Contents of Lecture 10

- Introduction to Data Dependence Analysis
- The GCD Test
- The Fourier-Motzkin Test
There are **data dependencies** in the following code:

- **S1**: $x = a + b$
- **S2**: $y = x + 1$
- **S3**: $x = b \times c$

The value written to $x$ in **S1** is read in **S2**.

This is called a **true dependence** and is written $S_1 \delta^t S_2$.

In a true data dependence between two statements both statements access the same memory location, and the first statements writes a value which the other statements reads.
Data dependencies can be at several different levels, including:
- Instructions
- Statements
- Loop iterations
- Functions
- Threads

In this course we will use dependencies at the instruction and loop levels.

Parallelizing compilers usually find parallelism between different loop iterations of a loop.

Instruction scheduling finds parallelism between different instructions in a basic block or loop iterations close to each other.
Different Types of Data Dependencies

- When we write ”instruction” below our sentence is also valid for the other levels (statement, loop etc).
- Below instruction $I_1$ always executes before instruction $I_2$.
- Recall, in a **true dependence**, written $I_1 \delta^t I_2$, $I_1$ produces a value consumed by $I_2$.
- In an **anti dependence**, written $I_1 \delta^a I_2$, $I_1$ reads a memory location later overwritten by $I_2$.
- In an **output dependence**, written $I_1 \delta^o I_2$, $I_1$ writes a memory location later overwritten by $I_2$.
- In an **input dependence**, written $I_1 \delta^i I_2$, both $I_1$ and $I_2$ read the same memory location.
- The first three types of dependencies create partial orderings among all instructions, which parallelizing compilers should exploit by ordering instructions to improve performance.
Let us classify all dependencies in the code:

S1: \( x = a + b; \)
S2: \( y = x + 1; \)
S3: \( x = b \times c; \)

- The is a true dependence from S1 to S2 due to \( x \).
- The is an anti dependence from S2 to S3 due to \( x \).
- The is an output dependence from S1 to S3 due to \( x \).
- The is an input dependence from S1 to S3 due to \( b \).
In the loop

for (i = 3; i < 100; i += 1)
    a[i] = a[i-3] + x;

There is a true dependence from iteration \( i \) to iteration \( i + 3 \).

Iteration \( i = 3 \) writes to \( a_3 \) which is read in iteration \( i = 6 \).

A loop level true dependence means one iteration writes to a memory location which a later reads.

Typically this means one iteration \( l \) writes to an array element \( a_x \) which a later iteration reads.
**Perfect Loop Nests**

- **A perfect loop nest** $L$ is a nest of $m$ nested `for` loops $L_1, L_2, ... L_m$ such that the body of $L_i$, $i < m$, consists of $L_{i+1}$ and the body of $L_m$ consists of a sequence of assignment statements.

- For $1 < r \leq m$ $p_r$ and $q_r$ are linear functions of $l_1, ..., l_{r-1}$.

```
for (l_1 = p_1; l_1 <= q_1; l_1+ = 1) {
    for (l_2 = p_2; l_2 <= q_2; l_2+ = 1) {
        ...
        for (l_m = p_m; l_m <= q_m; l_m+ = 1) {
            h(l_1, l_2, ..., l_m);
        }
    }
}
```
All assignments, except to the loop index variables, are in the innermost loop.

There may any number of assignment statements in the innermost loop.

```c
for (i = 0; i < 100; i += 1) {
    for (j = 3 + i; j < 2 * i + 10; j += 1) {
        for (k = i - j; k < j - i; k += 1) {
            a[i][j][k] += b[k][j][i];
        }
    }
}
```
The lower bound for $l_1$ is $p_{10} \leq l_1$.

The lower bound for $l_2$ is

$$
\begin{align*}
    l_2 & \geq p_{20} + p_{21} l_1 \\
    p_{20} & \leq l_2 - p_{21} l_1 \\
    p_{20} & \leq -p_{21} l_1 + l_2
\end{align*}
$$

(1)

The lower bound for $l_3$ is

$$
\begin{align*}
    l_3 & \geq p_{30} + p_{31} l_1 + p_{32} l_2 \\
    p_{30} & \leq l_3 - p_{31} l_1 - p_{32} l_2 \\
    p_{30} & \leq -p_{31} l_1 - p_{32} l_2 + l_3
\end{align*}
$$

(2)

and so forth. We represent this on matrix form as $\mathbf{p}_0 \leq \mathbf{I} \mathbf{p}$, or... see next page.
Loop Bounds on Matrix Form

\[ P = \begin{pmatrix}
1 & -p_{21} & -p_{31} & \cdots & -p_{m1} \\
0 & 1 & -p_{32} & \cdots & -p_{m2} \\
0 & 0 & 1 & \cdots & -p_{m3} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & 1
\end{pmatrix} \quad \text{and} \quad p_0 = (p_{10}, p_{20}, \ldots, p_{m0}). \]

Similarly, the upper bounds are represented as \( \mathbf{IQ} \leq q_0 \).

The loop bounds, thus, are represented by the system:

\[
\begin{align*}
p_0 & \leq \mathbf{IP} \\
\mathbf{IQ} & \leq q_0
\end{align*}
\]
The assignment to $c_{ij}$ before the innermost loop makes it a non-perfect loop nest.

Sometimes non-perfect loop nest can be split up, or **distributed**, into perfect loop nests.

See next slides.

```c
for (i = 0; i < 100; i += 1) {
    for (j = 0; j < 100; j += 1) {
        c[i][j] = 0;
        for (k = 0; k < 100; k += 1) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```
Result of loop distribution.

for (i = 0; i < 100; i += 1)
  for (j = 0; j < 100; j += 1)
    c[i][j] = 0;
for (i = 0; i < 100; i += 1)
  for (j = 0; j < 100; j += 1)
    for (k = 0; k < 100; k += 1)
      c[i][j] += a[i][k] * b[k][j];
Some Terminology

- The index vector \( \mathbf{l} = (l_1, l_2, \ldots, l_m) \) is the vector of index variables.
- The index values of \( \mathbf{L} \) are the values of \((l_1, l_2, \ldots, l_m)\).
- The index space of \( \mathbf{L} \) is the subspace of \( \mathbb{Z}^m \) consisting of all the index values.
- An **affine array reference** is an array reference in which all subscripts are linear functions of the loop index variables.
Data dependence analysis is normally restricted to affine array references.

In practice, however, subscripts often contain *symbolic constants* as shown below which is test s171 in the C version of the Argonne Test Suite for Vectorizing Compilers.

There is no dependence between the iterations in this test.

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

```c
scanf("%d", &x);

for (i = 3; i < 100; i += 1) {
    S1: a[i] = a[x] + 1;
    S2: b[i] = b[c[i-1]] + 2;
    S3: d[i] = d[i * i] + 3;
}
```

Few compilers, if any, attempt to determine whether the code above has data dependencies.

While $S_3$ is not difficult, almost all parallelizing compilers focus on index expressions which are linear functions of the loop variables.

Some compilers try to do runtime dependence testing.
Let $X$ be an $n$-dimensional array. Then an affine reference has the form:

$$X[a_{11}i_1 + a_{21}i_2 ... a_{m1}i_m + a_{01}] ... [a_{1n}i_1 + a_{2n}i_2 ... a_{mn}i_m + a_{0n}]$$

This is conveniently represented as a matrix and a vector $X[IA + a_0]$, where

$$A = \begin{pmatrix}
  a_{11} & a_{12} & \ldots & a_{1n} \\
  a_{21} & a_{22} & \ldots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m1} & a_{m2} & \ldots & a_{mn}
\end{pmatrix}$$

and

$$a_0 = (a_{10}, a_{20}, \ldots, a_{n0}).$$

We will refer to $A$ and $a_0$ as the **coefficient matrix** and the **constant term**, respectively.
The Data Dependence Equation

- For two references $X[iA + a_0]$ and $X[iB + b_0]$ to refer to the same array element there must be two index values, $i$ and $j$ such that $iA + a_0 = jB + b_0$ which we can write as $iA - jB = b_0 - a_0$.

- This system of Diophantine’s equations has $n$ (the dimension of the array $X$) scalar equations and $2m$ variables, where $m$ is the nesting depth of the loop.

- It can also be written in the following form:

$$\begin{pmatrix} i; j \end{pmatrix} \begin{pmatrix} A \\ -B \end{pmatrix} = b_0 - a_0. \quad (3)$$

- We solve the system of linear Diophantine equations in (3) using a method presented shortly.
for (i = 0; i < 100; i += 1)
  for (j = 2*i + 4; j < i + 40; j += 1)
    a[2i-3j-1][2i+j-3] = f(a[-3i+4j+1][-i+2j+7]);

- The above loop nest has the following two array reference representations:

  \[
  A = \begin{pmatrix} 2 & 2 \\ -3 & 1 \end{pmatrix} \quad \text{and} \quad a_0 = (-1, -3).
  \]

  \[
  B = \begin{pmatrix} -3 & -1 \\ 4 & 2 \end{pmatrix} \quad \text{and} \quad b_0 = (1, 7).
  \]
Let $\prec_\ell$ be a relation in $\mathbb{Z}^m$ such that $i \prec j$ if $i_1 = j_1$, $i_2 = j_2$, ..., $i_{\ell-1} = j_{\ell-1}$, and $i_{\ell} < j_{\ell}$.

For example: $(1, 3, 4) \prec_3 (1, 3, 9)$.

The lexicographic order $\prec$ in $\mathbb{Z}^m$ is the union of all the relations $\prec_\ell$: $i \prec j$ iff $i \prec_\ell j$ for some $\ell$ in $1 \leq \ell \leq m$.

The sequential execution of the iterations a loop nest follows the lexicographic order.

Assume that $(i; j)$ is a solution to (3), and that $i \prec j$. Then $d = j - i$ is the dependence distance of the dependence.
Uniform Dependence Distance

- If a dependence distance $d$ is a constant vector then the dependence is said to be uniform.
- The dependence distance $d = (1, 2)$ is uniform, while the dependence distance $d = (1, t_2)$ is nonuniform.
- Uniform distance vectors are very desirable since loops with only uniform distance vectors can be optimized with unimodular transformations.
- For this, the set of all distance vectors $d_i$ of a loop nest $L$ are arranged in a matrix with $n$ rows and $m$ columns where $n$ is the number of dependencies in $L$ and $m$ is the number of index variables in $L$.
- Note that a zero distance between references within the same statement does not cause a dependence, e.g. $a[i] = a[i] + x$. 

jonasskeppstedt.net Lecture 10 2018 20/62
A loop independent dependence is a dependence such that \( d = j - i = (0, \ldots, 0) \).

A loop independent dependence does not prevent concurrent execution of different iterations a loop. Rather, it constrains the scheduling of instructions in the loop body.

A loop carried dependence is a dependence which is not loop independent, or, in other words, the dependence is between two different iterations of a loop nest.

A dependence has level \( \ell \) if in \( d = j - i \), \( d_1 = 0 \), \( d_2 = 0 \), \ldots, \( d_{\ell-1} = 0 \), and \( d_{\ell} > 0 \).

Only a loop carried dependence has a level.
The GCD Test

- The GCD test was first described 1973.
- Consider the loop
  
  ```
  for (i = lb; i <= ub; ++i)
      x[a1 * i + c1] = x[a2 * i + c2] + y;
  ```

- To prove independence, we must show that the Diophantine's equation

  \[ a_1 i_1 - a_2 i_2 = c_2 - c_1 \]  \hspace{1cm} (4)

  has no solutions.
- We compute the gcd of \( a_1 \) and \( a_2 \) and check whether it divides \( c_2 - c_1 \), and if it does not, there is no solution and we have proved independence, otherwise we must use another test.
There are two weaknesses of the GCD test:

1. It does not exploit knowledge about the loop bounds.
2. Most often the gcd is one.

The first weakness means the GCD Test might be unable to prove independence despite the solution to (4) actually lies outside the index space of the loop.

The second weakness means dependence cannot be disproved.
The GCD Test can be extended to cover nested loops and multidimensional arrays.
The solution is then a vector and it usually contains unknowns.
The Fourier-Motzkin Test described shortly takes the solution vector from this GCD Test and checks whether the solution lies within the loop bounds.
Next we will look at unimodular matrices and Fourier-Motzkin elimination used by the Fourier-Motzkin Test.
Unimodular matrices

- An integer square matrix $A$ is unimodular if its determinant $\det(A) = \pm 1$.
- If $A$ and $B$ are unimodular, then $A^{-1}$ exists and is itself unimodular, and $A \times B$ is unimodular.
- $I$ is the $m \times m$ identity matrix.
Elementary row operations

The operations

- **reversal**: multiply a row by $-1$,
- **interchange**: interchange two rows, and
- **skewing**: add an integer multiple of one row to another row,

are called the elementary row operations. With each elementary row operation, there is a corresponding *elementary matrix*. 
3 \times 3 \text{ reversal matrices}

\[
\begin{pmatrix}
-1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix},
\]

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1
\end{pmatrix},
\]

and

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & -1
\end{pmatrix}.
\]
$3 \times 3$ interchange matrices

\[
\begin{pmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{pmatrix},
\]

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0
\end{pmatrix},
\]

and

\[
\begin{pmatrix}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0
\end{pmatrix}.
\]
3 $\times$ 3 upper skewing matrices

\[
\begin{pmatrix}
 1 & z & 0 \\
 0 & 1 & 0 \\
 0 & 0 & 1
\end{pmatrix},
\]

\[
\begin{pmatrix}
 1 & 0 & z \\
 0 & 1 & 0 \\
 0 & 0 & 1
\end{pmatrix},
\]

and

\[
\begin{pmatrix}
 1 & 0 & 0 \\
 0 & 1 & z \\
 0 & 0 & 1
\end{pmatrix}.
\]
3 \times 3 \text{ lower skewing matrices}

\[
\begin{pmatrix}
1 & 0 & 0 \\
z & 1 & 0 \\
0 & 0 & 1
\end{pmatrix},
\]

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
z & 0 & 1
\end{pmatrix},
\]

and

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & z & 1
\end{pmatrix}.
\]
Performing Elementary Row Operations

- To perform an elementary row operation on a matrix $A$, we can pre-multiply it with the corresponding elementary matrix.
- Assume we wish to interchange rows 1 and 3 in a $3 \times 3$ matrix $A$. The resulting matrix is formed by

$$
\begin{pmatrix}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0
\end{pmatrix} \times A.
$$

- The elementary matrices are all unimodular.
Let $l_{\rho}$ denote the column number of the first nonzero element of a matrix row.

A given $m \times n$ matrix $A$, is an echelon matrix if the following are satisfied for some integer $\rho$ in $0 \leq \rho \leq m$:

- rows 1 through $\rho$ are nonzero rows,
- rows $\rho + 1$ through $m$ are zero rows,
- for $1 \leq i \leq \rho + 1$, each element in column $l_i$ below row $i$ is zero, and
- $l_1 < l_2 < \ldots < l_{\rho}$.

The following are examples of echelon matrices:

\[
\begin{pmatrix}
1 & 2 & 3 \\
0 & 4 & 5 \\
0 & 0 & 6
\end{pmatrix}
\begin{pmatrix}
1 & 2 & 3 \\
0 & 0 & 4 \\
0 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
1 & 2 & 3 \\
0 & 4 & 5 \\
0 & 0 & 6 \\
0 & 0 & 0
\end{pmatrix}
\]
Echelon Reduction

- Given an $m \times n$ matrix $A$, Echelon reduction finds two matrices $U$ and $S$ such that $U \times A = S$, where $U$ is unimodular and $S$ is echelon.
- $U$ remains unimodular since we only apply elementary row operations.

```python
function echelon_reduce(A)
    U ← $I_m$
    S ← A
    $i_0 ← 0$
    for ($j ← 1; j ≤ n; j ← j + 1$) {
        if (there is a nonzero $s_{ij}$ with $i_0 < i ≤ m$) {
            $i_0 ← i_0 + 1$
            $i ← m$
            while ($i ≥ i_0 + 1$) {
                while ($s_{ij} ≠ 0$) {
                    $σ ← \text{sign}(s_{(i-1)j} × s_{ij})$
                    $z ← \lfloor |s_{(i-1)j}|/|s_{ij}| \rfloor$
                    subtract $σz$ (row $i$) from (row $i - 1$) in ($U; S$)
                }
                $i ← i - 1$
            }
        }
    }
    return $U$ and $S$
end
```
We will now show how one can echelon reduce the following matrix:

\[ A = \begin{pmatrix}
2 & 2 \\
-3 & 1 \\
3 & 1 \\
-4 & -2
\end{pmatrix}. \]

We start with \( U = I_4 \) and \( S = A \) which we write as:

\[ (U; S) = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 & 2 \\
0 & 1 & 0 & 0 & -3 & 1 \\
0 & 0 & 1 & 0 & 3 & 1 \\
0 & 0 & 0 & 1 & -4 & -2
\end{pmatrix}. \]

Then we will eliminate the nonzero elements in \( S \) starting with \( s_{41}, s_{31}, s_{21}, s_{42} \) and so on.
Example Echelon Reduction

- \( j = 1, i_0 = 1, i = 4 \). We always wish to eliminate \( s_{ij} \), which currently means \( s_{41} \).
- \( \sigma \leftarrow -1 \) and \( z \leftarrow 0 \). Nothing is subtracted from row 3.
- Then rows 3 and 4 are interchanged in \((U; S)\), resulting in:

\[
(U; S) = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 & 2 \\
0 & 1 & 0 & 0 & -3 & 1 \\
0 & 0 & 0 & 1 & -4 & -2 \\
0 & 0 & 1 & 0 & 3 & 1 \\
\end{pmatrix}
\]
We continue the inner while loop and find that $\sigma \leftarrow -1$ and $z \leftarrow 1$. Then $-1 \times \text{row 4}$ is subtracted from row 3, resulting in:

\[
(U; S) = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 & 2 \\
0 & 1 & 0 & 0 & -3 & 1 \\
0 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 1 & 0 & 3 & 1 \\
\end{pmatrix}.
\]

Then rows 3 and 4 are interchanged, resulting in:

\[
(U; S) = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 & 2 \\
0 & 1 & 0 & 0 & -3 & 1 \\
0 & 0 & 1 & 0 & 3 & 1 \\
0 & 0 & 1 & 1 & -1 & -1 \\
\end{pmatrix}.
\]
Example Echelon Reduction

- \(s_{41}\) is still zero, and the inner while loop is continued and \(\sigma \leftarrow -1\) and \(z \leftarrow 3\). Then \(-3 \times \) row 4 is subtracted from row 3:

\[
(U; S) = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 & 2 \\
0 & 1 & 0 & 0 & -3 & 1 \\
0 & 0 & 4 & 3 & 0 & -2 \\
0 & 0 & 1 & 1 & -1 & -1 \\
\end{pmatrix}.
\]

- Then rows 3 and 4 are interchanged, resulting in:

\[
(U; S) = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 & 2 \\
0 & 1 & 0 & 0 & -3 & 1 \\
0 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 4 & 3 & 0 & -2 \\
\end{pmatrix}.
\]

- Now the first \(ij\) has become zero and \(i\) is decremented.
Example Echelon Reduction

- \( j = 1, i_0 = 1, i = 3 \). We now wish to eliminate \( s_{31} \). \( \sigma \leftarrow +1 \) and \( z \leftarrow 3 \). Then \( 3 \times \) row 3 is subtracted from row 2:

\[
(U; S) = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 & 2 \\
0 & 1 & -3 & -3 & 0 & 4 \\
0 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 4 & 3 & 0 & -2
\end{pmatrix}.
\]

- Then rows 2 and 3 are interchanged, resulting in:

\[
(U; S) = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 & 2 \\
0 & 0 & 1 & 1 & -1 & -1 \\
0 & 1 & -3 & -3 & 0 & 4 \\
0 & 0 & 4 & 3 & 0 & -2
\end{pmatrix}.
\]
Example Echelon Reduction

- \( j = 1, i_0 = 1, i = 2 \). We now wish to eliminate \( s_{21} \). \( \sigma \leftarrow -1 \) and \( z \leftarrow 2 \). Then \(-2 \times \) row 2 is subtracted from row 1:

\[
(U; S) = \begin{pmatrix}
1 & 0 & 2 & 2 & 0 & 0 \\
0 & 0 & 1 & 1 & -1 & -1 \\
0 & 1 & -3 & -3 & 0 & 4 \\
0 & 0 & 4 & 3 & 0 & -2
\end{pmatrix}.
\]

- Interchanging rows 2 and 1 results in:

\[
(U; S) = \begin{pmatrix}
0 & 0 & 1 & 1 & -1 & -1 \\
1 & 0 & 2 & 2 & 0 & 0 \\
0 & 1 & -3 & -3 & 0 & 4 \\
0 & 0 & 4 & 3 & 0 & -2
\end{pmatrix}.
\]

jonasskeppstedt.net
Lecture 10
2018
39 / 62
Example Echelon Reduction

- \( j = 2, i_0 = 2, i = 4 \). We now wish to eliminate \( s_{42} \). \( \sigma \leftarrow -1 \) and \( z \leftarrow 2 \). \(-2 \times \) row 4 is subtracted from row 3:

\[
(U; S) = \begin{pmatrix}
0 & 0 & 1 & 1 & -1 & -1 \\
1 & 0 & 2 & 2 & 0 & 0 \\
0 & 1 & 5 & 3 & 0 & 0 \\
0 & 0 & 4 & 3 & 0 & -2 \\
\end{pmatrix}.
\]

- Interchanging rows 4 and 3 results in:

\[
(U; S) = \begin{pmatrix}
0 & 0 & 1 & 1 & -1 & -1 \\
1 & 0 & 2 & 2 & 0 & 0 \\
0 & 0 & 4 & 3 & 0 & -2 \\
0 & 1 & 5 & 3 & 0 & 0 \\
\end{pmatrix}.
\]
Example Echelon Reduction

- $j = 2, i_0 = 2, i = 3$. We now wish to eliminate $s_{32}$. $\sigma \leftarrow 0$ and $z \leftarrow 0$. Nothing is subtracted from row 2 but rows 3 and 2 are interchanged:

$$
(U; S) = \begin{pmatrix}
0 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 4 & 3 & 0 & -2 \\
1 & 0 & 2 & 2 & 0 & 0 \\
0 & 1 & 5 & 3 & 0 & 0 \\
\end{pmatrix}.
$$

At this point $S$ is an echelon matrix and the algorithm stops (the outer while loop since $i = i_0$). As will turn out to be convenient later, we prefer positive values of $s_{11}$ and therefore multiply with $-1$ finally resulting in:

$$
(U; S) = \begin{pmatrix}
0 & 0 & -1 & -1 & 1 & 1 \\
0 & 0 & 4 & 3 & 0 & -2 \\
1 & 0 & 2 & 2 & 0 & 0 \\
0 & 1 & 5 & 3 & 0 & 0 \\
\end{pmatrix}.
$$
GCD of multiple integers

- Let \( a_1, a_2, \ldots, a_m \) denote a list of integers, not all zero,
- \( U \) an \( m \times m \) unimodular matrix,
- \( S = (s_{11}, 0, \ldots, 0)^T \) an \( m \times 1 \) echelon matrix, such that \( UA = S \) where \( A \) is the \( m \times 1 \) matrix \((a_1, a_2, \ldots, a_m)^T\),
- then \( \gcd(a_1, a_2, \ldots, a_m) = |s_{11}|. \)
Data dependence testing in the simplest form with a single for loop and a one-dimensional array can be based on checking the GCD of the coefficients of the references.

It is possible to use the GCD test for multidimensional arrays and multiple nested for-loops.

With one-dimensional arrays and multiple nested for-loops, we get an equation of the form:

\[ a_1x_1 + a_2x_2 + \ldots + a_mx_m = c. \]

It is trivial to solve this when \( m = 1 \). Simply check if \( a_1 \) divides \( c \).

We will use Echelon reduction to rewrite the general case so that it can be solved trivially.
We write the equation on matrix form:

$$\mathbf{xA} = c$$  \hspace{1cm} (5)

By echelon reducing $\mathbf{A}$ such that $\mathbf{UA} = \mathbf{S}$ and choosing a positive $s_{11}$.

Recall $s_{11} = g = \gcd(a_1, a_2, ..., a_m)$.

The linear diophantine equation $\mathbf{xA} = c$ has a solution iff the gcd $g$ of its coefficients divides $c$. When a solution exists, the set of all solutions is given by

$$\mathbf{x} = \left(\frac{c}{g}, t_2, t_3, ..., t_m\right)\mathbf{U}$$  \hspace{1cm} (6)

where $t_i$ are arbitrary integers and $\mathbf{U}$ is any $m \times m$ unimodular matrix such that $\mathbf{UA} = (g, 0, ..., 0)^T$. 
To do data dependence analysis for multidimensional arrays we need to consider the general case of a system of \( n \) linear diophantine equations in \( m \) variables.

We have \( m/2 \) nested loops and \( n \) array dimensions.

\( m/2 \) variables for each reference i.e. \( m = 2 \) in
\[
a[2*i+3] = a[4*i+5]
\]

\[
xA = c
\]  
(7)

Here \( x \) (again) is an \( 1 \times m \) integer matrix, \( A \) is an \( m \times n \) integer matrix, and \( c \) is an \( 1 \times n \) integer matrix.

(7) is easy to solve if \( A \) is an echelon matrix.

With echelon reduction we find \( U \) and \( S \) such that \( UA = S \).

Then we will check if there is an integer solution to \( tS = c \) instead.
Linear Diophantine Equations

Theorem

Let $A$ be a given $m \times n$ integer matrix and $c$ a given integer $n$ vector.

Let $U$ denote an $m \times m$ integer matrix and $S$ an $m \times n$ integer echelon matrix, such that $UA = S$.

The system of equations

$$xA = c$$

(8)

has a solution iff there exists an integer $m$-vector $t$ such that $tS = c$. When a solution exists, the set of all solutions is given by the formula

$$x = tU$$

(9)

where $t$ is the integer vector which satisfies $tS = c$. 
Proof.

An integer vector $x = tU$ will be a solution to (8) iff

$$c = xA = tUA = tS$$  \hspace{1cm} (10)

If there is no such integer vector $t$ such that $tS = c$, then there is no integer solution to $x = tU$.

If there is such a $t$, then all solutions have the form $x = tU$, where $t$ is integral and $tS = c$. 
To illustrate how equations of the form $\mathbf{xA} = \mathbf{c}$ can be solved using the techniques introduced above, let us solve

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \end{pmatrix} \begin{pmatrix} 2 & 2 \\ -3 & 1 \\ 3 & 1 \\ -4 & -2 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix} \quad (11)$$

Firstly we use echelon reduction to find the matrices $\mathbf{U}$ and $\mathbf{S}$.

Then we formulate the equation $\mathbf{tS} = \mathbf{c}$:

$$\begin{pmatrix} t_1 & t_2 & t_3 & t_4 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & -2 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix} \quad (12)$$

It is trivially solved and we find that $\mathbf{t} = (2, -1, t_3, t_4)$, where $t_3$ and $t_4$ are arbitrary integers.
We then find $x$:

$$x = tU = \begin{pmatrix} 2 & -1 & t_3 & t_4 \end{pmatrix} \begin{pmatrix} 0 & 0 & -1 & -1 \\ 0 & 0 & 4 & 3 \\ 1 & 0 & 2 & 2 \\ 0 & 1 & 5 & 3 \end{pmatrix} = \begin{pmatrix} t_3, t_4, 2t_3 + 5t_4 - 7, 2t_3 + 3t_4 - 5 \end{pmatrix}$$
Suppose we find, during data dependence analysis, an integer vector $x$ which is a solution to $\phi$.

Then what we can conclude is that there exist index variables such that the two array references being tested can reference the same memory location.

If no solution can be found then we know there is no dependence. If there is a solution, then there may be a dependence.

If the solution $x$ represents index variables which are out of the loop bounds, then $x$ does not prove that a data dependence exists. So, we need also solve a linear inequality when the solution $x$ exists.

An additional constraint is, of course, that the solution is integer. Unfortunately, the problem of solving a linear integer inequality is NP-complete.
In 1827 Fourier published a method for solving linear inequalities in the real case. This method is known as Fourier-Motzkin elimination and is used in compilers as an approximation.

If Fourier-Motzkin elimination finds that there is no real solution, then there certainly is no integer either. But if there is a real solution, there may or may not be an integer solution.

Fourier-Motzkin elimination is regarded as a time-consuming algorithm and to apply it so perhaps thousands of data dependence tests may make the compiler too slow. Therefore, it is used as a backup tests when other faster tests fail to prove independence.
An interesting question is how frequently Fourier-Motzkin elimination finds a real solution when there is no integer solution. Some special cases can be exploited.

For instance, if a variable $x_i$ must satisfy $2.2 \leq x_i \leq 2.8$ then it is easy to see that no integer solution can exist. Otherwise, if we find eg that $2.2 \leq x_i \leq 4.8$ then we may try the two cases of setting $x_i = 3$ and $x_i = 4$, and see if there still is a real solution. Is easiest to understand Fourier-Motzkin elimination if we first look at an example.
Fourier-Motzkin Elimination

Assume we wish to solve the following system linear inequalities.

\[
\begin{align*}
2x_1 - 11x_2 & \leq 3 \\
-3x_1 + 2x_2 & \leq -5 \\
x_1 + 3x_2 & \leq 4 \\
-2x_1 & \leq -3
\end{align*}
\]

(15)

We will first eliminate \( x_2 \) from the system, and then check whether the remaining inequalities can be satisfied. To eliminate \( x_2 \), we start out with sorting the rows with respect to the coefficients of \( x_2 \):

\[
\begin{align*}
-3x_1 + 2x_2 & \leq -5 \\
x_1 + 3x_2 & \leq 4 \\
2x_1 - 11x_2 & \leq 3 \\
-2x_1 & \leq -3
\end{align*}
\]

(16)
Fourier-Motzkin Elimination

- First we want to have rows with positive coefficients of $x_2$, then negative, and lastly zero coefficients.
- Next we divide each row by its coefficient (if it is nonzero) of $x_2$:

\[
\begin{align*}
-\frac{3}{2}x_1 & + x_2 \leq \frac{-5}{2} \\
\frac{1}{3}x_1 & + x_2 \leq \frac{4}{3} \\
\frac{2}{11}x_1 & - x_2 \geq \frac{3}{11}
\end{align*}
\]  

(17)

Of course, the $\leq$ becomes $\geq$ when dividing with a negative coefficient. We can now rearrange the system to isolate $x_2$:

\[
\begin{align*}
x_2 & \leq \frac{3}{2}x_1 - \frac{5}{2} \\
x_2 & \leq -\frac{1}{3}x_1 + \frac{5}{3} \\
\frac{2}{11}x_1 - \frac{3}{11} & \leq x_2
\end{align*}
\]  

(18)
At this point, we make a record of the minimum and maximum values that $x_2$ can have, expressed as functions of $x_1$. We have:

$$b_2(x_1) \leq x_2 \leq B_2(x_1)$$  \hspace{1cm} (19)

where

$$b_2(x_1) = 2x_1$$
$$B_2(x_1) = \min\left(\frac{3}{2}x_1 - \frac{5}{2}, -\frac{1}{3}x_1 + \frac{4}{3}\right)$$ \hspace{1cm} (20)
To eliminate $x_2$ from the system, we simply combine the inequalities which had positive coefficients of $x_2$ with those which had negative coefficients (ie, one with positive coefficient is combined with one with negative coefficient):

$$\begin{align*}
\frac{2}{11} x_1 - \frac{3}{11} &\leq \frac{3}{2} x_1 - \frac{5}{2} \\
\frac{2}{11} x_1 - \frac{3}{11} &\leq -\frac{1}{3} x_1 + \frac{4}{3}
\end{align*}$$

(21)

These are simplified and the inequality with the zero coefficient of $x_2$ is brought back:

$$\begin{align*}
-\frac{29}{22} x_1 &\leq -\frac{49}{22} \\
-\frac{17}{33} x_1 &\leq \frac{53}{33} \\
-2x_1 &\leq -3
\end{align*}$$

(22)
We can now repeat parts of the procedure above:

\[
\begin{align*}
\textbf{x}_1 & \leq \frac{53}{17} \\
\textbf{x}_1 & \geq \frac{49}{29} \\
\textbf{x}_1 & \geq \frac{3}{2}
\end{align*}
\]  

We find that

\[
\begin{align*}
b_1() &= \max(49/29, 3/2) = 49/29 \\
B_1() &= 53/17
\end{align*}
\]  

The solution to the system is $\frac{49}{29} \leq x_1 \leq \frac{53}{17}$ and $b_2(x_1) \leq B_2(x_1)$ for each value of $x_1$. 
procedure fourier_motzkin_elimination (x, A, c)
    \( r \leftarrow m, \quad s \leftarrow n, \quad T \leftarrow A, \quad q \leftarrow c \)
    while (1) 
        \( n_1 \leftarrow \) number of inequalities with positive \( t_{rj} \)
        \( n_2 \leftarrow n_1 \) + number of inequalities with negative \( t_{rj} \)
        Sort the inequalities so that the \( n_1 \) with \( t_{rj} > 0 \) come first,
        then the \( n_2 - n_1 \) with \( t_{rj} < 0 \) come next,
        and the ones with \( t_{rj} = 0 \) come last.
        for \( (i = 1; i \leq r - 1; i \leftarrow i + 1) \)
            for \( (j = 1; i \leq n_2; j \leftarrow j + 1) \)
                \( t_{ij} \leftarrow t_{ij} / t_{rj} \)
            for \( (j = 1; i \leq n_2; j \leftarrow j + 1) \)
                \( q_j \leftarrow q_j / t_{rj} \)
        if \( (n_2 > n_1) \)
            \( b_r(x_1, x_2, \ldots, x_{r-1}) = \max_{n_1 + 1 \leq j \leq n_2} \left( -\sum_{i=1}^{r-1} t_{ij} x_i + q_i \right) \)
        else
            \( b_r \leftarrow -\infty \)
        if \( (n_1 > 0) \)
            \( j_r(x_1, x_2, \ldots, x_{r-1}) = \min_{n_1 + 1 \leq j \leq n_2} \left( -\sum_{i=1}^{r-1} t_{ij} x_i + q_i \right) \)
        else
            \( B_r \leftarrow \infty \)
        if \( (r = 1) \)
            return make_solution()
/* We will now eliminate \(x_r\). */

\[
s' \leftarrow s - n_2 + n_1(n_2 - n_1)
\]

if \((s' = 0)\) {

/* We have not discovered any inconsistency and */
/* we have no more inequalities to check. */
/* The system has a solution. */

The solution set consists of all real vectors \((x_1, x_2, \ldots, x_m)\),
where \(x_{r-1}, x_{r-2}, \ldots, x_1\) are chosen arbitrarily, and
\(x_m, x_{m-1}, \ldots, x_r\) must satisfy
\[
b_i(x_1, x_2, \ldots, x_{i-1}) \leq x_i \leq B_i(x_1, x_2, \ldots, x_{i-1}) \text{ for } r \leq i \leq m.
\]

return solution set.

} /* There are now \(s'\) inequalities in \(r-1\) variables. */

The new system of inequalities is made of two parts:
\[
\sum_{i}^{r-1} (t_{ik} - t_{ij})x_i \leq q_k - q_j \text{ for } 1 \leq k \leq n_1, n_1 + 1 \leq j \leq n_2
\]
\[
\sum_{i}^{r-1} t_{ij}x_i \leq q_j \text{ for } n_2 + 1 \leq j \leq s
\]
and becomes by setting \(r = r \leftarrow 1\) and \(s \leftarrow s'\):
\[
\sum_{i}^{r} t_{ij}x_i \leq q_j \text{ for } 1 \leq j \leq s
\]
}

} /* We have come to the last variable \(x_1\). */

if \((b_1 > B_1 \text{ or (there is a } q_j < 0 \text{ for } n_2 + 1 \leq j \leq s))\)

return there is no solution

The solution set consists of all real vectors \((x_1, x_2, \ldots, x_m)\),
such that \(b_i(x_1, x_2, \ldots, x_m) \leq x_i \leq B_i(x_1, x_2, \ldots, x_m)\) for \(1 \leq i \leq m\).

return solution set.

end
Summary, Step 1: GCD Test

- In the case of a loop nest of height $m$ and an $n$-dimensional array, we use the matrix representation of the references $iA + a_0 = jB + b_0$, or equivalently:

\[(i; j) \begin{pmatrix} A \\ -B \end{pmatrix} = b_0 - a_0, \quad (25)\]

where the $A$ and $B$ have $m$ rows and $n$ columns.

- We find a $2m \times 2m$ unimodular matrix $U$ and a $2m \times n$ echelon matrix $S$ such that

\[U \begin{pmatrix} A \\ -B \end{pmatrix} = S. \quad (26)\]

- If there is a $2m$ vector $t$ which satisfies $tS = b_0 - a_0$ then the GCD test cannot exclude dependence, and if so...

- ..., the computed $t$ will be input to the Fourier-Motzkin Test.
If the GCD Test found a solution vector $t$ to $tS = c$, these solutions will be tested to see if they are within the loop bounds.

Recall we wrote

$$x = (i; j) \begin{pmatrix} A \\ -B \end{pmatrix} = b_0 - a_0.$$  \hfill (27)

We find $x$ from:

$$x = (i; j) = tU$$  \hfill (28)

With $U_1$ being the left half of $U$ and $U_2$ the right half we have:

$$i = tU_1$$  \hfill (29)

$$j = tU_2$$  \hfill (30)

These should be inserted to loop bounds constraints.
Recall the original loop bounds are:

\[
\begin{align*}
p_0 & \leq IP \\
IQ & \leq q_0
\end{align*}
\]

The solution vector \( t \) must satisfy:

\[
\begin{align*}
p_0 & \leq tU_1P \\
tU_1Q & \leq q_0 \\
p_0 & \leq tU_2P \\
tU_2Q & \leq q_0
\end{align*}
\]

If there is no integer solution to this system, there is no dependence. Recall, however, the system is solved with real or rational numbers so the Fourier-Motzkin Test may fail to exclude independence.