CS 526

Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526

DEPENDENCE TRANSFORMS

The slides adapted from Vikram Adve

Reordering Transformation

Definition. Legal Transformation preserves the meaning of that program, i.e., **all externally visible outputs are identical to the original program**, and in identical order.

- We consider two programs equivalent (i.e., the transformation preserving the program meaning) if on the same inputs both the original and transformed programs, after being executed, produce the same outputs.

Theorem. A ***reordering*** transformation that preserves all data dependences in a program is a ***legal*** transformation.

- See Lecture 6 for an argument why.

Motivation

Memory hierarchy optimizations
Goal 1: Improving reuse of data values within loop nest
Goal 2: Exploit reuse to reduce cache, TLB misses

Tiling
Goal 1: Exploit temporal reuse when data size > cache size
Goal 2: In parallel loops, reduce synchronization overhead

Software Prefetching
Goal: Prefetch predictable accesses k iterations ahead

Software Pipelining
Goal: Extract ILP from multiple consecutive iterations

Automatic parallelization Also, auto-vectorization
Goal 1: Enhance parallelism
Goal 2: Convert scalar loop to explicitly parallel
Goal 3: Improve performance of parallel code

Reordering Transformations

Name	Purpose	Benefit
	Preprocessing transformations	
Loop normalization	Make loops canonical	Simplify,improve dep. analysis
Ind. var. substitution	Identify aux. induction vars	Improve dependence information
Scalar expansion	Replace scalar with array	Eliminate spurious dependences
Scalar/array privatization	Treat var. as iteration-private	Eliminate spurious dependences
Variable renaming	Use multiple copies of vars	Eliminate anti- and output-dependences
Reduction recognition	Recognize reductions	Ignore special-case dependences
	Reordering transformations	
Loop interchange	Change loop nesting order	Cache, parallelism, vectorization
Loop strip-mining	Make 2 nested loops	”
Loop skewing	Change wavefront loop to parallel	Improve loop parallelism
Loop reversal	Run loop backwards	Reduce array storage
Index set splitting	Break loop by index space	Remove some deps.
Loop distribution	Break loop by statements	Simplify parallelization, vectorization
Loop alignment	Change carried to indep.	Simplify parallelization, vectorization
Loop fusion	Join loops by statements	Improve cache reuse

Control-Flow Analysis

Consider now a program with conditionals:

```
for j = 1 to n {
  A[j] = A[j] * C[j]   // S1
  if (A[j] > k)
     B[j] = B[j] + D[j]   // S2
  else
     B[j] = B[j] - 1.0f
}
```

Control flow dependency exists between S1 and S2
(B[j] will be assigned the value only if A[j] has some value)

Control-Flow Analysis

We can convert the control dependency into a data dependency. Key steps:

- Consider ***guarded statements*** (if (bool_var) Stmt) and
- Transform the program to ***extract*** complicated expressions from the conditionals

```
for j = 1 to n {
  A[j] = A[j] * C[j]   // S1
  m = A[j] > k
  if (m) B[j] = B[j] + D[j]
  if (!m) B[j] = B[j] – 1.0f
}
```

Control-Flow Analysis (Forward)

```
for j = 1 to n {
  A[j] = A[j] * C[j]   // S1
  m = A[j] > k
  if (m) B[j] = B[j] + D[j]
  if (!m) B[j] = B[j] – 1.0f
}
```

The transformed program preserves all dependencies

This code can be readily vectorized:

- Compute the mask vector m[1…n]
- Compute the then branch result by filtering on m
- Compute the else branch result by filtering on m

E.g., SSE has operations that admit the mask.

Control-Flow Analysis (Exit)

```
for j = 1 to n {
  A[j] = A[j] * C[j]
  if (A[j] > k) break;
  B[j] = B[j] + D[j]
}
```

This is harder to transform with guarded form:

- If the condition is true once, exiting the loop is the same as if it fuly executed
- The condition depends on all iterations so far.
- Sketch of a solution. What is missing?

```
for j = 1 to n {
  if (m) break;
  A[j] = A[j] * C[j]
  m = m || A[j] > k
  if (m) break; // ?
  B[j] = B[j] + D[j]

}
```

```
for j = 1 to n {
  m1 = m2
  if (!m1) A[j] = A[j] * C[j]
  if (!m1) m2 = m2 || A[j] > k
  if (!m2) B[j] = B[j] + D[j]
}
```

Control-Flow Analysis (Backward)

```
for j = 1 to n {
      if (A[j] < m) continue;
  S1: k = k + 1
      A[j] = B[k] + D[j]
      if (A[j] > k) break;
}
```

Appears when there is an inner loop like structure

- Applying just the forward analysis would yield potentially wrong code when combined with forward analysis
- It is transformed in conjunction with the related forward branches
- Simple heuristic: identify all code affected by a backward branch untouched and treat as a black-box. However, inefficient; for a more powerful analysis see e.g., *Conversion of Control Dependence to Data Depemdence; J.R. Allen and Ken Kennedy; POPL 1983*

Loop Interchange

Informal Definition: Change nesting order of loops in a **perfect loop nest**, with no other changes.

```
do i=2, N
   do j=2, M-1
       A[i,j] = A[i,j]*2
    enddo
enddo
```

```
do j=2, M-1
   do i=2, N
       A[i,j] = A[i,j]*2
    enddo
enddo
```

Uses of Loop Interchange

1. Move independent loop innermost
2. Move independent loop outermost
3. Make accesses stride-1 in memory
4. Loop tiling (combine with strip-mining)
5. Unroll-and-jam (combine with unrolling)

Loop Interchange

Direction Vectors and Loop Interchange:

If δ is a direction vector of a particular dependence S1 → S2 in a loop nest and the order of loops in the loop nest is permuted, then the same permutation can be applied to δ to obtain the new direction vector for the conflicting instances of S1 and S2

Direction Matrix: A matrix where each row is the direction vector of a single dependence, i.e.,

each row ↔ a dependence

each column ↔ a loop

Direction Matrix

Direction Matrix:

each row ↔ a dependence

each column ↔ a loop

A[i,j]/A[i,j]	= =
A[i,j]/A[i-1,j]	+ =
B[i,j]/B[i-1,j-1]	+ +

```
do i=2, N
   do j=2, M-1
       A[i,j] = ... * B[i-1,j-1]
       B[i,j] = ... + A[i,j] + A[i-1,j]
   enddo
enddo
```

Direction Matrix (Illegal)

Direction Matrix:

each row ↔ a dependence

each column ↔ a loop

A[i,j]/A[i,j]	=	=
A[i,j]/A[i-1,j+1]	+	-
B[1,j]/B[i-1,j-1]	+	+

```
do i=2, N
   do j=2, M-1
      A[i,j] = ... * B[i-1,j-1]
      B[i,j] = ... + A[i,j] + A[i-1,j+1]
   enddo
enddo
```

Loop Interchange Properties

Legality: A permutation of the loops in a perfect nest is legal iff the direction matrix, <u>after the permutation is applied</u>, has no “**-**” direction as the leftmost non-“=“ direction in any row

Profitability: machine-dependent:

1. vector machines
2. parallel machines
3. caches with single outstanding loads
4. caches with multiple outstanding loads

Applying Loop Interchange

1. **Single** ’+’ entry: a “serial loop”
   - Move loop outermost for vectorization
   - Move loop innermost for parallelization

2. **Multiple** ’+’ entries: Outermost one carries dependence
   - Loop carrying the dependence *changes* after permutation!
   - May still benefit by moving carried-dependences to the outermost loop

Loop Reversal

Informal Definition: Reverse the order of execution of the iterations of a loop

```
do i=2 to N
  do j=2 to M-1
    do k=1 to L
      A[i,j,k] = A[i,j-1,k+1]
               + A[i-1,j,k+1]
    enddo
  enddo
enddo
```

```
do i=2 to N
  do j=2 to M-1
    do k=L to 1 step -1
      A[i,j,k] = A[i,j-1,k+1]
               + A[i-1,j,k+1]
    enddo
  enddo
enddo
```

Loop Reversal

```
do i=2, N
  do j=2, M-1
    do k=1, L
      A[i,j,k] = A[i,j-1,k+1]
               + A[i-1,j,k+1]
    enddo
  enddo
enddo
```

$$\boxed{\begin{matrix} = & + & - \\ + & = & - \end{matrix}}$$

```
do i=2, N
  do j=2, M-1
    do k=L, 1, -1
      A[i,j,k] = A[i,j-1,k+1]
               + A[i-1,j,k+1]
    enddo
  enddo
enddo
```

$$\boxed{\begin{matrix} = & + & + \\ + & = & + \end{matrix}}$$

Uses of Loop Reversal

Convert a ’-’ to a ’+’ in a direction vector to enable other transformations, e.g., loop interchange.

Scalarize a vector statement (e.g., in Fortran 90) by ensuring that values are read before being written.

- Vectorized code: `A[2:64] = A[1:63] * e`
- Scalarized code:

```
do i = 64, 2, -1
    A[i] = A[i-1] * e
enddo
```

Loop Distribution

Informal Definition: Convert a loop nest containing two or more statements into two or more distinct loop nests so that each statement appears in only a single resulting loop nest.

```
     do i=2,N
S1:     A[i] = B[i] + C[i]
S2:     D[i] = A[i] * 2.0
S3:     B[i+1] = A[i] * 3.0
    enddo
```

```
     do i=2,N
S1:     A[i] = B[i] + C[i]
S3:     B[i+1] = A[i] * 3.0
     enddo
     do i=2,N
S2:     D[i] = A[i] * 2.0
     enddo
```

Loop Distribution Applications

- Create perfect loops nests for other transformations like loop interchange
- Convert a loop-carried dependence within a loop into a loop-independent dependence crossing two loops:

```
     do i=2,N
S1:      A[i] = B[i] + C[i]
S2:      D[i] = A[i-1] * 2.0
     enddo
```

```
     do i=2,N
S1:      A[i] = B[i] + C[i]
     enddo
     do i=2,N
S2:      D[i] = A[i-1] * 2.0
     enddo
```

Maximal Loop Distribution

- Identify the SCCs of the data dependence graph, to group statements in an SCC in a single loop nest
- Sort the SCCs using a topological sort on the dependence graph
- Generate distinct loop nests, one for each SCC, in sorted order

Loop Fusion

Informal Definition: Merge two or more distinct (perhaps non-adjacent) loops with identical loop bounds into a single loop.

```
do i=1,N
   A[i] = i*i
enddo
do i=1,N
   B[i] = A[i] + 1
enddo
```

```
do i=1,N
   A[i] = i*i
   B[i] = A[i] + 1
enddo
```

Loop Fusion

```
do i=1,M
  do j=1,N-1
    A[j,i] = i*i + j*j
  enddo

  do j=1,N
    B[j,i] = A[j,i] + i + j
  enddo
enddo
```

```
do i=1,M
  do j=1,N-1
    A[j,i] = i*i + j*j
    B[j,i] = A[j,i] + i + j
  enddo
  // peel last iteration:
    j=N
    B[j,i] = A[j,i] + i + j
enddo
```

Loop Fusion Motivation

- Increase cache reuse (if same array accessed in two loops) Fundamental optimization for array languages (e.g., Fortran 90, HPF, MATLAB, APL)

  Example in F90:

```
A[1:M, 1:N] = B[1:M, 1:N] * 2
C[1:M, 1:N] = A[1:M, 1:N] + 1
```

- Increase granularity of parallelism (work per iteration) Important for shared-memory parallelism (the model with parallel loop and barriers)

Legality of Loop Fusion

Fusion-Preventing Dependence: A loop-independent dependence from S1 to S2 in different loops is fusion-preventing if fusing the two loops causes the dependence to become a loop-carried dependence from S2 to S1.

Legality of Loop Fusion: Two loops can be fused if all 3 conditions are satisfied:

1. Both have identical bounds (*transform loops if needed*)
2. There is no fusion-preventing dependence between them.
3. There is no path of loop-independent dependences between them that contains a loop or statement that is not being fused with them.

Loop Fusion: Illegal Cases

```
do i=1,M
  do j=2,N
    A[j,i] = B[j-1,i] * 2
  enddo

  do j=2,N
    B[j,i] = A[j,i] * 3
  enddo
enddo
```

```
do i=1,M
  do j=2,N
    t[j] = B[j-1,i]
  enddo

  do j=2,N
    A[j,i] = t[j] * 2
    B[j,i] = A[j,i] * 3
  enddo
enddo
```

Create temporary array to make fusion possible

Loop Strip Mining

Informal Definition Convert a single loop into two nested loops for a specified "block size" (*Always safe.*)

```
do i=1,N
     A[i] = x + B[i] * 2
enddo
```

```
do ii=1,N,B
   do i=ii, min(ii+B-1, N), 1
       A[i] = x + B[i] * 2
   enddo
enddo
```

Loop Strip Mining Applications

- **Loop tiling:** strip-mine and then interchange multiple uses. Can be useful for increasing cache locality or blocking parallel loops;
- **Prefetching:** strip-mine by cache line size; prefetch once per outer iteration
- **Instruction scheduling:** strip-mine and then unroll inner loop

Loop Alignment

Informal Definition: Eliminate a carried dependence by increasing the number of iterations and executing statements on different subsets of the iterations (*Always safe)*

```
do i=2 to N
   A[i] = B[i] + C[i]
   D[i] = A[i-1] * 2.0
enddo
```

```
i = 1
D[i+1] = A[i] * 2

do i=2 to N
   A[i] = B[i] + C[i]
   D[i+1] = A[i] * 2.0
enddo

i = N
A[i] = B[i] + C[i]
```

Scalar Replacement

Informal Definition: Replace an array reference with a scalar temporary. (Use dependences to locate consistent re-use patterns)

```
do i = 1 to n
   do j = 2 to n
      x(j,i) = a(i) +
               x(j-1,i) +
               b(j,i)
   enddo
enddo
```

```
do i = 1 to n
   t1 = a(i);
   do j = 2 to n
      x(j,i) = t1 +
               x(j-1,i) +
               b(j,i)
   enddo
enddo
```

Scalar Replacement

Informal Definition: Replace an array reference with a scalar temporary. (Use dependences to locate consistent re-use patterns)

```
do i = 1, n
   do j = 2, n
      x(j,i) = a(i) +
               x(j-1,i) +
               b(j,i)
   enddo
enddo
```

```
do i = 1, n
   t1 = a(i);
   t2 = x(1, i)
   do j = 2, n
      x(j,i) = t1 +
               x(j-1,i) +
               b(j,i)
      t2 = x(j,i)
   enddo
enddo
```

Scalar Replacement

Informal Definition: Replace an array reference with a scalar temporary. (Use dependences to locate consistent re-use patterns)

```
do i = 1, n
   do j = 2, n
      x(j,i) = a(i) +
               x(j-1,i) +
               b(j,i)
   enddo
enddo
```

```
do i = 1, n
   t1 = a(i);
   t2 = x(1, i)
   do j = 2, n
      x(j,i) = t1 +
               t2 +
               b(j,i)
      t2 = x(j,i)
   enddo
enddo
```

Unroll and Jam

Informal Definition: Unroll the outer loop by k, then fuse the resulting k inner loops into a single loop

```
do i = 1 to n
   do j = 1 to n
      a(i) = a(i) + b(j)
   enddo
enddo
```

```
do i = 1 to n step 2
   do j = 1 to n
      a(i) = a(i) + b(j)
      a(i+1) = a(i+1) + b(j)
   enddo
enddo
```

Unroll and Jam Example

```
do i = 1 to n step 2
   do j = 1 to n
      a(i) = a(i) + b(j)
      a(i+1) = a(i+1) + b(j)
   enddo
enddo
```

```
do i = 1 to n step 2
   t0 = a(i+0)
   t1 = a(i+1)
   do j = 1 to n
      t0 = t0 + b(j)
      t1 = t1 + b(j)
   enddo
   a(i+0) = t0
   a(i+1) = t1
enddo
```

Unroll and Jam Example

```
do i = 1 to n step 2
   do j = 1 to n
      a(i) = a(i) + b(j)
      a(i+1) = a(i+1) + b(j)
   enddo
enddo
```

```
do i = 1 to n step 2
   t0 = a(i+0)
   t1 = a(i+1)
   do j = 1 to n
      t3 = b(j)
      t0 = t0 + t3
      t1 = t1 + t3
   enddo
   a(i+0) = t0
   a(i+1) = t1
enddo
```

Loop Skewing

Informal Definition: Increase dependence distance by n by substituting loop index j with jj = j + n $*$ i.

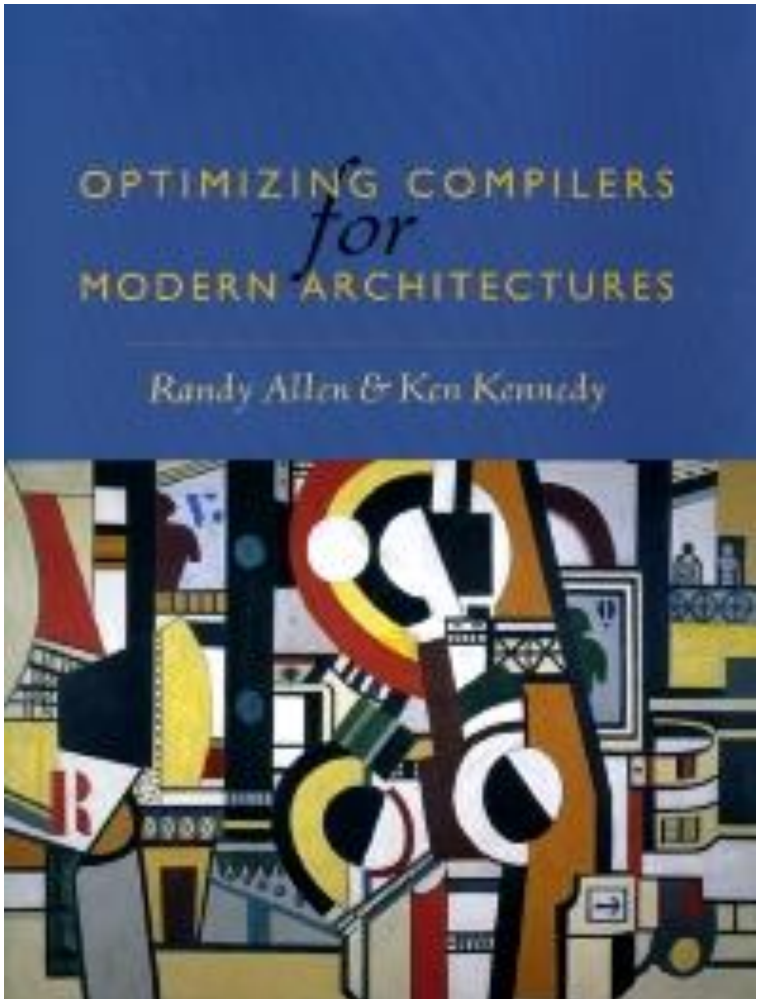
E.g., For n = 1, we use jj = j + 1

```
do i=2,N
  do j=2,N
    A[i,j] = A[i-1,j]
           + A[i,j-1]
  enddo
enddo
```

```
do i=2,N
  do jj=i+2,i+N
    A[i,jj-i] = A[i-1,jj-i]
              + A[i,jj-i-1]
  enddo
enddo
```

Uses of Loop Skewing

- Improve parallelism by converting '=' to '+' in a direction vector
- Improve vectorization in a similar way
- (Rarely) Could be used to *simplify* index expressions



More details:

Optimizing Compilers for Modern Architectures

Allen and Kennedy

Academic Press

Polyhedral Compilation

Brief Introduction to Polyhedral Compilation Techniques:

Basic polyhedral concepts in program analysis

Iteration spaces; array references

Dependence analysis

Loop transformations: representation

Loop transformations: code generation

Polyhedra

k-tuple: A point in Z^k, e.g., (1,−4, 3) or J = (i1, i2, . . . , ik)

Tuple set: A set of tuple points (0,1,2), (2,3) …

Tagged tuple set: A set of tuple points A(1,2), C(3)

- Can be represented as a tuple, where e.g., map(A)=0, map(C)=2

Polyhedron: A tuple set defined by affine inequalities

General: $\{(i_1, i_2, \ldots, i_k) : A \cdot \vec{\imath} \leq \vec{U}\}$

e.g. $\{(i_1, i_2) : L_1 \leq i_1 < U_1 \wedge L_2 \leq i_2 < U_2\}$

- Focus on convex polyhedral
- Integer polyhedron: all in/out points are integers
- Integer hull: set of integer points that bounds rational polyhedron

Tuple Relations

Tuple relation (or *relation* or *mapping*.) A mapping from tuple sets to tuple sets, e.g.,

$\{(i,j) \rightarrow (ii,jj) : 0 \leq i < N \wedge 0 \leq j < N \wedge ii = i \wedge jj = i + j - 1\}$

A relation, R, "applied" to a tuple set, S, yields a new tuple set, R(S).

E.g., $S = \{(i) : 0 \leq i \leq N\}, R = \{(i) \rightarrow (ii) : 0 \leq i \leq N \wedge ii = 2i + 1\}$,

results in $R(S) = \{(ii) : \exists k : ii = 2k + 1 \wedge 1 \leq ii \leq 2N + 1\}$.

Analysis Steps

1. Extract model from the code
    - Affine iteration spaces as Polyhedra
    - Array references as polyhedral mappings
2. Dependence analysis:
    - Turn into polyhedral satisfaction problem
3. Transformations:
    - Permutations/transformations on the model, specified by tuple relations
    - Generate code from the model (original code and the transformed iteration spaces)

Affine Iteration Spaces as Polyhedra

```
do i1 = L1 to U1
  S1
  do i2 = L2 to U2
    S2
    . . .
      do ik = Lk to Uk
        Sk
      enddo
    . . .
  enddo
enddo
```

Every statement in the program has an associated iteration space, describing the enclosing loops:

$$L = \{(i_1, i_2, \ldots i_k) : L_1 \leq i_1 < U_1 \\ \wedge \; L_2 \leq i_2 < U_2 \\ \wedge \; L_k \leq i_k < U_k\}$$

- For polyhedral analysis, L_i, U_i must be affine functions of index variables (i), loop-invariant program variables and constants.

Array References as Polyhedral Mappings

```
do i1 = L1 to U1
  S1
  do i2 = L2 to U2
    S2
    . . .
      do ik = Lk to Uk
        A[i1,…,ik] = …
      enddo
     . . .
  enddo
enddo
```

Every array reference in the program is a mapping from the iteration space (of the statement) to array elements. E.g.,

$$
\begin{aligned}
L \rightarrow A : \{(\vec{\imath}, \vec{a}) : \vec{\imath} \in L \\
\wedge\ a_1 = f_1(\vec{\imath}) \ldots \\
\wedge\ a_r = f_r(\vec{\imath})\}
\end{aligned}
$$

- For polyhedral analysis, f_i, must be affine functions of index variables (i), loop-invariant program variables and constants.

Checking for Data Dependence

There is a data dependence between

$A(f_1(\vec{\imath}), f_2(\vec{\imath}), \ldots, f_r(\vec{\imath}))$ and $A(g_1(\vec{\imath}), g_2(\vec{\imath}), \ldots, g_r(\vec{\imath}))$

iff the following polyhedron contains integer points:

$$\{(i_1, i_2, \ldots, i_r, j_1, j_2, \ldots, j_r) : \vec{\imath} \in L \wedge \vec{\jmath} \in L \wedge$$
$$f1(\vec{\imath}) = g1(\vec{\jmath}) \wedge \ldots \wedge f_r(\vec{\imath}) = g_r(\vec{\jmath})\}$$

Program Transformations

Program transformations as polyhedral mappings: Many program transformations can be represented as a mapping (for each original program statement) from its iteration space in the original program to its iteration space in the transformed program.

Loop reordering transformations:
a transformation on a perfect loop nest that reorders the loop iteration space but does not modify the relative order of statements within the innermost loop (sometimes called an atomic block).

$$L \rightarrow L : \{(\vec{\imath}) \rightarrow (\overrightarrow{ii}) : \vec{\imath} \in L$$
$$\wedge\, ii_1 = \varphi_1(\vec{\imath}) \wedge \ldots \wedge ii_k = \varphi_k(\vec{\imath})\}$$

Loop Transformations and Matrices

Alternate representation for loop transformations – as a matrix:
$\Phi(\vec{i}) = T \cdot \vec{i} + \vec{t}$

- The transformation is affine iff T is a constant matrix and $\vec{t}$ is a parametric vector consisting of loop-invariant program variables and constants.
- Each column in the matrix product represents a single input loop. Each row in the matrix product represents a single output loop.
- The transformation is called *unimodular* if T is unimodular (i.e., square integer matrix with determinant +1 or -1)

Loop Transformations and Matrices

A transformation is called *unimodular* if the matrix T is unimodular (i.e., square integer matrix with determinant +1 or -1)

Loop interchange: $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \vec{t} = \vec{0}$

Loop reversal: $T = [-1], \vec{t} = (U_1 - 1)$

Example Transformations

Loop reversal: $\Phi = \{(i) \rightarrow (ii) : L_1 \leq i \leq U_1 \wedge ii = U_1 - i + 1\}$

```
do i = L_1 to U_1
  A(i)= B(i) + C(i)
enddo
```

$\Longrightarrow$

```
do ii = U_1 to L_1 by -1
  A(ii) = B(ii) + C(ii)
enddo
```

Example Transformations

Loop reversal: $\Phi = \{(i) \to (ii) : L_1 \leq i \leq U_1 \wedge ii = U_1 - i + 1\}$

```
do i = L_1 to U_1                    do ii = U_1 to L_1 by -1
  A(i)= B(i) + C(i)        ==>         A(ii) = B(ii) + C(ii)
enddo                                enddo
```

Loop interchange: $$\Phi = \{(i,j) \to (jj,ii) : L_1 \leq i \leq U_1 \wedge L_2 \leq j \leq U_2 \\ \wedge ii = i \wedge jj = j\}$$

```
do i = L_1 to U_1                    do jj = L_2 to U_2
  do j = L_2 to U_2                    do ii = L_1 to U_1
    A(i,j) = B(i+j,i-j) + 1  ==>         A(ii,jj) = B(ii+jj,ii-jj) + 1
  enddo                                enddo
enddo                                enddo
```

Example Transformations

Loop tiling: Tile sizes = (s_1, s_2)

$$\begin{aligned}\Phi \;=\; \{(i,j) \rightarrow (ti, tj, ii, jj) : L_1 \leq i \leq U_1 \wedge L_2 \leq j \leq U_2 \\ \wedge\, ti = s_1 \times \lfloor \tfrac{i-L_1}{s_1} \rfloor \;\wedge\, tj = s_2 \times \lfloor \tfrac{j-L_2}{s_2} \rfloor \\ \wedge\, ii = i \;\wedge\; ti \leq ii \leq min(ti + s_1 - 1, U_1) \\ \wedge\, jj = j \;\wedge\; tj \leq jj \leq min(tj + s_2 - 1, U_2)\}\end{aligned}$$

```
do i = L_1 to U_1
  do j = L_2 to U_2
    C[i,j] += A[i,k] * B[k,j]
  enddo
enddo
```

$\Longrightarrow$

```
do ti = L_1 to U_1 by s_1
  do tj = L_2 to U_2 by s_2
    do ii = ti to min(ti+s_1-1, U_1)
      do jj = tj to min(tj+s_2-1, U_2)
        C[ii,jj] += ...
enddo enddo enddo endddo
```

Imperfect Loop Nests

General approach: Add an extra ("sequencing") dimension in the iteration space to enforce ordering on individual statements:

```
do i = L_1 to U_1
    S1(i)
    do j = L_2 to U_2
        S2(i,j)
    enddo
    S3(i)
enddo
```

$L(S1) = \{(i, \mathbf{0}, j)\colon L1 \leq i \leq U1 \wedge j = L2\}$

$L(S2) = \{(i, \mathbf{1}, j)\colon L1 \leq i \leq U1 \wedge L2 \leq j \leq U2\}$

$L(S3) = \{(i, \mathbf{2}, j)\colon L1 \leq i \leq U1 \wedge j = U2\}$

Pros and Cons

Pros:

- Principled representation
- Fine-grained optimization and analysis using mathematical programming
- Simplify loop transformations

Cons:

- In general, NP-complete problem:
  boils down to Integer programming
- Memory consuming, especially for irregular nests with control flow

References

Courses/Lectures:

- Louis-Noël Pouchet course: http://web.cse.ohio-state.edu/~pouchet/#lectures
- Pollylabs video and written tutorials: http://www.pollylabs.org/education.html

Tools: GCC Graphite, URUK, Omega, Loop…

Polly (LLVM):

- Tool: http://polly.llvm.org
- Interactive playground: http://playground.pollylabs.org/