CS 526
Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526
DEPENDENCE ANALYSIS

The slides adapted from Vikram Adve and David Padua
Motivation: Vectorization

```
for (i=0; i<n; i++)
c[i] = a[i] + b[i];
```

*Slide from Maria Garzaran and David Padua*
Motivation: Vectorization

```c
void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

*Slide from Maria Garzaran and David Padua

** AVX code from Intel’s Software & Services Group talk
Data Dependence

A data dependence from statement $S_1$ to statement $S_2$ exists if

1. there is a feasible execution path from $S_1$ to $S_2$, and
2. an instance of $S_1$ references the same memory location as an instance of $S_2$ in some execution of the program, and
3. at least one of the references is a store.
Kinds of Data Dependence

Direct Dependence

\[ X = \ldots \]
\[ \ldots = X + \ldots \]

Anti-dependence

\[ \ldots = X \]
\[ X = \ldots \]

Output Dependence

\[ X = \ldots \]
\[ X = \ldots \]
A dependence graph is a graph with:

- Each node represents a statement, and
- Each directed edge from S1 to S2, if there is a data dependence between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).

- S1 is known as a source node
- S2 is known as a sink node
Kinds of Data Dependence

**Direct Dependence**

\[ S_1: X = \ldots \]
\[ S_2: \ldots = X + \ldots \]

**Anti-dependence**

\[ S_1: \ldots = X \]
\[ S_2: X = \ldots \]

**Output Dependence**

\[ S_1: X = \ldots \]
\[ S_2: X = \ldots \]
(Repeat) A dependence graph is a graph with:

• one node per statement, and

• a directed edge from S1 to S2 if there is a data dependence between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).

For loops: dependence graph is a summary of unrolled dependencies for different iterations

• Some (detailed) information may be lost
Dependence in Loops

def X(), Y(), a(), i;
do i = 1 to N
S1: X(i) = a(i) + 2
S2: Y(i) = X(i) + 1
enddo
Dependence in Loops

```python
def X(), Y(), a(), i;
do i = 1 to N
S1: X(i+1) = a(i) + 2
S2: Y(i) = X(i) + 1
enddo
```

S1
S2
Dependence in Loops

```python
def X(), Y(), a(), i;
do i = 2 to N
S1: X(i) = a(i) + 2
S2: Y(i) = X(i-1) + 1
enddo
```
dependence in loops

```python
def X(), Y(), a(), i;
do i = 1 to N
S1: X(i) = a(i) + 2
S2: Y(i) = X(i+1) + 1
enddo
```
Dependence in Loops

```python
def X(), Y(), a(), t, i;
doi = 1 to N
S1:    t = a(i) + 2
S2:    Y(i) = t + 1
enddo
```
Reordering Transformation

Reordering Transformation: merely changes the order of execution of computations in a program, without adding or deleting executions of any computations.

Preserving Dependence: a reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink statements of the dependence.
Reordering Transformation

**Definition.** Legal Transformation preserves the meaning of that program, i.e., **all externally visible outputs** are identical to the original program, and in identical order.

- We consider two programs equivalent (i.e., the transformation preserving the program meaning) if on the same inputs both the original and transformed programs, after being executed, produce the same outputs.

**Theorem.** A **reordering** transformation that preserves all data dependences in a program is a **legal** transformation.
Proof of Theorem 1
(by contradiction)

Loop-free program:
Let $S_1, \ldots S_n$ be the original execution order, and $i_1 \ldots i_n$ a permutation of
the statement indices in the reordered program. If we reorder code
without violating dependencies, but the output changed, then at least one
statement would need to produce a different output. Since the statement
is the same as in the original program, then its error must have propagated
from the inputs. But in that case, there must have been a previous
statement that violated (flow, anti, or output) dependence. Contradiction!

Loops:
The previous argument directly extends, by unrolling (and the index
of the loop iteration represents the part of the permutation index).

Conditionals:
If there are conditional statements, the theorem must include
control dependences in addition to data dependences.
(We will come back to this point next week)
Dependence in Loop Nests

**Goal:** Supporting transformations of a given loop nest (Assume perfect loop nest here)

**Canonical Loop Nest:** A loop nest is in canonical form if both lower bound and step of each loop are +1.

Do $i_1 = 1$ to $n_1$
   
   Do $i_2 = 1$ to $n_2$
     
     ... 
     
     Do $i_k = 1$ to $n_k$
       
       Statements
     
     Enddo
   
   ... 
   
   Enddo
   
   Enddo

**Rectangular Loop Nest:** The value of $n_1$ to $n_k$ does not change during the execution
Dependence in Loop Nests

**Iteration space**

The *iteration space* of the loop nest is a set of points in a k-dimensional integer space (i.e., a polyhedron):

\[
L = \{ [i_1, \ldots, i_n] : \\
1 \leq i_1 \leq n_1 \land \ldots \land \\
1 \leq i_k \leq n_k \}
\]

Each element \([i_1, \ldots, i_n]\) is an *iteration vector*.

```
do i1 = 1 to n1
  do i2 = 1 to n2
    . . .
    do ik = 1 to nk
       statements
    enddo
  enddo
. . .
enddo
```
Dependence in Loop Nests

Lexicographic Order: for iteration vectors 
\([i_1, \ldots, i_n]\) and \([j_1, \ldots, j_n]\):

\([i_1, \ldots, i_n] < [j_1, \ldots, j_n]\) iff there is a subscript \(k\), such that \(i_1 = j_1, \ldots, i_{k-1} = j_{k-1}\) but \(i_k < j_k\)

If \(I = [i_1, \ldots, i_n] < [j_1, \ldots, j_n] = J\) we say that the iteration \(I\) preceds the iteration \(J\)
Dependence in Loop Nests

\[
\begin{align*}
\text{do } & v_1 = 1 \text{ to } n_1 \\
& \quad \text{do } v_2 = 1 \text{ to } n_2 \\
& \quad \quad \quad \ldots \\
& \quad \quad \quad \text{do } v_k = 1 \text{ to } n_k \\
& \quad \quad \quad \quad \quad X(f_1(I), \ldots, f_k(I)) = \ldots \\
& \quad \quad \quad \quad \quad \ldots = X(g_1(J), \ldots, g_k(J)) \\
& \quad \quad \quad \text{enddo} \\
& \quad \quad \quad \ldots \\
& \quad \text{enddo} \\
& \text{enddo}
\end{align*}
\]

\[I = (v_1', v_2', \ldots, v_k')\]
\[J = (v_1'', v_2'', \ldots, v_k'')\]
Direct (Flow) Dependence in Loops

We say that $S_1 \rightarrow S_2$ ($S_1 \delta S_2$) iff there exist $I, J \in L$ and $I \leq J$ where

1. There is a feasible path from instance $I$ of statement $S_1$ to instance $J$ of statement $S_2$,

   $S_1$: $X(f_1(I), \ldots, f_k(I)) = \ldots$

   $\ldots$

   $S_2$: $\ldots = X(g_1(J), \ldots, g_k(J))$

2. $f_s(I) = g_s(J), \forall 1 \leq s \leq k$

The statement $S_1$ in iteration $I$ writes and $S_2$ in iteration $J$ reads from the same memory location $M$.
Antidependence in Loops

We say that \( S_1 \leftrightarrow S_2 \) (\( S_1 \delta^{-1} S_2 \)) iff there exist \( I, J \in L \) and \( I \leq J \):

1. There is a feasible path from instance \( I \) of statement \( S_1 \) to instance \( J \) of statement \( S_2 \),

   \[ \begin{align*}
   S_1 : \quad & \ldots = X(f_1(I), \ldots, f_k(I)) \\
   & \ldots \\
   S_2 : \quad & X(g_1(J), \ldots, g_k(J)) = \ldots
   \end{align*} \]

2. \( f_s(I) = g_s(J), \forall \ 1 \leq s \leq k \)

   The statement \( S_1 \) in iteration \( I \) reads and \( S_2 \) in iteration \( J \) writes to the same memory location \( M \)
Output Dependence in Loops

We say that $S_1 \Rightarrow S_2$ ($S_1 \delta^0 S_2$) iff there exist $I, J \in L$ and $I \leq J$:

1. There is a feasible path from instance $I$ of statement $S_1$ to instance $J$ of statement $S_2$,

   $S_1$: $X(f_1(I), \ldots, f_k(I)) = \ldots$

   $\ldots$

   $S_2$: $X(g_1(J), \ldots, g_k(J)) = \ldots$

2. $f_s(I) = g_s(J), \forall \ 1 \leq s \leq k$

   The statement $S_1$ in iteration $I$ and $S_2$ in iteration $J$ both write to the same memory location $M$
Dependence Distance

**Dependence Distance:** If there is a dependence from statement S1 on iteration \( \vec{i} \) and statement S2 on iteration \( \vec{j} \) then the corresponding dependence distance vector is

\[
d_{\vec{i},\vec{j}} = [j_1 - i_1, \ldots, j_k - i_k]
\]

*Note: Computing distance vectors is harder than testing dependence*
Convergence Distance

Direction Vector: For a distance vector of the form \(d_{i,j} = [\vec{j}_1 - \vec{i}_1, \ldots, \vec{j}_k - \vec{i}_k]\) the corresponding direction vector is \(\delta_{i,j} = [\delta_1, \ldots, \delta_k]\), where

\[
\delta_i = \begin{cases} 
- , & \text{if } \vec{j}_1 - \vec{i}_1 < 0 \\
+ , & \text{if } \vec{j}_1 - \vec{i}_1 > 0 \\
= , & \text{if } \vec{j}_1 - \vec{i}_1 = 0 \\
* , & \text{if } \text{sign} <,>,= 
\end{cases}
\]

Note: \(I < J\) iff the leftmost non-’=’ entry in \(\delta(I, J)\) is ’+’.

- We use the property of lexicographical ordering
Loop-Carried Dependence

Statement S2 has a loop carried dependence on statement S1 iff S1 references location M on iteration I, S2 references M on iteration J and \( d(I,J) > 0 \).

\[
\text{do } i = 1 \text{ to } N \\
A(i+1) = B(i) \\
B(i+1) = A(i) \\
\text{enddo}
\]

**Level** of loop-carried dependence is the leftmost non-“=” sign in the direction vector

- Forward dependence: S1 appears before S2 in the loop body
- Backward dependence: S2 appears before S1 in the loop body
Loop-Independent Dependence

Statement S2 has a loop independent dependence on statement S1 iff S1 references location M on iteration I, S2 references M on iteration J and \( d(I,J) = 0 \).

```
    do i = 1 to N
        A(i+1) = B(i)
        B(i+1) = A(i+1)
    enddo
```

Determines the order in which the code is executed within the nest of loops (compare to loop carried dependence!)
Dependence in Loops

\[
do \ i = 2 \ \text{to} \ N \\
S1: \quad X(i-1) = X(i) + 1 \\
\]
\[
enddo \\
\]

\[
do \ i = 1 \ \text{to} \ N \\
S1: \quad X(i+1) = X(i) + 1 \\
\]
\[
enddo \\
\]
Dependence in Loops

do j = 1 to 10
    do i = 1 to 100
        S1: \( X(i,j) = W(i,j) + 1 \)
        S2: \( Y(i,j) = X(100-i,j) \)
    enddo
enddo
Dependence in Loops

for i = 1 to N
    for j = 1 to M
        for k = 1 to 100
            S1: \( X(i,j,k+1) = X(i,j,k) + 1 \)
        endfor
    endfor
endfor

Direction Vector: =,=,+  
Level: 3
Loop-Carried Dependence

**Recall:** Statement $S_2$ has a loop carried dependence on statement $S_1$ iff $S_1$ references location $M$ on iteration $I$, $S_2$ references $M$ on iteration $J$ and $d(I,J) > 0$.

So, in the direction vector for any dependence, the leftmost non-’=’ entry must be ’+’ (if any non-’=’ entry is present).

*Equivalently:* the distance vector $d(I,J) \geq 0$. 
Transformations and Direction Vectors

**Theorem:** Consider a transformation $T$ on a loop nest that does not reorder statements within a loop body.

- Only changes how the program iterates the loops.

Such a transformation is legal if, after applying the corresponding transformation to the direction vectors of each dependence, none of them have a leftmost non-`=' entry that is `-' . Equivalently, distance cannot be $d<0$.

- **Equivalently:** none of the dependences have had the order of their source and sink reversed.
Transformations and Direction Vectors

Theorem: Any transformation that does not change the order of loops, and does not reorder the iterations of the level-k loop preserves all level-k dependences in that loop.

Theorem: Any transformation that reorders the iterations of the level-k loop and makes no other changes is legal if the loop carries no dependences.

For discussion, see Allen and Kennedy book.
Dependence Testing

Dependence testing requires finding a solution to
\{ f_s(I) = g_s(J), \forall 1 \leq s \leq n \}
under the inequality constraints \( I, J \in L \)

```
doi1 = L_1 to U_1
do i2 = L_2 to U_2
  
  do ik = L_k to U_k
    statements
  enddo
  
endo```

**Complexity:** undecidable in general

- Indirection arrays (e.g. \( X[Y[i]] \))
- Indirection arrays may only be known at runtime, without a specific application knowledge
- General alias analysis
- Non-linear subscript expressions
Dependence Testing

Assume linear subscript expressions, e.g., each $f_s$ and $g_s$ is
$$c_0 + c_1 i_1 + \ldots + c_n i_n,$$
where $i_1 \ldots i_n$ are loop index variables and $c$’s are constants.

So we now have a system of equations
$$a_{l0} + a_{l1} i_1 + \ldots + a_{ln} i_n = b_{l0} + b_{l1} j_1 + \ldots + b_{ln} j_n$$
$$\ldots$$
$$a_{k0} + a_{k1} i_1 + \ldots + a_{kn} i_n = b_{k0} + b_{k1} j_1 + \ldots + b_{kn} j_n$$

And for all $I: L_1 \leq i_1 \leq U_1 \ldots L_n \leq i_n \leq U_n$ and same for $J$

Instance of integer programming
$\Rightarrow$ NP-complete in general
Simplifications

Two major simplifications in practice:

- Subscript expressions are usually simple: most often $i_k$ or $a_1 i_k + a_0$
- Be **conservative**: Check if a dependence *may exist*. 
Simplifications

**ZIV, SIV, MIV** A subscript expression containing zero, single, or multiple index variable respectively:

**Separable Subscripts** : A subscript position is said to be **separable** if the index variables used in that subscript position are not used in any other subscript position.
E.g., A[i+1, j, k] and A[i, j, k]

**Coupled Subscripts** : Two subscript positions are said to be coupled if the same index variable is used in both positions.
E.g., A[i+1, i, k] and A[i, j+i, k]
Exact Solutions for SIV

A pair of subscripts with index variable $i_j$ are **Strong SIV** if the subscript expressions are the form $a_{i_j} + b_1$ and $a_{i_j} + b_2$

- The loop iterates between one and $n_j$.

Dependence exists *iff* either of these hold:

1. $a = 0$ and $b_1 = b_2$, or
2. $|d_j| \leq n_j - 1$, where $d_j = (b_1 - b_2)/a$

*Assumes:* $n_j$, $a$, $b_1$, $b_2$ are known
Exact Solutions for Weak SIV

The set of subscripts with index variable $i_j$ are **Weak SIV** if the subscripts are of the form $a_1 i_j + b_1$ and $a_2 i_j + b_2$

Each such subscript position $j$ gives an equation of the form:

$$a_1 i_j = a_2 i_j + b_2 - b_1$$

Approach for each index variable $i_j$:
1. Solve up to $r$ simultaneous equations in 2 unknowns.
2. Check if solutions satisfy 2 inequalities from the previous slide
Exact Solutions for Weak SIV

Special case: one of $a_1$ or $a_2$ is zero: \textbf{Weak-Zero SIV} (solution is similar to strong SIV)

**General problem:** Find if $a_1i_1 + a_0 = b_1i_2 + b_0$

*(Lemma) An extended GCD property:* For any pair of values $(x, y)$, the Euclidian GCD algorithm can also compute a triplet $(g, n_x, n_y)$ such that

$$g = n_x x + n_y y = \gcd(x, y)$$
Exact Solutions for Weak SIV

**Theorem.** Let \((g, n_a, n_b)\) be such a triplet for pair \((a_1, -b_1)\).

Let \(x_k\) and \(y_k\) be given by:

\[
\begin{align*}
x_k &= n_a \left( \frac{b_0 - a_0}{g} \right) + k \frac{b_1}{g} \\
y_k &= n_b \left( \frac{b_0 - a_0}{g} \right) + k \frac{a_1}{g}
\end{align*}
\]

Then \((x_k, y_k)\) is a solution of \(a_1 i_1 + a_0 = b_1 i_2 + b_0\) for an integral value of \(k\).

Furthermore, for any solution \((x, y)\) there is a \(k\) such that \(x = x_k\) and \(y = y_k\).

**Solution strategy:**

1. Compute \(x_0, y_0\) using the above equations
2. Then find all values of \(k\) for which \(x_0 + k \frac{b_1}{g}\) falls within loop bounds, and similarly for \(y_k\).
3. For dependence to exist, the solution \((x_k, y_k)\) must be within the region bounded by loop bounds.
GCD Test

Simplifications
1. ignore loop bounds!
2. only test if a solution is possible (GCD property)
3. test each subscript position separately

GCD Property for Single Variable
Let \( f(i) = a_1 i + a_0 \) and \( g(i) = b_1 i + b_0 \)
\( f(i_1) = g(i_2) \Rightarrow a_1 i_1 + a_0 = b_1 i_2 + b_0. \)

**GCD Property:** If there is a solution to the previous equation, then \( g = \gcd(a_1, b_1) \) divides \( a_0 - b_0. \)

**Proof:** Let \( a_1 = n_1 g, b_1 = m_1 g. \) Then \( g \times (n_1 i_1 - m_1 i_2) = a_0 - b_0, \) and the term in parenthesis must be an integer.
GCD Test for Multiple Indices

Let \( f(I) = a_k i_k + \ldots + a_0 \) and
\( g(I) = b_k i_k + \ldots + b_0. \)

**GCD Property:** If there is a solution to the equation
\[ a_k i_k + \ldots + a_0 = b_k i_{k2} + \ldots + b_0, \]
then
\[ g = \gcd(a_1, \ldots, a_k, b_1, \ldots, b_k) \] divides \( (a_0 - b_0). \)

More tests: E.g., Banerjee test, Lamport test, Delta test...
Solving Complicated Indices

E.g.  \[ A[x+2y-1, 2y, z, w+z, v, 1] \].

Simplify the problem by identifying common special cases:

1. Separate subscript positions into coupled groups
2. Label each subscript as ZIV, SIV, or MIV
3. For each separable subscript, apply appropriate test (ZIV, SIV, or MIV). Yields direction vectors.
4. For each coupled group, apply a coupled subscript test; e.g., GCD test or Delta test or …
5. If no test yields independence, a dependence exists.
6. Concatenate direction vectors from different groups
6. [10 points]: We studied several tests for independence (ZIV, SIV, MIV, GCD). Which test would you use to test for a possible dependence in the following loops? Apply the test of your choice and report if there are dependences. Assume that the array boundaries are correct. (Use the space to the right for work).

- for (i=0; i<100; i++)
  for (j=0; j<100; i++)
    b[i]=b[i-1]+a[j];

- for (i=0; i<n; i++)
  for (j=0; j<n; i++)
    b[3*i-2] = b[2*i+5]+a[j];

- for (i=0; i<n; i++)
  for (j=0; j<n; i++)
    b[6*i+2*j+2] = b[2*i+4*j+4]+a[i];
DEPENDENCE TRANSFORMS

(Preview)
Reordering Transformation

Definition. Legal Transformation preserves the meaning of that program, i.e., all externally visible outputs are identical to the original program, and in identical order.

• We consider two programs equivalent (i.e., the transformation preserving the program meaning) if on the same inputs both the original and transformed programs, after being executed, produce the same outputs.

Theorem. A reordering transformation that preserves all data dependences in a program is a legal transformation.

• See Lecture 6 for an argument why.
Motivation

Memory hierarchy optimizations
Goal 1: Improving reuse of data values within loop nest
Goal 2: Exploit reuse to reduce cache, TLB misses

Tiling
Goal 1: Exploit temporal reuse when data size > cache size
Goal 2: In parallel loops, reduce synchronization overhead

Software Prefetching
Goal: Prefetch predictable accesses k iterations ahead

Software Pipelining
Goal: Extract ILP from multiple consecutive iterations

Automatic parallelization Also, auto-vectorization
Goal 1: Enhance parallelism
Goal 2: Convert scalar loop to explicitly parallel
Goal 3: Improve performance of parallel code
Loop Interchange

**Informal Definition:** Change nesting order of loops in a **perfect loop nest**, with no other changes.

```plaintext
do i=2, N
   do j=2, M-1
   enddo
endo
do j=2, M-1
   do i=2, N
   enddo
endo
```
Uses of Loop Interchange

1. Move independent loop innermost
2. Move independent loop outermost
3. Make accesses stride-1 in memory
4. Loop tiling (combine with strip-mining)
5. Unroll-and-jam (combine with unrolling)
Loop Reversal

Informal Definition: Reverse the order of execution of the iterations of a loop

\[
\begin{align*}
doi=2 \text{ to } N & \\
doj=2 \text{ to } M-1 & \\
dok=1 \text{ to } L & \\
enddo & \\
enddo & \\
enddo&
\end{align*}
\]
Loop Distribution

Informal Definition: Convert a loop nest containing two or more statements into two or more distinct loop nests so that each statement appears in only a single resulting loop nest.

```plaintext
do i=2 to N
S1:   A[i] = B[i] + C[i]
S2:   D[i] = A[i] * 2.0
  enddo
```

```plaintext
do i=2 to N
S1:   A[i] = B[i] + C[i]
S2:   D[i] = A[i] * 2.0
S2:   D[i] = A[i] * 2.0
  enddo
```
**Loop Fusion**

**Informal Definition:** Merge two or more distinct (perhaps non-adjacent) loops with identical loop bounds into a single loop.

\[
\text{do } i=1 \text{ to } N \\
\quad A[i] = i \times i \\
\text{enddo}
\]

\[
\text{do } i=1 \text{ to } N \\
\quad B[i] = A[i] + 1 \\
\text{enddo}
\]
Loop Alignment

**Informal Definition:** Eliminate a carried dependence by increasing the number of iterations and executing statements on different subsets of the iterations

*(Always safe)*

```plaintext

i = 1
D[i+1] = A[i] * 2

do i=2 to N-1
    A[i] = B[i] + C[i]
    D[i+1] = A[i] * 2.0
enddo

i = N
A[i] = B[i] + C[i]
```

```plaintext

do i=2 to N
    A[i] = B[i] + C[i]
    D[i+1] = A[i] * 2.0
enddo
```
Scalar Replacement

**Informal Definition:** Replace an array reference with a scalar temporary. (Use dependences to locate consistent re-use patterns)

```plaintext
do i = 1 to n
do j = 2 to n
  x(j,i) = a(i) + x(j-1,i) + b(j,i)
endo
dodo
```

```plaintext
do i = 1 to n
t1 = a(i);
do j = 2 to n
  x(j,i) = t1 + x(j-1,i) + b(j,i)
endo
dodo
```