Contents of Lecture 10

- Introduction to Data Dependence Analysis
- The GCD Test
- The Fourier-Motzkin Test

Introduction to Data Dependence Analysis

- There are **data dependencies** in the following code:

```
S1: x = a + b;
S2: y = x + 1;
S3: x = b * c;
```

- The value written to `x` in S_1 is read in S_2.
- 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 at Different Levels

- Data dependencies can be at several different levels, including:
  - Instructions
  - Statements
  - Loop iterations
  - Functions
  - Threads
- We will focus on dependencies at the instruction and loop levels.
- Parallelizing compilers focus on loop iterations.
- 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.

Dependencies in the Example Code

- Let us classify all dependencies in the code:

```
S1: x = a + b;
S2: y = x + 1;
S3: x = b * 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`.

Loop Level Data Dependencies

- 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 $i + 3$.
- E.g. 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 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 $I_1, ..., I_{r-1}$.

$$
\begin{array}{l}
\textbf{for } (I_1 = p_1;\ I_1 <= q_1;\ I_1 += 1)\ \{ \\
\quad \textbf{for } (I_2 = p_2;\ I_2 <= q_2;\ I_2 += 1)\ \{ \\
\quad\quad \vdots \\
\quad\quad \textbf{for } (I_m = p_m;\ I_m <= q_m;\ I_m += 1)\ \{ \\
\quad\quad\quad h(I_1, I_2, ..., I_m); \\
\quad\quad \} \\
\quad \} \\
\}
\end{array}
$$

Example Perfect Loop Nest

- All assignments, **except** to the loop index variables are in the innermost loop.
- There may be any number of assignment statements in the innermost loop.

```
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];
                }
        }
}
```

Loop Bounds

- The lower bound for l_1 is $p_{10} \leq l_1$.
- The lower bound for l_2 is

$$\begin{aligned} l_2 &\geq p_{20} + p_{21} l_1 \\ p_{20} &\leq l_2 - p_{21} l_1 \\ p_{20} &\leq -p_{21} l_1 + l_2 \end{aligned} \tag{1}$$

- The lower bound for l_3 is

$$\begin{aligned} l_3 &\geq p_{30} + p_{31} l_1 + p_{32} l_2 \\ p_{30} &\leq l_3 - p_{31} l_1 - p_{32} l_2 \\ p_{30} &\leq -p_{31} l_1 - p_{32} l_2 + l_3 \end{aligned} \tag{2}$$

and so forth. We represent this on matrix form as $\mathbf{p_0} \leq \mathbf{lP}$.

Loop Bounds on Matrix Form

- $\mathbf{P} = \begin{pmatrix} 1 & -p_{21} & -p_{31} & \dots & -p_{m1} \\ 0 & 1 & -p_{32} & \dots & -p_{m2} \\ 0 & 0 & 1 & \dots & -p_{m3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$ and $\mathbf{p_0} = (p_{10}, p_{20}, \dots, p_{m0})$.
- Similarly, the upper bounds are represented as $\mathbf{IQ} \le \mathbf{q_0}$.
- The loop bounds, thus, are represented by the system:

$$\left.\begin{array}{ccccc} \mathbf{p_0} & \le & \mathbf{IP} & & \\ & & \mathbf{IQ} & \le & \mathbf{q_0} \end{array}\right\}$$

Example Non-Perfect Loop Nest

- 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.

```
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];
                }
        }
}
```

Loop Distribution

- 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 $\mathbf{I} = (I_1, I_2, ..., I_m)$ is the vector of index variables.
- The index values of $\mathbf{L}$ are the values of $(I_1, I_2, ..., I_m)$.
- The index space of $\mathbf{L}$ is the subspace of 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.

Symbolic Analysis

- 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.

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

Problematic Non-Affine Index Functions Problems

- In the loop

```
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.

Representing Array References

- 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 $X[\mathbf{IA} + \mathbf{a_0}]$, where
- $\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$ and $\mathbf{a}_0 = (a_{10}, a_{20}, ..., a_{n0})$.
- We will refer to $\mathbf{A}$ and $\mathbf{a_0}$ as the **coefficient matrix** and the **constant term**, respectively.

The Data Dependence Equation

- For two references $X[\mathbf{IA} + \mathbf{a_0}]$ and $X[\mathbf{IB} + \mathbf{b_0}]$ to refer to the same array element there must be two index values, $\mathbf{i}$ and $\mathbf{j}$ such that $\mathbf{iA} + \mathbf{a_0} = \mathbf{jB} + \mathbf{b_0}$ which we can write as $\mathbf{iA} - \mathbf{jB} = \mathbf{b_0} - \mathbf{a_0}$.
- This system of Diophantine 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:

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

- We solve the system of linear Diophantine equations in (3) using a method presented shortly.

An Example

```
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:

  $\mathbf{A} = \begin{pmatrix} 2 & 2 \\ -3 & 1 \end{pmatrix}$ and $\mathbf{a}_0 = (-1, -3)$.

  $\mathbf{B} = \begin{pmatrix} -3 & -1 \\ 4 & 2 \end{pmatrix}$ and $\mathbf{b}_0 = (1, 7)$.

Dependence Distances

- Let $\prec_\ell$ be a relation in $\mathbf{Z}^m$ such that $\mathbf{i} \prec_\ell \mathbf{j}$ if $i_1 = j_1$, $i_2 = j_2$, ..., $i_{l-1} = j_{l-1}$, and $i_l < j_l$.
- For example: $(1, 3, 4) \prec_3 (1, 3, 9)$.
- The lexicographic order $\prec$ in $\mathbf{Z}^m$ is the union of all the relations $\prec_\ell$: $\mathbf{i} \prec \mathbf{j}$ iff $\mathbf{i} \prec_\ell \mathbf{j}$ for some ℓ in $1 \leq \ell \leq m$.
- The sequential execution of the iterations a loop nest follows the lexicographic order.
- Assume that $(\mathbf{i}; \mathbf{j})$ is a solution to (3), and that $\mathbf{i} \prec \mathbf{j}$. Then $\mathbf{d} = \mathbf{j} - \mathbf{i}$ is the **dependence distance** of the dependence.

Uniform Dependence Distance

- If a dependence distance $\mathbf{d}$ is a constant vector then the dependence is said to be uniform.
- The dependence distance $\mathbf{d} = (1, 2)$ is uniform, while the dependence distance $\mathbf{d} = (1, t_2)$ 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 $\mathbf{d_i}$ of a loop nest $\mathbf{L}$ are arranged in a matrix with n rows and m columns where n is the number of dependencies in $\mathbf{L}$ and m is the number of index variables in $\mathbf{L}$.

Loop Independent and Loop Carried Dependencies

- A loop independent dependence is a dependence such that $\mathbf{d} = \mathbf{j} - \mathbf{i} = (0, ..., 0)$.
- 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 ℓ if in $\mathbf{d} = \mathbf{j} - \mathbf{i}$, $\mathbf{d}_1 = 0, \mathbf{d}_2 = 0, ..., \mathbf{d}_{l-1} = 0$, and $\mathbf{d}_l > 0$.
- 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 equation

$$a_1 i_1 - a_2 i_2 = c_2 - c_1 \tag{4}$$

  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.

Weaknesses of The GCD 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.

GCD Test for Nested Loops and Multidimensional Arrays

- 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 $\mathbf{A}$ is unimodular if its determinant $det(\mathbf{A}) = \pm 1$.
- If $\mathbf{A}$ and $\mathbf{B}$ are unimodular, then $\mathbf{A}^{-1}$ exists and is itself unimodular, and $\mathbf{A} \times \mathbf{B}$ is unimodular.
- $\mathcal{I}$ 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*.

3×3 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
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}.$$

3×3 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
- $$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

3×3 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
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & z \\ 0 & 0 & 1 \end{pmatrix}.$$

3×3 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 $\mathbf{A}$, we can pre-multiply it with the corresponding elementary matrix.
- Assume we wish to interchange rows 1 and 3 in a 3×3 matrix $\mathbf{A}$. The resulting matrix is formed by

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

- The elementary matrices are all unimodular.

Echelon Matrices

- Let l_ρ denote the column number of the first nonzero element of a matrix row.
- A given $m \times n$ matrix $\mathbf{A}$, is an *echelon matrix* if the following are satisfied for some integer ρ in $0 \leq \rho \leq m$:
  - rows 1 through ρ 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
  - $l_1 < l_2 < ... < l_\rho$.
- 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 $\mathbf{A}$, Echelon reduction finds two matrices $\mathbf{U}$ and $\mathbf{S}$ such that $\mathbf{U} \times \mathbf{A} = \mathbf{S}$, where $\mathbf{U}$ is unimodular and $\mathbf{S}$ is echelon.
- $\mathbf{U}$ remains unimodular since we only apply elementary row operations.

```
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 ← sign(s_(i−1)j × s_ij)
                        z ← ⌊|s_(i−1)j| / |s_ij|⌋
                        subtract σz(row i) from (row i − 1) in (U; S)
                        interchange rows i and i − 1 in (U; S)
                    }
                    i ← i − 1
                }
            }
        }
    return U and S
end
```

Example Echelon Reduction

- We will now show how one can echelon reduce the following matrix:

$$\mathbf{A} = \begin{pmatrix} 2 & 2 \\ -3 & 1 \\ 3 & 1 \\ -4 & -2 \end{pmatrix}.$$

- We start with with $\mathbf{U} = \mathbf{I_4}$ and $\mathbf{S} = \mathbf{A}$ which we write as:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 0 & 3 & 1 \\ 0 & 0 & 0 & 1 & -4 & -2 \end{array}\right).$$

- Then we will eliminate the nonzero elements in $\mathbf{S}$ starting with $s_{41}, s_{31}, s_{21}, s_{42}$ and so on.

Example Echelon Reduction

- $\mathbf{j = 1, i_0 = 1, i = 4}$. 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 $(\mathbf{U}; \mathbf{S})$, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|rr} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 0 & 1 & -4 & -2 \\ 0 & 0 & 1 & 0 & 3 & 1 \end{array}\right).$$

Example Echelon Reduction

- We continue the inner while loop and find that $\sigma \leftarrow -1$ and $z \leftarrow 1$. Then $-1\times$ row 4 is subtracted from row 3, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 1 & 0 & 3 & 1 \end{array}\right).$$

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

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 0 & 3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \end{array}\right).$$

Example Echelon Reduction

- s_{41} is still nonzero, 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:

$$(\mathbf{U};\mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 0 & 0 & 1 & 1 & -1 & -1 \end{array}\right).$$

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

$$(\mathbf{U};\mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Now the first $_{ij}$ has become zero and i is decremented.

Example Echelon Reduction

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

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{rrrr|rr} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

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

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{rrrr|rr} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

Example Echelon Reduction

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

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Interchanging rows 2 and 1 results in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

Example Echelon Reduction

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

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Interchanging rows 4 and 3 results in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

Example Echelon Reduction

- $\mathbf{j = 2, i_0 = 2, i = 3}$. 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:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|rr} 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

At this point $\mathbf{S}$ 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:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|rr} 0 & 0 & -1 & -1 & 1 & 1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

GCD of multiple integers

- Let $a_1, a_2, ..., a_m$ denote a list of integers, not all zero,
- $\mathbf{U}$ an $m \times m$ unimodular matrix,
- $\mathbf{S} = (s_{11}, 0, ...0)^T$ an $m \times 1$ echelon matrix, such that $\mathbf{UA} = \mathbf{S}$ where $\mathbf{A}$ is the $m \times 1$ matrix $(a_1, a_2, ..., a_m)^T$,
- then $\gcd(a_1, a_2, ..., a_m) = |s_{11}|$.

Nested loop and one dimensional array 1(2)

- 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:

$$a_1x_1 + a_2x_2 + ... + a_mx_m = c.$$

  It is trivial to solve this when $m = 1$. Simply check if a_1 divides c.
- We can use echelon reduction to rewrite the general case so that it can be solved trivially.

Nested loop and one dimensional array 2(2)

- We write the equation on matrix form:

$$\mathbf{xA} = c \tag{5}$$

- We echelon reduce $\mathbf{A}$ such that $\mathbf{UA} = \mathbf{S}$ and select 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

$$\mathbf{x} = (c/g, t_2, t_3, ..., t_m)\mathbf{U} \tag{6}$$

where t_i are arbitrary integers and $\mathbf{U}$ is any $m \times m$ unimodular matrix such that $\mathbf{UA} = (g, 0, ..., 0)^T$.

Nested loop and multidimensional array

- General case: array with n dimensions and $m/2$ loop levels.

$$\mathbf{xA} = \mathbf{c} \tag{7}$$

  Here $\mathbf{x}$ (again) is an $1 \times m$ integer matrix, $\mathbf{A}$ is an $m \times n$ integer matrix, and $\mathbf{c}$ is an $1 \times n$ integer matrix.
- (7) is easy to solve if $\mathbf{A}$ is an echelon matrix (but it is not).
- With echelon reduction we instead find $\mathbf{U}$ and $\mathbf{S}$ such that $\mathbf{UA} = \mathbf{S}$.
- Then we will check if there is an integer solution to $\mathbf{tS} = \mathbf{c}$ instead.

Linear Diophantine Equations

Theorem

- *Let* $\mathbf{A}$ *be a given* $m \times n$ *integer matrix and* $\mathbf{c}$ *a given integer* n *vector.*
- *Let* $\mathbf{U}$ *denote an* $m \times m$ *integer matrix and* $\mathbf{S}$ *an* $m \times n$ *integer echelon matrix, such that* $\mathbf{UA} = \mathbf{S}$.

The system of equations

$$\mathbf{xA} = \mathbf{c} \tag{8}$$

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

$$\mathbf{x} = \mathbf{tU} \tag{9}$$

where $\mathbf{t}$ *is the integer vector which satisfies* $\mathbf{tS} = \mathbf{c}$.

Linear Diophantine Equations

Proof.

- An integer m-vector $\mathbf{x} = \mathbf{tU}$ will be a solution to (8) iff

$$\mathbf{c} = \mathbf{xA} = \mathbf{tUA} = \mathbf{tS} \tag{10}$$

- If there is no integer vector $\mathbf{t}$ such that $\mathbf{tS} = \mathbf{c}$, then there is no integer solution to $\mathbf{xA} = \mathbf{c}$ either.
- If there is such a $\mathbf{t}$, then all solutions have the form $\mathbf{x} = \mathbf{tU}$, where $\mathbf{t}$ is integral and $\mathbf{tS} = \mathbf{c}$.

□

Linear Diophantine Equations

- To illustrate how equations of the form $\mathbf{xA} = \mathbf{c}$ can be solved using the techniques introduced above, let us solve

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

- Firstly we use echelon reduction to find the matrices $\mathbf{U}$ and $\mathbf{S}$.
- Then we formulate the equation $\mathbf{tS} = \mathbf{c}$:

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

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 $\mathbf{x}$:

$$\mathbf{x} = \mathbf{tU} = \begin{pmatrix} 2 & -1 & t_3 & t_4 \end{pmatrix} \begin{pmatrix} 0 & 0 & -1 & -1 \\ 0 & 0 & 4 & 3 \\ 1 & 0 & 2 & 2 \\ 0 & 1 & 5 & 3 \end{pmatrix} = \tag{13}$$

$$(t_3, t_4, 2t_3 + 5t_4 - 7, 2t_3 + 3t_4 - 5) \tag{14}$$

Fourier-Motzkin Elimination

- If no solution was found then we know there is no dependence.
- Suppose we found a solution integer vector $\mathbf{x}$.
- 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 the solution $\mathbf{x}$ represents index variables which are out of the loop bounds, then $\mathbf{x}$ does not prove that a data dependence exists. So, we need also solve a linear inequality when the solution $\mathbf{x}$ exists.

Fourier-Motzkin Elimination

- 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.

Fourier-Motzkin Elimination

- For instance, if a variable x_i must satisfy $2.2 \leq x_i \leq 2.8$ then no integer solution can exist.
- 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.

Fourier-Motzkin Elimination

- Assume we wish to solve the following system linear inequalities.

$$\begin{array}{rcrcr} 2x_1 & - & 11x_2 & \leq & 3 \\ -3x_1 & + & 2x_2 & \leq & -5 \\ x_1 & + & 3x_2 & \leq & 4 \\ -2x_1 & & & \leq & -3 \end{array} \tag{15}$$

- 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:

$$\begin{array}{rcrcr} -3x_1 & + & 2x_2 & \leq & -5 \\ x_1 & + & 3x_2 & \leq & 4 \\ 2x_1 & - & 11x_2 & \leq & 3 \\ -2x_1 & & & \leq & -3 \end{array} \tag{16}$$

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:

$$\begin{array}{rcrcr} \frac{-3}{2}x_1 & + & x_2 & \leq & \frac{-5}{2} \\ \frac{1}{3}x_1 & + & x_2 & \leq & \frac{4}{3} \\ \frac{2}{11}x_1 & - & x_2 & \geq & \frac{3}{11} \end{array} \tag{17}$$

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

$$\begin{array}{rcrcrcrcr} & & & & x_2 & \leq & \frac{3}{2}x_1 & - & \frac{5}{2} \\ & & & & x_2 & \leq & -\frac{1}{3}x_1 & + & \frac{4}{3} \\ \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & x_2 & & & & \end{array} \tag{18}$$

Fourier-Motzkin Elimination

- 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:

$$b_2(x_1) \leq x_2 \leq B_2(x_1) \tag{19}$$

where

$$\begin{aligned} b_2(x_1) &= \frac{2}{11}x_1 \\ B_2(x_1) &= \min(\tfrac{3}{2}x_1 - \tfrac{5}{2}, -\tfrac{1}{3}x_1 + \tfrac{4}{3}) \end{aligned} \tag{20}$$

Fourier-Motzkin Elimination

- 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):

$$\begin{array}{rcrcrcr} \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & \frac{3}{2}x_1 & - & \frac{5}{2} \\ \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & -\frac{1}{3}x_1 & + & \frac{4}{3} \end{array} \tag{21}$$

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

$$\begin{array}{rcr} -\frac{29}{22}x_1 & \leq & -\frac{49}{22} \\ -\frac{17}{33}x_1 & \leq & \frac{53}{33} \\ -2x_1 & \leq & -3 \end{array} \tag{22}$$

Fourier-Motzkin Elimination

- We can now repeat parts of the procedure above:

$$\begin{aligned} x_1 &\leq \frac{53}{17} \\ x_1 &\geq \frac{49}{29} \\ x_1 &\geq \frac{3}{2} \end{aligned} \tag{23}$$

- We find that

$$\begin{aligned} b_1() &= \max(49/29, 3/2) = 49/29 \\ B_1() &= 53/17 \end{aligned} \tag{24}$$

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.

Fourier-Motzkin Elimination

procedure *fourier_motzkin_elimination* (x, A, c)

$r \leftarrow m$, $s \leftarrow n$, $\mathbf{T} \leftarrow \mathbf{A}$, $\mathbf{q} \leftarrow \mathbf{c}$

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 $(n_2 > n_1)$

$b_r(x_1, x_2, \ldots, x_{r-1}) = \max_{n_1+1 \leq j \leq n_2}(-\sum_{i=1}^{r-1} t_{ij}x_i + q_i)$

else

$b_r \leftarrow -\infty$

if $(n_1 > 0)$

$j_r(x_1, x_2, \ldots, x_{r-1}) = \min_{n_1+1 \leq j \leq n_2}(-\sum_{i=1}^{r-1} t_{ij}x_i + q_i)$

else

$B_r \leftarrow \infty$

if $(r = 1)$

return *make_solution()*

Fourier-Motzkin Elimination

```
        /* We will now eliminate x_r. */
        s' ← 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, ..., x_m),
            where x_{r−1}, x_{r−2}, ..., x_1 are chosen arbitrarily, and
            x_m, x_{m−1}, ..., x_r must satisfy
            b_i(x_1, x_2, ..., x_{i−1}) ≤ x_i ≤ B_i(x_1, x_2, ..., x_{i−1}) for r ≤ i ≤ m.
            return solution set.
        }
        /* There are now s' inequalities in r − 1 variables. */
        The new system of inequalities is made of two parts:
        Σ_i^{r−1} (t_ik − t_il) x_i ≤ q_k − q_j for 1 ≤ k ≤ n_1, n_1 + 1 ≤ j ≤ n_2
        Σ_i^{r−1} t_ij x_i ≤ q_j for n_2 + 1 ≤ j ≤ s
        and becomes by setting r = r ← 1 and s ← s':
        Σ_i^r t_ij x_i ≤ q_j for 1 ≤ j ≤ s
} end


function make_solution ()
    /* We have come to the last variable x_1. */
    if (b_1 > B_1  or (there is a q_j < 0 for n_2 + 1 ≤ j ≤ s))
        return there is no solution
    The solution set consists of all real vectors (x_1, x_2, ..., x_m),
        such that b_i(x_1, x_2, ..., x_m) ≤ x_i ≤ B_i(x_1, x_2, ..., x_m) for 1 ≤ i ≤ 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 $\mathbf{iA} + \mathbf{a_0} = \mathbf{jB} + \mathbf{b_0}$, or equivalently:

$$(\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b_0} - \mathbf{a_0}, \tag{25}$$

  where the $\mathbf{A}$ and $\mathbf{B}$ have m rows and n columns.
- We find a $2m \times 2m$ unimodular matrix $\mathbf{U}$ and a $2m \times n$ echelon matrix $\mathbf{S}$ such that

$$\mathbf{U} \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{S}. \tag{26}$$

- If there is a $2m$ vector $\mathbf{t}$ which satisfies $\mathbf{tS} = \mathbf{b_0} - \mathbf{a_0}$ then the GCD test cannot exclude dependence, and if so...
- ..., the computed $\mathbf{t}$ will be input to the Fourier-Motzkin Test.

Summary, Step 2: Fourier-Motzkin Test 1(2)

- If the GCD Test found a solution vector $\mathbf{t}$ to $\mathbf{tS} = \mathbf{c}$, these solutions will be tested to see if they are within the loop bounds.
- Recall we wrote

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

- We find $\mathbf{x}$ from:

$$\mathbf{x} = (\mathbf{i}; \mathbf{j}) = \mathbf{tU} \tag{28}$$

- With $\mathbf{U_1}$ being the left half of $\mathbf{U}$ and $\mathbf{U_2}$ the right half we have:

$$\mathbf{i} = \mathbf{tU_1} \tag{29}$$

$$\mathbf{j} = \mathbf{tU_2} \tag{30}$$

- These should be inserted to loop bounds constraints.

Summary, Step 2: Fourier-Motzkin Test 2(2)

- Recall the original loop bounds are:

$$\left.\begin{array}{rcccl} \mathbf{p_0} & \leq & \mathbf{IP} & & \\ & & \mathbf{IQ} & \leq & \mathbf{q_0} \end{array}\right\}$$

- The solution vector $\mathbf{t}$ must satisfy:

$$\left.\begin{array}{rcccl} \mathbf{p_0} & \leq & \mathbf{tU_1P} & & \\ & & \mathbf{tU_1Q} & \leq & \mathbf{q_0} \\ \mathbf{p_0} & \leq & \mathbf{tU_2P} & & \\ & & \mathbf{tU_2Q} & \leq & \mathbf{q_0} \end{array}\right\} \tag{31}$$

- 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.