15-745
Optimizing For Data Locality - 1

Seth Copen Goldstein

Seth@cs.cmu.Edu

CMU

Based on "A Data Locality Optimizing Algorithm, Wolf & Lam, PLDI '91

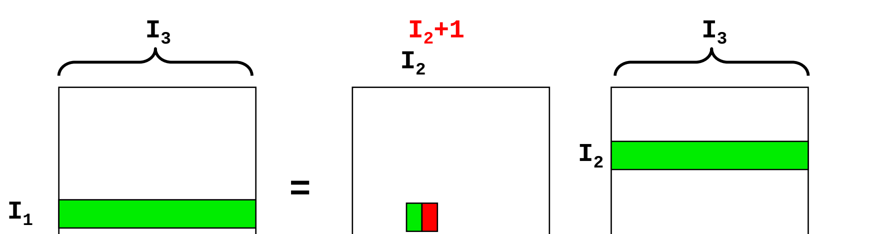
Outline

- The Problem
- Loop Transformations
  - dependence vectors
  - Transformations
  - Unimodular transformations
- Locality Analysis
- SRP

The Issue

- Improve cache reuse in nested loops
- Canonical simple case: Matrix Multiply

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

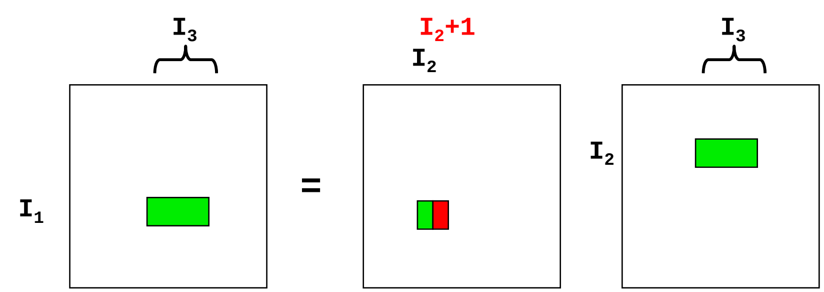


In next iteration of I_2 previous data that could be reused has been replaced in cache.

Tiling solves problem

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

```
for II2 := 1 to n by s
   for II3 := 1 to n by s
      for I1 := 1 to n
         for I2 := II2 to min(II2 + s - 1,n)
            for I3 := II3 to min(II3 + s - 1,n)
               C[I1,I3] += A[I1,I2] * B[I2,I3];
```

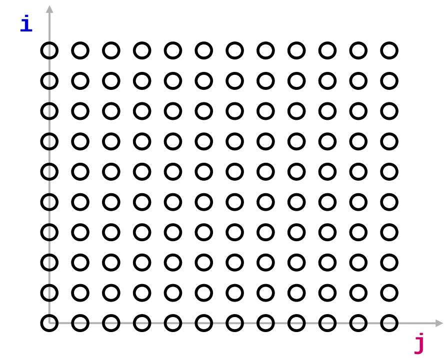


The Problem

- 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)

Handy Representation: "Iteration Space"

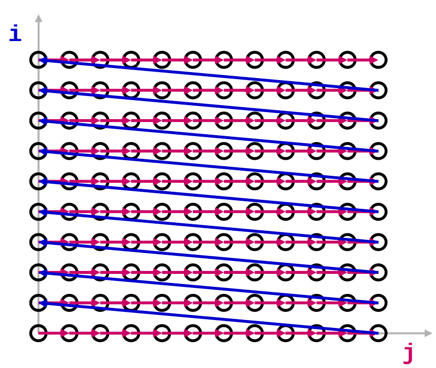
```
for i = 0 to N-1
  for j = 0 to N-1
   A[i][j] = B[j][i];
```



- 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];
```

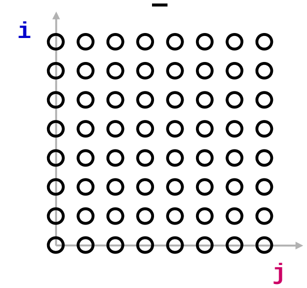


- Note: iteration space is not data space

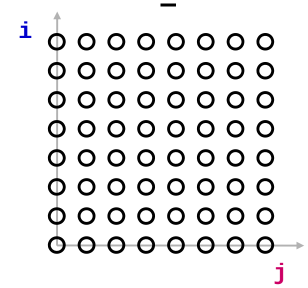
When Do Cache Misses Occur?

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

A

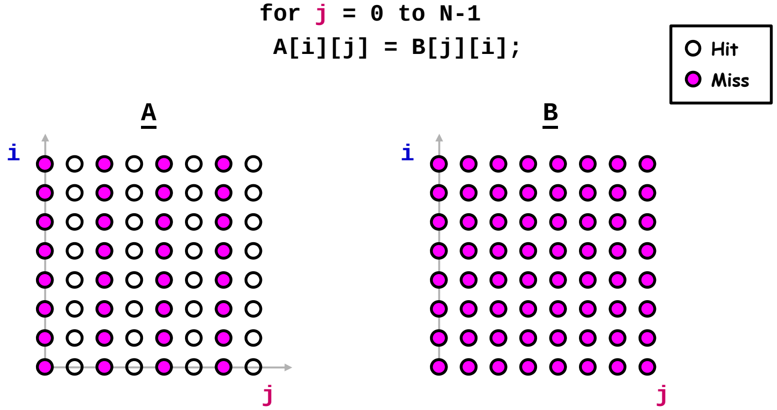


B



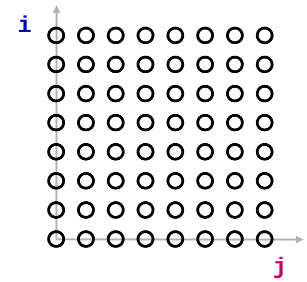
When Do Cache Misses Occur?

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



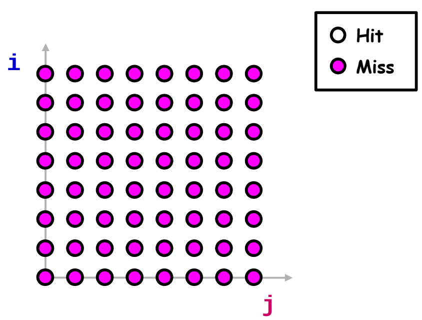
When Do Cache Misses Occur?

```
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 change the order of the iterations (or possibly data layout) 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
- ...

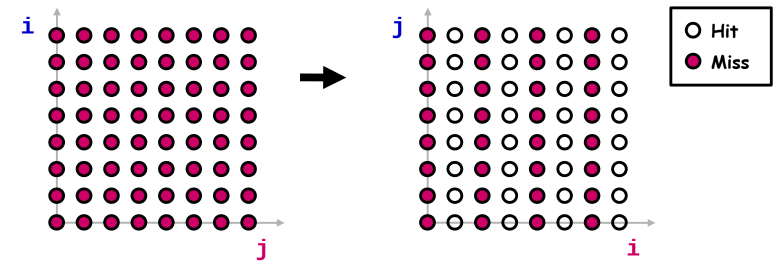
Loop Interchange, Cache Blocking: Can improve locality

Skewing, Loop Reversal: Can enable above

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;
```

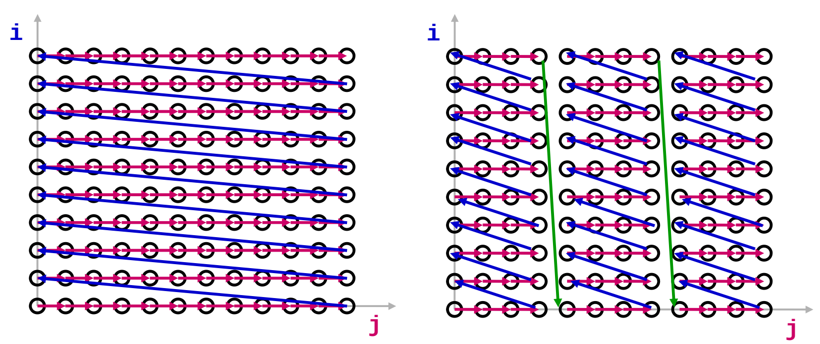


- *(assuming N is large relative to cache size)*

Impact on Visitation Order in Iteration Space

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

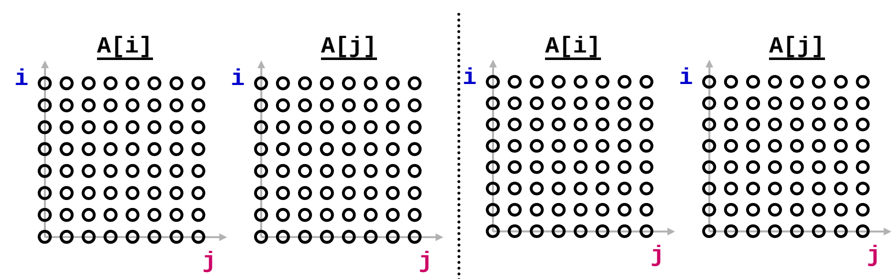
```
for JJ = 0 to N-1 by B
  for i = 0 to N-1
    for j = JJ to max(N-1,JJ+B-1)
      f(A[i],A[j]);
```



Cache Blocking (aka "Tiling")

```
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]);
```



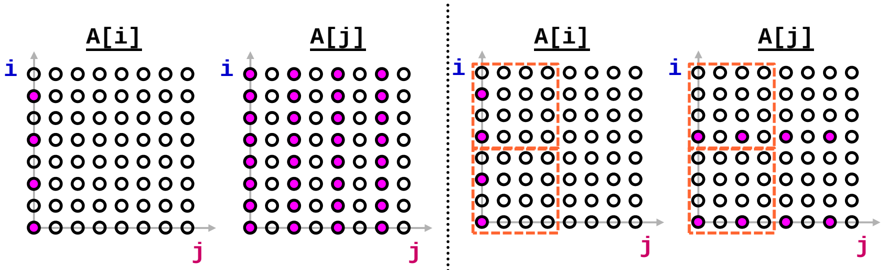
now we can exploit locality

Cache Blocking (aka "Tiling")

```
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]);
```



now we can exploit temporal locality

Cache Blocking in Two Dimensions

```
for i = 0 to N-1
  for j = 0 to N-1
    for k = 0 to N-1
      c[i,k] += a[i,j]*b[j,k];
```

```
for JJ = 0 to N-1 by B
  for KK = 0 to N-1 by B
    for i = 0 to N-1
      for j = JJ to max(N-1,JJ+B-1)
        for k = KK to max(N-1,KK+B-1)
          c[i,k] += a[i,j]*b[j,k];
```

- brings square sub-blocks of matrix "**b**" into the cache
- completely uses them up before moving on

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

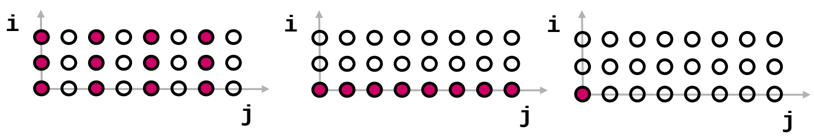
1. Find data reuse
   - if caches were infinitely large, we would be finished
2. Determine "localized iteration space"
   - set of inner loops where the data accessed by an iteration is expected to fit within the cache
3. Find data locality:
   - reuse $\supseteq$ localized iteration space $\supseteq$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

Types of Data Reuse/Locality

```
for i = 0 to 2
   for j = 0 to 100
       A[i][j] = B[j][0] + B[j+1][0];
```

○ Hit
● Miss

A[i][j] B[j][0] B[j+1][0]



Spatial Temporal Group (temporal)

Kinds of reuse and the factor

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

What kinds of reuse are there?

A[i]?

A[j]?

Kinds of reuse and the factor

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

Kinds of reuse and the factor

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

self-temporal in 1, self-spatial in 2
Also, group spatial in 2

What is different about this and previous?

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

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

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

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

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

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

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

Quantifying Reuse

- Why should we quantify reuse?
- How do we quantify locality?

Quantifying Reuse

- Why should we quantify reuse?
- How do we quantify locality?

- Use vector spaces to identify loops with reuse
- We convert that reuse into locality by making the "best" loop the inner loop
- Metric: memory accesses/iter of innermost loop. No locality $\rightarrow$ mem access

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
- There is locality if R_{ST} is in the localized vector space.

Recall that for nxm matrix A, the ker A = nullspace(A) = {$\mathbf{x}^m$ | A$\mathbf{x}$ = $\mathbf{0}$}

- Reuse is $s^{\dim(Rst)}$
- R_{ST} insersect L = locality
- # of mem refs = 1/above

Example of self-temporal reuse

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

Access	H	ker H	reuse?	Local?
$C[I_1,I_3]$	$\begin{pmatrix}1&0&0\\0&0&1\end{pmatrix}$	span{(0,1,0)}	n in I_2	
$A[I_1,I_2]$	$\begin{pmatrix}1&0&0\\0&1&0\end{pmatrix}$	span{(0,0,1)}		
$B[I_2,I_3]$	$\begin{pmatrix}0&1&0\\0&0&1\end{pmatrix}$	span{(1,0,0)}		

Self-Spatial

- Occurs when we access in order
  - A[i,j]: best gain, l
  - A[i,j*k]: best gain, l/k if |k| <= l
- How do we get spatial reuse for UG: H?

Self-Spatial

- Occurs when we access in order
  - A[i,j]: best gain, l
  - A[i,j*k]: best gain, l/k if |k| <= l
- How do we get spatial reuse for UG: H?
- Since all but row must be identical, set last row in H to 0, H_s
  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 SS

Example of self-spatial reuse

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

Access	H_s	ker H_s	reuse?	Local?
$C[I_1,I_3]$	$\begin{pmatrix}1&0&0\\0&0&0\end{pmatrix}$	span{(0,1,0), (0,0,1)}		1/l
$A[I_1,I_2]$	$\begin{pmatrix}1&0&0\\0&0&0\end{pmatrix}$	span{(0,0,1), (0,1,0)}		
$B[I_2,I_3]$	$\begin{pmatrix}0&1&0\\0&0&0\end{pmatrix}$	span{(1,0,0), (0,0,1)}		

Self-spatial reuse/locality

- $\mathrm{Dim}(R_{SS})$ is dimensionality of reuse vector space.
- If $R_{SS}=0 \rightarrow$ no reuse
- If $R_{SS}=R_{ST}$ no extra reuse from spatial
- Reuse of each element is $k/ls^{\dim(R\_SS)}$ where, s is number of iters per dim.
- $R_{SS} \cap L$ is amount of reuse exploited, therefore number of memory references generated is:
  $k/ls^{\dim(R\_ST \cap L)}$

Group Temporal

- Two refs $A[H\mathbf{i}+\mathbf{c}]$ and $A[H\mathbf{i}+\mathbf{d}]$ can have group temporal reuse in L iff
  - they are from same uniformly generated set
  - There is an $\mathbf{r} \in L$ s.t. $H\mathbf{r} = \mathbf{c} - \mathbf{d}$
- if $\mathbf{c}-\mathbf{d} = \mathbf{r_p}$, then there is group temporal reuse, $R_{GT} = \ker H + \mathrm{span}\{\mathbf{r_p}\}$
- However, there is no extra benefit if $R_{GT} \cap L = R_{ST} \cap L$

Example:

```
For i = 1 to n
  for j=i to n
    A[i,j] = 0.2*(A[i,j]+A[i+1,j]+
                  A[i-1,j]+A[i,j+1]+A[i,j-1])
```

If L = span{j}, since ker H = $\emptyset$:

A[i,j] and A[i,j-1] → (0,0)-(0,-1) $\in$span{(0,1)} yes

A[i,j-1] and A[i+1,j] → (0,-1)-(1,0) $\notin$span{(0,1)} no

Notice equivalence classes

Evaluating group temporal reuse

- Divide all references from a uniformly generated set into equiv classes that satisfy the R_{GT}
- For a particular L and g references
  - Don't count any group reuse when $R_{GT} \cap L = R_{ST} \cap L$
  - number of equiv classes is g_T.
  - Number of mem references is g_T instead of g

Total memory accesses

- For each uniformly generated set localized space, L line size, z

$$\frac{g_S+(g_T-g_S)/z}{z^e s^{\dim(R\_SS \cap L)}}$$

$$\text{where } e = \begin{cases} 0 \text{ if } R_{ST} \cap L = R_{SS} \cap L \\ 1 \text{ otherwise} \end{cases}$$

Now what?

- We have a way to characterize
  - Reuse (potential for locality)
  - Local iteration space
- Can we transform loop to take advantage of reuse?
- If so, can we?

Loop Transformation Theory

- Iteration Space
- Dependence vectors
- Unimodular transformations

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 $\mathbf{p}=(p_1, p_2, \ldots, p_n)$ and $\mathbf{q}=(q_1, q_2, \ldots, q_n)$ if $\mathbf{p} \succ_k \mathbf{q}$ iff for $1 \le k \le n$,

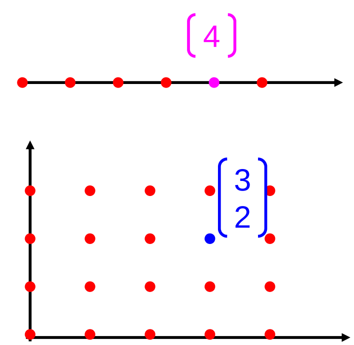
  $\forall\ 1 \le i < k, (p_i = q_i)$ and $p_k > q_k$
- For MM:
  - (1,1,1), (1,1,2), (1,1,3), …,
    (1,2,1), (1,2,2), (1,2,3), …,
    …,
    (2,1,1), (2,1,2), (2,1,3), …
  - $(1,2,1) \succ_2 (1,1,2), (2,1,1) \succ_1 (1,4,2)$, etc.

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++) {
     •••
}
```



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

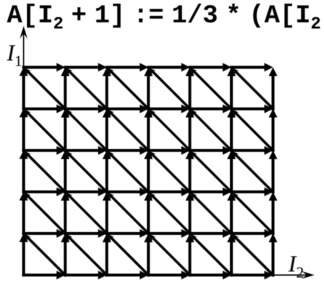
- Common ranges in dependence vectors
  - [1, ∞] as + or >
  - [- ∞, -1] as - or <
  - [- ∞, ∞] as ± or *
- A distance vector is the difference between the target and source iterations (for a dependent ref), e.g.,
  $$\mathbf{d} = \mathbf{I}_t - \mathbf{I}_s$$

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)

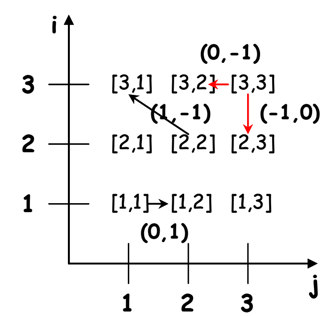
```
for I1 := 0 to 5
  for I2 := 0 to 6
    A[I2 + 1] := 1/3 * (A[I2] + A[I2 + 1] + A[I2 + 2])
```



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

Plausible Dependence vectors

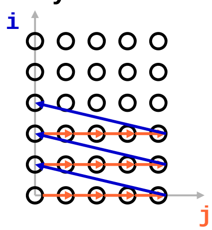
- A dependence vector is plausible iff it is lexicographically non-negative.
- All sequential programs have plausible dependence vectors. Why?
- Plausible: (1,-1)
- implausible (-1,0)



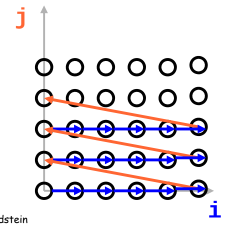
Loop Transforms

- 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 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

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

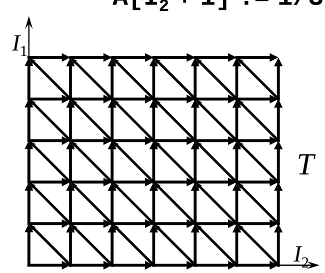
- Example for 2D

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

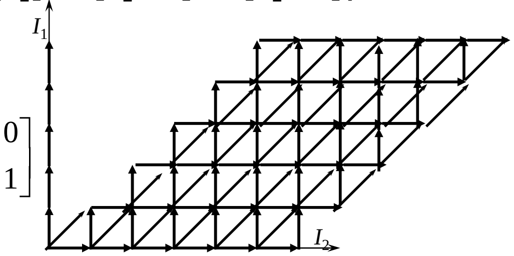
Loop Skewing Example

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

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



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




```
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)}

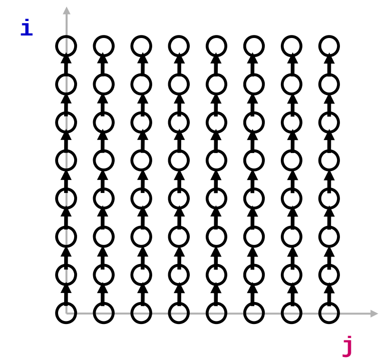
But...is the transform legal?

- 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?

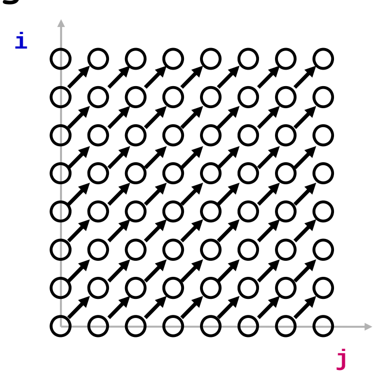
```
for i = 0 to N-1
  for j = 0 to N-1
A[i+1][j] += A[i][j];
```



But...is the transform legal?

- Loop reversal ok?
- Loop interchange ok?

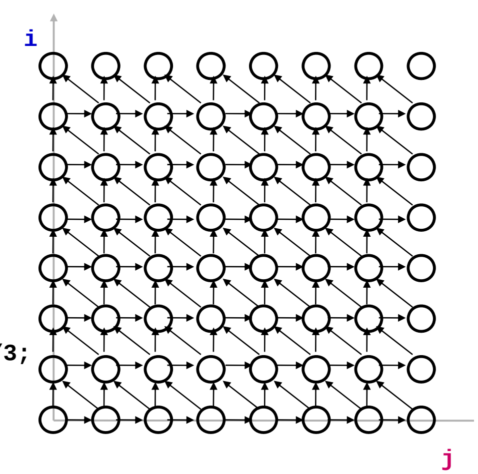
```
for i = 0 to N-1
  for j = 0 to N-1
A[i+1][j+1] += A[i][j];
```



But...is the transform legal?

- What other visit order is legal here?

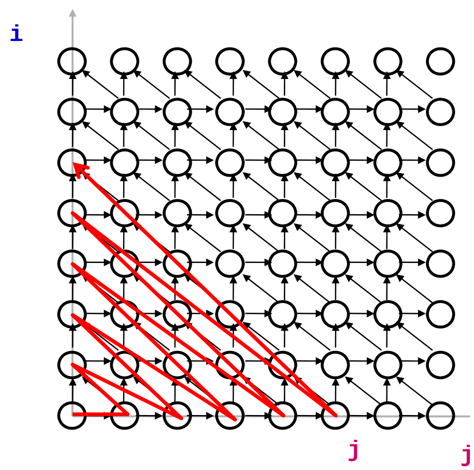
```
for i = 0 to TS
  for j = 0 to N-2
      A[j+1] =
(A[j] + A[j+1] + A[j+2])/3;
```



But...is the transform legal?

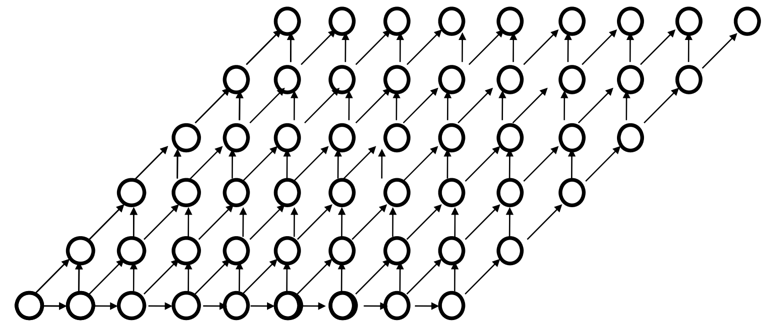
- What other visit order is legal here?

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



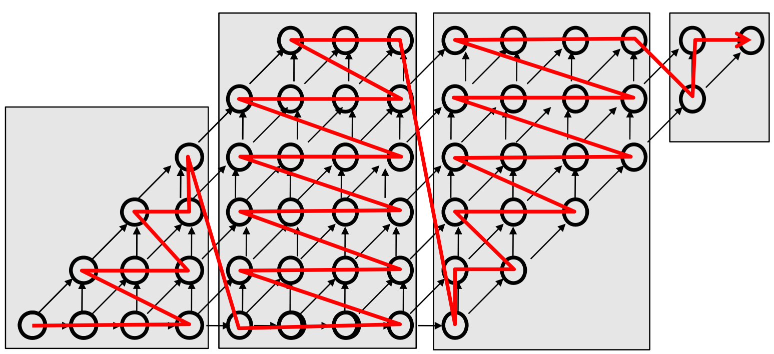
But...is the transform legal?

- Skewing...



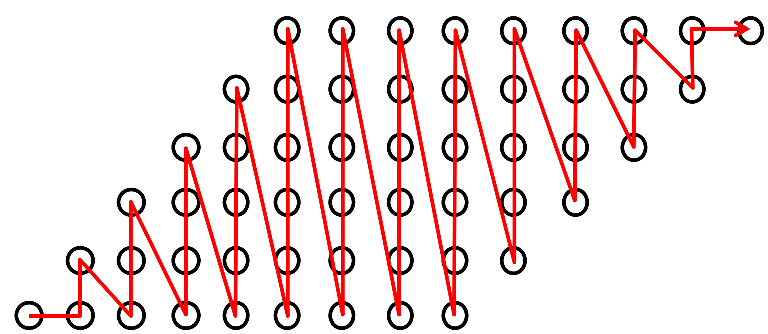
But...is the transform legal?

- 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

Next Time

- Putting it all together: SRP
- Other loop transformations for locality

Linear Algebra

- Vector Spaces
- Linear Combinations
- dimensions
- Spans
- Kernels

Vector Spaces

- **n** is a point in n-space
- $V = \{ v_1, v_2, \ldots, v_m\}$ is a finite set of n-vectors over m $\Re^n$.
- Linear combination of vectors of V is a vector **x** as defined by
  $$\mathbf{x} = \alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2 + \ldots + \alpha_m\mathbf{v}_m$$
  where α_i are real numbers.
- V is linearly dependent if a combination results in the **0** vector, otherwise it is linearly independent.

Dim and Basis

- dimensionality of V is dim(V)
  the number of independent vectors in V
- A basis for an m-dimensional vector space is a set of linearly independent vectors such that every point in V can be expressed as a linear comb of the vectors in the basis.
  - The vectors in the basis are called basis vectors

Subspaces and span

- Let V be a set of vectors
- The subspace spanned by V, span(V), is a subset of $\Re^n$ such that
  - $V \subseteq$ span(V)
  - $\mathbf{x},\mathbf{y} \in$ span(V) $\Rightarrow \mathbf{x}+\mathbf{y} \in$ span(V)
  - $\mathbf{x} \in$ span(V) and $\alpha \in \Re \Rightarrow \alpha\, \mathbf{x} \in$ span(V)

Range, Span, Kernel

- A matrix A can be viewed as a set of column vectors.
- Range A^{nxm} is $\{Ax | x \in \Re^m\}$
- span(A) = Range A^{nxm}
- nullspace(A) = ker(A) = ker(A^{nxm}) = $\{\mathbf{x}^m | A\mathbf{x} \in \mathbf{0}\}$
- rank(A) = dim(span(A))
- nullity(A) = dim(ker(A))
- rank(A)+nullity(A) = n, for A^{nxm}