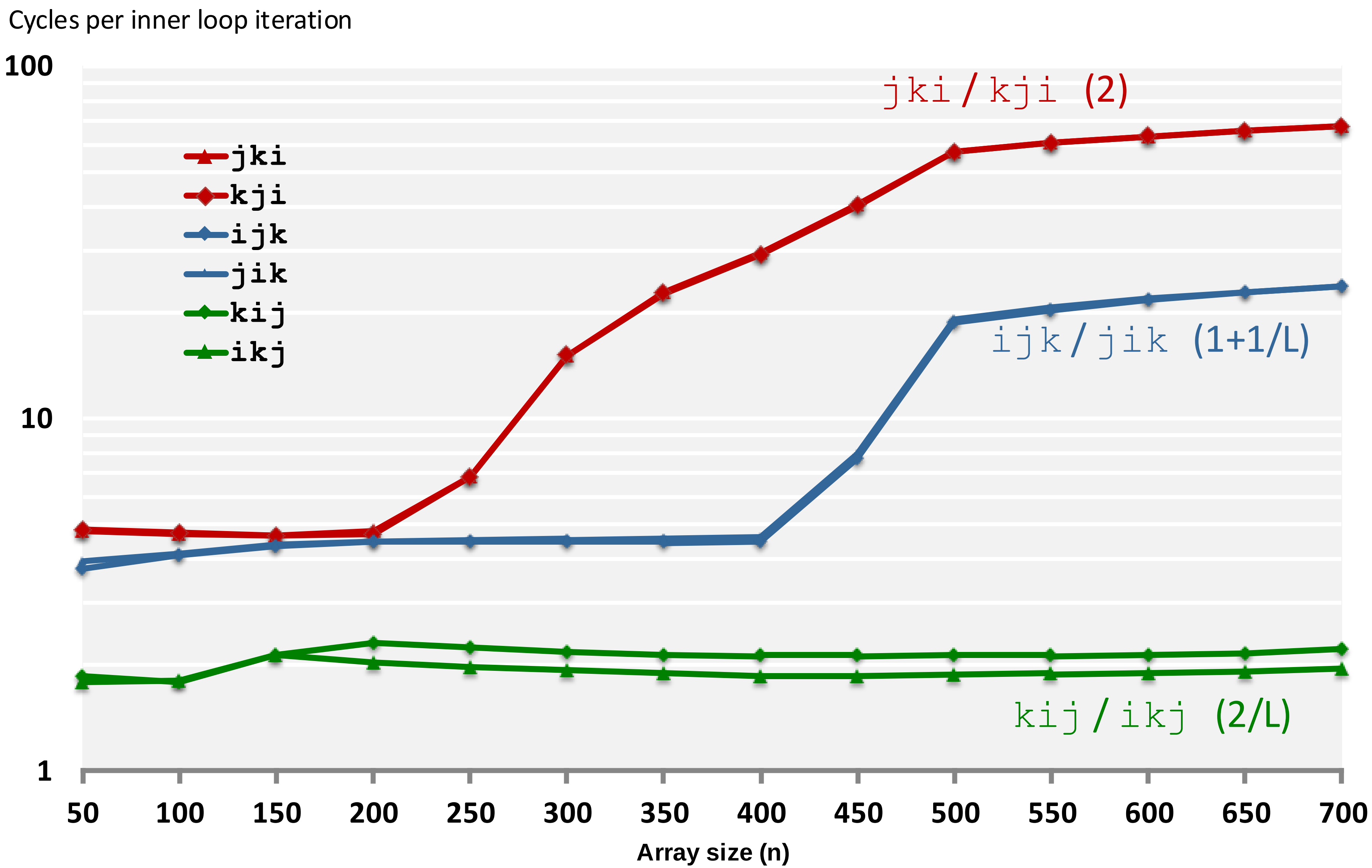
```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

`kij` (& `ikj`):
- 2 loads, 1 store
- avg misses/iter = 2/L

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

`jki` (& `kji`):
- 2 loads, 1 store
- avg misses/iter = 2

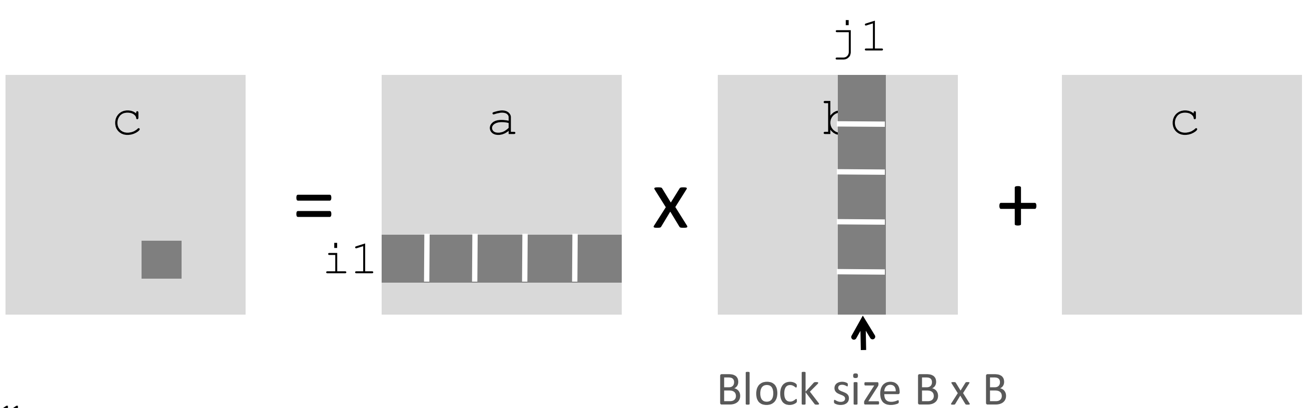
Core i7 Matrix Multiply Performance



Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```
matmult/bmm.c



Blocking Summary

- No blocking: (9/8) n^3 misses
- Blocking: (1/(4B)) n^3 misses
- Use largest block B, such that B satisfies $3B^2 < C$
  - Fit three blocks in cache! Two input, one output.
- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly

Or, compiled properly!

The Problem

- How to increase locality by transforming loop nest
- Matrix Mult is simple as it is both
  - legal to tile
  - advantageous to tile
- Can we determine the benefit?
  (reuse vector space and locality vector space)
- Is it legal (and if so, how) to transform loop?
  (unimodular transformations)

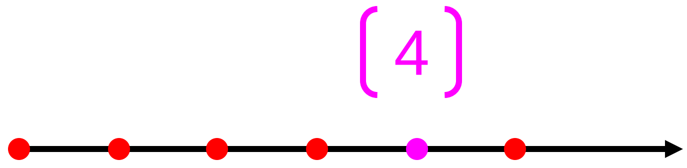
Loop Transformation Theory

- Iteration Space
- Dependence vectors
- Unimodular transformations

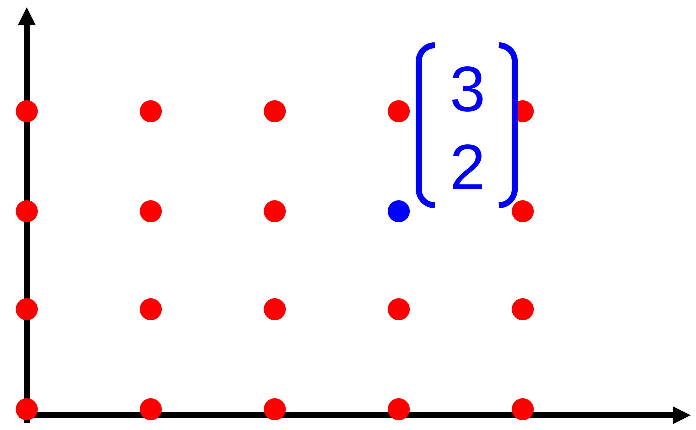
Iteration Space

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++) {
        …
}
```



Loop Nests and the Iter space

- General form of tightly nested loop

```
for I1 := low1 to high1 by step1
    for I2 := low2 to high2 by step2
        ...
          for Ii := lowi to highi by stepi
            ...
              for In := lown to highn by stepn
                  Stmts
```

- The iteration space is a convex polyhedron in $\mathcal{Z}^n$ bounded by the loop bounds.
- Each iteration is a node in the polyhedron identified by its vector: $\mathbf{p}=(p_1, p_2, \ldots, p_n)$

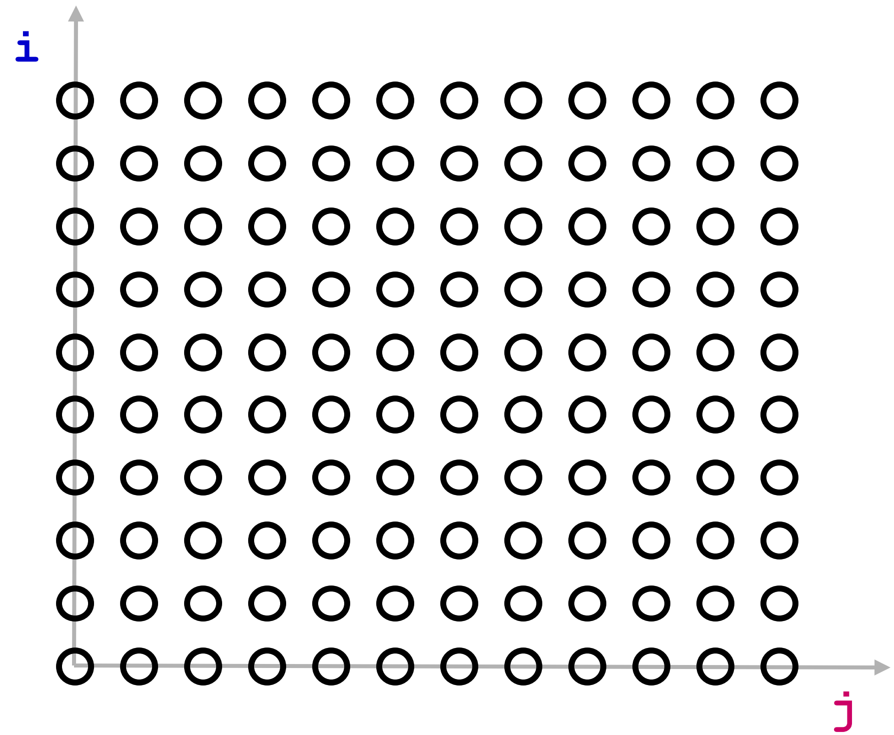
Lexicographical Ordering

- Iterations are executed in lexicographic order.
- for **p**=(p_1, p_2, …, p_n) and **q**=(q_1, q_2, …, q_n)
  if **p** $\succ_k$ **q** iff for $1 \leq k \leq n$,

  $\forall\ 1 \leq i < k$, ($p_i = q_i$) and $p_k > q_k$
- For MM:
  - (1,1,1), (1,1,2), (1,1,3), …,
    (1,2,1), (1,2,2), (1,2,3), …,
    …,
    (2,1,1), (2,1,2), (2,1,3), …
  - (1,2,1) $\succ_2$ (1,1,2), (2,1,1) $\succ_1$ (1,4,2), etc.

Handy Representation: "Iteration Space"

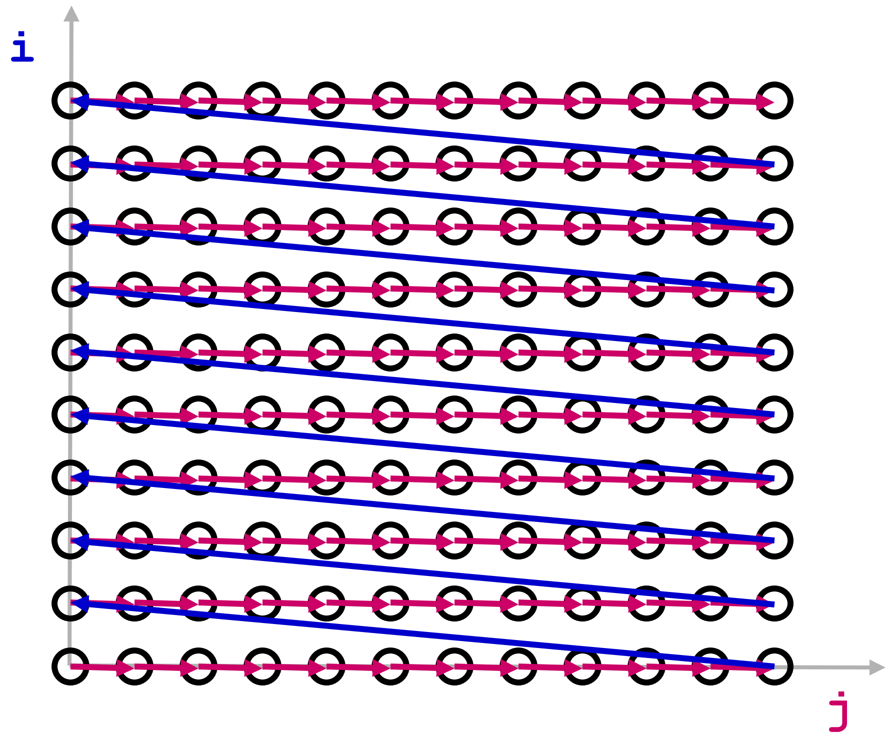
```
for i = 0 to N-1
  for j = 0 to N-1
   A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

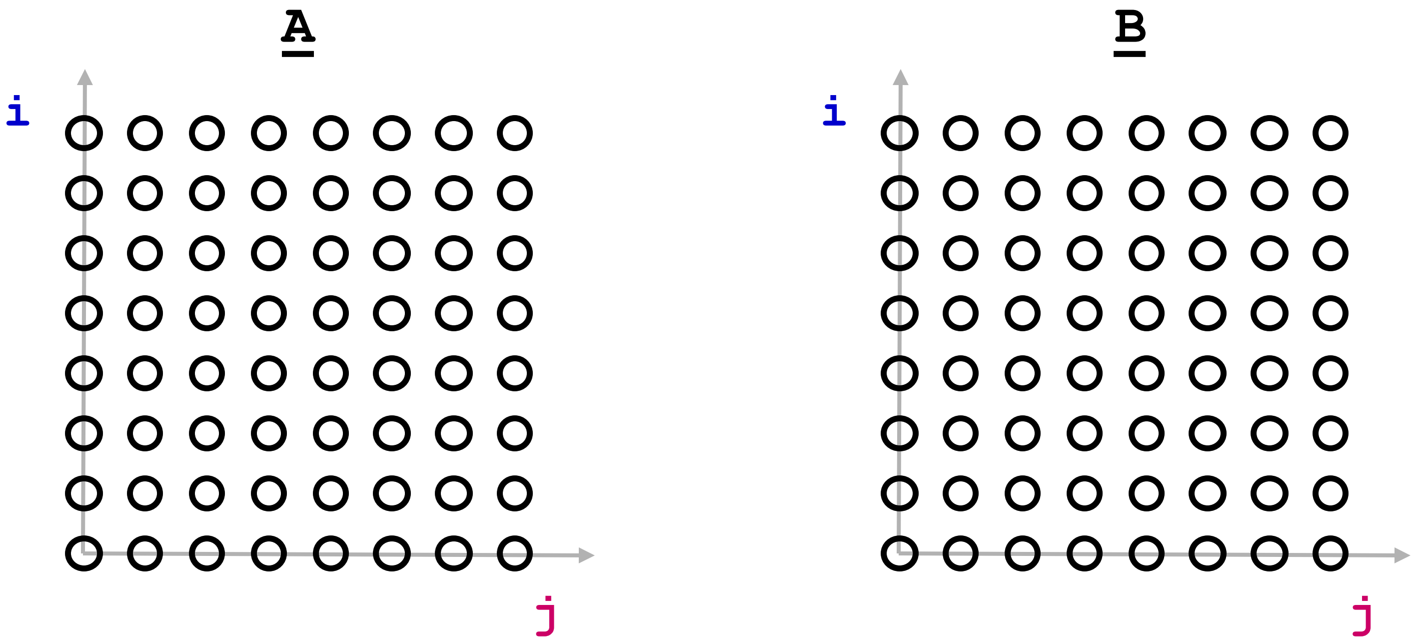
```
for i = 0 to N-1
  for j = 0 to N-1
  A[i][j] = B[j][i];
```



- Note: iteration space is not data space

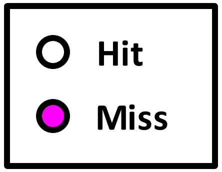
When Do Cache Misses Occur?

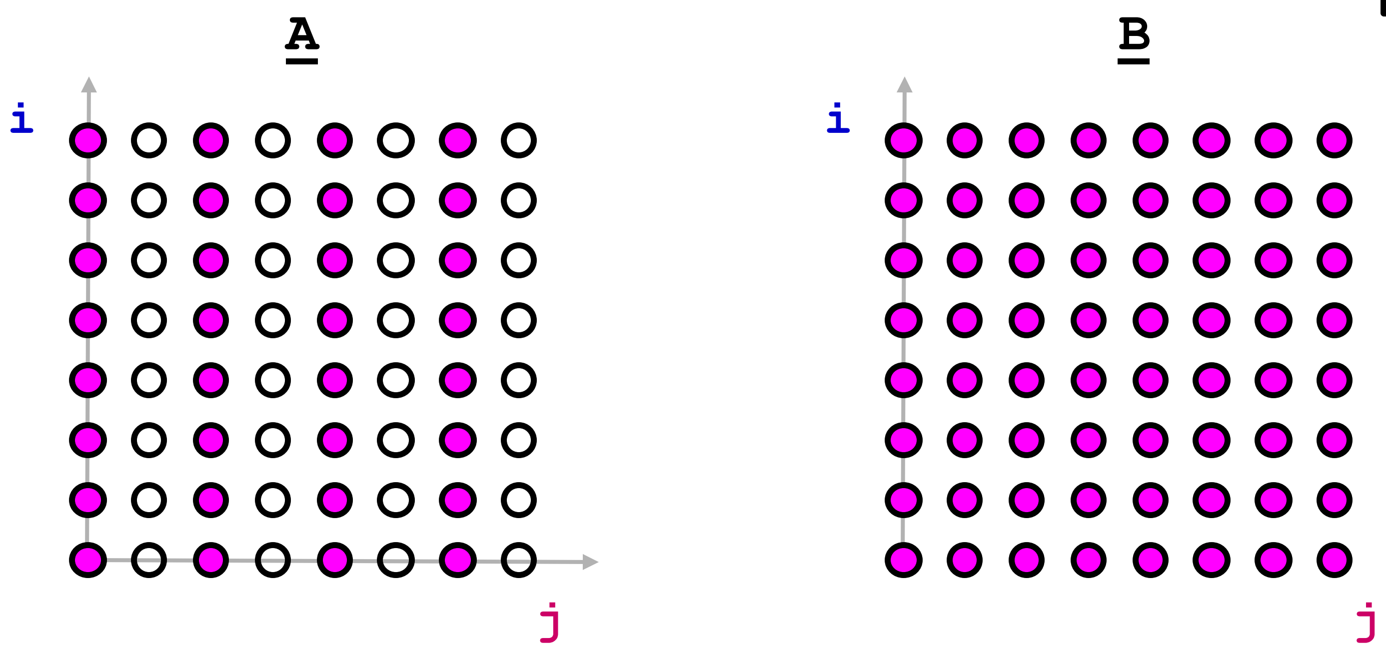
```
for i = 0 to N-1
  for j = 0 to N-1
   A[i][j] = B[j][i];
```



When Do Cache Misses Occur?

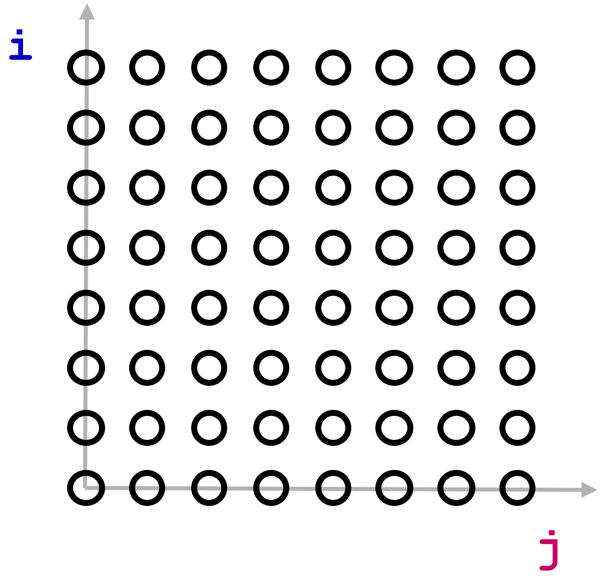
```
for i = 0 to N-1
  for j = 0 to N-1
    A[i][j] = B[j][i];
```





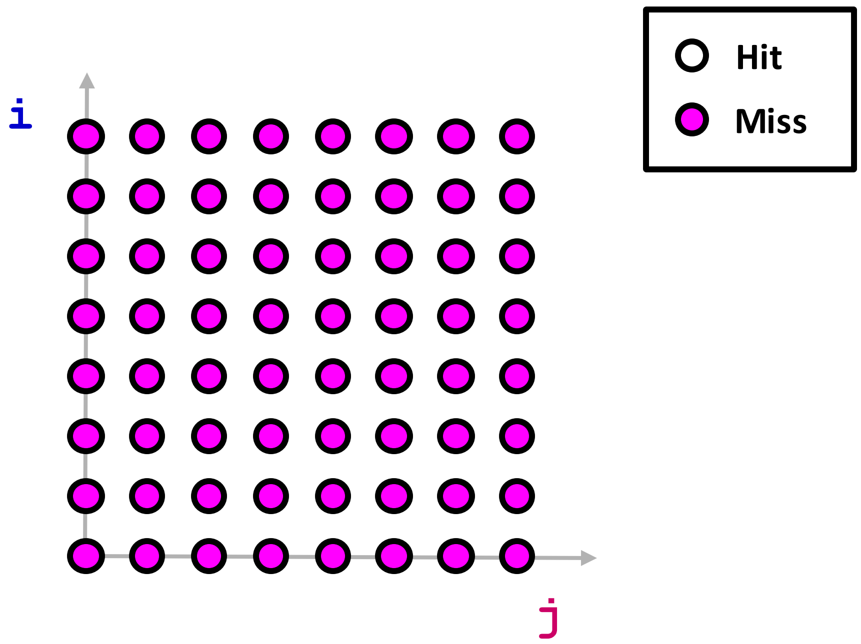
When Do Cache Misses Occur?

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



When Do Cache Misses Occur?

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



Optimizing the Cache Behavior of Array Accesses

- We need to answer the following questions:
  - when do cache misses occur?
    - use "locality analysis"
  - can we transform loop (i.e., change the order of the iterations) to produce better behavior?
    - evaluate the cost of various alternatives
  - does the new ordering/layout still produce correct results?
    - use "dependence analysis"

Examples of Loop Transformations

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

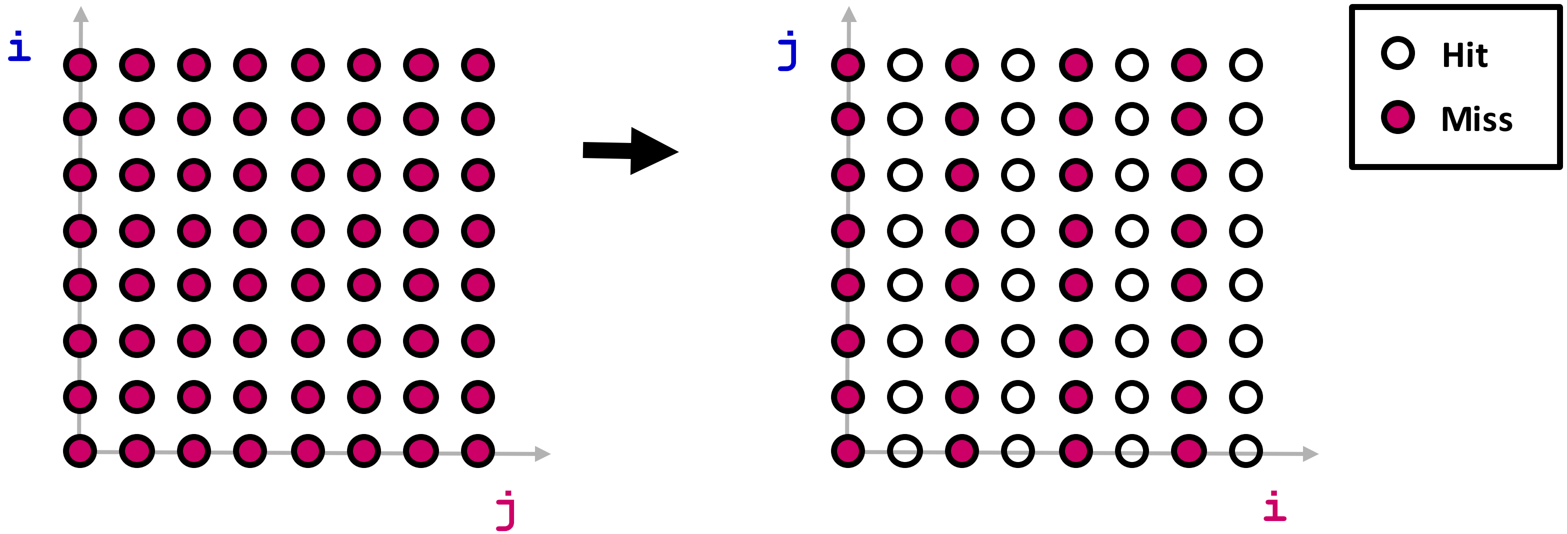
Can improve locality (Loop Interchange, Cache Blocking)

Can enable above (Skewing, Loop Reversal)

Loop Interchange

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

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

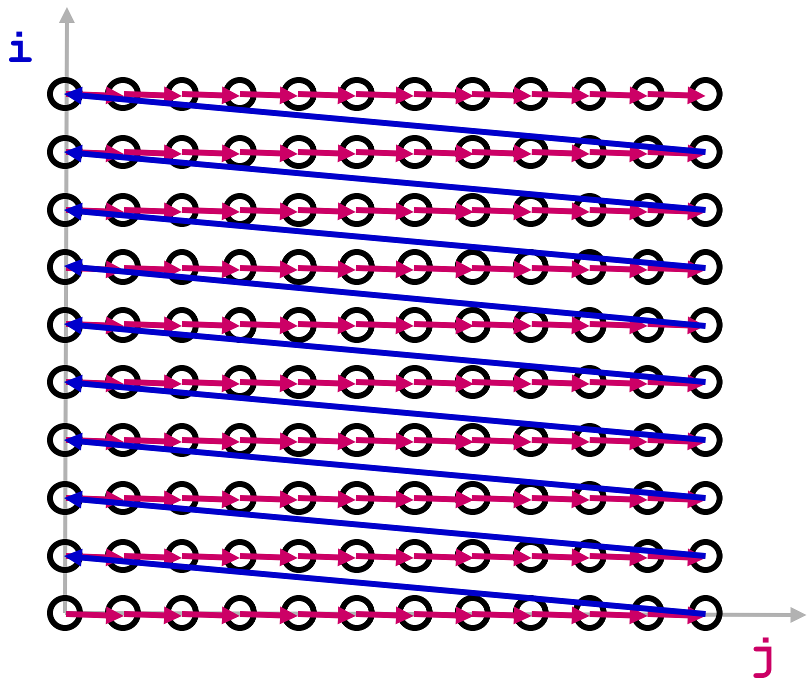


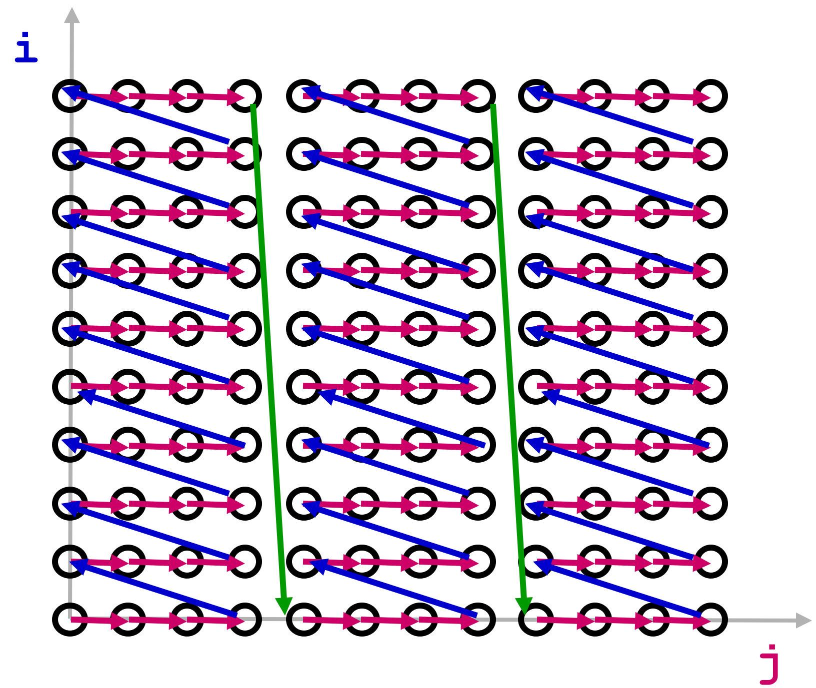
- *(assuming N is large relative to cache size)*

Impact on Visitation Order in Iteration Space

```
for i = 0 to N-1
  for j = 0 to N-1
     f(A[i],A[j]);
```

```
for JJ = 0 to N-1 by B
  for i = 0 to N-1
    for j = JJ to max(N-1,JJ+B-1)
       f(A[i],A[j]);
```





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	W ➔ R	δ^f	} true
• Anti-Dependence	R ➔ W	δ^a	} false
• Output Dependence	W ➔ W	δ^o	

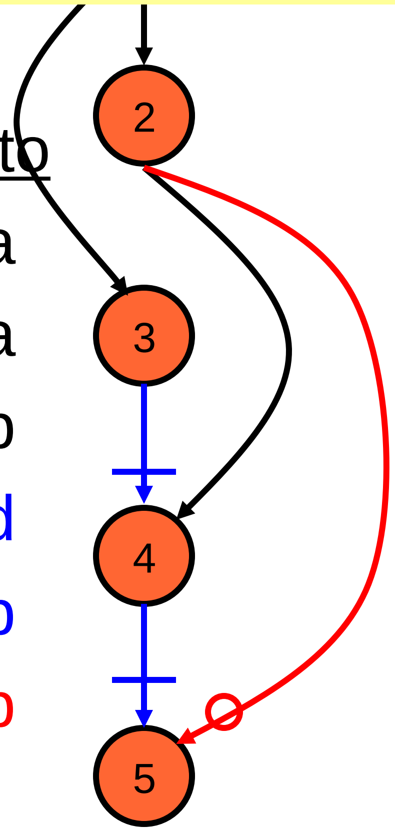
```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

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.

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

δ^f loop carried

δ^f 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

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

i=0

i=1

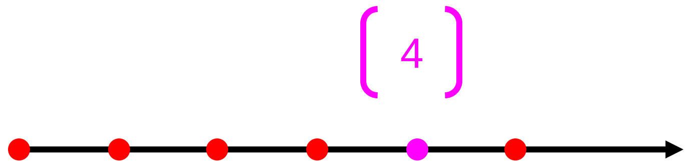
i=2

Distance/Direction of the dependence is also important.

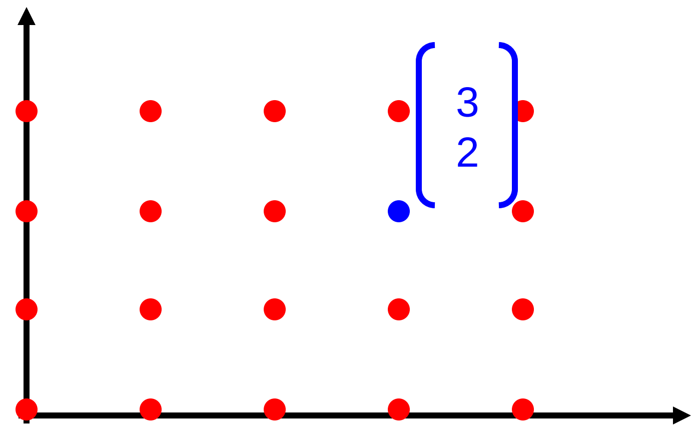
Iteration Space

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.

$$\mathbf{d} = \mathbf{I}_t - \mathbf{I}_s$$

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

$$\mathbf{I}_s + \mathbf{d} = \mathbf{I}_t$$

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] ;
   }
```

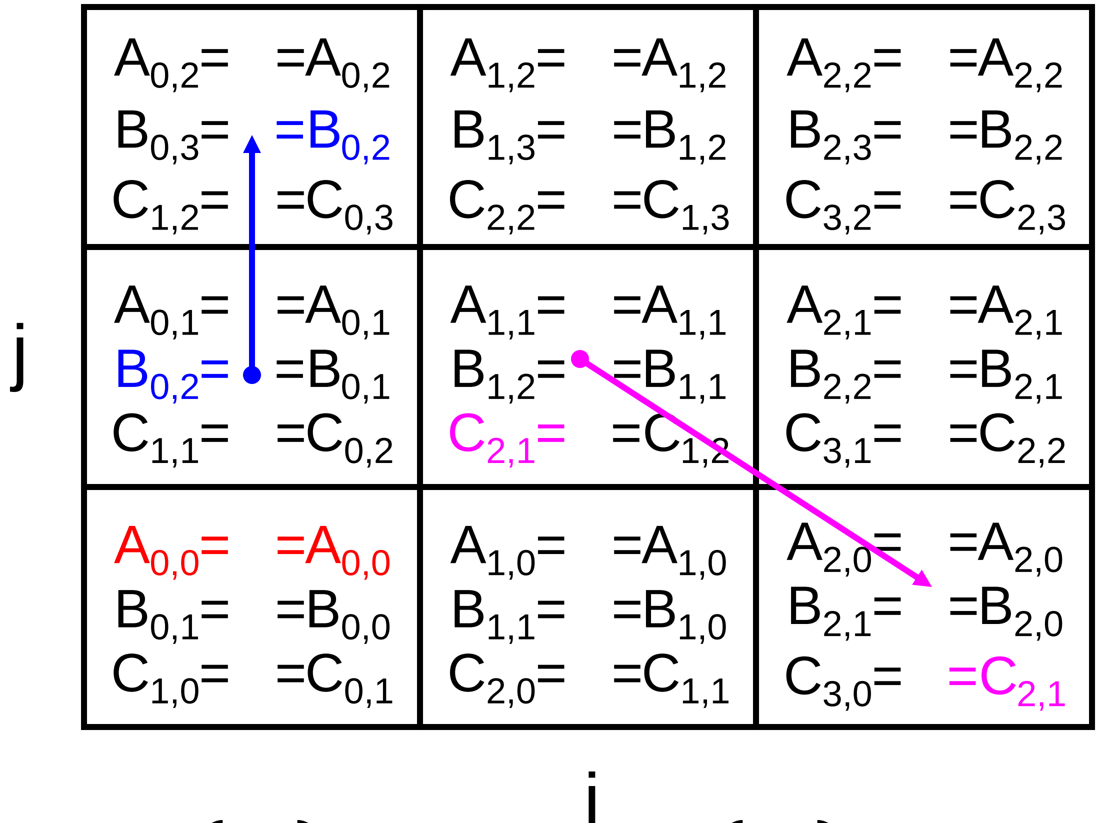
j

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

i

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

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

Eg of Uniformly generated sets

These references all belong to the same uniformly generated set: H = [0 1]

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

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

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

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

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: $\mathbf{d}=(d_1, d_2, \ldots, d_n)$.
- 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
  - $[1, \infty]$ as + or >
  - $[-\infty, -1]$ as – or <
  - $[-\infty, \infty]$ as $\pm$ or *

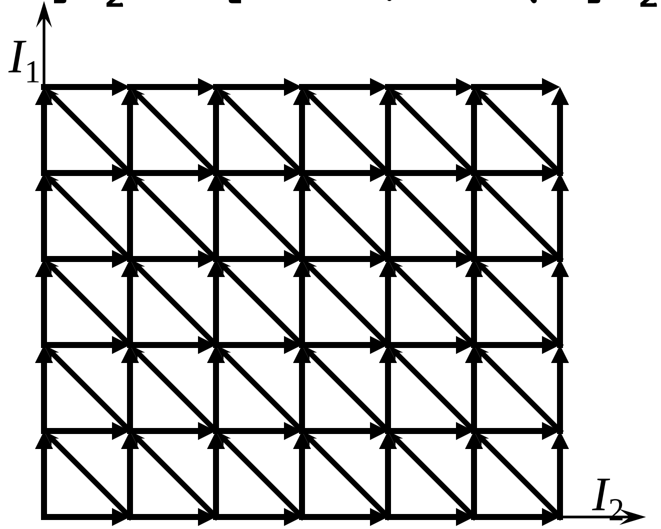
- A distance vector is the difference between the target and source iterations (for a dependent ref), e.g., $\mathbf{d} = \mathbf{I}_t - \mathbf{I}_s$

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)

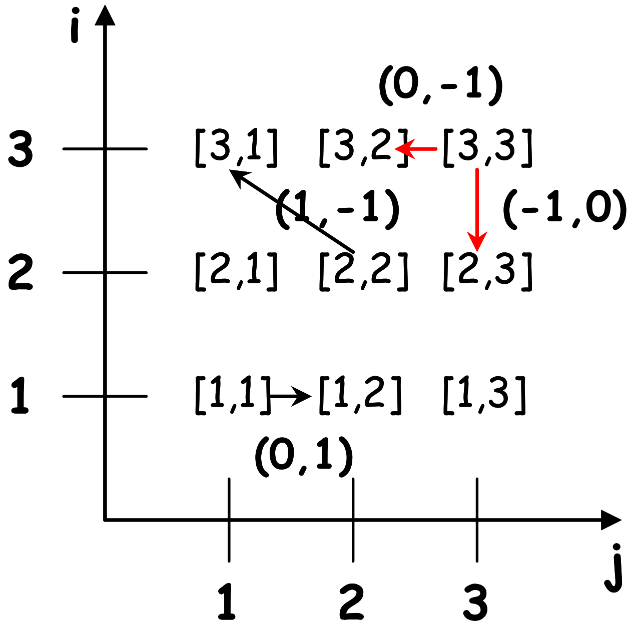
```
for I1 := 0 to 5
  for I2 := 0 to 6
    A[I2 + 1] := 1/3 * (A[I2] + A[I2 + 1] + A[I2 + 2])
```



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

Plausible Dependence vectors

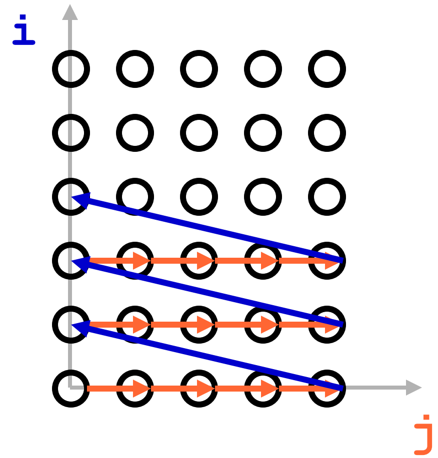
- A dependence vector is plausible iff it is lexicographically non-negative.
- All sequential programs have plausible dependence vectors. Why?
- Plausible: (1,-1)
- implausible (-1,0)



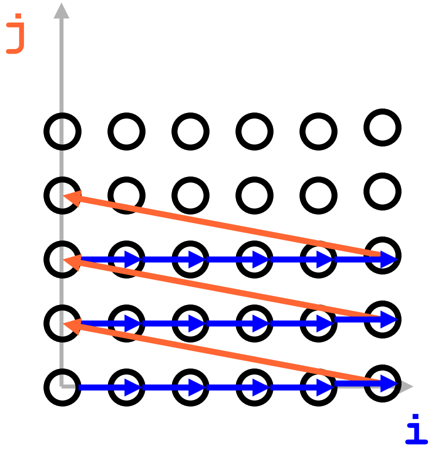
Loop Transforms

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

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

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

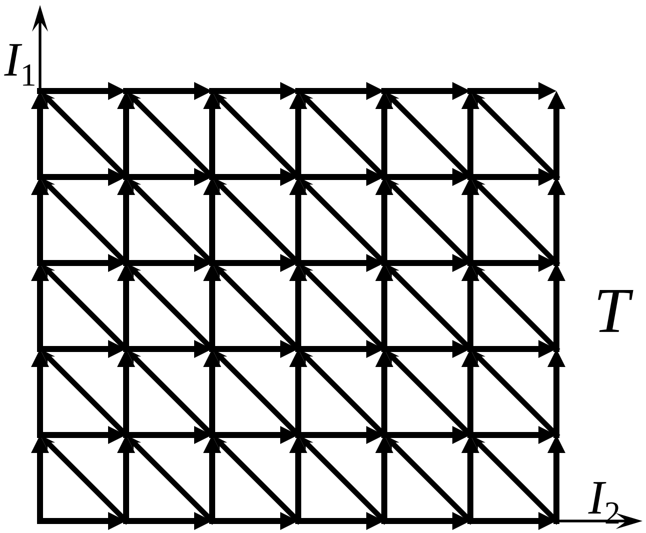
- Example for 2D

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

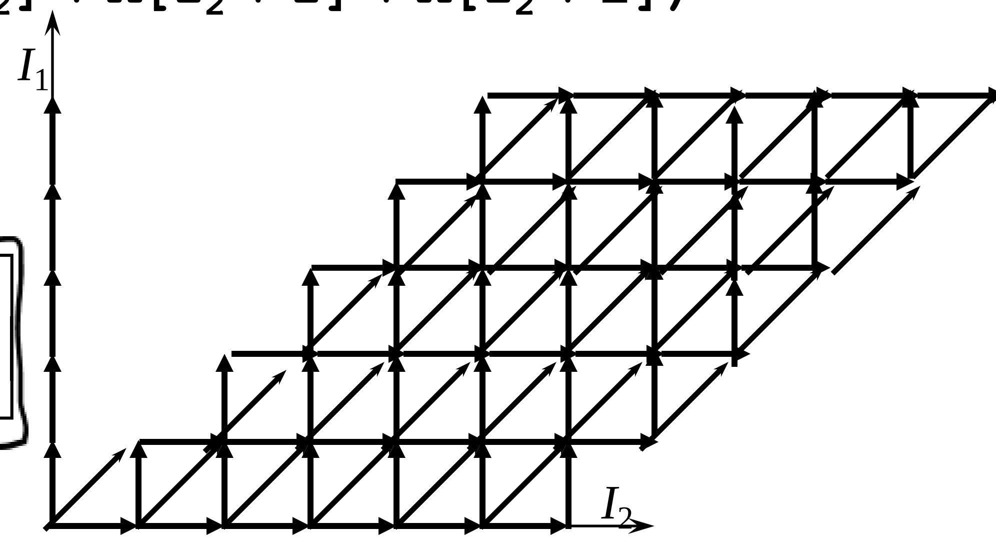
Loop Skewing Example

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

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



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



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

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

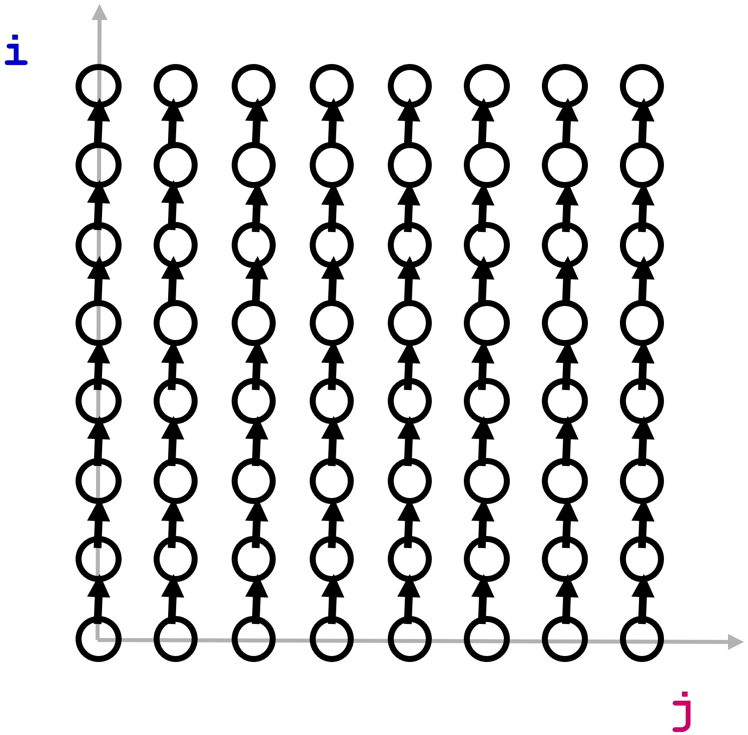
But...is the transform legal?

- 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?

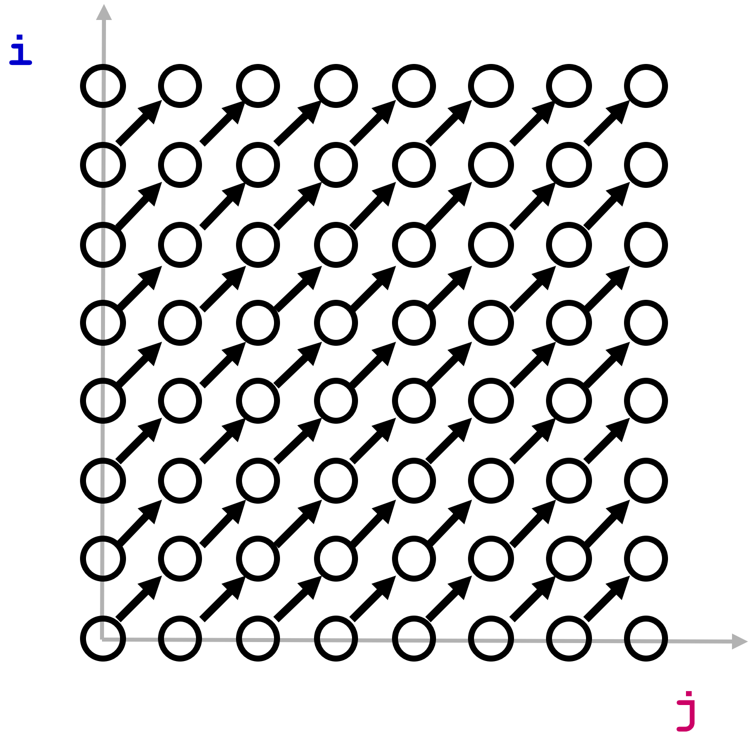
```
for i = 0 to N-1
    for j = 0 to N-1
      A[i+1][j] += A[i][j];
```



But...is the transform legal?

- Loop reversal ok?
- Loop interchange ok?

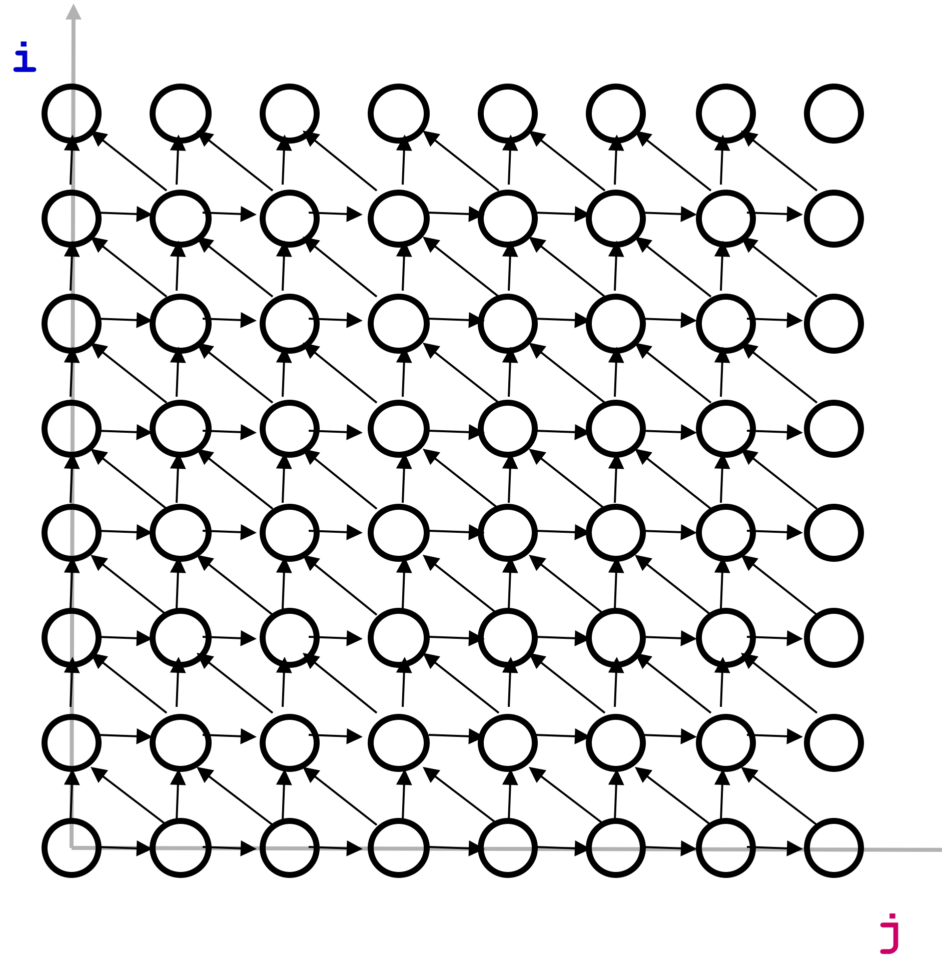
```
for i = 0 to N-1
    for j = 0 to N-1
        A[i+1][j+1] += A[i][j];
```



But...is the transform legal?

- What other visit order is legal here?

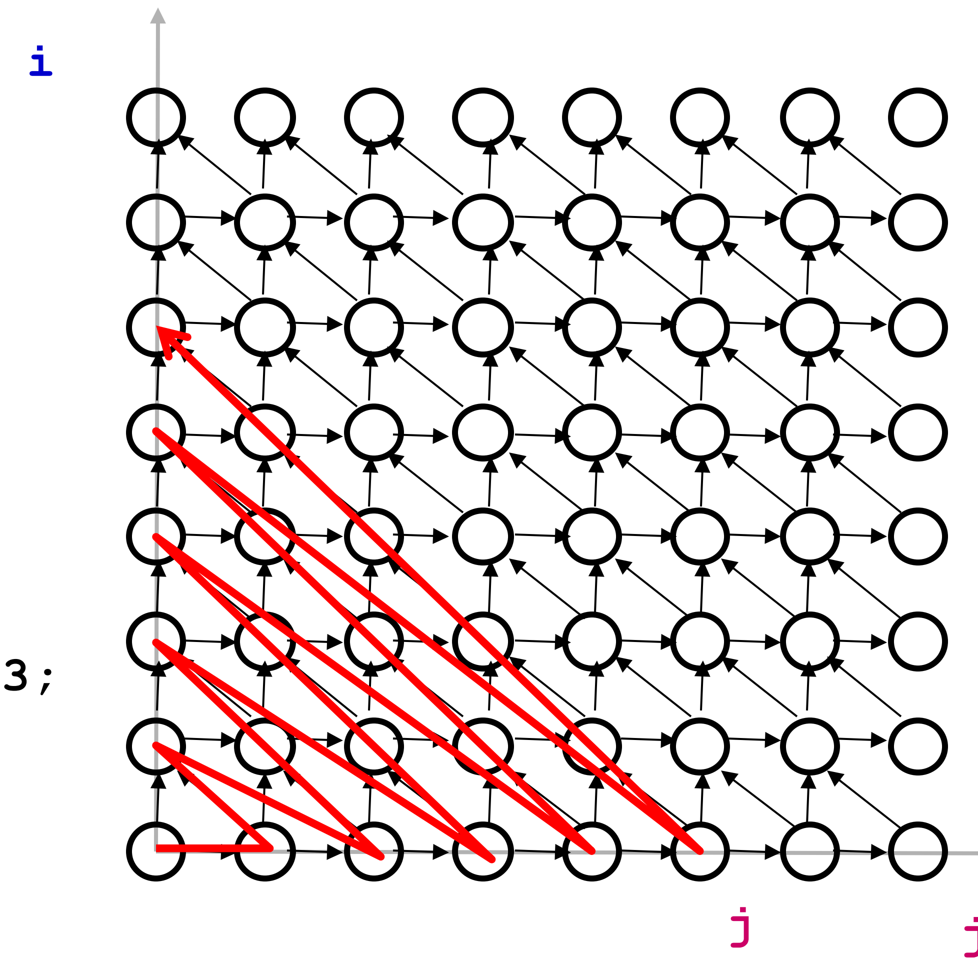
```
for i = 0 to TS
  for j = 0 to N-2
    A[j+1] =
      (A[j] + A[j+1] + A[j+2])/3;
```



But...is the transform legal?

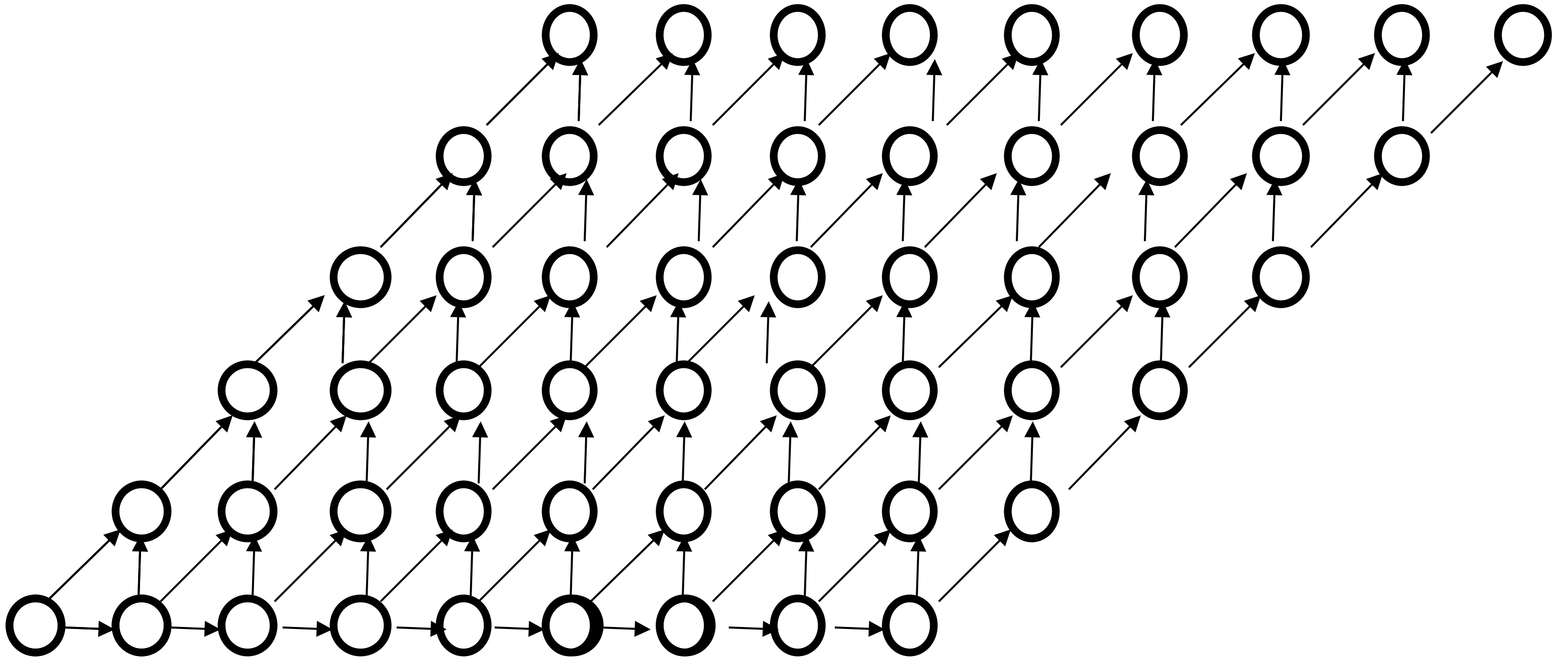
- What other visit order is legal here?

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



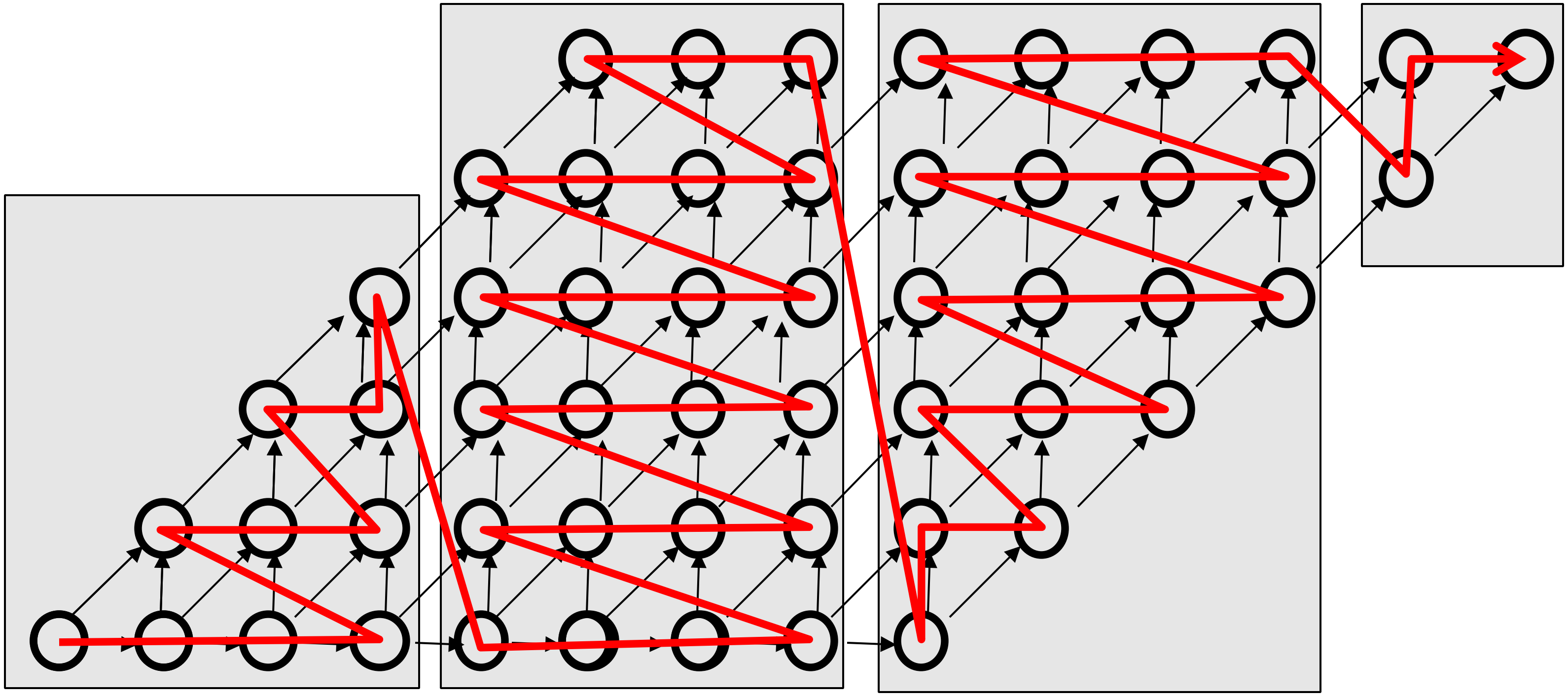
But...is the transform legal?

- Skewing...



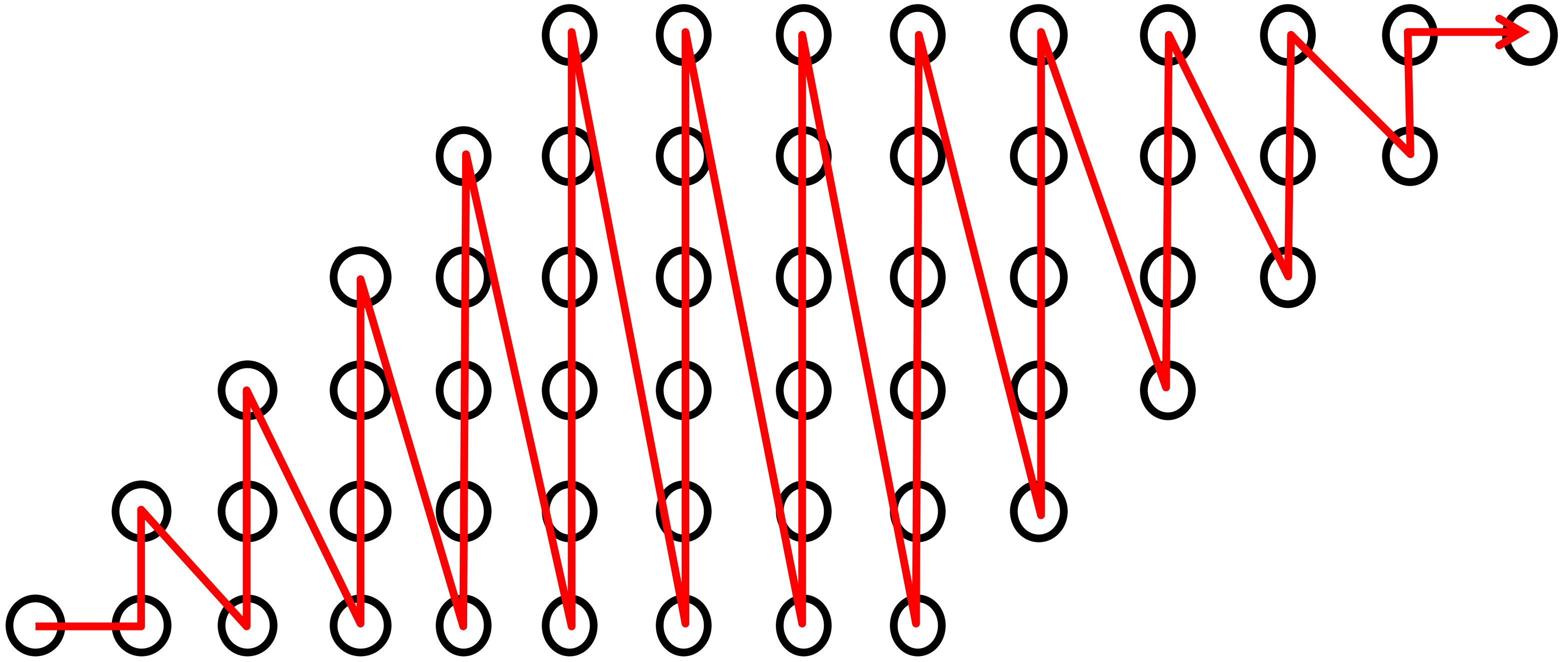
But…is the transform legal?

- 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, $1 \leq i \leq m$ (common access requirement)

General Solver?

- Looking for an interger solution to:

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

- N-deep loop nest
- M subscripts per array reference
- General case, too hard
- Restrict to linear functions of loop-indices
- System of linear equations (2xn 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 <u>PAIR</u> 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

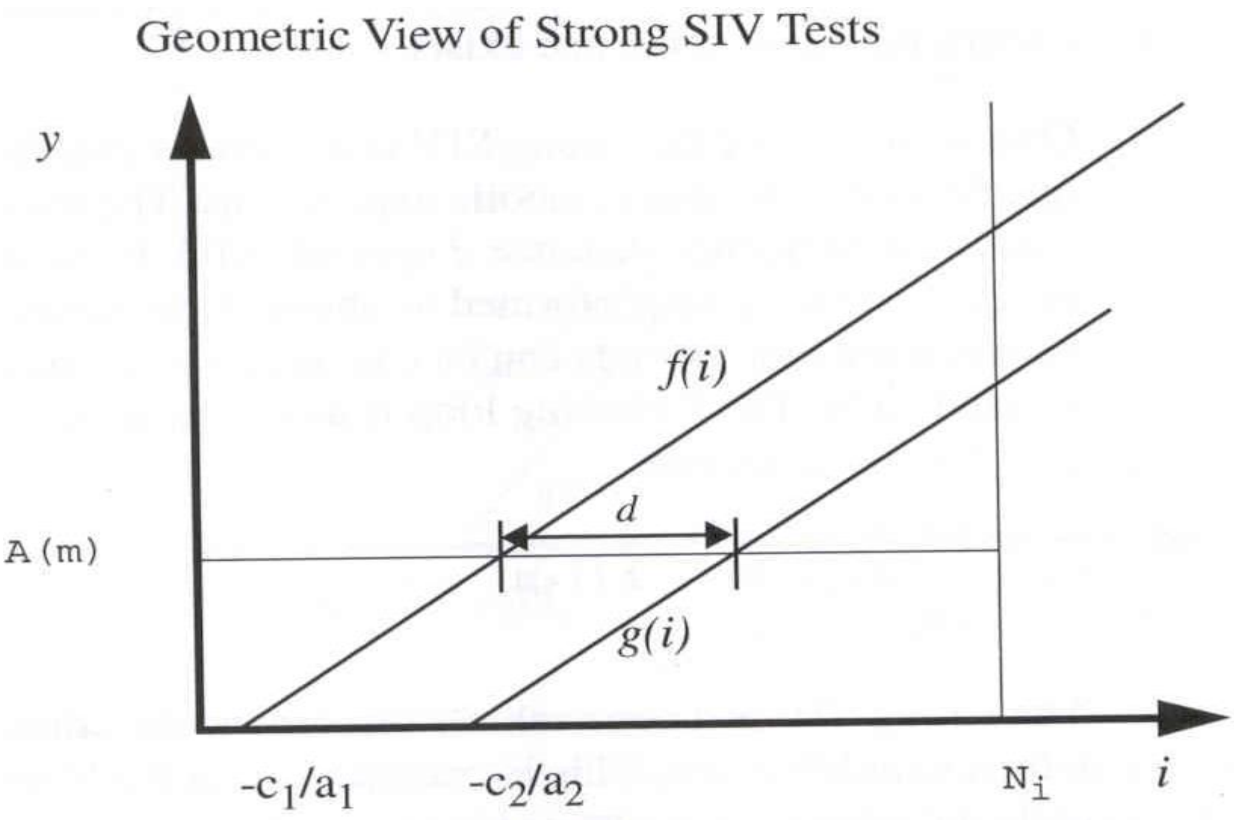
$$\langle i + 1, i \rangle$$

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

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



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

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

Weak SIV Tests

- Weak SIV subscripts are of the form

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

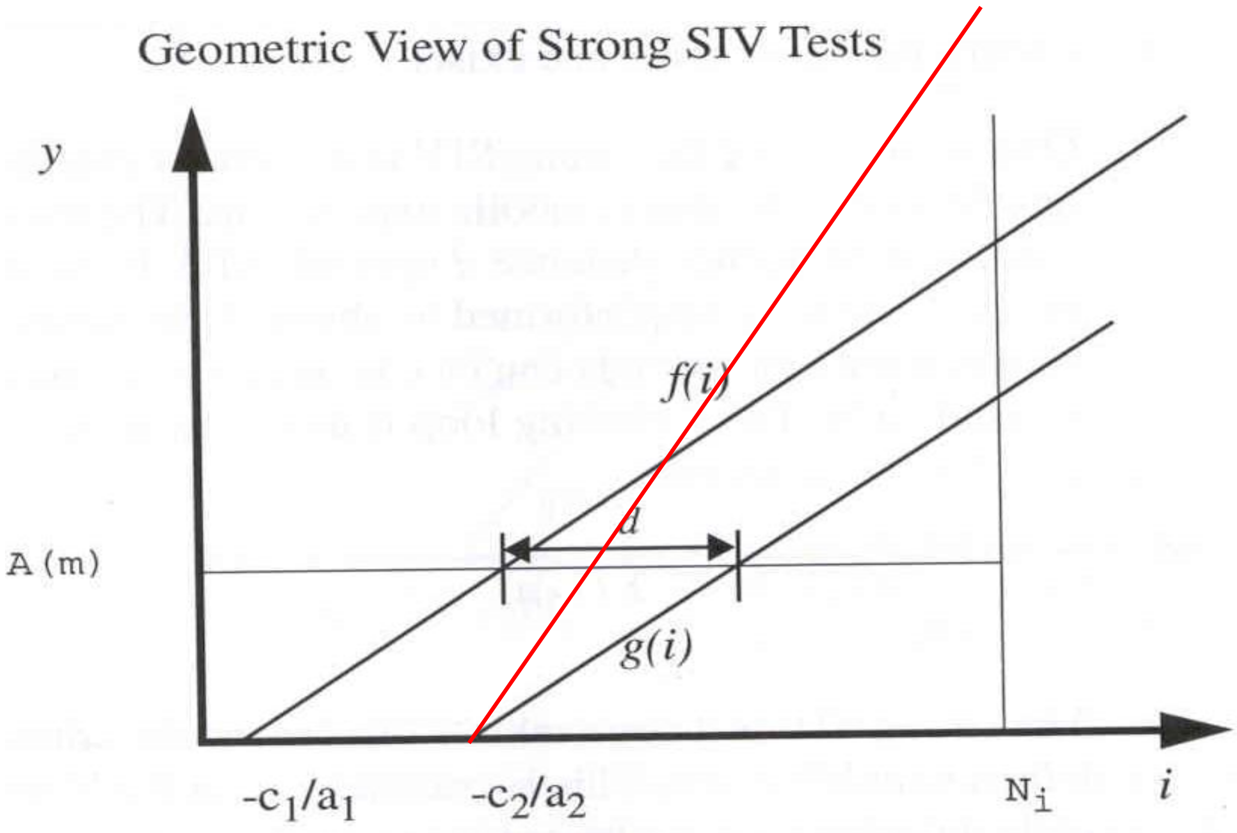
- For example the following are weak SIV subscripts

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

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

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

Geometric view of weak SIV

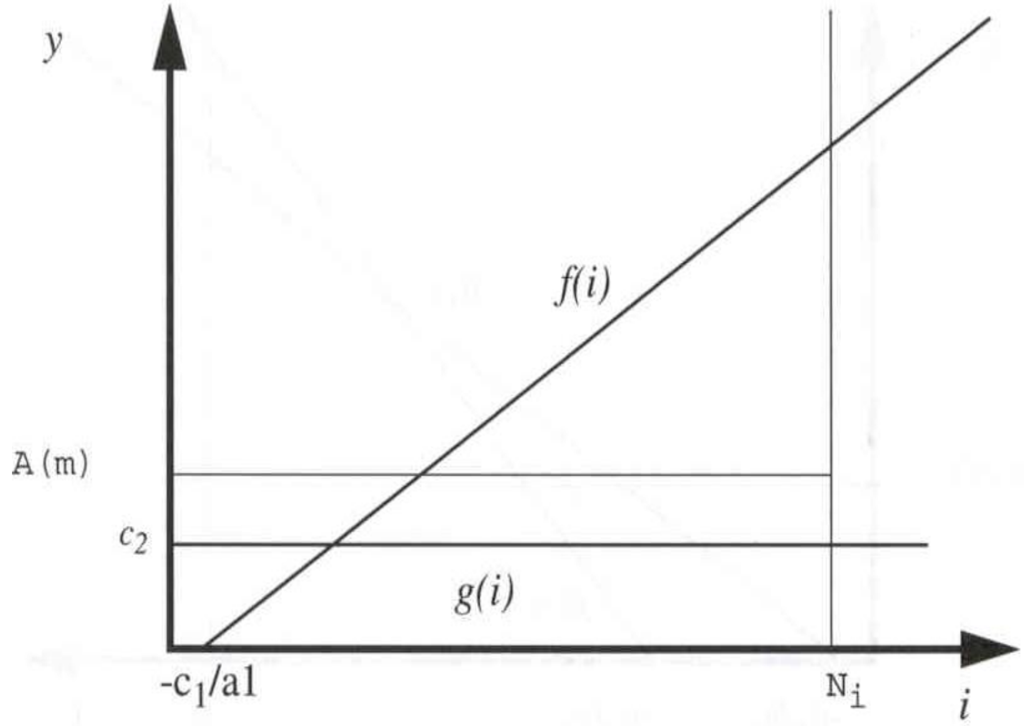


Weak-zero SIV Test

- 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 $i = \frac{c_2 - c_1}{a_1}$ and, $L \leq i \leq U$

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)
```

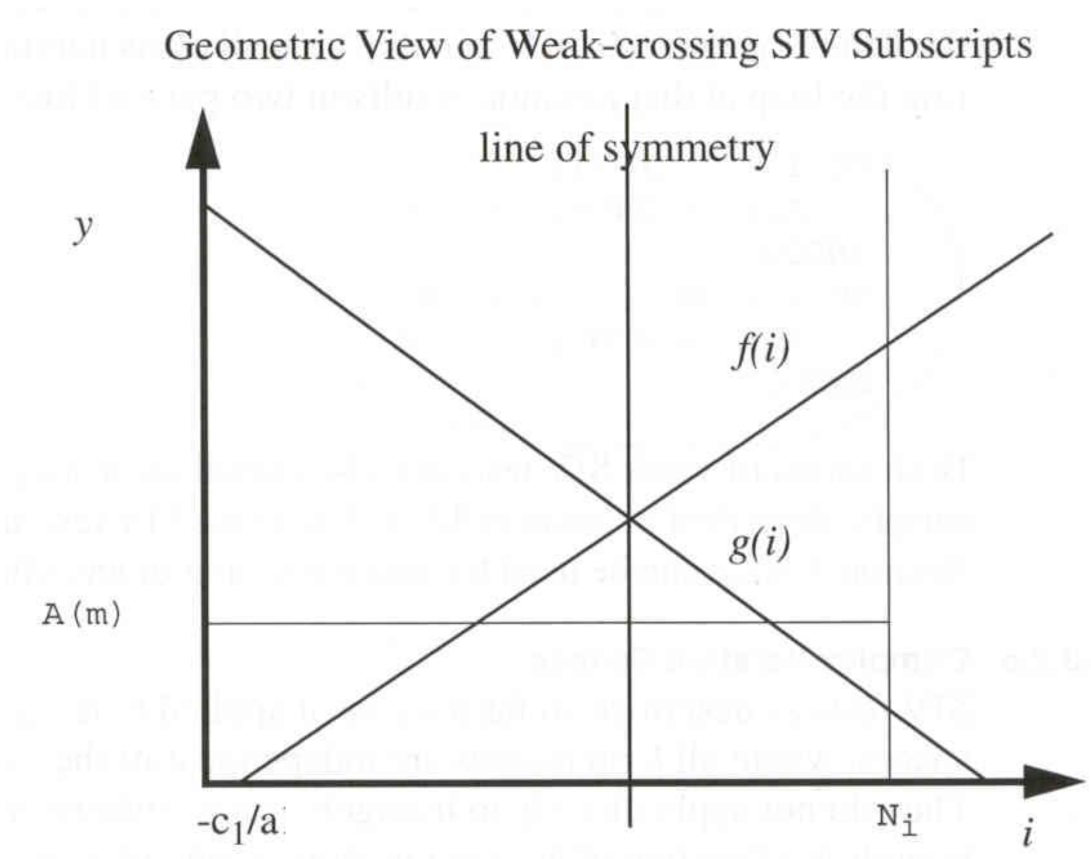
Weak-crossing SIV Test

- 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, $i = \frac{c_2 - c_1}{2a}$

  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)

More Complex Tests

- GCD-based testing
- Banerjee Inequalities
- Delta Test
- Omega Test
- …

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)