THEORY II

Vectorization examples

Basics: How to calculate the performance

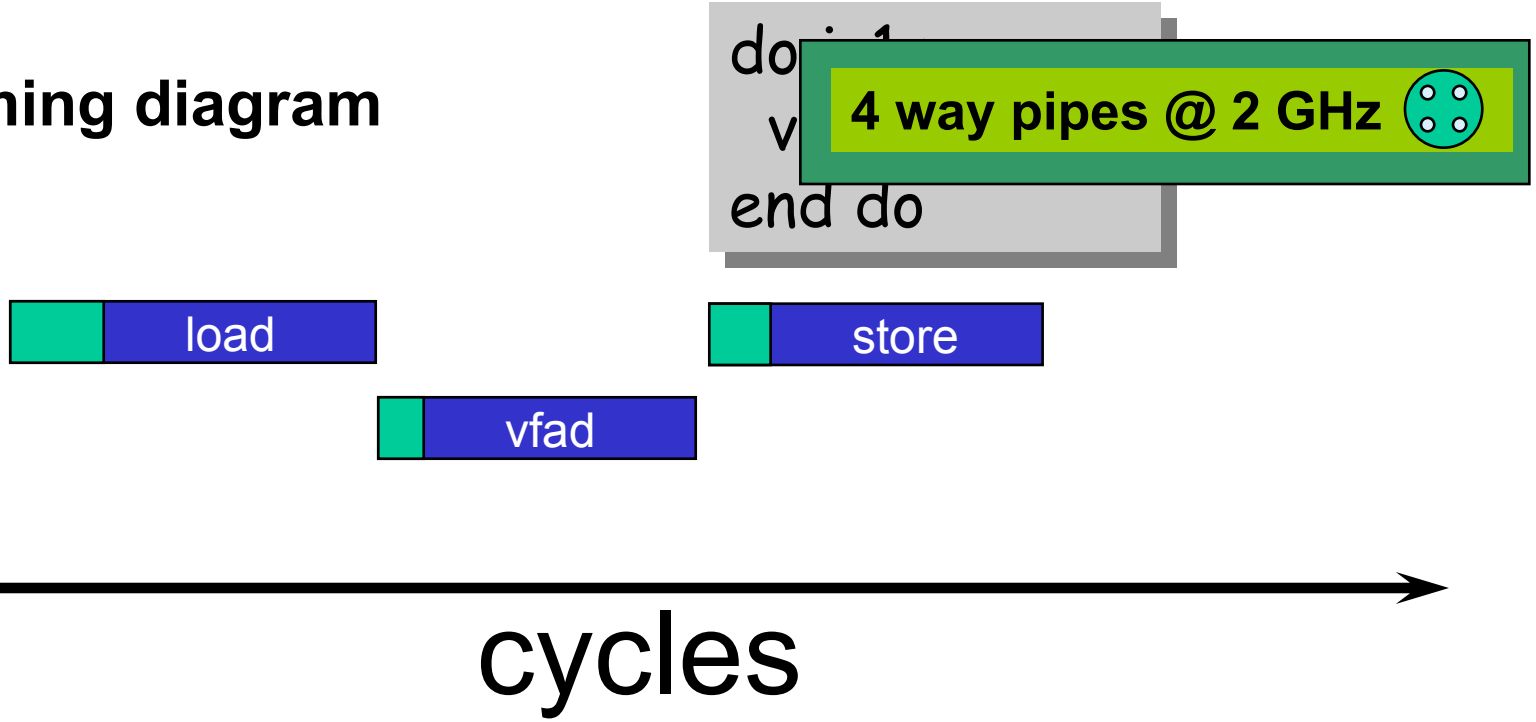
Vectorization examples

v = Vector s = Scalar

- v = s + v
- v = v + v
- v = v + s * v
- s = s + v * v
- matrix multiply

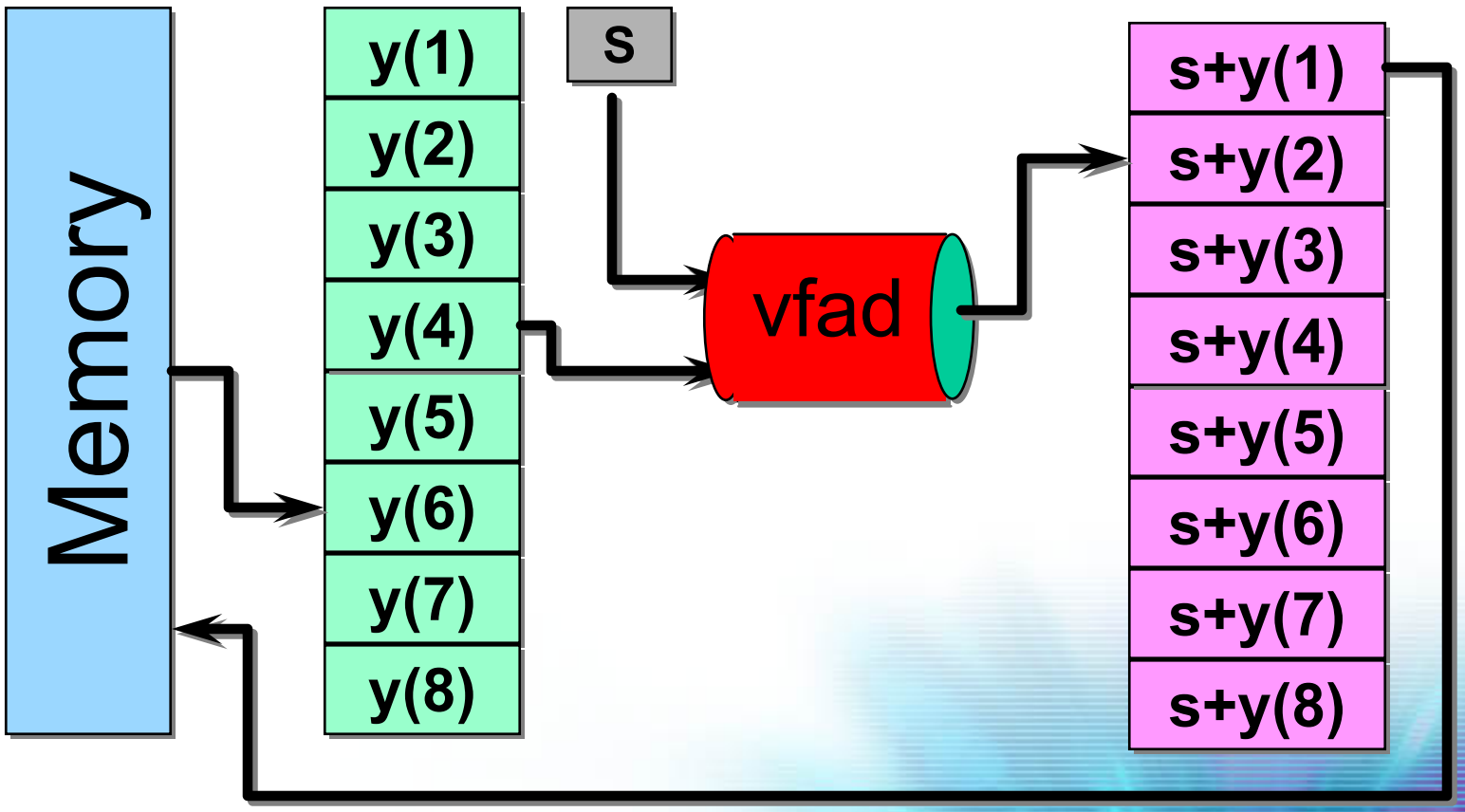
ex. 1: v = s + v

- timing diagram



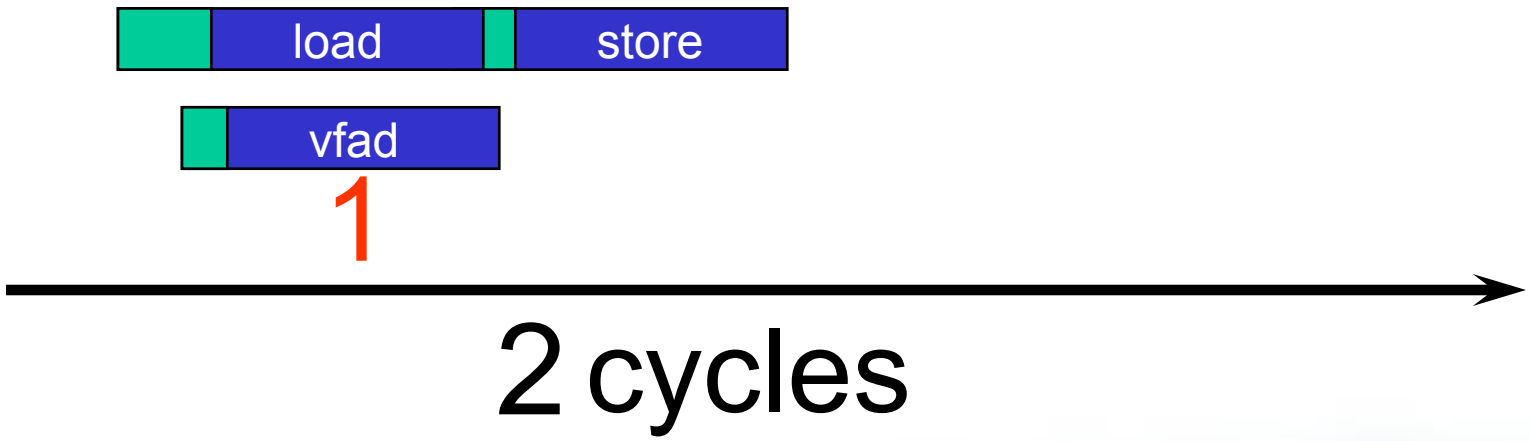
- 3 cycles are needed to perform one calculation
- R < 4 * R0 * 1 / 3 = 2667 MFlops (R0 = 2000 MFlops)
- measured: 3969 MFlops
- What's wrong?

Chaining


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

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

- chaining, timing diagram:



- estimate: R < 4 * R0 *1/ 2 = 4000 MFlops
- measured: R = 3969 MFlops

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

- 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

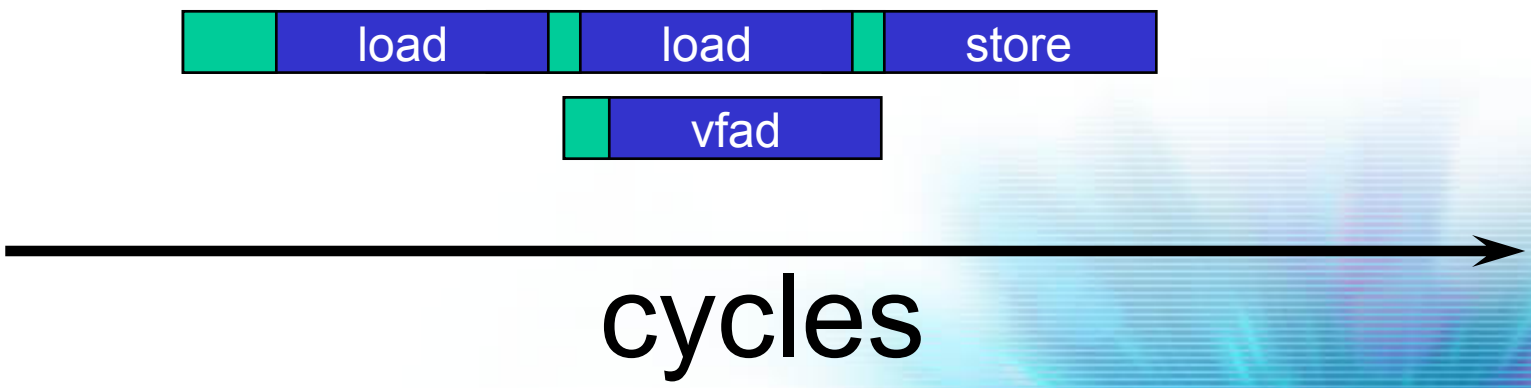
- f90:

```
x(:) = y(:) + z(:)
```

- f77:

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

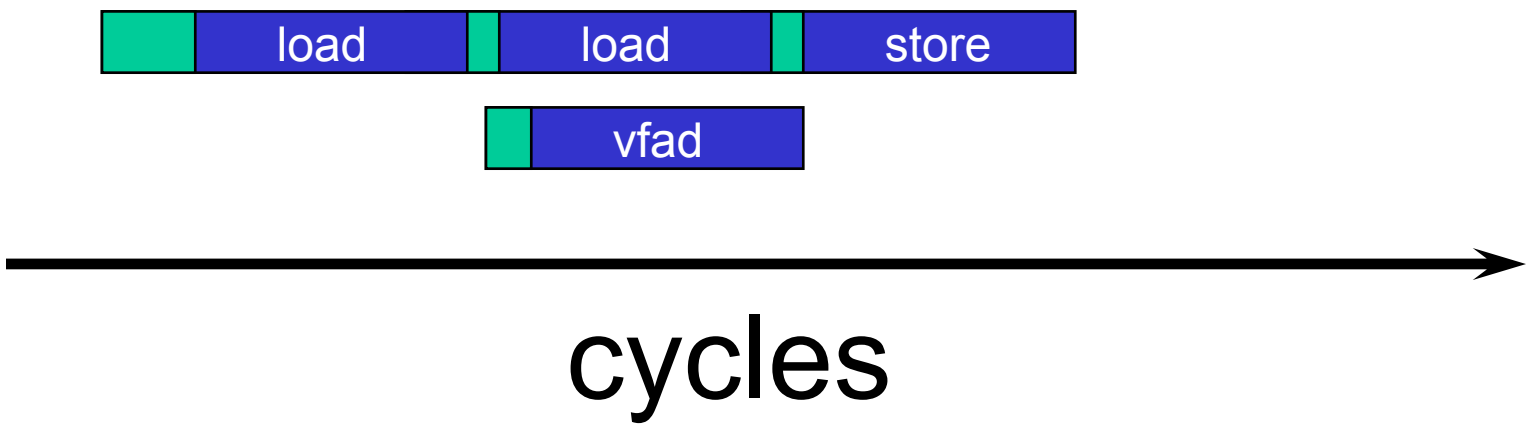
- timing diagram:



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

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

- timing diagram:



- estimate: R < 4 * R0 * 1 / 3 = 2667 Mflops
- about what is measured

ex. 3: v = v + s * v

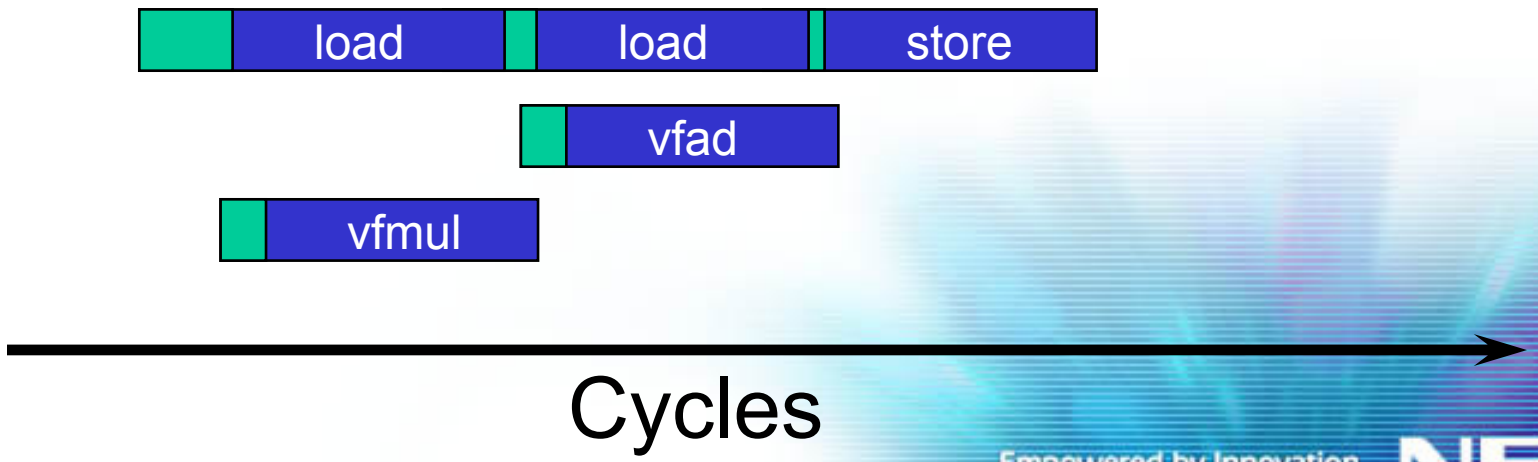
- f90:

```
x(:) = y(:) + s * z(:)
```

- f77:

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

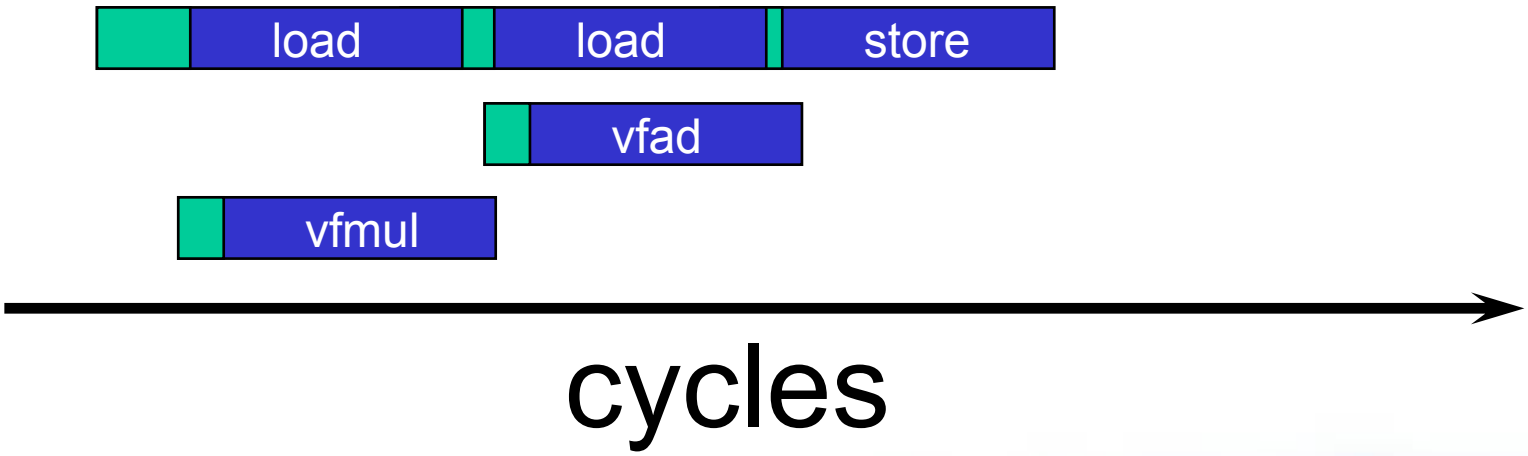
- timing diagram:



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

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

- timing diagram:



- estimate: R < 4 * R0 * 2 / 3 = 5333 Mflops
- about what is measured

ex. 4: s = s + v * v

- f90:

```
s = dot_product(x,y)
```

- f77:

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

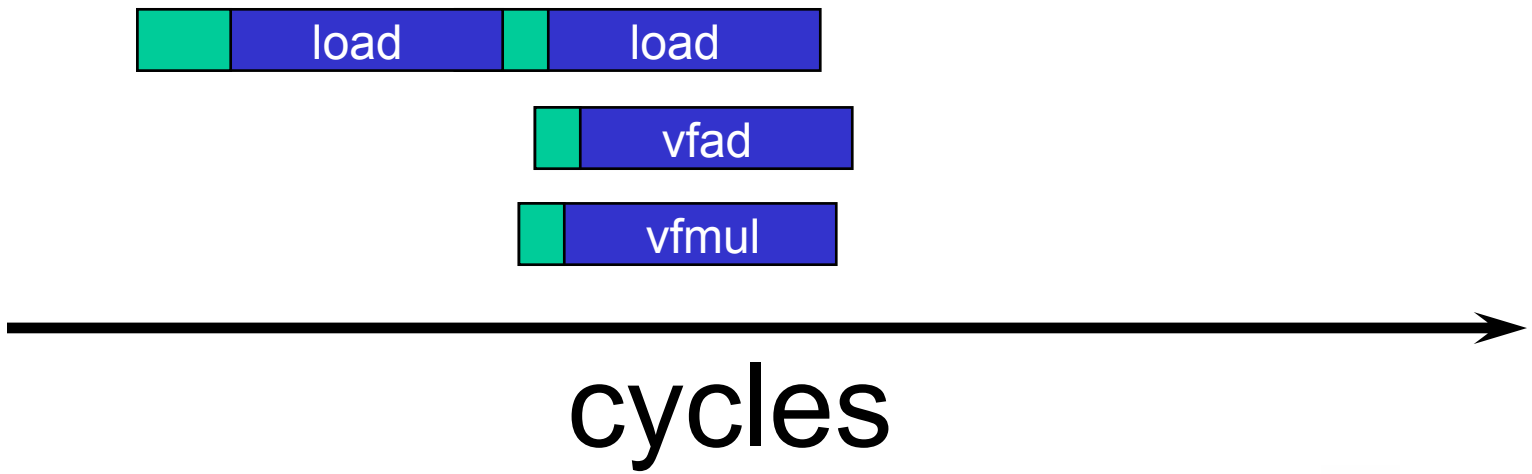
- Recursion? NO! Generated Code:

```
stemp(1:256) = 0.0
do i0=1,n,256
  do i=i0,min(n,i0+255)
    stemp(i-i0+1) = stemp(i-io+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 * v (cont.)

- timing diagram:



- estimate: R < 4 * R0 * 2 / 2 = 8000 Mflops
- about what is measured

ex. 5: matrix multiply

- 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. 5: matrix multiply (cont.)

```
f90: vec(4): matmul.f90, line 22: Vectorized array expression.
f90: vec(4): matmul.f90, line 22: Vectorized array expression.
f90: vec(4): matmul.f90, line 25: Vectorized array expression.
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 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

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

```
!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
```

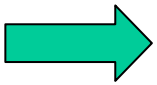
Basic Rules for Performance

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
```

⇓

```
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!

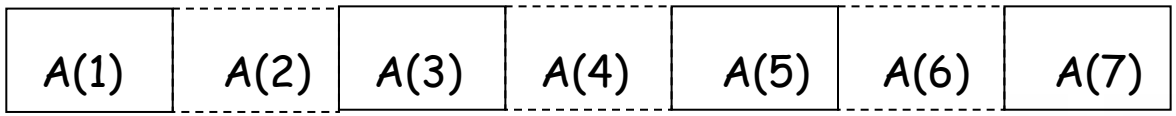
Basic Rules for Performance

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).



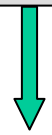
Array elements in a loop to be vectorized should be referenced so that the index variables, such as the loop index variables, **appear in the first dimension** wherever possible. The values of subscript expressions should increment or decrement by 1 (or an odd number) at each loop iteration.

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

```
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:

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

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

NEC

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:

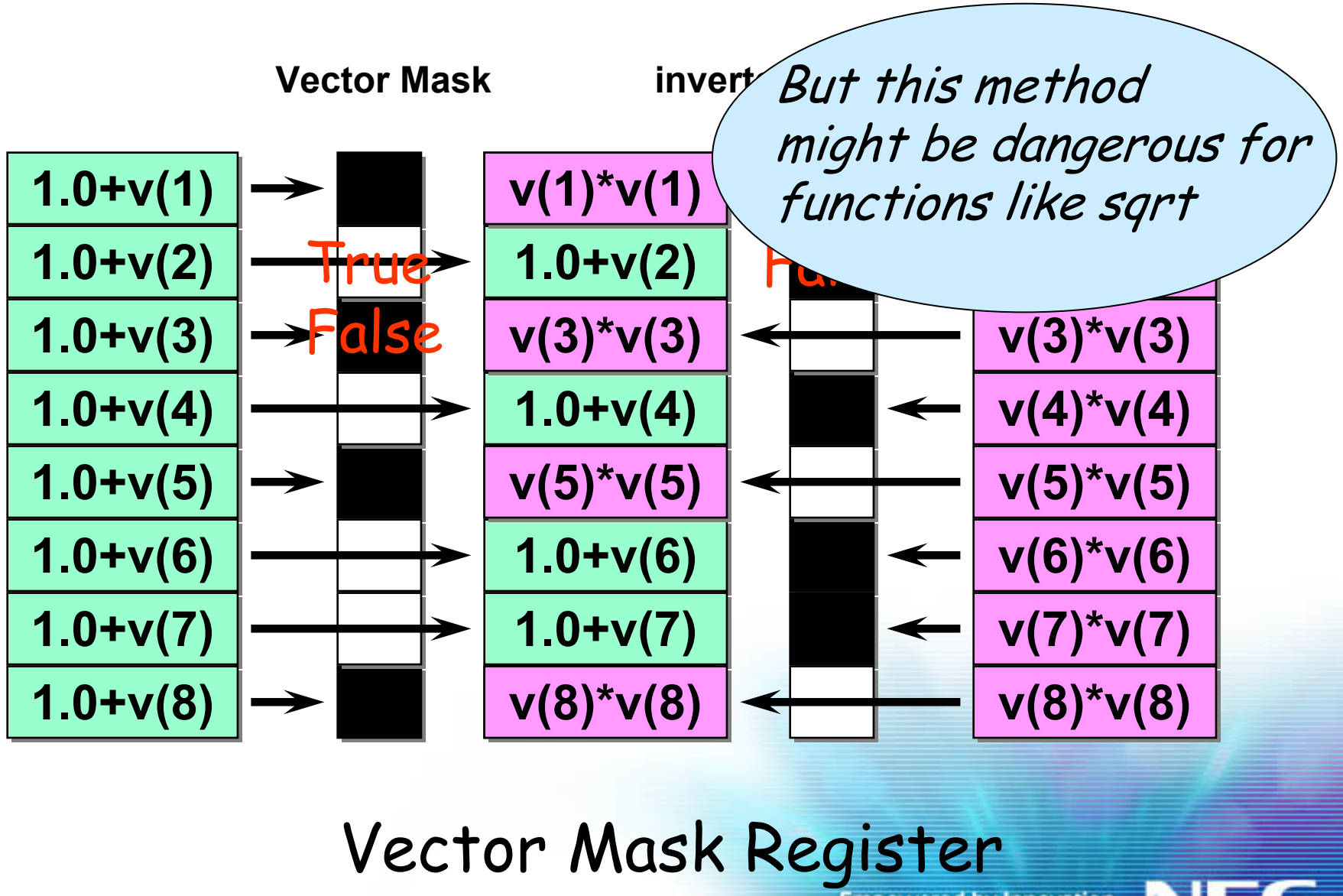
```
where(y>0.5)
  x = 1.0 + y
elsewhere
  x = y * y
end where
```

- f77:

```
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

vectorization of if-blocks



Vector Mask Register

vectorization of if-blocks

- f90:

```
where(y>=0.0) x = sqrt(y)
```

- f77:

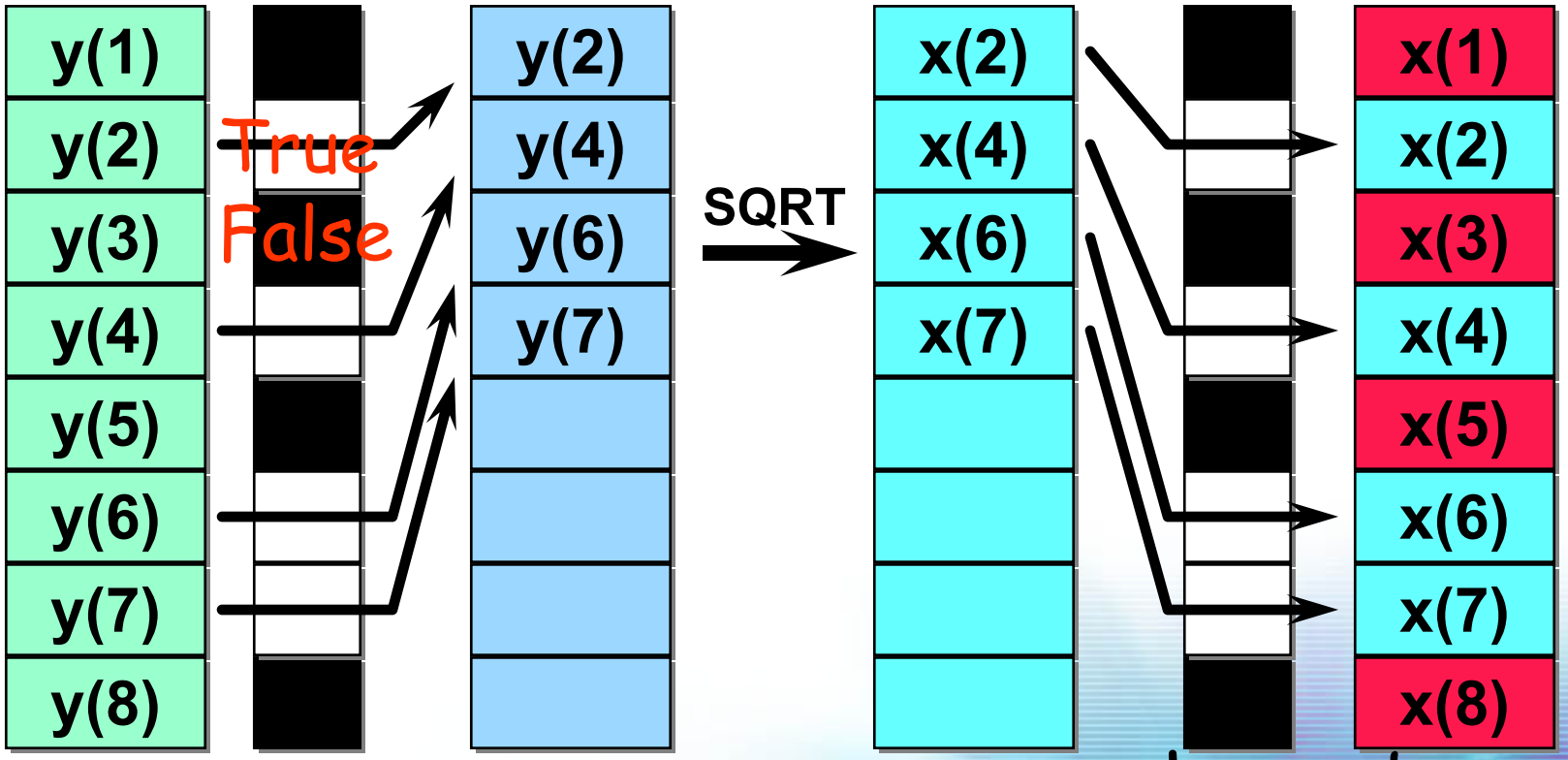
```
do i=1,n
  if( y(i) .ge. 0.0 ) then
    x(i) = sqrt( y(i) )
  end if
end do
```

- 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 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

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

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

```
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
```

Generation of list vector

use !CDIR NODEP

constant if

- 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

```
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:

```
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):

```
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:

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

- 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

```
real accu(256)
do j=1,n
  do i0=1,n,256
    iend=min(n,i0+255)
    do i=i0,iend
      accu(i)=c(i,j)
    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
end do
```

IO optimization

- F_SETBUF Modify I/O-Buffer (n Kb)
- avoid formatted IO
- vectorize IO:
  - slow:

```
real a(n,m)
write(13)((a(i,j),i=2,n-1),j=2,m-1)
```

  - fast:

```
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

!CDIR COLLAPSE:

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

```
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
```

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

Stride for 1d-Arrays

- Example how to generate a stride ‚istride‘

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

- Example: istride = 3

Bank	1	2	3	4	5	6	7	8
	a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)
	a(9)	a(10)	a(11)	a(12)	a(13)	a(14)	a(15)	a(16)
	a(17)	a(18)	a(19)	a(20)	a(21)	a(22)	a(23)	a(24)

ASSUME stride 8 !

Bank busy time measured

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

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

Optimization examples

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

loop interchange

- 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
```

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

- switch indices? depends on leading dimension

loop interchange

- 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:

```
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

```
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

```
 LINE   LOOP    FORTRAN STATEMENT

    1:                 program test
    2:                  real a(100,4),b(100,4),c(100,4)
    3: V------>            do i=1,100
    4: |+----->             do j=1,4
    5: ||                     a(i,j)=a(i,j)*b(i,j)+c(i,j)
    6: |+-----             end do
    7: V------            end do
    8:                 end
~
```

loop expansion

sxf90 -Wf,-L fmtlist map summary transform

```
LINE                    FORTRAN STATEMENT

  1          program test
  2          real a(100,4),b(100,4),c(100,4)
  3          do i=1,100
  .  !CDIR NODEP
  .              do i = 1, 100
  4                  do j=1,4
  5                      a(i,j)=a(i,j)*b(i,j)+c(i,j)
  6                  end do
  .                  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)
  7              end do
  8          end do
```

loop division

- f77:

```
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

loop division

-C vopt delivers:

```
LINE   LOOP    FORTRAN STATEMENT

   1:            program test
   2:             real a(100,100),b(100,100),z(100,100),y(100,100),x(100,100)
   3: X------>         do j=1,n
   4: |+----->         do i=2,n
   5: ||                b(i,j) = sqrt(x(i,j))
   6: ||                a(i,j) = a(i-1,j)*b(i,j)+z(i,j)
   7: ||                y(i,j) = sin(a(i,j))
   8: |+-----          end do
   9: X------         end do
  10:               end
```

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
```

b1 is constant in 1. loop (j)

Unrolling and interchange

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

call to function

- **FORTRAN:**

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

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

- **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
```

Runge-Kutta Timestep

Outermost loop running over particles: not vectorizable

vectorization example (2)

- Vectorized Implementation(1/3)

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

Reduction of drops

vectorization example (3)

- Vectorized Implementation(2/3)

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

Runge-Kutta Timestep

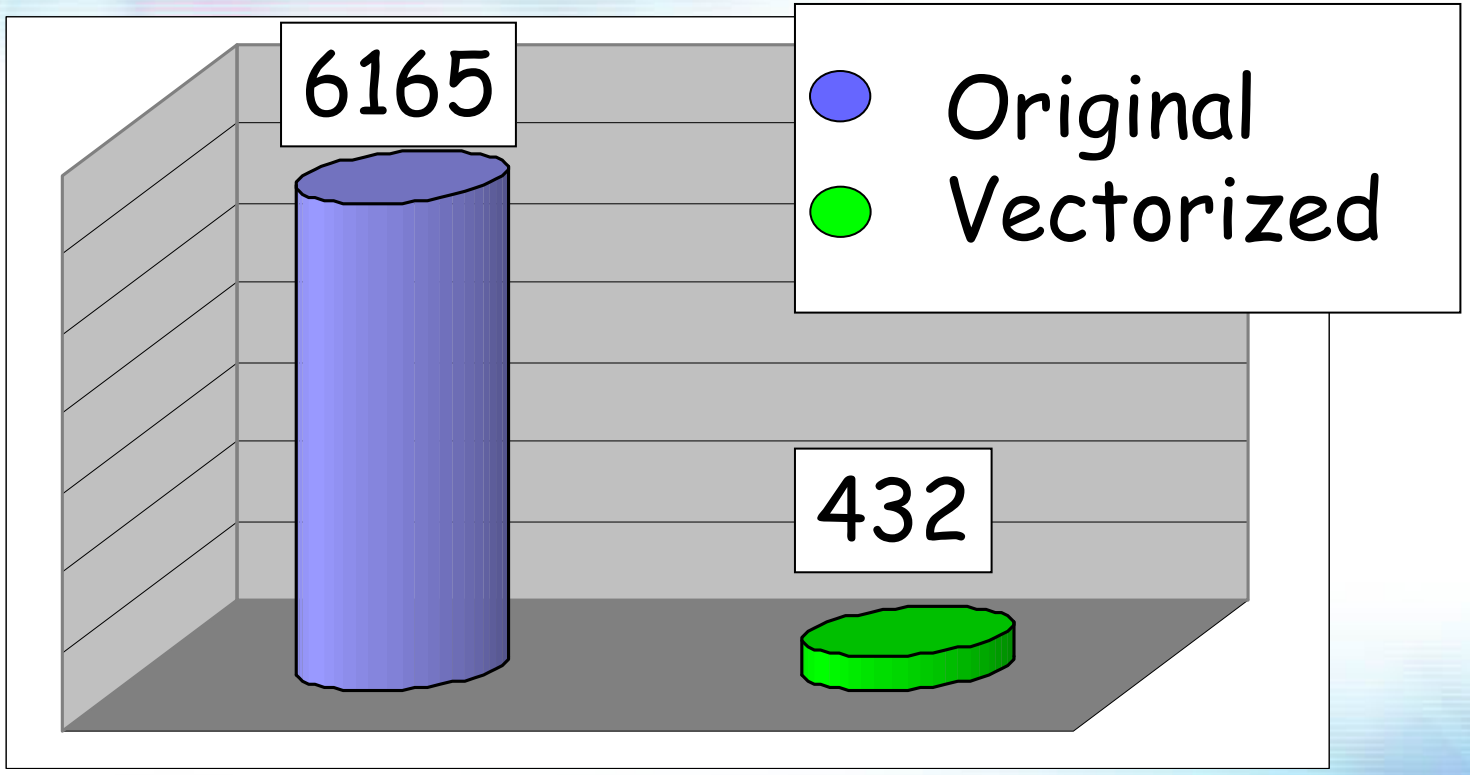
vectorization example (4)

- Vectorized Implementation(3/3)

```
  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



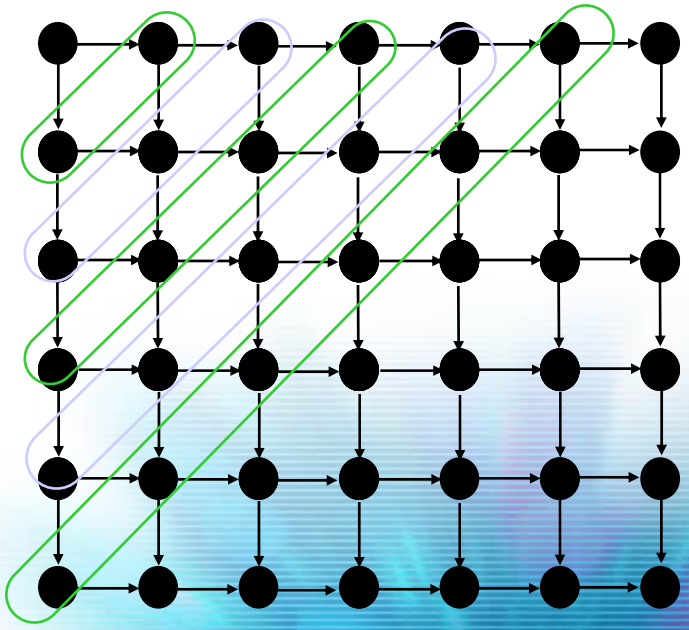
2D recursion

- 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!):

```
  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)

SUPER-UX R14.1 Manual Global Contents

Please refer to

General Usage
User's Guide
User's Reference Manual

Programming
Programmer's Guide
Programmer's Reference Manual

Network
Network Programmer's Guide
STREAMS Programmer's Guide

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

Language
Programming Language Support Reference Manual
FORTRAN90/SX Language Reference Manual
FORTRAN90/SX Programmer's Guide
FORTRAN90/SX Multitasking User's Guide
MPI/SX User's Guide
Assembly Language Reference Manual
DBX User's Guide
PDBX User's Guide
XDBX User's Guide
OpenGL Programmer's Guide
PSUITE User's Guide
C++/SX Programmer's Guide
HPF/SX V2 User's Guide
HPF/SX V2 User's Guide
Math Libraries (MathKeisan) User's Guide

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