Linear Loop Transformations for Locality Enhancement

Story so far

- Cache performance can be improved by tiling and permutation
- Permutation of perfectly nested loop can be modeled as a *linear transformation* on the iteration space of the loop nest.
- Legality of permutation can be determined from the dependence matrix of the loop nest.
- Transformed code can be generated using ILP calculator.

Theory for permutations applies to other loop transformations that can be modeled as linear transformations: skewing, reversal,scaling.

Transformation matrix: T (a non-singular matrix)

Dependence matrix: D

Matrix in which each column is a distance/direction vector

Legality: $T.D \succ 0$

Dependence matrix of transformed program: $T.D$

Small complication with code generation if scaling is included.

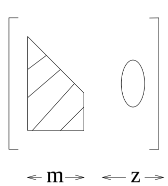
Overview of lecture

- Linear loop transformations: permutation, skewing, reversal,scaling
- Compositions of linear loop transformations: matrix viewpoint
- Locality matrix: model for temporal and spatial locality
- Two key ideas:
  1. Height reduction
  2. Making a loop nest fully permutable
- A unified algorithm for improving cache performance
- Lecture based on work we did for HP's compiler product line

Key algorithm: finding basis for null space of an integer matrix

Definition: A vector x is in null space of a matrix M if Mx = 0.

It is easy to find a basis for null space of matrix in column-echelon form.



Null space basis: $\begin{bmatrix} 0^{mxz} \\ I^{zxz} \end{bmatrix}$

General matrix M:

- Find unimodular matrix U so that MU = L is in column-echelon form.
- If x is a vector in null space of L, Ux is in null space of M.
- If B is basis for null space of L, UB is basis for null space of M.

Example: M*U = L

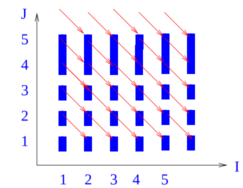
$$\begin{pmatrix} 1 & 0 & 2 \\ 0 & 2 & 2 \\ 1 & -1 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 1 & -1 & 0 \end{pmatrix}$$

Basis for null space of M = U* (0 0 1)' = (-2 -1 1)'

Some observations:

- We have used reduction to column-echelon form: MU = L.
- We can also view this as a matrix decomposition: M = LQ where Q is inverse of U from previous step.
- Analogy: QR factorization of real matrices.
  Unimodular matrices are like orthogonal matrices in reals.
- Special form of this decomposition: M = LQ where all diagonal elements of L are non-negative.
- This is called a psuedo-Hermite normal form of matrix.

Motivating example: Wavefront code




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

Dependence matrix = $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$

Dependence between two iterations
=> iterations touch the same location
=> potential for exploiting data reuse!

`N` iteration points between executions of dependent iterations.

Can we schedule dependent iterations closer together?

For now, focus only on reuse opportunities between dependent iterations.

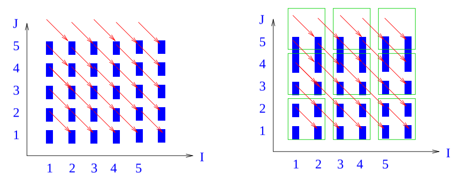
This exploits only one kind of temporal locality.

There are other kinds of locality that are important:

- iterations that *read* from the same location: input dependences
- spatial locality

Both are easy to add to basic model as we will see later.

Exploiting temporal locality in wavefront



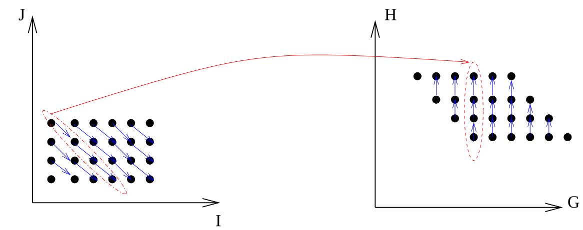
Permutation is illegal!   Tiling is illegal!

We have studied two transformations: permutation and tiling.

Permutation and tiling are both illegal.

Height Reduction

One solution: schedule iterations along 45 degree planes !



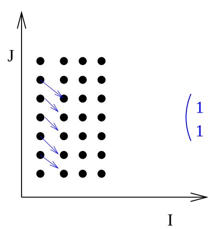
Note:

- Transformation is legal.
- Dependent iterations are scheduled close together, so good for locality.

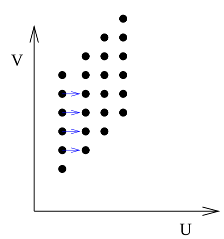
Can we view this in terms of loop transformation?

Loop skewing followed by loop reversal.

Loop Skewing: a linear loop transformation



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



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

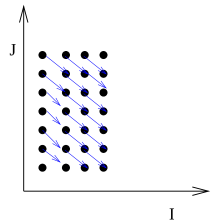
Skewing of inner loop by an outer loop: always legal

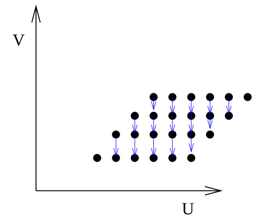
New dependence vectors: compute T*D

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

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

Skewing outer loop by inner loop





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

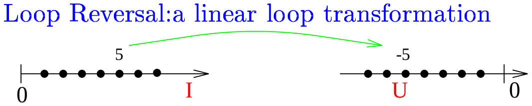
Outer loop skewing: $\begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix}$

Skewing of outer loop by inner loop: not necessarily legal

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

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

Loop Reversal:a linear loop transformation



U = [-1][I]

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

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

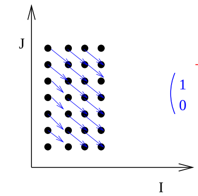
Transformation matrix = [-1]

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

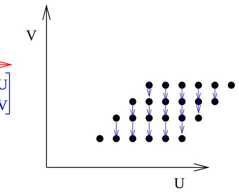
Legality of loop reversal: Apply transformation matrix to all dependences & verify lex +ve

Code generation: easy

Need for composite transformations



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



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

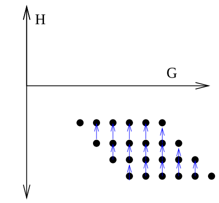
Transformation: skewing followed by reversal

In final program, dependent iterations are close together!

Composition of linear transformations = another linear transformation!

Composite transformation matrix is

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



How do we synthesize this composite transformation??

Some facts about permutation/reversal/skewing

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

Synthesizing composite transformations using matrix-based approaches

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

Height reduction: move reuse into inner loops

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

Prefer not to have dependent iterations in different outer loop iterations.

So dependence vector in transformed program should look like $\begin{pmatrix} 0 \\ ?? \end{pmatrix}$.

So $T * \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ ?? \end{pmatrix}$

This says first row of T is orthogonal to $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

So first row of T can be chosen to be (1 1).

What about second row?

Second row of T (call it s) should satisfy the following properties:

- s should be linearly independent of first row of T (non-singular matrix).
- $T * D$ should be a lexicographically positive vector.
- Determinant of matrix should be +1 or -1 (unimodularity).

One choice: $(0 \quad -1)$, giving $T = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}$.

General questions

1. How do we choose the first few rows of the transformation matrix to push reuses down?
2. How do we fill in the rest of the matrix to get unimodular matrix?

Choosing first few rows of transformation matrix:

For now, assume dependence matrix D has only distances in it.

Algorithm:

- Let B be a basis for null space of D'.
- Use transpose of basis vectors as first few rows of transformation matrix.

Example: study running example in previous slides.

Extension to direction vectors: easy

Consider $D = \begin{pmatrix} + & 1 \\ 0 & - \\ 1 & 2 \\ -1 & -2 \end{pmatrix}$

Let first row of transformation be $(r_1\ r_2\ r_3\ r_4)$.
If this is orthogonal to D, it is clear that r_1 and r_2 must be zero.
Ignore 1st and 2nd rows of D and find basis for null space of remaining rows of D : (1 1)
Pad with zeros to get partial transformation: (0 0 1 1)

General picture: knock out rows of D with direction entries, use algorithm for distance vectors, and pad resulting null space vectors with 0's.

Filling in rest of matrix:

- Turns out to be easier if we drop unimodularity requirement
- Completion procedure: generate a non-singular matrix T given first few rows of this matrix
- Problem: how do we generate code if transformation matrix is a general non-singular matrix?

Completion Procedure

Goal: to generate a complete transformation from a partially specified one

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

Given:

- a dependence matrix D
- a *partial* transformation P (first few rows)

generate a non-singular matrix T that extends P and that satisfies all the dependences in D ($TD \succ 0$).

Precondition:

- rows of P are linearly independent
- $P * D \succeq 0$ (no dependences violated yet)

Completion algorithm: iterative algorithm

1. From D, delete all dependences d for which $Pd \succ 0$.
2. If D is now empty, use null space basis determination to add more rows to P to generate a non-singular matrix.
3. Otherwise, let k be the first row of (reduced) D that has a non-zero entry. Use e_k (unit row vector with 1 in column k and zero everywhere else) as the next row of the transformation.
4. Repeat from step (i).

Proof: Need to show that e_k is linearly independent of existing rows of partial transformation, and does not violate any dependences. Easy (see Li/Pingali paper).

Example:

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

No dependence is satisfied by partial transformation.
Next row of transformation = (1 0 0)

New partial dependence matrix is

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

All dependences are now satisfied, so we are left with the problem of completing transformation with a row independent of existing ones.

Basis for null space of partial transformation = (0 1 -1)’

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

Problem: transformation matrix T we obtain is non-singular but may not be unimodular.

Two solutions: based on T = LQ pseudo-Hermite normal form decomposition where L is lower-triangular with positive diagonal elements

- Develop code generation technology for non-singular matrices.
- Use Q as transformation matrix.
  Question: How do we know this is legal, and is good for locality?

First, let us understand what transformations are modeled by non-singular matrices that are not unimodular.

Loop scaling:change step size of loop

```
DO 10 I = 1,100        vs        DO 10 U = 2,200,2
10    Y(I) = I                   10 Y(U/2) = U/2
```

Scaling matrix: non-unimodular transformation

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

- Scaling, like reversal, is not important by itself.
- Nonsingular linear loop transformations: permutation/skewing/reversal/scaling
- Any non-singular integer matrix can be decomposed into product of permutation/skewing/reversal/scaling matrices.
- Standard decompositions: (pseudo)-Hermite form $T = L * Q$ where Q is unimodular and L is lower triangular with positive diagonal elements, (pseudo)-Smith normal form $T = UDV$ where U, V are unimodular and D is diagonal with positive elements.
- Including scaling in repertoire of loop transformations makes it easier to synthesize transformations but complicates code generation.

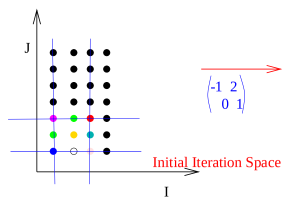
Code generation with a non-singular transformation T

Code generation for unimodular matrix U:

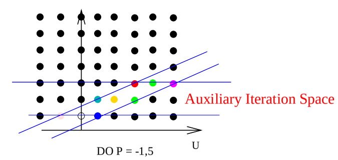
- Original bounds: $A * \underline{I} \leq b$
- Transformation: $\underline{J} = U * \underline{I}$
- New bounds: compute from $A * U^{-1}\underline{J} \leq b$

Key problem here: T^{-1} is not necessarily an integer matrix.

Running example: factorize $T = L * Q$



$\begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}$

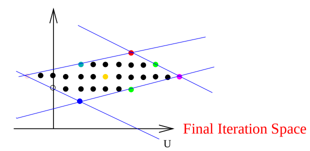



```
DO P = -1,5
  DO Q = max(1,ceil(p+1/2)),min(3,floor(p+3/2)
    A(2P+3,-P+Q) = Q
```

```
DO I = 1,3
  DO J = 1,3
    A(4J-2I+3,I+J) = J;
```

$\begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix}$

$$\boxed{\begin{pmatrix} -2 & 4 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}}$$




```
DO U = -2,10,2
  DO V = -U/2 +3 max(1,ceil(u/2+1/2)),
         -U/2 + 3min(3,floor(U/2+3/2)),3
    A[U+3,V] = (U+2V)/6
```


- Step sizes of loops: diagonal entries of L
- Change of variables in body

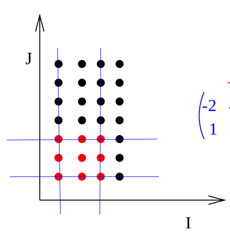
$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} -1/6 & 4/6 \\ 1/6 & 2/6 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}$$

- Good to use strength reduction to replace the integer divisions by additions and multiplications.

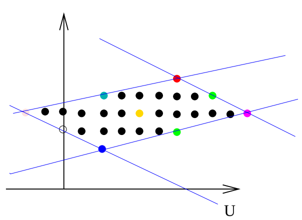
Complete transformation with non-singular matrix is needed for generating code for distributed-memory parallel machines and similar problems (see Wolfe's book).

Bottom line: transformation synthesis procedure can focus on producing non-singular matrix. Unimodularity is not important.

Difficulties:



$$\begin{pmatrix} -2 & 4 \\ 1 & 1 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$




```
DO I = 1,3
  DO J = 1,3
    A(4J-2I+3,I+J) = J;
```

```
DO U = -2,10,2
  DO V = -U/2 +3 max(1,ceil(u/2+1/2)),
         -U/2 + 3min(3,floor(U/2+3/2)),3
    A[U+3,V] = (U+2V)/6
```

Key problems:

How do we determine integer lower and upper bounds?
(rounding up from rational bounds is not correct)

How do we determine step sizes?

Solution: use Hermite normal form decomposition $T = L * Q$

Using non-singular matrix T as transformation matrix.

Running example:

- Given $T = L * Q$ where L is triangular with positive diagonal elements and Q is unimodular.

$$\begin{pmatrix} -2 & 4 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}$$

- Use Q as the transformation to generate bounds for auxiliary space.

$$-1 \leq \quad p \quad \leq 5$$
$$max(1, ceil((p+1)/2)) \leq \quad q \quad \leq min(3, floor((p+3)/2))$$

- Read off bounds for final iteration space from L

$$\begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix}$$

$$-2 \leq \quad u \quad \leq 10$$
$$-u/2 + 3max(1, ceil((u/2+1)/2)) \leq \quad v \quad \leq -u/2 + 3 * min(3, floor((u/2+3)/2))$$

Solution 2 to code generation with non-singular transformation matrices

Lemma: If L is a $n \times n$ triangular matrix with positive diagonal elements, L maps lexicographically positive vectors into lexicographically positive vectors.

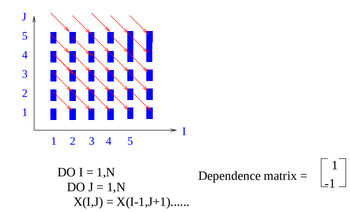
Intuition: L does not change the order in which iterations are done but just renumbers them!

Proof: Consider $d = L * \Delta$ where Δ is a lexicographically positive vector and show that d is lexicographically positive as well.

- Auxiliary space iterations are performed in same order as corresponding iterations in final space.
- Memory system behavior of auxiliary program is same as memory system behavior of final program!

For memory system optimization, we can use a non-singular transformation matrix T as follows: compute Hermite normal form of $T = L * Q$ and use Q as transformation matrix!

Putting it all together for wavefront example:



- Find rows orthogonal to space spanned by dependence vectors, and use them as the partial transformation. In this example, the orthogonal subspace is spanned by the vector (1 1).
- Apply completion procedure to partial transformation to generate full transformation. We would add row (1 0). So transformation is

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

General picture:

Exploit *dependences + spatial locality + read-only data reuse*

1. a dependence matrix D with dependence vectors
2. a locality matrix $L \supseteq D$ with locality vectors

Locality matrix has additional vectors to account for spatial locality and read-only data reuse.

Intuitively, we would like to scan along directions in L in innermost loops, provided dependences permit it.

Modeling spatial locality: scan iteration space points so that we get unit stride accesses (or some multiple thereof)

Example: from Assignment 1

```
DO I = ..
 DO J = ...
  C[I,J] = A[I,J] + B[J,I]
```

If arrays are stored in column-major order, we should interchange the two loops. How do we determine this in general?

Example: array stored in column-major order, reference $X[A\underline{i} + \underline{a}]$

Suppose we do iteration $\underline{i}_1$ and then iteration $\underline{i}_2$.
Array element accessed in iteration $\underline{i}_1 = X[A\underline{i}_1 + \underline{a}]$
Array element accessed in iteration $\underline{i}_2 = X[A\underline{i}_2 + \underline{a}]$

We want these two accesses to be within a column of A. So

$$A * (\underline{i}_1 - \underline{i}_2) = \begin{pmatrix} c \\ 0 \\ \dots \\ 0 \end{pmatrix}$$

Let $A' = A$ with first row deleted. So we want to solve

$A' * R = \underline{0}$ where R is "reuse direction".

Solution: R is any vector in null space of A'.

Spatial locality matrix: matrix containing a basis for null space of A'

Example: for i,j,k ...A[i+j+k,k+j,2k+2j] ..

In what direction(s) can we move so we get accesses along first dimension of A for the most part?

$$A' = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 2 & 2 \end{pmatrix}$$

Spatial locality matrix is $\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix}$

Example from Assignment 1: exploiting spatial locality in MVM matrix accesses

```
for I = 1,N
  for J = 1,N
    Y(I) = Y(I) + A(I,J)*X(J)
```

Reference matrix is $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

Truncated matrix is $\begin{pmatrix} 0 & 1 \end{pmatrix}$.

So spatial locality matrix is

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

Enhancing spatial locality: perform height reduction on locality matrix

For our example, we get the transformation

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

That is, we would permute the loops which is correct.

Read-only data reuse: add reuse directions to locality matrix

```
DO I = 1,N
 DO J = 1,N
   DO K = 1,N
    C[I,J] = C[I,J] + A[I,K]*B[K,J]
```

Consider reuse vector for reference `A[I,K]`:

$$\begin{aligned} 1 &\leq I_1, I_2, J_1, J_2, K_1, K_2 \leq N \\ (I_1, J_1, K_1) &\prec (I_2, J_2, K_2) \\ I_1 &= I_2 \\ K_1 &= K_2 \\ \Delta 1 &= I_2 - I_1 \\ \Delta 2 &= J_2 - J_1 \\ \Delta 3 &= K_2 - K_1 \end{aligned}$$

Reuse vector is $\begin{pmatrix} 0 \\ + \\ 0 \end{pmatrix}$.

Considering all the references, we get the following locality matrix:

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

General Algorithm for Height Reduction

- Compute the locality matrix L which contains all vectors along which we would like to exploit locality (*dependences + read-only data reuse + spatial locality*)
- Determine a basis B for the null space of L^T and use that B^T as the first few rows of the transformation.
- Call the completion algorithm to generate a complete transformation.
- Do a pseudo-Hermite form decomposition of transformation matrix and use the unimodular part as transformation.

Flexibility: if null space of L^T has only the zero vector in it, it may be good to drop some of the "less important" read-only reuse vectors and spatial locality vectors from consideration.

In some codes, height reduction may fail since locality vectors span entire space.

Example: MMM as shown in previous slide (actually even MVM if we want to exploit locality in both matrix A and vectors x,y)

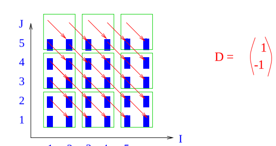
We can still exploit most of the locality provided we can tile loops.

Tiling is not always legal...

Solution: apply linear loop transformations to enable tiling.

Linear loop transformations to enable tiling

In general, tiling is not legal.



Tiling is illegal!

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

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

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

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

Example: wavefront

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

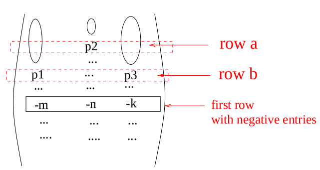
Dependence matrix of transformed program must have all positive entries.

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

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

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

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



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

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

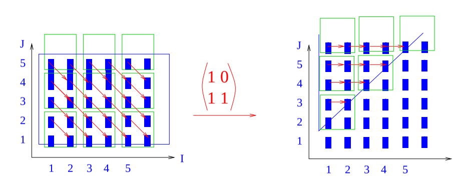
Transformation: $\begin{pmatrix} I \\ 0\ 0\ ..0\ \text{ceiling}(n/p2)\ 0\ 0\ \text{ceiling}(\max(m/p1,k/p3))0...0 \\ I \end{pmatrix}$

General algorithm for making loop nest fully permutable:

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

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

Result of tiling transformed wavefront



Original loop　　　Tiled fully permutable loop

Tiling generates a 4-deep loop nest.

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

What happens with direction vectors?

In general, we cannot make loop nest fully permutable.

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

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

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

Idea: algorithm will find *bands* of fully permutable loop nests.

Example for general algorithm:

$$\begin{pmatrix} 0 & 1 & 0 \\ + & - & + \\ -2 & -3 & -1 \\ - & + & -2 \\ 1 & 3 & 4 \end{pmatrix}$$

Begin first band of fully permutable loops with first loop.

Second row has -ve direction which cannot be knocked out. But we can interchange fifth row with second row to continue band.

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 3 & 4 \\ -2 & -3 & -1 \\ - & + & -2 \\ + & - & + \end{pmatrix}$$

New second loop can be part of first fully permutable band. Knock out -ve distances in third row by adding 2*second row to third row.

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 3 & 4 \\ 0 & 3 & 7 \\ - & + & -2 \\ + & - & + \end{pmatrix}$$

Cannot make band any bigger, so terminate band, drop all satisfied dependences, and start second band.

In this case, all dependences are satisfied, so last two loops form second band of fully permutable loops.

How do we determine transformation matrix for performing this operations?

If you have n loops, start with $T = I^{n \times n}$.

Every time you apply a transformation to the dependence matrix, apply same transformation to T.

At the end, T is your transformation matrix.

Proof sketch: $(U_2 * (U_1 * D)) = (U_2 * (U_1 * I)) * D$

For the previous example, we get

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

Interchange the second and fifth rows to get

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

Add 2*second row to third.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Conversion to fully permutable bands

1. Compute dependence matrix D. Current-row = row 1.
2. Begin new fully permutable band.
3. If all entries in D are non-negative, add all remaining loops to current band and exit.
4. Otherwise, find first row r with one or more negative entries. Each negative entry must be guarded by a positive entry above it in its dependence vector. Add all loops between current-row and row (r-1) to current band.
5. If row does not both positive and negative directions, fix it as follows.
   If all direction entries are negative, negate row.
   At this point, all direction entries must be positive, and any remaining negative entries must be distances. If there are negative entries remaining, determine appropriate multiple of

guard rows to be added to that row (treating + guard entries as 1) to convert all negative entries in row r to strictly positive.
Update transformation matrix and add row r to current band.
Current-row = r+1
Go to step 3.

6. Otherwise, see if a row below current row without both positive and negative directions can be interchanged legally with current row.
   If so, perform interchange, update transformation matrix and go to step 4.
   Otherwise, terminate current fully permutable band, drop all satisfied dependence vectors, and go to step 2.

Summary

- Height reduction can also be viewed as outer-loop parallelization: if no dependences are carried by outer loops, outer loops are parallel!
- First paper on transformation synthesis in this style: Leslie Lamport (1969) on multiprocessor scheduling.
- Algorithm here is based on Wei Li's Cornell PhD thesis (Wei is now a compiler hot-shot at Intel!).
- A version of this algorithm was implemented by us in HP's compiler product line.
- It is easy to combine height reduction and conversion to fully permutable band form into a single transformation matrix synthesis, but you can work it out for yourself.

Ye Grande Locality Enhancement Algorithm

1. Compute locality matrix L.
2. Perform height reduction by finding a basis for null space of L^T. If null space is too small, you have the option of dropping less important locality vectors.
3. Compute resulting dependence matrix after height reduction. If all entries are non-negative, declare loop nest to be fully permutable.
4. Otherwise, apply algorithm to convert to bands of fully permutable loop nests.
5. Perform pseudo-Hermite normal form decomposition of resulting transformation matrix, and use unimodular extract to perform code transformation.
6. Tile fully permutable bands choosing appropriate tile sizes.