Transforming Imperfectly Nested Loops

Classes of loop transformations:

- Iteration re-numbering: (eg) loop interchange

  Example

```
DO 10 J = 1,100                 DO 10 I = 1,100
  DO 10 I = 1,100         vs     DO 10 J = 1,100
    Y(I) = Y(I)+A(I,J)*X(J)        Y(I) = Y(I) + A(I,J)*X(J)
10  Z(I) = ....                 10 Z(I) = ....
```

  All statements in body affected identically.

- Statement re-ordering: (eg) loop distribution/jamming

  Example

```
DO 10 I = 1,100                 DO 10 I = 1,100
     Y(I) = ....                  10 Y(I) = ...
 10  Z(I) = ...Y(I)...   vs.    DO 20 J = 1,100
                                  20 Z(I) = ...Y(I)..
```

  Statement re-ordering can be static or dynamic

- Statement transformation:

  Example: scalar expansion

```
DO 10 I = 1,100                     DO 10 I = 1,100
  T = f(I)              vs            T[I] = f(I)
10 X(I,J) = T*T                     10 X(I,J) = T[I]*T[I]
```

  Statements themselves are altered.

Iteration renumbering transformations

We have already studied linear loop transformations.

Index set splitting: $N \to N1 + N2$

```
DO 10 I = 1, N            DO 10 I = 1, N1
10   S                    10   S
                    vs
                          DO 20 I = N1+1, N
                          10   S
```

Special case: loop peeling - only the first/last/both first and last iterations are done separately from main loop.

Legality: always legal

Typical use: Eliminate a ‘problem iteration’

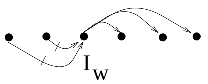
```
DO 10 I = 1, N
10 X(aI + b) = X(c) + ....   vs
```

Weak SIV subscript: dependence equation is $aI_w + b = c$

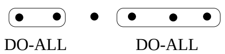
$$\Rightarrow I_w = (c - b)/a$$

Split index set of loop into 3 parts:

- DO-ALL loop that does all iterations before I_w
- Iteration I_w by itself
- DO-ALL loop that does all iterations after I_w



Original Loop

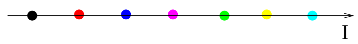


After Index-set Splitting

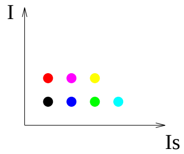
Note: distance/direction are not adequate abstractions

Strip-mining: $N = N1 * N2$

```
DO 10 I = 1, N          DO 10 Is = 1, N, s
10 Y(I) = X(I)+1   =>     DO 10 I = Is, min(Is + s - 1, N)
                          10  Y(I) = X(I) + 1
```



Original Loop



Stripmined Loop: strip size = 2

Inner loop does ‘s’ iterations at a time.
Important transformation for vector machines:
‘s’ = vector register length
Strip-mining is always legal.

To get clean bounds for inner loop, do last ‘N mod s’ iterations separately: index-set splitting

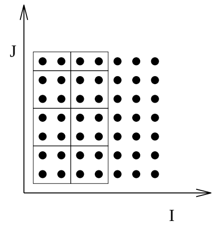
```
DO 10 Is = 1, N, s
  DO 10 I = Is, min(Is + s - 1, N)
  10  Y(I) = X(I) + 1


   =>


DO 10 Is = 1, s*(N div s)
 DO 10 I = Is, Is + s - 1
 10 Y(I) = X(I) + 1


DO 20 I = (N div s)*s + 1 to N
   20 Y(I) = X(I) + 1
```


Tiling: multi-dimensional strip-mining $N1XN2 = t1 * t2 * N3 * N4$




```
DO  I = ...               DO Ti = ...
  DO J = ...        =>      DO Tj = ...
    S                          DO I = ...
                                 DO J = ...
                                   S
```

Old names for tiling: stripmine and interchange, loop quantization

Statement Sinking: useful for converting some imperfectly-nested loops into perfectly-nested ones

```
do k = 1, N
  A(k,k) = sqrt (A(k,k))
  do i = k+1, N
    A(i,k) = A(i,k) / A(k,k) <---- sink into inner loop
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)

=>

do k = 1, N
  A(k,k) = sqrt (A(k,k))
  do i = k+1, N
    do j = k, i
      if (j==k) A(i,k) = A(i,k) / A(k,k)
      if (j!=k) A(i,j) -= A(i,k) * A(j,k)
```


Basic idea of statement sinking:

1. Execute a pre/post-iteration of loop in which only sunk statement is executed.
2. Requires insertion of guards for all statements in new loop.

Singly-nested loop (SNL): imperfectly-nested loop in which each loop has only one other loop nested immediately within it.

Locality enhancement of SNL's in MIPSPro compiler:

- convert to perfectly-nested loop by statement sinking,
- locality-enhance perfectly-nested loop, and
- convert back to imperfectly-nested loop in code generation.

Statement Reordering Transformations

loop jamming/fusion <=> loop distribution/fission

Example

```
DO 10 I = 1,100                DO 10 I = 1,100
     Y(I) = ....                 10 Y(I) = ...
 10  Z(I) = ...Y(I)...    vs.  DO 20 J = 1,100
                                 20 Z(I) = ...Y(I)..
```

Utility of distribution: Can produce parallel loops as below

```
DO 10 I = 1, 100               DOALL 10 I = 1,100
    Y(I) = ....           vs.  10 Y(I) = ....
10  Z(I) = Y(I-1)....          DOALL 20 I' = 1,100
                               20 Z(I') = Y(I'-1) .....
```

Loop fusion: promote reuse, eliminate array temporaries

Legality of loop fission: build the statement dependence graph

```
DO I = 1,N
  A(I) = A(I) + B(I-1)
  B(I) = C(I-1)*X + 1
  C(I) = 1/B(I)
  D(I) = sqrt(C(I))
```

Program

Statement Dependence Graph

Acyclic Condensate

```
DO I = 1,N
  B(I) = C(I-1)*X +1
  C(I) = 1/B(I)
DO I = 1,N
  A(I) = A(I)+B(I-1)
DO I = 1,N
  D(I) = sqrt(C(I))
```

New Code

- Build the statement dependence graph:
  nodes: assignment statements/if-then-else's
  edges: dependences between statements (distance/direction is irrelevant)
- Find the acyclic condensate of statement dependence graph
- Each node in acyclic condensate can become one loop nest
- Order of new loop nests: any topological sort of condensate
- Nested loop fission: do in inside-out order, treating inner loop nests as black boxes

Statement transformation:

Example: scalar expansion

```
DO 10 I = 1,100                 DO 10 I = 1,100
  T = f(I)            vs          T[I] = f(I)
10 X(I,J) = T*T                 10 X(I,J) = T[I]*T[I]
```

Anti- and output-dependences (*resource dependences*)arise from "storage reuse" in imperative languages (cf. functional languages).

Eliminating resource dependences: eliminate storage reuse.

Standard transformations: scalar/array expansion (shown above)

Automatic Blocking of Cholesky Factorization

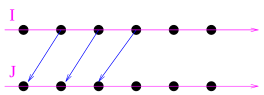
“There are some things which cannot be learned quickly, and time, which is all we have, must be paid heavily for their acquiring. They are the very simplest things and because it takes a man’s life to know them the little new that each man gets from life is very costly and the only heritage he has to leave.”

Hemingway in "Death in the afternoon"

Legality of loop fusion:

```
DO   I = 1,N
  X(I) = .....

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



illegal

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

Usually, we do not compute dependences across different loop nests.
Easy to compute though:

Flow dependence: test for fusion preventing dependence

$$\boxed{\begin{array}{l} Iw = Jr + 1 \\ Jr < Iw \\ 1 \leq Iw \leq N \\ 1 \leq Jr \leq N \end{array}}$$

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

We got into perfectly-nested loop transformations by studying the effect of interchange and tiling on key kernels like matrix-vector product and matrix-matrix multiplication.

Let us study how imperfectly-nested loop transformations can be applied to other key routines to get a feel for the issues in applying these transformations.

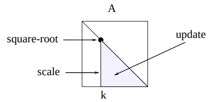
Cholesky factorization from a numerical analyst's viewpoint:

- used to solve a system of linear equations $Ax = b$
- A must be symmetric positive-definite
- compute L such that $L * L^T = A$, overwriting lower-triangular part of A with L
- obtain x be solving two triangular systems

Cholesky factorization from a compiler writer's viewpoint:

- Cholesky factorization has 6 loops like MMM, but loops are imperfectly-nested.
- All 6 permutations of these loops are legal.
- Variations of these 6 basic versions can be generated by transformations like loop distribution.

Column Cholesky: kij, right-looking versions




```
do k = 1, N
  A(k,k) = sqrt (A(k,k))  //square root statement
  do i = k+1, N
    A(i,k) = A(i,k) / A(k,k) //scale statement
  do i = k+1, N
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k) //update statement
```

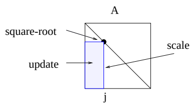
- Three assignment statements are called square root, scale and update statements.
- Compute columns of L column-by-column (indexed by `k`).
- Eagerly update portion of matrix to right of current column.
- Note: most data references and computations in update.

Interchanging i and j loops in `kij` version gives `kji` version.

Update is performed row by row.

```
do k = 1, N
  A(k,k) = sqrt (A(k,k))
do i = k+1, N
  A(i,k) = A(i,k) / A(k,k)
do j = k+1, N
  do i = j, N
    A(i,j) -= A(i,k) * A(j,k)
```

Column Cholesky: jik left-looking versions




```
do j = 1, N
  do i = j, N    //interchange i and k loops for jki version
    do k = 1, j-1
      A(i,j) -= A(i,k) * A(j,k)
  A(j,j) = sqrt (A(j,j))
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)
```

- Compute columns of L column-by-column.
- Updates to column are done lazily, not eagerly.
- To compute column `j`, portion of matrix to left of column is used to update current column.

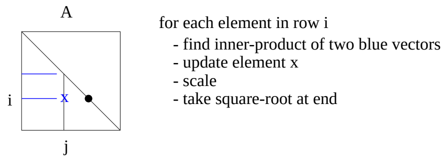
Locality enhancement in Cholesky factorization

- Most of data accesses are in update step.
- Ideal situation: distribute loops to isolate update and tile update loops.
- Unfortunately, loop distribution is not legal because it requires delaying all the updates till the end.

Fusion of the two `i` loops in `kij` version produces a SNL.

```
do k = 1, N
  A(k,k) = sqrt (A(k,k))
  do i = k+1, N
    A(i,k) = A(i,k) / A(k,k)
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
```


Row Cholesky versions



These compute the matrix L row by row. Here is `ijk`-version of row Cholesky.

```
do i = 1, N
  do j = 1, i
    do k = 1, j-1
       A(i,j) -= A(i,k) * A(j,k)
    if (j < i) A(i,j) = A(i,j)/A(j,j)
    else       A(i,i) = sqrt (A(i,i))
```

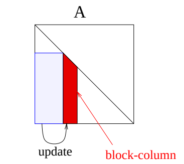
After distribution, we could have tiled update statement, and obtained great performance....

```
do k = 1, N
  do i = k+1, N
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k) //update statement
```

Dependence vectors:

```
(A(i,j) -> A(i,j)):    (+,0,0)
(A(i,j) -> A(i,k)):    (+,0,+)
(A(i,j) -> A(j,k)):    (+,0+,+)
```


Key idea used in LAPACK library: "partial" distribution



- do processing on block-columns
- do updates to block-columns lazily
- processing of a block-column:
  1. apply all delayed updates to current block-column
  2. perform square root, scale and local update steps on current block column
- Key point: applying delayed updates to current block-column can be performed by calling BLAS-3 matrix-matrix multiplication.


```
  do k = 1, N
    A(k,k) = sqrt (A(k,k))  //square root statement
    do i = k+1, N
      A(i,k) = A(i,k) / A(k,k) //scale statement
    do i = k+1, N
      do j = k+1, i
        A(i,j) -= A(i,k) * A(j,k) //update statement

=> loop distribution (illegal because of dependences)

  do k = 1, N
    A(k,k) = sqrt (A(k,k))  //square root statement
    do i = k+1, N
      A(i,k) = A(i,k) / A(k,k) //scale statement
  do k = 1, N
    do i = k+1, N
      do j = k+1, i
        A(i,j) -= A(i,k) * A(j,k) //update statement
```


Let us study two distinct approaches to locality enhancement of Cholesky factorization:

- transformations to extract MMM computations hidden within Cholesky factorization: improvement of BLAS-3 content
- transformations to permit tiling of imperfectly-nested code

How do we think about this in terms of loop transformations?

Intermediate representation of Cholesky factorization

Perfectly-nested loop that performs Cholesky factorization:

```
do k = 1, N
  do i = k, N
    do j = k, i
      if (i == k && j == k) A(k,k) = sqrt (A(k,k));
      if (i <  k && j == k) A(i,k) = A(i,k) / A(k,k);
      if (i >  k && j >  k) A(i,j) -= A(i,k) * A(j,k);
```

Easy to show that

- loop nest is fully permutable, and
- guards are mutually exclusive, so order of statement is irrelevant.

Generating intermediate form of Cholesky:

Converting kij-Fused version: only requires code sinking.

Converting kji version:

- interchange i and j loops to get kij version,
- apply loop fusion to i loops to get SNL, and
- use code sinking.

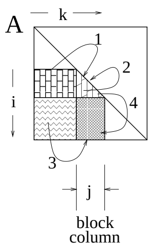
Converting other versions: much more challenging....

Convenient to express loop bounds of fully permutable perfectly nested loop in the following form:

```
do {i,j,k} in 1 <= k <= j <= i <= N

  if (i == k && j == k) A(k,k) = sqrt (A(k,k));
  if (i >  k && j == k) A(i,k) = A(i,k) / A(k,k);
  if (i >  k && j  > k) A(i,j) -= A(i,k) * A(j,k);
```

LAPACK-style blocking of intermediate form



Computation 1: MMM
Computation 2: unblocked Cholesky
Computation 3: MMM
Computation 4: Triangular solve

Two levels of blocking:

1. convert to block-column computations to expose BLAS-3 computations
2. use handwritten codes to execute the BLAS-3 kernels

(1) Stripmine the j loop into blocks of size B:

```
do js = 0, N/B -1  //js enumerates block columns
  do j = B*js +1, B*js+B
    do {i,k} in 1 <= k <= j <= i <= N

     if (i == k && j == k) A(k,k) = sqrt (A(k,k));
     if (i >  k && j == k) A(i,k) = A(i,k) / A(k,k);
     if (i >  k && j  > k) A(i,j) -= A(i,k) * A(j,k);
```

(2) Interchange the j loop into the innermost position:

```
do js = 0, N/B -1
  do i = B*js +1, N
    do k = 1, min(i,B*js+B)
      do j = max(B*js +1,k), min(i,B*js+B)
         if (i == k && j == k) A(k,k) = sqrt (A(k,k));
         if (i >  k && j == k) A(i,k) = A(i,k) / A(k,k);
         if (i >  k && j  > k) A(i,j) -= A(i,k) * A(j,k);
```


(3) Index-set split i loop into B*js +1:B*js +B and B*js +B+1:N.
(4) Index-set split k loop into 1:B*js and B*js +1:min(i,B*js+B).

```
do js = 0, N/B -1

   //Computation 1: an MMM
   do i= B*js +1, B*js +B
      do k = 1,B*js
         do j = B*js +1,i
            A(i,j) -= A(i,k) * A(j,k);

    //Computation 2: a small Cholesky factorization
   do i = B*js +1,B*js +B
      do k = B*js+1,i
         do j = k,i
            if (i == k && j == k) A(k,k) = sqrt (A(k,k));
            if (i >  k && j == k) A(i,k) = A(i,k) / A(k,k);
            if (i >  k && j  > k) A(i,j) -= A(i,k) * A(j,k);
```



```
//Computation 3: an MMM
do i = B*js+ B+1,N
   do k = 1,B*js
      do j = B*js+1,B*js+B
          A(i,j) -= A(i,k) * A(j,k);

//Computation 4: a triangular solve
do i = B*js+ B+1,N
   do k = B*js+1,B*js+B
      do j = k,B*js+B
         if (j == k) A(i,k) = A(i,k) / A(k,k);
         if (j  > k) A(i,j) -= A(i,k) * A(j,k);
```

Observations on code:

- Computations 1 and 3 are MMM. Call BLAS-3 kernel to execute them.
- Computation 4 is a block triangular-solve. Call BLAS-3 kernel to execute it.
- Only unblocked computations are in the small Cholesky factorization.

Critique of this development from compiler perspective:

- How does a compiler where BLAS-3 computations are hiding in complex codes?
- How do we recognize BLAS-3 operations when we expose them?
- How does a compiler synthesize such a complex sequence of transformations?

Compiler approach:

Tile the fully-permutable intermediate form of Cholesky:

```
do {is,js,ks}   0 <= ks <= js <= is <= N/B -1

  do {i,j,k}    B*is < i <= B*is + B
                B*js < j <= B*js + B
                B*ks < k <= B*ks + B

    if (i == k && j == k) A(k,k) = sqrt (A(k,k));
    if (i >  k && j == k) A(i,k) = A(i,k) / A(k,k);
    if (i >  k && j  > k) A(i,j) -= A(i,k) * A(j,k);
```

- Loop nest is,js,ks is fully permutable, as is i,j,k loop nest.
- Choose k,j,i order to get good spatial locality.

Strategy for locality-enhancement of imperfectly-nested loops:

1. Convert an imperfectly-nested loop into a perfectly-nested intermediate form with guards by code sinking/fusion/etc.
2. Transform intermediate form as before to enhance locality.
3. Convert resulting perfectly-nested loop with guards back into imperfectly-nested loop by index-set splitting/peeling.

How do we make all this work smoothly?