Array Dependence Analysis

COMP 621 Special Topics
By
Nurudeen Lameed
nlamee@cs.mcgill.ca

Outline

- Introduction
- Basic concepts
  - Affine functions
  - Iteration Space
  - Data Space
  - Affine Array-Index functions
  - Matrix formulation
- Array Data-Dependence Analysis
- Questions?

Introduction – Why?

- The traditional data flow model is inadequate for parallelization. For instance, it does not distinguish between different executions of the same statement in a loop.
- Array dependence analysis enables optimization for parallelism in programs involving arrays.

Affine functions

- A function of one or more variables, $i_1, i_2, \ldots, i_n$ is affine, if it can be expressed as a sum of a constant, plus constant multiples of the variables. i.e.

$$f = c_0 + \sum_{i=1}^{n} c_i x_i$$

- Array subscript expressions are usually affine functions involving loop induction variables.

Affine functions(2)

- Sometimes, affine functions are called linear functions. Examples:
  - a[i] affine
  - a[i+j -1] affine
  - a[i*j] non-linear, not affine
  - a[2*i+1, i*j] linear, non-linear; not affine
  - a[b[i] + 1] ?
    - Non linear (indexed subscript), not affine

Iteration Space(1)

- Iteration space is the set of iterations, whose ID's are given by the values held by the loop index variables.

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

The iteration space for the loop is the set

{2, 5, 8, 11, … , 98} – the set contains the value of the loop index *i* at each iteration of the loop.

Iteration Space(2)

- The iteration space can be normalized. For example, the loop in the previous slide can be written as

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

In general, $i^n = (i - lowerBound) / i_{step}$

Iteration Space(3)

- How about nested loops?

```
for (i = 3; i <= 7;  i++)
    for (j = 6; j >= 2; j = j – 2 )
        Z[i, j] = Z[i, j+2] + 1
```

The iteration space is given by the set of vectors: {[3,6], [3,4], [3,2], [4,6], [4,4], [4,2], [5,6], [5,4], [5,2], [6,6], [6,4], [6,2], [7,6], [7,4], [7,2]}

Q1: Rewrite the loop using normalized iteration vectors?

Dependence types

- We consider three kinds of dependence.
  - Flow dependence (true dependence)
    - A variable assigned in one statement is used subsequently in another statement.
  - Anti-dependence
    - A variable is used in one statement and reassigned in a subsequently executed statement.
  - Output dependence
    - A variable is assigned in one statement and subsequently reassigned in another statement.

Dependence Graph

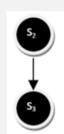
- Graph can be drawn to show data dependence between statements within a loop.

```
S1:    for (i = 2; i<= 5; ++i){
S2:        X[i] = Y[i] + Z[i]
S3:        A[i] = X[i-1] + 1
       }
```

i=2 →(1) i=3 →(1) i=4 →(1) i=5

S_2: X[2] X[3] X[4] X[5]

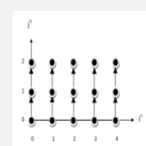
S_3: X[1] X[2] X[3] X[4]



Iteration space dependence Graph

```
for (i = 3; i <= 7;  i++)
    for (j = 6; j >= 2; j = j – 2 )
        Z[i, j] = Z[i, j+2] + 1
```

- Iteration space dependence graph (normalized)



Data Space

- Array declaration specifies the data space.
  - float Z[50];

  declares an array whose elements are indexed by 0, 1 , .. 49.
- Note that iteration space is different from data space

Matrix formulation (Iteration space)

- We can represent iterations in a *d*-deep loop mathematically as

  $\{ \mathbf{i} \text{ in } Z^d \mid \mathbf{Bi} + \mathbf{b} \geq \mathbf{0} \}$

  Where Z = set of integers; B is a *d* x *d* integer matrix; *b* is an integer vector of length *d,* and 0 is a vector of *d* 0's

```
for ( i = 0; i <= 5; i++)        ➔   i ≥ 0, i ≤ 5;
    for ( j = i; j <= 7; j++)    ➔   j ≥ i, j ≤ 7;
        Z[j, i] = 0;                 // express this in the form
                                     // ci.i + cj.j + c ≥ 0
```

Matrix formulation(2)

- $i \geq 0$ ⟺ $1.i + 0.j + 0 \geq 0$
- $i \leq 5$ ⟺ $-1.i + 0.j + 5 \geq 0$
- $j \geq i$ ⟺ $-1.i + 1.j + 0 \geq 0$
- $j \leq 7$ ⟺ $0.i - 1.j + 7 \geq 0$

Thus,

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

Affine Array Access

- Affine functions provide a mapping from the iteration space to data space; they make it easier to identify iterations that map to the same data.
- An array access is affine if:
  - the bounds of the loop and the index of each dimension of the array are affine expressions of loop variables and symbolic constants.
- Affine access can also be represented as matrix-vector calculation.

Matrix formulation(Array Access)

- Like iteration space, array access can be represented as **Fi** + **f**; **F** and **f** represent the functions of the loop-index variables.
- Formally, an array access, *A*= <**F**, **f**, **B**, **b**>; where **i** = index variable vector; *A* maps **i** within the bounds

  $\mathbf{Bi} + \mathbf{b} \geq \mathbf{0}$

  to the array element location

  $\mathbf{Fi} + \mathbf{f}$

Matrix formulation (Array Access-2)

Access	Affine Expression
X[i, j]	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
X[6 – j*2]	$\begin{bmatrix} 0 & 2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 6 \end{bmatrix}$
X[1,5]	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 5 \end{bmatrix}$
X[0, i-5, 2*i + j]	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ -5 \\ 0 \end{bmatrix}$

Array Dependence Analysis(1)

- Consider two static accesses A in a *d*-deep loop nest and A' in a *d'*-deep loop nest respectively defined as

  A= <**F**, **f**, **B**, **b**> and A' = <**F'**, **f'**, **B'**, **b'**>

A and A' are data dependent if

- **Bi** ≥ **0** ; **B'i'** ≥ **0** and
- **Fi** + **f** = **F'i'** + **f'**
- (and **i** ≠ **i'** for dependencies between instances of the same static access)

Array Dependence Analysis(2)

```
for (i = 1; i < 10; i++) {
   X[i] = X[i-1]
}
```

To find all the data dependences, we check if

1. X[i-1] and X[i] refer to the same location;
2. different instances of X[i] refer to the same location.

For 1, we solve for i and i’ in

$1 \le i \le 10$, $1 \le i' \le 10$ and $i - 1 = i'$

Array Dependence Analysis(3)

For 2, we solve for i and i’ in

$1 \le i \le 10$, $1 \le i' \le 10$, $i = i'$ and $i \ne i'$ (between different dynamic accesses)

There is a dependence since there exist integer solutions to 1. e.g. (i=2, i’=1), (i=3,i’=2). 9 solutions exist.

There is no dependences among different instances of X[i] because 2 has no solutions!

Array Dependence Analysis(4)

- Array data dependence basically requires finding integer solutions to a system(often refers to as dependence system) consisting of equalities and inequalities.
- Equalities are derived from array accesses.
- Inequalities from the loop bounds.
- It is an integer linear programming problem.
- ILP is an NP-Complete problem.
- Several Heuristics have been developed.

Question 2

- Q2: Rewrite this loop using normalized iteration space?

```
for (i = 2; i <= 50;  i= i+5)
   Z[i] = 0;
```

Solution Q2

- The iteration space for the loop is the set

{ 2, 7, 12, … , 47} – the set contains the value of the loop index *i* at different iteration of the loop. The normalized version of the loop is

```
for (iⁿ = 0; iⁿ <= 9;  iⁿ++)
      Z[5*iⁿ + 2] = 0;
```

Question 3

For the following loop

```
for (i = 1;  i <= 6;  i= i++)
   X[i] = X[6-i];
```

indicate all the

1. Flow dependences (True dependences)
2. Anti-dependences
3. Output dependences

Solution Q3

i=1	i=2	i=3	i=4	i=5	i=6
X[1] = X[5]	X[2] = X[4]	X[3] = X[3]	X[4] = X[2]	X[5] = X[1]	X[6] = X[0]

- Flow dependence:
  - { (X[1], 1 → 5), (X[2], 2 → 4) }
- Anti-dependence:
  - { (X[5], 1 → 5), (X[4], 2 → 4), (X[3], 3 → 3) }
- Output-dependence:
  - {}

References

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, 2007. “Compilers: Principles, Techniques, and Tools”. (2nd Edition). Addison-Wesley, CA.
- Michael Wolfe, 1989. “Optimizing Super-compilers for Supercomputers”.MIT Press.
- Michael Wolfe, 1996. “High Performance Compilers for Parallel Computing“. Addison-Wesley, CA.