Overview of Today’s Lecture

Outline:

▶ Follow-up on \( \mathbb{Z} \)-polyhedra
▶ Data dependence
  ▶ Dependence representations
  ▶ Various analysis
  ▶ Data dependence algorithm in Candl/PoCC/Pluto

Mathematical concepts:

▶ Affine mapping
▶ Image, preimage by an affine mapping
▶ Cartesian product of polyhedra
Affine Function and Lattices (Reminder)

**Definition (Affine function)**

A function \( f : \mathbb{K}^m \to \mathbb{K}^n \) is affine if there exists a vector \( \vec{b} \in \mathbb{K}^n \) and a matrix \( A \in \mathbb{K}^{m \times n} \) such that:

\[
\forall \vec{x} \in \mathbb{K}^m, \quad f(\vec{x}) = A\vec{x} + \vec{b}
\]

**Definition (Lattice)**

A subset \( L \) in \( \mathbb{Q}^n \) is a lattice if is generated by integral combination of finitely many vectors: \( a_1, a_2, \ldots, a_n \) (\( a_i \in \mathbb{Q}^n \)).

\[
L = L(a_1, \ldots, a_n) = \{ \lambda_1 a_1 + \ldots + \lambda_n a_n \mid \lambda_i \in \mathbb{Z} \}
\]

If the \( a_i \) vectors have integral coordinates, \( L \) is an integer lattice.

Example: \( L_1 = \{ 2i + 1, 3j + 5 \mid i, j \in \mathbb{Z} \} \) is a lattice.
Image and Preimage

**Definition (Image)**

The image of a polyhedron $P \in \mathbb{Z}^n$ by an affine function $f : \mathbb{Z}^n \rightarrow \mathbb{Z}^m$ is a $\mathbb{Z}$-polyhedron $P'$:

$$P' = \{ f(\bar{x}) \in \mathbb{Z}^m \mid \bar{x} \in P \}$$

**Definition (Preimage)**

The preimage of a polyhedron $P \in \mathbb{Z}^n$ by an affine function $f : \mathbb{Z}^n \rightarrow \mathbb{Z}^m$ is a $\mathbb{Z}$-polyhedron $P'$:

$$P' = \{ \bar{x} \in \mathbb{Z}^n \mid f(\bar{x}) \in P \}$$

We have $\text{Image}(f^{-1}, P) = \text{Preimage}(f, P)$ if $f$ is invertible.
Relation Between Image, Preimage and $\mathbb{Z}$-polyhedra

- The image of a $\mathbb{Z}$-polyhedron by an unimodular function is a $\mathbb{Z}$-polyhedron.

- The preimage of a $\mathbb{Z}$-polyhedron by an affine function is a $\mathbb{Z}$-polyhedron.

- The image of a polyhedron by an affine invertible function is a $\mathbb{Z}$-polyhedron.

- The preimage of a $\mathbb{Z}$-polyhedron by an affine function is a $\mathbb{Z}$-polyhedron.

- The image by a non-invertible function is **not** a $\mathbb{Z}$-polyhedron.
Returning to the Example

Exercise: Compute the set of cells of A accessed

Example

```
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j)
        A[2i + 3][4j] = i * j;
```

- $\mathcal{D}_S: \{i, j \mid 0 \leq i < N, i \leq j < N\}$
- Function: $f_A : \{2i + 3, 4j \mid i, j \in \mathbb{Z}\}$
- $\text{Image}(f_A, \mathcal{D}_S)$ is the set of cells of A accessed (a $\mathbb{Z}$-polyhedron):
  - Polyhedron: $Q : \{i, j \mid 3 \leq i < 2N + 2, 0 \leq j < 4N\}$
  - Lattice: $L : \{2i + 3, 4j \mid i, j \in \mathbb{Z}\}$
Data Dependence

Definition (Bernstein conditions)

Given two references, there exists a dependence between them if the three following conditions hold:

- they reference the same array (cell)
- one of this access is a write
- the two associated statements are executed

Three categories of dependences:

- RAW (Read-After-Write, aka flow): first reference writes, second reads
- WAR (Write-After-Read, aka anti): first reference reads, second writes
- WAW (Write-After-Write, aka output): both references writes

Another kind: RAR (Read-After-Read dependences), used for locality/reuse computations
Purpose of Dependence Analysis

- Not all program transformations preserve the semantics
- Semantics is preserved if the dependence are preserved

- In standard frameworks, it means reordering statements
  - Statements containing dependent references should not be executed in a different order
  - Granularity: usually a reference to an array

- In the polyhedral framework, it means reordering statement instances
  - Statement instances containing dependent references should not be executed in a different order
  - Granularity: a reference to an array cell
## Illustrations

### Example

```c
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        A[i][j] = A[i + N][j];
```

### Example

```c
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        A[i][j] = i * j;
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        B[i][j] = A[i][j];
```
An Intuitive Dependence Test Algorithm

Idea: compute the sets associated to the Bernstein conditions

Given two references $a$ and $b$ to the same array:

- Compute $W_a': Image(f_a, D_a)$ if $a$ is a write, $\emptyset$ otherwise
- Compute $R_a: Image(f_a, D_a)$ if $a$ is a read, $\emptyset$ otherwise
- Compute $W_b': Image(f_b, D_b)$ if $b$ is a write, $\emptyset$ otherwise
- Compute $R_b: Image(f_b, D_b)$ if $b$ is a read, $\emptyset$ otherwise

- If $W_a \cap R_b \neq \emptyset \lor W_a \cap W_b \neq \emptyset \lor R_a \cap W_b \neq \emptyset$ then $a \delta b$
A (Naive) Dependence Test Algorithm

Exercise: Write a dependence test algorithm for a program
A (Naive) Dependence Test Algorithm

Exercise: Write a dependence test algorithm for a program

- Create the Data Dependence Graph, with one node per statement
- For all pairs \( a, b \) of distinct references, do
  - If \( a \) and \( b \) reference the same array, do
    (i) Compute \( W_a, R_a, W_b, R_v \)
    (ii) If \( W_a \cap R_b \neq \emptyset \lor W_a \cap W_b \neq \emptyset \lor R_a \cap W_b \neq \emptyset \) then
      Add an edge between the statement with the reference \( a \) and the statement with the reference \( b \) in the DDG
Connection with Statement Instances

Objective: get the set of instances which are in dependence, not only statements

Exercise: Compute this set, from $W_a$ and $R_b$ (RAW dependence)
Connection with Statement Instances

Objective: get the set of instances which are in dependence, not only statements

Exercise: Compute this set, from $W_a$ and $R_b$ (RAW dependence)

- Idea: $Preimage(f_a, W_a \cap R_b)$ gives the set of instances
- Must generalize to multiple references, we lose convexity (unions)
Some Terminology on Dependence Relations

We categorize the dependence relation in three kinds:

- **Uniform dependences:** the distance between two dependent iterations is a constant
  - ex: $i \rightarrow i + 1$
  - ex: $i, j \rightarrow i + 1, j + 1$

- **Non-uniform dependences:** the distance between two dependent iterations varies along the execution
  - ex: $i \rightarrow i + j$
  - ex: $i \rightarrow 2i$

- **Parametric dependences:** at least a parameter is involved in the dependence relation
  - ex: $i \rightarrow i + N$
  - ex: $i + N \rightarrow j + M$
Data Dependence Analysis

Objective: compute the set of statement instances which are in dependence

Some of the several possible approaches:

▶ Compute the transitive closure of the access function
  ▶ Problems: transitive closure is not convex in general, and not even computable in many situations

▶ Compute an indicator of the distance between two dependent iterations
  ▶ Problems: approximative for non-uniform dependences

▶ dependence cone: do the union of dependence relations
  ▶ Problems: over-approximative as it requires union and transitive closure to model all dependences in a single cone

▶ Retained solution: dependence polyhedron, list of sets of dependent instances
Dependence Polyhedron [1/3]

Principle: model all pairs of instances in dependence

Definition (Dependence of statement instances)
A statement $S$ depends on a statement $R$ (written $R \rightarrow S$) if there exists an operation $S(\vec{x}_S)$ and $R(\vec{x}_R)$ and a memory location $m$ such that:

1. $S(\vec{x}_S)$ and $R(\vec{x}_R)$ refer to the same memory location $m$, and at least one of them writes to that location,
2. $x_S$ and $x_R$ belongs to the iteration domain of $R$ and $S$,
3. in the original sequential order, $S(\vec{x}_S)$ is executed before $R(\vec{x}_R)$. 
Dependence Polyhedra:

Dependence Polyhedron [2/3]

1. **Same memory location**: equality of the subscript functions of a pair of references to the same array: \( F_S \vec{x}_S + a_S = F_R \vec{x}_R + a_R \).

2. **Iteration domains**: both \( S \) and \( R \) iteration domains can be described using affine inequalities, respectively \( A_S \vec{x}_S + c_S \geq 0 \) and \( A_R \vec{x}_R + c_R \geq 0 \).

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 the dependence only exists if \( S \) is textually before \( R \).

   Such constraints can be written using linear inequalities: \( P_{l,S} \vec{x}_S - P_{l,R} \vec{x}_R + b \geq 0 \).
**Dependence Polyhedron [3/3]**

The dependence polyhedron for $R \rightarrow S$ at a given level $l$ and for a given pair of references $f_R, f_S$ is described as [Feautrier/Bastoul]:

$$
\mathcal{D}_{R,S,f_R,f_S,l} : D \left( \begin{array}{c} \vec{x}_S \\ \vec{x}_R \end{array} \right) + d = \begin{bmatrix} F_S & -F_R \\ A_S & 0 \\ 0 & A_R \\ PS & -P_R \end{bmatrix} \left( \begin{array}{c} \vec{x}_S \\ \vec{x}_R \end{array} \right) + \begin{bmatrix} a_S - a_R \\ c_S \\ c_R \\ b \end{bmatrix} \geq 0
$$

A few properties:

- We can always build the dep polyhedron for a given pair of affine array accesses (it is convex)
- It is exact, if the iteration domain and the access functions are also exact
- It is over-approximated if the iteration domain or the access function is an approximation
Polyhedral Representation of Programs

Static Control Parts

- Loops have affine control only (over-approximation otherwise)
Polyhedral Representation of Programs

Static Control Parts

- Loops have affine control only (over-approximation otherwise)
- Iteration domain: represented as integer polyhedra

```
for (i=1; i<=n; ++i)
  for (j=1; j<=n; ++j)
    if (i<=n-j+2)
      s[i] = ...
```
Polyhedral Representation of Programs

Static Control Parts

- Loops have affine control only (over-approximation otherwise)
- Iteration domain: represented as integer polyhedra
- Memory accesses: static references, represented as affine functions of $\vec{x}_S$ and $\vec{p}$

```c
for (i=0; i<n; ++i) {
    s[i] = 0;
    for (j=0; j<n; ++j)
        s[i] = s[i] + a[i][j]*x[j];
}
```

$$f_s(\vec{x}_S) = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} \vec{x}_S \ 1 \end{pmatrix}$$

$$f_a(\vec{x}_S) = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} \vec{x}_S \ 1 \end{pmatrix}$$

$$f_x(\vec{x}_S) = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \begin{pmatrix} \vec{x}_S \ 1 \end{pmatrix}$$
Polyhedral Representation of Programs

Static Control Parts

- Loops have affine control only (over-approximation otherwise)
- Iteration domain: represented as integer polyhedra
- Memory accesses: static references, represented as affine functions of $\vec{x}_S$ and $\vec{p}$
- Data dependence between S1 and S2: a subset of the Cartesian product of $\mathcal{D}_{S1}$ and $\mathcal{D}_{S2}$ (exact analysis)

```c
for (i=1; i<=3; ++i) {
  s[i] = 0;
  for (j=1; j<=3; ++j)
    s[i] = s[i] + 1;
}
```

\[
\mathcal{D}_{S1} \delta \mathcal{D}_{S2} := \begin{bmatrix}
1 & -1 & 0 & 0 \\
1 & 0 & 0 & -1 \\
-1 & 0 & 0 & 3 \\
0 & 1 & 0 & -1 \\
0 & -1 & 0 & 3 \\
0 & 0 & 1 & -1 \\
0 & 0 & -1 & 3
\end{bmatrix} \begin{pmatrix} i_{S1} \\ i_{S2} \\ s_{S2} \\ 1 \end{pmatrix} \geq 0
\]

$S1$ iterations $\rightarrow$ $i$

$S2$ iterations $\rightarrow$ $j$
A Dependence Polyhedra Construction Algorithm

1. Initialize reduced dependence graph with one node per program statement
2. For each pairs of statements $R, S$ do
3.     For each pairs of distinct references $f_R, f_S$ to the same array, do
4.         If $R, S$ does not share any loop, $min\_depth = 0$ else $min\_depth = 1$
5.     For each level $l$ from $min\_depth$ to $nb\_common\_loops$, do
6.         Build the dependence polyhedron $\mathcal{D}_{R,S,f_R,f_S,l}$
7.         If $\mathcal{D}_{R,S,f_R,f_S,l} \neq \emptyset$ then
8.             If $f_R$ is a write and $f_S$ is a read, $type = RAW$
9.             If $f_R$ is a write and $f_S$ is a write, $type = WAW$
10.            If $f_R$ is a read and $f_S$ is a write, $type = WAR$
11.            If $f_R$ is a read and $f_S$ is a read, $type = RAR$
12.            $add\_edge(R, S, \{l, \mathcal{D}_{R,S,f_R,f_S,l}, type\})$
The PolyLib Matrix Format

All our tools use this notation (Candl, Pluto, Cloog, PIPLib, etc.)

Given \( D_{R,S} \):

\[
\begin{bmatrix}
1 & -1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & -1 & 1 & 0
\end{bmatrix}
\cdot
\begin{pmatrix}
i_R \\
i_S \\
j_S \\
n \\
1
\end{pmatrix}
\geq 0
\]

It is written:

\[
\begin{bmatrix}
0 & 1 & -1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
1 & -1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & -1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & -1 & 1 & 0
\end{bmatrix}
\cdot
\begin{pmatrix}
i_R \\
i_S \\
j_S \\
n \\
1
\end{pmatrix}
\geq 0
\]

On the first column, 0 stands for = 0, 1 for ≥ 0
Practicing

Exercise: Give all dependence polyhedra

Example

```plaintext
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        A[i][j] = A[i + 1][j + 1];
```

Example

```plaintext
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        A[i][j] = i * j;
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        B[i][j] = A[i][j];
```
Connection with Parallelism

- A dependence is loop-carried if 2 iterations of this loop access the same array cell
- If no such dependence exists, the loop is parallel
- A parallel loop can be transformed arbitrarily
- OpenMP free parallelization or vectorization is possible

Example

```cpp
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        C[i][j] = 0;

for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        for (k = 0; k < N; ++k)
            C[i][j] += A[i][k] * B[k][j];
```
Practicing Parallelism

Exercise: Give all parallel loops

Example

```c
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        A[i][j] = i * j;
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        B[i][j] = A[i][j];
```

Example

```c
for (t = 0; t < L; ++t)
    for (j = 1; j < N - 1; ++j)
```
Visual Intuition

- Synchronization-free parallelism means "slices" in the dependence polyhedron

- The shape of the independent slices gives an intuition of which loop of the program are parallel

- Transforming the code may expose (more) parallelism possibilities

- Be careful of multiple references: must do the union of the dependence relations
Other Techniques for Dependence Analysis

- Scalars are a particular case of array \( (c = c[0]) \)
- **Privatization**: a variable is written before it is read (use-def chains)
- **Renaming**: two privatized variables having the same name
- **Expansion**: remove dependences by increasing the array dimension
- Transform program to Single-Assignment-Form (SSA)

- Scalar privatization / renaming / expansion is implemented in Candl
- Maximal static expansion is efficient but difficult!
Hands On!

Demo of Clan + Candl
A First Intuition About Scheduling

Intuition: the source iteration must be executed before the target iteration

Definition (Precedence condition)

Given $\Theta_R$ a schedule for the instances of $R$, $\Theta_S$ a schedule for the instances of $S$. Then, $\forall \langle \bar{x}_R, \bar{x}_S \rangle \in D_{R,S}$:

$$\Theta_R(\bar{x}_R) \prec \Theta_S(\bar{x}_S)$$

Next week: scheduling and semantics preservation (Farkas method, convex space of legal schedules, tiling hyperplane method)