The Polyhedral Model

Chirantan Mukherjee

University of Western Ontario

June 20, 2024

Content

What is the Polyhedral Model?

What is the Polyhedral Model?

1. A framework for performing loop transformation.

What is the Polyhedral Model?

1. A framework for performing loop transformation.
2. Loop representation: using polytopes to achieve fine-grain representation of program.

What is the Polyhedral Model?

1. A framework for performing loop transformation.
2. Loop representation: using polytopes to achieve fine-grain representation of program.
3. Loop transformation: transforming loop by doing affine transformation on polytopes.

What is the Polyhedral Model?

1. A framework for performing loop transformation.
2. Loop representation: using polytopes to achieve fine-grain representation of program.
3. Loop transformation: transforming loop by doing affine transformation on polytopes.
4. Dependency test: several mathematical methods for validating transformation on loop polytopes.

What is the Polyhedral Model?

1. A framework for performing loop transformation.
2. Loop representation: using polytopes to achieve fine-grain representation of program.
3. Loop transformation: transforming loop by doing affine transformation on polytopes.
4. Dependency test: several mathematical methods for validating transformation on loop polytopes.
5. Code generation: generate transformed code from loop polytopes.

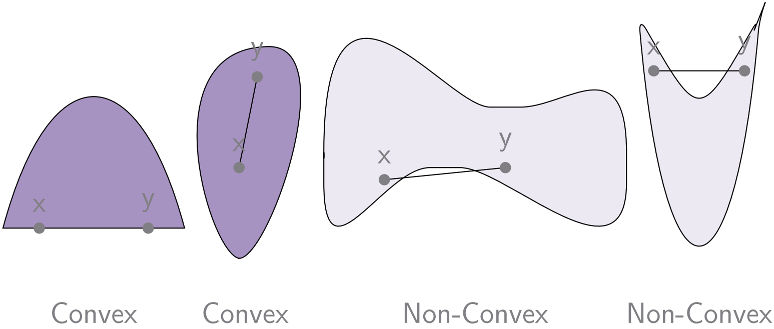
Convexity is the central concept of polyhedral optimization

Convexity is the central concept of polyhedral optimization

Definition

A set S is called convex if the line joining any two points in S is in S, i.e.,

$$\forall x, y \in S, \forall \lambda \in [0,1], \lambda x + (1-\lambda) y \in S.$$



Definition

A polyhedron P is a set which can be expressed as the intersection of finite number of (closed) half spaces, that is $\{\overrightarrow{x} \in \mathbb{R}^n \mid A\overrightarrow{x} \leq \overrightarrow{b}\}$.

Definition

A polyhedron P is a set which can be expressed as the intersection of finite number of (closed) half spaces, that is $\{\overrightarrow{x} \in \mathbb{R}^n \mid A\overrightarrow{x} \leq \overrightarrow{b}\}$.

Definition

A polytope is a bounded polyhedron.

Definition

A polyhedron P is a set which can be expressed as the intersection of finite number of (closed) half spaces, that is $\{\overrightarrow{x} \in \mathbb{R}^n \mid A\overrightarrow{x} \leq \overrightarrow{b}\}$.

Definition

A polytope is a bounded polyhedron.

Definition

A $\mathbb{Z}$-polyhedron is a polyhedron where all its extreme points are integer valued.

In most situation loop counters are integers. So we use $\mathbb{Z}$-polyhedron to represent loop iteration domain.

Modeling Iteration Domains

Modeling Iteration Domains

1. Dimension of Iteration Domain: Decided by loop nesting levels

Modeling Iteration Domains

1. Dimension of Iteration Domain: Decided by loop nesting levels
2. Bounds of Iteration Domain: Decided by loop bounds

Modeling Iteration Domains

1. Dimension of Iteration Domain: Decided by loop nesting levels
2. Bounds of Iteration Domain: Decided by loop bounds

```
for(i = 1; i <= n; i++){
  for(j = 1; j <= n; j++){
      if (i <= n + 2 - j)
          b[j] = b[j] + a[i];
      }
  }
```

Modeling Iteration Domains

1. Dimension of Iteration Domain: Decided by loop nesting levels
2. Bounds of Iteration Domain: Decided by loop bounds

```
for(i = 1; i <= n; i++){
  for(j = 1; j <= n; j++){
      if (i <= n + 2 - j)
          b[j] = b[j] + a[i];
      }
  }
```

Inequalities:

$$
\begin{aligned}
1 \leq i &\leq n \\
1 \leq j &\leq n \\
i &\leq n + 2 - j
\end{aligned}
$$

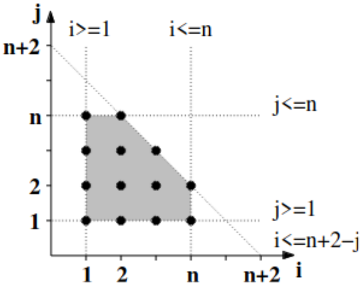
Modeling Iteration Domains

1. Dimension of Iteration Domain: Decided by loop nesting levels
2. Bounds of Iteration Domain: Decided by loop bounds

```
for(i = 1; i <= n; i++){
  for(j = 1; j <= n; j++){
      if (i <= n + 2 - j)
          b[j] = b[j] + a[i];
      }
  }
```

Inequalities:

$$1 \le i \le n$$
$$1 \le j \le n$$
$$i \le n + 2 - j$$



Representing iteration bounds by affine function

Representing iteration bounds by affine function

$$1 \leq i \leq n \qquad \begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \end{pmatrix} \geq \overrightarrow{0}$$

$$1 \leq j \leq n \qquad \begin{pmatrix} 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \end{pmatrix} \geq \overrightarrow{0}$$

$$1 \leq n + 2 - j \leq n \qquad \begin{pmatrix} -1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + (n + 2) \geq 0$$

Representing iteration bounds by affine function

$$1 \le i \le n \qquad \begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \end{pmatrix} \ge \overrightarrow{0}$$

$$1 \le j \le n \qquad \begin{pmatrix} 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \end{pmatrix} \ge \overrightarrow{0}$$

$$1 \le n + 2 - j \le n \qquad \begin{pmatrix} -1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + (n + 2) \ge 0$$

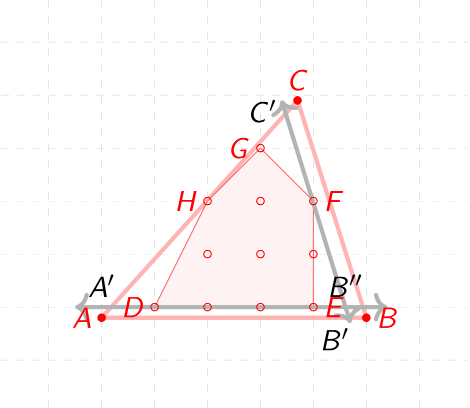
Iteration Domain:

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ n + 2 \end{pmatrix} \ge \overrightarrow{0}$$

Definition

The integer hull P_I of a convex polyhedral set P is the convex hull of integer points of P.

Example



Definition

The dual representation of the polyhedron P can be expressed as a combination of lines L, rays R (forming the polyhedral cone) and vertices V (forming the polytope), that is
$\{x \in \mathbb{R}^n \mid \lambda L + \mu R + \rho V$ such that $\lambda, \mu, \rho \geq 0$ and $\lambda + \mu + \rho = 1\}$.

Definition

The dual representation of the polyhedron P can be expressed as a combination of lines L, rays R (forming the polyhedral cone) and vertices V (forming the polytope), that is $\{x \in \mathbb{R}^n \mid \lambda L + \mu R + \rho V \text{ such that } \lambda, \mu, \rho \geq 0 \text{ and } \lambda + \mu + \rho = 1\}$.

Definition

A face of the polyhedron P is the intersection of P with the supporting hyperplane of P. A face of maximum dimension is called the facet of P.

Definition

The dual representation of the polyhedron P can be expressed as a combination of lines L, rays R (forming the polyhedral cone) and vertices V (forming the polytope), that is $\{x \in \mathbb{R}^n \mid \lambda L + \mu R + \rho V$ such that $\lambda, \mu, \rho \geq 0$ and $\lambda + \mu + \rho = 1\}$.

Definition

A face of the polyhedron P is the intersection of P with the supporting hyperplane of P. A face of maximum dimension is called the facet of P.

Theorem (Fundamental theorem of polyhedral decomposition)

Every polyhedron P can be decomposed into a polytope V and a polyhedral cone L.

Definition

Given $\overrightarrow{m}$ the vector of symbolic parameters, a parametric polyhedron P is defined by $\{\overrightarrow{x} \in \mathbb{R}^n \mid A\overrightarrow{x} \leq B\overrightarrow{m} + \overrightarrow{b}\}$.

One-dimensional array

```
for(int i = 0; i < n; i++){
  for(int j = i + 1; j < n; j ++)
    A[i * n + j] = A[(n * j - n + j - i - 1];
    }
```

One-dimensional array

```
for(int i = 0; i < n; i++){
  for(int j = i + 1; j < n; j ++)
    A[i * n + j] = A[(n * j - n + j - i - 1];
    }
```

1. Can we parallelize the two `for-loops`?

One-dimensional array

```
for(int i = 0; i < n; i++){
  for(int j = i + 1; j < n; j ++)
    A[i * n + j] = A[(n * j - n + j - i - 1];
    }
```

1. Can we parallelize the two `for-loops`?
2. Is there data dependence between two different iterations of the nest?

One-dimensional array

```
for(int i = 0; i < n; i++){
  for(int j = i + 1; j < n; j ++)
    A[i * n + j] = A[(n * j - n + j - i - 1];
    }
```

1. Can we parallelize the two `for-loops`?
2. Is there data dependence between two different iterations of the nest?
3. Are there integer solutions to the following system of linear inequalities?

$$\begin{cases} 0 \leq i_1 < n \\ i_1 + 1 \leq j_1 < n \\ 0 \leq i_2 < n \\ i_2 + 1 \leq j_2 < n \\ i_1 \times n + j_1 = n \times j_2 - n + j_2 - i_2 - 1 \end{cases}$$

Delinearize the array accesses

Linearized one-dimensional array

```
for(int i = 0; i < n; i++)
  for(int j = i + 1; j < n; j ++)
    A[i * n + j] =
        A[(n * j - n + j - i - 1];
```

Delinearize the array accesses

Linearized one-dimensional array

```
for(int i = 0; i < n; i++)
  for(int j = i + 1; j < n; j ++)
    A[i * n + j] =
        A[(n * j - n + j - i - 1];
```

Delinearized multi-dimensional array

```
for(int i = 0; i < n; i++)
  for(int j = i + 1; j < n; j ++)
    B[i][j] = B[j - 1][j - i - 1];
```

Delinearize the array accesses

Linearized one-dimensional array

```
for(int i = 0; i < n; i++)
  for(int j = i + 1; j < n; j ++)
    A[i * n + j] =
        A[(n * j - n + j - i - 1];
```

$$\begin{cases} 0 \leq i_1 < n \\ i_1 + 1 \leq j_1 < n \\ 0 \leq i_2 < n \\ i_2 + 1 \leq j_2 < n \\ i_1 \times n + j_1 = n \times j_2 - n + j_2 - i_2 - 1 \end{cases}$$

Delinearized multi-dimensional array

```
for(int i = 0; i < n; i++)
  for(int j = i + 1; j < n; j ++)
    B[i][j] = B[j - 1][j - i - 1];
```

Delinearize the array accesses

Linearized one-dimensional array

```
for(int i = 0; i < n; i++)
  for(int j = i + 1; j < n; j ++)
    A[i * n + j] =
        A[(n * j - n + j - i - 1];
```

$$\begin{cases} 0 \leq i_1 < n \\ i_1 + 1 \leq j_1 < n \\ 0 \leq i_2 < n \\ i_2 + 1 \leq j_2 < n \\ i_1 \times n + j_1 = n \times j_2 - n + j_2 - i_2 - 1 \end{cases}$$

Delinearized multi-dimensional array

```
for(int i = 0; i < n; i++)
  for(int j = i + 1; j < n; j ++)
    B[i][j] = B[j - 1][j - i - 1];
```

$$\begin{cases} 0 \leq i_1 < n \\ i_1 + 1 \leq j_1 < n \\ 0 \leq i_2 < n \\ i_2 + 1 \leq j_2 < n \\ i_1 = j_2 - 1 \\ j_1 = j_2 - i_2 - 1 \end{cases}$$

Delinearize the array accesses

Linearized one-dimensional array

```
for(int i = 0; i < n; i++)
  for(int j = i + 1; j < n; j ++)
    A[i * n + j] =
        A[(n * j - n + j - i - 1];
```

$$\begin{cases} 0 \le i_1 < n \\ i_1 + 1 \le j_1 < n \\ 0 \le i_2 < n \\ i_2 + 1 \le j_2 < n \\ i_1 \times n + j_1 = n \times j_2 - n + j_2 - i_2 - 1 \end{cases}$$

Delinearized multi-dimensional array

```
for(int i = 0; i < n; i++)
  for(int j = i + 1; j < n; j ++)
    B[i][j] = B[j - 1][j - i - 1];
```

$$\begin{cases} 0 \le i_1 < n \\ i_1 + 1 \le j_1 < n \\ 0 \le i_2 < n \\ i_2 + 1 \le j_2 < n \\ i_1 = j_2 - 1 \\ j_1 = j_2 - i_2 - 1 \end{cases}$$

There is no integer solution, therefore, no dependence.

Bernstein Conditions

Definition

Given two references, there exists a dependence between them if the following conditions are satisfied:

1. they reference the same array (cells)
2. one of this access is a write
3. the two associated statements are executed

Bernstein Conditions

Definition

Given two references, there exists a dependence between them if the following conditions are satisfied:

1. they reference the same array (cells)
2. one of this access is a write
3. the two associated statements are executed

There are three types of dependencies:

1. True dependency (read-after-write), `A = 3, B = A, C = B`

Bernstein Conditions

Definition

Given two references, there exists a dependence between them if the following conditions are satisfied:

1. they reference the same array (cells)
2. one of this access is a write
3. the two associated statements are executed

There are three types of dependencies:

1. True dependency (read-after-write), `A = 3, B = A, C = B`
2. Anti-dependency (write-after-read), `B = 3, A = B + 1, B = 7`

Bernstein Conditions

Definition

Given two references, there exists a dependence between them if the following conditions are satisfied:

1. they reference the same array (cells)
2. one of this access is a write
3. the two associated statements are executed

There are three types of dependencies:

1. True dependency (read-after-write), `A = 3, B = A, C = B`
2. Anti-dependency (write-after-read), `B = 3, A = B + 1, B = 7`
3. Output dependency (write-after-write), `B = 3, B = 7`

Dependence Analysis Methods

1. Compute the transitive closure of the access function
    - transitive closure is not convex in general, and not even computable in many situations

Dependence Analysis Methods

1. Compute the transitive closure of the access function
    - transitive closure is not convex in general, and not even computable in many situations
2. Compute an indicator of the distance between two dependent iterations
    - approximative for non-uniform dependences

Dependence Analysis Methods

1. Compute the transitive closure of the access function
    - transitive closure is not convex in general, and not even computable in many situations
2. Compute an indicator of the distance between two dependent iterations
    - approximative for non-uniform dependences
3. Dependence cone: do the union of dependence relations
    - over-approximative as it requires union and transitive closure to model all dependences in a single cone

Dependence Analysis Methods

1. Compute the transitive closure of the access function
    - transitive closure is not convex in general, and not even computable in many situations
2. Compute an indicator of the distance between two dependent iterations
    - approximative for non-uniform dependences
3. Dependence cone: do the union of dependence relations
    - over-approximative as it requires union and transitive closure to model all dependences in a single cone
4. Dependence polyhedron, list of sets of dependent instances

Dependence Relation

Definition

A statement R is dependent on a statement S, denoted as $R \rightarrow S$ if there exists operations $S(\overrightarrow{x_S})$, $R(\overrightarrow{x_R})$ and a memory location m such that,

Dependence Relation

Definition

A statement R is dependent on a statement S, denoted as $R \to S$ if there exists operations $S(\overrightarrow{x_S})$, $R(\overrightarrow{x_R})$ and a memory location m such that,

1. $S(\overrightarrow{x_S})$ and $R(\overrightarrow{x_R})$ refers to the same memory location m, and atleast one of them writes to that location

Dependence Relation

Definition

A statement R is dependent on a statement S, denoted as $R \rightarrow S$ if there exists operations $S(\overrightarrow{x_S})$, $R(\overrightarrow{x_R})$ and a memory location m such that,

1. $S(\overrightarrow{x_S})$ and $R(\overrightarrow{x_R})$ refers to the same memory location m, and atleast one of them writes to that location
2. $\overrightarrow{x_S}$ and $\overrightarrow{x_R}$ belongs to the iteration domain S and R respectively

Dependence Relation

Definition

A statement R is dependent on a statement S, denoted as $R \rightarrow S$ if there exists operations $S(\overrightarrow{x_S})$, $R(\overrightarrow{x_R})$ and a memory location m such that,

1. $S(\overrightarrow{x_S})$ and $R(\overrightarrow{x_R})$ refers to the same memory location m, and atleast one of them writes to that location
2. $\overrightarrow{x_S}$ and $\overrightarrow{x_R}$ belongs to the iteration domain S and R respectively
3. In the original sequential order $S(\overrightarrow{x_S})$ is executed before $R(\overrightarrow{x_R})$.

Dependence Relation

Definition

A statement R is dependent on a statement S, denoted as $R \rightarrow S$ if there exists operations $S(\overrightarrow{x_S})$, $R(\overrightarrow{x_R})$ and a memory location m such that,

1. $S(\overrightarrow{x_S})$ and $R(\overrightarrow{x_R})$ refers to the same memory location m, and atleast one of them writes to that location
2. $\overrightarrow{x_S}$ and $\overrightarrow{x_R}$ belongs to the iteration domain S and R respectively
3. In the original sequential order $S(\overrightarrow{x_S})$ is executed before $R(\overrightarrow{x_R})$.

Using this we can describe the dependence polyhedra of each dependence relation between two statements. It is a subset of cartesian product of iteration space R and S.

Dependence Polyhedra

In dependence polyhedra every integral point represents a dependence between two instances of the corresponding statements with components:

Dependence Polyhedra

In dependence polyhedra every integral point represents a dependence between two instances of the corresponding statements with components:

1. Same memory location: equality of the subscript functions of a pair of references to the same array, $F_S\overrightarrow{x_S} + a_S = F_R\overrightarrow{x_R} + a_R$.

Dependence Polyhedra

In dependence polyhedra every integral point represents a dependence between two instances of the corresponding statements with components:

1. Same memory location: equality of the subscript functions of a pair of references to the same array, $F_S\overrightarrow{x_S} + a_S = F_R\overrightarrow{x_R} + a_R$.
2. Iteration domains: both S and R iteration domains can be described using affine inequalities: $A_S\overrightarrow{x_S} + c_S \geq 0$ and $A_R\overrightarrow{x_R} + c_R \geq 0$ respectively.

Dependence Polyhedra

In dependence polyhedra every integral point represents a dependence between two instances of the corresponding statements with components:

1. Same memory location: equality of the subscript functions of a pair of references to the same array, $F_S\overrightarrow{x_S} + a_S = F_R\overrightarrow{x_R} + a_R$.
2. Iteration domains: both S and R iteration domains can be described using affine inequalities: $A_S\overrightarrow{x_S} + c_S \geq 0$ and $A_R\overrightarrow{x_R} + c_R \geq 0$ respectively.
3. Precedence order: each case corresponds to a common loop depth, and is called a dependence level.
   For each dependence level l, the precedence constraints are the equality of the loop index variables at depth lesser to l: $x_{R,i} = x_{S,i}$ for $i < l$ and $x_{R,l} > x_{S,l}$ if l is less than the common nesting loop level. Otherwise, there is no additional constraint and dependence exists if S is before R.
   Such constraints can be written using linear inequalities, $P_{l,S}\overrightarrow{x_S} - P_{l,R}\overrightarrow{x_R} + \overrightarrow{b} \geq 0$.

Algorithm 1 A Dependence Polyhedra Construction Algorithm

Require: Initialize reduced dependence graph with one node per statement
Ensure: Dependence polyhedra

```
for all pair $R$, $S$ do
    for all distinct references $f_R$, $f_S$ to the same array do
        if commonLoops($R, S$) $= \emptyset$ then
            minDepth = 0
        else
            minDepth = 1
        end if
        for $l$ = minDepth to |commonLoops| do
            Build $\mathcal{D}_{R,S}$
            if $\mathcal{D}_{R,S} \neq \emptyset$ then
                type = concatenate(type($f_R$), A, type($f_S$)) { WAW, RAW, WAR, RAR}
            end if
            addDegree($R, S, \{l, \mathcal{D}_{R,S}, type\}$)
        end for
    end for
end for
```

Example of Dependence Polyhedron

```
for (i = 0; i <= n; i++){
    for (j = 0; j <= n; j++)
        a[i][j] = a[i+1][j+1]; //S1 }
```

Example of Dependence Polyhedron

```
for (i = 0; i <= n; i++){
    for (j = 0; j <= n; j++)
        a[i][j] = a[i+1][j+1]; //S1 }
```

Iteration Domain:

$$\mathcal{D}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

Example of Dependence Polyhedron

```
for (i = 0; i <= n; i++){
    for (j = 0; j <= n; j++)
        a[i][j] = a[i+1][j+1]; //S1 }
```

Iteration Domain:

$$\mathcal{D}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \overrightarrow{0}$$

Array Reference Function:

$$F_{A}\overrightarrow{x_{S1}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \qquad F'_{A}\overrightarrow{x_{S1}} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

Example of Dependence Polyhedron

```
for (i = 0; i <= n; i++){
    for (j = 0; j <= n; j++)
        a[i][j] = a[i+1][j+1]; //S1 }
```

Precedence Order:

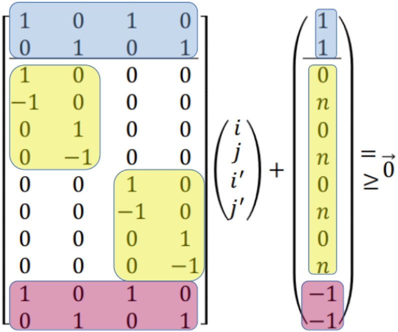
For statement `S1` in two consecutive loop, `i - i' = 1, j - j' = 1,`

$$P_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

To satisfy, $P_{S1}\overrightarrow{x_{S1}} - P_R\overrightarrow{x_R} + \overrightarrow{b} \geq \overrightarrow{0}$, where $\overrightarrow{b} \in [-1, 1]$.

Example of Dependence Polyhedron

```
for (i = 0; i <= n; i++){
    for (j = 0; j <= n; j++)
        a[i][j] = a[i+1][j+1]; //S1 }
```



$$\left[\begin{array}{cccc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{array}\right] \begin{pmatrix} i \\ j \\ i' \\ j' \end{pmatrix} + \left(\begin{array}{c} 1 \\ 1 \\ \hline 0 \\ n \\ 0 \\ n \\ 0 \\ n \\ 0 \\ n \\ -1 \\ -1 \end{array}\right) \begin{smallmatrix} = \\ \geq \end{smallmatrix} \vec{0}$$

🟦 Array Reference Function

🟨 Iteration Domain

🟥 Precedence Order

Source iteration must be executed before target iteration

Source iteration must be executed before target iteration

Definition

Given a statement S, a p-dimensional affine schedule Θ_S is an affine form on the outer loop iterators $\overrightarrow{x_S}$ and the global parameters $\overrightarrow{n}$,

$$\Theta_S(\overrightarrow{x_S}) = T_S \begin{pmatrix} \overrightarrow{x_S} \\ \overrightarrow{n} \\ 1 \end{pmatrix}$$

where $T_S \in \mathbb{R}^{p \times \dim(\overrightarrow{x_S}) + \dim(\overrightarrow{n}) + 1}$.

Source iteration must be executed before target iteration

Definition

Given a statement S, a p-dimensional affine schedule Θ_S is an affine form on the outer loop iterators $\overrightarrow{x_S}$ and the global parameters $\overrightarrow{n}$,

$$\Theta_S(\overrightarrow{x_S}) = T_S \begin{pmatrix} \overrightarrow{x_S} \\ \overrightarrow{n} \\ 1 \end{pmatrix}$$

where $T_S \in \mathbb{R}^{p \times \dim(\overrightarrow{x_S}) + \dim(\overrightarrow{n}) + 1}$.

- If T_S is a vector, Θ_S is called a one-dimensional schedule.
- If T_S is a matrix, Θ_S is called a multi-dimensional schedule.

1. A schedule can assign a time point for every iteration, and a code generator can generate code that will scan them in that specified order.

1. A schedule can assign a time point for every iteration, and a code generator can generate code that will scan them in that specified order.
2. Schedules in our context are assumed to be affine functions, hence affine schedule.

1. A schedule can assign a time point for every iteration, and a code generator can generate code that will scan them in that specified order.
2. Schedules in our context are assumed to be affine functions, hence affine schedule.
3. A one-dimensional schedule, expresses the program as a single sequential loop, possibly enclosing one or more parallel loops.

1. A schedule can assign a time point for every iteration, and a code generator can generate code that will scan them in that specified order.
2. Schedules in our context are assumed to be affine functions, hence affine schedule.
3. A one-dimensional schedule, expresses the program as a single sequential loop, possibly enclosing one or more parallel loops.
4. A multidimensional schedule expresses the program as one or more nested sequential loops, possibly enclosing one or more parallel loops.

1. A schedule can assign a time point for every iteration, and a code generator can generate code that will scan them in that specified order.
2. Schedules in our context are assumed to be affine functions, hence affine schedule.
3. A one-dimensional schedule, expresses the program as a single sequential loop, possibly enclosing one or more parallel loops.
4. A multidimensional schedule expresses the program as one or more nested sequential loops, possibly enclosing one or more parallel loops.
5. Program transformation in the polyhedral model can be specified by a well chosen scheduling function.

1. A schedule can assign a time point for every iteration, and a code generator can generate code that will scan them in that specified order.
2. Schedules in our context are assumed to be affine functions, hence affine schedule.
3. A one-dimensional schedule, expresses the program as a single sequential loop, possibly enclosing one or more parallel loops.
4. A multidimensional schedule expresses the program as one or more nested sequential loops, possibly enclosing one or more parallel loops.
5. Program transformation in the polyhedral model can be specified by a well chosen scheduling function.
6. Dependence graph can be used to represent scheduling constraints between the program operations.

1. A schedule can assign a time point for every iteration, and a code generator can generate code that will scan them in that specified order.
2. Schedules in our context are assumed to be affine functions, hence affine schedule.
3. A one-dimensional schedule, expresses the program as a single sequential loop, possibly enclosing one or more parallel loops.
4. A multidimensional schedule expresses the program as one or more nested sequential loops, possibly enclosing one or more parallel loops.
5. Program transformation in the polyhedral model can be specified by a well chosen scheduling function.
6. Dependence graph can be used to represent scheduling constraints between the program operations.
7. Hyperplanes can be interpreted as schedules.

Example of one-dimensional schedule

Time	Code	Time Stamp
`T = 0`	`x = a + b; //S1`	`T_S1 = 0;`
`T = 1`	`y = a + b; //S2`	`T_S2 = 1;`
`T = 2`	`z = x + y; //S3`	`T_S3 = 2;`

Function `T` returns the logical date of each statement.

Example of multi-dimensional schedule

```
 Time              Code                         Time Stamp
T = 0         x = a + b; //S1                   T_S1 = (0);
T = 1         for (i = 0; i < 2; i ++){
  i = 0             a[i] = x; //S2              T_S2(0) = (1,0);
  i = 1             }                           T_S2(1) = (1,1);
T = 2         z = x + y; //S3                   T_S3 = (2);
```

Function `T` returns the logical date of each statement.
Logical dates may be multi-dimensional:

- Lexicographical Order: $T_{S1} < T_{S2} < T_{S3} \iff (0) < (1, i) < (2)$.

Unlike one-dimensional schedules, it is always possible to build a legal multidimensional schedule for a SCoP.

Theorem ([Fea97])

Every static control program has a multi-dimensional affine schedule.

1. Bernstein conditions are useful to decide if a program transformation is legal if $\begin{cases} \mathcal{W}_a \cap \mathcal{W}_b = \emptyset \\ \mathcal{W}_a \cap \mathcal{R}_b = \emptyset \\ \mathcal{R}_a \cap \mathcal{W}_b = \emptyset \end{cases}$.

1. Bernstein conditions are useful to decide if a program transformation is legal if $\begin{cases} \mathcal{W}_a \cap \mathcal{W}_b = \emptyset \\ \mathcal{W}_a \cap \mathcal{R}_b = \emptyset \\ \mathcal{R}_a \cap \mathcal{W}_b = \emptyset \end{cases}$.
2. A transformation is illegal if a dependence crosses the hyperplane backwards.

1. Bernstein conditions are useful to decide if a program transformation is legal if $\begin{cases} \mathcal{W}_a \cap \mathcal{W}_b = \emptyset \\ \mathcal{W}_a \cap \mathcal{R}_b = \emptyset \\ \mathcal{R}_a \cap \mathcal{W}_b = \emptyset \end{cases}$.
2. A transformation is illegal if a dependence crosses the hyperplane backwards.
3. A dependence going forward between 2 hyperplanes indicates sequentiality.

1. Bernstein conditions are useful to decide if a program transformation is legal if $\begin{cases} \mathcal{W}_a \cap \mathcal{W}_b = \emptyset \\ \mathcal{W}_a \cap \mathcal{R}_b = \emptyset \\ \mathcal{R}_a \cap \mathcal{W}_b = \emptyset \end{cases}$.
2. A transformation is illegal if a dependence crosses the hyperplane backwards.
3. A dependence going forward between 2 hyperplanes indicates sequentiality.
4. No dependence between any point of the hyperplane indicates parallelism.

1. Bernstein conditions are useful to decide if a program transformation is legal if $\begin{cases} \mathcal{W}_a \cap \mathcal{W}_b = \emptyset \\ \mathcal{W}_a \cap \mathcal{R}_b = \emptyset \\ \mathcal{R}_a \cap \mathcal{W}_b = \emptyset \end{cases}$.
2. A transformation is illegal if a dependence crosses the hyperplane backwards.
3. A dependence going forward between 2 hyperplanes indicates sequentiality.
4. No dependence between any point of the hyperplane indicates parallelism.

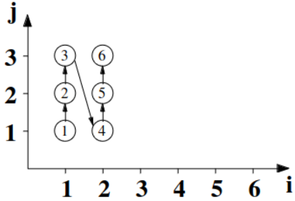
Definition (Precedence condition)

Θ_R and Θ_S are legal schedule for instances R and S respectively if for all $< \overrightarrow{x_R}, \overrightarrow{x_S} > \in \mathcal{D}_{R,S}$ then $\Theta_R(\overrightarrow{x_R}) < \Theta_S(\overrightarrow{x_S})$ holds.

Example of Transformation (Loop Interchange)

Original

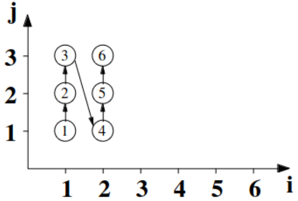
```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



Example of Transformation (Loop Interchange)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

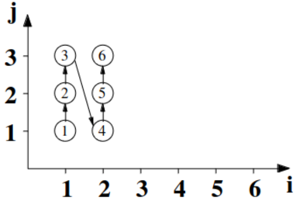


$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \overrightarrow{0}$$

Example of Transformation (Loop Interchange)

Original

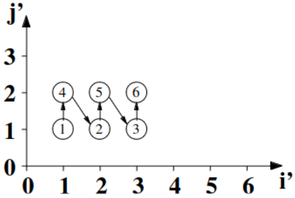
```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

New

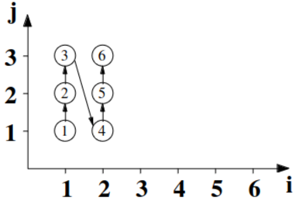
```
for(j = 1; j <= 3; j++){
  for(i = 1; i <= 2; i++)
      b[i][j] = ...; } //S1
```



Example of Transformation (Loop Interchange)

Original

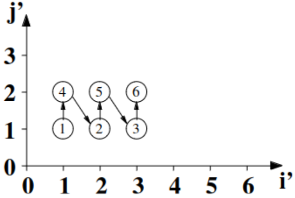
```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \overrightarrow{0}$$

New

```
for(j = 1; j <= 3; j++){
  for(i = 1; i <= 2; i++)
      b[i][j] = ...; } //S1
```

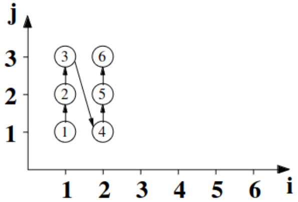


$$\begin{pmatrix} 0 & 1 \\ 0 & -1 \\ 1 & 0 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \overrightarrow{0}$$

Example of Transformation (Loop Interchange)

Original

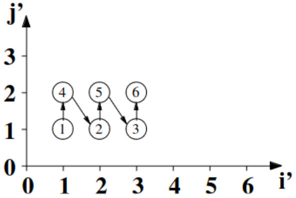
```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

New

```
for(j = 1; j <= 3; j++){
  for(i = 1; i <= 2; i++)
      b[i][j] = ...; } //S1
```



$$\begin{pmatrix} 0 & 1 \\ 0 & -1 \\ 1 & 0 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

Transformation Function: $\begin{pmatrix} i' \\ j' \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + \begin{pmatrix} i \\ j \end{pmatrix} \geq \vec{0}$.

Example of Transformation (Loop Interchange)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

Example of Transformation (Loop Interchange)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

Original Schedule

```
T_S1(i,j) = (i ,j);
```

Example of Transformation (Loop Interchange)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

New

```
for(j = 1; j <= 3; j++){
  for(i = 1; i <= 2; i++)
      b[i][j] = ...; } //S1
```

Original Schedule

```
T_S1(i,j) = (i ,j);
```

Example of Transformation (Loop Interchange)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

New

```
for(j = 1; j <= 3; j++){
  for(i = 1; i <= 2; i++)
      b[i][j] = ...; } //S1
```

Original Schedule

```
T_S1(i,j) = (i ,j);
```

New Schedule

```
T_S1(i,j) = (j ,i);
```

Example of Transformation (Loop Interchange)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

New

```
for(j = 1; j <= 3; j++){
  for(i = 1; i <= 2; i++)
      b[i][j] = ...; } //S1
```

Original Schedule

`T_S1(i,j) = (i ,j);`

New Schedule

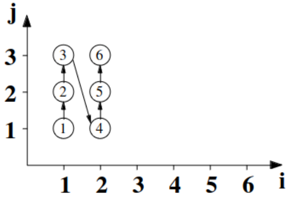
`T_S1(i,j) = (j ,i);`

$$T_{S1}(i,j) = \underbrace{\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}}_{\text{Transformation}} \underbrace{\begin{pmatrix} i \\ j \end{pmatrix}}_{\text{Iteration vector}} = \underbrace{\begin{pmatrix} j \\ i \end{pmatrix}}_{\text{New Schedule}} .$$

Example of Transformation (Loop Reversal)

Original

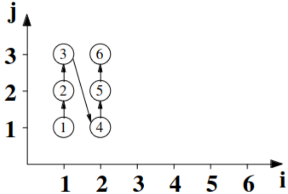
```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



Example of Transformation (Loop Reversal)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

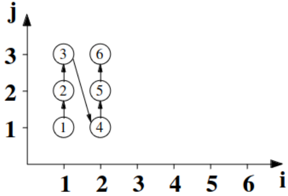


$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

Example of Transformation (Loop Reversal)

Original

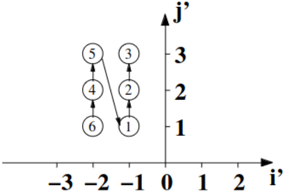
```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

New

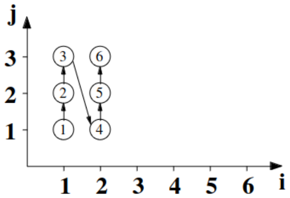
```
for(i = -1; i >= -2; i--){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



Example of Transformation (Loop Reversal)

Original

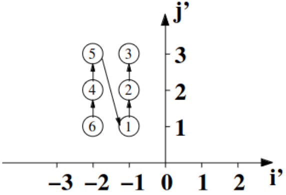
```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \overrightarrow{0}$$

New

```
for(i = -1; i >= -2; i--){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

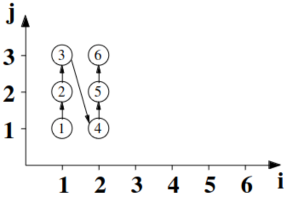


$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \overrightarrow{0}$$

Example of Transformation (Loop Reversal)

Original

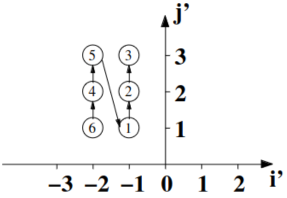
```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

New

```
for(i = -1; i >= -2; i--){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```



$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

Transformation Function: $\begin{pmatrix} i' \\ j' \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} i \\ j \end{pmatrix} \geq \vec{0}$.

Example of Transformation (Loop Reversal)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

Example of Transformation (Loop Reversal)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

Original Schedule

```
T_S1(i,j) = (i ,j);
```

Example of Transformation (Loop Reversal)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

New

```
for(i = -1; i >= -2; i--){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

Original Schedule

```
T_S1(i,j) = (i ,j);
```

Example of Transformation (Loop Reversal)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

New

```
for(i = -1; i >= -2; i--){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

Original Schedule

```
T_S1(i,j) = (i ,j);
```

New Schedule

```
T_S1(i,j) = (-i ,j);
```

Example of Transformation (Loop Reversal)

Original

```
for(i = 1; i <= 2; i++){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

New

```
for(i = -1; i >= -2; i--){
  for(j = 1; j <= 3; j++)
      b[i][j] = ...; } //S1
```

Original Schedule

`T_S1(i,j) = (i ,j);`

New Schedule

`T_S1(i,j) = (-i ,j);`

$$T_{S1}(i,j) = \underbrace{\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}}_{\text{Transformation}} \underbrace{\begin{pmatrix} i \\ j \end{pmatrix}}_{\text{Iteration vector}} = \underbrace{\begin{pmatrix} -i \\ j \end{pmatrix}}_{\text{New Schedule}} .$$

Loop Tiling

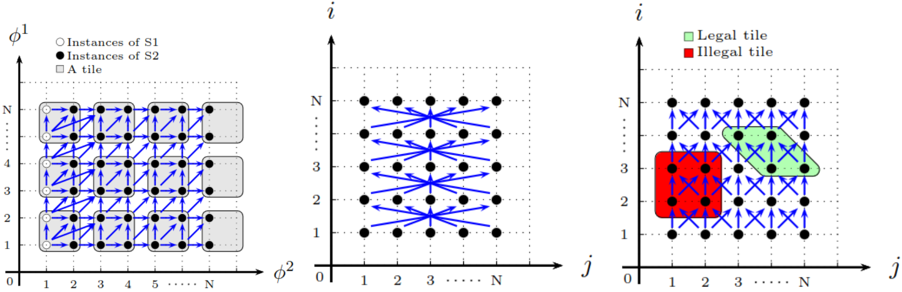
Loop tiling [IT88, WL91, Xue00] is a key transformation in optimizing for parallelism and data locality.

Loop Tiling

Loop tiling [IT88, WL91, Xue00] is a key transformation in optimizing for parallelism and data locality.

Theorem

Two one-dimensional schedules are valid tiling hyperplanes if and only if they satisfy the precedence conditions.



Example of Tiling: Transpose matrix vector multiply [BRS10]

Original code

```
for(i = 0; i < N; i++){
  P: x[i] = 0;
  for(j = 0; j < N; j++)
      Q: x[i] += a[j][i] * y[j];
  }
```

Example of Tiling: Transpose matrix vector multiply [BRS10]

Original code

```
for(i = 0; i < N; i++){
  P: x[i] = 0;
  for(j = 0; j < N; j++)
      Q: x[i] += a[j][i] * y[j];
  }
```

Iteration Space

$$\mathcal{D}_Q^{\text{orig}} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0, \ \mathcal{D}_Q^{\text{tiled}} \begin{pmatrix} it \\ jt \\ i \\ j \\ N \\ 1 \end{pmatrix} \geq 0$$

Example of Tiling: Transpose matrix vector multiply [BRS10]

Original code

```
for(i = 0; i < N; i++){
  P: x[i] = 0;
  for(j = 0; j < N; j++)
      Q: x[i] += a[j][i] * y[j];
  }
```

Iteration Space

$$\mathcal{D}_Q^{\text{orig}} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0, \quad \mathcal{D}_Q^{\text{tiled}} \begin{pmatrix} it \\ jt \\ i \\ j \\ N \\ 1 \end{pmatrix} \geq 0$$

Tiled code

```
for(it = 0; it <= floord(N - 1, 32); it++){
  for(jt = 0; jt <= floord(N - 1, 32); jt++)
    if(jt == 0){
      for(i = max(32 * it, 0);
        i <= min(32 * it + 31, N - 1); i++){
          P: x[i] = 0;
          Q: x[i] = x[i] + a[0][i] * y[0];
          }
      }
    for(i = max(32 * it, 0);
        i <= min(32 * it + 31, N - 1); i++){
      for(j = max(32 * jt, 1);
        j <= min(32 * jt + 31, N - 1); j++){
        Q: x[i] = x[i] + a[j][i] * y[j];
      }
    }
  }
```

Drawbacks of the Polyhedral Model

Drawbacks of the Polyhedral Model

1. Compile-time efficiency, most optimization problems in the polyhedral model are modeled as Integer Linear Programming, which is NP-hard.

Drawbacks of the Polyhedral Model

1. Compile-time efficiency, most optimization problems in the polyhedral model are modeled as Integer Linear Programming, which is NP-hard.
2. Building polyhedrons in compile time is also memory consuming.

References I

- Louis-Noël Pouchet.
  Polyhedral Compilation Foundations, 2010.
  https://www.cs.colostate.edu/~pouchet/lectures/888.11.lect1.html#lect1
- Cédric Bastoul.
  Improving Data Locality in Static Control Programs.
  Ph.D. Thesis, 2004.
- Fangzhou Jiao.
  An Overview to Polyhedral Model, 2010.
  https://homes.luddy.indiana.edu/achauhan/Teaching/B629/2010-Fall/StudentPresns/PolyhedralModelOverview.pdf

References II

- Paul Feautrier.
  Some efficient solutions to the affine scheduling problem: Part II Multidimensional time
  nternational Journal of Parallel Programming, 1997.
- Francois Irigoin and Rémi Triolet.
  Supernode partitioning.
  ACM SIGPLAN Principles of Programming Languages, 1988.
- Michael E. Wolf and Monica S. Lam.
  A data locality optimizing algorithm.
  ACM SIGPLAN symposium on Programming Languages Design and Implementation, 1991.

References III

Jingling Xue.
Loop tiling for parallelism.
Kluwer Academic Publishers, 2000.

Muthu Manikandan Baskaran, Jeyakumar Ramanujam and Ponnuswamy Sadayappan.
Automatic C-to-CUDA Code Generation for Affine Programs.
International Conference on Compiler Construction, 2010.