Orchestrating a brighter world NEC

NEC Deutschland GmbH
Fritz-Vomfelde-Straße 14-16
D-40547 Düsseldorf/Germany

3rd Aurora Deep Dive Workshop

Introduction to Vectorization

NEC & RWTH Aachen University

Orchestrating a brighter world

NEC brings together and integrates technology and expertise to create the ICT-enabled society of tomorrow.
We collaborate closely with partners and customers around the world, orchestrating each project to ensure all its parts are fine-tuned to local needs.

Every day, our innovative solutions for society contribute to greater safety, security, efficiency and equality, and enable people to live brighter lives.

Introduction to Vectorization

1. What is Vector Computing?
   - Vectorizable Structures
   - Memory access patterns
2. Fundamental Compiler Usage
   - Compilation and Execution
   - Compiler Diagnostics
3. Profiling
   - Ftrace
4. Vectorization Techniques
   - Loop Collapsing
   - Loop Pushing
   - Loop Unrolling

What is Vector Computing?

Vector Idea

‘Scalar` Approach:

```
For all data execute:
   read instruction
   decode instruction
   fetch some data
   perform operation on data
   store result
```

“There is a grid point, particle, equation, element,….
What am I going to do with it?”

‘Vector` Approach:

```
read vector instruction
decode vector instruction
fetch vector data
perform operation on data
   simultaneously
store vector results
```

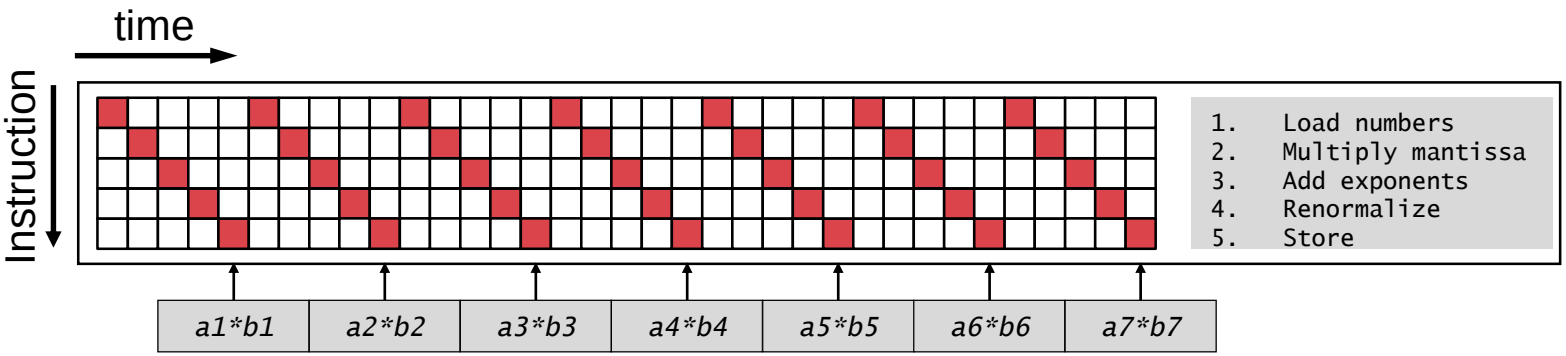
“There is certain operation. To which grid point, particle, equation, element,… am I going to apply it simultaneously?”

Instead of constantly reading/decoding instructions and fetching data, a vector computer reads one instruction and applies it to a set of vector data.

© NEC Corporation 2019

Scalar Processing

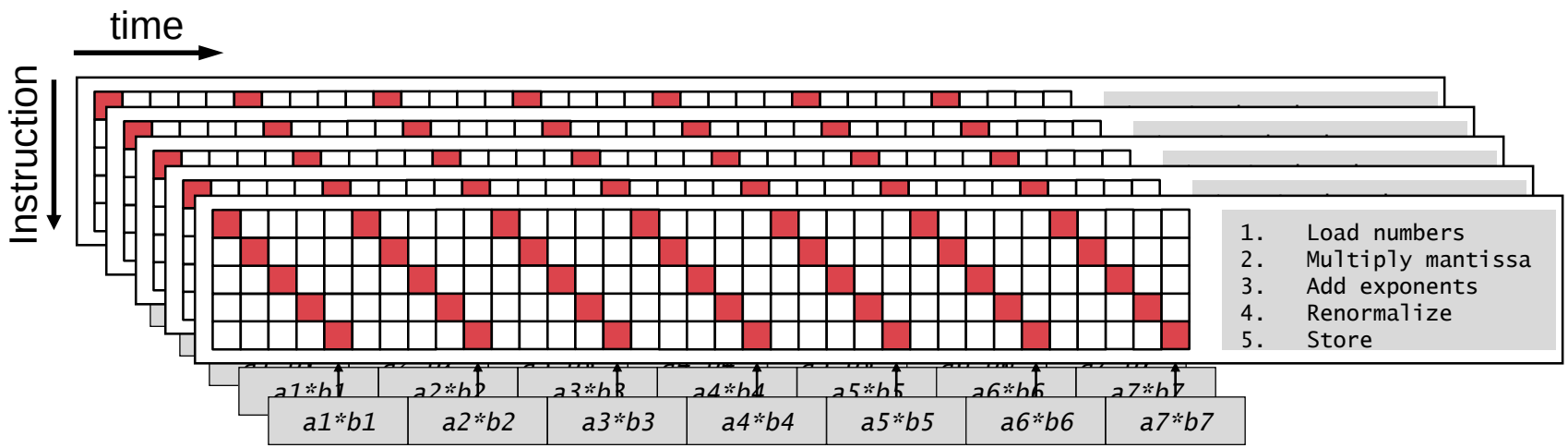
Without Pipelining + Single Pipe:
Only one instruction is executed at a time



© NEC Corporation 2019

SIMD Processing (Modern Scalar)

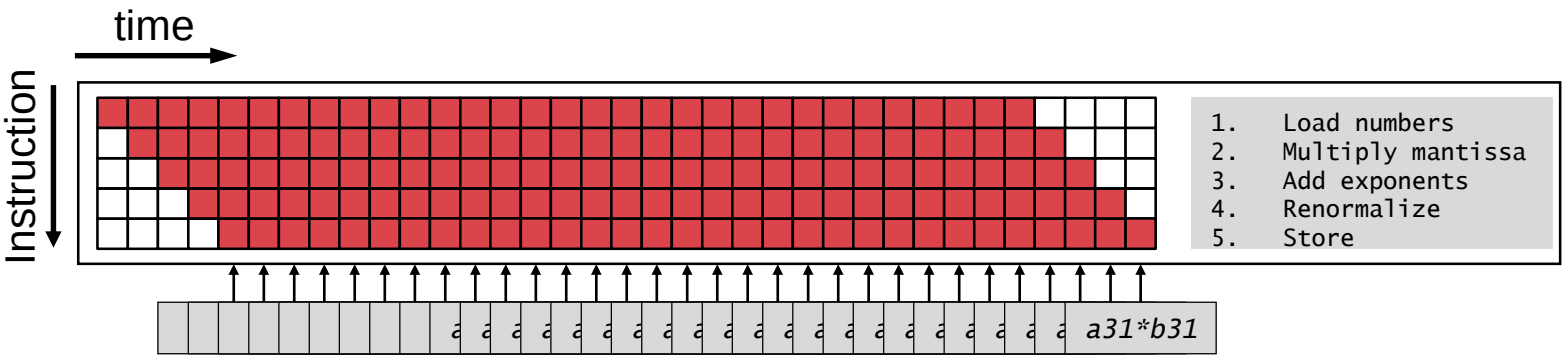
Without Pipelining + Multiple Pipes:
Only one instruction is executed at a time
Parallel execution via multiple pipes



© NEC Corporation 2019

Vector Processing

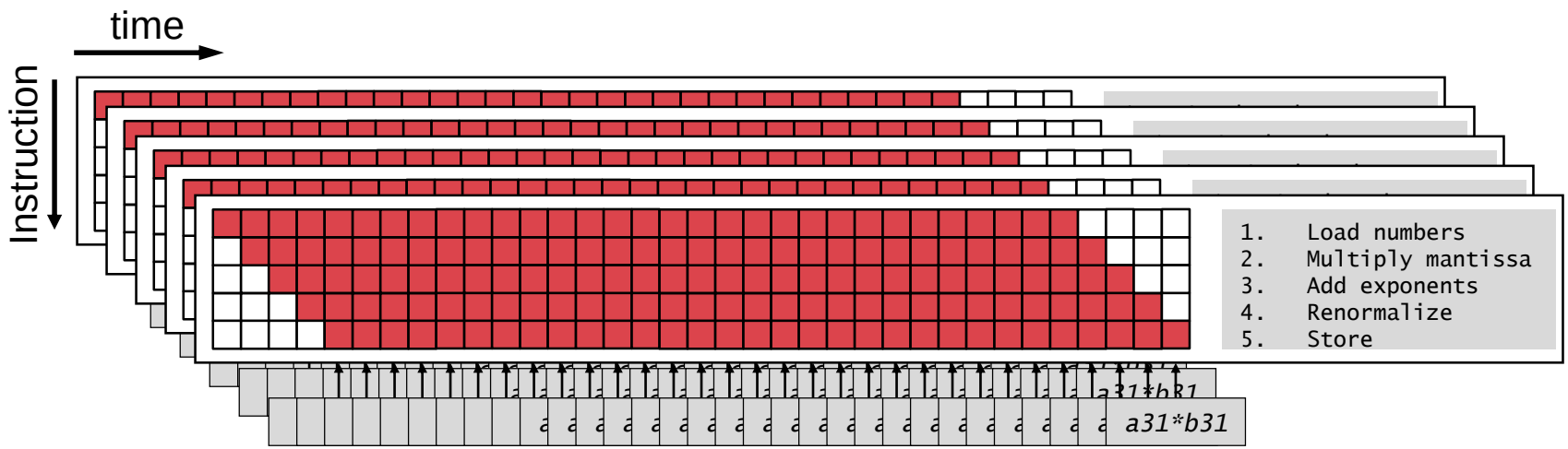
With Pipelining + Single Pipe:
Execute instructions in parallel to hide latency



© NEC Corporation 2019

Parallel Vector Processing (NEC Aurora)

With Pipelining + Multiple Pipes:
Execute instructions in parallel to hide latency.
Parallel execution via multiple pipes



© NEC Corporation 2019

Vectorizable Structures

Vectorizable Structures

- In order to be (automatically) vectorizable a loop structure needs to fulfill certain criteria:
  - Loop count needs to be known upon entering the loop

```
! This might be vectorized
DO i = 1, n
   doing stuff
END DO

! This is not vectorized
! in general
DO WHILE (stuff to do)
   doing stuff
END DO
```


© NEC Corporation 2019

Vectorizable Structures

- In order to be (automatically) vectorizable a loop structure needs to fulfill certain criteria:
  - Loop count needs to be known upon entering the loop.
  - No I/O operations inside the loop.

```
! This does not vectorize
DO i = 1, n
   WRITE(*,*) stuff
END DO
```


© NEC Corporation 2019

Vectorizable Structures

- In order to be (automatically) vectorizable a loop structure needs to fulfill certain criteria:
  - Loop count needs to be known upon entering the loop.
  - No I/O operations inside the loop.
  - Data needs to be parallel. Order of operation must not matter. (Exception for scatter instructions)

```
! This does vectorize
DO i = 1, n
   A(i) = A(i) + B(i)
END DO

! This does not vectorize
DO i = 1, n
   A(i) = A(i-1) + B(i)
END DO
! The compiler is able to
! Build a slower pseudo
! vectorized version of this
! Lookout for “IDIOM detected”
! in the diagnostics list
```


© NEC Corporation 2019

Vectorizable Structures

- In order to be (automatically) vectorizable a loop structure needs to fulfill certain criteria:
  - Loop count needs to be known upon entering the loop.
  - No I/O operations inside the loop.
  - Data needs to be parallel. Order of operation must not matter. (Exception for scatter instructions)
  - No complicated function or routine calls (small functions/routines can be inlined automatically).

```
! This vectorizes as the
! functions can be inlined
DO i = 1, n
   A(i) = inlinable_fkt(B(i))
   A(i) = SQRT(A(i))
END DO

! This does not vectorize
DO i = 1, n
   CALL very_long_routine(A(i))
END DO
```


© NEC Corporation 2019

Vectorizable Structures

- In order to be (automatically) vectorizable a loop structure needs to fulfill certain criteria:
  - Loop count needs to be known upon entering the loop.
  - No I/O operations inside the loop.
  - Data needs to be parallel. Order of operation must not matter. (Exception for scatter instructions)
  - No complicated function or routine calls (small functions/routines can be inlined automatically).
  - No work on non vectorizable data structures (e.g. strings)

```
! This does not vectorize
DO i = 1, n
   A(i) = “Hello ”//”World !”
END DO
```


© NEC Corporation 2019

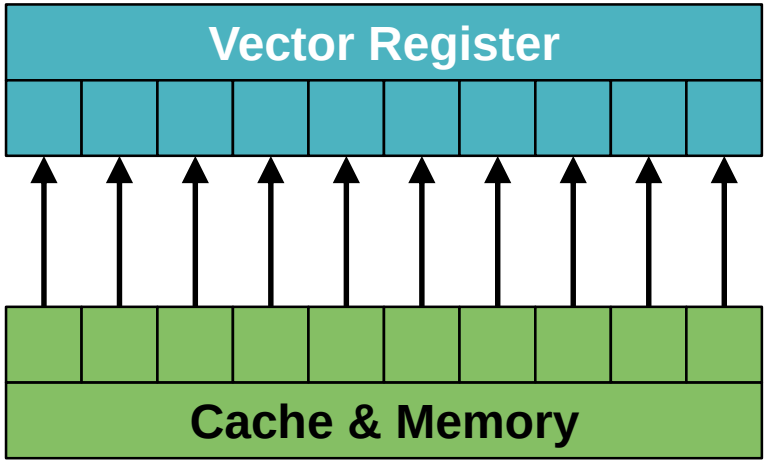
Memory Access Patterns

Memory Access in Vector Computers

- Vector processors have huge data throughput.
- Memory access performance depends on the pattern:
  1. Stride 1

Example:

```
A(i) = B(i)
```



Optimal memory access.

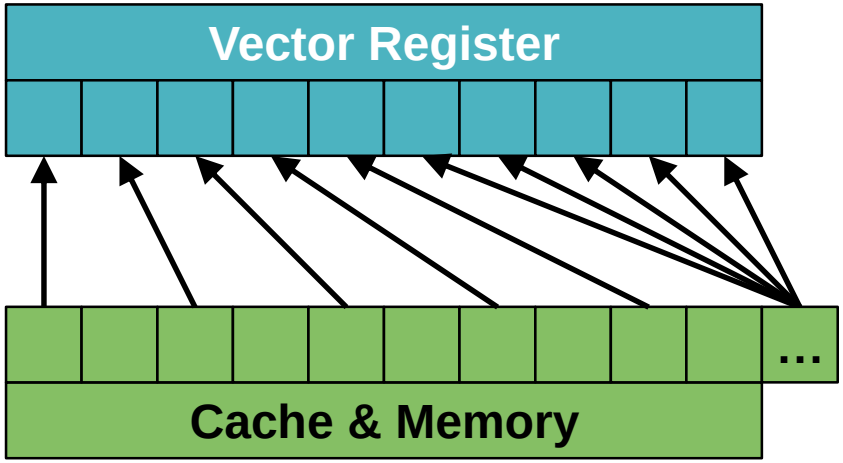
© NEC Corporation 2019

Memory Access in Vector Computers

- Vector processors have huge data throughput.
- Memory access performance depends on the pattern:
  1. Stride 1
  2. Strided

Example:

```
DO i = 1, n, 2
   A(i) = B(i)
```



Not optimal due to only partially used cache lines.

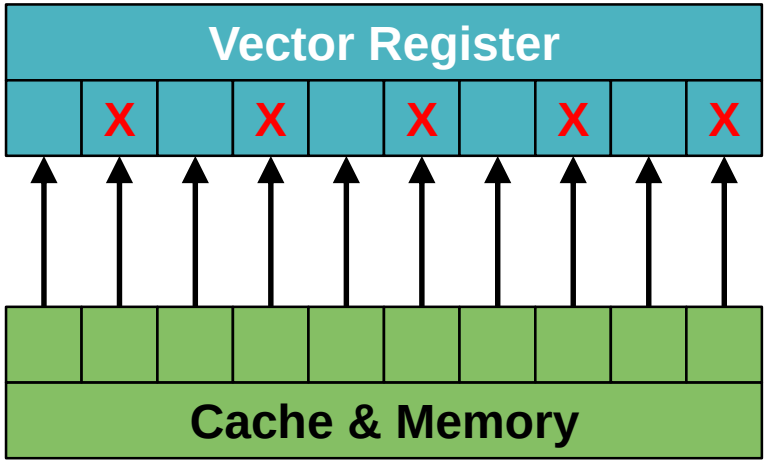
© NEC Corporation 2019

Memory Access in Vector Computers

- Vector processors have huge data throughput.
- Memory access performance depends on the pattern:
  1. Stride 1
  2. Strided
  3. Mask

Example:

```
IF (MOD(i,2) == 0) &
   A(i) = B(i)
```



Not optimal as not every element of a cache line is needed.

Note that all elements are loaded into the vector registers and operated on, but the write back is only performed if the condition applies.

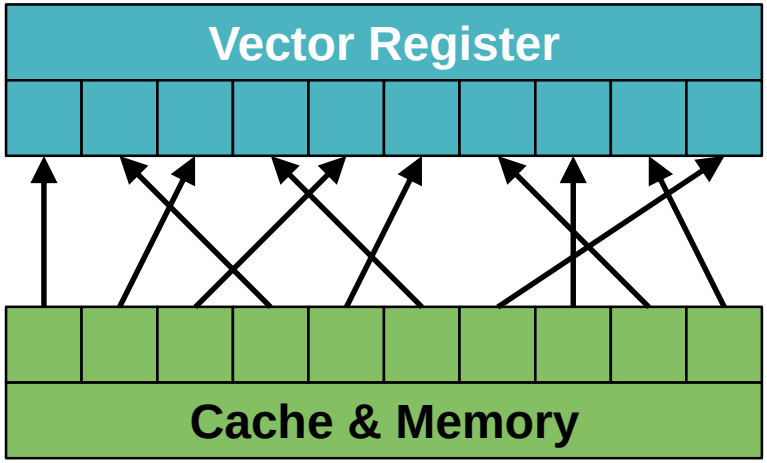
© NEC Corporation 2019

Memory Access in Vector Computers

- Vector processors have huge data throughput.
- Memory access performance depends on the pattern:
  1. Stride 1
  2. Strided
  3. Mask
  4. Gather

Example:

```
A(i) = B(idx(i))
```



Inefficient due to random memory access, potential bank conflicts, partially used cache lines.

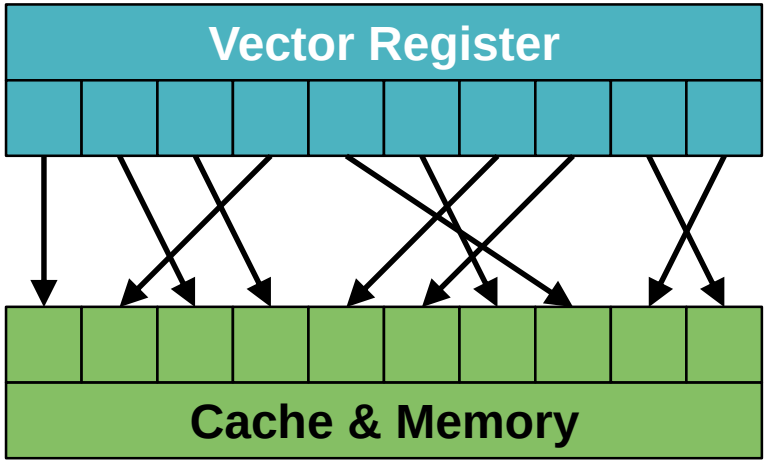
© NEC Corporation 2019

Memory Access in Vector Computers

- Vector processors have huge data throughput.
- Memory access performance depends on the pattern:
  1. Stride 1
  2. Strided
  3. Mask
  4. Gather
  5. Scatter

Example:

```
A(idx(i)) = B(i)
```



Inefficient due to random memory access, potential bank conflicts, partially used cache lines.

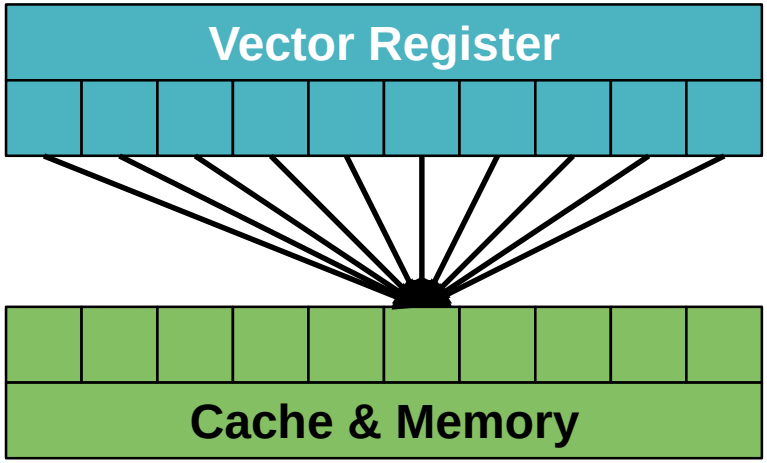
© NEC Corporation 2019

Memory Access in Vector Computers

- Vector processors have huge data throughput.
- Memory access performance depends on the pattern:
  1. Stride 1
  2. Strided
  3. Mask
  4. Gather
  5. Scatter
  6. Reduction

Example:

```
A = A + B(i)
```



Not optimal due to condensation into partial sums up to one value

Note that a reduction is usually executed by accumulating partial sums/products/….

© NEC Corporation 2019

Fundamental Compiler Usage

Compilation and Execution

Aurora Compilers

```
ncc                    C Compiler
nc++                   C++ Compiler
nfort                  Fortran Compiler
```

```
Syntax:
$ <compiler> <flags> <source>

Examples:
$ nfort -o test.x test.F90 #compilation of Fortran
$ nfort -c test.F90                #object creation
$ nfort -o test.x test1.o test2.o #linking

$ ncc -o test.x test.c           #compilation of C
$ ncc -c test.c                  #object creation
$ ncc -o text.x test1.o test2.o #linking
```

Segmentations faults during compilation are often due to a small stack limit. It can be increased with: `ulimit -s unlimited`

© NEC Corporation 2019

Running Aurora Programs

- In order to run Aurora programs it is sufficient to start the executable. It is automatically detected that it is a VE-executable. The OS selects a VE to run on.

```
./test.x
```


© NEC Corporation 2019

Running Aurora Programs

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   const char *ve = getenv("VE_NODE_NUMBER");
   const char *hn = getenv("HOSTNAME");
   printf("Running on %s VE%s\n", hn, ve);
   return 0;
}
```

```
$ ncc -o where_am_i.x where_am_i.c
$ ./where_am_i.x
Running on a1sb8-005 VE4
```


© NEC Corporation 2019

\Orchestrating a brighter world NEC

Compiler Diagnostics

Diagnostics File – Compiler Options List

The *.L file gives information about the compilation and vectorization. Generated with "-report-all –fdiag-vector=3"

```
NEC Fortran Compiler (3.1.0) for Vector Engine   Mon Feb 22 10:21:57 2021
FILE NAME : test.F90

  COMPILER OPTIONS : -fpp -o test.x -report-all -fdiag-vector=3

  PARAMETER :

      Optimization Options :
        …
      Parallelization Options :
        …
      Inlining Options :
        …
      Code Generation Options :
        …
      Debugging Options :
        …
      …
```

Note that generating the diagnostics can slow down the compilation significantly!

© NEC Corporation 2019

Diagnostics File – Diagnostics and Format List

```
NEC Fortran Compiler (3.1.0) for Vector
Engine   Mon Feb 22 10:21:57 2021
FILE NAME: test.F90

PROCEDURE NAME: TEST
DIAGNOSTIC LIST

 LINE              DIAGNOSTIC MESSAGE

    10: vec( 101): Vectorized loop.
    10: err( 504): The number of VLOAD,
VSTORE.:  2,  1.
    10: err( 505): The number of VGT,   VSC.
:  0,  0.
    11: vec( 128): Fused multiply-add
operation applied.
    14: vec( 103): Unvectorized loop.
    14: vec( 180): I/O statement obstructs
vectorization.
    17: opt(1118): This I/O statement
inhibits optimization of loop.
```

```
NEC Fortran Compiler (3.1.0) for Vector
Engine   Mon Feb 22 10:21:57 2021
FILE NAME: test.F90

PROCEDURE NAME: TEST
FORMAT LIST

 LINE   LOOP      STATEMENT

 1:           PROGRAM test
 2:           IMPLICIT NONE
 3:
 4:           REAL(8), DIMENSION(2048) :: &
 5:              A, B
 6:           INTEGER :: i
 7:           CALL RANDOM_NUMBER(A)
 8:           CALL RANDOM_NUMBER(B)
 9:
10: V------>  DO i = 2, 2048
11: |       F    B(i)=B(i) + 2*A(i-1)
12: V------   END DO
13:
14: +------>  DO i = 2, 2048
15: |            B(i) = SQRT(B(i))
16: |            IF (MOD(i,256) == 0) &
17: |               WRITE(*,*) i
18: +------   END DO
19:
20:           END PROGRAM test
```


© NEC Corporation 2019

Diagnostics File – Diagnostics and Format List

```
10: vec( 101): Vectorized loop.
10: err( 504): The number of VLOAD, VSTORE.:2,1.
10: err( 505): The number of VGT,   VSC.   :0,0.
11: vec( 128): Fused multiply-add operation
applied.
...

 1:            PROGRAM test
 2:            IMPLICIT NONE
 3:
 4:            REAL(8), DIMENSION(2048) :: &
 5:               A, B
 6:            INTEGER :: i
 7:            CALL RANDOM_NUMBER(A)
 8:            CALL RANDOM_NUMBER(B)
 9:
10: V------>   DO i = 2, 2048
11: |       F     B(i) = B(i) + 2*A(i-1)
12: V------    END DO
13:
14: +------>   DO i = 2, 2048
15: |             B(i) = SQRT(B(i))
16: |             IF (MOD(i,256) == 0) &
17: |                WRITE(*,*) i
18: +------    END DO
19:
20:            END PROGRAM test
```

- The loop in line 10 is vectorized (“V”).

© NEC Corporation 2019

Diagnostics File – Diagnostics and Format List

```
10: vec( 101): Vectorized loop.
10: err( 504): The number of VLOAD, VSTORE.:2,1.
10: err( 505): The number of VGT,   VSC.   :0,0.
11: vec( 128): Fused multiply-add operation
applied.
...

 1:            PROGRAM test
 2:            IMPLICIT NONE
 3:
 4:            REAL(8), DIMENSION(2048) :: &
 5:               A, B
 6:            INTEGER :: i
 7:            CALL RANDOM_NUMBER(A)
 8:            CALL RANDOM_NUMBER(B)
 9:
10: V------>   DO i = 2, 2048
11: |       F     B(i) = B(i) + 2*A(i-1)
12: V------    END DO
13:
14: +------>   DO i = 2, 2048
15: |             B(i) = SQRT(B(i))
16: |             IF (MOD(i,256) == 0) &
17: |                WRITE(*,*) i
18: +------    END DO
19:
20:            END PROGRAM test
```

- The loop in line 10 is vectorized (“V”).
- Two vector loads (VLOAD) for `B(i)` and `A(i-1)` are used.
- One vector store (VSTORE) for `B(i)` is used.

© NEC Corporation 2019

Diagnostics File – Diagnostics and Format List

```
10: vec( 101): Vectorized loop.
10: err( 504): The number of VLOAD, VSTORE.:2,1.
10: err( 505): The number of VGT,   VSC.   :0,0.
11: vec( 128): Fused multiply-add operation
applied.
...

 1:            PROGRAM test
 2:            IMPLICIT NONE
 3:
 4:            REAL(8), DIMENSION(2048) :: &
 5:               A, B
 6:            INTEGER :: i
 7:            CALL RANDOM_NUMBER(A)
 8:            CALL RANDOM_NUMBER(B)
 9:
10: V------>   DO i = 2, 2048
11: |       F     B(i) = B(i) + 2*A(i-1)
12: V------    END DO
13:
14: +------>   DO i = 2, 2048
15: |             B(i) = SQRT(B(i))
16: |             IF (MOD(i,256) == 0) &
17: |                WRITE(*,*) i
18: +------    END DO
19:
20:            END PROGRAM test
```

- The loop in line 10 is vectorized (“V”).
- Two vector loads (VLOAD) for `B(i)` and `A(i-1)` are used.
- One vector store (VSTORE) for `B(i)` is used.
- No vector gather (VGT) or scatter (VSC) instructions are required.

© NEC Corporation 2019

Diagnostics File – Diagnostics and Format List

```
10: vec( 101): Vectorized loop.
10: err( 504): The number of VLOAD, VSTORE.:2,1.
10: err( 505): The number of VGT,   VSC.   :0,0.
11: vec( 128): Fused multiply-add operation
applied.
...

 1:            PROGRAM test
 2:            IMPLICIT NONE
 3:
 4:            REAL(8), DIMENSION(2048) :: &
 5:               A, B
 6:            INTEGER :: i
 7:            CALL RANDOM_NUMBER(A)
 8:            CALL RANDOM_NUMBER(B)
 9:
10: V------>   DO i = 2, 2048
11: |       F     B(i) = B(i) + 2*A(i-1)
12: V------    END DO
13:
14: +------>   DO i = 2, 2048
15: |             B(i) = SQRT(B(i))
16: |             IF (MOD(i,256) == 0) &
17: |                WRITE(*,*) i
18: +------    END DO
19:
20:            END PROGRAM test
```

- The loop in line 10 is vectorized (“V”).
- Two vector loads (VLOAD) for `B(i)` and `A(i-1)` are used.
- One vector store (VSTORE) for `B(i)` is used.
- No vector gather (VGT) or scatter (VSC) instructions are required.
- Special instruction Fused multiply-add (“F”) is used.

© NEC Corporation 2019

Diagnostics File – Diagnostics and Format List

```
14: vec( 103): Unvectorized loop.
14: vec( 180): I/O statement obstructs
vectorization.
17: opt(1118): This I/O statement inhibits
optimization of loop.
...

 1:            PROGRAM test
 2:            IMPLICIT NONE
 3:
 4:            REAL(8), DIMENSION(2048) :: &
 5:               A, B
 6:            INTEGER :: i
 7:            CALL RANDOM_NUMBER(A)
 8:            CALL RANDOM_NUMBER(B)
 9:
10: V------>   DO i = 2, 2048
11: |       F     B(i) = B(i) + 2*A(i-1)
12: V------    END DO
13:
14: +------>   DO i = 2, 2048
15: |             B(i) = SQRT(B(i))
16: |             IF (MOD(i,256) == 0) &
17: |                WRITE(*,*) i
18: +------    END DO
19:
20:            END PROGRAM test
```

- The loop in line 10 is vectorized (“V”).
- Two vector loads (VLOAD) for `B(i)` and `A(i-1)` are used.
- One vector store (VSTORE) for `B(i)` is used.
- No vector gather (VGT) or scatter (VSC) instructions are required.
- Special instruction Fused multiply-add (“F”) is used.
- The loop in line 14 is not vectorized(“+”).

© NEC Corporation 2019

Diagnostics File – Diagnostics and Format List

```
14: vec( 103): Unvectorized loop.
14: vec( 180): I/O statement obstructs
vectorization.
17: opt(1118): This I/O statement inhibits
optimization of loop.
...

 1:            PROGRAM test
 2:            IMPLICIT NONE
 3:
 4:            REAL(8), DIMENSION(2048) :: &
 5:               A, B
 6:            INTEGER :: i
 7:            CALL RANDOM_NUMBER(A)
 8:            CALL RANDOM_NUMBER(B)
 9:
10: V------>   DO i = 2, 2048
11: |       F     B(i) = B(i) + 2*A(i-1)
12: V------    END DO
13:
14: +------>   DO i = 2, 2048
15: |             B(i) = SQRT(B(i))
16: |             IF (MOD(i,256) == 0) &
17: |                WRITE(*,*) i
18: +------    END DO
19:
20:            END PROGRAM test
```

- The loop in line 10 is vectorized (“V”).
- Two vector loads (VLOAD) for `B(i)` and `A(i-1)` are used.
- One vector store (VSTORE) for `B(i)` is used.
- No vector gather (VGT) or scatter (VSC) instructions are required.
- Special instruction Fused multiply-add (“F”) is used.
- The loop in line 14 is not vectorized(“+”).
- An IO Statement (`WRITE`) in line 17 prohibits vectorization.

© NEC Corporation 2019

Diagnostics File – Vectorization List

```
NEC Fortran Compiler (3.1.0) for Vector Engine   Mon Feb 22 10:21:57 2021
FILE NAME: test.F90

PROCEDURE NAME: TEST
VECTORIZATION LIST

  NOTE: The number of operation does not include the instruction
        for register spill and restore.

  LOOP BEGIN: (test.F90:10)
    <Vectorized loop.>
    *** The number of VGT,   VSC.    :  0,  0. (test.F90:10)
    *** The number of VLOAD, VSTORE. :  2,  1. (test.F90:10)
  LOOP END

  LOOP BEGIN: (test.F90:14)
    <Unvectorized loop.>
    *** I/O statement obstructs vectorization. (test.F90:14)
  LOOP END
```


© NEC Corporation 2019

Profiling

Ftrace

FTRACE – Performance Analysis

```
$ nfort -ftrace -o test.x test.f90
$ ./test.x
$ ftrace -f ftrace.out

FREQUENCY  EXCLUSIVE       AVER.TIME       MFLOPS     PROC.NAME
           TIME[sec](  % )    [msec]

     1000     2.420(100.0)     2.420 ... 143773.7 ... DOIT
        1     0.000(  0.0)     0.419 ...      0.0 ... TEST
----------------------------------- ... -------- ... -------
     1001     2.421(100.0)     2.418 ... 143748.8 ... total

     1000     0.581( 24.0)     0.581 ... 130897.0 ... POW
     1000     0.463( 19.1)     0.463 ... 157510.0 ... EXP-LOG
     1000     0.274( 11.3)     0.274 ... 153136.7 ... LOG
     1000     0.263( 10.8)     0.263 ... 140903.9 ... TAN
     1000     0.158(  6.5)     0.158 ... 170371.7 ... COS
     1000     0.156(  6.4)     0.156 ... 166765.6 ... SIN
     1000     0.153(  6.3)     0.153 ... 202677.6 ... EXP
     1000     0.098(  4.0)     0.098 ... 183719.7 ... DIV
     1000     0.079(  3.3)     0.079 ... 177495.9 ... SQRT
     1000     0.064(  2.6)     0.064 ...  15715.3 ... ADD
     1000     0.064(  2.6)     0.064 ...  15727.1 ... MUL
     1000     0.063(  2.6)     0.063 ...  31550.5 ... FMA
----------------------------------- ... -------- ... -------
```

- FTRACE analyses the runtime of different region and routines in your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Call frequency

Note: ftrace produces call overhead. Compile routines with huge call count without "-ftrace"!

© NEC Corporation 2019

FTRACE – Performance Analysis

```
$ nfort -ftrace -o test.x test.f90
$ ./test.x
$ ftrace -f ftrace.out

FREQUENCY  EXCLUSIVE       AVER.TIME       MFLOPS     PROC.NAME
           TIME[sec](  % )    [msec]

     1000     2.420(100.0)     2.420 ... 143773.7 ... DOIT
        1     0.000(  0.0)     0.419 ...      0.0 ... TEST
----------------------------------- ... -------- ... -------
     1001     2.421(100.0)     2.418 ... 143748.8 ... total

     1000     0.581( 24.0)     0.581 ... 130897.0 ... POW
     1000     0.463( 19.1)     0.463 ... 157510.0 ... EXP-LOG
     1000     0.274( 11.3)     0.274 ... 153136.7 ... LOG
     1000     0.263( 10.8)     0.263 ... 140903.9 ... TAN
     1000     0.158(  6.5)     0.158 ... 170371.7 ... COS
     1000     0.156(  6.4)     0.156 ... 166765.6 ... SIN
     1000     0.153(  6.3)     0.153 ... 202677.6 ... EXP
     1000     0.098(  4.0)     0.098 ... 183719.7 ... DIV
     1000     0.079(  3.3)     0.079 ... 177495.9 ... SQRT
     1000     0.064(  2.6)     0.064 ...  15715.3 ... ADD
     1000     0.064(  2.6)     0.064 ...  15727.1 ... MUL
     1000     0.063(  2.6)     0.063 ...  31550.5 ... FMA
----------------------------------- ... -------- ... -------
```

- FTRACE analyses the runtime of different region and routines in your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Call frequency
  - Exclusive runtime

Note: ftrace produces call overhead. Compile routines with huge call count without "-ftrace"!

© NEC Corporation 2019

FTRACE – Performance Analysis

```
$ nfort -ftrace -o test.x test.f90
$ ./test.x
$ ftrace -f ftrace.out

FREQUENCY  EXCLUSIVE       AVER.TIME      MFLOPS     PROC.NAME
           TIME[sec](  % )   [msec]

     1000    2.420(100.0)     2.420 ... 143773.7 ... DOIT
        1    0.000(  0.0)     0.419 ...      0.0 ... TEST
---------  ---------------  --------- ... -------- ... -------
     1001    2.421(100.0)     2.418 ... 143748.8 ... total

     1000    0.581( 24.0)     0.581 ... 130897.0 ... POW
     1000    0.463( 19.1)     0.463 ... 157510.0 ... EXP-LOG
     1000    0.274( 11.3)     0.274 ... 153136.7 ... LOG
     1000    0.263( 10.8)     0.263 ... 140903.9 ... TAN
     1000    0.158(  6.5)     0.158 ... 170371.7 ... COS
     1000    0.156(  6.4)     0.156 ... 166765.6 ... SIN
     1000    0.153(  6.3)     0.153 ... 202677.6 ... EXP
     1000    0.098(  4.0)     0.098 ... 183719.7 ... DIV
     1000    0.079(  3.3)     0.079 ... 177495.9 ... SQRT
     1000    0.064(  2.6)     0.064 ...  15715.3 ... ADD
     1000    0.064(  2.6)     0.064 ...  15727.1 ... MUL
     1000    0.063(  2.6)     0.063 ...  31550.5 ... FMA
---------  ---------------  --------- ... -------- ... -------
```

- FTRACE analyses the runtime of different region and routines in your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Call frequency
  - Exclusive runtime
  - Average runtime

Note: ftrace produces call overhead. Compile routines with huge call count without "-ftrace"!

© NEC Corporation 2019

FTRACE – Performance Analysis

```
$ nfort -ftrace -o test.x test.f90
$ ./test.x
$ ftrace -f ftrace.out

FREQUENCY  EXCLUSIVE        AVER.TIME      MFLOPS     PROC.NAME
           TIME[sec](  % )    [msec]

     1000    2.420(100.0)     2.420 ... 143773.7 ... DOIT
        1    0.000(  0.0)     0.419 ...      0.0 ... TEST
---------  ---------------  --------- ... -------- ... -------
     1001    2.421(100.0)     2.418 ... 143748.8 ... total

     1000    0.581( 24.0)     0.581 ... 130897.0 ... POW
     1000    0.463( 19.1)     0.463 ... 157510.0 ... EXP-LOG
     1000    0.274( 11.3)     0.274 ... 153136.7 ... LOG
     1000    0.263( 10.8)     0.263 ... 140903.9 ... TAN
     1000    0.158(  6.5)     0.158 ... 170371.7 ... COS
     1000    0.156(  6.4)     0.156 ... 166765.6 ... SIN
     1000    0.153(  6.3)     0.153 ... 202677.6 ... EXP
     1000    0.098(  4.0)     0.098 ... 183719.7 ... DIV
     1000    0.079(  3.3)     0.079 ... 177495.9 ... SQRT
     1000    0.064(  2.6)     0.064 ...  15715.3 ... ADD
     1000    0.064(  2.6)     0.064 ...  15727.1 ... MUL
     1000    0.063(  2.6)     0.063 ...  31550.5 ... FMA
---------  ---------------  --------- ... -------- ... -------
```

- FTRACE analyses the runtime of different region and routines in your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Call frequency
  - Exclusive runtime
  - Average runtime
  - Megaflops

Note: ftrace produces call overhead. Compile routines with huge call count without "-ftrace"!

© NEC Corporation 2019

FTRACE – Performance Analysis

```
$ nfort -ftrace -o test.x test.f90
$ ./test.x
$ ftrace -f ftrace.out

FREQUENCY  EXCLUSIVE        AVER.TIME      MFLOPS     PROC.NAME
           TIME[sec](  % )    [msec]

     1000    2.420(100.0)     2.420 ... 143773.7 ... DOIT
        1    0.000(  0.0)     0.419 ...      0.0 ... TEST
---------  ---------------  --------- ... -------- ... -------
     1001    2.421(100.0)     2.418 ... 143748.8 ... total

     1000    0.581( 24.0)     0.581 ... 130897.0 ... POW
     1000    0.463( 19.1)     0.463 ... 157510.0 ... EXP-LOG
     1000    0.274( 11.3)     0.274 ... 153136.7 ... LOG
     1000    0.263( 10.8)     0.263 ... 140903.9 ... TAN
     1000    0.158(  6.5)     0.158 ... 170371.7 ... COS
     1000    0.156(  6.4)     0.156 ... 166765.6 ... SIN
     1000    0.153(  6.3)     0.153 ... 202677.6 ... EXP
     1000    0.098(  4.0)     0.098 ... 183719.7 ... DIV
     1000    0.079(  3.3)     0.079 ... 177495.9 ... SQRT
     1000    0.064(  2.6)     0.064 ...  15715.3 ... ADD
     1000    0.064(  2.6)     0.064 ...  15727.1 ... MUL
     1000    0.063(  2.6)     0.063 ...  31550.5 ... FMA
---------  ---------------  --------- ... -------- ... -------
```

Note: ftrace produces call overhead. Compile routines with huge call count without "`-ftrace`"!

- FTRACE analyses the runtime of different region and routines in your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Call frequency
  - Exclusive runtime
  - Average runtime
  - Megaflops
  - Section/Routine name
    (as given as call argument)
- Rows sorted by Exclusive Runtime (sum of user times over all processes)

© NEC Corporation 2019

FTRACE – Vector Performance

```
$ ./test.x
$ ftrace -f ftrace.out

EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

  7.234(100.0)   ... 83.72  93.6     3.141 ... TEST
---------------  ... --------------------- ... ---------
  7.234(100.0)   ... 83.72  93.6     3.141 ... total

  2.588( 35.8)   ... 91.90 250.0     0.640 ... BAD_VTIME
  2.428( 33.6)   ... 54.76   4.6     2.429 ... BAD_VLEN
  2.217( 30.6)   ... 60.88 256.0     0.072 ... BAD_VOPR
---------------  ... --------------------- ... ---------

$./test2.x
$ftrace -f ftrace.out

EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

   1.418(100.0)  ... 99.20 250.8     1.415 ... TEST
---------------  ... --------------------- ... ---------
   1.417(100.0)  ... 99.20 250.8     1.415 ... total

   0.800( 56.4)  ... 99.23 250.0     0.799 ... BAD_VTIME
   0.574( 40.5)  ... 99.05 255.0     0.571 ... BAD_VOPR
   0.044(  3.1)  ... 99.11 250.0     0.044 ... BAD_VLEN
---------------  ... --------------------- ... ---------
```

- FTRACE analyses the vector performance of your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Exclusive runtime

FTRACE instrumentation causes execution time overhead – recompile without "**-ftrace**" after optimization !

© NEC Corporation 2019

FTRACE – Vector Performance

```
$ ./test.x
$ ftrace -f ftrace.out

EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

  7.234(100.0)   ... 83.72  93.6     3.141 ... TEST
---------------  ... --------------------- ... ---------
  7.234(100.0)   ... 83.72  93.6     3.141 ... total

  2.588( 35.8)   ... 91.90 250.0     0.640 ... BAD_VTIME
  2.428( 33.6)   ... 54.76   4.6     2.429 ... BAD_VLEN
  2.217( 30.6)   ... 60.88 256.0     0.072 ... BAD_VOPR
---------------  ... --------------------- ... ---------

$./test2.x
$ftrace -f ftrace.out

EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

   1.418(100.0)  ... 99.20 250.8     1.415 ... TEST
---------------  ... --------------------- ... ---------
   1.417(100.0)  ... 99.20 250.8     1.415 ... total

   0.800( 56.4)  ... 99.23 250.0     0.799 ... BAD_VTIME
   0.574( 40.5)  ... 99.05 255.0     0.571 ... BAD_VOPR
   0.044(  3.1)  ... 99.11 250.0     0.044 ... BAD_VLEN
---------------  ... --------------------- ... ---------
```

- FTRACE analyses the vector performance of your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Exclusive runtime
  - Vector operation ratio
    Aim for close to 100%.
    (90% is still very bad)

FTRACE instrumentation causes execution time overhead – recompile without "**-ftrace**" after optimization !

© NEC Corporation 2019

FTRACE – Vector Performance

```
$ ./test.x
$ ftrace -f ftrace.out

EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

  7.234(100.0)   ... 83.72  93.6     3.141 ... TEST
---------------  ... ----- ------ --------- ... ---------
  7.234(100.0)   ... 83.72  93.6     3.141 ... total

  2.588( 35.8)   ... 91.90 250.0     0.640 ... BAD_VTIME
  2.428( 33.6)   ... 54.76   4.6     2.429 ... BAD_VLEN
  2.217( 30.6)   ... 60.88 256.0     0.072 ... BAD_VOPR
---------------  ... ----- ------ --------- ... ---------

$./test2.x
$ftrace -f ftrace.out

EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

   1.418(100.0)  ... 99.20 250.8     1.415 ... TEST
---------------  ... ----- ------ --------- ... ---------
   1.417(100.0)  ... 99.20 250.8     1.415 ... total

   0.800( 56.4)  ... 99.23 250.0     0.799 ... BAD_VTIME
   0.574( 40.5)  ... 99.05 255.0     0.571 ... BAD_VOPR
   0.044(  3.1)  ... 99.11 250.0     0.044 ... BAD_VLEN
---------------  ... ----- ------ --------- ... ---------
```

- FTRACE analyses the vector performance of your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Exclusive runtime
  - Vector operation ratio
    Aim for close to 100%.
    (90% is still very bad)
  - Average vector length
    Aim for close to 256.

FTRACE instrumentation causes execution time overhead – recompile without "**-ftrace**" after optimization !

© NEC Corporation 2019

FTRACE – Vector Performance

```
$ ./test.x
$ ftrace -f ftrace.out

EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

  7.234(100.0)   ... 83.72  93.6     3.141 ... TEST
---------------  ... ----------------------- ... ---------
  7.234(100.0)   ... 83.72  93.6     3.141 ... total

  2.588( 35.8)   ... 91.90 250.0     0.640 ... BAD_VTIME
  2.428( 33.6)   ... 54.76   4.6     2.429 ... BAD_VLEN
  2.217( 30.6)   ... 60.88 256.0     0.072 ... BAD_VOPR
---------------  ... ----------------------- ... ---------

$./test2.x
$ftrace -f ftrace.out

EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

   1.418(100.0)  ... 99.20 250.8     1.415 ... TEST
---------------  ... ----------------------- ... ---------
   1.417(100.0)  ... 99.20 250.8     1.415 ... total

   0.800( 56.4)  ... 99.23 250.0     0.799 ... BAD_VTIME
   0.574( 40.5)  ... 99.05 255.0     0.571 ... BAD_VOPR
   0.044(  3.1)  ... 99.11 250.0     0.044 ... BAD_VLEN
---------------  ... ----------------------- ... ---------
```

- FTRACE analyses the vector performance of your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Exclusive runtime
  - Vector operation ratio
    Aim for close to 100%.
    (90% is still very bad)
  - Average vector length
    Aim for close to 256.
  - Vector time
    Aim for close to exclusive time

FTRACE instrumentation causes execution time overhead – recompile without "**-ftrace**" after optimization !

© NEC Corporation 2019

FTRACE – Vector Performance

```
$ ./test.x
$ ftrace -f ftrace.out

EXCLUSIVE        ...  V.OP   AVER.    VECTOR  ...  PROC.NAME
TIME[sec](  % )  ...  RATIO  V.LEN     TIME   ...

  7.234(100.0)   ...  83.72   93.6    3.141   ...  TEST
---------------  ...  -----  ------  -------  ...  ---------
  7.234(100.0)   ...  83.72   93.6    3.141   ...  total

  2.588( 35.8)   ...  91.90  250.0    0.640   ...  BAD_VTIME
  2.428( 33.6)   ...  54.76    4.6    2.429   ...  BAD_VLEN
  2.217( 30.6)   ...  60.88  256.0    0.072   ...  BAD_VOPR
---------------  ...  -----  ------  -------  ...  ---------

$./test2.x
$ftrace -f ftrace.out

EXCLUSIVE        ...  V.OP   AVER.    VECTOR  ...  PROC.NAME
TIME[sec](  % )  ...  RATIO  V.LEN     TIME   ...

   1.418(100.0)  ...  99.20  250.8    1.415   ...  TEST
---------------  ...  -----  ------  -------  ...  ---------
   1.417(100.0)  ...  99.20  250.8    1.415   ...  total

   0.800( 56.4)  ...  99.23  250.0    0.799   ...  BAD_VTIME
   0.574( 40.5)  ...  99.05  255.0    0.571   ...  BAD_VOPR
   0.044(  3.1)  ...  99.11  250.0    0.044   ...  BAD_VLEN
---------------  ...  -----  ------  -------  ...  ---------
```

- FTRACE analyses the vector performance of your program.
- Create own region:
  - call ftrace_region_begin("name")
    call ftrace_region_end("name")
- For now important:
  - Exclusive runtime
  - Vector operation ratio
    Aim for close to 100%.
    (90% is still very bad)
  - Average vector length
    Aim for close to 256.
  - Vector time
    Aim for close to exclusive time
  - Section/Routine name
    (as given as call argument)

FTRACE instrumentation causes execution time overhead – recompile without "**-ftrace**" after optimization !

© NEC Corporation 2019

PROGINF – Program Performance Information

```
********  Program  Information  ********
 Real Time (sec)                      :      2618.485888
 User Time (sec)                      :      2610.633052
 Vector Time (sec)                    :      1780.148224
 Inst. Count                          :    2724948429716
 V. Inst. Count                       :     560487363761
 V. Element Count                     :  123427068317590
 V. Load Element Count                :   17545610562224
 FLOP Count                           :  111281389268730
 MOPS                                 :     65113.186196
 MOPS (Real)                          :     64540.332614
 MFLOPS                               :     42876.181900
 MFLOPS (Real)                        :     42498.965305
 A. V. Length                         :       220.213829
 V. Op. Ratio (%)                     :        98.719220
 L1 Cache Miss (sec)                  :       192.719754
 CPU Port Conf. (sec)                 :         0.798491
 V. Arith. Exec. (sec)                :      1006.263169
 V. Load Exec. (sec)                  :       583.891587
 VLD LLC Hit Element Ratio (%)        :        78.225915
 Power Throttling (sec)               :         0.000000
 Thermal Throttling (sec)             :         0.000000
 Memory Size Used (MB)                :     10930.000000

 Start Time (date)        :  Thu Oct 24 15:46:19 2019 CEST
 End   Time (date)        :  Thu Oct 24 16:29:57 2019 CEST
```

- PROGINF analyses the vector performance of your program.
- Set **`VE_PROGINF=DETAIL`** in run-time environment.
- Important:
  - Real time
  - Vector operation ratio
    Aim for close to 100%.
    (90% is still very bad)
  - Average vector length
    Aim for close to 256.
  - Vector time
    Aim for close to real time

Note that the vector operation ratio, the average vector length, and the vector time can be independently bad! First check the ratio of vector to exclusive time, next check the average vector length for optimal values.

© NEC Corporation 2019

Vectorization Techniques

Loop Collapsing

Collapsing – Increasing the Vector Length

Consider the following nested loop:

```
DO j = 1, m
   DO i = 1, n
      A(i,j) = 2.0*A(i,j)
   END DO
END DO
```

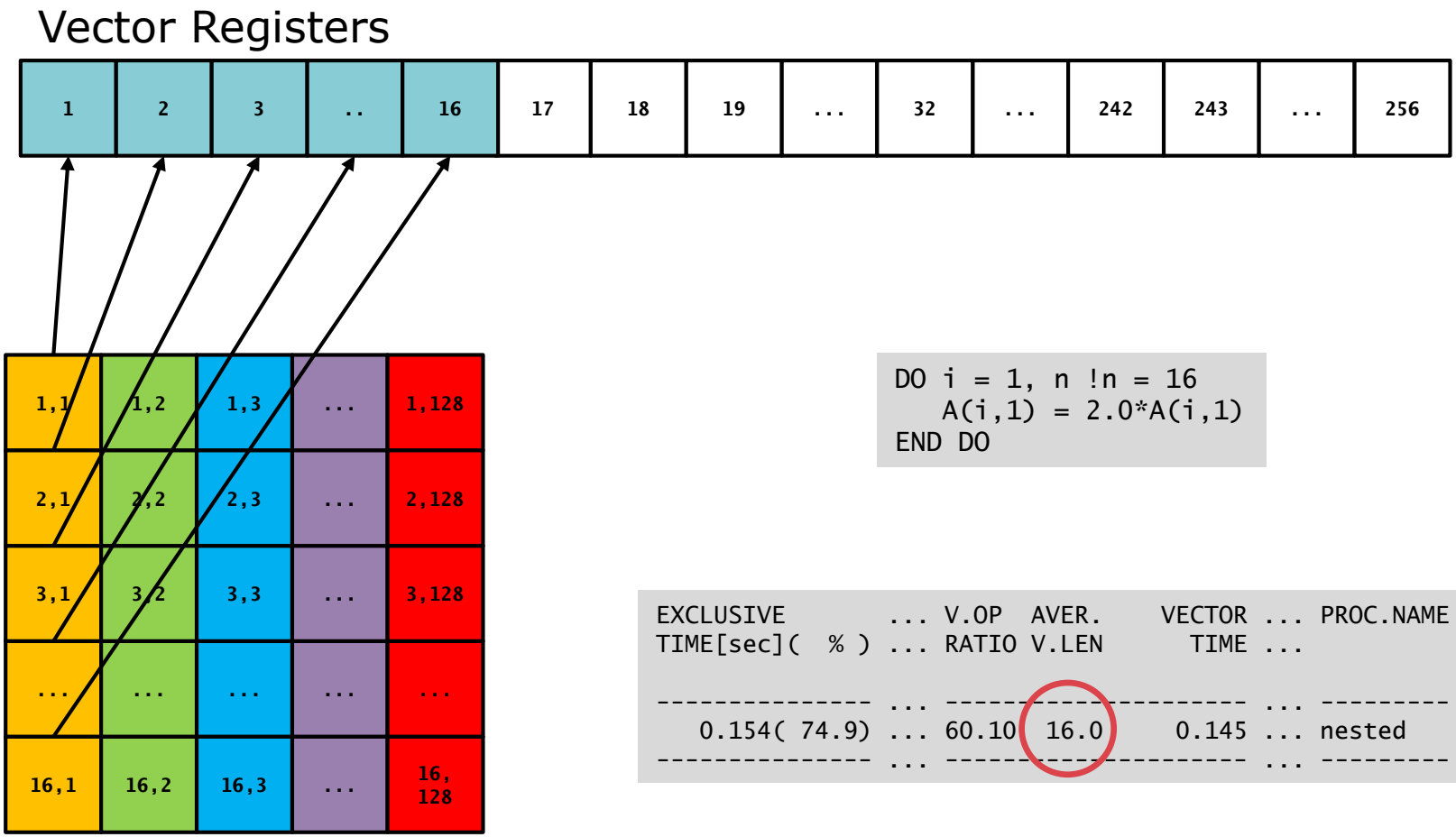
Innermost loop is vectorized:

```
13: +------>     DO j = 1, m
14: |V----->         DO i = 1, n
15: ||                  A(i,j) = 2.0*A(i,j)
16: |V-----         END DO
17: +------      END DO
```

```
let n = 16; m = 128
```


© NEC Corporation 2019

Collapsing – Increasing the Vector Length



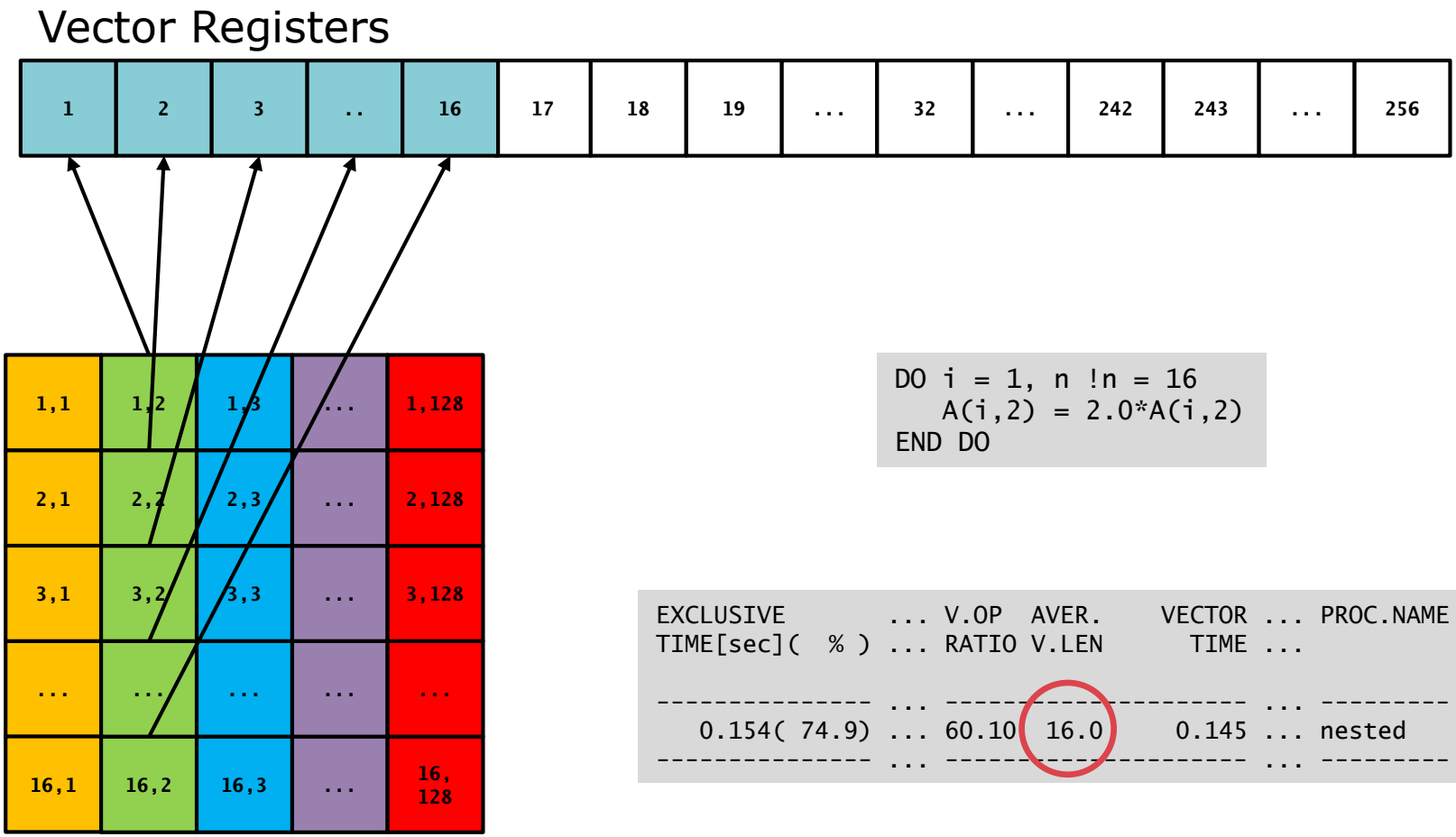

```
DO i = 1, n !n = 16
   A(i,1) = 2.0*A(i,1)
END DO
```

```
EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

---------------  ... -------------------- ... ---------
   0.154( 74.9)  ... 60.10  16.0     0.145 ... nested
---------------  ... -------------------- ... ---------
```


© NEC Corporation 2019

Collapsing – Increasing the Vector Length




```
DO i = 1, n !n = 16
   A(i,2) = 2.0*A(i,2)
END DO
```

```
EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % )  ... RATIO V.LEN      TIME ...

---------------  ... -------------------- ... ---------
   0.154( 74.9)  ... 60.10  16.0     0.145 ... nested
---------------  ... -------------------- ... ---------
```


© NEC Corporation 2019

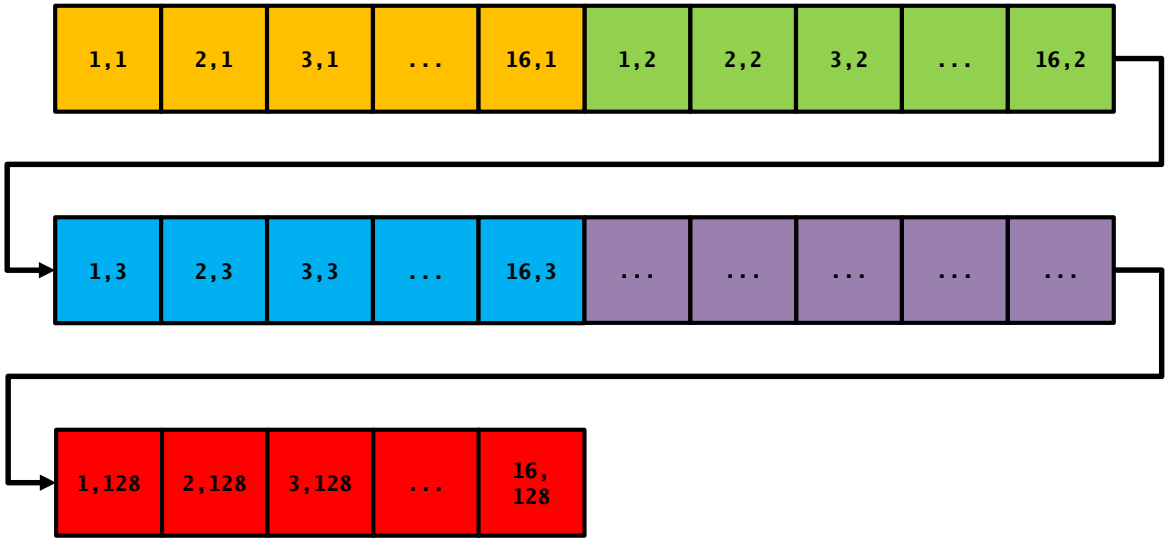
Collapsing – Increasing the Vector Length

Memory Layout in Fortran

Matrix Representation

1,1	1,2	1,3	...	1,128
2,1	2,2	2,3	...	2,128
3,1	3,2	3,3	...	3,128
...	...	...	...	...
16,1	16,2	16,3	...	16, 128

Actual Memory Layout



Matrix Address:
A(i,j)

Actual Address:
A(i,j) = LOC(A(1,1)) + (j-1)*n + i

A matrix of size (n,m) has the same memory layout as
A matrix of size (n*m,1)!

© NEC Corporation 2019

Collapsing – Increasing the Vector Length

Consider the following collapsed loop:

```
DO i = 1, n*m
   A(i,1) = 2.0*A(i,1)
END DO
```

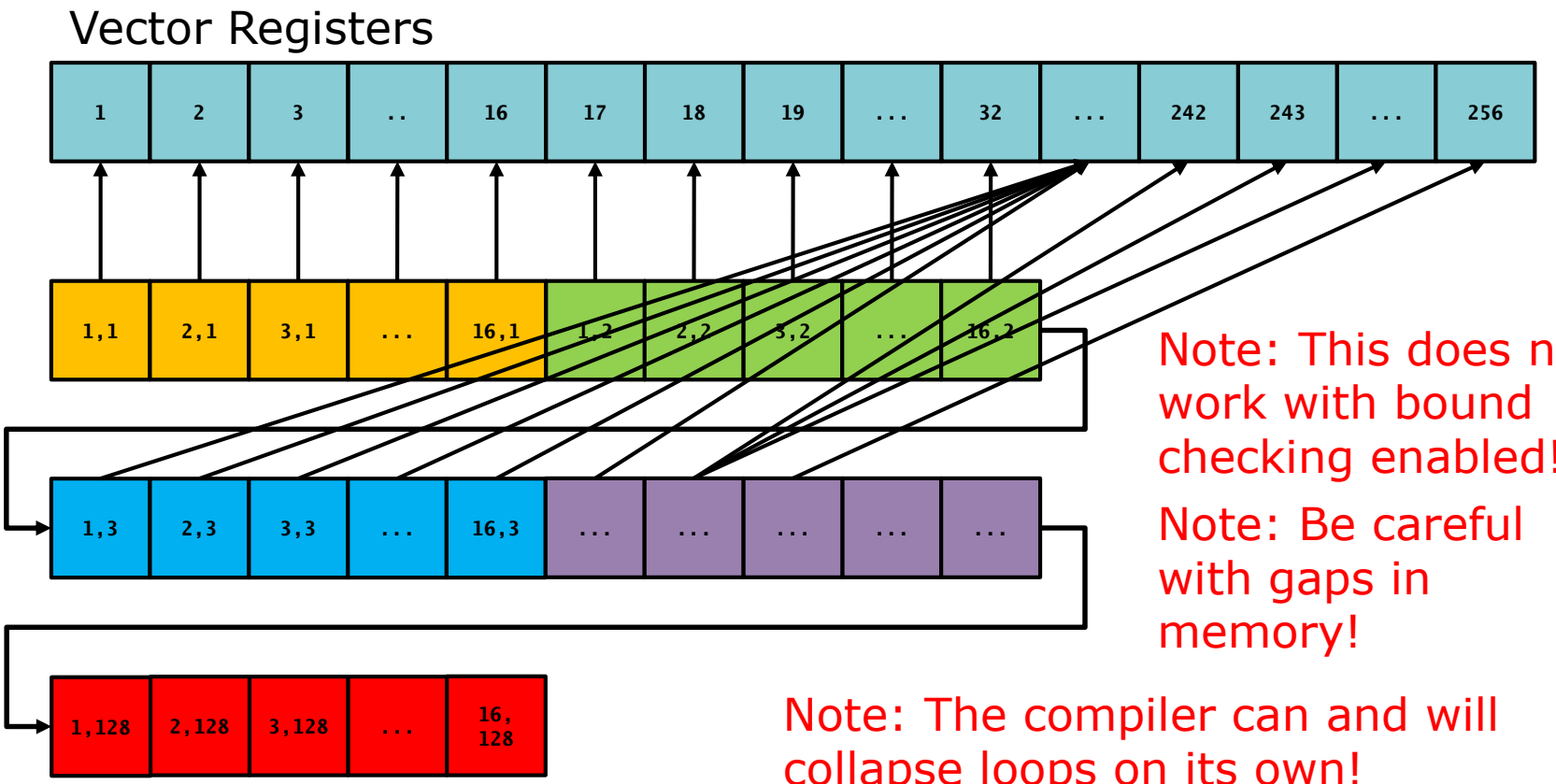
Innermost (only) loop is vectorized:

```
21: V------>     DO i = 1, n*m
22: |               A(i,1) = 2.0*A(i,1)
23: V------      END DO
```

let n = 16; m = 128 ➔ n*m = 2048

© NEC Corporation 2019

Collapsing – Increasing the Vector Length



Note: This does not work with bound checking enabled!

Note: Be careful with gaps in memory!

Note: The compiler can and will collapse loops on its own!

```
DO i = 1, n*m ! 1-256
   A(i,1) = 2.0*A(i,1)
END DO
```

```
EXCLUSIVE        ... V.OP  AVER.    VECTOR ... PROC.NAME
TIME[sec](  % ) ... RATIO V.LEN      TIME ...

--------------- ... ------------------- ... ---------
   0.154( 74.9) ... 60.10   16.0     0.145 ... nested
   0.020(  9.8) ... 94.80  256.0     0.018 ... collapsed
--------------- ... ------------------- ... ---------
```


© NEC Corporation 2019

Loop Pushing

Loop pushing

Call to routine obstructs vectorization

```
SUBROUTINE work1(a, b,c)
REAL, DIMENSION(:,:,:) :: a, b, c
INTEGER :: i, j, k
REAL, DIMENSION(ke) :: loc

DO j=1,je
   DO i=1,ie
      DO k=1,ke
         a(i,j,k)=b(i,j,k)+REAL(k)
      END DO
      CALL work2(i,j,b,c,loc)
      DO k=1,ke
         a(i,j,k)=loc(k)*c(i,j,k)
      END DO
   END DO
END DO
END SUBROUTINE
```

Loop length in routine is inefficient

```
SUBROUTINE work2(i,j,b,c,loc)
INTEGER i,j,k
REAL, DIMENSION(:,:,:) :: b, c
REAL, DIMENSION(:) :: loc
loc(:) = 0
DO k=1,ke
   b(i,j,k)=b(i,j,k)+c(i,j,k)
END DO
DO k=2,ke-1
   loc(k)=REAL(i+j)/b(i,j,k)
END DO
END SUBROUTINE
```


© NEC Corporation 2019

Loop pushing

Goal: Separate calculation and subroutine calls.
Every intermediate step will compile and run correctly.

```
SUBROUTINE work1(a, b,c)
REAL, DIMENSION(:,:,:) :: a, b, c
INTEGER :: i, j, k
REAL, DIMENSION(      ke) :: loc
DO j=1,je
   DO i=1,ie
      DO k=1,ke
         a(i,j,k)=b(i,j,k)+REAL(k)
      END DO




      CALL work2(i,j,b,c,loc)




      DO k=1,ke
         a(i,j,k)=loc(    k)*c(i,j,k)
      END DO
   END DO
END DO
END SUBROUTINE
```

```
SUBROUTINE work2(i,j,b,c,loc)
INTEGER i,j,k
REAL, DIMENSION(:,:,:) :: b, c
REAL, DIMENSION(    :) :: loc
loc(    :) = 0


      DO k=1,ke
         b(i,j,k)=b(i,j,k)+c(i,j,k)
      END DO




      DO k=2,ke-1
         loc(    k)=REAL(i+j)/b(i,j,k)
      END DO



END SUBROUTINE
```


© NEC Corporation 2019

Loop pushing

1. Promote variables to arrays

```
SUBROUTINE work1(a, b,c)
REAL, DIMENSION(:,:,:) :: a, b, c
INTEGER :: i, j, k
REAL, DIMENSION(ie,je,ke) :: loc = 0
DO j=1,je
   DO i=1,ie
      DO k=1,ke
         a(i,j,k)=b(i,j,k)+REAL(k)
      END DO




      CALL work2(i,j,b,c,loc)




      DO k=1,ke
         a(i,j,k)=loc(i,j,k)*c(i,j,k)
      END DO
   END DO
END DO
END SUBROUTINE
```

```
SUBROUTINE work2(i,j,b,c,loc)
INTEGER i,j,k
REAL, DIMENSION(:,:,:) :: b, c
REAL, DIMENSION(:,:,:) :: loc
loc(:,:,:) = 0


      DO k=1,ke
         b(i,j,k)=b(i,j,k)+c(i,j,k)
      END DO




      DO k=2,ke-1
         loc(i,j,k)=REAL(i+j)/b(i,j,k)
      END DO



END SUBROUTINE
```


© NEC Corporation 2019

Loop pushing

2. Separate loops and isolate subroutine

```
SUBROUTINE work1(a, b,c)
REAL, DIMENSION(:,:,:) :: a, b, c
INTEGER :: i, j, k
REAL, DIMENSION(ie,je,ke) :: loc
DO j=1,je
   DO i=1,ie
      DO k=1,ke
         a(i,j,k)=b(i,j,k)+REAL(k)
      END DO
   END DO
END DO
DO j=1,je
   DO i=1,ie
      CALL work2(i,j,b,c,loc)
   END DO
END DO
DO j=1,je
   DO i=1,ie
      DO k=1,ke
         a(i,j,k)=loc(i,j,k)*c(i,j,k)
      END DO
   END DO
END DO
END SUBROUTINE
```

```
SUBROUTINE work2(i,j,b,c,loc)
INTEGER i,j,k
REAL, DIMENSION(:,:,:) :: b, c
REAL, DIMENSION(:,:,:) :: loc
loc(:,:,:) = 0


      DO k=1,ke
         b(i,j,k)=b(i,j,k)+c(i,j,k)
      END DO




      DO k=2,ke-1
         loc(i,j,k)=REAL(i+j)/b(i,j,k)
      END DO



END SUBROUTINE
```


© NEC Corporation 2019

Loop pushing

3. Push loops into subroutine

```
SUBROUTINE work1(a, b,c)
REAL, DIMENSION(:,:,:) :: a, b, c
INTEGER :: i, j, k
REAL, DIMENSION(ie,je,ke) :: loc
DO j=1,je
   DO i=1,ie
      DO k=1,ke
         a(i,j,k)=b(i,j,k)+REAL(k)
      END DO
   END DO
END DO


      CALL work2(i,j,b,c,loc)


DO j=1,je
   DO i=1,ie
      DO k=1,ke
         a(i,j,k)=loc(i,j,k)*c(i,j,k)
      END DO
   END DO
END DO
END SUBROUTINE
```

```
SUBROUTINE work2(i,j,b,c,loc)
INTEGER i,j,k
REAL, DIMENSION(:,:,:) :: b, c
REAL, DIMENSION(:,:,:) :: loc
loc(:,:,:) = 0
DO j=1,je
   DO i=1,ie
      DO k=1,ke
         b(i,j,k)=b(i,j,k)+c(i,j,k)
      END DO




      DO k=2,ke-1
         loc(i,j,k)=REAL(i+j)/b(i,j,k)
      END DO
   END DO
END DO

END SUBROUTINE
```


© NEC Corporation 2019

Loop pushing

4. Change loop order for optimal memory access

```
SUBROUTINE work1(a, b,c)
REAL, DIMENSION(:,:,:) :: a, b, c
INTEGER :: i, j, k
REAL, DIMENSION(ie,je,ke) :: loc
DO k=1,ke
   DO j=1,je
      DO i=1,ie
         a(i,j,k)=b(i,j,k)+REAL(k)
      END DO
   END DO
END DO


      CALL work2(    b,c,loc)


DO k=1,ke
   DO j=1,je
      DO i=1,ie
         a(i,j,k)=loc(i,j,k)*c(i,j,k)
      END DO
   END DO
END DO
END SUBROUTINE
```

```
SUBROUTINE work2(    b,c,loc)
INTEGER i,j,k
REAL, DIMENSION(:,:,:) :: b, c
REAL, DIMENSION(:,:,:) :: loc
loc(:,:,:) = 0
DO k=1,ke
   DO j=1,je
      DO i=1,ie
         b(i,j,k)=b(i,j,k)+c(i,j,k)
      END DO
   END DO
END DO
DO k=2,ke-1
   DO j=1,je
      DO i=1,ie
         loc(i,j,k)=REAL(i+j)/b(i,j,k)
      END DO
   END DO
END DO

END SUBROUTINE
```


© NEC Corporation 2019

Loop Unrolling

Loop Unrolling

```
!Loads for one i iteration: 2


DO j = 1, m-1, 1

   DO i = 1, n
      A(i,j  ) = B(i,j  ) + B(i,j+1)



   END DO
END DO
```

- Every A(i,j) depends on two in j consecutive values of B.
- This generates two loading instructions (B(i,j),B(i,j+1)) for one iteration of i.
- B(i,j+1) will again be loaded in the next iteration of j, thus creating unnecessary loads.

© NEC Corporation 2019

Loop Unrolling

```
!Loads for two i iterations: 3


DO j = 1, m-1, 2
   !NEC$ ivdep
   DO i = 1, n
      A(i,j  ) = B(i,j  ) + B(i,j+1)
      A(i,j+1) = B(i,j+1) + B(i,j+2)


   END DO
END DO
```

- Partially unrolling the j loop the loading is improved
- This generates three loading instructions (B(i,j),B(i,j+1),B(i,j+2)) for two iterations of i.
- This is not generalized, as the remainder due to the stride might be untreated

© NEC Corporation 2019

Loop Unrolling

```
!Loads for four i iterations: 5


DO j = 1, m-1, 4
   !NEC$ ivdep
   DO i = 1, n
      A(i,j  ) = B(i,j  ) + B(i,j+1)
      A(i,j+1) = B(i,j+1) + B(i,j+2)
      A(i,j+2) = B(i,j+2) + B(i,j+3)
      A(i,j+3) = B(i,j+3) + B(i,j+4)
   END DO
END DO
```

- Partially unrolling the j loop the loading is improved
- This generates five loading instructions (B(i,j),B(i,j+1),B(i,j+2), B(i,j+3),B(i,j+4)) for four iterations of i.
- This is not generalized, as the remainder due to the stride might be untreated

© NEC Corporation 2019

Loop Unrolling

```
!Loads for four i iterations: 5

!NEC$ outerloop_unroll(4)
DO j = 1, m-1
   !NEC$ ivdep
   DO i = 1, n
      A(i,j  ) = B(i,j  ) + B(i,j+1)



   END DO
END DO
```

- Utilizing the outerloop_unroll directive prevents mistakes and allows for more flexibility
- This generates five loading instructions (B(i,j),B(i,j+1),B(i,j+2), B(i,j+3),B(i,j+4)) for four iterations of i.
- This automatically treats a possible remainder correctly.
- Compiler can and will usually unroll by itself with a length of 4. (-O3 optimization)

© NEC Corporation 2019

Orchestrating a brighter world

NEC