Vectorization examples

Basics: How to calculate the performance
Vectorization examples

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

- \[ \mathbf{v} = \mathbf{s} + \mathbf{v} \]
- \[ \mathbf{v} = \mathbf{v} + \mathbf{v} \]
- \[ \mathbf{v} = \mathbf{v} + \mathbf{s} \times \mathbf{v} \]
- \[ \mathbf{s} = \mathbf{s} + \mathbf{v} \times \mathbf{v} \]
- matrix multiply

ex. 1: \( \mathbf{v} = \mathbf{s} + \mathbf{v} \) (cont.)

- timing diagram 1

\[
\begin{align*}
\text{do } i=1,n \\
v(i) &= s + w(i) \\
\text{end do}
\end{align*}
\]

@ 565 MHZ

load

vfad

store

3 cycles are needed to perform one calculation
\[ R < 8 \times R_0 \times 1/3 = 1507 \text{ MFlops} \quad (R_0 = 565 \text{ MFLOPs}) \]
measured: 2258 MFlops

What’s wrong?
Independent LOAD/STORE Unit

- **Central Processing Unit**
  - 565 MHz System Clock
  - Mask Reg.
  - Logical
  - Multiply
  - Add/Shift
  - Divide
  - Vector Registers
  - Scalar Registers
  - Scalar Execution unit
  - 8-set Vector Pipelines
  - Max. 8 CPU

- **Shared Main Memory**
  - 36 GB/s Bandwidth per CPU
  - Mask Reg.
  - Logical
  - Multiply
  - Add/Shift
  - Divide
  - Vector Registers
  - Scalar Registers
  - Scalar Execution unit

- **Input/Output Processor (IOP)**
  - Max. 4 IOP

- **Scalar Unit**
  - Scalar Execution unit
  - Scalar Register

- **Cache Memory**
  - Load/Store
  - Mask Reg.
  - Logical
  - Multiply
  - Add/Shift
  - Divide
  - Vector Registers
  - Scalar Registers
  - Scalar Execution unit

- **Memory**
  - y(1)
  - y(2)
  - y(3)
  - y(4)
  - y(5)
  - y(6)
  - y(7)
  - y(8)

- **vfad**

**Chaining**

- y(1) + s
- y(2) + s
- y(3) + s
- y(4) + s
- y(5) + s
- y(6) + s
- y(7) + s
- y(8) + s

**Scalar Performance**
- 1,13 GFLOPS

**System Clock**
- 565 MHz

**Max. 4 IOP**

**Max. 8 CPU**

**36 GB/s Bandwidth per CPU**

**1,13 GFLOPS**

**Scalar Performance**

**vfad**
• chaining, timing diagram:

```
load          store
vfad
```

cycles 2

• estimate: \( R < 8 \times R_0 \times \frac{1}{2} = 2260 \text{ MFlops} \)
• measured: \( R = 2258 \text{ MFlops} \)

ex. 1: \( v = s + v \) (cont.)

ex. 1: \( v = s + v \)

• f90:
  \[
  v(:) = s + w(:)
  \]

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

- Timing Diagram SX-6+:

  ![Timing Diagram SX-6+](image)

  cycles

ex. 2: \( v = v + v \) (cont.)

- Timing Diagram SX-6+:

  ![Timing Diagram SX-6+](image)

  cycles

- estimate: \( R < 8 \times R_0 \times \frac{1}{3} = 1507 \text{ Mflops} \)
- about what is measured
ex. 3: \( v = v + s \cdot v \)

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

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

- **Timing Diagram SX-6+:**
  
  ![Timing Diagram]

  - store
  - vfad
  - vfmul

  Cycles

**ex. 3: \( v = v + s \cdot v \) (cont.)**

- **Timing Diagram SX-6+:**
  
  ![Timing Diagram]

  - do i=1,n \\
  x(i) = y(i) + s \cdot z(i) \\
  \text{end do}

  - store
  - vfad
  - vfmul

  Cycles

- estimate: \( R < 8 \cdot R_0 \cdot 2 / 3 = 3013 \text{ Mflops} \)
- about what is measured
ex. 4: $s = s + v \cdot v$

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

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

- Recursion? NO! Generated Code:
  ```fortran
  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
  s = reduction(stemp)
  ```

Sometimes you have to do this by hand

ex. 4: $s = s + v \cdot v$ (cont.)

- Timing Diagram SX-6+:
  ```text
  load load vfad vfmul cycles
  ```

- $R < 8 \times R_0 \times \frac{2}{2} = 4520$ Mflops

about what was 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
```

- 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. s1: matrix multiply

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

- Segmentation Pipelining √
- multiple pipes (8-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 nonvectorization

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

```fortran
!CDIR NODEP
DO J=1,N
   X(J-1)=X(JW)*Y(J)
   JW=JW+1
   WRITE(6,*) Y(J)
END DO
```

```fortran
!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
```

In most cases automatically applied by compiler.

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
```

Basic Rules for Performance

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
```

```
N=10000, M=10
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!
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 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.
do I = 1, n
if( i .eq. 1 ) then
   a(I) = b(I)
else
   a(I) = 0.5 * ( b(I) + b(I-1) )
end if
end do

-C hopf

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

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

```plaintext
temp=1/value

do i=1,1000
   a(i)=b(i)*temp
endo

.. ..

do i=1,1000
   a(i)=b(i)/value
endo

.. ..
```

**Examples:**

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

```plaintext
!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

```fortran
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:**

```fortran
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:**

```fortran
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)

Empowered by Innovation

NEC
vectorization of if-blocks

- f90:
  ```fortran
  where(y>0.5)
  x = 1.0 + y
elsewhere
  x = y * y
end where
  ```

- f77:
  ```fortran
  do i=1,n
    if( y(i) .gt. 0.5 ) then
      x(i) = 1.0 + y(i)
    else
      x(i) = y(i) * y(i)
    end if
  end do
  ```

- can be vectorized by using **mask-registers**
- complete loop is computed twice
- Mask selects correct values

**But this method might be dangerous for functions like sqrt**
vectorization of if-blocks

- f90:
  ```fortran
  where(y>=0.0) x = sqrt(y)
  ```

- f77:
  ```fortran
  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`

```
do i=1,n
  if( y(i) .ge. 0.0 ) then
    x(i) = sqrt( y(i) )
  end if
end do
```

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

Example for manual compressing

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

Work arrays
```
K=0
DO I=1,N
  IF(A(I).NE.0.0)THEN
    K=K+1
    IX(K)=I
    AA(K)=A(I)
    BB(K)=B(I)
  : : 
  END IF
END DO
DO I=1,K
  CC(I)=AA(I)*SIN(BB(I))
  : : 
END DO
DO I=1,K
  C(IX(I))=CC(I)
  : : 
END DO
```

Compression
Computation
Expansion
vectorization of if-blocks

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.

```fortran
K=0
DO I=1,N
  IF(A(I).NE.0.0)THEN
    K=K+1
    IX(K)=I
  END IF
END DO
DO I=1,K
  C(IX(I))=A(IX(I))*SIN(B(IX(I)))
END DO
```

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. s1: matrix multiply

• f90:

```fortran
c = matmult(a,b)
```

• f77:

```fortran
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. s1: matrix multiply(2)

FORTRAN equivalent to library call

```fortran
real accu(512)
do j=1, n
  do i0=1, n, 512
    iend=min(n, i0+511)
    do i=i0, iend
      accu(i)=c(i, j)
    end do
  end do
  do k=1, n
    do i=i0, iend
      accu(i-i0+1)=accu(i-i0+1)+a(i, k)*b(k, j)
    end do
  end do
  do i=i0, iend
    c(i, j)=accu(i-i0+1)
  end do
end do
```

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

FORTRAN 90 !cdir

- !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`

ICDIR 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
```

Stride for 1d-Arrays

- Example how to generate a stride `istride`

```fortran
  do i=1,n,istride  
a(i) = b(i)  
  end do
```

- Example: `istride = 3`

<table>
<thead>
<tr>
<th>Rank</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a(1)</td>
<td>a(2)</td>
<td>a(3)</td>
<td>a(4)</td>
<td>a(5)</td>
<td>a(6)</td>
<td>a(7)</td>
<td>a(8)</td>
</tr>
<tr>
<td></td>
<td>a(9)</td>
<td>a(10)</td>
<td>a(11)</td>
<td>a(12)</td>
<td>a(13)</td>
<td>a(14)</td>
<td>a(15)</td>
<td>a(16)</td>
</tr>
<tr>
<td></td>
<td>a(17)</td>
<td>a(18)</td>
<td>a(19)</td>
<td>a(20)</td>
<td>a(21)</td>
<td>a(22)</td>
<td>a(23)</td>
<td>a(24)</td>
</tr>
</tbody>
</table>

ASSUME stride 8!
Bank busy time measured

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

- observations?
- lessons?

<table>
<thead>
<tr>
<th>case</th>
<th>stride</th>
<th>time</th>
<th>performance</th>
<th>factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5.027E-05</td>
<td>1989.18</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>7.512E-05</td>
<td>1331.20</td>
<td>1.49</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5.023E-05</td>
<td>1990.76</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.413E-04</td>
<td>707.69</td>
<td>2.81</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5.022E-05</td>
<td>1991.08</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7.512E-05</td>
<td>1331.20</td>
<td>1.49</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>2.915E-04</td>
<td>343.08</td>
<td>5.80</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>7.512E-05</td>
<td>1331.20</td>
<td>1.49</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>1.413E-04</td>
<td>707.69</td>
<td>2.81</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>6.820E-04</td>
<td>146.62</td>
<td>13.57</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
<td>2.914E-04</td>
<td>343.15</td>
<td>5.80</td>
</tr>
<tr>
<td>12</td>
<td>32</td>
<td>1.485E-03</td>
<td>67.33</td>
<td>29.54</td>
</tr>
<tr>
<td>13</td>
<td>48</td>
<td>6.840E-04</td>
<td>146.20</td>
<td>13.61</td>
</tr>
<tr>
<td>14</td>
<td>64</td>
<td>1.485E-03</td>
<td>67.33</td>
<td>29.54</td>
</tr>
<tr>
<td>15</td>
<td>128</td>
<td>1.486E-03</td>
<td>67.31</td>
<td>29.55</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>1.501E-03</td>
<td>66.60</td>
<td>29.87</td>
</tr>
<tr>
<td>17</td>
<td>512</td>
<td>2.749E-03</td>
<td>36.44</td>
<td>54.59</td>
</tr>
<tr>
<td>18</td>
<td>1024</td>
<td>3.592E-03</td>
<td>27.84</td>
<td>71.46</td>
</tr>
<tr>
<td>19</td>
<td>2048</td>
<td>6.761E-03</td>
<td>14.79</td>
<td>134.50</td>
</tr>
<tr>
<td>20</td>
<td>4096</td>
<td>1.344E-02</td>
<td>7.44</td>
<td>267.28</td>
</tr>
<tr>
<td>21</td>
<td>4097</td>
<td>5.042E-05</td>
<td>1983.18</td>
<td>1.00</td>
</tr>
<tr>
<td>22</td>
<td>8192</td>
<td>1.344E-02</td>
<td>7.44</td>
<td>267.28</td>
</tr>
</tbody>
</table>

Optimization examples

- Loop interchange
- Loop expansion
- Loop division
- call to function
- 2 D recursion
- indirect addressing
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
```

- in spite 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

- $f_{77}$:

\[
\begin{align*}
&\text{do } i=1,n \\
&\quad \text{do } j=1,4 \\
&\quad \quad a(i,j)=a(i,j) \times b(i,j)+c(i,j) \\
&\quad \text{end do} \\
&\text{end do}
\end{align*}
\]

-C vopt leads to

\[
\begin{align*}
&\text{do } i=1,n \\
&\quad a(i,1)=a(i,1) \times b(i,1)+c(i,1) \\
&\quad a(i,2)=a(i,2) \times b(i,2)+c(i,2) \\
&\quad a(i,3)=a(i,3) \times b(i,3)+c(i,3) \\
&\quad a(i,4)=a(i,4) \times b(i,4)+c(i,4) \\
&\text{end do}
\end{align*}
\]

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>do j=1,4</td>
</tr>
<tr>
<td>5:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td></td>
<td>end do</td>
</tr>
<tr>
<td>7:</td>
<td>V----&gt;</td>
<td>end do</td>
</tr>
<tr>
<td>8:</td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
**loop expansion**

<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>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>
</tbody>
</table>

sxf90 -Wf,-L fmtlist map summary transform

**loop division**

- f77:

```fortran
  do j=1, n
    do i=2, n
      b(i, j) = sqrt(x(i, j))
      a(i, j) = a(i-1, j) * b(i, j) + z(i, j)
      y(i, j) = sin(a(i, j))
    end do
  end do
```

**Problem:** Recursion, inner loop not vectorizable, sqrt is expensive
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-----</td>
<td>end do</td>
</tr>
<tr>
<td>10:</td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

Unrolling and interchange
bl is constant in 1. loop (j)
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

```fortran
real function myfun( a )
  myfun = sqrt(a)
  return
end
```

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

vectorization example

- Vectorization of Spray module

```fortran
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)

```fortran
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
end do
```

Reduction of drops

vectorization example (3)

- 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
Innermost loops running over particles: vectorizable

```
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
```
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
do idiag=1,m+n-1
  !CDIR NODEP
  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)
Please refer to

General Usage
- User's Guide

Programming
- Programmer's Guide
- Programmer's Reference Manual

Batch Processing
- User's Guide

Window System
- X Windows System User's Guide
- Xlib Programming Manual
- X Toolkit Programming Manual
- X Window System Programmer's Guide
- X/SX Programmer's Reference

Language
- Programming Language Support Reference Manual
- C Programmer's Guide
- FORTRAN90/SX Language Reference Manual
- FORTRAN90/SX Programmer's Guide
- FORTRAN90/SX Multitasking User's Guide
- MPI/SX User's Guide
- DBX User's Guide
- PDBX User's Guide
- XDBX User's Guide
- PSUITE User's Guide
- C++/SX Programmer's Guide
- FSA/SX User's Guide

FORTRAN90/SX Programmer's Guide Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview</td>
</tr>
<tr>
<td>2</td>
<td>Compilation and Execution</td>
</tr>
<tr>
<td>3</td>
<td>Directives</td>
</tr>
<tr>
<td>4</td>
<td>Optimization Features</td>
</tr>
<tr>
<td>5</td>
<td>Vectorization Functions</td>
</tr>
<tr>
<td>6</td>
<td>Program Notes Depending on the Language Specification</td>
</tr>
<tr>
<td>7</td>
<td>Run-Time Input/Output</td>
</tr>
<tr>
<td>8</td>
<td>Run-Time Error Handling</td>
</tr>
<tr>
<td>9</td>
<td>Program Debugging</td>
</tr>
<tr>
<td>10</td>
<td>Program Tuning</td>
</tr>
<tr>
<td>11</td>
<td>Compile-Time Output Lists</td>
</tr>
<tr>
<td>12</td>
<td>XMU Array Function</td>
</tr>
</tbody>
</table>

7.10 Maximum Record Length
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’