Compiler
Optimizations for
Uniprocessors

HPC
Prof. Robert van Engelen
Compiler Optimizations

- General optimizations
- Single loop optimizations
- Nested loop optimizations
- Global optimizations
# General Optimizations

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduce memory instructions</th>
<th>Reduce FP instructions</th>
<th>Reduce integer instructions</th>
<th>Reduce branch instructions</th>
<th>Reduce memory latencies and/or memory bandwidth</th>
<th>Reduce Memory latencies</th>
<th>Reduce instruction effect of instruction latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data speculation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Control speculation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Register allocation</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>C/C++ register type</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>C/C++ asm macro</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>C/C++ include &lt;math.h&gt;</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Uniqueness of memory addr</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Dead code elimination</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Constant folding and propagation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Common subexpr. elimination</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Strength reductions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Increase FMA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Fill branch delay slots</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Data and Control Dependences

- Fundamental property of all computations

- Data dependence: data is produced and consumed in the correct order

- Control dependence: a dependence that arises as a result of control flow

\[
\begin{align*}
S_1 & \quad \text{PI} = 3.14 \\
S_2 & \quad R = 5.0 \\
S_3 & \quad \text{AREA} = \text{PI} \times R^{**} 2
\end{align*}
\]

\[
\begin{align*}
S_1 & \quad \text{IF} (T \text{.EQ.} 0.0) \ \text{GOTO} \ S_3 \\
S_2 & \quad A = A / T \\
S_3 & \quad \text{CONTINUE}
\end{align*}
\]
Data Dependence

Definition

There is a data dependence from statement $S_1$ to statement $S_2$ iff

1) both statements $S_1$ and $S_2$ access the same memory location and at least one of them writes to it, and

2) there is a feasible run-time execution path from statement $S_1$ to $S_2$
Data Dependence Classification

- **Definition**

  A dependence relation $\delta$ is

  - A flow dependence (RAW), denoted $S_1 \delta S_2$
  - An anti dependence (WAR), denoted $S_1 \delta^{-1} S_2$
  - An output dependence (WAW), denoted $S_1 \delta^o S_2$
Data Speculation (Again)

- When a store to address A1 is followed by a load from address A2 then there is a flow dependence when A1=A2
  - A compiler assumes there is a flow dependence if it cannot disprove A1=A2
- The *advanced load instruction* allows a process to ignore a potential flow dependence and sort out the conflict at run time when the store address is known and is the same

```c
double *p, a[10];
...
*p = 0;
s += a[i];

r0 = 0
r1 = p
...

r2 = a+i
store r0 in M[r1]
load M[r2] into r3

r2 = a+i
adv load M[r2] into r3
...

r0 = 0
r1 = p
store r0 in M[r1]
check adv load address: reload r3
```
Control Speculation (Again)

Control speculation allows conditional instructions to be executed before the conditional branch in which the instruction occurs

- Hide memory latencies

- A speculative load instruction performs a load operation
  - Enables loading early
  - Exceptions are ignored

- A check operation verifies whether the load triggered an exception (e.g. bus error)
  - Reraise the exception

```c
if (i<n)
    x = a[i]

if (i>n) jump to skip
load a[i] into r0
skip:
    ...

speculative load a[i] into r0
if (i>n) jump to skip
    check spec load exceptions
skip:
    ...
```
Register Allocation

Example

```plaintext
a := read();
b := read();
c := read();
a := a + b + c;
if (a<10)
  {   d := c + 8;
    write(c);
  } else if (a<20)
  {   e := 10;
    d := e + a;
    write(e);
  } else
    f := 12;
    d := f + a;
    write(f);
write(d);
```

Live range of a variable is a path from the write to the last read statement of the variable.
Register Allocation

Construct interference graph

Solve

Assign registers:
- a = r2
- b = r3
- c = r1
- d = r2
- e = r2
- f = r1

\[
\begin{align*}
\text{r2} & := \text{read}(); \\
\text{r3} & := \text{read}(); \\
\text{r1} & := \text{read}(); \\
\text{r2} & := \text{r2} + \text{r3} + \text{r1}; \\
\text{if} \ (\text{r2}<10) \\
\{ & \quad \text{r2} := \text{r1} + 8; \\
& \quad \text{write(r1);} \\
\} & \quad \text{else if} \ (\text{r2}<20) \\
\{ & \quad \text{r1} := 10; \\
& \quad \text{r2} := \text{r1} + \text{r2}; \\
& \quad \text{write(r1);} \\
\} & \quad \text{else} \\
\{ & \quad \text{r1} := 12; \\
& \quad \text{r2} := \text{r1} + \text{r2}; \\
& \quad \text{write(r1);} \\
\} & \quad \text{write(r2);} \\
\end{align*}
\]

- **Register pressure**: when there is an insufficient number of registers to hold all live variables, then register values have to be stored in memory (*register spill*)
- This results in lower performance when spill occurs frequently such as within a loop
C/C++ Register Data Type

- The C/C++ register data type is a hint to the compiler to store the scalar variable in a register to avoid register spill
- Additionally prevents potential pointer aliases to this variable

```c
register int i;
int k = 10;
int a[100];
int *p = init(a, &k);
for (i = 0; i < 100; i++)
    *p++ = k;
```

int *init(int a[], int *k)
{
    return &a[*k];
}

Suppose this does not return a pointer that covers a memory area with `k` but compiler may not have that information.

Pointer arithmetic can make it difficult for a compiler to disprove that `p` does not point to other integer variables such as `k` though it can never point to `i`.
Uniqueness of Addresses

- Aliases are references to the same data via two variables, function arguments, or pointers
- Aliases prevent a compiler from moving and optimizing load/stores
  - Data speculation gives the compiler some optimization opportunities
- Compilers treat potential aliases as a fait accompli
  - Use compiler options
  - Use C/C++ restrict keyword when pointer variables and argument pointers always point to different objects

```c
void copy_dt(dt * restrict p, dt * restrict q, int n) {
    int i;
    for (i = 0; i < n; i++)
        p[i] = q[i];
}
```

Memory accesses cannot overlap
Uniqueness of Addresses (cont’d)

- Compiler options (old gcc versions)
  - gcc -fargument-noalias -fargument-noalias-global
  - suncc -xrestrict
  - icc -restrict

- Use additional compiler options to force *strict aliasing rules* in C/C++ that prohibit the use of pointers of different types to point to the same address
  - gcc -fstrict-aliasing (now the default in gcc, gives warnings)
  - suncc -falias-level=basic

```c
int *p;
double x;
p = (int*)&x; // prohibited
```

Register allocation is effective, because there are no aliases possible: `p`’s pointer dereferences cannot refer to `x`
Dead Code Elimination

\[ i = 0; \]
\[ \text{if } (i \neq 0) \ b = x + y \]

Remove unreachable code

\[ \text{if true goto L2} \]
\[ b = x + y \]
\[ ... \]

Remove useless assignment

\[ b = a + 1 \]
\[ a = b + c \]
\[ ... \]

Assuming \( a \) is dead (has no use in code below)

\[ b = a + 1 \]
\[ ... \]

But what if another process reads \( a \)?
Compilers are \textit{“brain dead”: many optimizations are not multithread-safe!}
Constant Folding and Propagation

- Folds arithmetic operations on constants into a single constant
- Constant assignments are eliminated when possible and the constants are propagated to the register uses

\[\begin{align*}
  r2 &= 1 \\
  \ldots \\
  r5 &= 2 \times r4 \\
  r6 &= r7 / 4 \\
  r2 &= r5 + r6 \\
  r7 &= r2 + 1 \\
  \ldots \\
  r5 &= 2 \times r4 \\
  r6 &= r7 / 4 \\
  r2 &= r5 + r6 \\
  r7 &= 2
\end{align*} \]
Common Subexpression Elimination

- Remove redundant computations
- Increases the number of dependences and therefore may also increase register pressure and hamper scheduling

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= a - d \\
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= b
\end{align*}
\]

\[
\begin{align*}
t1 &= b \times c \\
t2 &= a - t1 \\
t3 &= b \times c \\
t4 &= t2 + t3
\end{align*}
\]

\[
\begin{align*}
t1 &= b \times c \\
t2 &= a - t1 \\
t4 &= t2 + t1
\end{align*}
\]
Strength Reductions

- Replace expensive operations with cheaper ones
  - Replace integer multiplications and divisions with bit shifts
  - Replace multiplications with additions
  - Replace floating point division by multiplication when denominator is a constant

Replace with

\[
\begin{align*}
  r3 &= r4 \times 0.333 \\
  r5 &= r4 + r4 \\
  r6 &= r4 \gg 2
\end{align*}
\]
Many processors have compound floating-point fused multiply and add (fma) instructions and fused negate multiply and add (fnma) instructions.

Compiler options:
- `suncc -fma=fused`
- `icc -IPF-fma`

Example: complex arithmetic
- `x = (xr, xi), y = (yr, yi), z = (zr, zi), u = (ur, ui), compute u = x*y+z`

```
f1 = xr*yr  fma  f1 = xr*yr+zr
f2 = xr*yi  fma  f2 = xr*yi+zi
f3 = xi*yi  fnma ur = -xi*yi+f1
f3 = f1-f3  fma  ui = xr*yr+f2
f4 = xi*yr
f4 = f4+f2
ur = zr+f3
ui = zi+f4
```

*With fma instructions*

```
Without fma
```

1/31/17
Fill Branch Delay Slots

- **Branch delay slots** reduce the penalty of pipeline stalls caused by mispredicted branches

- Compiler fills branch delay slots when optimizing code
  - gcc -fdelayed-branch

- One or more instructions that occur after a branch instruction are always executed
  - These instructions were already fetched in the pipeline and in the IF, ID, and possibly the EX stage

```c
for (i = 0; i < n; i++)
    s = s + 3.0

i = -1
loop:
    s = s + 3.0
    i = i + 1
    if (i<n) jump to loop
    nop ! Branch delay slot

i = -1
loop:
    i = i + 1
    if (i<n) jump to loop
    s = s + 3.0
```
# Single Loop Optimizations

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduce memory instructions</th>
<th>Reduce FP instructions</th>
<th>Reduce integer instructions</th>
<th>Reduce branch instructions</th>
<th>Reduce memory latencies and/or memory bandwidth</th>
<th>Memory latencies</th>
<th>Instruction latencies</th>
<th>Reduce effect of instruction latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Induction variable optimization</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefetching</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Test promotion in loops</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop peeling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Loop fusion</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop fission</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Loop vectorization</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Copying</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Block and copy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Array padding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unrolling</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Software pipelining</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Loop invariant code motion</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimizing reductions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Induction Variable Optimization

- Loop counters and other variables that are iteratively updated are *induction variables* (iv)
- a) *Loop strength reduction* replaces expressions by one or more iv
- b) *Induction variable elimination* removes redundant iv

```plaintext
DO i = 0,n,2
   IA(i) = i*k+m
ENDDO

ic = m
is = 2*k
DO i = 0,n,2
   IA(i) = ic
   ic = ic+is
ENDDO
```

```plaintext
j = 1
DO i = 0,n
   IA(j) = IB(i)
   j = j+1
ENDDO

DO i = 0,n
   IA(i+1) = IB(i)
ENDDO
```
Prefetching

- **Hardware prefetching**: the processor detects a sequential, possibly up/down strided access pattern to memory and prefetches D-cache lines.
- **Software prefetching**: specialized instruction inserted by the compiler or added by programmer (see example) to load D-cache line, or to load specific choice of caches.
  - gcc -fprefetch-loop-arrays
  - suncc -xprefetch=auto
  - icc -prefetch (IA64 only)

```c
for (i = 0; i < n; i++)
    a[i] = (b[i] + 1) * 2;
```

*Original loop*

```c
#include <sun_prefetch.h>
static double a[N], b[N];
int i, m = (n/8) * 8;
for (i = 0; i < m; i += 8) {
    sun_prefetch_read_many(&b[i]+256);
    sun_prefetch_write_many(&a[i]+256);
    a[i] = (b[i] + 1) * 2;
    a[i+1] = (b[i+1] + 1) * 2;
    a[i+2] = (b[i+2] + 1) * 2;
    a[i+3] = (b[i+3] + 1) * 2;
    a[i+4] = (b[i+4] + 1) * 2;
    a[i+5] = (b[i+5] + 1) * 2;
    a[i+6] = (b[i+6] + 1) * 2;
    a[i+7] = (b[i+7] + 1) * 2;
}
```

*New code with prefetch*
Test Promotion in Loops

Test promotion, also called loop unswitching, moves a loop-independent test out of the loop and duplicates the loop body.

Removes branches that can otherwise interfere with pipelining, loop parallelization and vectorization.

```fortran
DO I = 1,N
  IF (A .GT. 0) THEN
    X(I) = X(I) + 1
  ELSE
    X(I) = 0
  ENDIF
ENDDO
```

```fortran
IF (A .GT. 0) THEN
  DO I = 1,N
    X(I) = X(I) + 1
  ENDDO
ELSE
  DO I = 1,N
    X(I) = 0
  ENDDO
ENDIF
```
Loop Peeling

- Some numerical codes include loops that handle boundary conditions in the first and/or last iteration.
- Loop peeling takes the first/last iteration out of the loop and duplicates the loop body for these cases.

```
DO I = 1,N
   IF (I .EQ. 1) THEN
      X(I) = 0
   ELSEIF (I .EQ. N) THEN
      X(I) = N
   ELSE
      X(I) = X(I) + Y(I)
   ENDIF
ENDDO
```

```
X(1) = 0
DO I = 2,N-1
   X(I) = X(I) + Y(I)
ENDDO
X(N) = N
```

*First and last iteration peeled, assuming N>2*
To understand dependences in loops, think about the code in unrolled form

```c
for (I=3; I<7; I++) {
    S1:  A[I] = B[I];
    S2:  C[I] = A[I-2];
}
```

- Flow dependence from \( S_1[3] \) to \( S_2[5] \)
- Flow dependence from \( S_1[4] \) to \( S_2[6] \)
Dependence in Loops

The statement instances $S_1[i]$ for iterations $i = 1, \ldots, N$ represent the loop execution.

We have the following flow dependences:

- $S_1[1] \delta S_1[2]
- S_1[2] \delta S_1[3]
- \ldots
- S_1[N-1] \delta S_1[N]

Statement instances with flow dependences

**Dependence graph**
Loop Fusion/Merge

- Fuse (merge) two loops into one by merging the loop bodies
  - Checks for array-based dependences between loops: if a dependence exists then loop fusion may not be possible
  - Resulting loop must not be too large (may lead to register spill)

```c
for (i = 0; i < n; i++)
    temp[i] = x[i] * y[i];
for (i = 0; i < n; i++)
    z[i] = w[i] + temp[i];
```

```c
DO I = 2, N
    B(I) = T(I) * X(I)
ENDDO
DO I = 2, N
    A(I) = B(I) - B(I-1)
ENDDO
```
Loop Vectorization

- A single-statement loop that carries no dependence can be vectorized.

```fortran
DO I = 1, 4
    S1 X(I) = X(I) + C
ENDDO
```

Fortran 90 array statement

$$S_1 \mathbf{X(1:4)} = \mathbf{X(1:4)} + \mathbf{C}$$

Vector operation

$$\begin{pmatrix} X(1) & X(2) & X(3) & X(4) \\ \end{pmatrix} + \begin{pmatrix} C & C & C & C & C \end{pmatrix} = \begin{pmatrix} X(1) & X(2) & X(3) & X(4) \end{pmatrix}$$
Loop Vectorization (cont’d)

- Because only single loop statements can be vectorized, loops with multiple statements must be transformed using the *loop fission (or loop distribution)* transformation.
Loop Vectorization (cont’d)

- When a loop has backward flow dependences and no loop-independent dependences, interchange the statements to enable loop distribution.

```
DO I = 1, N
S1 D(I) = A(I) + E
S2 A(I+1) = B(I) + C
ENDDO

DO I = 1, N
S2 A(I+1) = B(I) + C
ENDDO
DO I = 1, N
S1 D(I) = A(I) + E
ENDDO
```

```
S2 A(2:N+1) = B(1:N) + C
S1 D(1:N) = A(1:N) + E
```
General Loop Fission Method

- Loop fission (loop distribution) splits a loop into two or more loops.
- Compiler computes the acyclic condensation of the dependence graph to find a legal order of the split loops.

```
S_1 DO I = 1, 10
S_2 A(I) = A(I) + B(I-1)
S_3 B(I) = C(I-1)*X + Z
S_4 C(I) = 1/B(I)
S_5 D(I) = sqrt(C(I))
S_6 ENDDO
```

```
S_1 DO I = 1, 10
S_2 A(I) = A(I) + B(I-1)
S_3 B(I) = C(I-1)*X + Z
S_4 C(I) = 1/B(I)
S_5 D(I) = sqrt(C(I))
S_x ENDDO
S_y DO I = 1, 10
S_z ENDDO
S_u DO I = 1, 10
S_v ENDDO
```
Scalar Expansion

- Breaks anti-dependence relations by expanding or promoting a scalar into an array
- Scalar anti-dependence relations prevent certain loop transformations such as loop fission and loop interchange
Copying

- Copy array to a new “safe” location to avoid cache thrashing when two or more arrays in a loop map to the same cache line (and level of cache associativity is low)

```c
double X[N], Y[N];
for (I=0; I<N; I++)
    Y[I] = Y[I] + X[I];
```

*Original loop suffering from cache thrashing*
Block and Copy

- Use blocking to reduce memory storage overhead when arrays are large and much larger than the cache.

```c
double X[N], Y[N];
for (I=0; I<N; I++)
    Y[I] = Y[I] + X[I];
```

*Original loop suffering from cache thrashing*
Array Padding

- Memory is organized in banks that are a power of two in size
- Avoid allocating arrays that are a power of two in size
- Array padding: when two or more arrays share cache lines, then padding the array with additional leading elements may help to move the start of the array access up
- Simpler approach than block-and-copy

Before padding

After padding
Loop Unrolling

- Unroll loop by a constant, usually between 2 and 8 iterations
  - gcc -funroll-loops -funroll-all-loops -fvariable-expansion-in-unroller
  - suncc -unroll=n
  - icc -unroll

DO I=1,N
  Y(I) = Y(I) + X(I)
ENDDO

DO I=1,N,4
  Y(I)   = Y(I)   + X(I)
  Y(I+1) = Y(I+1) + X(I+1)
  Y(I+2) = Y(I+2) + X(I+2)
  Y(I+3) = Y(I+3) + X(I+3)
ENDDO

Unrolled by compiler
assuming N-1 is multiple of 4
Software Pipelining with Rotating Registers

- Increase the dependence height by increasing the distance between load/stores
- Registers are renumbered by the processor with each iteration of the loop
- This is a form of *software pipelining*
  - Register values stay alive for a fixed iteration window thereby increase the distance between load/store dependences
  - Dependences are overlapped in each iteration and spread across the loop iterations

```c
DO I = 0,N
  Y(I) = X(I)
ENDDO

DO I = 0,N
  load X(I) into r0
  store r0 to Y(I)
ENDDO

load X(0) into r3
load X(1) into r2
load X(2) into r1
DO I = 0, N-3
  load X(I+3) into r0
  store r3 to Y(I)
  rotate r0 to r3
ENDDO
store r2 into Y(N-2)
store r1 into Y(N-1)
store r0 into Y(N)
```
Modulo Scheduling

- A form of software pipelining that copies the chunk of instructions of the loop body to schedule them in parallel in each iteration of the new loop.
- Uses a rotating register file.
- Consider the loop:

\[
\text{DO } i = 0, 6 \\
\text{A} \\
\text{B} \\
\text{C} \\
\text{D} \\
\text{E} \\
\text{F} \\
\text{G} \\
\text{H} \\
\text{ENDDO}
\]
Loop Invariant Code Motion

- A *loop-invariant expression* does not change its value from one iteration to another
- *Loop-invariant code motion* moves loop-invariant code out of the loop to be executed only once
  - *Hoisting*: moves code before the loop
  - *Sinking*: moves code after the loop

```fortran
DO I=1,N
  Y(J) = Y(J) + 2 * K * X(I)
ENDDO

IF (N .GE. 1)
  t1 = Y(J)
  t2 = 2*K
ENDIF
DO I=1,N
  t1 = t1 + t2 * X(I)
ENDDO
IF (N .GE. 1)
  Y(J) = t1
ENDIF
```

Load and store from same address repeatedly
Optimizing Reductions

- Optimize reductions: rewrite reduction loops or use library calls to highly optimized code or parallelize the reduction
- Not all compilers support this kind of optimization

```
a = 0.0
DO i=1,n
   a = a + X(i)
ENDDO
```

```
a = 0.0
b = 0.0
c = 0.0
d = 0.0
DO i=1,n,4
   a = a + X(i)
   b = b + X(i+1)
   c = c + X(i+2)
   d = d + X(i+3)
ENDDO
a = a + b + c + d
```

*Unrolled and optimized assuming n-1 is multiple of 4*
# Nested Loop Optimizations

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduce memory instructions</th>
<th>Reduce FP instructions</th>
<th>Reduce integer instructions</th>
<th>Reduce branch instructions</th>
<th>Reduce memory latencies and/or memory bandwidth</th>
<th>Reduce instruction latencies</th>
<th>Reduce effect of instruction latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop interchange</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Loop parallelization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Outer loop unrolling</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Unroll and jam</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Blocking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Block and copy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Iteration Vector

Definition

Given a nest of $n$ loops, the iteration vector $i$ of a particular iteration of the innermost loop is a vector of integers

$$i = (i_1, i_2, \ldots, i_n)$$

where $i_k$ represents the iteration number for the loop at nesting level $k$

The set of all possible iteration vectors spans an iteration space over loop statement instances $S_j[i]$
Iteration Vector Example

The loop iteration space of the statement at $S_1$ is the set of iteration vectors $\{(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)\}$

Example: at iteration $i = (2,1)$ statement instance $S_1[i]$ assigns the value of $A(1,2)$ to $A(2,1)$
Iteration Vector Ordering

- The iteration vectors are naturally ordered according to a *lexicographical order*
  - For example, iteration \((1,2)\) precedes \((2,1)\) and \((2,2)\)

- **Definition**

Iteration \(i\) *precedes* iteration \(j\), denoted \(i < j\), iff

1) \(i[1:n-1] < j[1:n-1]\), or

2) \(i[1:n-1] = j[1:n-1]\) and \(i_n < j_n\)
Cross-Iteration Dependence

Definition

There exist a dependence from $S_1$ to $S_2$ in a loop nest iff there exist two iteration vectors $i$ and $j$ such that

1) $i < j$ and there is a path from $S_1$ to $S_2$

2) $S_1$ accesses memory location $M$ on iteration $i$ and $S_2$ accesses memory location $M$ on iteration $j$

3) one of these accesses is a write (RAW, WAR, WAW)
**Dependence Example**

Show that the loop has no cross-iteration dependence

Answer: there are no iteration vectors \( i \) and \( j \) in \{(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)\} such that \( i < j \) and

- \( S_1 \) in \( i \) writes to the same element of \( A \) that is read at \( S_1 \) in iteration \( j \)
- or \( S_1 \) in iteration \( i \) reads an element \( A \) that is written at \( S_1 \) in iteration \( j \)

**Loop iteration space**

```plaintext
DO I = 1, 3
  DO J = 1, I
    S1 A(I,J) = A(J,I)
  ENDDO
ENDDO
```
Dependence Distance and Direction Vectors

Definition

Given a dependence from $S_1$ on iteration $i$ to $S_2$ on iteration $j$, the *dependence distance vector* $d(i, j)$ is defined as $d(i, j) = j - i$

Given a dependence from $S_1$ on iteration $i$ and $S_2$ on iteration $j$, the *dependence direction vector* $D(i, j)$ is defined for the $k^{th}$ component as

$$D(i, j)_k = \begin{cases} 
"<" & \text{if } d(i, j)_k > 0 \\
"=\" & \text{if } d(i, j)_k = 0 \\
">\" & \text{if } d(i, j)_k < 0 
\end{cases}$$
Direction of Dependence

- There is a flow dependence if the iteration vector of the write is **lexicographically less than** the iteration vector of the read.
- In other words, if the direction vector is **lexicographically positive**

$$D(i, j)_k = (=, =, ..., =, <, ...)$$

All ‘=’   Any ‘<’, ‘=’, or ‘>’
Representing Dependences with Data Dependence Graphs

It is generally infeasible to represent all data dependences that arise in a program.

Usually only static data dependences are recorded.

- \( S_1 \delta(=) S_2 \) means \( S_1[i] \delta S_2[i] \) for all \( i = 1, \ldots, 10000 \)
- \( S_2 \delta(<) S_2 \) means \( S_1[i] \delta S_2[j] \) for all \( i,j = 1, \ldots, 10000 \) with \( i < j \)

DO I = 1, 10000
\[ S_1 A(I) = B(I) \times 5 \]
\[ S_2 C(I+1) = C(I) + A(I) \]
ENDDO

Static data dependences for accesses to \( A \) and \( C \):
- \( S_1 \delta(=) S_2 \) and \( S_2 \delta(<) S_2 \)

\[ \begin{array}{c}
S_1 \\
\downarrow \delta(=) \\
S_2
\end{array} \quad \begin{array}{c}
S_2 \\
\downarrow \delta(<)
\end{array} \]

\( S_1 \delta(=) S_2 \) and \( S_2 \delta(<) S_2 \)

Data dependence graph
Example 1

DO I = 1, 3
  DO J = 1, I
    S_1 A(I+1,J) = A(I,J)
  ENDDO
ENDDO

Flow dependence between $S_1$ and itself on:
$i = (1,1)$ and $j = (2,1)$: $d(i,j) = (1,0)$, $D(i,j) = (<, =)$
$i = (2,1)$ and $j = (3,1)$: $d(i,j) = (1,0)$, $D(i,j) = (<, =)$
$i = (2,2)$ and $j = (3,2)$: $d(i,j) = (1,0)$, $D(i,j) = (<, =)$
Example 2

DO I = 1, 4
    DO J = 1, 4
    S_1 \ A(I,J+1) = A(I-1,J)
    ENDDO
ENDDO

Distance vector is (1,1)

S_1 \ \delta_{(\langle,\langle)} \ S_1
Example 3

DO I = 1, 4
  DO J = 1, 5-I
    $S_1 \ A(I+1,J+I-1) = A(I,J+I-1)$
  ENDDO
ENDDO

Distance vector is (1, -1)

$S_1 \ \delta_{(<,>)} \ S_1$
Example 4

Distance vectors are (0,1) and (1,0)

\[ S_1 \delta (=,<) S_1 \]
\[ S_2 \delta (<,=) S_1 \]
\[ S_2 \delta (<,=) S_2 \]
Loop Interchange

- *Loop interchange* exchanges outer with inner loop
- Checks for array-based dependences loops: if a dependence exists then loop interchange may not be possible
- May require loop fusion of inner loops first

```
DO I = 1, N
    DO J = 1, M
        A(I,J) = A(I,J-1) + B(I,J)
    ENDDO
ENDDO

DO J = 1, M
    DO I = 1, N
        A(I,J) = A(I,J-1) + B(I,J)
    ENDDO
ENDDO
```

*Note: Fortran uses column-major storage while C uses row-major storage*
Loop Interchange

- Compute the direction matrix and find which columns can be permuted without violating dependence relations in original loop nest (pair wise rows maintain lexicographical order)

```
S_1 DO I = 1, N
S_2 DO J = 1, M
S_3 DO K = 1, L
S_4 A(I+1,J+1,K) = A(I,J,K) + A(I,J+1,K+1)
S_5 ENDDO
S_6 ENDDO
S_7 ENDDO
```

Direction matrix

\[
\begin{pmatrix}
< & < & = \\
< & = & > \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
< & < & = \\
< & = & > \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
< & < & = \\
< & = & > \\
\end{pmatrix}
\]

Invalid 😞

\[
\begin{pmatrix}
< & < & = \\
< & = & > \\
\end{pmatrix}
\]

Valid 😊
Loop Parallelization

- It is valid to convert a sequential outermost loop to a parallel loop if the loop carries no flow dependence (anti dependences can be removed by copying)

```
DO I = 1, 4
  DO J = 1, 4
    A(I,J+1) = A(I,J)
  ENDDO
ENDDO
```

```
PARALLEL DO I = 1, 4
  DO J = 1, 4
    A(I,J+1) = A(I,J)
  ENDDO
ENDDO
```
Loop Parallelization

- It is valid to convert a sequential inner loop to a parallel loop if the outer loop(s) carries all dependences.

```plaintext
DO I = 1, 4
  DO J = 1, 4
    A(I,J+1) = A(I-1,J)
  ENDDO
ENDDO

parallelize

DO I = 1, 4
  PARALLEL DO J = 1, 4
    A(I,J+1) = A(I-1,J)
  ENDDO
ENDDO
```
Outer Loop Unrolling

- Unrolls the outer loop, usually followed by fusion of inner loops
- Reduces branches and may reduce memory operations

Unrolled twice and optimized assuming \( N \) is multiple of 2

```
DO J = 1, N, 2
  DO I = 1, M
    A(I,J) = A(I,J) + X(I) * Y(J)
  ENDDO
  DO I = 1, M
    A(I,J+1) = A(I,J+1) + X(I) * Y(J+1)
  ENDDO
ENDDO
```

**Need to load** \( X[I] \) **only once**
**Unroll and Jam**

- *Unroll and jam* refers to unrolling inner and outer loops and jamming them back together in ways that aim to reduce the number of memory operations.

```
DO K = 1, N, 2
  DO J = 1, N, 2
    DO I = 1, N
      C(I,K) = C(I,K) + A(I,J) * B(J,K)
    ENDDO
    C(I,K+1) = C(I,K+1) + A(I,J) * B(J+1,K)
  ENDDO
  DO J = 1, N, 2
    DO I = 1, N
      C(I,K+1) = C(I,K+1) + A(I,J) * B(J+1,K+1)
    ENDDO
  ENDDO
ENDDO
```

*Step 1: unroll outer loops twice*
Unroll and Jam (cont’d)

Step 2: jam the inner loops back together with loop fusion and combine assignments

```
DO K = 1, N, 2
  DO J = 1, N, 2
    DO I = 1, N
      C(I,K) = C(I,K) + A(I,J) * B(J,K)
      C(I,K) = C(I,K) + A(I,J+1) * B(J+1,K)
      C(I,K+1) = C(I,K+1) + A(I,J) * B(J,K+1)
      C(I,K+1) = C(I,K+1) + A(I,J+1) * B(J+1,K+1)
    ENDDO
  ENDDO
ENDDO
```

```
DO K = 1, N, 2
  DO J = 1, N, 2
    DO I = 1, N
      C(I,K) = C(I,K) + A(I,J) * B(J,K) + A(I,J+1) * B(J+1,K)
      C(I,K+1) = C(I,K+1) + A(I,J) * B(J,K+1) + A(I,J+1) * B(J+1,K+1)
    ENDDO
  ENDDO
ENDDO
ENDDO
```
Loop Blocking/Tiling

- Blocking aims to decrease cache misses in nested loops

```fortran
DO J = 1,N
  DO I = 1,N
    Y(I) = Y(I) + A(I,J)
  ENDDO
ENDDO

NBLOCK = 1000
DO IOUTER = 1,N,BLOCK
  DO J = 1,N
    DO I = IOUTER,MIN(N,N+BLOCK-1)
      Y(I) = Y(I) + A(I,J)
    ENDDO
  ENDDO
ENDDO
```
Block and Copy

- Combination of blocking with array copying

```plaintext
DO K = 1,N
  DO J = 1,N
    DO I = 1,N
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO

DO I = 1,N,NBLOCK
  DO J = 1,N,NBLOCK
    ! Copy C block at (I,J) to T3
    DO K = 1,N,NBLOCK
      ! Copy A block at (I,K) to T1
      ! Copy B block at (K,J) to T2
      CALL MATMUL(NBLOCK,NBLOCK,T1,T2,T3)
      ENDDO
    ! Copy T3 to C block at (I,J)
    ENDDO
  ENDDO
ENDDO

Perform A×B in blocks where each block multiply uses a local temporary copy of A, B, and C that fit in cache.
```
Dependence System: Formulating Flow Dependence

Assuming \( n \) normalized loop iteration spaces

A dependence system consists of a dependence equation along with a set of constraints:

- Solution must lie within loop bounds
- Solution must be integer
- Need dependence distance or direction vector (flow/anti)

\[
\begin{align*}
0 & \leq \alpha_i, \beta_i \leq 98 \\
0 & \leq \alpha_j \leq \alpha_i \\
0 & \leq \beta_j \leq \beta_i \\
\alpha_i & < \beta_i
\end{align*}
\]

\[
\begin{align*}
\alpha_i + 2 &= \beta_j + 1 \\
\alpha_j + 1 &= \beta_i + 2 \\
\end{align*}
\]

\[ S_1 \quad A(i+2,j+1) = A(j+1,i+2) \]

\[ \text{Dependence equations} \]

\[ \text{Loop constraints} \]

\[ \text{Constraint for (<,*) dep. direction} \]
Dependence System: Matrix Notation

**Dependence equations**

\[
\begin{bmatrix}
1 & 0 & 0 & -1 \\
0 & -1 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
\alpha_I \\
\beta_I \\
\alpha_J \\
\beta_J
\end{bmatrix}
= 
\begin{bmatrix}
-1 \\
1
\end{bmatrix}
\]

- The polyhedral model: solving a linear (affine) dependence system in matrix notation
  - Polyhedral \{ x : Ax \leq c \} for some matrix A and bounds vector c
  - A contains the loop constraints (inequalities)
  - Rewrite dependence equation as set of inequalities: \( Ax = b \Rightarrow Ax \leq b \) and \(-Ax \leq -b\)

- If no point lies in the polyhedral (a polytope) then there is no solution and thus there is no dependence
Fourier-Motzkin Projection

\[
\begin{pmatrix}
0 & 1 \\
1 & 1 \\
1 & -1 \\
-2 & -1 \\
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\end{pmatrix}
\leq
\begin{pmatrix}
6 \\
9 \\
5 \\
-7 \\
\end{pmatrix}
\]

System of linear inequalities

\[Ax \leq b\]

Projections on \(x_1\) and \(x_2\)

If (one of the) projections is empty then there is no solution because the enclosed space is empty

(Equalities are rewritten to inequalities and added to the system)
Fourier-Motzkin Variable Elimination (FMVE)

**FMVE procedure:**
1. Select an unknown $x_j$
2. $L = \{i \mid a_{ij} < 0\}$
3. $U = \{i \mid a_{ij} > 0\}$
4. if $L=\emptyset$ or $U=\emptyset$ then $x_j$ is unconstrained (delete it)
5. for $i \in L \cup U$
   \[ A_{[i]} := A_{[i]} / |a_{ij}| \]
   \[ b_i := b_i / |a_{ij}| \]
6. for $i \in L$
   for $k \in U$
   add new inequality
   \[ A_{[i]} + A_{[k]} \leq b_i + b_k \]
7. Delete old rows $L$ and $U$

Select $x_2$: $L = \{3,4\}$, $U = \{1,2\}$
new system:

\[
\begin{bmatrix}
0 & 1 & \vdots & 6 \\
1 & 1 & \vdots & 9 \\
1 & -1 & \vdots & 5 \\
-2 & -1 & \vdots & -7
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2
\end{bmatrix}
\leq
\begin{bmatrix}
1 \\
9 \\
5 \\
-7
\end{bmatrix}
\]

$\max(1/2,-2) \leq x_1 \leq \min(11,7)$
Unimodular Loop Transformations

- A unimodular matrix $U$ is a matrix with integer entries and determinant $\pm 1$
- Such a matrix maps an object onto another object with exactly the same number of integer points in it
- The set of all unimodular transformations forms a group, called the modular group
  - The identity matrix is the identity element
  - The inverse $U^{-1}$ of $U$ exist and is also unimodular
  - Unimodular transformations are closed under multiplication: a composition of unimodular transformations is unimodular
Unimodular Loop Transformations (cont’d)

Many loop transformations can be represented as matrix operations using unimodular matrices, including loop interchange, loop reversal, loop interchange, …

\[
U_{\text{interchange}} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}
\]

\[
U_{\text{reversal}} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}
\]

\[
U_{\text{skewing}} = \begin{pmatrix} 1 & 0 \\ -p & 1 \end{pmatrix}
\]

DO K = 1, 100
DO L = 1, 50
A(K,L) = …
ENDDO
ENDDO

DO K = 1, 100
DO L = 1, 100
A(L,K) = …
ENDDO
ENDDO

DO K = -100, -1
DO L = 1, 50
A(-K,L) = …
ENDDO
ENDDO

DO K = 1, 100
DO L = 1+p*K, 50+p*K
A(K,L-p*K) = …
ENDDO
ENDDO
Unimodular Loop Transformations (cont’d)

The unimodular transformation is also applied to the dependence vectors: if the result is lexicographically positive, the transformation is legal.

\[
\begin{align*}
&\text{DO } K = 2, N \\
&\quad \text{DO } L = 1, N-1 \\
&\quad \quad A(L,K) = A(L+1,K-1) \\
&\quad \text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
U_{\text{interchange}} \quad d &= d_{\text{new}} \\
\begin{pmatrix}
0 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
1 \\
-1
\end{pmatrix}
&= 
\begin{pmatrix}
-1 \\
1
\end{pmatrix}
\end{align*}
\]

Loop has distance vector \( d(i,j) = (1,-1) \) for the writes at iterations \( i=(i_1,i_2) \) and reads at iterations \( j=(j_1,j_2) \).
Unimodular Loop Transformations (cont’d)

- Transformed loop nest is given by $AU^{-1}I \leq b$
  - The new loop body has indices $I' = U^{-1}I$
  - Iteration space $IS = \{I | AI \leq b\}$ after transformation $IS' = \{I' | AU^{-1}I' \leq b\}$

- Transformed loop nest needs to be normalized by means of Fourier-Motzkin elimination to ensure that loop bounds are affine expressions in more outer loop indices

$$\begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix} \leq \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} M \\ N \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} K \\ L \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \end{bmatrix} \leq \begin{bmatrix} 3 \\ -1 \\ -1 \end{bmatrix}$$

$$FMVE: \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} L \\ K \end{bmatrix} \leq \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix}$$

$$2 \leq K \leq 4$$

$$1 \leq L \leq 3 \text{ and } L \leq K-1$$
Global Optimizations

- **Interprocedural analysis (IPA)** determines properties of a function code to help optimize caller’s code
  - For example, aliases can be resolved by determining which variables and parameters are actually changed by the function

- **Interprocedural optimizations (IPO)**
  - *Inlining* (**inline** keyword in C/C++) expands a function in-line to avoid function calling overhead and to enable optimizations on the function’s body in the current code region
    - gcc -finline-functions
    - suncc -xinline=auto
    - icc -finline -finline-functions
  - *Cloning* creates specialized copies of functions that are optimized based on actual parameter values
Practical Suggestions

- Split code in compute-intensive part and the rest (such as file access)
- Write clear code to help compiler detect opportunities for optimization
- Avoid bulky loops
- Use regular data structures whenever possible and set them up so that they are accessed element by element in loops to improve spatial locality
- Avoid global data (statically allocated data)
- Keep branch conditions simple
Further Reading

- [HPC] pages 81-124