Course Outline

• Technology Trends => Multi-core Ubiquity
• Parallel Programming Models
• Introduction to Compiler Optimizations
  – Introduction
  – Dependences, Transformations
  – Polyhedral Models
• Optimizations for General-Purpose Multicore Architectures
• Optimizations for GPUs
Compilers (1)

- Compilers: translate the abstract operational semantics of a program into a form that makes effective use of a highly complex machine architecture.
- Different architectural features exist and sometimes interact in complex ways.
- There is often a trade-off between exploiting parallelism and exploiting locality to reduce yet another widening gap: the memory wall.
- For the compiler: this means combining multiple program transformations (polyhedral models are useful here).
- Access latency and bandwidth of the memory subsystems have always been a bottleneck. Get worse with multi-core.
Compilers (2)

- Program optimization is over huge and unstructured search spaces: this combinational task is poorly achieved in general, resulting in weak scalability and disappointing sustained performance.

- Even when programming models are explicitly parallel (data-parallelism, threads, etc.,) advanced compiler technology is needed
  - to relieve the programmer from scheduling and mapping the application to computational cores,
  - for understanding the memory model and communication details.
Data Dependences

- Essential constraints:

\[
\begin{align*}
S1: \quad & a = b + c \\
S2: \quad & d = a \times 2 \\
S3: \quad & a = c + 2 \\
S4: \quad & e = d + c + 2
\end{align*}
\]
Data Dependences

- Essential constraints:

\[
\begin{align*}
S1: & \quad a = b + c \\
S2: & \quad d = a \times 2 \\
S3: & \quad a = c + 2 \\
S4: & \quad e = d + c + 2
\end{align*}
\]

- S1 and S2 cannot execute concurrently
Data Dependences

• Essential constraints:

\[
\begin{align*}
S1: & \quad a = b + c \\
S2: & \quad d = a \times 2 \\
S3: & \quad a = c + 2 \\
S4: & \quad e = d + c + 2
\end{align*}
\]

• S2 and S3 cannot execute concurrently
Data Dependences

• Essential constraints:

S1:  \( a = b + c \)
S2:  \( d = a \times 2 \)
S3:  \( a = c + 2 \)
S4:  \( e = d + c + 2 \)

• S1 and S3 cannot execute concurrently
Data Dependences

- Essential constraints:

\[
\begin{align*}
S1: & \quad a = b + c \\
S2: & \quad d = a \cdot 2 \\
S3: & \quad a = c + 2 \\
S4: & \quad e = d + c + 2
\end{align*}
\]

- But S3 and S4 can execute concurrently
Data Dependences

• Essential constraints:

\[
\begin{align*}
S1: & \quad a = b + c \\
S2: & \quad d = a \times 2 \\
S3: & \quad a = c + 2 \\
S4: & \quad e = d + c + 2
\end{align*}
\]

• S1 and S2 cannot execute concurrently
• S2 and S3 cannot execute concurrently
• S1 and S3 cannot execute concurrently
• But S3 and S4 can execute concurrently

• Execution conditions due to Bernstein (1966)
Types of Dependences

- **Flow-dependence** occurs when a variable which is assigned a value in one statement say S1 is read in another statement, say S2 later.

\[
\begin{align*}
\text{S1:} & \quad a = b + c \\
\text{S2:} & \quad d = a \times 3
\end{align*}
\]
Types of Dependences

- **Anti-dependence** occurs when a variable which is read in one statement say S1 is assigned a value in another statement, say S2, later.

```
S1:   d = a * 3
S2:   a = b + c
```
Types of Dependences

- **Output-dependence** occurs when a variable which is assigned a value in one statement say S1 is later reassigned in another statement, say S2.

\[
\begin{align*}
S1: & \quad a = b + c \\
S2: & \quad a = d \times 3
\end{align*}
\]
Types of Dependences

• **Input-dependence** occurs when a variable is read in two different statements say S1 and S2. **Relative ordering** of S1 and S2 is **not important for input dependence**.

\[
\begin{align*}
S1: & \quad a = b + c \\
S2: & \quad d = b \times 3
\end{align*}
\]
Data Dependences in Loops

• Associate a dynamic instance to each statement. For example

\[
\begin{align*}
\text{DO } i & = 1 \text{ to } 50 \\
S1: & \quad A(i) = B(i-1) + C(i) \\
S2: & \quad B(i) = A(i+2) + C(i)
\end{align*}
\]

• Statements S1 and S2 are executed 50 times. We say S2(10) to mean the execution of S2 when i = 10.

• Dependences are based on dynamic instances of statements.
Data Dependences in Loops

• Unrolling loops can help one figure out dependences:

\[
\begin{align*}
S1(1) : & \quad A(1) = B(0) + C(1) \\
S2(1) : & \quad B(1) = A(3) + C(1) \\
S1(2) : & \quad A(2) = B(1) + C(2) \\
S2(2) : & \quad B(2) = A(4) + C(2) \\
S1(3) : & \quad A(3) = B(2) + C(3) \\
S2(3) : & \quad B(3) = A(5) + C(3) \\
& \ldots \ldots \ldots \ldots \ldots \\
S1(50) : & \quad A(50) = B(49) + C(50) \\
S2(50) : & \quad B(50) = A(52) + C(50)
\end{align*}
\]
Iteration Spaces

- Nested loops define an iteration space:

\[
\begin{align*}
& \text{DO } i = 1 \text{ to } 4 \\
& \quad \text{DO } j = 1 \text{ to } 4 \\
& \quad \quad A(i,j) = B(i,j) + C(j) \\
& \quad \text{ENDDO} \\
& \text{ENDDO}
\end{align*}
\]

- Sequential execution (traversal order):

- Dimensionality of iteration space = loop nest level; arbitrary convex shapes are allowed
Dependences in Loop Nests

$L_j$ and $U_j$ are affine functions of enclosing loop indices $I_1, \cdots, I_{j-1}$

There is a dependence in a loop nest if there are iterations $\vec{i} = (i_1, i_2, \cdots, i_n)$ and $\vec{j} = (j_1, j_2, \cdots, j_n)$ and some memory location $M$ such that:

- $\vec{i}$ and $\vec{j}$ are valid iteration instances
- $\vec{i} \prec \vec{j}$, i.e., $\vec{i}$ executes before $\vec{j}$ or $\vec{i}$ precedes $\vec{j}$
- $S(\vec{i})$ and $S(\vec{j})$ reference $M$

```
DO I1 = L1, U1
  DO I2 = L2, U2
  ... 
  DO In = Ln, Un
  S(I1,\ldots,In)
```
Dependence Level

We say that \( \vec{I} \prec_u \vec{J} \) (for \( u = 1, \ldots, n \)) if:
- \( I_k = J_k \) for \( k = 1, \ldots, u - 1 \) and,
- \( I_u < J_u \)

\( u \) is defined as the level of the dependence.

If \( I_k = J_k \) for all values of \( k \), then we say that the dependence is loop independent.

If \( \vec{I} \prec_u \vec{J} \), then the dependence is at level \( u \); we also refer to this as a dependence carried by the loop \( i_u \).

Dependence level is well-defined for perfectly nested loops but not for general nesting
Dependence Distance

• Suppose there is a dependence from $S(i_1, i_2, \cdots, i_n)$ to $S(j_1, j_2, \cdots, j_n)$ in a loop nest.

• We define the dependence *distance vector* as

  $$(j_1 - i_1, j_2 - i_2, \cdots, j_n - i_n)$$

• Legality of transformations (in non-polyhedral frameworks) defined in terms of distance vectors or approximations to distance vectors, such as direction vectors and dependence levels.
**Dependence Direction**

- We define sign of a real number $x$ as follows:
  - $\text{sign}(x) = 0$ if $x=0$
  - $\text{sign}(x) = +$ if $x > 0$
  - $\text{sign}(x) = -$ if $x < 0$

- Suppose we have a dependence from $S(i_1, \ldots, i_n)$ to $S(j_1, \ldots, j_n)$. The dependence distance is $(j_1-i_1, \ldots, j_n-i_n)$

- The **direction vector** for this dependence is:
  \[( \text{sign}(j_1-i_1), \text{sign}(j_2-i_2), \ldots, \text{sign}(j_n-i_n) )\]

For loop nests that have loops that count up, the first non-zero component of any dependence vector must be positive
Example of Dependences

do i = 1 to 5
   do j = i to 5
      S: \[ A(i,j) = A(i,j-3) + A(i-2,j) + A(i-1,j+2) + A(i+1,j-1) \]
   enddo
enddo

<table>
<thead>
<tr>
<th>RHS reference</th>
<th>Type</th>
<th>Distance Vector</th>
<th>Direction Vector</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i,j-3)</td>
<td>Flow</td>
<td>(0,3)</td>
<td>(0,+)</td>
<td>2</td>
</tr>
<tr>
<td>A(i-2,j)</td>
<td>Flow</td>
<td>(2,0)</td>
<td>(+,0)</td>
<td>1</td>
</tr>
<tr>
<td>A(i-1,j+2)</td>
<td>Flow</td>
<td>(1,-2)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
<tr>
<td>A(i+1,j-1)</td>
<td>Anti</td>
<td>(1,-1)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
</tbody>
</table>
Example of Dependences

\[
do i = 1 \text{ to } 5 \\
\quad \text{do } j = i \text{ to } 5 \\
S: \quad A(i,j) = A(i,j-3) + A(i-2,j) + A(i-1,j+2) + A(i+1,j-1) \\
\quad \text{enddo} \\
\quad \text{enddo}
\]

<table>
<thead>
<tr>
<th>RHS reference</th>
<th>Type</th>
<th>Distance Vector</th>
<th>Direction Vector</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i,j-3)</td>
<td>Flow</td>
<td>(0,3)</td>
<td>(0,+)</td>
<td>2</td>
</tr>
<tr>
<td>A(i-2,j)</td>
<td>Flow</td>
<td>(2,0)</td>
<td>(+,0)</td>
<td>1</td>
</tr>
<tr>
<td>A(i-1,j+2)</td>
<td>Flow</td>
<td>(1,-2)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
<tr>
<td>A(i+1,j-1)</td>
<td>Anti</td>
<td>(1,-1)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
</tbody>
</table>
Example of Dependences

\[ \text{do } i = 1 \text{ to } 5 \]
\[ \text{do } j = i \text{ to } 5 \]
\[ S: \quad A(i,j) = A(i,j-3) + A(i-2,j) + A(i-1,j+2) + A(i+1,j-1) \]
\[ \text{enddo} \]
\[ \text{enddo} \]

<table>
<thead>
<tr>
<th>RHS reference</th>
<th>Type</th>
<th>Distance Vector</th>
<th>Direction Vector</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i,j-3)</td>
<td>Flow</td>
<td>(0,3)</td>
<td>(0,+)</td>
<td>2</td>
</tr>
<tr>
<td>A(i-2,j)</td>
<td>Flow</td>
<td>(2,0)</td>
<td>(+,0)</td>
<td>1</td>
</tr>
<tr>
<td>A(i-1,j+2)</td>
<td>Flow</td>
<td>(1,-2)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
<tr>
<td>A(i+1,j-1)</td>
<td>Anti</td>
<td>(1,-1)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
</tbody>
</table>
Example of Dependences

\[
\text{do } i = 1 \text{ to } 5 \\
\text{do } j = i \text{ to } 5 \\
S: \quad A(i,j) = A(i,j-3) + A(i-2,j) + A(i-1,j+2) + A(i+1,j-1) \\
\text{enddo} \\
\text{enddo}
\]

<table>
<thead>
<tr>
<th>RHS reference</th>
<th>Type</th>
<th>Distance Vector</th>
<th>Direction Vector</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i,j-3)</td>
<td>Flow</td>
<td>(0,3)</td>
<td>(0,+)</td>
<td>2</td>
</tr>
<tr>
<td>A(i-2,j)</td>
<td>Flow</td>
<td>(2,0)</td>
<td>(+,0)</td>
<td>1</td>
</tr>
<tr>
<td>A(i-1,j+2)</td>
<td>Flow</td>
<td>(1,-2)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
<tr>
<td>A(i+1,j-1)</td>
<td>Anti</td>
<td>(1,-1)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
</tbody>
</table>
Example of Dependences

\[
\begin{align*}
\text{do } i &= 1 \text{ to } 5 \\
\text{do } j &= i \text{ to } 5 \\
S: & \quad A(i,j) = A(i,j-3) + A(i-2,j) + A(i-1,j+2) + A(i+1,j-1) \\
\end{align*}
\]


<table>
<thead>
<tr>
<th>RHS reference</th>
<th>Type</th>
<th>Distance Vector</th>
<th>Direction Vector</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i,j-3)</td>
<td>Flow</td>
<td>(0,3)</td>
<td>(0,+)</td>
<td>2</td>
</tr>
<tr>
<td>A(i-2,j)</td>
<td>Flow</td>
<td>(2,0)</td>
<td>(+,0)</td>
<td>1</td>
</tr>
<tr>
<td>A(i-1,j+2)</td>
<td>Flow</td>
<td>(1,-2)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
<tr>
<td>A(i+1,j-1)</td>
<td>Anti</td>
<td>(1,-1)</td>
<td>(+,-)</td>
<td>1</td>
</tr>
</tbody>
</table>
Valid (Legal) Transformations

- Recall: We need to talk about instances of statements when dealing with loops.
- In a program, let there be a dependence from some instance of statement $S_1$ (called source) to some instance of statement $S_2$ (called sink).
- A transformation of a program is said to be valid if in the transformed version, every sink executes after its corresponding source.
Transformations: Loop interchange

- Interchange: Exchanges two loops in a perfectly nested loop, also known as permutation

\[
\begin{align*}
&\text{DO } I = 1, N \\
&\quad \text{DO } J = 1, M \\
&\quad S(I,J)
\end{align*}
\]

- Interchange can improve locality
- It is not always legal; can change the dependence level, direction and distance. \((d_1,d_2)\) becomes \((d_2,d_1)\)
- One of the most useful transformations
Loop Interchange - 2

Suppose array $A$ is stored with a column-major layout:

| A(1,1) | A(2,1) | ... | A(1,2) | A(2,2) | ... | A(1,3) | ... |

This loop has poor locality

```fortran
DO I = 1, N
  DO J = 1, M
    A(I,J) = A(I,J) + 5
  ENDDO
ENDDO
```

This loop has good locality

```fortran
DO J = 1, M
  DO I = 1, N
    A(I,J) = A(I,J) + 5
  ENDDO
ENDDO
```
Transformations: Parallel Loop

• A loop in a loop nest can be executed in parallel if it does not carry a dependence

\[
\text{DO } I = 1, N \\
\text{DO } J = 1, M \\
A(I,J) = A(I-1,J)
\]

• The dependence distance is (1,0); the outer (I) loop carries the dependence.

• The inner (J) loop does not carry any dependences

\[
\text{DO } I = 1, N \\
\text{DOPAR } J = 1, M \\
A(I,J) = A(I-1,J)
\]
Examples of Parallelization

\[
\begin{align*}
\text{DO } & I = 1, 10 \\
& A(I) = A(I) + 5
\end{align*}
\]

Parallel

\[
\begin{align*}
\text{DO } & I = 1, 10 \\
& A(I) = A(I-1) + 5
\end{align*}
\]

Not parallel

\[
\begin{align*}
\text{DO } & I = 1, 10 \\
& A(I) = A(I-10) + 5
\end{align*}
\]

Parallel (Why?)
Transformations: Loop skewing

- Skewing: Makes the bounds of a given loop depend on an outer loop counter; here the inner loop J is skewed with respect to the outer loop by a factor f (f is any integer):

  \[
  \begin{array}{l}
  \text{DO } I = 1, N \\
  \text{DO } J = 1, N \\
  S(I,J)
  \end{array}
  \begin{array}{l}
  \text{DO } I = 1, N \\
  \text{DO } J = 1+f*I, N+f*I \\
  S(I,J-f*I)
  \end{array}
  \]

- By itself, loop skewing is always legal. Dependence level does not change. \((d1,d2)\) becomes \((d1,d2+f*d1)\)

- Usually this enables other transformations such as tiling, creation of outer sequential loops, etc.
Use of Loop Skewing

\[
\begin{align*}
\text{DO } & I = 1, N \\
\text{DO } & J = 1, N \\
\text{A}(I,J) &= w \times ( \text{A}(I,J-1) + \text{A}(I-1,J+1) + \ldots )
\end{align*}
\]

- The loop has dependence vectors (0,1) and (1,-1)
- Both the loops carry dependences
- Suppose, loop interchange is useful. But it is illegal!

Solution:
- Skew the J loop with respect to the I loop by 1;
- This leads to dependence vectors (0,1) and (1,0).
- Now, we can interchange loops
Transformations: Loop reversal

• Loop Reversal: Changes the direction in which a loop traverses its iteration range

```
DO I = 1, N
  DO J = 1, N
    S(I,J)
  DO J = N, 1, -1
    S(I,J)
```

• Here we have reversed the j loop; any parallel loop can be reversed

• Usually this enables other transformations (by itself, it is not very useful)

• Reverse outer loop: (d1,d2) becomes (-d1,d2)

• Reverse inner loop: (d1,d2) becomes (d1,-d2)
Transformations: Loop fusion

• Fusion: Fuses two loops, also known as jamming (useful for locality enhancement). In example below, after fusion, you cannot have dependencies from S2 to S1

\[
\begin{align*}
\text{DO } & I = 1, N \\
\text{S1: } & A(I) = B(I) + C(I) \\
& \text{ENDDO} \\
\text{DO } & I = 1, N \\
\text{S2: } & E(I) = A(I) \times D(I) \\
& \text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } & I = 1, N \\
A(I) & = B(I) + C(I) \\
E(I) & = A(I) \times D(I) \\
& \text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } & I = 1, N \\
& a = B(I) + C(I) \\
E(I) & = a \times D(I) \\
& \text{ENDDO}
\end{align*}
\]
Illegal Loop Fusion Example

\[
\begin{align*}
\text{DO } I & = 1, N \\
\text{S1: } A(I) & = B(I) + C(I) \\
& \text{ENDDO} \\
\text{DO } I & = 1, N \\
\text{S2: } E(I) & = A(I+1) \times D(I) \\
& \text{ENDDO}
\end{align*}
\]

We have flow dependences from S1 to S2
Illegal Loop Fusion Example

We have flow dependences from S1 to S2

Illegal fusion: On fusing the two loops, we have a dependence from instances of S2 to S1, which was impossible
Transformations: Loop distribution

- **Loop Distribution:** Splits a single loop nest into many, also known as *loop fission.*

```plaintext
DO I = 1, N
  A(I) = B(I) + C(I)
ENDDO

S1:  A(I) = B(I) + C(I)

S2:  E(I) = A(I) * D(I)
ENDDO

DO I = 1, N
  E(I) = A(I) * D(I)
ENDDO
```

- Cycles in dependence graph make this illegal: If there are dependencies from S1 to S2 and dependencies from S2 to S1, you cannot do this.
- **Needed for vectorization**
Loop distribution - 2

- Statements (or loops containing statements may need to be reordered for distribution.

```
DO I = 1, N
   S1:  C(I+1) = A(I) + B(I+1)
   S2:  A(I+1) = D(I) * E(I)
ENDDO
```

- We have dependencies from S2(i) to S1(i+1). No cycles of dependencies among the statements. We need to ensure S2(i) executes before S1(i+1) after distribution.
Loop distribution - 3

DO I = 1, N
S1:  C(I+1) = A(I)+B(I+1)
S2:  A(I+1) = D(I)*E(I)
ENDDO

The loop containing S2 should execute first.

DO I = 1, N
S2:  A(I+1) = D(I)*E(I)
ENDDO
DO I = 1, N
S1:  C(I+1) = A(I)+B(I+1)
ENDDO

Note: Each loop in the code above is parallel!
Loop Tiling or Blocking - 1

• Suppose arrays A and B have row-major layouts

```fortran
DO I = 1, 1000
  DO J = 1, 1000
    A(I,J) = A(I,J) + B(J,I)
  ENDDO
ENDDO
```

• Array A had good locality whereas B has poor locality
• If we interchange, A will have poor locality but not B
• How do we improve locality here: Access smaller sections? Change array layout?
• Is tiling always legal? **No** (more on this later)
Loop Tiling or Blocking - 2

- Use loop tiling or blocking: Create loop nests with $2n$ loops from $n$ loops for one-level of tiling

```fortran
DO IT = 1, 1000, 25
  DO JT = 1, 1000, 25
    DO I = IT, IT+24
      DO J = JT, JT+24
        A(I,J) = A(I,J) + B(J,I)
      ENDDO
    ENDDO
  ENDDO
ENDDO

- Useful for exploiting locality of reference; reduces communication overhead in message-passing by clustering iteration of loop nests
Iteration Space (Loop) Tiling

- A tile in an n-dimensional iteration space is an n-dimensional subset of the iteration space.
- A tile is defined by a set of boundaries, spaced apart in a regular fashion.
- Each tile boundary is an (n-1) dimensional plane.

```plaintext
DO I = 1, N
    DO J = 1, N
        A(I,J) = A(I-1,J) + A(I,J-1)
    ENDDO
ENDDO
```
Iteration Space (Loop) Tiling

```
DO I = 1, N
    DO J = 1, N
        A(I,J) = A(I-1,J) + A(I,J-1)
    ENDDO
ENDDO
```

2 x 2 tiles

[Diagram of iteration space and tiles]
Iteration Space (Loop) Tiling

\[
\text{DO } I = 1, N \\
\text{DO } J = 1, N \\
\quad A(I,J) = A(I-1,J) + A(I,J-1) \\
\text{ENDDO} \\
\text{ENDDO}
\]

2 sets of tile boundaries

Set 1
\(j = \text{constant}\)

Set 2
\(i = \text{constant}\)
Iteration Space (Loop) Tiling

\[
\begin{align*}
&\text{DO } I = 1, N \\
&\quad \text{DO } J = 1, N \\
&\quad \quad A(I,J) = A(I-1,J) + A(I,J-1) \\
&\quad \text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]
Tiling - 1

- Tiling: Clustering iterations into *atomic* execution units.
- *Atomic* execution: Once a tile starts, we should complete all the computations in a tile before moving to another tile. This will avoid any interspersed synchronization once a tile starts execution.
- Atomic execution is guaranteed if there are no cycles of dependences among tiles (or equivalently, tiling is the result of a convex partition)

Sufficient condition to avoid cycles:
All dependences that cross a tile boundary must all do so in the same direction (from one tile to another)
Sufficient condition to avoid cycles:
All dependences that cross a tile boundary must all do so in the same direction (from one tile to another)

Let \( \mathbf{H} \) be a normal to an (n-1) dimensional plane that defines a tile boundary

Then, for all dependence vectors \( \mathbf{d} \), we need

\[
\mathbf{H} \cdot \mathbf{d} \geq 0
\]

This must hold for all boundaries. Condition from Irigoin and Triolet (1988).

A collection of tile boundaries \{H1, ..., Hn\} is valid for n-dimensional tiling if the Hi are linearly independent.
Tiling Example - 1

\[
\begin{align*}
\text{DO } & I = 1, N \\
\text{DO } & J = 1, N \\
A(I,J) & = A(I-1,J+1) + A(I-1,J) + A(I,J-1)
\end{align*}
\]

There are three dependence vectors: (1,-1), (1,0), (0,1)

Iteration Space:
Tiling Example - 2

Suppose we form tile with boundaries whose normals are (1,0) and (0,1). We have cycles among tiles.
Tiling Example - 2

Suppose we form tile with boundaries whose normals are (1,0) and (0,1). We have cycles among tiles...
Other Transformations: Loop Peeling

![DO I = 1, N
    A(I) = B(I) + C(I)
  ENDDO](image)

- Peeling: Extracts one iteration of a loop. Can peel the first or the last iteration off. **Peeling is always legal.**
- By itself, peeling is not useful. Peeling is an enabling transformation
Other Transformations: Loop Peeling

```plaintext
DO I = 1, N
    A(I) = B(I) + C(I)
ENDDO
```

• Peeling: Extracts one iteration of a loop. Can peel the first or the last iteration off. *Peeling is always legal.*

• By itself, peeling is not useful. Peeling is an enabling transformation

  Peel off first iteration

```plaintext
A(1) = B(1) + C(1)
DO I = 2, N
    A(I) = B(I) + C(I)
ENDDO
```
Other Transformations: Loop Peeling

Peeling: Extracts one iteration of a loop. Can peel the first or the last iteration off. **Peeling is always legal.**

By itself, peeling is not useful. Peeling is an enabling transformation.

Peel off last iteration

\[
\begin{align*}
\text{DO } & I = 1, N-1 \\
A(I) & = B(I) + C(I) \\
\text{ENDDO}
\end{align*}
\]

\[
A(N) = B(N) + C(N)
\]
Other Transformations: Loop Shifting

```
DO I = 1, N
   A(I) = B(I) + C(I)
ENDDO
```

• Shifting: Allows to reorder loops, enables loop fusion, ...
  Shifting is always legal and is an enabling transformation
Other Transformations: Loop Shifting

```
DO I = 1, N
   A(I) = B(I) + C(I)
ENDDO
```

- Shifting: Allows to reorder loops, enables loop fusion, ... Shifting is always legal and is an enabling transformation
- Shift the loop index in the code above by +1
Other Transformations: Loop Shifting

• Shifting: Allows to reorder loops, enables loop fusion, ...
  Shifting is always legal and is an enabling transformation

• Shift the loop index in the code above by +1

```fortran
DO I = 1, N
  A(I) = B(I) + C(I)
ENDDO
```

```fortran
DO I = 1+1, N+1
  A(I-1) = B(I-1) + C(I-1)
ENDDO
```
Other Transformations: Loop Shifting

- Shifting: Allows to reorder loops, enables loop fusion, ...
  Shifting is always legal and is an enabling transformation
- Shift the loop index in the code above by +1

\[
\begin{align*}
\text{DO } & I = 1, N \\
A(I) & = B(I) + C(I) \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } & I = 1+1, N+1 \\
A(I-1) & = B(I-1) + C(I-1) \\
\text{ENDDO}
\end{align*}
\]

- Shift the loop index \( I \) by \( d \): Add \( d \) to the loop bounds of \( I \); subtract \( d \) from \( I \) everywhere in the loop body
Code Generation

Suppose we have the following loop (e.g., after skewing):

\[
\begin{align*}
\text{DO } & \ I = 1, \ N \\
\text{DO } & \ J = 1+I, \ N+I \\
\ & \ A(I,J)= w * ( A(I,J-I-1) + A(I-1,J-I+1))
\end{align*}
\]

Now, suppose we use loop interchange, the correct code is:

\[
\begin{align*}
\text{DO } & \ J = 2, \ 2*N \\
\text{DO } & \ I = \text{max}(1,J-N), \ \text{min}(N,J-1) \\
\ & \ A(I,J)= w * ( A(I,J-I-1) + A(I-1,J-I+1))
\end{align*}
\]

How do we get that?
We use *Fourier-Motzkin* variable elimination technique
Fourier Motzkin Elimination - 1

Here is the original loop:

\[
\text{DO } I = 1, N \\
\quad \text{DO } J = 1+I, N+I \\
\quad \quad A(I,J) = w * ( A(I,J-I-1) + A(I-1,J-I+1) )
\]
Here is the original loop:

```
DO  I = 1, N
    DO  J = 1+I, N+I
        A(I,J)= w * ( A(I,J-I-1) + A(I-1,J-I+1))
```

Write out the bounds on loop indices $I$ and $J$ as inequalities:

- $I \geq 1$
- $I \leq N$
- $J \geq 1 + I$
- $J \leq N + I$
Fourier Motzkin Elimination - 1

Here is the original loop:

```
DO I = 1, N
   DO J = 1+I, N+I
      A(I,J)= w * ( A(I,J-I-1) + A(I-1,J-I+1))
   END DO
END DO
```

Write out the bounds on loop indices I and J as inequalities:

- \( I \geq 1 \)
- \( I \leq N \)
- \( J \geq 1 + I \)
- \( J \leq N + I \)

After interchange, J is the new outer loop, I the new inner loop. The bounds on I can depend on J but not vice-versa.
Fourier Motzkin Elimination - 2

The bounds on $I$ can depend on $J$ but not vice-versa.
Idea: Create an equivalent system of inequalities from which variable $I$ is eliminated. Do this successively starting from the new innermost loop to the outermost loop ($J$ here).
The bounds on $I$ can depend on $J$ but not vice-versa. Idea: Create an equivalent system of inequalities from which variable $I$ is eliminated. Do this successively starting from the new innermost loop to the outermost loop ($J$ here)

\begin{align*}
I & \geq 1 \\
I & \leq N \\
J & \geq 1 + I \\
J & \leq N + I
\end{align*}
The bounds on I can depend on J but not vice-versa.
Idea: Create an equivalent system of inequalities from which variable I is eliminated. Do this successively starting from the new innermost loop to the outermost loop (J here)

\[
\begin{array}{ll}
I \geq 1 & I \geq 1 \\
I \leq N & I \leq N \\
J \geq 1 + I & I \leq J - 1 \\
J \leq N + I & I \geq J - N
\end{array}
\]
Fourier Motzkin Elimination - 2

The bounds on I can depend on J but not vice-versa.

Idea: Create an equivalent system of inequalities from which variable I is eliminated. Do this successively starting from the new innermost loop to the outermost loop (J here)

| I >= 1    | I >= 1    |
| I <= N    | I <= N    |
| J >= 1 + I| I <= J - 1|
| J <= N + I| I >= J - N|

What are the bounds of loop I?

| I >= 1 |
| I <= N |
| I <= J - 1 |
| I >= J - N |
Fourier Motzkin Elimination - 2

The bounds on I can depend on J but not vice-versa.

Idea: Create an equivalent system of inequalities from which variable I is eliminated. Do this successively starting from the new innermost loop to the outermost loop (J here)

\[
\begin{align*}
I & \geq 1 \\
I & \leq N \\
J & \geq 1 + I \\
J & \leq N + I
\end{align*}
\]

What are the bounds of loop I?

\[
\begin{align*}
I & \geq 1 \\
I & \leq N \\
I & \leq J - 1 \\
I & \geq J - N
\end{align*}
\]
Fourier Motzkin Elimination - 2

The bounds on $I$ can depend on $J$ but not vice-versa.

Idea: Create an equivalent system of inequalities from which variable $I$ is eliminated. Do this successively starting from the new innermost loop to the outermost loop ($J$ here)

| $I \geq 1$ | $I \geq 1$ |
| $I \leq N$ | $I \leq N$ |
| $J \geq 1 + I$ | $I \leq J - 1$ |
| $J \leq N + I$ | $I \geq J - N$ |

What are the bounds of loop $I$?

| $I \geq 1$ | $I \geq \max(1, J-N)$ |
| $I \leq N$ | $I \leq \min(N, J-1)$ |
| $I \leq J - 1$ | 
| $I \geq J - N$ | do $I = \max(1, J-N), \min(N, J-1)$ |
The bounds on $I$ can depend on $J$ but not vice-versa.

Idea: Create an equivalent system of inequalities from which variable $I$ is eliminated. Do this successively starting from the new innermost loop to the outermost loop ($J$ here)

\[
\begin{align*}
I & \geq 1 & I & \geq 1 \\
I & \leq N & I & \leq N \\
J & \geq 1 + I & I & \leq J - 1 \\
J & \leq N + I & I & \geq J - N
\end{align*}
\]

Eliminate $I$: Set every **lower bound of $I$** less than or equal to every **upper bound of $I$**
Fourier Motzkin Elimination - 3

The bounds on $I$ can depend on $J$ but not vice-versa.

Idea: Create an equivalent system of inequalities from which variable $I$ is eliminated. Do this successively starting from the new innermost loop to the outermost loop ($J$ here).

$$
\begin{align*}
I & \geq 1 & I & \geq 1 \\
I & \leq N & I & \leq N \\
J & \geq 1 + I & I & \leq J - 1 \\
J & \leq N + I & I & \geq J - N
\end{align*}
$$

Eliminate $I$: Set every lower bound of $I$ less than or equal to every upper bound of $I$

$$
\begin{align*}
1 & \leq N \\
1 & \leq J - 1 \\
J - N & \leq N \\
J - N & \leq J - 1
\end{align*}
$$
Fourier Motzkin Elimination - 3

The bounds on $I$ can depend on $J$ but not vice-versa.

Idea: Create an equivalent system of inequalities from which variable $I$ is eliminated. Do this successively starting from the new innermost loop to the outermost loop ($J$ here).

\[
\begin{align*}
I & \geq 1 & & I \geq 1 \\
I & \leq N & & I \leq N \\
J & \geq 1 + I & & I \leq J - 1 \\
J & \leq N + I & & I \geq J - N
\end{align*}
\]

Eliminate $I$: Set every lower bound of $I$ less than or equal to every upper bound of $I$.

\[
\begin{align*}
1 & \leq N & & 1 \leq N \\
1 & \leq J - 1 & & J \geq 2 \\
J - N & \leq N & & J \leq 2N \\
J - N & \leq J - 1 & & 1 \leq N
\end{align*}
\]
The bounds on I can depend on J but not vice-versa.

Idea: Create an equivalent system of inequalities from which variable I is eliminated. Do this successively starting from the new innermost loop to the outermost loop (J here)

\[
\begin{align*}
1 & \leq N \\
J & \geq 2 \\
J & \leq 2N \\
1 & \leq N
\end{align*}
\]

What are the limits of J?

\[
\begin{align*}
J & \geq 2 \\
J & \leq 2N
\end{align*}
\]

\[
do \ J = 2, 2*N
\]
The bounds on I can depend on J but not vice-versa.

Idea: Create an equivalent system of inequalities from which variable I is eliminated. Do this successively starting from the new innermost loop to the outermost loop (J here)

\[
\begin{align*}
1 & \leq N \\
J & \geq 2 \\
J & \leq 2N \\
1 & \leq N
\end{align*}
\]

Eliminate J: Set every lower bound of J less than or equal to every upper bound of J

\[
\begin{align*}
J & \geq 2 \\
J & \leq 2N \\
2 & \leq 2N
\end{align*}
\]
Fourier Motzkin Elimination - 6

The bounds on I can depend on J but not vice-versa. Idea: Create an equivalent system of inequalities from which variable I is eliminated. Do this successively starting from the new innermost loop (I here) to the outermost loop (J here)

Here is the code after using Fourier-Motzkin elimination:

```c
IF ( (1 <= N) && (2 <= 2*N) )
  DO J = 2, 2*N
    DO I = max(1,J-N), min(N,J-1)
      A(I,J) = w * ( A(I,J-I-1) + A(I-1,J-I+1) )
  END DO
END DO
```

The Ohio State University
Fourier Motzkin Elimination - 6

The bounds on I can depend on J but not vice-versa.
Idea: Create an equivalent system of inequalities from which variable I is eliminated. Do this successively starting from the new innermost loop (I here) to the outermost loop (J here)

Here is the code after using Fourier-Motzkin elimination:

```
DO J = 2, 2*N
  DO I = max(1,J-N), min(N,J-1)
    A(I,J)= w * ( A(I,J-I-1) + A(I-1,J-I+1) )
```

Non-polyhedral Approaches

• Non-polyhedral approaches work well for perfectly-nested loops
• The dependence distance and related abstractions are not adequate to handle arbitrary nestings of loops and sequences of loops
• Often, one is interested in a target transformed code for which we need to answer questions (See papers by Vasilache et al. for work on this):
  – Is the target code legal?
  – How do we get there
• Composition of transformations is a problem
Why a polyhedral model?

- A polyhedral model can be used to abstract program information

In the polyhedral model (Feautrier, 92):
- Composition of transformations are easily expressed
- Transformation legality is easily checked
- Natural expression of parallelism
Background: Affine Hyperplane

The set $X$ of all vectors $x \in \mathbb{Z}^n$ such that $h \cdot \vec{x} + c = k$ for $k \in \mathbb{Z}$, forms an affine hyperplane.

The set of parallel hyperplane instances corresponding to different values of $k$ is characterized by the vector $\vec{h}$ which is normal to the hyperplane and the constant $c$.

Each instance of a hyperplane is an $n-1$ dimensional affine sub-space of the $n$-dimensional space.

Two vectors $\vec{x}_1$ and $\vec{x}_2$ lie in the same hyperplane if $h \cdot \vec{x}_1 = h \cdot \vec{x}_2$. 
Background: Polyhedron, Polytope

The set of all vectors $\vec{x} \in \mathbb{Z}^n$ such that $A\vec{x} + \vec{b} \geq \vec{0}$, where $A$ is an integer matrix, defines a (convex) integer polyhedron.

A polytope is a bounded polyhedron.

Each run-time instance of a statement $S$ is defined by its iteration vector $\vec{i}$ which contains values for the indices of the loops surrounding $S$, from the outermost to the innermost.
Polyhedral Representation (1)

- Loops have only affine control
- Loop (i and j) bounds are affine (linear plus constant term) functions of outer loop indices and structure parameters (n); so are the conditionals in the loop bodies

```c
for (i=1; i<=n; ++i)
  for (j=1; j<=n; ++j)
    if (i <= n-j+2)
      S1: A[i] = ...
```
Polyhedral Representation (2)

• Iteration domain represented as integer polyhedron

\[
\text{for (i=1; i<=n; ++i) for (j=1; j<=n; ++j) if (i <= n-j+2) S1: } \quad A[i] = ...
\]

\[
D_{S1} = \begin{bmatrix}
1 & 0 & 0 & -1 \\
-1 & 0 & 1 & 0 \\
0 & 1 & 0 & -1 \\
0 & -1 & 1 & 0 \\
-1 & -1 & 1 & 2
\end{bmatrix} \begin{pmatrix}
i \\
j \\
n \\
1
\end{pmatrix} \geq \bar{0}
\]
Polyhedral Representation (2)

• Iteration domain represented as integer polyhedron

\[
\begin{align*}
\text{for } (i=1; & \ i\leq n; ++i) \\
& \text{for } (j=1; \ j\leq n; ++j) \\
& \text{if } (i \leq n-j+2) \\
\text{S1: } A[i] = \ldots
\end{align*}
\]

\[
D_{S1} = \begin{bmatrix}
1 & 0 & 0 & -1 \\
-1 & 0 & 1 & 0 \\
0 & 1 & 0 & -1 \\
0 & -1 & 1 & 0 \\
-1 & -1 & 1 & 2
\end{bmatrix} \begin{pmatrix}
i \\
j \\
\bar{j} \\
n \end{pmatrix} \geq \mathbf{0}
\]
Polyhedral Representation (3)

- Iteration domain represented as integer polyhedron

```plaintext
for (i=1; i<=n; ++i)
    for (j=1; j<=n; ++j)
        if (i<=n-j+2)
            S1: A[i] = ...
```

Iteration domain of $S_1$
Polyhedral Representation (4)

- Memory accesses: static references, represented as affine functions of loop indices and structure parameters

```cpp
for (i=1; i<=n; ++i) {
    S1: c[i] = 0;
    for (j=1; j<=n; ++j)
        S2: c[i] = c[i] + a[i][j] * b[j];
}
```

- Loop indices (represented from outermost to innermost) around S2 referred to as

\[ x_{S2} = \begin{pmatrix} i \\ j \end{pmatrix} \]
Polyhedral Representation (5): Memory access

for (i=1; i<=n; ++i) {
    S1: c[i] = 0;
    for (j=1; j<=n; ++j)
        S2: c[i] = c[i] + a[i][j] * b[j];
}

\[
x_{S2} = \begin{pmatrix} i \\ j \end{pmatrix}
\]

\[
f_c(x_{S2}) = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} x_{S2} \\ n \\ 1 \end{pmatrix}
\]

\[
f_b(x_{S2}) = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \begin{pmatrix} x_{S2} \\ n \\ 1 \end{pmatrix}
\]

\[
f_a(x_{S2}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{pmatrix} x_{S2} \\ n \\ 1 \end{pmatrix}
\]
Polyhedral Representation (6): Dependence

```
for (i=1; i<=3; ++i) {
    S1: y[i] = 0;
    for (j=1; j<=3; ++j)
        S2: y[i] = y[i] + 1;
}
```

Data dependence between S1 and S2: a subset of the Cartesian product of $D_{S1}$ and $D_{S2}$

There is a dependence from S1 to S2 ($D(S1 \delta S2)$) if we can find valid instances $S1(i_{S1})$ and $S2(i_{S2}, j_{S2})$ that access the same memory location.
Polyhedral Representation (6): Dependence

\[
\text{for } (i=1; i<=3; ++i) \{
\text{S1: } y[i] = 0;
\text{ for } (j=1; j<=3; ++j)
\text{S2: } y[i] = y[i] + 1;
\}
\]

Data dependence between S1 and S2: a subset of the Cartesian product of
\( D_{S1} \) and \( D_{S2} \)

There is a dependence between S1 to S2 \( D(S1 \delta S2) \) if we can find valid instances \( S1(i_{S1}) \) and \( S2(i_{S2}, j_{S2}) \) that access the same memory location

\[
D(S1 \delta S2) = \begin{bmatrix}
1 & 0 & 0 & -1 \\
-1 & 0 & 0 & 3 \\
0 & 1 & 0 & -1 \\
0 & -1 & 0 & 3 \\
0 & 0 & 1 & -1 \\
0 & 0 & -1 & 3 \\
1 & -1 & 0 & 0
\end{bmatrix} \begin{pmatrix}
i_{S1} \\
i_{S2} \\
j_{S2} \\
1
\end{pmatrix} \geq \tilde{0}
\]

\[
\begin{align*}
\text{and } & \quad = 0
\end{align*}
\]
Polyhedral Representation (6): Dependence

for (i=1; i<=3; ++i) {
    S1: y[i] = 0;
        for (j=1; j<=3; ++j)
    S2:    y[i] = y[i] + 1;
}

Data dependence between S1 and S2: a subset of the Cartesian product of $D_{S1}$ and $D_{S2}$

There is a dependence between S1 to S2 ($D(S1\delta S2)$) if we can find valid instances $S1(i_{S1})$ and $S2(i_{S2}, j_{S2})$ that access the same memory location.

Bounds of i for S1

$$D(S1\delta S2) = \begin{bmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 3 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 3 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 3 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{pmatrix} i_{S1} \\ i_{S2} \\ j_{S2} \\ 1 \end{pmatrix} \geq \bar{0}$$

$$= 0$$
Polyhedral Representation (6): Dependence

for (i=1; i<=3; ++i) {
    S1: y[i] = 0;
    for (j=1; j<=3; ++j)
        S2: y[i] = y[i] + 1;
}

Data dependence between S1 and S2: a subset of the Cartesian product of $D_{S1}$ and $D_{S2}$

There is a dependence between S1 to S2 ($D(S1\delta S2)$) if we can find valid instances $S1(i_{S1})$ and $S2(i_{S2}, j_{S2})$ that access the same memory location.

Bounds of i for S2

$$D(S1\delta S2) = \begin{bmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 3 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 3 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 3 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{pmatrix} i_{S1} \\ i_{S2} \\ j_{S2} \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$
Polyhedral Representation (6): Dependence

for (i=1; i<=3; ++i) {
S1: y[i] = 0;
    for (j=1; j<=3; ++j)
S2: y[i] = y[i] + 1;
}

Data dependence between S1 and S2: a subset of the Cartesian product of $D_{S1}$ and $D_{S2}$

There is a dependence between S1 to S2 ($D(S1 \delta S2)$) if we can find valid instances $S1(i_{S1})$ and $S2(i_{S2}, j_{S2})$ that access the same memory location
Polyhedral Representation (6): Dependence

```c
for (i=1; i<=3; ++i) {
    S1: y[i] = 0;
    for (j=1; j<=3; ++j)
    S2: y[i] = y[i] + 1;
}
```

Data dependence between S1 and S2: a subset of the Cartesian product of $D_{S1}$ and $D_{S2}$

There is a dependence between S1 to S2 ($D(S1 \delta S2)$) if we can find valid instances $S1(i_{S1})$ and $S2(i_{S2}, j_{S2})$ that access the same memory location.

$$D(S1 \delta S2) = \begin{bmatrix}
1 & 0 & 0 & -1 \\
-1 & 0 & 0 & 3 \\
0 & 1 & 0 & -1 \\
0 & -1 & 0 & 3 \\
0 & 0 & 1 & -1 \\
0 & 0 & -1 & 3 \\
\end{bmatrix} \begin{pmatrix}
i_{S1} \\
i_{S2} \\
j_{S2} \\
1
\end{pmatrix} \geq \vec{0} = 0$$
How prevalent are affine codes?

• Innermost core computations in many codes
  – Dense linear algebra
  – Image and signal processing
  – Computational Electromagnetics (FDTD)
  – Explicit PDE solvers (e.g. SWIM, SWEEP3D)
  – Integral transforms in quantum chemistry (AO-to-MO)

• May increase in the future (esp. scientific apps)
  – Codes with direct data access significantly better than indirect-data access: power & performance
  – Structured-sparse (block sparse) is better than arbitrary sparse (e.g. OSKI)
  – RINO (Regular-Inner-Nonregular-Outer) algorithms should be attractive for many-core processors
Polyhedral Model: One-dimensional scheduling

- One-dimensional scheduling: Time (scalar) at which a statement instance is executed.
- Only specify the outermost loop

Original Schedule
Initial outermost loop is i

\[ \theta(S_1(i)) = i \]
\[ \theta(S_2(i, j)) = i \]
Polyhedral Model: One-dimensional scheduling

- Only specify the outermost loop

\[
\theta(S1(i)) = i \\
\theta(S2(i,j)) = i + n
\]

All instances of S1 executed before any instance of S2

```c
for (i=0; i<=n; ++i) {
    S1(i);
    for (j=0; j<=n; ++j)
        S2(i,j);
}
```

Distribute loops

```c
for (i=0; i<=n; ++i)
    S1(i);
for (i=n; i<2*n; ++i)
    for (j=0; j<=n; ++j)
        S2(i-n,j);
```
Polyhedral Model: One-dimensional scheduling

- Only specify the outermost loop

\[
\theta(S_1(i)) = i
\]
\[
\theta(S_2(i, j)) = j + n
\]

The outermost loop for \(S_2\) becomes \(j\)

\[
\text{for (} i=0; \ i<=n; \ ++i) \ \{ \\
\quad S_1(i); \\
\quad \text{for (} j=0; \ j<=n; \ ++j) \\
\quad \quad S_2(i, j); \\
\}\}
\]

Distribute loops and interchange loops around \(S_2\)

\[
\text{for (} i=0; \ i<=n; \ ++i) \ \{ \\
\quad S_1(i); \\
\quad \text{for (} j=n; \ j<2*n; \ ++j) \\
\quad \quad \text{for (} i=0; \ i<=n; \ ++i) \\
\quad \quad \quad S_2(i, j-n); \\
\}\}
\]
Affine schedules (one-dimensional)

Property (Causality condition for schedules)

Given a dependence from \( R \) to \( S \) written as \( R \delta S \), schedules \( \theta_R \) and \( \theta_S \) are legal if and only if for each pair of instances in the \( R \delta S \) dependence:

\[
\theta_R(x_R^\rightarrow) < \theta_S(x_S^\rightarrow)
\]

Equivalently:

\[
\theta_S(x_S^\rightarrow) - \theta_R(x_R^\rightarrow) - 1 \geq 0
\]
Code Generation from Schedules (CLooG)

• But how do schedules help? One needs to generate code
• The answer: CLooG (Thanks to Cedric Bastoul)
• CLooG is a free software and library to generate code for scanning polyhedra from schedules
• CLooG available at
  – http://www.cloog.org/
• CLooG is used now in various areas (outside of code generation from polyhedral models)
  – to build control automata for high-level synthesis (VHDL code generation from loop nests)
  – to find the best polynomial approximation of a function
Polyhedral model: Scheduling-based (1)

• Pouchet et al. (CGO 2007, PLDI 2008a): Combines iterative optimization with scheduling-based polyhedral model
  – Pouchet et al. (CGO 2007) use one-dimensional time schedules (which create an outer sequential loop and inner parallel loops);
  – in PLDI 2008a, they use multi-dimensional time schedules.
• Tiling is done as a later step after finding a loop structure based on one-dimensional schedules
• Iterative optimization uses for tile size determination and for finding good loop unroll factors
• Scheduling-based polyhedral model can be traced back to systolic array synthesis
Polyhedral model: Scheduling-based (2)

• Step 1: Search space construction:
  – Efficiently construct a space of all legal, distinct one-dimensional (time) affine schedules (LeTSeE)
  – Rely on the polyhedral model and Integer Linear Programming to guarantee completeness and correctness of the search space properties
  – Search space will encompass unique, distinct compositions of reversal, skewing, interchange, fusion, peeling, shifting, distribution
Polyhedral model: Scheduling-based (3)

• Step 2: Search space exploration
  – Perform exhaustive scan to discover wall clock optimal schedule, and evidences of intricacy of the best transformation
  – Build an efficient heuristic to accelerate the space traversal
References - 1


• C. Ancourt and F. Irigoin. **Scanning polyhedra with DO loops.** in PPOPP'91, 1991.

References - 2


References - 3


References - 4


