Statement re-ordering can be static or dynamic.

\[ 0 < Z(I) \rightarrow Y(I) \rightarrow \ldots = 1 \]

Do \( Z = 1 \rightarrow 100 \)

\[ \ldots \rightarrow Y(I) \rightarrow \ldots = 1 \to Z(I) \rightarrow Y(I) \rightarrow \ldots = 1 \to \]

Do \( I = 1 \rightarrow 100 \)

Do \( I = 1 \rightarrow 100 \)

---

**Example**

Statement re-ordering: (6) Loop distribution/jamming:

All statements in body affected identically.

\[ \ldots = (I)Z \rightarrow (I) \to \]

\[ \ldots = (I)Z \rightarrow (I) \to \]

\( X* \to (I) \to (I) \to \)

\( A + (I) \to (I) \to (I) \to \)

\( Y = (I) \to (I) \to (I) \to \)

Do \( I = 1 \rightarrow 100 \)

Do \( I = 1 \rightarrow 100 \)

Do \( I = 1 \rightarrow 100 \)

---

**Example**

Iteration re-numbering: (6) Loop interchange:

Classes of loop transformations.
Statements themselves are altered.

\[ [I][I][I] \times (I', J) \]

\[ [I][I][I] \times (I, J) \]

DO 10 I = 1, 100

\[ \text{Example: scalar expansion} \]

Statement transformation: ●
Leqality: always legal

Iterations are done separately from main loop.

Special case: loop peeling - only the first/last/both first and last

\[
\text{S to S}
\]
\[
\text{DO 20 I = N1+1, N}
\]
\[
\text{VS}
\]
\[
\text{S to S}
\]
\[
\text{DO 10 I = 1, N1}
\]
\[
\text{DO 10 I = 1, N1}
\]

Index set splitting: \( N \leftarrow N1 + N2 \)

We have already studied linear loop transformations.

Iteration renumbeiring transformations
Note: distance/direction are not adequate abstractions.

\[ mI \]

**DO-ALL** loop that does all iterations after

**iteration** by itself

\[ mI \]

**DO-ALL** loop that does all iterations before

**Split index set of loop into 3 parts**

\[ \eta / (q - \epsilon) = mI \leftarrow \]

**Weak SIV subscript dependence equation is**

\[ \epsilon = q + mI \]

\[ \text{as } \cdots + (c^T \eta (q) = x(q) + x(q) = q) \]

\[ \text{DO } I = 1, N \]

**Typical use:** Eliminate a problem iteration.
Strip-mining is always legal.

$S = \text{vector register length}$

Important transformation for vector machines:

Inner loop does $S$ iterations at a time.

Strip-mining Loop: strip size = 2

Original Loop

$\text{strip-mining: } N \times N' = N$

7

20 Yi(I) X(I) = (N div s) x i + i m N to N

DO 20 I = 1 TO N

I + (I) X(I) = (I)

I = I + s, IS = 1, s = (N div s)

DO 10 I = 1 TO IS

<=

I + (I) X(I) = (I)

(N min(I), s - I, N, I) = 1, s, N

DO 10 I = 1 TO IS

separately: index-set splitting

To get clean bounds for inner loop, do last 'N mod s' iterations
Old names for tilings: strip-packing and interchange loop quantization

\[
\begin{align*}
S & \ \\
\cdots & = j \ do \ \leftarrow \\
\cdots & = i \ do \ \rightarrow \\
\cdots & = \text{IF} \ do \ \leftarrow \\
\cdots & = i \ do \ \rightarrow \\
\end{align*}
\]

\[\forall N \exists N \exists I \forall N \times N \times I = \text{I}\]
Statement Sinking: useful for converting some imperfectly-nested loops into perfectly-nested ones
1. Execute a pre/post-iteration of loop in which only sunk

2. Requires insertion of guards for all statements in new loop.

Locality enhancement of SNL's in MIPSPro compiler:

Locality-enhanced perfectly-nested loop, and

convert back to imperfectly-nested loop in code generation.

Singly-nested loop (SNL) imperfectly-nested loop in which each loop has only one other loop nested immediately within it.
Loop Fusion: Promote reuse, eliminate array temporaries

\[
\begin{align*}
\cdots & \quad (\gamma(I)_{-1})_{20} \\
&\text{DOALL 20 } I = 1,100 \\
\cdots & \quad (\gamma(I)_{-1})_{10} \\
&\text{DO } 10 \ I = 1,100 \\
\cdots & \quad (\gamma(I))_{10} \\
&\text{DO } 10 \ I = 1,100 \\
\end{align*}
\]

Utility or distribution: Can produce parallel loops as below

\[
\begin{align*}
\cdots & \quad (\gamma(I))_{20} \\
&\text{DO } 20 \ J = 1,100 \\
\cdots & \quad (\gamma(I))_{10} \\
&\text{DO } 10 \ I = 1,100 \\
\cdots & \quad (\gamma(I))_{10} \\
&\text{DO } 10 \ I = 1,100 \\
\end{align*}
\]

Example: loop fusion/distribution

Statement Reordering Transformations

loop fusion/distribution ⇔ loop fusion/distribution
Nested loop fission: do in inside-out order, treating inner loop nests as black boxes

Order of new loop nests: any topological sort of acyclic condensate

Each node in acyclic condensate can become one loop nest

Find the acyclic condensate of statement dependence graph

edges: dependences between statements (distance/direction is irrelevant)
nodes: assignment statements/If-then-else's

Build the statement dependence graph:

<table>
<thead>
<tr>
<th>New Code</th>
<th>Acyclic Condensate</th>
<th>Statement Dependence Graph</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(I) = sgn(C(I))</td>
<td></td>
<td></td>
<td>D(I) = sgn(C(I))</td>
</tr>
<tr>
<td>DO I = 1,N</td>
<td></td>
<td></td>
<td>C(I) = I/B(I)</td>
</tr>
<tr>
<td>A(I) = V(I) + B(I-1)</td>
<td></td>
<td></td>
<td>B(I) = C(I-1) + I + X* (C(I-1))</td>
</tr>
<tr>
<td>DO I = 1,N</td>
<td></td>
<td></td>
<td>A(I) = A(I) + B(I-1)</td>
</tr>
<tr>
<td>C(I) = I/B(I)</td>
<td></td>
<td></td>
<td>D(I) = sqrt(C(I))</td>
</tr>
<tr>
<td>DO I = 1,N</td>
<td></td>
<td></td>
<td>DO I = 1,N</td>
</tr>
</tbody>
</table>

Loop 1

Loop 2

Loop 3

Legality of Loop Fission: build the statement dependence graph.
Legality of loop fusion:

(iii) no fusion-preventing dependence
(ii) loops are adjacent
(i) loop bounds are identical

Loop fusion is illegal if:

\[ \text{Illegal if } \]

Usually, we do not compute dependences across different loop nests.

Flow dependence test for fusion preventing dependence:

Easy to compute though:

\[ I_w = J_r + 1 \]

\[ J_r < I_w \]
Standard transformation: scalar/array expansion (shown above)

Eliminate resource dependence: eliminate storeage reuse.

"Storeage reuse" in imperative languages (cf. functional languages).

Ani- and output-dependence (resource dependences) arise from

\[
[I]T^*([I]T[I]T)_{10} = (J, I)X_{10}
\]

\[
(I)_{\mathbb{F}} = [I]T
\]

\[
\text{DO 10 I = 1, 100}
\]

\[
[I]T^* = (J, I)X_{10}
\]

\[
(I)_{\mathbb{F}} = [I]T
\]

\[
\text{DO 10 I = 1, 100}
\]

Example: scalar expansion

Statement transformation:
these transformations.

applied to other key routines to get a feel for the issues in applying

Let us study how imperfectly-nested loop transformations can be

product and matrix-matrix multiplication.

effect of interchange and tilting on key kernels like matrix-vector

We got into perfectly-nested loop transformations by studying the
Hemingway in “Death in the Afternoon” and the only heritage he has to leave.

know them the little new that each man gets from life is very costly are the very simplest things and because it takes a man’s life to which is all we have, must be paid heavily for their acquiring. They are some things which cannot be learned quickly, and time;
obtain $x$ by solving two triangular systems

part of $A$ with $\mathcal{L}^T \mathcal{L}^T = \mathcal{L}^T \mathcal{L}$, overwriting lower-triangular

compute such that $\mathcal{L}^T \mathcal{L}$ is symmetric positive-definite $A$

used to solve a system of linear equations viewpoint:

Cholesky factorization from a numerical analyst's viewpoint:
Transformations like loop distribution.

Variations of these 6 basic versions can be generated by

All 6 permutations of these loops are legal.

Imperfectly-nested.

Cholesky factorization has 6 loops like MIN, but loops are

Cholesky factorization from a compiler writer's viewpoint:
Note: most data references and computations in update.

Fasterly update portion of matrix to right of current column.

Compute columns of L column-by-column (indexed by k).

Update statements.

Three assignment statements are called square root, scale and

\[
\begin{align*}
\text{update statement} & : \quad A(1:j,k) = A(1:j,k) * A(1,k) \\
\text{scale statement} & : \quad \frac{A(k,k)}{A(1,k)} = A(k,k) \\
\text{square root statement} & : \quad \sqrt{A(k,k)} = \text{sqrt}(A(k,k))
\end{align*}
\]
\[(\mathcal{A}, \mathcal{E}) \forall \mathcal{A} \exists \mathcal{C}, \mathcal{F} = \mathcal{C}
\]

\[\text{N, } i = \mathcal{C} \quad \text{N, } \mathcal{F} = \mathcal{C} \quad \text{N, } \mathcal{K}_i = \mathcal{C} \quad \text{N, } \mathcal{K}_{i+1} = \mathcal{C}
\]

\[(\mathcal{A}, \mathcal{C}) \forall \mathcal{A} \exists \mathcal{C}, \mathcal{F} = \mathcal{C}
\]

\[\text{N, } i = \mathcal{C} \quad \text{N, } \mathcal{F} = \mathcal{C} \quad \text{N, } \mathcal{K}_i = \mathcal{C} \quad \text{N, } \mathcal{K}_{i+1} = \mathcal{C}
\]

\[\text{Update is performed row by row.}
\]

\[\bigcup \text{Interchanging } i \text{ and } j \text{ loops in } K_i \text{ gives } K_j \text{ version.}
\]
\[\forall \{\tau\}, \forall \{\rho\} \forall \tau \in \{\tau\}, \forall \rho \in \{\rho\} \text{ do } i = 1, \ldots, N \text{ do } k = 1, \ldots, N \text{ do }\]

Fusion of the two \( \tau \) loops in \( K \) version produces a SNT.
Update current column.

To compute column \( j \)’s portion of matrix to left of column is used to update:

- Updates to column are done lazily, not eagerly.

- Compute columns of \( I \)-column-by-column.

\[
(\forall j) \quad \forall (i,j,k) = (\forall (i,j), k) * (\forall (j,k)) \\
\text{do } k = 1 \text{ to } j-1 \\
\text{do } i = 1 \text{ to } N, \\
\text{do } j = 1 \text{ to } N, \\
\text{// interchange } i \text{ and } k \text{ loops for } j^{th} \text{ version}
\]

Column Cholesky: jik Left-Looking Versions
Compute the matrix \( A \) by row. Here is the \( i \)-th version of the Cholesky factorization:

\[
((1,1), A) = \begin{cases}
    \sqrt{A_{11}}, & \text{if } i = 1 \\
    \frac{A_{ij}}{A_{jj}}, & \text{if } i > j \\
    A_{ij} - \text{scale} + \text{update element} x, & \text{if } i = j
\end{cases}
\]

For each element in row \( i \):

- Take square-root at end.
- Scale.
- Update element \( x \).
- Find inner-product of two blue vectors.

Row Cholesky versions.
delaying all the updates till the end.

Unfortunately, loop distribution is not legal because it requires update loops.

Update loops.

Ideal situation: distribute loops to isolate update and the

Most of data accesses are in update step.

Locality enhancement in Cholesky factorization
update statement

\[ A(i', j) = A(i, j) \times A(k, i') \]

\( i = 1 \rightarrow N \) \( j = 1 \rightarrow N \) \( k = 1 \rightarrow N \)

scale statement

\[ A(i, k') = A(i, k) \times A(k', i) \]

\( i = 1 \rightarrow N \) \( k = 1 \rightarrow N \)

square root statement

\[ \sqrt{A(k, k')} = \sqrt{A(k, k) \times A(k', k)} \]

\( i = 1 \rightarrow N \) \( k = 1 \rightarrow N \) \( k' = 1 \rightarrow N \)

Do not distribute integer because of dependencies
\[(+',0,+') : ((f',k) \forall \langle a, i \rangle,\forall)\]
\[(+',0,+') : ((f',k) \forall \langle a, i \rangle,\forall)\]
\[(0',0,+') : ((\langle a, i \rangle,\forall)\forall)\]

Dependence vectors:

\[\forall (\langle a, i \rangle) \leftarrow \forall (\langle a, i \rangle) * (\forall (\langle a, i \rangle) \forall)\]

\[
\text{update statement} \]

//

\[
\text{do } j = k + 1, i \\
\text{do } i = k + 1, n \\
\text{do } k = 1, n
\]

obtained great performance...

After distribution, we could have the update statement, and
Let us study two distinct approaches to locality enhancement of Cholesky factorization.

 transformations to permit timing of imperfectly-nested code

 Cholesky factorization: Improvement of BLAS-3 content
 transformations to extract MM computations hidden within

 Cholesky factorization:
Key point: apply all delayed updates to current block-column

1. Perform square root, scale and local update steps on current block-column.
2. Perform processing of a block-column:
   - Do updates to block-columns lazily
   - Do processing on block-columns

Key idea used in LAPACK library: "Partial distribution"
How do we think about this in terms of loop transformations?
Guards are mutually exclusive, so order of statement is irrelevant.

Loop nest is fully permutable, and

Easy to show that

\[
\begin{align*}
\forall \mathbf{v}(\mathbf{r}, \mathbf{t}) & \Rightarrow \forall \mathbf{v}(\mathbf{r}, \mathbf{t}) \quad \forall (\mathbf{r}, \mathbf{t}) \quad \forall (\mathbf{r}, \mathbf{t}) \quad \forall (\mathbf{r}, \mathbf{t})
\end{align*}
\]

Perfectly-nested loop that performs Cholesky factorization:

Intermediate representation of Cholesky factorization
Converting other versions: much more challenging...

- Use code sinking.
- Apply loop fusion to loops to get SNT, and
- Interchange i and j loops to get Kfi version.

Converting Kfi version: only requires code sinking.

Converting Kfi-Fused version: only requires code sinking.

Generative intermediate form of Cholesky:
Nested loop in the following form:

Convenient to express loop bounds of fully permutable perfectly
2. Use handwritten codes to execute the BLAS-3 kernels

1. Convert to block-column computations to expose BLAS-3

LAPACK-style blocking of intermediate form
\[ \text{(1)} \quad \text{Stirp} \text{p} \text{i} \text{n} \text{e} \text{t} \text{h} \text{e} \text{ } \text{J} \text{ } \text{l} \text{oo} \text{p} \text{ } \text{i} \text{n} \text{t} \text{o} \text{ } \text{b} \text{l} \text{o} \text{c} \text{k} \text{ } \text{c} \text{o} \text{l} \text{u} \text{m} \text{n} \text{s} \text{.} \]

\[ \text{(2)} \quad \text{I} \text{nt} \text{e} \text{r} \text{n} \text{c} \text{h} \text{a} \text{n} \text{e} \text{t} \text{h} \text{e} \text{ } \text{J} \text{ } \text{l} \text{oo} \text{p} \text{ i} \text{n} \text{t} \text{o} \text{ } \text{i} \text{n} \text{t} \text{e} \text{r} \text{m} \text{o} \text{s} \text{t} \text{ } \text{p} \text{o} \text{s} \text{i} \text{t} \text{i} \text{o} \text{n}: \]

\[ \text{do } \{ \text{t'} \text{, r'} \} \text{ in } \text{J} \text{ } \text{i} \text{n} \text{t} \text{o} \text{ } \text{J} \text{ } \text{L} \text{oop} \]
\( (\forall \phi \forall (\lambda_j \lambda_k) \forall (\lambda_1 \lambda_2) \text{ if } \lambda_1 < \lambda_2 < \lambda_3 \text{ then } ) \)
\( (\forall \chi \forall (\lambda_j \lambda_k) (\forall (x_k \chi) \forall (\lambda_1 \lambda_2) = (\lambda_1 \lambda_2) ) \text{ if } \lambda_1 < \lambda_2 < \lambda_3 \text{ then } ) \)
\( (\forall (\lambda_j \lambda_k) \forall (\lambda_1 \lambda_2) \forall (x_k \chi) \forall (\lambda_1 \lambda_2) \text{ if } \lambda_1 < \lambda_2 < \lambda_3 \text{ then } ) \)
\( (\forall (\lambda_j \lambda_k) \forall (\lambda_1 \lambda_2) \forall (x_k \chi) \forall (\lambda_1 \lambda_2) \text{ if } \lambda_1 < \lambda_2 < \lambda_3 \text{ then } ) \)
\( (\forall (\lambda_j \lambda_k) \forall (\lambda_1 \lambda_2) \forall (x_k \chi) \forall (\lambda_1 \lambda_2) \text{ if } \lambda_1 < \lambda_2 < \lambda_3 \text{ then } ) \)

\[
\begin{align*}
\text{do } \lambda = 0, \text{ while } N/B - 1
\end{align*}
\]

\(" \text{Computation 2: a small Cholesky factorization}"

\[
\begin{align*}
\text{do } \lambda = 0, \text{ while } N/B - 1
\end{align*}
\]

\(" \text{Computation 1: an MMM}"

\[
\begin{align*}
\text{do } \lambda = 0, \text{ while } N/B - 1
\end{align*}
\]
\( \forall (\forall (x, y) \land (x < y)) \Rightarrow \)
\( \forall (\forall (x, y) \land (x = y)) \Rightarrow \)
\( \forall (\forall (x, y) \land (x = y)) \Rightarrow \)

\[
\text{do } j = b*fs+j \text{ do } k = b*fs+j \text{ do } i = b*fs+j \text{ do } n = b*fs+j \\
\text{compute an } MMM //
\]
Factorization.

- Only unblocked computations are in the small Cholesky.
- Compute a block triangular solve.
- Call BLAS-3 kernel.

execute them.

Computations 1 and 3 are MM, call BLAS-3 kernel to

observations on code:
transformations?

How does a compiler synthesize such a complex sequence of

How do we recognize BLAS-3 operations when we expose them?

complex code?

How does a compiler where BLAS-3 computations are hiding in

Critique of this development from compiler perspective:
Choose \( k_{ij} \) order to get good spatial locality.

Loop nest is is \( k_{ij} \) is fully permutable, as is \( i_{ij} \) Loop nest.

\[
\begin{align*}
\forall (i', j', k') \in \mathbb{N} \times \mathbb{N} \\
&
\forall (i, j, k) \in \mathbb{N} \\
&
\forall (i, j, k) \in \mathbb{N} \\
&
\forall (i, j, k) \in \mathbb{N} \\
\end{align*}
\]

- \( bs + b \rightarrow bs \mathbf{b} \)
- \( bs + b \rightarrow bs \mathbf{j} \rightarrow j \mathbf{b} \rightarrow b \)
- \( bs + b \rightarrow bs \mathbf{j} \rightarrow j \mathbf{b} \rightarrow b \)

\[
\begin{align*}
\text{do} & \{ i' \mathbf{j'}, k' \} \\
\text{do} & \{ j' \mathbf{is}, k' \} \\
\text{do} & \{ k' \mathbf{is}, j' \mathbf{is}, k' \} \\
\end{align*}
\]

Tile the \text{fully} permutable intermediate form of Cholesky.

Compiler approach:
How do we make all this work smoothly?

3. Convert resulting perfectly-nested loop into a perfectly-nested loop by index-splittling/peeling.
2. Transform intermediate form as before to enhance locality.
1. Convert an imperfectly-nested loop into a perfectly-nested loop.

Strategy for locality-enhancement of imperfectly-nested loops: