A Loop Transformation Theory and an Algorithm to Maximize Parallelism

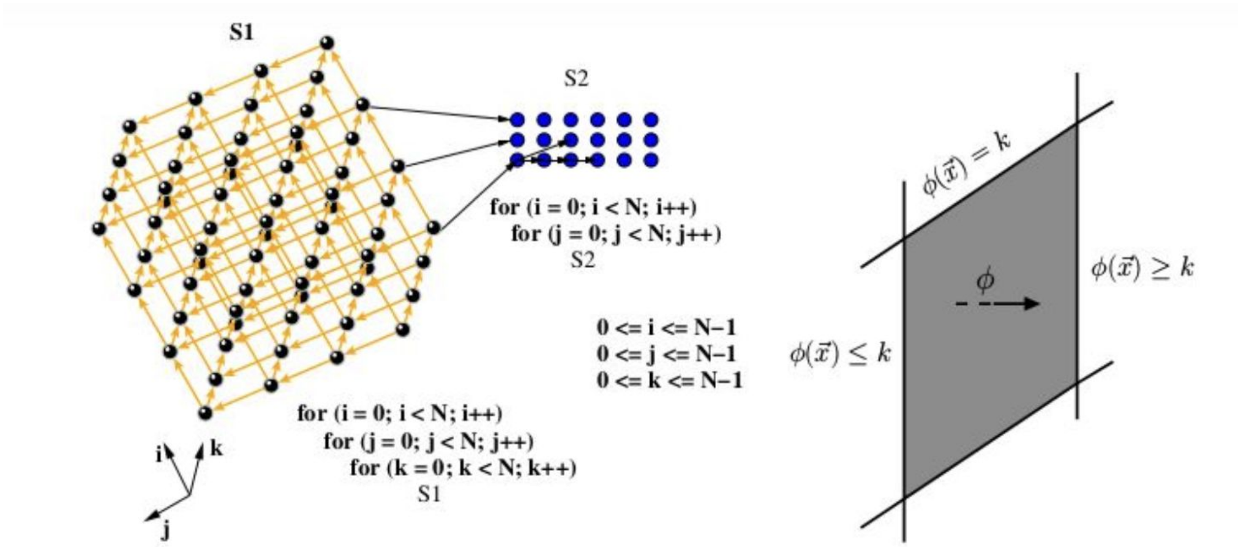
Z. Wang, M. Yuan, Z. Zhang , Z. Zhou
From Group 7

Outline

- Introduction
- Distance vectors & Unimodular Transformation
- Direction vectors
- Implementation
- Summary

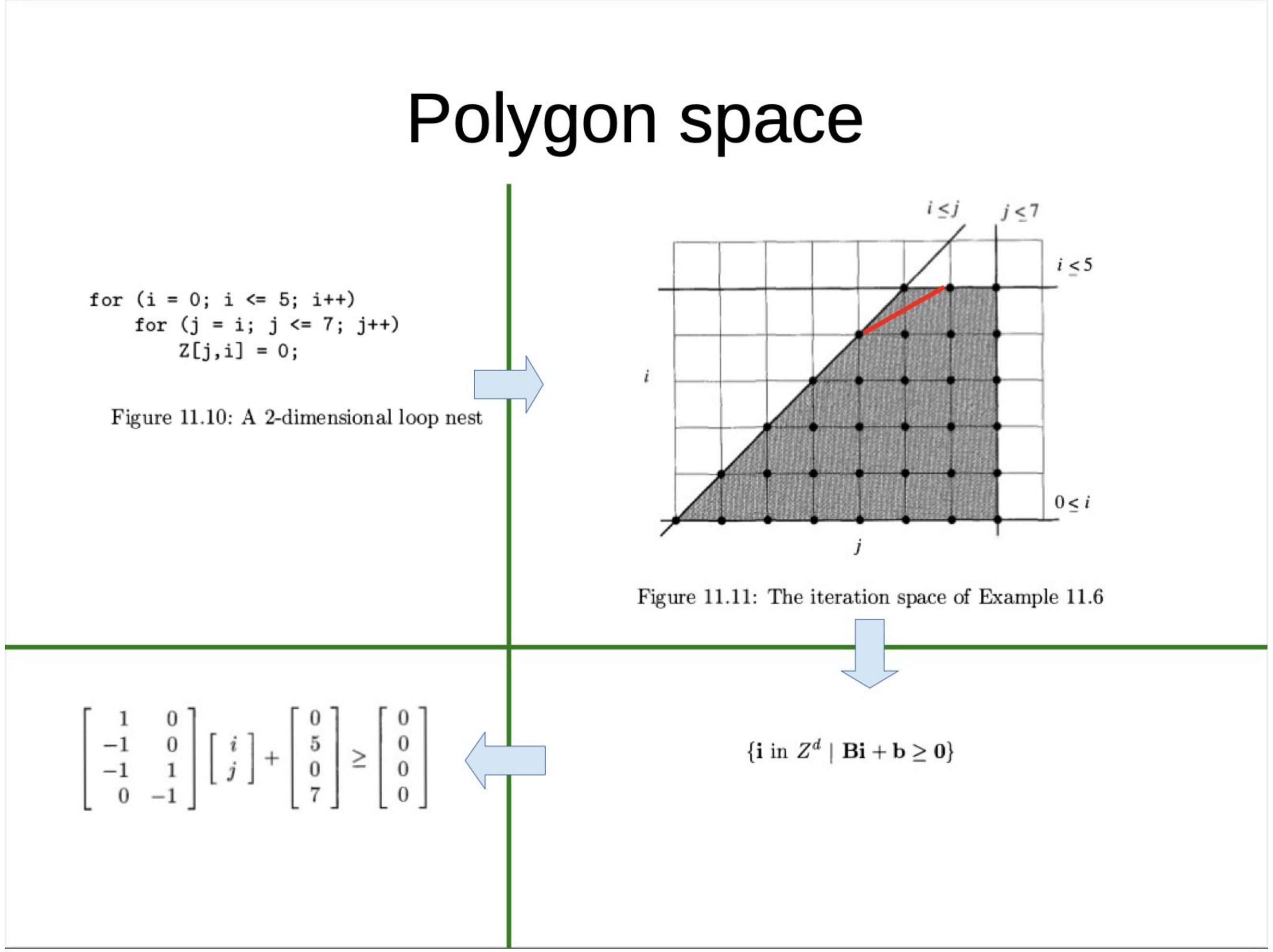
Introduction

- Why we need loop parallelization
- Model to represent large number of instructions
    - Polyhedral Model



Polyhedral Model

- Polyhedral Model, or Polytope Model, is a mathematical framework for programs that perform large numbers of operations -- too large to be explicitly enumerated
- Mostly commonly used in nest loop optimization
- Lattice points are used to denote the instance of each statement

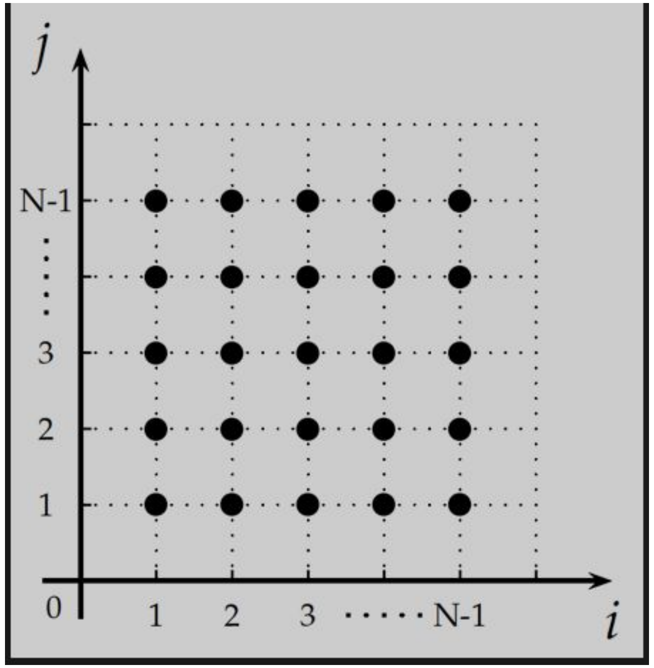


$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Example of Polyhedral Model

- In the below example, each instance / execution of statement A[i,j] is mapped to a dot from the right coordinate

```
for (int i = 1; i < N; i++)
    for (int j = 1; j < N; j++)
        A[i, j] = f(A[i - 1][j], A[i][j - 1]);
```



Two types of vector

- Distance vector:
  - represent the "**shape**" of dependence, but no information of source and sink

- Direction vectors:
  - captures of the "direction" of dependence, but no information of “shape”

source: https://engineering.purdue.edu/~milind/ece573/2011spring/lecture-14.pdf

Distance Vectors

Let's say in a loop, to compute V[i], we need the value of V[i - n]. then the distance vector can be represented as (i - (i - n)) = (n)

In 2D, to compute V[i + 1][j - 2] we need the value of V[i][j] and V[i - 1][j - 2]. then the distance vector can be represented as ((1, -2),(2, 0))

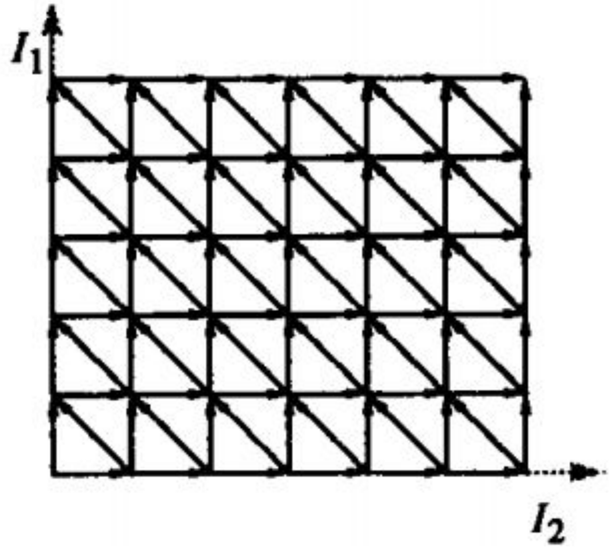
Why do we say the distance vector captures the information of "shape"?

Distance Vectors

(a)

for I_1 := *0* **to** *5* **do**
  for I_2 := *0* **to** *6* **do**
    $a[I_2 + 1] := 1/3 * (a[I_2] + a[I_2 + 1] + a[I_2 + 2]);$

$$D = \{(0,1),(1,0),(1,-1)\}.$$



legality of Unimodular Transformation

- Dependence vector must be **lexicographically positive**
- A unimodular transformation is legal if and only if:

$$\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$$

we will see what they mean later

Unimodular Transformation

There are three elementary transformations:

Permutation:

A permutation σ on a loop nest transfer iteration $(p_1, ..., p_n)$ to $(p_{\sigma_1}, ..., \sigma_n)$

Reversal:

Reversal of the $i\text{th}$ loop, for example: $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$.

Unimodular Transformation

There are three elementary transformations:

Skewing:

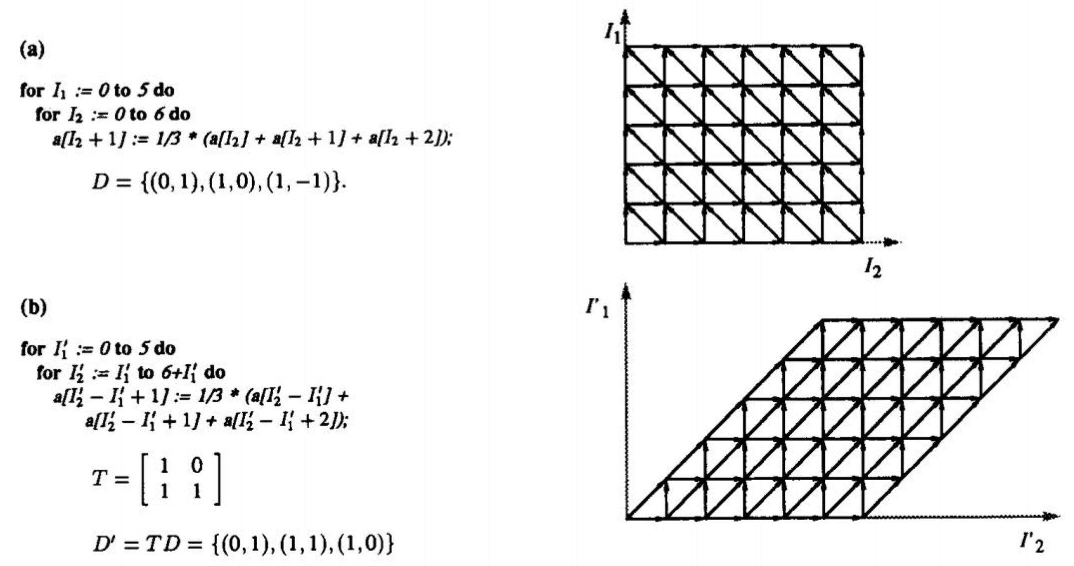


Fig. 1. Iteration space and dependences of (a) a source loop nest, and the (b) skewed loop nest.

Compounding of unimodular Transformation

```
for i from 1 to N:
    for j from 1 to N:
        a[i][j] = a[i][j] + a[i + 1][j - 1]
```

We can see it has dependence $d = (1, -1)$, let's say we apply a loop interchange $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

is Td legal?

No! it must be lexicographically positive, but $Td = (-1, 1)$.

Compounding of unimodular Transformation

$$T' = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

If we compounding the interchange with a reversal, T'.

The new transformation is legal because $T'(1,-1) = (1,1)$, which is lexicographically positive.

Direction Vectors

The problem with distance vectors: cannot represent general loop nests,

```
for I1 := 0 to N do
    for I2 := 0 to N do
        b := g(b)
    end for
end for
```

Itr (i, j) must precede itr $(i, j{+}1) \longrightarrow (0, 1)$
Itr (i, N) must precede itr $(i{+}1, 0) \longrightarrow (0, -N)$

Example 1 does not have any exploitable parallelism.

```
for I1 := 0 to N do
    for I2 := 0 to N do
        a[I1, I2] := a[I1 + 1, b[I2]]
    end for
end for
```

Iteration (i, j) must precede iteration $(i{+}1, b[j])$

Example 2 contains parallelism, but cannot be represented by distance vectors.

Direction Vectors

Direction vector: each component d_i of a dependence vector $\vec{d}$ now a possibly infinite range of integers, represented by $[d_i^{min}, d_i^{max}]$,

$$d_i^{min} \in \mathcal{Z} \cup \{-\infty\},\ d_i^{max} \in \mathcal{Z} \cup \{\infty\} \text{ and } d_i^{min} \leq d_i^{max}$$

A direction vector therefore represents a set $\mathcal{E}(\vec{d})$ of distance vectors,

$$\vec{d} = ([1,2],[1,2]) \text{ represents } \mathcal{E}(\vec{d}) = \{(1,1),(1,2),(2,1),(2,2)\}$$

Some conventions

d_i^{min}	$d_i^{min} = d_i^{max}$	([1, 1], [2, 2]) ⟶ (1, 2)
+	[1, ∞]	([1, ∞], [2, 2]) ⟶ (+, 2)
−	[-∞, -1]	([-∞, -1], [2, 2]) ⟶ (−, 2)
±	[-∞, ∞]	([-∞, ∞], [2, 2]) ⟶ (±, 2)

Direction Vectors

As mentioned before, a unimodular transformation is legal for $\vec{d}$ if,

$$\forall \vec{d} : \forall \vec{e} \in \mathcal{E}(\vec{d}) : T\vec{e} \succ \vec{0}$$

- The direction vector can represent an infinite set of distance vectors!
  - Problem: hard to handle infinite distance vectors.
  - Solution: an arithmetic is defined to operate on direction vectors directly.

positive	$d^{min} > 0$	$d > 0$
nonnegative	$d^{min} \geq 0$	$d \geq 0$
negative	$d^{max} < 0$	$d < 0$
nonpositive	$d^{max} \leq 0$	$d \leq 0$

Lexicographically positive can apply to general dependence vectors!

Direction Vectors

To enable unimodular transformation, we define component addition to be,

$$[a, b] + [c, d] = [a+c, b+d]$$

we define multiplication by a scalar as,

$$s[a,b] = \begin{cases} [sa, sb] & \text{if } s > 0 \\ [0,0] & \text{if } s = 0 \\ [sb, sa] & \text{otherwise} \end{cases}$$

Let D be the set of dependence vectors of a computation. A unimodular transformation T is legal if

$$\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$$

Just like distance vectors!

Direction Vectors - An Example

```
for I_1 := 0 to N do
    for I_2 := 0 to N do
        for I_3 := 0 to N do
            (a[I_1, I_3], b[I_1, I_2, I_3]) :=
            f(a[I_1, I_3], a[I_1 + 1, I_3 − 1], b[I_1, I_2, I_3], b[I_1, I_2, I_3 − 1])
        end for
    end for
end for
```

The dependence vectors for this nest are,

$$D = \{(0, +, 0), (1, \pm, -1), (0, 0, 1)\}$$

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

wavefront skew   skewing middle loop   permuting I_2 and I_3

Make the two outermost loops to canonical form

Produce a loop nest with parallelism

The transformed dependences D' becomes

$$D' = \{(0, 0, +), (1, 1, \pm), (1, 0, 0)\}$$

parallelize the loop nest with distance vector

1. **Canonical form**

Transfer a loop nest into a **fully permutable** loop nest.

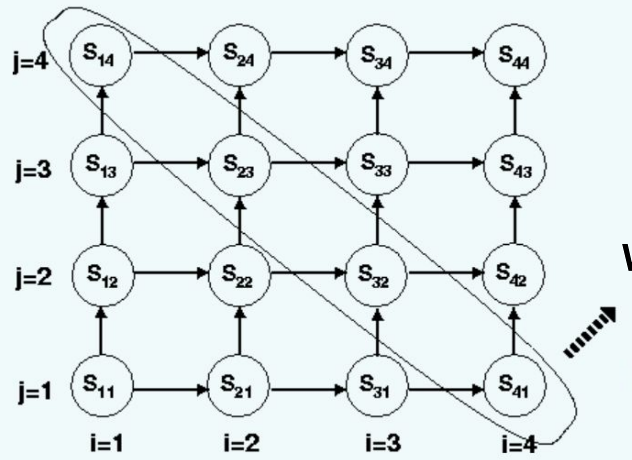
2. **Wavefront transform**

Use **wavefront transformation** to transform the loop nest for parallelization.

What is fully permutable loop nest?

```
for i = 1:N {
  for j = 1:M {
    A(i,j) = A(i-1,j) + A(i,j-1);
  }
}
```

i ↔ j , loops are still correct!



What is Wavefront transformation?

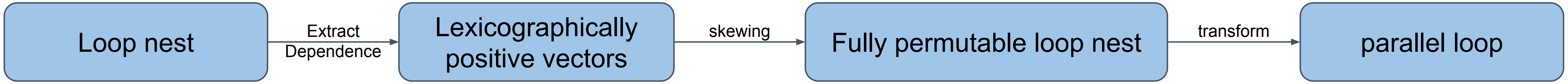
$$\begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

Wavefront

```
for c = 2:N+M {
  for i = max(1,c-M), min(N,c-1) {
    A(i, c-i) = A(i-1, c-i) + A(i, c-i-1);
  }
}
```

All inner loops (n-1 loops) are parallelizable after those transformations!

Implementation in general



- Same procedure
- Distance vectors → Direction vectors
- skewing → SRP transformation
- Not all loops are parallelizable:

1) serializing loops, loops with dependence components including both $+\infty$ and $-\infty$; these loop cannot be included in the outermost fully permutable nest and can be ignored for that nest.
2) loops that can be included via the SRP transformation, an efficient transformation that combines permutation, reversal, and skewing.
3) the remaining loops; they may possibly be included via a general transformation using the time cone method.

- loops are no longer fully permutable
- no longer n-1 parallel loops

Theorem 6.2: Let $L = \{I_1, \cdots, I_n\}$ be a loop nest with lexicographically positive dependences $\vec{d} \in D$, and $D^i = \{\vec{d} \in D | (d_1, \cdots, d_{i-1}) \nsucc \vec{0}\}$. Loop I_j can be made into a fully permutable nest with loop I_i, where $i < j$, via reversal and/or skewing, if

$$\forall \vec{d} \in D^i : (d_j^{\min} \neq -\infty \wedge (d_j^{\min} < 0 \rightarrow d_i^{\min} > 0)) \text{ or}$$
$$\forall \vec{d} \in D^i : (d_j^{\max} \neq \infty \wedge (d_j^{\max} > 0 \rightarrow d_i^{\min} > 0)) .$$

Proof: All dependence vectors for which $(d_1, \cdots, d_{i-1}) \succ \vec{0}$ do not prevent loops I_i and I_j from being fully permutable and can be ignored. If

$$\forall \vec{d} \in D^i : (d_j^{\min} \neq -\infty \wedge (d_j^{\min} < 0 \rightarrow d_i^{\min} > 0))$$

then we can skew loop I_j by a factor of f with respect to loop I_i where

$$f \geq \max_{\{\vec{d} | \vec{d} \in D^i \wedge d_i^{\min} \neq 0\}} \lceil -d_j^{\min} / d_i^{\min} \rceil$$

to make loop I_j fully permutable with loop I_i. If instead the condition

$$\forall \vec{d} \in D^i : (d_j^{\max} \neq \infty \wedge (d_j^{\max} > 0 \rightarrow d_i^{\min} > 0)) .$$

holds, then we can reverse loop I_j and proceed as above. □

How to transform to parallel loops?

Suppose we have the loop nest

$$
\begin{array}{l}
\textbf{for } I_1 := \cdots \\
\quad \cdots \\
\quad\quad \textbf{for } I_n := \cdots \\
\quad\quad\quad S(I_1, \cdots, I_n);
\end{array}
$$

Transform indices:

$$\begin{bmatrix} I_1 \\ \vdots \\ I_n \end{bmatrix} = T^{-1} \begin{bmatrix} I'_1 \\ \vdots \\ I'_n \end{bmatrix}.$$

Transform bounds:

1. Extract inequalities
2. Find absolute maximum and minimum for each loop
3. transform indices
4. calculate new loop bounds

An example loop nest:

$$
\begin{array}{l}
\textbf{for } I_1 := 1 \textbf{ to } n_1 \textbf{ do} \\
\quad \textbf{for } I_2 := 2I_1 \textbf{ to } n_2 \textbf{ do} \\
\quad\quad \textbf{for } I_3 := 2I_1 + I_2 - 1 \textbf{ to } \min(I_2, n_3) \textbf{ do} \\
\quad\quad\quad S(I_1, I_2, I_3);
\end{array}
$$

Step 1: Extract inequalities:

$$
\begin{array}{ll}
I_1 \geq 1 & I_1 \leq n_1 \\
I_2 \geq 2I_1 & I_2 \leq n_2 \\
I_3 \geq 2I_1 + I_2 - 1 & I_3 \leq I_2 \quad I_3 \leq n_3
\end{array}
$$

Step 2: Find absolute maximum and minimum for each loop index:

$$
\begin{array}{ll}
I_1^{\min} = 1 & I_1^{\max} = n_1 \\
I_2^{\min} = 2 \times 1 = 2 & I_2^{\max} = n_2 \\
I_3^{\min} = 2 \times 1 + 2 - 1 = 3 & I_3^{\max} = \min(n_2, n_3)
\end{array}
$$

Step 3: Transform indices: $I_1 \Rightarrow I'_3 \quad I_2 \Rightarrow I'_2 \quad I_3 \Rightarrow I'_1$

Inequalities

$$
\begin{array}{lll}
I'_3 \geq 1 & I'_3 \leq n_1 & \\
I'_2 \geq 2I'_3 & I'_2 \leq n_2 & \\
I'_1 \geq 2I'_3 + I'_2 - 1 & I'_1 \leq I'_2 & I'_1 \leq n_3
\end{array}
$$

Maxima and minima

$$
\begin{array}{ll}
I_1'^{\min} = 3 & I_1'^{\max} = \min(n_2, n_3) \\
I_2'^{\min} = 2 & I_2'^{\max} = n_2 \\
I_3'^{\min} = 1 & I_3'^{\max} = n_1
\end{array}
$$

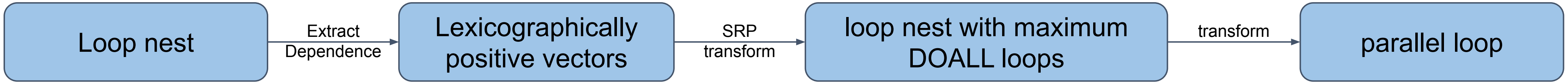
Step 4: Calculate new loop bounds:

Index	Inequality (index on LHS)	Substituting in $I'^{\min}$ and $I'^{\max}$	Result
I'_2	$I'_2 \geq 2I'_3$	$I'_2 \geq 2$	$I'_2 \geq 2$
	$I'_2 \geq I'_1$		$I'_2 \geq I'_1$
	$I'_2 \leq n_2$		$I'_2 \leq n_2$
	$I'_2 \leq I'_1 - 2I'_3 + 1$	$I'_2 \leq I'_1 - 1$	$I'_2 \leq I'_1 - 1$
I'_3	$I'_3 \geq 1$		$I'_3 \geq 1$
	$I'_3 \leq n_1$		$I'_3 \leq n_1$
	$I'_3 \leq (I'_1 - I'_2 + 1)/2$		$I'_3 \leq \lfloor (I'_1 - I'_2 - 1)/2 \rfloor$

The loop nest after transformation:

$$
\begin{array}{l}
\textbf{for } I'_1 := 3 \textbf{ to } \min(n_3, n_2) \textbf{ do} \\
\quad \textbf{for } I'_2 := I'_1 \textbf{ to } \min(n_2, I'_1 - 1) \textbf{ do} \\
\quad\quad \textbf{for } I'_3 := 1 \textbf{ to } \min(n_1, \lfloor (I'_1 - I'_2 + 1)/2 \rfloor) \textbf{ do} \\
\quad\quad\quad S(I'_3, I'_2, I'_1);
\end{array}
$$

Summary



The general step for loop transformations:

- Extract direction vector (lexicographically positive)
- Use SRP transformation to get maximum parallelizable loops
- Get transformed the indices and boundaries

Our plan — polyhedral model + affine transformation

- Handles a wider class of programs and transformations than the unimodular framework
- Automatic parallelization
- Data locality optimizations
- Memory management optimizations

Backup

Affine Transformation

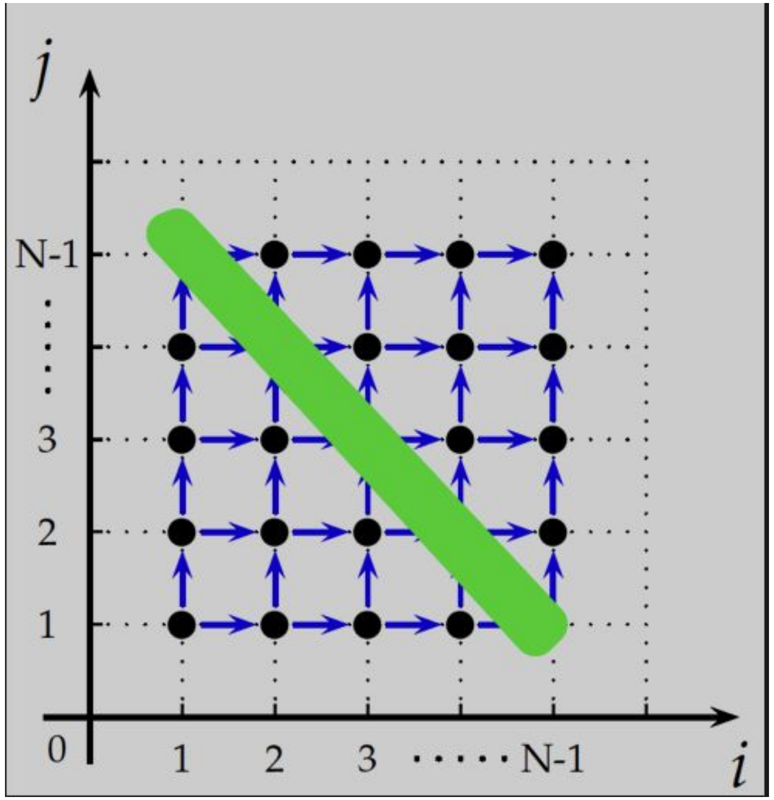
- $c_0 + c_1 * v_1 + c_2 * v_2 + \ldots + c_n * v_n$
- Such expression is also known as linear expression
- Strictly speaking, an affine transformation is linear only if c_0 is 0

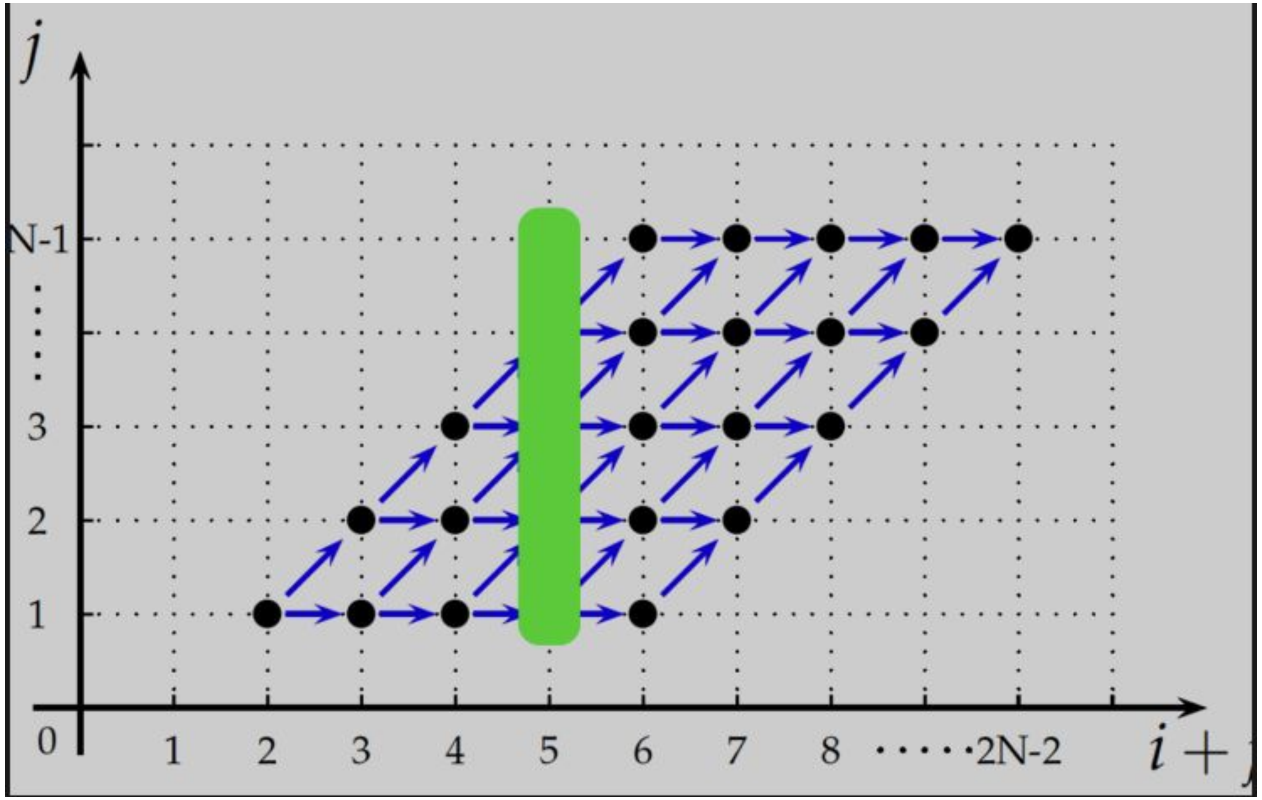
```
for (i = 2; i <= 100; i = i+3)
    Z[i] = 0;
```

```
for (j = 0; j <= 32; j++)
    Z[3*j+2] = 0;
```

Apply Affine Transformation

(i, j) maps to (i + j, j)





Optimized Code (using OpenMP)

```
for (int i = 1; i < N; i++)
    for (int j = 1; j < N; j++)
        A[i, j] = f(A[i - 1][j], A[i][j - 1]);
```

```
for (int c0 = 2; c0 <= 2 * N - 2; c0 += 1)
  #pragma omp parallel for
  for (int c1 = max(1, c0 - N + 1); c1 <= min(N - 1, c0 - 1); c1 += 1)
    A[c1][c0 - c1] = (A[c1 - 1][c0 - c1] + A[c1][c0 - c1 - 1]);
```