THEORY II
Vectorization examples

Basics: How to calculate the performance
Vectorization examples

\[ v = \text{Vector} \quad s = \text{Scalar} \]

- \[ v = s + v \]
- \[ v = v + v \]
- \[ v = v + s \times v \]
- \[ s = s + v \times v \]
- matrix multiply
ex. 1: $v = s + v$

- 3 cycles are needed to perform one calculation
- $R < 4 \times R_0 \times 1 / 3 = 2667$ MFlops ($R_0 = 2000$ MFlops)
- measured: 3969 MFlops
- What’s wrong?
Chaining

Memory

\[
\begin{align*}
y(1) \\
y(2) \\
y(3) \\
y(4) \\
y(5) \\
y(6) \\
y(7) \\
y(8)
\end{align*}
\]

vfad

\[
\begin{align*}
s+y(1) \\
s+y(2) \\
s+y(3) \\
s+y(4) \\
s+y(5) \\
s+y(6) \\
s+y(7) \\
s+y(8)
\end{align*}
\]
ex. 1: \( v = s + v \) (cont.)

- chaining, timing diagram:

- estimate: \( R < 4 \times R_0 \times \frac{1}{2} = 4000 \text{ MFlops} \)
- measured: \( R = 3969 \text{ MFlops} \)
ex. 1: $v = s + v$ (cont.)

- f90:
  
  $v(:) = s + w(:)$
  
  do $i=1,n$
  
  $v(i) = s + w(i)$
  
  end do

- f77:
  
  $v(i) = s + w(i)$
  
  end do

- what the compiler generates: ‘stripmining’, if $n>256$

  do $i0=1,n,256$
  
  do $i=i0,min(n,i0+255)$
  
  $v(i) = s + w(i)$
  
  end do
  
  end do
ex. 2: \( v = v + v \)

- **f90:**
  \[
  x(:,) = y(:,) + z(:,)
  \]

- **f77:**
  
  \[
  \text{do } i=1,n \\
  \hspace{1cm} x(i) = y(i) + z(i) \\
  \text{end do}
  \]

- **timing diagram:**

![Timing diagram](image-url)
ex. 2: \( v = v + v \) (cont.)

- timing diagram:

\[
\text{load} \quad \text{load} \quad \text{store} \quad \text{vfad}
\]

\[
\text{cycles}
\]

- estimate: \( R < 4 \times R_0 \times \frac{1}{3} = 2667 \text{ Mflops} \)
- about what is measured
ex. 3: \( v = v + s \cdot v \)

- **f90:**
  \[
  x(:) = y(:) + s \cdot z(:)
  \]

- **f77:**
  
  ```
  do i=1,n
      x(i) = y(i) + s * z(i)
  end do
  ```

- **timing diagram:**

![Timing diagram with cycles and load, store, vfad, vfmul instructions](image-url)
ex. 3: $v = v + s \times v$ (cont.)

- timing diagram:

```
do i=1,n
  x(i) = y(i) + s \times z(i)
end do
```

- estimate: $R < 4 \times R0 \times 2 / 3 = 5333$ Mflops
- about what is measured
**ex. 4: s = s + v * v**

- **f90:**
  
  ```f90```
  ```
  s = dot_product(x,y)
  ```
  ```
  ```

- **f77:**
  
  ```f77```
  ```
  do i=1,n
      s = s + x(i) * y(i)
  end do
  ```
  ```
  ```

- **Recursion? NO! Generated Code:**

  ```
  stemp(1:256) = 0.0
  do i0=1,n,256
      do i=i0,min(n,i0+255)
          stemp(i-i0+1) = stemp(i-i0+1) + x(i) * y(i)
      end do
  end do
  end do
  s = reduction(stemp)
  ```

Sometimes you have to do this by hand!
ex. 4: $s = s + v \times v$ (cont.)

- timing diagram:

![Timing Diagram]

- estimate: $R < 4 \times R_0 \times 2 / 2 = 8000$ Mflops
- about what is measured
ex. 5: matrix multiply

- FORTRAN:

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

- Multiplication of matrices
- Why is k inner loop? C. vectorizes usually inner loop
- saves a lot of stores (c is scalar)
- replaced by lib-call matmul (compiler!)
ex. 5: matrix multiply (cont.)

f90: opt(1800): matmul.f90, line 33: Idiom detected (matrix multiply).
Levels of Parallelism

- Segmentation Pipelining
- multiple pipes (4-fold)
- parallel usage of functional units
- parallel CPUs (later)
- parallel nodes (later)
Basic Rules to achieve good performance
Basic Rules to achieve Performance

- RAISING THE VECTORIZATION RATIO
  Ratio of the number of vector instructions to the total number of execution instructions

- Improving Vector Instruction Efficiency
Basic Rules for Performance

RAISING THE VECTORIZATION RATIO

The vectorization ratio can be improved by removing the cause of non-vectorization

```fortran
DO J=1,N
   X(J-1)=X(JW)*Y(J)
   JW=JW+1
END DO
```

In this example, the compiler cannot determine whether the correct dependency between definition and reference (X(J-1) on the left side and X(JW) on the right side) would be maintained because the initial value of JW is unknown.

Use !CDIR NODEP(X) if possible
Basic Rules for Performance

RAISING THE VECTORIZATION RATIO

The vectorization ratio can be improved by removing the cause of nonvectorization

!CDIR NODEP
DO J=1,N
   X(J-1)=X(JW)*Y(J)
   JW=JW+1
   WRITE(6,*) Y(J)
END DO
Basic Rules for Performance
RAISING THE VECTORIZATION RATIO

The vectorization ratio can be improved by removing the cause of nonvectorization

```plaintext
!CDIR NODEP
DO J=1,N
   X(J-1)=X(JW)*Y(J)
   JW=JW+1
   IF (JW.GT.100) GO TO 20
END DO
```
Basic Rules for Performance

RAISING THE VECTORIZATION RATIO

Loops containing user defined procedure references need to be expanded inline for vectorizing.

```
DO I=1,N
   CALL MAT(A(I),B(I),C(I),D(I),X,Y)
ENDDO

SUBROUTINE MAT(S,T,Y,V,A,B)
   A=S*U+T*V
   B=S*V-U*T
   RETURN
END
```

**Inline expansion**

```
DO I=1,N
   X=A(I)*C(I)+B(I)*D(I)
   Y=A(I)*D(I)-B(I)*C(I)
ENDDO
```

In most cases automatically applied by compiler.
Basic Rules for Performance
Improving Vector Instruction Efficiency

LENGTHENING THE LOOP

- Before a vectorized loop is executed, some preparatory processing must be accomplished for each vector instruction before the arithmetic begins.
- Start-up time is almost constant regardless of the loop length.
- Small loop length significantly reduces the efficiency of vectorization.

```
DO J=1,N
   DO I=1,M
      A(I,J)=X*B(I,J)+C(I,J)
   END DO
END DO
```

```
DO I=1,M
   DO J=1,N
      A(I,J)=X*B(I,J)+C(I,J)
   END DO
END DO
```

maximize the length of the innermost loop!
Basic Rules for Performance
Improving Vector Instruction Efficiency

IMPROVING ARRAY REFERENCE PATTERNS

To process data by vector instructions, a vector must be loaded from memory and stored again after processing. It does not always take the same time to load a vector and write it to memory again.

Loading and storing speed is **highest** for a continuous or a constant stride vector with odd stride (the interval between elements is an odd number).

![Array Elements](image)

Array elements in a loop to be vectorized should be referenced so that the index variables, such as the loop index variables, **appear in the first dimension** wherever possible. The values of subscript expressions should increment or decrement by 1 (or an odd number) at each loop iteration.
Basic Rules for Performance
Improving Vector Instruction Efficiency

IMPROVING ARRAY REFERENCE PATTERNS

REAL, DIMENSION (100,100) :: A, B, C

DO I=1,N
  DO J=1,N
    A(I,J) = B(I,J) + X*C(I,J)
  END DO
END DO

DO J=1,N
  DO I=1,N
    A(I,J) = B(I,J) + X*C(I,J)
  END DO
END DO
Basic Rules for Performance
Improving Vector Instruction Efficiency

REMOVING IF STATEMENTS

```
do I = 1, n
    if( i .eq. 1 ) then
        a(I) = b(1)
    else
        a(I) = 0.5 * ( b(I) + b(I-1) )
    end if
end do
```

-C hopt

```
a(1) = b(1)
do I = 2, n
    a(I) = 0.5 * ( b(I) + b(I-1) )
end do
```
Basic Rules for Performance

- Lengthening the Loop
- Improving Array Reference Patterns
- Removing If Statements
- Increasing Concurrency
Basic Rules for Performance: Concurrency

- Vector addition, subtraction, multiplication, vector shift operations and logical operations can be executed in parallel.
- Thus, it is efficient to put as many of these operations together in the same loop as possible.

**Example**

```plaintext
DO I=1,N
    A(I) = B(I) + C(I)
    X(I) = Y(I) * Z(I)
END DO
```

```plaintext
DO I=1,N
    A(I) = B(I) + C(I)
    X(I) = Y(I) * Z(I)
END DO
```
Basic Rules for Performance

- Lengthening the loop
- Improving array reference patterns
- Removing if statements
- Avoiding loop division
- Increasing concurrency
- Avoiding division
Basic Rules for Performance

Since vector division is slower than other vector arithmetic operations, minimize the number of divisions by converting them to multiplication or use algorithms that do not contain division.

Example:

```
  do i=1,1000
    a(i)=b(i)/value
  enddo
  ..

  temp=1/value
  do i=1,1000
    a(i)=b(i)*temp
  enddo
  ..
  ..
```
Basic Rules for Performance

- USING VECTORIZATION OPTIONS AND DIRECTIVES

  e.g. NOVECTOR option

  1) If branch-out occurs at element 2, it makes no sense to vectorize it. Please help the compiler.

  ```
  !CDIR NOVECTOR
  DO I=1,1000
      IF(A(I)-B(I)LT.1.0E-10) EXIT
      Z(I)=A(I)-B(I)
  END DO
  ```

  2) To test the effect of vectorization on accuracy
Basic Rules for Performance: Pointers

REAL, DIMENSION(:), POINTER:: X
REAL, DIMENSION(100), TARGET:: Y
DO I = 1, N
   X(I) = Y(I) * 2.0
END
DO

Since X may be associated with Y, the compiler assumes data dependency in the loop. Therefore, the loop is unvectorized.

If X is never associated with Y, you can specify the following compiler directive:

!CDIR NOOVERLAP(X, Y)

Then the compiler will vectorize the loop.
Basic Rules for Performance

Use variables for work space instead of using arrays.

Example 1:

```
DO I=1,N
   X=A(I)+B(I)
   Y=C(I)-D(I)
   E(I)=S*X+T*Y
   F(I)=S*Y+T*X
END DO
```

Example 2:

```
DO I=1,N
   WX(I)=A(I)+B(I)
   WY(I)=C(I)-D(I)
   E(I)=S*WX(I)+T*WY(I)
   F(I)=S*WY(I)-T*WX(I)
END DO
```

-C hopt creates work-arrays inefficient because additional loads are necessary

But sometimes it is needed (complex loops)
vectorization of if-blocks

• f90:

\[
\text{where}(y > 0.5) \\
\quad x = 1.0 + y \\
\text{elsewhere} \\
\quad x = y \times y \\
\text{end where}
\]

• f77:

\[
\text{do } i=1,n \\
\quad \text{if}( y(i) \text{.gt.} 0.5 ) \text{ then} \\
\quad \quad x(i) = 1.0 + y(i) \\
\quad \text{else} \\
\quad \quad x(i) = y(i) \times y(i) \\
\quad \text{end if} \\
\text{end do}
\]

• can be vectorized by using \textbf{mask-registers}
• complete loop is computed twice
• Mask selects correct values
### Vectorization of If-Blocks

#### But this method might be dangerous for functions like sqrt

<table>
<thead>
<tr>
<th>Vector Mask</th>
<th>Vector Mask Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 + v(1)</td>
<td>v(1) \times v(1)</td>
</tr>
<tr>
<td>1.0 + v(2)</td>
<td>1.0 + v(2)</td>
</tr>
<tr>
<td>1.0 + v(3)</td>
<td>v(3) \times v(3)</td>
</tr>
<tr>
<td>1.0 + v(4)</td>
<td>1.0 + v(4)</td>
</tr>
<tr>
<td>1.0 + v(5)</td>
<td>v(5) \times v(5)</td>
</tr>
<tr>
<td>1.0 + v(6)</td>
<td>1.0 + v(6)</td>
</tr>
<tr>
<td>1.0 + v(7)</td>
<td>v(7) \times v(7)</td>
</tr>
<tr>
<td>1.0 + v(8)</td>
<td>v(8) \times v(8)</td>
</tr>
</tbody>
</table>

**True**

**False**
vectorization of if-blocks

- **f90:**
  ```fortran
  where(y>=0.0) x = sqrt(y)
  
  do i=1,n
    if( y(i) .ge. 0.0 ) then
      x(i) = sqrt( y(i) )
    end if
  end do
  ```

- **f77:**
  ```fortran
  where(y>=0.0) x = sqrt(y)
  
  do i=1,n
    if( y(i) .ge. 0.0 ) then
      x(i) = sqrt( y(i) )
    end if
  end do
  ```

- problem if y(i) < 0.
- Alternate Method for one branch and expensive operations
- Vectorization by compiler
- Option or directive to force:
  - `-Wf"-pvctl compress"
  - `!CDIR COMPRESS`
vectorization of if-blocks

selects only indices „true“

use compress / expand

same mask can be used for expanding
vectorization of if-blocks

Example for manual compressing

```
DO I=1,N
   IF(A(I).NE.0.0)THEN
      C(I)=A(I)*SIN(B(I))
      :
      :
      END IF
END IF
END DO
```

Assume: N=1000, A has a value of more or less zero only for 40 elements

→ Gathering only the relevant data in consecutive work array in advance
vectorization of if-blocks

\[
\begin{align*}
K &= 0 \\
\text{DO } I &= 1, N \\
\text{IF } (A(I) \neq 0.0) \text{THEN} \\
& \quad K = K + 1 \\
& \quad IX(K) = I \\
& \quad AA(K) = A(I) \\
& \quad BB(K) = B(I) \\
& \quad \vdots \\
& \quad \vdots \\
& \quad \text{END IF} \\
\text{END DO} \\
\text{DO } I &= 1, K \\
& \quad CC(I) = AA(I) \times \sin(BB(I)) \\
& \quad \vdots \\
& \quad \vdots \\
& \quad \text{END DO} \\
\text{DO } I &= 1, K \\
& \quad C(IX(I)) = CC(I) \\
& \quad \vdots \\
& \quad \vdots \\
& \quad \text{END DO}
\end{align*}
\]
Another method is to generate vectors containing only the indexes of the relevant data (list vectors) without performing data compression and expansion. These vectors can then be used as subscripts directly in the loop.

\[ \text{K=0 DO I=1,N} \]
\[ \quad \text{IF(A(I).NE.0.0)THEN} \]
\[ \quad \quad \text{K=K+1} \]
\[ \quad \quad \text{IX(K)=I} \]
\[ \quad \text{END IF} \]
\[ \text{END DO DO I=1,K} \]
\[ \quad \text{C(IX(I))=A(IX(I))*SIN(B(IX(I))) : :} \]
\[ \text{END DO} \]

use !CDIR NODEP
constant if

• FORTRAN:

```fortran
  do I = 1, n
    if( ifirst .eq. 1 ) then
      a(I) = b(I)
    else
      a(I) = 0.5 * ( b(I) + bold(I) )
    end if
  end do
```

• solution (-Chopt): moves the if outside of the loop

```fortran
if( ifirst .eq. 1 ) then
  do I = 1, n
    a(I) = b(I)
  end do
else
  do I = 1, n
    a(I) = 0.5 * ( b(I) + bold(I) )
  end do
end if
```
‘boundary’ if

• FORTRAN:

```fortran
do I = 1, n
  if( i .eq. 1 ) then
    a(I) = b(1)
  else
    a(I) = 0.5 * ( b(I) + b(I-1) )
  end if
end do
```

• solution (-Chopt):

```fortran
a(1) = b(1)
do I = 2, n
  a(I) = 0.5 * ( b(I) + b(I-1) )
end do
```
ex. 5: matrix multiply (2)

• f90: 
  
  \[
  \begin{align*}
  c &= \matmult(a,b) \\
  \text{do } j=1,n \\
  \quad \text{do } i=1,n \\
  \quad \quad \text{do } k=1,n \\
  \quad \quad \quad c(i,j) &= c(i,j) + a(i,k) * b(k,j) \\
  \quad \quad \text{end do} \\
  \quad \text{end do} \\
  \text{end do}
  \end{align*}
  \]

• f77:

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

• which order of loops?
• totally different on assembler level
• replaced by library-call (compiler!)
ex. 5: matrix multiply (2)

FORTRAN equivalent to library call

```fortran
real accu(256)
do j=1,n
doi0=1,n,256
iend=min(n,i0+255)
do i=i0,iend
  accu(i)=c(i,j)
end do
do k=1,n
doi=i0,iend
  accu(i-i0+1)=accu(i-i0+1)+a(i,k)*b(k,j)
end do
do i=i0,iend
  c(i,j)=accu(i-i0+1)
end do
doi0=1,n,256
```

FORTRAN equivalent to library call
IO optimization

- F_SETBUF Modify I/O-Buffer (n Kb)
- avoid formatted IO
- vectorize IO:
  - slow:
    ```fortran
    real a(n,m)
    write(13)((a(i,j),i=2,n-1),j=2,m-1)
    ```
  - fast:
    ```fortran
    real a(n,m)
    write(13) a
    ```
Basic Rules for Performance

- vectorize important portions
- data parallelism or reduction for innermost loop
- long innermost loop
- lots of instructions in innermost loop
- avoid unnecessary memory traffic
- stride one or at least odd stride
- avoid indirect addressing
- keep loop structure ‘simple’
- avoid if-statements
Compiler directives
FORTRAN 90 !cdir

Support the compiler without changing your code
Informations only known by the user

- (no)altcode: affects generation of alternative code
- (no)assume: assume loop length
- (no)compress: compress / expand or masked operation
- (no)divloop: affects loop division for vectorisation
- loopcnt = … : define expected loopcnt
- nodep (most important): do vectorisation even if dependency might occur
- shortloop: loop length will not exceed vector register length
- (no)vector: do (not) vectorise loop if possible
- (no)overlap: for usage with pointers
• !CDIR ALTCODE
  generate alternative version of following loop
Default is ALTCODE

The compiler generates both a vectorized loop and a scalar loop for vectorizable loops. At execution, the most efficient loop is selected based on data dependencies and/or loop length
FORTRAN 90 !cdir

!CDIR COLLAPSE:

• The compiler collapses the specified loops
• But: check whether the arrays occupy contiguous areas.
• If not, execution result would be incorrect

```fortran
subroutine sub(a)    
    real,dimension(:,:,:) :: a
    !cdir collapse
    do k=1,l
      do j=1,m
        do i=1,n
          a(i,j,k) = ...
          end do
        end do
      end do
    end do
  end sub
```

```fortran
do i1=1,n*m*l
  a(i1,1,1) = ...
end do
```

```fortran
```

```
```
Stride for 1d-Arrays

- Example how to generate a stride \( \text{istride} \)

\[
\text{do } i=1, n, \text{istride} \\
\quad \text{a}(i) = b(i) \\
\text{end do}
\]

- Example: \( \text{istride} = 3 \)

Assume \text{stride} 8!

---

**Bank** 1 2 3 4 5 6 7 8  
\[
\begin{array}{cccccccc}
\text{a}(1) & \text{a}(2) & \text{a}(3) & \text{a}(4) & \text{a}(5) & \text{a}(6) & \text{a}(7) & \text{a}(8) \\
\text{a}(9) & \text{a}(10) & \text{a}(11) & \text{a}(12) & \text{a}(13) & \text{a}(14) & \text{a}(15) & \text{a}(16) \\
\text{a}(17) & \text{a}(18) & \text{a}(19) & \text{a}(20) & \text{a}(21) & \text{a}(22) & \text{a}(23) & \text{a}(24)
\end{array}
\]
Bank busy time measured

- \( v = s + v \)
  - constant loop count
  - varying strides!
  - banking.f90
- observations?
- lessons?

<table>
<thead>
<tr>
<th>Stride</th>
<th>SX-8</th>
<th>SX-6</th>
<th>SX-8</th>
<th>SX-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3938</td>
<td>1989.18</td>
<td>98.5%</td>
<td>99.5%</td>
</tr>
<tr>
<td>2</td>
<td>3928</td>
<td>1331.20</td>
<td>98.2%</td>
<td>66.6%</td>
</tr>
<tr>
<td>3</td>
<td>3986</td>
<td>1990.76</td>
<td>99.6%</td>
<td>99.5%</td>
</tr>
<tr>
<td>4</td>
<td>2984</td>
<td>707.69</td>
<td>74.6%</td>
<td>35.4%</td>
</tr>
<tr>
<td>5</td>
<td>3988</td>
<td>1991.08</td>
<td>99.7%</td>
<td>99.6%</td>
</tr>
<tr>
<td>6</td>
<td>3963</td>
<td>1331.20</td>
<td>99.1%</td>
<td>66.6%</td>
</tr>
<tr>
<td>8</td>
<td>1401</td>
<td>343.08</td>
<td>35.0%</td>
<td>17.2%</td>
</tr>
<tr>
<td>10</td>
<td>3956</td>
<td>1331.20</td>
<td>98.9%</td>
<td>66.6%</td>
</tr>
<tr>
<td>12</td>
<td>1999</td>
<td>707.69</td>
<td>50.0%</td>
<td>35.4%</td>
</tr>
<tr>
<td>16</td>
<td>574</td>
<td>146.62</td>
<td>14.4%</td>
<td>7.3%</td>
</tr>
<tr>
<td>32</td>
<td>265</td>
<td>67.33</td>
<td>6.6%</td>
<td>3.4%</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>67.33</td>
<td>3.2%</td>
<td>3.4%</td>
</tr>
<tr>
<td>128</td>
<td>127</td>
<td>67.31</td>
<td>3.2%</td>
<td>3.4%</td>
</tr>
<tr>
<td>256</td>
<td>120</td>
<td>66.60</td>
<td>3.0%</td>
<td>3.3%</td>
</tr>
<tr>
<td>512</td>
<td>63</td>
<td>36.44</td>
<td>1.6%</td>
<td>1.8%</td>
</tr>
<tr>
<td>1024</td>
<td>34</td>
<td>27.84</td>
<td>0.9%</td>
<td>1.4%</td>
</tr>
<tr>
<td>2048</td>
<td>17</td>
<td>14.79</td>
<td>0.4%</td>
<td>0.7%</td>
</tr>
<tr>
<td>4096</td>
<td>9</td>
<td>7.44</td>
<td>0.2%</td>
<td>0.4%</td>
</tr>
<tr>
<td>8096</td>
<td>266</td>
<td></td>
<td></td>
<td>6.6%</td>
</tr>
<tr>
<td>16384</td>
<td>9</td>
<td></td>
<td></td>
<td>0.2%</td>
</tr>
</tbody>
</table>
Optimization examples

- Loop interchange
- Loop expansion
- Loop division
- call to function
- 2 D recursion
loop interchange

- **FORTRAN:**

```fortran
  do j = 1, n
    do i = 2, n
      a(i,j) = a(i-1,j) * b(i,j) + c(i,j)
    end do
  end do
end do
```

- because of linear recurrence instruction, exchange of loops will improve performance

- switch indices? depends on leading dimension
loop interchange

• FORTRAN:

```fortran
  do i = 2, n
    do j = 1, n
      a(i,j) = a(i-1,j) * b(i,j) + c(i,j)
    end do
  end do
```

• no linear recurrence by exchange of loops
loop expansion

- **f77:**

```fortran
  do i=1,n
    do j=1,4
      a(i,j)=a(i,j)*b(i,j)+c(i,j)
    end do
  end do
```

-C vopt leads to

```fortran
  do i=1,n
    a(i,1)=a(i,1)*b(i,1)+c(i,1)
    a(i,2)=a(i,2)*b(i,2)+c(i,2)
    a(i,3)=a(i,3)*b(i,3)+c(i,3)
    a(i,4)=a(i,4)*b(i,4)+c(i,4)
  end do
```

How to control?
### Loop Expansion

```
sxf90 -Wf,-L fmtlist map summary transform
```

<table>
<thead>
<tr>
<th>LINE</th>
<th>LOOP</th>
<th>FORTRAN STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td></td>
<td>program test</td>
</tr>
<tr>
<td>2:</td>
<td></td>
<td>real a(100,4),b(100,4),c(100,4)</td>
</tr>
<tr>
<td>3:</td>
<td>V----&gt;</td>
<td>do i=1,100</td>
</tr>
<tr>
<td>4:</td>
<td></td>
<td>+-----</td>
</tr>
<tr>
<td>5:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td></td>
<td>+-----</td>
</tr>
<tr>
<td>7:</td>
<td>V-----</td>
<td>end do</td>
</tr>
<tr>
<td>8:</td>
<td></td>
<td>end</td>
</tr>
<tr>
<td>~</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
loop expansion

sxf90 -Wf,-L fmtlist map summary transform

<table>
<thead>
<tr>
<th>LINE</th>
<th>FORTRAN STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>program test</td>
</tr>
<tr>
<td>2</td>
<td>real a(100,4),b(100,4),c(100,4)</td>
</tr>
<tr>
<td>3</td>
<td>do i=1,100</td>
</tr>
<tr>
<td></td>
<td>!CDIR NODEP</td>
</tr>
<tr>
<td></td>
<td>do i = 1, 100</td>
</tr>
<tr>
<td>4</td>
<td>do j=1,4</td>
</tr>
<tr>
<td>5</td>
<td>a(i,j)=a(i,j)*b(i,j)+c(i,j)</td>
</tr>
<tr>
<td>6</td>
<td>end do</td>
</tr>
<tr>
<td></td>
<td>a(i,1) = a(i,1)*b(i,1) + c(i,1)</td>
</tr>
<tr>
<td></td>
<td>a(i,2) = a(i,2)*b(i,2) + c(i,2)</td>
</tr>
<tr>
<td></td>
<td>a(i,3) = a(i,3)*b(i,3) + c(i,3)</td>
</tr>
<tr>
<td></td>
<td>a(i,4) = a(i,4)*b(i,4) + c(i,4)</td>
</tr>
<tr>
<td>7</td>
<td>end do</td>
</tr>
<tr>
<td>8</td>
<td>end do</td>
</tr>
</tbody>
</table>
Problem: Recursion, inner loop not vectorizable
### loop division

- `C vopt` delivers:

<table>
<thead>
<tr>
<th>LINE</th>
<th>LOOP</th>
<th>FORTRAN STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td></td>
<td>program test</td>
</tr>
<tr>
<td>2:</td>
<td></td>
<td>real a(100,100), b(100,100), z(100,100), y(100,100), x(100,100)</td>
</tr>
<tr>
<td>3:</td>
<td>X----&gt;</td>
<td>do j=1,n</td>
</tr>
<tr>
<td>4:</td>
<td></td>
<td>do i=2,n</td>
</tr>
<tr>
<td>5:</td>
<td></td>
<td>b(i,j) = sqrt(x(i,j))</td>
</tr>
<tr>
<td>6:</td>
<td></td>
<td>a(i,j) = a(i-1,j)*b(i,j)+z(i,j)</td>
</tr>
<tr>
<td>7:</td>
<td></td>
<td>y(i,j) = sin(a(i,j))</td>
</tr>
<tr>
<td>8:</td>
<td></td>
<td>end do</td>
</tr>
<tr>
<td>9:</td>
<td>X----&gt;</td>
<td>end do</td>
</tr>
<tr>
<td>10:</td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
loop division (neues Beispiel machen!)

```
  !CDIR NODEP
  do j = 1, 100
    b1 = sqrt(x(2,j))
    a(2,j) = a(1,j)*b1 + z(2,j)
    y(2,j) = sin(a(2,j))
  end do
  do i = 2, 99, 2
    !CDIR NODEP
    do j = 1, 100
      b(i+1,j) = sqrt(x(i+1,j))
      b(i+2,j) = sqrt(x(i+2,j))
      a(i+1,j) = a(i,j)*b(i+1,j) + z(i+1,j)
      a(i+2,j) = a(i+1,j)*b(i+2,j) + z(i+2,j)
      y(i+1,j) = sin(a(i+1,j))
      y(i+2,j) = sin(a(i+2,j))
    end do
  end do
```

a and y are known for all j after 1. loop
call to function

- FORTRAN:

```fortran
    do i = 1, n
        y(i) = myfun(x(i))
    end do
```

- solution:
  - statement function
  - automatic inlining (sxf90 –pi auto)
  - Sometimes sxf90 –pi auto exp='function name'
  - Sometimes sxf90 –pi auto exp=..
expin=directory/filename
vectorization example

• Vectorization of Spray module

```
  do 100 idrop=1,ndrop
    do 200 while (drop_time.lt.gas_time)
      do 300 step=1,5
        compute derivatives
        update solution and drop-time
        compute error
      300 continue
      adjust drop timestep(depending on error)
      do special treatments (interactions etc.)
    200 continue
  100 continue
```

Outermost loop running over particles: not vectorizable
Vectorization example (2)

Vectorized Implementation (1/3)

```
nndrop=ndro
do 200 while (nndrop.gt.0)
   icount=0
   do idrop=1,nndrop
       if(drop_time(idrop).lt.gas_time) then
           icount=icount+1
           idrop_a(icount)=idrop
       end if
   end do
   nndrop=icount
```

Reduction of drops
Vectorized Implementation (2/3)

```fortran
  do 300 step=1,nstep
    do i=1,nndrop
      idrop=idrop_a(i)
      compute derivatives
    end do
    do i=1,nndrop
      idrop=idrop_a(i)
      update solution and drop-time
    end do
    do i=1,nndrop
      idrop=idrop_a(i)
      compute error
    end do
  end do
  300 continue
```

Runge-Kutta Timestep
• Vectorized Implementation (3/3)

```fortran
V-- do i=1,nndrop
   idrop=idrop_a(i)
   adjust drop timestep (depending on error)
V-- end do
V-- do i=1,nndrop
   idrop=idrop_a(i)
   do special treatments (interactions)
V-- end do
200 continue
```

Innermost loops running over particles: vectorizable
Execution Time in seconds

- Original: 6165
- Vectorized: 432
2D recursion

- FORTRAN:

```fortran
do j=2,n
   do i=2,m
      x(i,j) = rhs(i,j) - a(i,j)*x(i-1,j) - b(i,j)*x(i,j-1)
   end do
end do
```

- solution:
  hyperplane-ordering:
2D recursion (2)

- **FORTRAN** (needs directive!):

```fortran
!CDIR NODEP
do idiag=1,m+n-1
   ! i = injective!
   ! works for general cases, too (i.e. unstructured grids)
   do j = max(1,idiag+1-m), min(n,idiag)
      i=idiag+1-j
      x(i,j)=rhs(i,j)-a(i,j)*x(i-1,j)-b(i,j)*x(i,j-1)
   end do
end do
```

- challenge: get indices and loop parameters right!
- i=injective!
- works for general cases, too (i.e. unstructured grids)
**SUPER-UX R14.1 Manual Global Contents**

Please refer to

<table>
<thead>
<tr>
<th>General Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>User's Guide</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmer's Guide</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Programmer's Guide</td>
</tr>
<tr>
<td>STREAMS Programmer's Guide</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Window System</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Windows System User's Guide</td>
</tr>
<tr>
<td>Xlib Programming Manual</td>
</tr>
<tr>
<td>X Toolkit Programming Manual</td>
</tr>
<tr>
<td>X Window System Programmer's Guide</td>
</tr>
<tr>
<td>Mf/SX User's Guide</td>
</tr>
<tr>
<td>Mf/SX Programmer's Guide</td>
</tr>
<tr>
<td>Mf/SX Style Guide</td>
</tr>
<tr>
<td>Mf/SX Programmer's Reference</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN90/SX Language Reference Manual</td>
</tr>
<tr>
<td>FORTRAN90/SX Programmer's Guide</td>
</tr>
<tr>
<td>FORTRAN90/SX Multitasking User's Guide</td>
</tr>
<tr>
<td>MPI/SX User's Guide</td>
</tr>
<tr>
<td>Assembly Language Reference Manual</td>
</tr>
<tr>
<td>DBX User's Guide</td>
</tr>
<tr>
<td>PDBX User's Guide</td>
</tr>
<tr>
<td>XDBX User's Guide</td>
</tr>
<tr>
<td>OpenGL Programmer's Guide</td>
</tr>
<tr>
<td>PSUITE User's Guide</td>
</tr>
<tr>
<td>C++/SX Programmer's Guide</td>
</tr>
<tr>
<td>HPF/SX V2 User's Guide</td>
</tr>
<tr>
<td>HPF/SX V2 User's Guide</td>
</tr>
</tbody>
</table>
### FORTRAN90/SX Programmer's Guide Contents

#### Chapter 1 Overview
1.1 FORTRAN90/SX Characteristics  
1.2 FORTRAN90/SX Outline

#### Chapter 2 Compilation and Execution
2.1 Self Compiler and Cross Compiler  
2.2 f90 and sxf90 Command Format  
2.3 Files Generated by the Compiler  
2.4 Execution

#### Chapter 3 Directives
3.1 Option Directive Lines  
3.2 Compiler Directive Lines

#### Chapter 4 Optimization Features
4.1 Code Optimization  
4.2 Inline Expansion of Subprograms  
4.3 Optimizing by Instruction Scheduling  
4.4 Notes on the Optimization Function

#### Chapter 5 Vectorization Functions
5.1 Vectorization Conditions and Examples of Vectorization  
5.2 Vector Optimization  
5.3 Using Vectorization  
5.4 An Object for Maximum Vector Register Length, 512 (VL512)

#### Chapter 6 Program Notes Depending on the Language Specification
6.1 Comment Lines  
6.2 Hollerith Type  
6.3 Arrays and Substrings  
6.4 Expressions  
6.5 COMMON and EQUIVALENCE Statements  
6.6 EXTERNAL Statement  
6.7 Data Initialization Statements  
6.8 FORMAT 2 POINTER Statement  
6.9 ASSIGNMENT Statements  
6.10 IF Statements  
6.11 DO Statements and Implied DO Lists  
6.12 STOP and PAUSE Statements  
6.13 Input/Output Statements  
6.14 Program Unit and Statement Function  
6.15 INCLUDE Line  
6.16 Module  
6.17 Internal Representation of Data  
6.18 Data Types  
6.19 Compatibility Features  
6.20 Linkage with the C Language

#### Chapter 7 Run-Time Input/Output
7.1 File Organization  
7.2 Records and Files  
7.3 Preconnection  
7.4 Unnamed File  
7.5 Unique File Names  
7.6 INQUIRE Statement Return Values  
7.7 High-Speed I/O Functions  
7.8 Data Conversion Function  
7.9 Special Numbers in Floating-Point Data Format FLOAT0  
7.10 Maximum Record Length

#### Chapter 8 Run-Time Error Handling
8.1 Handling Arithmetic Exceptions  
8.2 Error-Handling Flow  
8.3 Intrinsic Subroutines for Error Control  
8.4 Error-Handling Control Information

#### Chapter 9 Program Debugging
9.1 Symbolic Debugger  
9.2 Debug Line  
9.3 Restrictions for Debugging at Optimization, Vectorization and Parallelization

#### Chapter 10 Program Tuning
10.1 Achieving Speed Through Vectorization  
10.2 High-Speed I/O Techniques  
10.3 Simple Performance Analysis Function

#### Chapter 11 Compile-Time Output Lists
11.1 Source Program List  
11.2 Cross Reference List  
11.3 Object List  
11.4 Summary List  
11.5 Transformation List  
11.6 Message List  
11.7 Format List
Basic Rules for Performance

• vectorize important portions
• data parallelism or reduction for innermost loop
• long innermost loop
• lots of instructions in innermost loop
• stride one or at least odd stride
• avoid indirect addressing
• keep loop structure ‘simple’
End of THEORY II