VECTORIZATION SEMINAR
VECTORIZATION

Techniques used

- Simple vectorization
- Strip mining
- Loop distribution
- Loop interchange
THE

FORTRAN

COMPILER
MAJOR FEATURES

- Conforms to ANSI Fortran-77 Standard (ANSI X3.9-1978)

- Optional Support of Fortran-66 features (ANSI X3.9-1966)

- VAX/VMS compatibility

- Interfaces to the symbolic debugger, CSD

- Interfaces to the performance analyzer

- Integrated into the CONVEX/UNIX environment
CONVEX EXTENSIONS

• Hollerith constants can be used where a character value is expected

• Symbolic names may be of arbitrary length

• Variety of data types
  • Logical *1, *2, *4, *8
  • Integer *1, *2, *4, *8
  • Real *4, *8
  • Complex *8, *16
  • Character *len
Differences C-1/VAX Fortran

Does not support certain VAX extensions

- INCLUDE statement
- NAMELIST statement
- REAL*16 data type
- %DESCR
- Byte ordering w.r.t. characters and parameter passing
- Low order bit test for IFs
- Numerical differences, due to floating-point representation and rounding method
- ‘cc...c’ form of octal constants is typeless
Differences C-1/VAX Fortran

Does not support certain VAX I/O extensions
- DELETE, UNLOCK statements
- NAMELIST directed I/O
- Variable format expressions
- Indexed I/O (key-indexed files)
- File sharing
- DEFINEFILE statement
- Certain OPEN keywords
- Certain CLOSE keywords
- Record specifier ‘r
- ASCII null as carriage control
- RECL keyword on OPENs is in bytes
- Certain internal record formats differ
- SEGMENTED record type
- No RMS related extensions
Differences C-1/VAX Fortran

Miscellaneous

- User takes advantage of implementation
  - mechanism to pass arguments
- User takes advantage of VAX architecture
  - representation of character strings
- No VMS OS specific extensions
  - pathnames
  - calling system services
- No PDP-11 Fortran compatibility
SCALAR

OPTIMIZATION
SCALAR OPTIMIZATION

2 Types of Optimization are Provided

- Local
- Global
SCALAR OPTIMIZATION

- Assignment substitution
- Redundant assignment elimination
- Redundant use elimination
- Redundant subexpression elimination
- Tree height reduction
- Constant propagation and folding
- Dead code elimination
- Code motion
- Strength reduction
- Instruction scheduling
- Branch optimization
- Register allocation
Assignment Substitution

Code:

\[
\begin{align*}
x &= z + 3 \\
d &= x + e
\end{align*}
\]

Becomes:

\[
\begin{align*}
x &= z + 3 \\
d &= (z + 3) + e
\end{align*}
\]
• Assignment Substitution

Unoptimized:

load @ 12(ap),s0 ; # 3, Z
add # 0x41400000,s0 ; # 3
stor s0,@ 8(ap) ; # 3, X
ldor @ 8(ap),s1 ; # 4, X
laod @ 4(ap),s2 ; # 4, E
add s2,s1 ; # 4
stor s1,@ 0(ap) ; # 4, D

Optimized:

load @ 12(ap),s0 ; # 3, Z
load @ 4(ap),s1 ; # 4, E
add # 0x41400000,s0 ; # 3
add s0,s1 ; # 4
stor s0,@ 8(ap) ; # 3, X
stor s1,@ 0(ap) ; # 4, D
• Assignment Substitution

Source Code:

```plaintext
subroutine assgn (d,e,x,z)
real d,e,x,z
x = z + 3
d = x + e
return
end
```

Unoptimized:

```
L2: ; Stmt _L1
   ld.w @ 12(ap),s0 ; # 3, Z
   add.s # 0x41400000,s0 ; # 3
   st.w s0,@ 8(ap) ; # 3, X
   .stabd 0x44,0,4
   ; Stmt _L2
L3:
   ld.w @ 8(ap),s1 ; # 4, X
   ld.w @ 4(ap),s2 ; # 4, E
   add.s s2,s1 ; # 4
   st.w s1,@ 0(ap) ; # 4, D
   .stabd 0x44,0,5
   ; Stmt _L3
L4:
   rtn ; # 5
```
. Assignment Substitution

Source Code:

```plaintext
subroutine assgn (d,e,x,z)
  real d,e,x,z
  x = z + 3
  d = x + e
  return
end
```

Optimized:

```
_assgn_

  ld.w  @ 12(ap),s0  ; # 3, Z
  ld.w  @ 4(ap),s1  ; # 4, E
  add.s  # 0x41400000,s0  ; # 3
  add.s  s0,s1  ; # 4
  st.w  s0,@ 8(ap)  ; # 3, X
  st.w  s1,@ 0(ap)  ; # 4, D
  rtn  ; # 5
```
Redundant Assignment Elimination

Code:

\[
\begin{align*}
    & x = a + b \\
    & x = e + f \times g
\end{align*}
\]

Becomes:

\[
\begin{align*}
    & x = e + f \times g
\end{align*}
\]
Redundant Assignment Elimination

Source Code:

subroutine redasg(x,a,b,e,f,g)
real x,a,b,e,f,g
x = a+b
x = e+f*g
return
end

Unoptimized:

L2: ; Stmt _L1
   ld.w @ 4(ap),s0 ; # 3, A
   ld.w @ 8(ap),s1 ; # 3, B
   add.s s1,s0 ; # 3
   st.w s0,@ 0(ap); # 3, X
   .stabd 0x44,0,4
L3: ; Stmt _L2
   ld.w @ 16(ap),s2 ; # 4, F
   ld.w @ 20(ap),s3 ; # 4, G
   ld.w @ 12(ap),s4 ; # 4, E
   mul.s s3,s2 ; # 4
   add.s s2,s4 ; # 4
   st.w s4,@ 0(ap); # 4, X
   .stabd 0x44,0,5
L4: ; Stmt _L3
   rtn ; # 5
• Redundant Assignment Elimination

Source Code:

```
subroutine redasg(x,a,b,e,f,g)
real x,a,b,e,f,g
x = a+b
x = e+f*g
return
end
```

Optimized:

```
_redasg_:  
ld.w   @ 16(ap),s0 ; # 4, F
ld.w   @ 20(ap),s1 ; # 4, G
ld.w   @ 12(ap),s2 ; # 4, E
mul.s  s1,s0
add.s  s0,s2
st.w   s2,@ 0(ap) ; # 4, X
rtn    ; # 5
```
• Redundant Assignment Elimination

Source:

```plaintext
program test
x = y*z
a = 0
if(a.gt.0) then
  a = x*y + z
else
  x = a - b*c
endif
end
```

Becomes:

```plaintext
program test
end
```

; INSTRUCTIONS

```
.text
ds.w 0x1040b07
ds.b "-O2"
.globl _MAIN_
.MAIN_
.rtn ; # 9
```
• Redundant Use Elimination

Source Code:

```plaintext
subroutine reduse(x,a,b,c,d)
real x,a,b,c,d
a = x*b
c = x+d
return
end
```

Unoptimized:

```
L2: ; Stmt _L1
    ld.w @ 0(ap),s0 ; # 3, X
    ld.w @ 8(ap),s1 ; # 3, B
    mul.s s1,s0 ; # 3
    st.w s0,@ 4(ap) ; # 3, A
    .stabd 0x44,0,4
L3: ; Stmt _L2
    ld.w @ 0(ap),s2 ; # 4, X
    ld.w @ 16(ap),s3 ; # 4, D
    add.s s3,s2 ; # 4
    st.w s2,@ 12(ap) ; # 4, C
    .stabd 0x44,0,5
L4: ; Stmt _L3
    rtn ; # 5
```
Redundant Use Elimination

Source Code:

```plaintext
subroutine reduse(x,a,b,c,d)
real x,a,b,c,d
a = x*b
c = x+d
return
end
```

Optimized:

```plaintext
_reduse_

ld.w @ 0(ap),s0 ; # 3, X
ld.w @ 16(ap),s1 ; # 4, D
ld.w @ 8(ap),s2 ; # 3, B
add.s s0,s1 ; # 4
mul.s s2,s0 ; # 3
st.w s1,@ 12(ap) ; # 4, C
st.w s0,@ 4(ap) ; # 3, A
rtn ; # 5
```
Redundant Subexpression Elimination

Code:

\[
a = x + y \\
b = c \cdot (x + y)
\]

Becomes:

\[
t = x + y \\
a = t \\
b = c \cdot t
\]
Redundant Subexpression Elimination

Source Code:

```plaintext
subroutine redsub(x,y,a,b,c)
real x,y,a,b,c,d
a = x+y
b = c*(x+y)
return
end
```

Unoptimized:

```plaintext
L2: ; Stmt _L1
    ld.w @ 0(ap),s0 ; # 3, X
    ld.w @ 4(ap),s1 ; # 3, Y
    add.s s1,s0 ; # 3
    st.w s0,@ 8(ap) ; # 3, A
    .stabd 0x44,0,4
L3: ; Stmt _L2
    ld.w @ 0(ap),s2 ; # 4, X
    ld.w @ 4(ap),s3 ; # 4, Y
    ld.w @ 16(ap),s4 ; # 4, C
    add.s s3,s2 ; # 4
    mul.s s2,s4 ; # 4
    st.w s4,@ 12(ap) ; # 4, B
    .stabd 0x44,0,5
L4: ; Stmt _L3
    rtn ; # 5
```
Redundant Subexpression Elimination

Source Code:

```plaintext
subroutine redsub(x,y,a,b,c)
real x,y,a,b,c,d
a = x+y
b = c*(x+y)
return
end
```

Optimized:

```plaintext
_redsub_

ld.w @ 0(ap),s0 ; # 3, X
ld.w @ 4(ap),s1 ; # 3, Y
ld.w @ 16(ap),s2 ; # 4, C
add.s s1,s0 ; # 3
mul.s s0,s2 ; # 4
st.w s0,@ 8(ap) ; # 3, A
st.w s2,@ 12(ap) ; # 4, B
rtn ; # 5
```
- Constant Propagation and Folding

Code:

```
\texttt{i = 5}
\texttt{j = 0}
\texttt{j = j + 2}
\texttt{n = k + i \times j}
```

Becomes:

```
\texttt{i = 5}
\texttt{j = 0}
\texttt{j = 2}
\texttt{n = k + 10}
```
. Constant Propagation and Folding

Code:

\[ j = 0 \]

(other work occurs, but value of j is not changed)

\[ j = j + 2 \]

\[ n = k + 5 \times j \]

Becomes:

\[ j = 0 \]

(other work occurs, but value of j is not changed)

\[ j = 2 \]

\[ n = k + 10 \]
. Constant Propagation and Folding

Code:

\[
\begin{align*}
    &a = 5 \\
    &b = 15 \\
    &\text{if}(i \leq 0) \text{ then} \\
    &\quad a = 6 \\
    &\quad c = a \\
    &\text{else} \\
    &\quad c = a + b \\
    &\text{endif}
\end{align*}
\]

Becomes:

\[
\begin{align*}
    &a = 5 \\
    &b = 15 \\
    &\text{if}(i \leq 0) \text{ then} \\
    &\quad a = 6 \\
    &\quad c = 6 \\
    &\text{else} \\
    &\quad c = 20 \\
    &\text{endif}
\end{align*}
\]
• **Constant Propagation and Folding**

Source Code:

```fortran
subroutine constp1(n,i,j,k)
i, j, k
i = 5
j = 0
j = j + 2
n = k + i * j
return
end
```

Unoptimized:

```
L2: ; Stmt_L1
  ld.w #0x00000005,s0 ; #3
  st.w s0,@4(ap) ; #3, I
  .stabd 0x44,0,4
L3: ; Stmt_L2
  ld.w #0x00000000,s1 ; #3, J
  st.w s1,@8(ap) ; #4
  .stabd 0x44,0,5
L4: ; Stmt_L3
  ld.w @8(ap),s2 ; #5, J
  add.w #0x00000002,s2 ; #5
  st.w s2,@8(ap) ; #5, J
  .stabd 0x44,0,6
L5: ; Stmt_L4
  ld.w @4(ap),s3 ; #6, I
  ld.w @8(ap),s4 ; #6, J
  ld.w @12(ap),s5 ; #6, K
  mul.w s4,s3 ; #6
  add.w s3,s5 ; #6
  st.w s5,@0(ap) ; #6, N
  .stabd 0x44,0,7
L6: ; Stmt_L5
  rtn ; #7
```
Constant Propagation and Folding

Source Code:

```
subroutine constp1(n,i,j,k)
  integer i,j,k
  i = 5
  j = 0
  j = j + 2
  n = k + i * j
  return
end
```

Optimized:

```
constp1:
  ld.w # 0x0000002, s1 ; # 5
  ld.w # 0x0000005, s0 ; # 3
  ld.w @ 12(ap), s2 ; # 6, K
  add.w # 0x000000a, s2 ; # 6
  st.w s1, @ 8(ap) ; # 5, J
  st.w s0, @ 4(ap) ; # 3, I
  st.w s2, @ 0(ap) ; # 6, N
  rtn ; # 7
```
Tree Height Reduction

Code:

```
(a + (b + (c + (d + (e + (f + (g + h)))))))
```

Usual:

```
(a + (b + (c + (d + (e + (f + (g + h)))))))
```

Becomes:

```
(((a + b) + (c + d)) + ((e + f) + (g + h)))
```
• Tree Height Reduction

Source Code:

subroutine treehgt(x,a,b,c,d,e,f,g,h)
real x,a,b,c,d,e,f,g,h
x=a+b+c+d+e+f+g+h
return
end

Optimized:

_treehgt_

ld.w @ 4(ap),s0 ; #3, A
ld.w @ 8(ap),s1 ; #3, B
ld.w @ 12(ap),s2 ; #3, C
ld.w @ 16(ap),s3 ; #3, D
ld.w @ 20(ap),s4 ; #3, E
ld.w @ 24(ap),s5 ; #3, F
ld.w @ 28(ap),s6 ; #3, G
ld.w @ 32(ap),s7 ; #3, H
add.s s1,s0 ; #3
add.s s3,s2 ; #3
add.s s5,s4 ; #3
add.s s7,s6 ; #3
add.s s2,s0 ; #3
add.s s6,s4 ; #3
add.s s4,s0 ; #3
st.w s0,@ 0(ap) ; #3, X
rtn ; #4
Dead Code Elimination

Code:

```plaintext
logical t
  t = .true.
  if(t) then
    print *,a,b
  else
    x = x*4.0
  endif
end
```

Becomes:

```plaintext
logical t
  print *,a,b
end
```
. Dead Code Elimination

program ddcde
real a,b
logical t
a = 4.3
b = 5.2
t = .true.
if(t) then
  print *,a,b
else
  x = x*4.0
endif
end

; INSTRUCTIONS

.text
ds.w    0x1060000
ds.b    "-O1"
.globl   _MAIN_

_MAIN_:      
  idea    LC+24,ap    ; #8, ?LC
  calls   _for$s_wsle ; #8, for$s_wsle
  idea    LC+44,ap    ; #8, ?LC
  calls   _for$do_lio ; #8, for$do_lio
  idea    LC+76,ap    ; #8, ?LC
  calls   _for$do_lio ; #8, for$do_lio
  idea    LC+96,ap    ; #8, ?LC
  calls   _for$e_wsle ; #8, for$e_wsle
  rtn     ; #0
. Code Motion

Code:

\[
\text{do } i=1,100
\begin{align*}
    a &= b + \frac{(c \times 4)}{(d + c)} \\
    ar(i) &= a + d^{**2}
\end{align*}
\text{enddo}
\]

Becomes:

\[
\text{a} = b + \frac{(c \times 4)}{(d + c)} \\
\text{t1} = a + d^{**2} \\
\text{do } i=1,100 \\
    \text{ar}(i) = \text{t1} \\
\text{enddo}
\]
• Code Motion

Source Code:

```fortran
subroutine motion(a,b,c,d,ar)
dimension ar(100)
do i=1,100
   a = b + (c*4)/(d + c)
   ar(i) = a + d**2
endo
d return
end
```
Code Motion

Unoptimized:

L2:  
    ld.w  @ 0x00000001, s0  #3  
    st.w  s0, LU  #3, I  
    ; Stmt_L1

L3:  
    ld.w  @ 8(ap), s1  #4, C  
    ld.w  @ 12(ap), s3  #4, D  
    ld.w  @ 4(ap), s4  #4, B  
    add.s  s1, s3  #4  
    mov.w  s1, s2  #4  
    mul.s  #0x41800000, s2  #4  
    div.s  s3, s2  #4  
    add.s  s2, s4  #4  
    st.w  s4, @ 0(ap)  #4, A  
    ; Stmt -1

L4:  
    ld.w  @ 12(ap), s5  #5, D  
    ld.w  @ 0(ap), s6  #5, A  
    ld.w  LU, a1  #5, I  
    ld.w  16(ap), a2  #5, AR  
    mul.s  s5, s5  #5  
    shf  #0x00000002, a1  #5  
    add.w  a2, a1  #5  
    add.s  s5, s6  #5  
    st.w  s6, -4(a1)  #5, AR  
    ; Stmt_L2

L5:  
    ld.w  LU, s7  #3, I  
    add.w  #0x00000001, s7  #3  
    st.w  s7, LU  #3, I  
    ; Stmt_L3

L6:  
    ld.w  LU, s0  #3, I  
    lt.w  #0x00000064, s0  #3  
    jbrs.f  L3  #3  
    ; Stmt_L4

L7:  
    rtn  #7  
    ; Stmt_L5
• Code Motion

Optimized:

```assembly
.sub.w #0x0000020,sp ; #1
ld.w #0x000004,s5 ; #5
ld.w #0x000190,s6 ; #5
ld.w @8(ap),s0 ; #0, C
ld.w @4(ap),s3 ; #0, B
ld.w @12(ap),s2 ; #0, D
ld.w @16(ap),a1 ; #5, AR
sub.w s6,s5 ; #3
mov.w s0,s1 ; #4
mul.s #0x41800000,s1 ; #4
add.s s2,s0 ; #4
mov.w s2,s4 ; #5
mul.s s2,s4 ; #5
mov a1,s7 ; #5
sub.w s5,s7 ; #3
div.s s0,s1 ; #4
add.s s1,s3 ; #4
add.s s3,s4 ; #5
st.w a1,-4(fp) ; #5, ?i1
st.w s7,-8(fp) ; #3, ?i2
st.w s3,@0(ap) ; #4, A
st.w s3,-24(fp) ; #0, t4
st.w s4,-32(fp) ; #0, t6

L2: ; Stmt -1
ld.w -4(fp),a2 ; #5, ?i1
ld.w -8(fp),a4 ; #3, ?i2
ld.w -32(fp),s0 ; #0, t6

L3: mov a2,a3 ; #5
add.w #0x0000004,a2 ; #5
le.w a2,a4 ; #3
st.w s0,0(a3) ; #5, AR
jbra.t L3 ; #3
st.w a2,-4(fp) ; #5, ?i1
rtn ; #7
```
• Strength Reduction

Code:

```
i = 1
10  j = i*k
    i = i+2
    if(i.le.100) go to 10
```

Becomes:

```
t1 = k
    t2 = 100*k
    t3 = 2*k
10  j = t1
    t1 = t1+t3
    if(t1.le.t2) go to 10
```
• Strength Reduction

Source Code:

```plaintext
subroutine gstren(j,k)
i = 1
10   j = i*k
     i = i+2
    if(i.le.100) go to 10
    return
end
```
. Strength Reduction

; INSTRUCTIONS

.text
  ds.w 0x1040b07
  ds.b "-O2"
  .globl _gstren_

_gstren_: 

  sub.w #0x0000010,sp ;#1
  ld.w @4(ap),s0 ;#3, K
  mov.w s0,s1 ;#0
  mul.w #0x0000064,s1 ;#0
  st.w s0,-4(fp) ;#0, t1
  st.w s1,-8(fp) ;#0, t2
  shf #0x0000001,s0 ;#0
  st.w s0,-12(fp) ;#0, t3

M2: ; Stmt 10
  ld.w -8(fp),s4 ;#0, t2

M3: 

  ld.w -12(fp),s3 ;#0, t3
  ld.w -4(fp),s2 ;#0, t1
  add.w s2,s3 ;#0
  le.w s3,s4 ;#5
  st.w s2,@0(ap) ;#3, J
  st.w s3,-4(fp) ;#0, t1
  jbrs.t M3 ;#5
  rtn ;#6
. Strength Reduction

Source Code:

```plaintext
subroutine gstren(j,k)
  i = 1
  j = i*k
  i = i + 2
  if(i.le.100) go to 10
  return
end
```

. Strength Reduction

Unoptimized:

L2:
  ld.w #0x0000001,s0  ; #2
  st.w s0,LU  ; #2, I
  .stabd 0x44,0,3

L3:
  ld.w LU,s1  ; #3, I
  ld.w @4(ap),s2  ; #3, K
  mul.w s2,s1  ; #3
  st.w s1,@0(ap)  ; #3, J
  .stabd 0x44,0,4

L4:
  ld.w LU,s3  ; #4, I
  add.w #0x0000002,s3  ; #4
  st.w s3,LU  ; #4, I
  .stabd 0x44,0,5

L5:
  ld.w LU,s4  ; #5, I
  it.w #0x0000064,s4  ; #5
  jbrs.r L3  ; #5
  .stabd 0x44,0,6

L6:
  rtn  ; #6
· Strength Reduction

Optimized:

```assembly
 gstren:
    sub.w #0x0000008,sp ; #1
    ld.w #0x0000001,s0 ; #2
    ld.w @4(ap),s1 ; #0, K
    st.w s0,LU ; #2, I
    st.w s1,-8(fp) ; #0, #12
    shf #0x0000001,s1 ; #3
    st.w s1,-4(fp) ; #3, #11

L2:
    ld.w LU,s2 ; #3, I
    ld.w -8(fp),s3 ; #0, #12
    ld.w -4(fp),s4 ; #3, #11

L3:
    add.w #0x0000002,s2 ; #4
    st.w s3,@0(ap) ; #3, J
    add.w s4,s3 ; #0
    lt.w #0x0000064,s2 ; #5
    jbrs.f L3 ; #5
    st.w s3,-8(fp) ; #0, #12
    st.w s2,LU ; #4, I
    rtn ; #6
```
VECTORORIZATION
Advantages of Vector Processing

- Eliminates overhead associated with loop control

- Reduces loops to a simple sequence of instructions
Vector Stride

- Contiguous or Unity
  
  ```
  DO 10 I = 1, 6
      A(I) = B(I)
  10 CONTINUE
  ```

- Constant

  ```
  DO 10 I = 1, 6, 2
      A(I) = B(I)
  10 CONTINUE
  ```

- Random

  ```
  DO 10 I = 1, 100
      A(I) = B(INDEX(I))
  10 CONTINUE
  ```
VECTORORIZATION CAPABILITIES

• Simple loops
• Strip mining
• Loop interchange
• Loop distribution
• Recognize reduction operators
• Generate vector of indices
• Perform scalar expansion
• Vectorize conditionals
• Partial vectorization
• Detect loop iteration count
• Accept all data types
Simple Vectorization

Source:

```fortran
  do 10 i = 1, 100
    a(i) = b(i) + c(i)
  10 continue
```
Scalar Instructions - Full Optimization

_simple_

sub.w  # 0x0000008,sp ; # 2
ld.w   # 0x0000000,s0 ; # 5
st.w   s0,-4(fp) ; # 5, .?i1
M2:
    ; Stmt -1
    ld.w -4(fp),a1 ; # 5, .?i1
M3:
    ld.w  4(ap),a3 ; # 5, B
    ld.w  8(ap),a4 ; # 5, C
    ld.w  0(ap),a5 ; # 5, A
    mov   a1,a2 ; # 5
    add.w a2,a3 ; # 5
    ld.l  0(a3),s1 ; # 5, B
    add.w a2,a4 ; # 5
    ld.l  0(a4),s2 ; # 5, C
    add.w # 0x00000008,a1 ; # 5
    add.w a5,a2 ; # 5
    add.d s2,s1 ; # 5
    lt.w  # 0x0000318,a1 ; # 4
    st.l  s1,0(a2) ; # 5, A
    jbra.f M3 ; # 4
    st.w  a1,-4(fp) ; # 5, .?i1
rtn ; # 7
Vector Instructions

_simple_

    ld.w  # 0x00000064,v1  ; # 5
    ld.w  # 0x00000008,vs ; # 5
    ld.w  4(ap),a1        ; # 5, B
    ld.w  8(ap),a2        ; # 5, C
    ld.w  0(ap),a3        ; # 5, A
    ld.l  0(a1),v0        ; # 5, B
    ld.l  0(a2),v1        ; # 5, C
    add.d v0,v1,v2        ; # 5
    st.l  v2,0(a3)        ; # 5, A
    rtn               ; # 7
Simple Vectorization

Source:

```fortran
subroutine simp(a,b,c)
real a(100),b(100),c(100)
do 10 i = 1, 100
   a(i) = b(i) + c(i)
10  continue
return
end
```

Compiled:

```fortran
subroutine simp(a,b,c)
real a(100),b(100),c(100)
C##3 [fc] Loop on line 3 of t1.f (DO I) fully vectorized%
   do 10 i = 1, 100
      a(i) = b(i) + c(i)
10  continue
return
end
```
. Simple Vectorization

Source:

```fortran
subroutine simp(a,b,c,n)
  real a(100),b(100),c(100)
  integer n
  do 10 i = 1, n
    a(i) = b(i) + c(i)
  10 continue
return
end
```

Compiled:

```fortran
subroutine simp(a,b,c,n)
  real a(100),b(100),c(100)
  integer n
C###4 [fc] Loop on line 4 of t1.f (DO I) fully vectorized%%% 
do 10 i = 1, n
  a(i) = b(i) + c(i)
  10 continue
return
end
```
. Simple Vectorization

Source:

```fortran
subroutine simp(a,b,c,n)
  real a(100),b(100),c(100)
  integer n
  do 10 i = 1, n, 2
    a(i) = b(i) + c(i)
  10 continue
  return
end
```

Compiled:

```fortran
subroutine simp(a,b,c,n)
  real a(100),b(100),c(100)
  integer n
  C###4 [fc] Loop on line 4 of t1.f (DO I) fully vectorized%  
  do 10 i = 1, n, 2
    a(i) = b(i) + c(i)
  10 continue
  return
end
```
Strip Mining

Code:

```
do 10 i = 1, n
   a(i) = b(i) * c(i)
10 continue
```

Becomes:

```
j = 0
do 20 lv = n, 0, -128
   do 10 i = 1, min(128,lv)
      a(i+j) = b(i+j) * c(i+j)
10 continue
   j = j + 128
20 continue
```
Loop Interchange

Code:

```plaintext
do 20 i = 1, n
   do 10 j = 1, m
      a(i,j) = b(i,j) * c(i,j)
   10 continue
20 continue

Becomes:

```plaintext
do 20 j = 1, n
   do 10 i = 1, m
      a(i,j) = b(i,j) * c(i,j)
   10 continue
20 continue
```
• Loop Interchange

Source:

```
subroutine t1 (a,b,c,n,m)
real a(n,m), b(n,m), c(n,m)
integer i, j, n, m
do 20 i = 1, n
do 10 j = 1, m
   a(i,j) = b(i,j) * c(i,j)
10 continue
20 continue
return
end
```

Compiled:

```
subroutine t1 (a,b,c,n,m)
real a(n,m), b(n,m), c(n,m)
integer i, j, n, m
C## # 4 [fc] Loop on line 4 of t1.f (DO I) fully vectorized?
C## # 4 [fc] Loop on line 4 of t1.f (DO I) interchanged to
   be innermost loop of nest%
   do 20 i = 1, n
do 10 j = 1, m
   a(i,j) = b(i,j) * c(i,j)
10 continue
20 continue
return
end
```
• Loop Distribution

Code:

```
  do 20 i = 1, n
      b(i,1) = 0
  do 10 j = 1, m
      a(i) = a(i) + b(i,j) * c(i,j)
  10 continue
  20 continue
```

Becomes:

```
  do 20a i = 1, n
      b(i,1) = 0
  20a continue
  
  do 20b i = 1, n
  do 10 j = 1, m
      a(i) = a(i) + b(i,j) * c(i,j)
  10 continue
  20b continue
```
Loop Distribution

Compiled:

subroutine t1 (a,b,c,n,m)
real a(n), b(n,m), c(n,m)
integer i, j, n, m

C##4 [fc] Loop on line 4 of t1.f (DO I)
(distributed loop # 2) fully vectorized%

C##4 [fc] Loop on line 4 of t1.f (DO I) interchanged to
be innermost loop of nest%

C##4 [fc] Loop on line 4 of t1.f (DO I)
(distributed loop # 1) fully vectorized%

C##4 [fc] Loop on line 4 of t1.f (DO I) distributed,
forming 2 loops%

   do 20 i = 1, n
      b(i,1) = 0
   do 10 j = 1, m
      a(i) = a(i) + b(i,j) * c(i,j)

10 continue
20 continue
   return
end
• Loop Distribution

Source:

•

\[
\begin{align*}
&\text{do } 20 \ i = 1, \ n \\
&\quad \text{b}(i,1) = 0 \\
&\text{do } 10 \ j = 1, \ m \\
&\quad \text{a}(i) = \text{a}(i) + \text{b}(i,j) * \text{c}(i,j) \\
&\quad 10 \ 	ext{continue} \\
&\quad \text{d}(i) = \text{e}(i) + \text{a}(i) \\
&\quad 20 \ 	ext{continue}
\end{align*}
\]

•

Becomes:

•

\[
\begin{align*}
&\text{do } 20a \ i = 1, \ n \\
&\quad \text{b}(i,1) = 0 \\
&\quad 20a \ 	ext{continue} \\
&\text{do } 20b \ i = 1, \ n \\
&\quad \text{do } 10 \ j = 1, \ m \\
&\quad \text{a}(i) = \text{a}(i) + \text{b}(i,j) * \text{c}(i,j) \\
&\quad 10 \ 	ext{continue} \\
&\quad 20b \ 	ext{continue} \\
&\text{do } 20c \ i = 1, \ n \\
&\quad \text{d}(i) = \text{e}(i) + \text{a}(i) \\
&\quad 20c \ 	ext{continue}
\end{align*}
\]

•
• Reduction Operators

Maximum:

\[
\begin{align*}
\text{do } & 10 \ i = 1, 100 \\
& t1 = \max(t1, a(i)) \\
10 & \text{ continue}
\end{align*}
\]

Minimum:

\[
\begin{align*}
\text{do } & 20 \ j = 1, 100 \\
& t3 = \min(t3, b(j)) \\
20 & \text{ continue}
\end{align*}
\]

Sum:

\[
\begin{align*}
\text{do } & 30 \ j = 1, 100 \\
& t2 = t2 + b(j) \\
30 & \text{ continue}
\end{align*}
\]

Product:

\[
\begin{align*}
\text{do } & 40 \ i = 1, 100 \\
& t3 = t3 * c(i) \\
40 & \text{ continue}
\end{align*}
\]
Reduction Operators

Source:

```fortran
subroutine rede (a,b,e,t1,t2,t3)
real*8 a(100), b(100), e(100), t1, t2, t3
integer*4 i, j
C###4 [fc] Loop on line 4 of rede.f (DO I) fully vectorized%%%
do 10 i = 1, 100
  t1 = max(t1,a(i))
10 continue
C###7 [fc] Loop on line 7 of rede.f (DO J) fully vectorized%%%
do 20 j = 1, 100
  t3 = min(t3,b(j))
20 continue
C###10 [fc] Loop on line 10 of rede.f (DO J) fully vectorized%%%
do 30 j = 1, 100
  t2 = t2 + b(j)
30 continue
C###13 [fc] Loop on line 13 of rede.f (DO I) fully vectorized%%%
do 40 i = 1, 100
  t3 = t3 * c(i)
40 continue
return
end
```
Reduction Operators

_redc_

ld.w # 0x0000064, vl ; # 6
ld.w # 0x0000008, vs ; # 6
ld.w 0(ap), a1 ; # 6, A
ld.l @ 12(ap), s0 ; # 6, T1
ld.l 0(a1), v0 ; # 6, A
max.d v0 ; # 6, DMAX1
st.l s0, @ 12(ap) ; # 6, T1
ld.w # 0x0000064, vl ; # 10
ld.w # 0x0000008, vs ; # 10
ld.w 4(ap), a2 ; # 10, B
ld.l @ 20(ap), s1 ; # 10, T3
ld.l 0(a2), v1 ; # 10, B
min.d v1 ; # 10, DMIN1
st.l s1, @ 20(ap) ; # 10, T3
ld.w # 0x0000064, vl ; # 14
ld.w # 0x0000008, vs ; # 14
ld.w 4(ap), a3 ; # 14, B
ld.l @ 16(ap), s2 ; # 14, T2
ld.l 0(a3), v2 ; # 14, B
sum.d v2 ; # 14
st.l s2, @ 16(ap) ; # 14, T2
ld.w # 0x0000064, vl ; # 18
ld.w # 0x0000008, vs ; # 18
ld.w 8(ap), a4 ; # 18, C
ld.l @ 20(ap), s3 ; # 18, T3
ld.l 0(a4), v3 ; # 18
prod.d v3 ; # 18
st.l s3, @ 20(ap) ; # 18, T3
rtn ; # 20
Vector of Indices

Source:

```plaintext
subroutine iotafn(x)
real x(100)
doi i=1,100
  x(i) = i
enddo
return
end
```

Assembler:

```plaintext
_iotafn_

ld.w 0(ap),a1 ; # 4, X
ld.w # 0x0000064,vl ; # 4
ld.w # 0x000004,vs ; # 4
ld.w _mth$r_indx,v0 ; # 4, mth$r_indx
st.w v0,0(a1) ; # 4, X
rtn ; # 6
```
Vector of Indices

Source:

```plaintext
subroutine iotafn(x)
real x(100)
do i=1,100
   x(i) = i
enddo
return
end
```

Assembler:

```plaintext
_iotafn_
load x,R1 ; starting address of array x
load 100,VL ; vector length
load 4,VS ; size of array elements
load _mth$r_indx,V0 ; invoke mth$r_indx
stor V0,x ; store values in x
rtn
```
Vector of Indices

Source:

```fortran
subroutine iotafn(x)
real x(100)
do i=1,100
   x(i) = i
enddo
return
end
```

Compiled:

```fortran
subroutine iotafn(x)
real x(100)
C## Loop on line 3 of iotafn.f (DO I) fully vectorized
C do i=1,100
   x(i) = i
endo
return
end
```
Vector of Indices - Scatter/Gather

Source:

```
   do 10 j = 1, 100
    ix(j) = j
   10 continue

   do 20 j = 1, 100
    a(j) = b(ix(j))
   20 continue
```
Vector of Indices - Scatter/Gather

Source:

```fortran
subroutine gath (a,b,c,ix)
  real*8 a(100), b(100), c(100)
  integer*4 j, ix(100)
  C###4 [fc] Loop on line 4 of gath.f (DO J) fully vectorized%%% 
    do 10 j = 1, 100
    ix(j) = j
  10 continue
  C###7 [fc] Loop on line 7 of gath.f (DO J) fully vectorized%%% 
    do 20 j = 1, 100
    a(j) = b(ix(j))
  20 continue
  return
end
```
Vector of Indices - Scatter/Gather

_gath_:

```
ld.w 12(ap),a1 ; # 5, IX
ld.w #0x00000064,vl ; # 5
ld.w #0x0000004,vs ; # 5
ld.w _mth$j_indx,v0 ; # 5, mth$j_indx
st.w v0,0(a1) ; # 5, IX
ld.w #0x00000064,vl ; # 8
ld.w #0x00000004,vs ; # 8
ld.w 12(ap),a2 ; # 8, IX
ld.w #0x00000008,s0 ; # 8
ld.w 4(ap),a5 ; # 8, B
ld.w 0(ap),a3 ; # 8, A
ld.w 0(a2),v1 ; # 8, IX
mul.w v1,s0,v2 ; # 8
ld.w #0x00000008,vs ; # 8
add.w #0xfffffff8,a5 ; # 8
ldvi.l v2,v3 ; # 8, B
st.l v3,0(a3) ; # 8, A
rtn ; # 10
```
• Scalar Expansion

Code:

doi=1,100
if(z(i).gt.0.0) then
t = y1(i)
else
t = y2(i)
endif
x(i) = t
enddo

Becomes:

doi=1,100
if(z(i).gt.0.0) then
temp(i) = y1(i)
else
temp(i) = y2(i)
endif
x(i) = temp(i)
enddo
Scalar Expansion

Source:

```fortran
subroutine sclrex(x,y1,y2,z)
real x(100),y1(100),y2(100),z(100)
do i=1,100
   if(z(i).gt.0.0) then
      t = y1(i)
   else
      t = y2(i)
   endif
   x(i) = t
enddo
return
end
```

Compiled:

```fortran
subroutine sclrex(x,y1,y2,z)
real x(100),y1(100),y2(100),z(100)
C###3 [fc] Loop on line 3 of scalep.f (DO I) fully vectorized%%% 
do i=1,100
   if(z(i).gt.0.0) then
      t = y1(i)
   else
      t = y2(i)
   endif
   x(i) = t
enddo
return
end
```
Scalar Expansion

```
_sclrex_

sub.w #0x0000028,sp ; #1
sub.w #0x0000190,sp ; #3, ?push
ldea -440(fp),a1
st.w a1,-40(fp)
ld.w #0x0000064,v1 ; #4
ld.w #0x0000004,vs ; #7
ld.w 12(ap),a2
ld.w 4(ap),a3
ld.w 8(ap),a5
ld.w -40(fp),a4
ld.w 0(ap),a1
ld.w #0x0000000,s0 ; #4, 0.0
ld.w 0(a2),v0
lt.s s0,v0
st.x vm,-20(fp)
ld.w 0(a3),v1
ld.w 0(a5),v5
ld.w 0(a4),v2
sub.w s1,s1
mov s1,vm,s2
not s2,s2
mov s1,s2,vm
add.w #0x0000001,s1 ; #3
mov s1,vm,s2
not s2,s2
mov s1,s2,vm
st.x vm,-36(fp)
ld.x -20(fp),vm
mask.t v2,v1,v3
st.w v3,0(a4)
ld.w 0(a4),v4
ld.x -36(fp),vm
mask.t v4,v5,v6
st.w v6,v7,0(a4)
ld.w 0(a4),v7
st.w v7,0(a1)
add.w #0x0000190,sp ; #3, ?push
rtn ; #11
```

Vectorization of Conditionals

Source:

```fortran
subroutine cond (a,b,c)
  real*8 a(100), b(100), c(100)
  integer*4 i, j
  do 10 i = 1, 100
    if ( a(i) .gt. 0.0 ) then
      c(i) = a(i) * b(i)
    else
      c(i) = b(i)
    endif
  10 continue
  do 20 j = 1, 100
    if ( a(j) .gt. 2.0 ) goto 15
    c(j) = a(j) + b(j)
    go to 20
  15 continue
  c(j) = a(j) - b(j)
  20 continue
  return
end
```
. Vectorization of Conditionals

Source:

```fortran
subroutine cond (a,b,c)
real*8 a(100), b(100), c(100)
integer*4 i, j
C###4 [fc] Loop on line 4 of cond.f (DO I) fully vectorized%%% 
do 10 i = 1, 100
if ( a(i) .gt. 0.0 ) then
   c(i) = a(i) * b(i)
else
   c(i) = b(i)
endif
10 continue
C## # 11 [fc] Loop on line 11 of cond.f (DO J) fully vectorized%%% 
do 20 j = 1, 100
if ( a(j) .gt. 2.0 ) goto 15
   c(j) = a(j) + b(j)
go to 20
15 continue
   c(j) = a(j) - b(j)
20 continue
return
end
```
Vectorization of Conditionals

Code:

```fortran
DO I = 1,100
  IF(ISWTCH(I).GE.0) THEN
    X(I) = Y(I)*Z(I)
  ELSE
    X(I) = Y(I)-Z(I)
  ENDIF
ENDDO
```

Both clauses of the IF are computed, and the results are masked together.
. Vectorization of Conditionals

Source:

```fortran
SUBROUTINE COND(X,Y,Z,ISWTCH,I)
REAL X(100),Y(100),Z(100)
INTEGER I,ISWTCH(100)
DO I = 1,100
  IF(ISWTCH(I).GE.0) THEN
    X(I) = Y(I)*Z(I)
  ELSE
    X(I) = Y(I)-Z(I)
  ENDIF
ENDDO
RETURN
END
```

Compiled:

```fortran
SUBROUTINE COND(X,Y,Z,ISWTCH,I)
REAL X(100),Y(100),Z(100)
INTEGER I,ISWTCH(100)
C###4 [fc] Loop on line 4 of t1.f (DO I) fully vectorized%%%
DO I = 1,100
  IF(ISWTCH(I).GE.0) THEN
    X(I) = Y(I)*Z(I)
  ELSE
    X(I) = Y(I)-Z(I)
  ENDIF
ENDDO
RETURN
END
```
Vectorization of Conditionals

_cond_:

```
sub.w #0x0000020,sp ; #1
ld.w #0x0000001,s0 ; #4
st.w s0,16(ap) ; #4, I
ld.w #0x000001,s1 ; #4
st.w s1,16(ap) ; #4, I
ld.w #0x000064,v1 ; #5
ld.w #0x0000004,vs ; #6
ld.w 12(ap),a1 ; #5, ISWTCH
ld.w 4(ap),a3 ; #6, Y
ld.w 8(ap),a2 ; #6, Z
ld.w 0(ap),a4 ; #8, X
ld.w #0x0000000,s5 ; #8
ld.w #0x40800000,s6 ; #6
ld.w #0x0000000,s2 ; #5
ld.w 0(a1),v0 ; #5, ISWTCH
le.w s2,v0 ; #5, $cg_xtemp0
st.x vm,-16(fp) ; #5, $cg_xtemp0
ld.w 0(a3),v3 ; #6, Y
ld.w 0(a2),v1 ; #6, Z
sub.w s3,s3 ; #4
mov s3,vm,s4 ; #4
not s4,s4 ; #4
mov s3,s4,vm ; #4
add.w #0x0000001,s3 ; #4
mov s3,vm,s4 ; #4
not s4,s4 ; #4
mov s3,s4,vm ; #4
st.x vm,-32(fp) ; #4, $cg_xtemp1
ld.x -16(fp),vm ; #6, $cg_xtemp0
mask.f v1,s6,v5 ; #6, $cg_xtemp0
mul.s v3,v5,v6 ; #6
ld.x -32(fp),vm ; #8, $cg_xtemp1
mask.f v1,s5,v2 ; #8, $cg_xtemp1
sub.s v3,v2,v4 ; #8
mask.t v6,v4,v7 ; #8, $cg_xtemp1
st.w v7,0(a4) ; #8, X
ld.w #0x0000065,s7 ; #4
st.w s7,16(ap) ; #4, I
rtn ; #11
```
• Partial Vectorization

Source:

\[
\text{do } i = 1, n \\
a(i) = a(i-1) + b(i) \times c(i) \\
\text{enddo}
\]

Becomes:

\[
\text{do } i = 1, n \\
t(i) = b(i) \times c(i) \\
\text{enddo}
\]

\[
\text{do } i = 1, n \\
a(i) = a(i-1) + t(i) \\
\text{enddo}
\]
• Partial Vectorization

Source:

```fortran
subroutine part (a,b,c,n)
real*8 a(100), b(100), c(100)
integer*4 n
  do i = 1, n
    a(i) = a(i-1) + b(i) * c(i)
  enddo
return
end
```

Compiled:

```fortran
subroutine part (a,b,c,n)
real*8 a(100), b(100), c(100)
integer*4 n
C###4 [fc] Loop on line 4 of part.f (DO I) partially vectorized%
C###4 [fc] Loop on line 4 of part.f (DO I) The assignment to A
C on line 5 appears to be in a recurrence%
  do i = 1, n
    a(i) = a(i-1) + b(i) * c(i)
  enddo
return
end
```
• Partial Vectorization

_part_

sub.w #0x0000030,sp ; #1
ld.w @12(ap),s0 ; #4, N
le.w #0x0000001,s0 ; #4
jbrs.f M2 ; #4
ld.w #0x0000000,s1 ; #4
ld.w @12(ap),a1 ; #4, N
ldea -48(fp),a3 ; #4, ?LA
st.w s1,-36(fp) ; #4, ?i4
st.w a1,-16(fp) ; #4, ?vl
shf #0x0000003,a1 ; #4
neg.w a1,a2 ; #4, ?push
sub.w a1,sp ; #4
add.w a2,a3 ; #4
st.w a1,-12(fp) ; #4, ?fs
st.w a1,-40(fp) ; #0, t1
st.w a3,-48(fp) ; #4, ?LL

M3:

ld.w -16(fp),a2 ; #4, ?vl
ld.w #0x0000008,vs ; #5
• Partial Vectorization

M4:

```
ld.w -36(fp),a4 ; #4, ?i4
ld.w 4(ap),a1 ; #5, B
mov a2,a3 ; #4
mov a4,a5 ; #4
mov a3,v1 ; #4
add.w #0xffffffff80,a2 ; #4
add.w a5,a1 ; #5
ld.l 0(a1),v0 ; #5, B
ld.w -48(fp),a1 ; #5, v_1
add.w #0x0000400,a4 ; #4
st.w a4,-36(fp) ; #4, ?i4
ld.w 8(ap),a4 ; #5, C
add.w a5,a4 ; #5
ld.l 0(a4),v1 ; #5, C
mul.d v0,v1,v2 ; #5
add.w a1,a5 ; #5
st.l v2,0(a5) ; #5, v_1
lt.w #0x00000000,a2 ; #4
jbra.t M4 ; #4
st.w a2,-16(fp) ; #4, ?v1
ld.w #0xffffffff8,s3 ; #5
ld.w -40(fp),s2 ; #0, t1
add.w #0xffffffff0,s2 ; #4
st.w s3,-24(fp) ; #5, ?i1
ld.w -24(fp),a3 ; #5, ?i1
```
Partial Vectorization

M6:

```assembly
ld.w -48(fp),a5 ; #5, v_1
ld.w 0(ap),a2 ; #5, A
mov a3,a4
mov a4,a1
add.w #0x0000008,a3 ; #5
add.w #0x0000008,a1 ; #5
add.w a2,a4 ;#5
ld.l 0(a4),s5 ; #5, A
add.w a1,a5 ; #5
ld.l 0(a5),s4 ; #5, v_1
ld.w -32(fp),a5 ; #4, ?i3
add.w a2,a1 ; #5
add.d s4,s5 ; #5
le.w a3,a5 ; #4
st.l s5,0(a1) ; #5, A
jbra.t M6 ; #4
st.w a3,-24(fp) ; #5, ?i1
ld.w -12(fp),a4 ; #4, ?fs
add.w a4,sp ; #4, ?pop
```

M2:

```assembly
rtn ; Stmt -2
rtn ; #7
```
Detect Loop Iteration Count

Source:

```fortran
subroutine count(a,b,c)
  real*8 a(10), b(10), c(10)
  integer*4 i, j
  do 10 i = 1,2
    a(i) = b(i) + c(i)
  10 continue
  do 20 j = 1,3
    a(j) = b(j) + c(j)
  20 continue
  return
end
```

Compiled:

```fortran
subroutine count(a,b,c)
  real*8 a(10), b(10), c(10)
  integer*4 i, j
  C### 5 [fc] Loop on line 5 of count.f (DO I) not vectorized%
  C### 5 [fc] Loop on line 5 of count.f (DO I) executed fewer
  C than 3 times%
  do 10 i = 1,2
    a(i) = b(i) + c(i)
  10 continue
  C### 8 [fc] Loop on line 8 of count.f (DO J) fully vectorized%
  do 20 j = 1,3
    a(j) = b(j) + c(j)
  20 continue
  return
end
```
• Accept All Data Types

Source:

```c
subroutine all(ail,ai2,ai4,ai8,ar4,ar8,ac8,ac16,
    x       all,ai2,al4,ai8,bi1,bi2,bi4,bi8,br4,
    x       br8,bc8,bc16,bl1,bl2,bl4,bl8,ci1,ci2,
    x       ci4,ci8,cr4,cr8,cc8,cc16,cl1,cl2,cl4,cl8)

c    integer*1 ail(100), bil(100), cil(100)
    integer*2 ai2(100), bi2(100), ci2(100)
    integer*4 ai4(100), bi4(100), ci4(100)
    integer*8 ai8(100), bi8(100), ci8(100)
    real*4   ar4(100), br4(100), cr4(100)
    real*8   ar8(100), br8(100), cr8(100)
    complex*8 ac8(100), bc8(100), cc8(100)
    complex*16 ac16(100), bc16(100), cc16(100)
    logical*1 al1(100), bl1(100), cl1(100)
    logical*2 al2(100), bl2(100), cl2(100)
    logical*4 al4(100), bl4(100), cl4(100)
    logical*8 al8(100), bl8(100), cl8(100)

c    integer*4 j

c    do 20 j = 1, 100
        ail(j) = bil(j) + cil(j)
        ai2(j) = bi2(j) * ci2(j)
        ai4(j) = bi4(j) + ci4(j)
        ai8(j) = bi8(j) * ci8(j)
        ar4(j) = br4(j) + cr4(j)
        ar8(j) = br8(j) * cr8(j)
        ac8(j) = bc8(j) + cc8(j)
        ac16(j) = bc16(j) * cc16(j)
        al1(j) = bl1(j) .and. cl1(j)
        al2(j) = bl2(j) .or. cl2(j)
        al4(j) = bl4(j) .and. cl4(j)
        al8(j) = bl8(j) .or. cl8(j)
    20 continue

return
end
```
- Accept All Data Types

Compiled:

```fortran
subroutine all(ai1,ai2,ai4,ai8,ar4,ar8,ac8,ac16,
  x   al1,al2,al4,al8,bi1,bi2,bi4,bi8,br4, 
  x   br8,bc8,bc16,bl1,bl2,bl4,bl8,ci1,ci2, 
  x   ci4,ci8,cr4,cr8,cc8,cc16,cl1,cl2,cl4,cl8)
  c
  integer*1 ai(100), bi(100), ci(100)
  integer*2 ai2(100), bi2(100), ci2(100)
  integer*4 ai4(100), bi4(100), ci4(100)
  integer*8 ai8(100), bi8(100), ci8(100)
  real*4  ar4(100), br4(100), cr4(100)
  real*8  ar8(100), br8(100), cr8(100)
  complex*8 ac8(100), bc8(100), cc8(100)
  complex*16 ac16(100), bc16(100), cc16(100)
  logical*1 al(100), bl(100), cl(100)
  logical*2 al2(100), bl2(100), cl2(100)
  logical*4 al4(100), bl4(100), cl4(100)
  logical*8 al8(100), bl8(100), cl8(100)
  c
  integer*4 j
  c
C##24 [fc] Loop on line 24 of all.f (DO J) fully vectorized%%% 
  do 20 j = 1, 100
    ai1(j) = bi1(j) + ci1(j)
    ai2(j) = bi2(j) * ci2(j)
    ai4(j) = bi4(j) + ci4(j)
    ai8(j) = bi8(j) * ci8(j)
    ar4(j) = br4(j) + cr4(j)
    ar8(j) = br8(j) * cr8(j)
    ac8(j) = bc8(j) + cc8(j)
    ac16(j) = bc16(j) * cc16(j)
    al1(j) = bl1(j) .and. cl1(j)
    al2(j) = bl2(j) .or. cl2(j)
    al4(j) = bl4(j) .and. cl4(j)
    al8(j) = bl8(j) .or. cl8(j)
  20 continue
  return
  end
```
VECTORIZATION LIMITATIONS

Loops containing

- Character data or operations
- Computed/assigned goto's
- Function/subroutine references
- I/O statements
- Multiple exits
- Loops whose DO parameter varies with respect to the outer loop
- Equivalenced variables
- Recurrences
- Certain IF constructs
. Character Variables

Source:

```plaintext
program cmp
character*100 str1,str2
logical ieq
ieq=.true.
do i=1,100
   if(str1(i:i).ne.str2(i:i)) ieq=.false.
enddo
end
```

Compiled:

```plaintext
program cmp
character*100 str1,str2
logical ieq
ieq=.true.
C## Loop on line 5 of cmp.f (DO I) not vectorized%
C## Loop on line 5 of cmp.f (DO I) contains character expressions%
do i=1,100
   if(str1(i:i).ne.str2(i:i)) ieq=.false.
enddo
end
```
• GO TO Statements

Source:

```fortran
subroutine gtos(a,b,imx,idir)
integer*4 j, imx, idir
real*8 a(imx), b(imx)
do 100 j = 1, imx
goto (20,40,60) idir
   a(j) = 4.0 * b(j)
go to 60
20 continue
   a(j) = b(j)
go to 100
40 continue
   a(j) = -b(j)
go to 100
60 continue
100 continue
return
end
```
GO TO Statements

Compiled:

subroutine gtos(a,b,imx,idir)
  integer*4 j, imx, idir
  real*8 a(imx), b(imx)
C###4 [fc] Loop on line 4 of f1.f (DO J) not vectorized%%%n
C###4 [fc] Loop on line 4 of f1.f (DO J) contains a computed goto%%%n
do 100 j = 1, imx
  goto (20,40,60) idir
  a(j) = 4.0 * b(j)
go to 60
20 continue
  a(j) = b(j)
go to 100
40 continue
  a(j) = -b(j)
go to 100
60 continue
100 continue
  return
end
. GO TO Statements

Reworked Code:

subroutine gtos(a,b,imx,idir)
  integer*4 j, imx, idir
  real*8 a(imx), b(imx)
  goto (10,30,50) idir
10 continue
  do 20 j = 1, imx
  a(j) = 4.0 * b(j)
20 continue
  return
30 continue
  do 40 j = 1, imx
  a(j) = b(j)
40 continue
  return
50 continue
  do 60 j = 1, imx
  a(j) = -b(j)
60 continue
  return
end
. GO TO Statements

Compiled:

```fortran
subroutine gtos(a,b,imx,idir)
integer*4 j, imx, idir
real*8 a(imx), b(imx)
goto (10,30,50) idir
10 continue
C### 6 [fc] Loop on line 6 of f2.f (DO J) fully vectorized
   do 20 j = 1, imx
   a(j) = 4.0 * b(j)
20 continue
   return
30 continue
C### 11 [fc] Loop on line 11 of f2.f (DO J) fully vectorized
   do 40 j = 1, imx
   a(j) = b(j)
40 continue
   return
50 continue
C### 16 [fc] Loop on line 16 of f2.f (DO J) fully vectorized
   do 60 j = 1, imx
   a(j) = -b(j)
60 continue
   return
end
```
• Subroutine Calls

Source:

program vsub
real * 8 a(100), b(100), c(100)
do 10 i = 1, 100
   a(i) = 1.0
   b(i) = 2.0
10 continue
do 20 j = 1, 100
   c(j) = 0.0
   call armult(c(j),a,b)
20 continue
stop
end

Compiled:

program vsub
real * 8 a(100), b(100), c(100)
C# #3 [fc] Loop on line 3 of vsub.f (DO I) fully vectorized%%%
do 10 i = 1, 100
   a(i) = 1.0
   b(i) = 2.0
10 continue
C# #8 [fc] Loop on line 8 of vsub.f (DO J) contains a subroutine or function call%%%
C# #8 [fc] Loop on line 8 of vsub.f (DO J) not vectorized%%%
do 20 j = 1, 100
   c(j) = 0.0
   call armult(c(j),a,b)
20 continue
stop
end
• Input/Output

Source:

```fortran
program ios
  real*8 a(100), b(100)
  integer*4 i
  do 10 i = 1, 100
    a(i) = 1.0
    b(i) = a(i) * 3.0
    print 100, b(i)
  10 continue
100 format(3x,'value of b(i):',f6.3)
stop
end
```

Compiled:

```fortran
program ios
  real*8 a(100), b(100)
  integer*4 i
C## #4 [fc] Loop on line 4 of ios.f (DO I) not vectorized%%%
C## #4 [fc] Loop on line 4 of ios.f (DO I) performs I/O%%%
  do 10 i = 1, 100
    a(i) = 1.0
    b(i) = a(i) * 3.0
    print 100, b(i)
  10 continue
100 format(3x,'value of b(i):',f6.3)
stop
end
```
• Multiple Exits

Terminates prematurely:

```
   do i = 1, 100
      if(x(i) .lt. 0.0) go to 100
      nelem = i
      x(i) = x(i) / 2.0
   enddo

   100 continue
```

Abnormal conditions:

```
   do i = 1, 100
      if(x(i) .gt. 1e19) go to 900
      x(i) = x(i) ** 2
   enddo

   900 print *, 'error, x out of range'
```
. Multiple Exits

Terminates prematurely:

C##\# 4 [fc] Loop on line 4 of trm.f (DO I) not vectorized%%%
C##\# 4 [fc] Loop on line 4 of trm.f (DO I) or a contained loop
C has multiple exits
   do i = 1, 100
      if(x(i) .lt. 0.0) go to 100
      nelem = i
      x(i) = x(i) / 2.0
   enddo
100 continue

Abnormal conditions:

C##\# 4 [fc] Loop on line 4 of abn.f (DO I) not vectorized%%%
C##\# 4 [fc] Loop on line 4 of abn.f (DO I) or a contained loop
C has multiple exits
   do i = 1, 100
      if(x(i) .gt. 1e19) go to 900
      x(i) = x(i) ** 2
   enddo

900 print *, 'error, x out of range'
• Inner Loops With Varying Parameters

Source:

```plaintext
subroutine nested(x,y)
  real x(100,100),y(100)
  do j=1,100
    y(j) = y(j)**2
    do i=1,j
      x(j,i) = y(j)
    enddo
  enddo
enddo
return
end
```

Compiled:

```plaintext
subroutine nested(x,y)
  real x(100,100),y(100)
  C###3 [fc] Loop on line 3 of nested.f (DO J) not vectorized%
  C###3 [fc] Loop on line 3 of nested.f (DO J) An induction variable
  C appears to vary with each iteration%
  do j=1,100
    y(j) = y(j)**2
  C###5 [fc] Loop on line 5 of nested.f (DO I) fully vectorized%
  do i=1,j
    x(j,i) = y(j)
  enddo
enddo
return
end
```
Loops With Equivalenced Variables

Source:

```
program equivl
integer*4 i1(100),i2(100)
integer*2 ilb(200)
equivalence (i1(1),ilb(1))
do i=1,100
```

Compiled:

```
program equivl
integer*4 i1(100),i2(100)
integer*2 ilb(200)
equivalence (i1(1),ilb(1))
      C###5 [fc] Loop on line 5 of equivl.f (DO I) not vectorized%
      C###5 [fc] Loop on line 5 of equivl.f (DO I) An equivalenced variable
      C   or array inhibits vectorization%
do i=1,100
   ilb((i*2)-1) = 0 !set upper two bytes to 0
   i2(i) = i2(i) + i1(i) !land add to i2
enddo
end
```
Loops With Equivalenced Variables

Source:

```fortran
program equiv1
integer*4 i1(100),i2(100)
integer*2 ilb(200)
equivalence (i1(1),ilb(1))

C###5 [fc] Loop on line 5 of equiv1.f (DO I) not vectorized%%%
C###5 [fc] Loop on line 5 of equiv1.f (DO I) An equivalenced variable
C or array inhibits vectorization%%%
do i=1,100
   ilb((i*2)-1) = 0
   i2(i) = i2(i) + i1(i)
enddo
end
```

After:

```fortran
program equiv2
integer*4 i1(100),i2(100)
integer*2 ilb(200)
equivalence (i1(1),ilb(1))

C###5 [fc] Loop on line 5 of equiv2.f (DO I) not vectorized%%%
C###5 [fc] Loop on line 5 of equiv2.f (DO I) An equivalenced variable
C or array inhibits vectorization%%%
do i=1,100
   ilb((i*2)-1) = 0 !set upper two bytes to 0
   i2(i) = i2(i) + i1(i) !and add to i2
enddo
end
```

C###8 [fc] Loop on line 8 of equiv2.f (DO I) fully vectorized%%%
do i=1,100
   i2(i) = i2(i) + i1(i) !and add to i2
enddo
```
• Recursion

Scalar Processing

• Allows dependencies such as calculating the value of an element and immediately using that result to calculate the value of the next element in the vector.

Vector Processing

• All elements of a vector are treated exactly the same and in a group.

• Vector elements must be independent of one another

• The value of one element may not be dependent on the result of a calculation involving any other element in the group.
• Recursion

Recursion can occur two ways:

• Result Not Ready

The operations in the current iteration depend on results from a previous iteration. Doing the operations in parallel (vectorization) requires that the calculations for an iteration only depend on "old" values for a vector.

• Value No Longer Available

Old values of a variable are modified as the current value is being used in calculations. Doing the operations in parallel implies that previous values and the current value are modified simultaneously.
Recursion

Dependency analysis looks at pairs of uses of a variable and identifies three types of recursion:

- **Conflict between usage-assignment pairs**
  
  ```
  do i=1,100
      x(i) = x(i-1) + y(i)
  enddo
  ```

- **Conflict between assignment-assignment pairs**
  
  ```
  do i=1,100
      x(i+j) = y(i)
      x(i+k) = z(i)
  enddo
  ```

- **Conflict between assignment-usage pairs**
  
  ```
  do i=1,100
      x(i-1) = y(i)
      z(i) = x(i)
  enddo
  ```
. Recursion

Usage-Assignment Recursion

\[
\text{do } i=2,100 \\
x(i) = x(i-1) + y(i) \\
\text{enddo}
\]

<table>
<thead>
<tr>
<th>Trip</th>
<th>x(1)</th>
<th>x(2)</th>
<th>x(3)</th>
<th>x(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>x₁ + y₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(x₁ + y₂ + y₃)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>(x₁ + y₂ + y₃ + y₄)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>x₁ + y₂</td>
<td>x₂ + y₃</td>
<td>x₃ + y₄</td>
<td></td>
</tr>
</tbody>
</table>
Recursion

Assignment-Usage Recursion

\[
\text{do } i = 2, 100 \\
x(i-1) = y(i) \\
x(i) = x(i)  \\
\text{enddo}
\]

<table>
<thead>
<tr>
<th>Trip</th>
<th>x(1)</th>
<th>x(2)</th>
<th>x(3)</th>
<th>x(4)</th>
<th>z(1)</th>
<th>z(2)</th>
<th>z(3)</th>
<th>z(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar 1</td>
<td>y2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>y3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>y4</td>
<td></td>
<td></td>
<td></td>
<td>x4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector 1</td>
<td>y2</td>
<td>y3</td>
<td>y4</td>
<td>y5</td>
<td>y2</td>
<td>y3</td>
<td>y4</td>
<td>y5</td>
</tr>
</tbody>
</table>
Recursion

Assignment-Assignment Recursion

do i=1,98
   x(i) = y(i)
   x(i+2) = z(i)
enddo

<table>
<thead>
<tr>
<th>Trip</th>
<th>x(1)</th>
<th>x(2)</th>
<th>x(3)</th>
<th>x(4)</th>
<th>x(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>y1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>y2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>y3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>y4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>y1</td>
<td>y2</td>
<td>z1</td>
<td>z2</td>
<td>z3</td>
</tr>
</tbody>
</table>
. Recursion

Source:

```fortran
subroutine recur (a,n)
real*8 a(100)
integer*4 n
do i = 1, n
  a(i) = a(i-1) + 5
endo
return
end
```

Compiled:

```fortran
subroutine recur (a,n)
real*8 a(100)
integer*4 n
C## #4 [fc] Loop on line 4 of recur.f (DO I) not vectorized%%%
C## #4 [fc] Loop on line 4 of recur.f (DO I). The assignment to A
C on line 5 appears to be in a recurrence%%%
do i = 1, n
  a(i) = a(i-1) + 5
endo
return
end
```
- Recurrence

Code:

```plaintext
  DO 10 J = 1, M
  A(J) = A(J-1) + B(J) * C(J)
  10 CONTINUE

Rewrite:

```plaintext
  DO 10 J = 1, M
  T(J) = B(J) * C(J)
  10 CONTINUE

  DO 20 J = 1, M
  A(J) = A(J-1) + T(J)
  20 CONTINUE
```
EXCEPTIONS
* Store Reversal

Source:

```fortran
subroutine recur2(x,y,z)  
 real x(100),y(100),z(100)  
 do i=2,100  
   x(i) = y(i)  
   z(i) = x(i)  
 enddo  
 return  
end
```

Compiled:

```fortran
subroutine recur2(x,y,z)  
 real x(100),y(100),z(100)  
C# # 3 [fc] Loop on line 3 of t2.f (DO 1) fully vectorized9  
 do i=2,100  
   x(i) = y(i)  
   z(i) = x(i)  
 enddo  
 return  
end
```
. Store Reversal

; INSTRUCTIONS

_recur2_: 

    ld.w       # 0x0000063,v1  ; # 5
    ld.w       # 0x0000004,vs ; # 5
    ld.w       4(ap),a2     ; # 5, Y
    ld.w       0(ap),a1     ; # 6, X
    ld.w       8(ap),a3     ; # 6, Z
    ld.w       4(a2),v1     ; # 5, Y
    ld.w       4(a1),v0     ; # 6, X
    st.w       v0,4(a3)     ; # 6, Z
    st.w       v1,0(a1)     ; # 5, X
    rtn        ; # 8


Set Overlap Analysis

Source:

```plaintext
subroutine recur3(x,y)
dimension x(100),y(100)
do i = 1,99,2
   x(i) = y(i)
   x(i+1) = -y(i+1)
enddo
return
end
```

Compiled:

```plaintext
subroutine recur3(x,y)
dimension x(100),y(100)
C# # 3 [fc] Loop on line 3 of t2.f (DO I) fully vectorized?
do i = 1,99,2
   x(i) = y(i)
   x(i+1) = -y(i+1)
enddo
return
end
```
• Set Overlap Analysis

; INSTRUCTIONS

_recur3_: 

    ld.w # 0x0000032, vl ; # 4
    ld.w # 0x0000008, vs ; # 5
    ld.w 4(ap), a1 ; # 5, Y
    ld.w 0(ap), a2 ; # 5, X
    ld.w 4(a1), v1 ; # 5, Y
    neg.s v1, v2 ; # 5
    st.w v2, 4(a2) ; # 5, X
    ld.w 0(a1), v0 ; # 4, Y
    st.w v0, 0(a2) ; # 4, X
    rtn ; # 7
Sum Reduction

Source:

```fortran
subroutine sumred(xsum,x)
    real x(100),xsum
    xsum = 0.0
    do i=1,100
        xsum = xsum + x(i)
    enddo
    return
end
```

; INSTRUCTIONS

```assembly
_sumred_:  
    ld.w   # 0x00000000,s0 ; # 3, 0.0
    st.w   s0,@ 0(ap)    ; # 3, XSUM
    ld.w   # 0x00000064,v1 ; # 5
    ld.w   # 0x0000004,vs ; # 5
    ld.w   4(ap),a1      ; # 5, X
    ld.w   @ 0(ap),s1    ; # 5, XSUM
    ld.w   0(a1),v0      ; # 5, X
    mov.w  s1,s0         ; # 5
    sum.s  v0            ; # 5
    st.w   s0,@ 0(ap)    ; # 5, XSUM
    rtn    ; # 7
```
EXTENSIONS
. Compiler Directives

NO_SIDE_EFFECTS

SCALAR

NO_RECURRENCE
OPPORTUNITIES
The major difference between the mathematical description of an algorithm and the program to execute it is the description and manipulation of the data structures.
Three performance levels

- Scalar
- Vector
- Super Vector
Memory Access Considerations

- Memory operations on consecutive memory locations (the first subscript of a FORTRAN array) are handled by the cache bypass hardware, loading one double precision or two single precision elements every clock cycle.

- Memory operations on non-consecutive memory locations are handled by the normal address-translation and cache hardware, and may take many cycles per element.
Memory Access Considerations

Source:

```fortran
program memacc
  real*4 x(1000000),z(1000000)
  time1 = extime(dummy)
  do i = 1,1000000,1
    x(i) = 0.0
    z(i) = 1.0
  enddo
  time2 = extime(dummy)
  write(6,1000) time2-time1
  do istr = 1,10
    time1 = extime(dummy)
    c$dir scalar
    do k = 1,istr
      do i = 1,1000000,istr
        x(i) = z(i)
      enddo
    enddo
    time2 = extime(dummy)
    write(6,1010) istr,time2-time1
  enddo
  stop
1000 format(' Initialization : ',f6.4,' sec.')
1010 format(' Copy with stride ',i2,': ',f6.4,' sec. ')
end
```
Memory Access Considerations

Compiled:

```fortran
program memacc
 real*4 x(1000000),z(1000000)
 time1 = extime(dummy)
 C###5 [fc] Loop on line 5 of memacc.f (DO I) fully vectorized%%%%
 do i = 1,1000000,1
   x(i) = 0.0
   z(i) = 1.0
 enddo
 time2 = extime(dummy)
 write(6,1000) time2-time1
 C###12 [fc] Loop on line 12 of memacc.f (DO ISTR) not vectorized%%%%
 C###12 [fc] Loop on line 12 of memacc.f (DO ISTR) An
 starting value or stride that appears to vary
 with each iteration%%%%
 do istr = 1,100
 time1 = extime(dummy)
 c$dir scalar
 C###15 [fc] Loop on line 15 of memacc.f (DO I) fully vectorized%%%%
 inhibited by SCALAR directive%%%%
 do k = 1,istr
 C###16 [fc] Loop on line 16 of memacc.f (DO I) fully vectorized%%%%
 do i = 1,1000000,istr
   x(i) = z(i)
 enddo
 enddo
 time2 = extime(dummy)
 write(6,1010) istr,time2-time1
 enddo
 stop
 1000 format(' Initialization : ','f6.4,' sec.')
 1010 format(' Copy with stride ',i2,': ','f6.4,' sec.')
 end```

```fortran```
. Memory Access Considerations

Timings:

Initialization : 0.2683 sec.

Copy with stride 1: 0.1511 sec.

Copy with stride 2: 0.8134 sec.

Copy with stride 3: 1.1837 sec.

Copy with stride 4: 1.5401 sec.

Copy with stride 5: 1.8226 sec.

Copy with stride 6: 2.1754 sec.

Copy with stride 7: 2.4533 sec.

Copy with stride 8: 2.6608 sec.

Copy with stride 9: 2.5925 sec.

Copy with stride 10: 2.6643 sec.
• Other Considerations

Memory access time as a function of vector stride

Vector spills and temps may overrun stack
• Matrix Initialization

Method 1:

\[
\begin{align*}
  & \text{do } j = 1, n \\
  & \quad \text{do } i = 1, n \\
  & \quad \quad \text{if ( } i \text{ .eq. } j \text{ ) then} \\
  & \quad \quad \quad a(i, j) = c \\
  & \quad \quad \text{else} \\
  & \quad \quad \quad a(i, j) = 0.0 \\
  & \quad \text{end if} \\
  & \quad \text{end do} \\
  & \text{end do}
\end{align*}
\]

Method 2:

\[
\begin{align*}
  & \text{do } j = 1, n \\
  & \quad \text{do } i = 1, n \\
  & \quad \quad a(i, j) = 0.0 \\
  & \quad \text{end do} \\
  & \quad a(j, j) = c \\
  & \text{end do}
\end{align*}
\]
Matrix Initialization

Method II

time1 = cputime(0.0)
time2 = cputime(time1)
overhd = time2 - time1
time1 = cputime(0.0)

C##37 [fc] Loop on line 37 of diag.f (DO J) (distributed loop #1)
  fully vectorized
C##37 [fc] Loop on line 37 of diag.f (DO J) (distributed loop #2)
  fully vectorized
C##37 [fc] Loop on line 37 of diag.f (DO J) distributed,
  forming 2 loops
  do j = 1,n
    do i = 1,n
      a(i,j) = 0.0
    end do
    a(j,j) = c
  end do

time2 = cputime(time1)
time = time2 - time1 - overhd
print *, 'Method II time : ', time, ' secs.'

Timing:

Method I time : 0.6618650 secs.
Method II time : 8.4289074E-02 secs.
- **Matrix Multiplication**

**Method 1:**

```plaintext
do j = 1,n
do i = 1,n
    sum = 0.0
    do k = 1,n
        sum = sum + a(j,k)*b(k,i)
    end do
    c(i,j) = sum
end do
end do
```

**Method 2:**

```plaintext
do j = 1,n
do i = 1,n
    c(i,j) = 0.0
    do k = 1,n
        c(i,j) = c(i,j) + a(j,k)*b(k,i)
    end do
end do
end do
```
Matrix Multiplication

program mult

c
Two versions of a matrix multiply

real*4 a(10,10), b(10,10), c(10,10), sum
real*4 time, time1, time2, ovrhd, cputime
integer*4 i, j, n

data a / 100*1.0/ , b / 100*1.0/
data n / 10 /

time1 = cputime(0.0)
time2 = cputime(time1)
overhd = time2 - time1

time1 = cputime(0.0)

C###17 [fc] Loop on line 17 of mult.f (DO J) unable to
distribute loop%%% C###17 [fc] Loop on line 17 of mult.f (DO J) not vectorized%%% do j = 1, n
C###18 [fc] Loop on line 18 of mult.f (DO I) unable to
distribute loop%%% C###18 [fc] Loop on line 18 of mult.f (DO I) not vectorized%%% do i = 1, n
sum = 0.0
C###20 [fc] Loop on line 20 of mult.f (DO K) fully vectorized%%% do k = 1, n
sum = sum + a(j,k)*b(k,i)
end do

c(i,j) = sum
end do

time2 = cputime(time1)
time = time2 - time1 - overhd
print *, 'Method I time: ', time, ' seconds.'
• Matrix Multiplication

c
  time1 = cputime(0.0)
  time2 = cputime(time1)
  ovrhd = time2 - time1

c
time1 = cputime(0.0)

c
  C# # # # 37 [fc] Loop on line 37 of mult.f (DO J) distributed, 
  C forming 2 loops
  C# # # # 37 [fc] Loop on line 37 of mult.f (DO J) (distributed loop #1) 
  C fully vectorized
  C# # # # 37 [fc] Loop on line 37 of mult.f (DO J) interchanged to be 
  C innermost loop of nest
  C# # # # 37 [fc] Loop on line 37 of mult.f (DO J) (distributed loop #2) 
  C fully vectorized

  do j = 1,n
      do i = 1,n
          c(i,j) = 0.0
          do k = 1,n
              c(i,j) = c(i,j) + a(j,k)*b(k,i)
          end do
      end do
  end do

c
  time2 = cputime(time1)
  time = time2 - time1 - ovrhd
  print *, 'Method II time: ',time,' seconds.'

c
  stop
  end

Timings:

  Method I time:  1.6579996E-03 seconds.
  Method II time:  9.5599983E-04 seconds.
Matrix Square Root

Method 1:

do j = 1,n
  do i = j,n
    g(i,j) = l(i,j) * sqrt(d(j))
  end do
end do

Method 2:

do j = 1,n
  do i = 1,n
    g(i,j) = l(i,j) * sqrt(d(j))
  end do
end do

Method 3:

do i = 1,n
  d(i) = sqrt( d(i) )
end do

do j = 1,n
  do i = 1,n
    g(i,j) = l(i,j) * d(j)
  end do
end do
Matrix Square Root

program square_root

Square root of a symmetric, square matrix

by the LDL decomposition

real *4 l(512,512), g(512,512), d(512), time1, time2,
&     time, overhd
integer *4 n

Put some stuff in l & d

Loop on line 12 of sqroot.f (DO J) An induction
variable of a contained loop has a starting value or
stride that appears to vary with each iteration

Loop on line 12 of sqroot.f (DO J) not vectorized

Loop on line 13 of sqroot.f (DO I) fully vectorized!

What's wrong with this loop?

Loop on line 18 of sqroot.f (DO I) fully vectorized

Matrix Square Root

Method I:

time1 = cputime ( 0.0 )
time2 = cputime ( time1 )
overhd = time2 - time1

time1 = cputime ( 0.0 )

C##29 [fc] Loop on line 29 of sqroot.f (DO J) not vectorized

C##29 [fc] Loop on line 29 of sqroot.f (DO J) An induction

An induction variable of a contained loop has a starting value or

stirde that appears to vary with each iteration

do j = 1,n

C##30 [fc] Loop on line 30 of sqroot.f (DO I) fully vectorized

do i = j,n

    g(i,j) = l(i,j) * sqrt(d(j))

end do

end do

time2 = cputime ( time1 )
time = time2 - time1 - overhd

print *, 'Method I time : ', time, ' secs.'
Matrix Square Root

Method II: (include upper triangular elements in multiplication, even though we're multiplying by zero. We get full vectorization, though)

time1 = cputime ( 0.0 )
time2 = cputime ( time1 )
overhd = time2 - time1

time1 = cputime ( 0.0 )

C##48 [fc] Loop on line 48 of sqroot.f (DO J) fully vectorized

Do j = 1,n
  Do i = 1,n
    g(i,j) = l(i,j) * sqrt(d(j))
  End do
End do

C
time2 = cputime ( time1 )
time = time2 - time1 - overhd
print *, 'Method II time : ', time, ' secs.'
Matrix Square Root

Method III:

time1 = cputime(0.0)
time2 = cputime(time1)
overhd = time2 - time1
time1 = cputime(0.0)

C###65 [fc] Loop on line 65 of sqroot.f (DO I) fully vectorized%%%
do i = 1,n
d(i) = sqrt(d(i))
end do
C###68 [fc] Loop on line 68 of sqroot.f (DO J) fully vectorized%%%
do j = 1,n
  do i = 1,n
    g(i,j) = l(i,j) * d(j)
  end do
end do

time2 = cputime(time1)
time = time2 - time1 - overhd
print *, 'Method III time: ', time, ' secs.'

stop
end

Timing:

Method I time :  6.7269996E-02 secs.
Method II time : 7.9784006E-02 secs.
Method III time : 4.8183024E-02 secs.
Matrix Transpose

Method 1:

\[
\begin{align*}
\text{do } j &= 1,500 \\
\text{do } i &= 1,100 \\
\text{atrans}(i,j) &= a(j,i) \\
\text{end do} \\
\text{end do}
\end{align*}
\]

Method 2:

\[
\begin{align*}
\text{call trans(500,100)} \\
\text{subroutine trans(nrow,ncol)} \\
\text{common a(500,100), atrans(100,500)} \\
\text{do } i = 1,\text{ncol} \\
\text{do } j = 1,\text{nrow} \\
\text{atrans}(i,j) &= a(j,i) \\
\text{end do} \\
\text{end do} \\
\text{return} \\
\text{end}
\end{align*}
\]
Matrix Transpose

Source:

program transpose
common a(500,100), atrans(100,500)

Illustrate matrix transposition for rectangular matrices in which the row dimension exceeds the column dimension.

Method I: Unit stride for output (transposed) matrix

time1 = cputime ( 0.0 )
time2 = cputime ( 0.0 )
overhd = time2 - time1
time1 = cputime ( 0.0 )
C###14 [fc] Loop on line 14 of transpose.f (DO J) fully vectorized%
   do j = 1,500
      do i = 1,100
         atrans(i,j) = a(j,i)
      end do
   end do
time2 = cputime ( 0.0 )
time = time2 - time1 - overhd
print *, 'Method I time : ', time, 'secs.'
Matrix Transpose

Method II: Get around loop interchange problem.

timel = cputime ( 0.0 )
time2 = cputime ( 0.0 )
overhd = time2 - timel

timel = cputime ( 0.0 )
    call trans ( 500, 100 )
time2 = cputime ( 0.0 )
time = time2 - time1 - overhd
print *, 'Method II time: ', time, 'secs.'

stop
end

subroutine trans ( nrow, ncol )
    common a(500,100), atrans(100,500)

C###40 [fc] Loop on line 40 of transpose.f (DO I) fully vectorized%
    do i = 1,ncol
        do j = 1,nrow
            atrans(i,j) = a(j,i)
        end do
    end do
return
end

Timings:

Method I time :  2.8062003E-02secs.
Method II time :  1.8757001E-02secs.
. Polynomial Evaluation

Method 1:

\[
\text{do } j = 1,10000 \\
p = a(n+1) \\
\text{do } i = 1,n \\
p = x*p + a(n-i+1) \\
\text{end do} \\
\text{end do}
\]

Method 2:

\[
\text{do } j = 1,10000 \\
t_1 = a(1) + a(2)*x \\
t_2 = a(3)*x*x + a(4)*(x*x)*x \\
p = t_1 + t_2 \\
\text{end do}
\]
Polynomial Evaluation

program polynomial
  
  Simple polynomial evaluation.
  
  real *4 a(4), x, p, time, time1, time2, cputime, overhd
  integer *4 n
  data n / 3 /, x / 2.0 /
  
  C###9 [fc] Loop on line 9 of poly.f (DO I) fully vectorized%
  do i = 1,n
    a(i) = 1.0
  end do
  
  C###18 [fc] Loop on line 18 of poly.f (DO J) not vectorized%
  C###18 [fc] Loop on line 18 of poly.f (DO J) unable to distribute loop%
  do j = 1,10000
    p = a(n+1)
  end do
  
  C###20 [fc] Loop on line 20 of poly.f (DO I) not-vectorized%
  C###20 [fc] Loop on line 20 of poly.f (DO I) has insufficient vectorizable code%
  C###20 [fc] Loop on line 20 of poly.f (DO I) The assignment to P on line 21 appears to be in a recurrence%
  do i = 1,n
    p = x*p + a(n-i+1)
  end do
  
  time2 = cputime ( time1 )
  time = time2 - time1 - overhd
  print *, 'Method I time : ', time, ' secs.'
• Polynomial Evaluation

c time1 = cputime ( 0.0 )
time2 = cputime ( time1 )
overhd = time2 - time1
time1 = cputime ( 0.0 )

c C$\text{DIR SCALAR}
C##35 [fc] Loop on line 35 of poly.f (DO J) vectorization inhibited
C by SCALAR directive%%%
do j = 1,10000
t1 = a(1) + a(2)*x
t2 = a(3)*(x*x) + a(4)*(x*x)*x
p = t1 + t2
end do

c time2 = cputime ( time1 )
time = time2 - time1 - overhd
print *, 'Method II time : ', time, ' secs.'

c stop
end

Timing:

Method I time :  0.1014940    secs.
Method II time :  3.2079965E-03 secs.
. Boundary Conditions

Method 1:

\[
\begin{align*}
&\text{do } 20 \text{ } j = 2, 100 \\
&\quad \text{do } 10 \text{ } i = 1, 100 \\
&\quad \quad \text{if ( } i \text{ .ne. } 1 \text{ ) then} \\
&\quad \quad \quad a(i,j) = b(i,j) \\
&\quad \quad \text{else} \\
&\quad \quad \quad a(i,j) = 0.0 \\
&\quad \text{endif} \\
&10 \text{ } \text{continue} \\
&20 \text{ } \text{continue}
\end{align*}
\]

Method 2:

\[
\begin{align*}
&\text{do } 40 \text{ } j = 2, 100 \\
&\quad a(1,j) = 0.0 \\
&\text{do } 30 \text{ } i = 2, 100 \\
&\quad a(i,j) = b(i,j) \\
&30 \text{ } \text{continue} \\
&40 \text{ } \text{continue}
\end{align*}
\]
• Boundary Conditions

c
  time1 = cputime(0.0)
  time2 = cputime(time1)
  ovrhd = time2 - time1

c
  time1 = cputime(0.0)

c
C###31 [fc] Loop on line 31 of bnd.f (DO J) (distributed loop #1)
C   fully vectorized%
C###31 [fc] Loop on line 31 of bnd.f (DO J) (distributed loop #2)
C   fully vectorized%
C###31 [fc] Loop on line 31 of bnd.f (DO J) distributed,
C   forming 2 loops%

    do 40 j = 2, 100
        a(l,j) = 0.0
    do 30 i = 2, 100
        a(i,j) = b(i,j)
    30 continue
    40 continue

c
  time2 = cputime(time1)
  time = time2 - time1 - ovrhd
  print *, 'Method II time: ',time,' seconds.'
  stop
end

Timings:

Method I time:  7.8180004E-03 seconds.
Method II time:  2.6069973E-03 seconds.
Boundary Conditions

```
program bnds
  real*8 a(100,100), b(100,100)
  real*4 time, time1, time2, ovrhd, cputime
  integer*4 i, j
  time1 = cputime(0.0)
  time2 = cputime(time1)
  ovrhd = time2 - time1
  
  c
  time1 = cputime(0.0)
  
  C### #11 [fc] Loop on line 11 of bnd.f (DO J) fully vectorized%
  do 20 j = 2, 100
    do 10 i = 1, 100
      if (i .ne. 1) then
        a(i,j) = b(i,j)
      else
        a(i,j) = 0.0
      endif
    10 continue
  20 continue
  
  C
  time2 = cputime(time1)
  time = time2 - time1 - ovrhd
  print *, 'Method I time: ',time,' seconds.'
```
RESTRUCTURING

CODE
Restructuring is the process by which existing source code is examined and modified in order to increase vectorization and optimization.
Profiling

Identify which routine or areas of code account for significant portions of the total CPU time used.

This is accomplished by using the (-p) option for compiling and linking the code.
## Profiling

Example of a profile

<table>
<thead>
<tr>
<th>%time</th>
<th>cumsecs</th>
<th># call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.2</td>
<td>43.07</td>
<td>3634464</td>
<td>0.01</td>
<td>mcount</td>
</tr>
<tr>
<td>15.3</td>
<td>66.48</td>
<td>1632</td>
<td>12.25</td>
<td><em>cvmgm</em></td>
</tr>
<tr>
<td>13.1</td>
<td>86.47</td>
<td>408</td>
<td>30.96</td>
<td><em>monot</em></td>
</tr>
<tr>
<td>8.3</td>
<td>99.10</td>
<td>408</td>
<td>6.48</td>
<td><em>flatten</em></td>
</tr>
<tr>
<td>6.9</td>
<td>109.68</td>
<td>408</td>
<td>25.12</td>
<td><em>riemann</em></td>
</tr>
<tr>
<td>6.8</td>
<td>120.13</td>
<td>408</td>
<td>25.61</td>
<td><em>states</em></td>
</tr>
<tr>
<td>6.7</td>
<td>130.38</td>
<td>408</td>
<td>25.12</td>
<td><em>mth$vr_sqrt</em></td>
</tr>
<tr>
<td>5.7</td>
<td>139.05</td>
<td>412488</td>
<td>0.02</td>
<td><em>detect</em></td>
</tr>
<tr>
<td>3.8</td>
<td>144.78</td>
<td>408</td>
<td>14.04</td>
<td><em>monostartup</em></td>
</tr>
<tr>
<td>0.9</td>
<td>146.18</td>
<td>247248</td>
<td>0.01</td>
<td><em>cvmgp</em></td>
</tr>
<tr>
<td>0.8</td>
<td>148.75</td>
<td>244800</td>
<td>0.01</td>
<td><em>cvmgz</em></td>
</tr>
<tr>
<td>0.7</td>
<td>149.75</td>
<td>5714</td>
<td>0.18</td>
<td><em>mth$vr_sqrt</em></td>
</tr>
<tr>
<td>0.3</td>
<td>150.25</td>
<td>408</td>
<td>1.23</td>
<td><em>hydrow</em></td>
</tr>
<tr>
<td>0.3</td>
<td>150.68</td>
<td>408</td>
<td>1.05</td>
<td><em>tstep</em></td>
</tr>
<tr>
<td>0.2</td>
<td>151.04</td>
<td>408</td>
<td>0.88</td>
<td><em>coeff</em></td>
</tr>
<tr>
<td>0.2</td>
<td>151.35</td>
<td>408</td>
<td>0.76</td>
<td><em>intrfc</em></td>
</tr>
<tr>
<td>0.1</td>
<td>151.57</td>
<td>1006</td>
<td>0.22</td>
<td>_cvt</td>
</tr>
<tr>
<td>0.1</td>
<td>151.78</td>
<td>1412</td>
<td>0.15</td>
<td>_for$do_fio</td>
</tr>
<tr>
<td>0.1</td>
<td>151.97</td>
<td>1</td>
<td>190.00</td>
<td><em>MAIN</em></td>
</tr>
<tr>
<td>0.1</td>
<td>152.16</td>
<td>14723</td>
<td>0.01</td>
<td>_x_putc</td>
</tr>
<tr>
<td>0.1</td>
<td>152.34</td>
<td>1006</td>
<td>0.18</td>
<td>_wrt_E</td>
</tr>
</tbody>
</table>
. Compiler Messages

Examine the compiler generated messages.

cvmgm.f:

cvmgp.f:

cvmgz.f:

detect.f:
Loop on line 38.1 of detect.f (DO I) fully vectorized
Loop on line 45.1 of detect.f (DO I) fully vectorized
Loop on line 49.1 of detect.f (DO I) fully vectorized
Loop on line 54.1 of detect.f (DO I) contains a subroutine or function call
Loop on line 54.1 of detect.f (DO I) not vectorized
Loop on line 63.1 of detect.f (DO I) contains a subroutine or function call
Loop on line 63.1 of detect.f (DO I) not vectorized
Loop on line 67.1 of detect.f (DO I) fully vectorized
Loop on line 72.1 of detect.f (DO I) contains a subroutine or function call
Loop on line 72.1 of detect.f (DO I) not vectorized
Loop on line 78.1 of detect.f (DO I) contains a subroutine or function call
Loop on line 78.1 of detect.f (DO I) not vectorized
Loop on line 86.1 of detect.f (DO I) fully vectorized
Loop on line 91.1 of detect.f (DO I) fully vectorized