CS 293S Optimizing for Parallelism and Locality: Affine Transformation

Yufei Ding

Reference Book:

"Optimizing Compilers for Modern Architecture" by Allen & Kennedy

Slides adapted from Louis-Noël Pouche, Mary Hall

Review of last this lecture

- Data Dependence
  - True, Anti-, Output dependence
  - Source and Sink
  - Distance vector, direction vector
  - Relation between Reordering transformation and Direction vector

Review

- Loop dependence
  - loop-carried dependence
  - Loop-Independent Dependences
- Dependence graph
- Dependence Tests
  - Greatest common divisor (GCD)
- Controlling execution order
  - determining the upper/lower bound through projection by Fourier-Motzkin elimination
  - General algorithms to determine loop bounds
    - inner to outer levels to generate
    - outer to inner levels to refine

Loop-Carried and Loop-Independent Dependences

- If in a loop statement S_2 depends on S_1, then there are two possible ways of this dependence occurring:

- Source and sink happen on different iterations
  - This is called a loop-carried dependence.

- S1 and S2 execute on the same iteration
  - This is called a loop-independent dependence

Loop-Carried Dependence

Example:

```
DO I = 1, N
S1    A(I+1) = F(I)
S2    F(I+1) = A(I)
ENDDO
```

Loop-Carried Dependence

- Dependence Level:

  Level of a loop-carried dependence is the index of the leftmost non-"=" of D(i,j) for the dependence.

For instance:

```
DO I = 1, 10
  DO J = 1, 10
    DO K = 1, 10
S1      A(I, J, K+1) = A(I, J, K)
    ENDDO
  ENDDO
ENDDO
```

- Direction vector for S1 is (=, =, <)
- Level of the dependence is 3
- A level-k true dependence between S_1 and S_2 is denoted by $S_1\ d_k\ S_2$

The iterations of a loop can be executed in parallel if the loop carries no dependences

Loop-Independent Dependences

Example:

```
DO I = 1, 10
S1     A(I) = ...
S2     ... = A(I)
ENDDO
```

More complicated example:

```
DO I = 1, 9
S1      A(I) = ...
S2     ...  = A(10-I)
ENDDO
```

Loop-Independent Dependences

- Theorem 2.5. If there is a loop-independent dependence from S_1 to S_2, any reordering transformation that does not move statement instances between iterations and preserves the relative order of S_1 and S_2 in the loop body preserves that dependence.

- S_2 depends on S_1 with a loop independent true dependence is denoted by $S_1 \; d_\infty \; S_2$
- The direction vector has entries that are all "=" for loop independent dependences

Is the reordering legal?

```
DO I = 1,  100
    DO J=1,  100
            A(I+1, J) = A(I, 5) + B
    ENDDO
ENDDO
```

(<, <)
(<, =)
(<, >)

```
DO J = 1,  100
    DO I=1,  100
            A(I+1, J) = A(I, 5) + B
    ENDDO
ENDDO
```

(<, <)
(=, <)
(>, <)

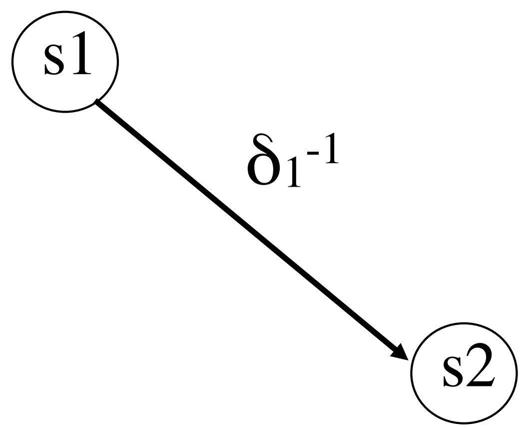
Dependence Graph

Important point: order of vectors depends on order of loops, not use in arrays

```
     DO I = 1, 100
S1      D(I) = A (5, I)
        DO J=1, 100
S2         A(J, I-1) = B(I) + C
        ENDDO
     ENDDO
```

from S1 to S2: (<)
level-1 antidependence
S1 is the source, S2 is the sink

$$S2 \quad \delta_1^{-1} \quad S1$$



Only consider common loops!

- Nodes for statements
- Edges for data dependences
  - Labels on edges for dependence levels and types

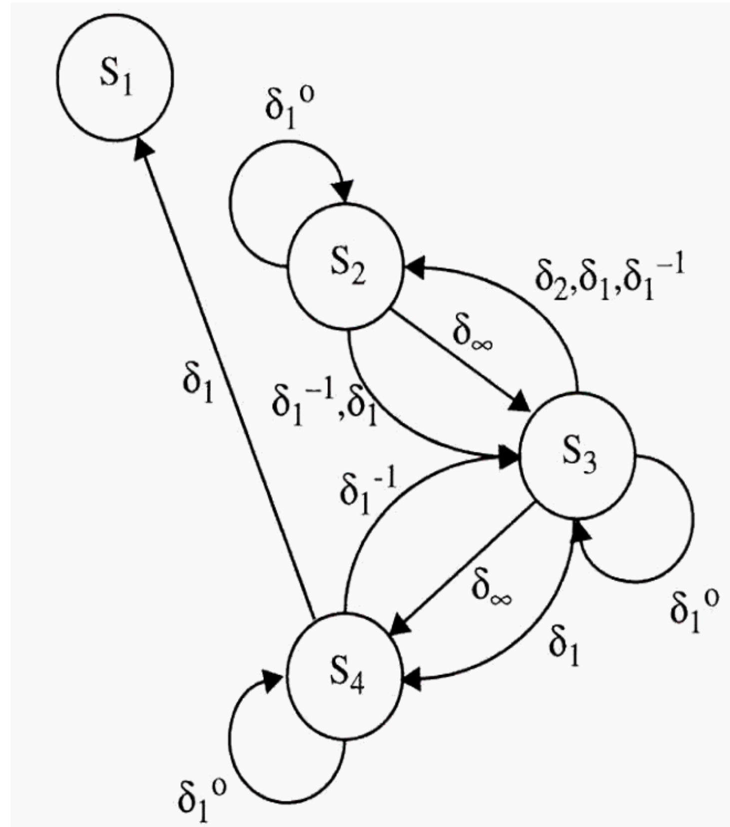
```
     DO I = 1, 100
S1       D(I) = A (102, I)
         DO J=1, 100
S2             A(J, I-1) = B(I) + C
       ENDDO
     ENDDO
```

no dependence

Dependence Graph

```
DO I = 1, 100
S1    X(I) = Y(I) + 10
      DO J = 1, 100
S2      B(J) = A(J,N)
        DO K = 1, 100
S3        A(J+1,K)=B(J)+C(J,K)
        ENDDO
S4      Y(I+J) = A(J+1, N)
      ENDDO
ENDDO
```

```
DO I = 1, 100
S1    X(I) = Y(I) + 10
        DO J = 1, 100
S2          B(J) = A(J,N)
            DO K = 1, 100
S3              A(J+1,K)=B(J)+C(J,K)
            ENDDO
S4          Y(I+J) = A(J+1, N)
        ENDDO
ENDDO
```



1. True dependences denoted by $S_i\ d\ S_j$
2. Antidependence denoted by $S_i\ d^{-1}\ S_j$
3. Output dependence denoted by $S_i\ d^0\ S_j$

d and δ are used interchangeably

Dependence Graph

Important point: order of vectors depends on order of loops, not use in arrays

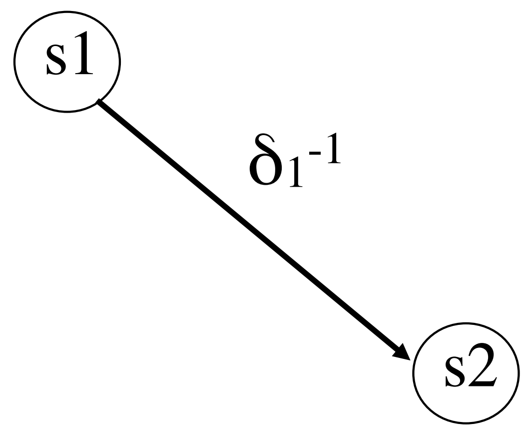
```
    DO I = 1, 100
S1      D(I) = A (5, I)
        DO J=1, 100
S2          A(J, I-1) = B(I) + C
        ENDDO
    ENDDO
```

from S1 to S2: (<)

level-1 antidependence

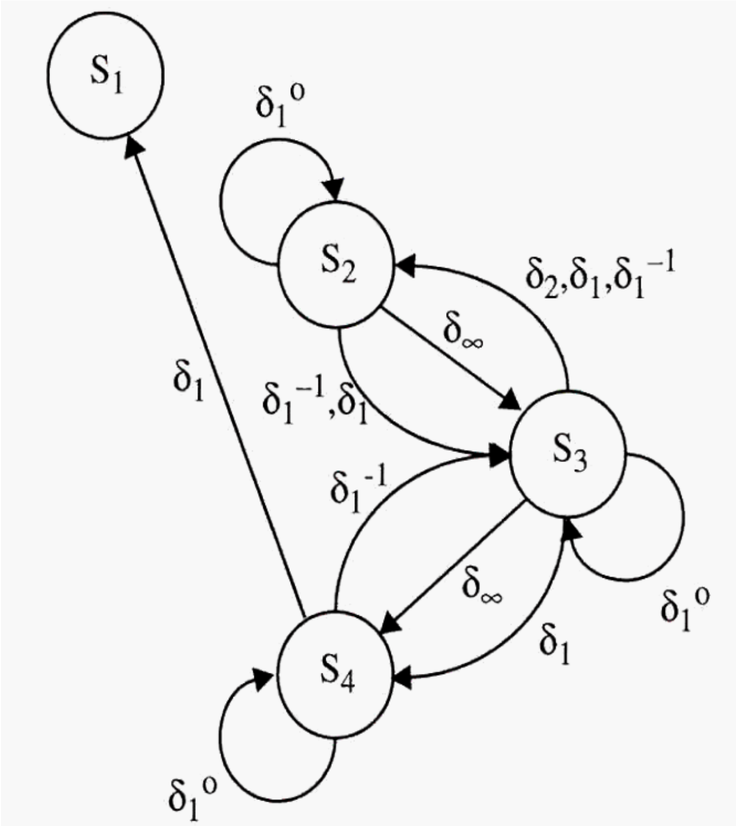
S1 is the source, S2 is the sink

S2 δ_1^{-1} S1



- Nodes for statements
- Edges for data dependences
  - Labels on edges for dependence levels and types

```
DO I = 1, 100
S1    X(I) = Y(I) + 10
        DO J = 1, 100
S2          B(J) = A(J,N)
            DO K = 1, 100
S3              A(J+1,K)=B(J)+C(J,K)
            ENDDO
S4          Y(I+J) = A(J+1, N)
        ENDDO
ENDDO
```



1. True dependences denoted by S_i d S_j
2. Antidependence denoted by S_i d^{-1} S_j
3. Output dependence denoted by S_i d^0 S_j

d and δ are used interchangeably

Data Dependence Tests

- Given the loop nest:

```
for (i = 0; i < N; i++)
    a[f(i)] = ...
        ... = a[g(i)]
```

- A dependence exists if there exist an integer i and an i' such that: f(i) = g(i')
  - 0 <= i, i' < N
  - If i < i', write happens before read (true dependence)
  - If i > i', write happens after read (anti dependence)

Solution: GCD test

- Does f(i) = g(i') have a solution?
  - assume f(i) = a*i + b g(i) = c*i + d
  - f(i) = g(i') ⇒ ai + b = ci' + d ⇒ a1*i + a2*i' = a3

- An equation a1*i + a2*i' = a3 has a solution iff gcd(a1, a2) evenly divides a3

Examples

```
for (i = 1; i < 10; i++) {
     Z[2*i] = . . .;
}
for (j = 1; j < 10; j++){
     Z[2*j+1] = . . .;
}
```

- 2i = 2j + 1
- gcd(2, -2) = 2, and 2 does not divide 1 evenly. Thus, there is no solution.

Other Examples:

15*i + 6*j - 9*k = 12 has a solution (gcd = 3)
2*i + 7*j = 3 has a solution (gcd = 1)
9*i + 6*j = 10 has no solution (gcd = 3)

Finding the GCD

- Finding GCD with Euclid's algorithm
- Repeat (suppose a>b)
  - a = a mod b
  - swap a and b
  - until b is 0 (resulting a is the gcd)
- Why? If g divides a and b, then g divides a mod b

gcd(27, 15):
Iter1: a = 27, b = 15
a = 27 mod 15 = 12
Iter2: a = 15, b = 12
a = 15 mod 12 = 3
Iter3: a = 12, b = 3
a = 12 mod 3 = 0
Iter4: a = 3, b = 0
gcd = 3

Downsides to GCD test

- If f(i) = g(i') fails the GCD test, then there is no i, i' that can produce a dependence → loop has no dependences
- If f(i) = g(i'), there might be a dependence, but might not
  - i and i' that satisfy equation might fall outside bounds
  - Loop may be parallelizable, but cannot tell
- Unfortunately, most loops have gcd(a, b) = 1, which divides everything
- Other optimizations (loop interchange) can tolerate dependences in certain situations

```
for (i = 1; i < 10; i++)
        Z[i] = Z[i+10];
```

Other dependence tests

- GCD test: doesn't account for loop bounds, does not provide useful information in many cases
- Banerjee test (Utpal Banerjee): more accurate test, takes directions and loop bounds into account
- Omega test (William Pugh): even more accurate test, precise but can be very slow
- Range test (Blume and Eigenmann): works for non-linear subscripts

- Compilers tend to perform simple tests and only perform more complex tests if they cannot prove non-existence of dependence

Code generation by loop transformation

```
for (i=0; i<=5; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;
```

→

```
for (j=0; j<=7; j++)
    for (i=0; i<=min(5, j); i++)
        Z[j, i] = 0;
```

- The problem of how we choose an ordering that honors the data dependences and optimizes for data locality and parallelism is generally hard.

- Here we assume that a legal and desirable ordering is given, and show how to generate code that enforce the ordering.

Code generation by loop transformation

- Analysis:
  - Rectangular: all loop bounds are constants → Easy
  - More complicated, but still quite realistic: the upper and/or lower bounds on one loop index can depend on the values of the indexes of the outer loops. → ??

- Goal:
  - outermost loop bounds: constants
  - inner loop bounds: linear combinations of outer loop index variables and constants.

Example

```
for (i=0; i<=5; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;
```

```
i>=0;
i<=5;
j>=i;
j<=7;
```

Loop constraints

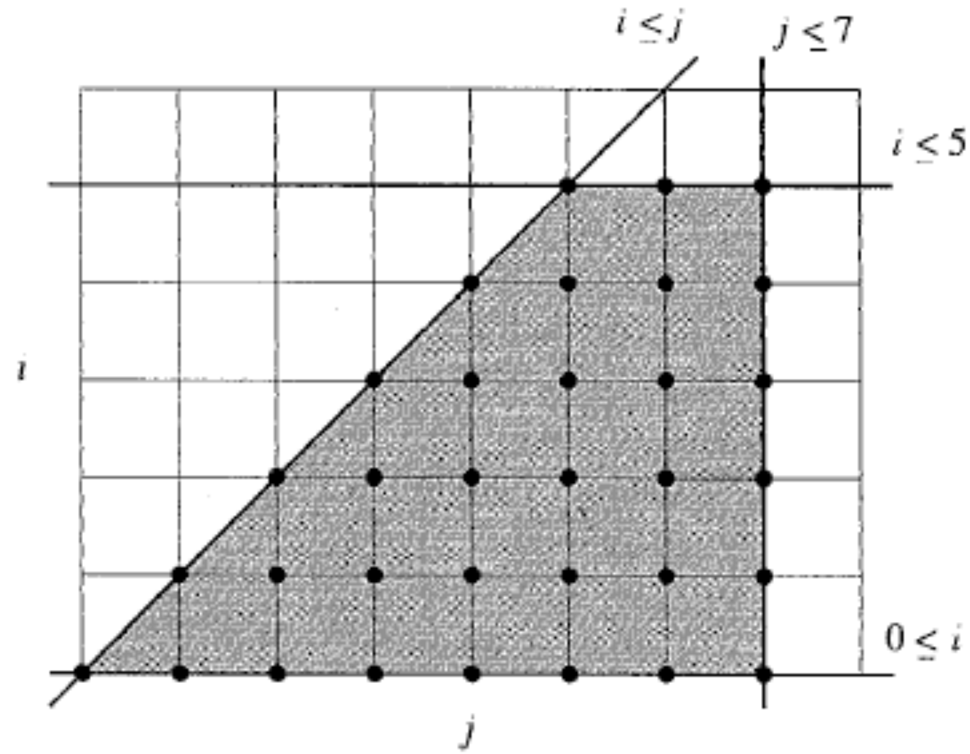
→

```
for (j=?; j<=?; j++)
    for (i= 0; i<= min(5,j); i++)
        Z[j, i] = 0;
```

To get the bounds for index j, we need to eliminate i from the loop constraints.

Fourier-Motzkin elimination

- Input: a polyhedron S defined by a set of linear constraints on $x_1, x_2, \ldots, x_n$. A given variable x_m that is to be eliminated.
- Output: a polyhedron S' defined by linear constraints on $x_1, x_2, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n$ that is a projection of S onto dimensions other than the x_m



Iteration space

```
for (i=0; i<=5; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;
```

Fourier-Motzkin Elimination

Algorithm:

- For every pair of a lower bound and an upper bound on x_m, such as $L <= c_1 x_m$ & $c_2 x_m <= U$, create a new constraint $c_2 L <= c_1 U$.

- S' is the set including all new constrains and those in S that do not contain x_m.

- It is possible that S' is an empty space.

Example

```
for (i=0; i<=5; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;
```

```
i>=0;
i<=5;
j>=i;
j<=7;
```

```
i>=0;
i<=min(5,j);
```

```
j>=0;
j<=7;
```

```
for (j=0; j<=7; j++)
    for (i= 0; i<= min(5,j); i++)
        Z[j, i] = 0;
```

To Eliminate i.

- one lower bound: 0 <= i
- two upper bounds: i <= j and i <= 5.
- This generates two constraints:
- 0 <= j and 0 <= 5.
  - The latter is trivially true and can be ignored.
  - The former gives the lower bound on j, and the original upper bound j < 7 gives the upper bound.

Loop-Bounds Generation Algorithm

- Compute the loop bounds from the innermost to the outer loops.

```
for (i=0; i<=5; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;
```

```
i>=0;
i<=5;
j>=i;
j<=7;
```

$S_n = S$;
for (i=n; i>=1; i--){
  L_{vi} = all the lower bounds on v_i in S_i;
  U_{vi} = all the upper bounds on v_i in S_i;
  S_{i-1} = Constraints by eliminating v_i from S_i;
}
/* remove redundancies */
$S' = \Phi$;
for (i=1; i<=n; i++){
  Remove any bounds in L_{vi} and U_{vi} implied by S';
  Add the remaining constraints of L_{vi} and U_{vi} on v_i to S';
}

target order: j,i

L_i: 0
U_i: 5,j
L_j: 0
U_j: 7

bounds on i is (0, min(5,j)); bounds on j is (0, 7).

Loop-Bounds Generation

- Compute the loop bounds from the innermost to the outer loops.

```
for (i=0; i<=8; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;
```

```
i>=0;
i<=8;
j>=i;
j<=7;
```

S_n = S;
for (i=n; i>=1; i--){
  L_{v_i} = all the lower bounds on v_i in S_i;
  U_{v_i} = all the upper bounds on v_i in S_i;
  S_{i-1} = Constraints by eliminating v_i from S_i;
}
/* remove redundancies */
S'=Φ;
for (i=1; i<=n; i++){
  Remove any bounds in L_{v_i} and U_{v_i} implied by S';
  Add the remaining constraints of L_{v_i} and U_{v_i} on v_i to S';
}

target order: j,i

L_i: 0
U_i: 8,j
L_j: 0
U_j: 7

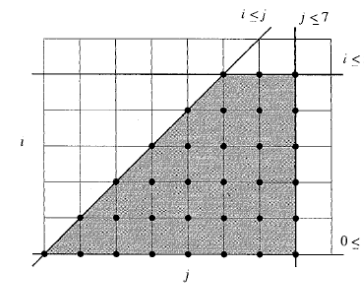
bounds on i is (0, j);
bounds on j is (0, 7).

```
for (i=0; i<=5; i++)
    for (j=i; j<=7; j++)
        Z[j, i] = 0;
```

```
i>=0;
i<=5;
j>=i;
j<=7;
```

Target: sweep through diagonally.

[0,0], [1,1], [2,2], [3,3], [4,4], [5,5]
[0,1], [1,2], [2,3], [3,4], [4,5]
[0,2], [1,3], [2,4], [3,5]
...
[0,6], [1,7]
[0,7]



k=j-i, order: k, j.

j-k>=0;
j-k<=5;
j>=j-k;
j<=7.

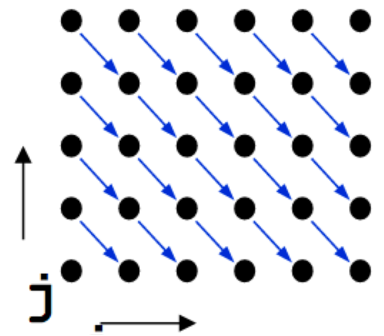
L_j: k
U_j: 5+k, 7
L_k: 0
U_k: 7

```
for (k=0; k<=7; k++)
    for (j=k; j<=min(5+k,7); j++)
        Z[j, j-k] =0;
```

Loop Skewing and Permutation

- Original Code:

```
for (i=0; i<=6; i++)
    for (j=0; j<=5; j++)
        A(i,j) = A(i-1,j+1)+1
```



- Distance vector: (1, -1)
- Goal to find a new set of (i', j'):

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i' \\ j' \end{bmatrix}$$

- New distance vector: (0,1)

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