Orchestrating a brighter world NEC

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

3rd Aurora Deep Dive Workshop

Vectorization Optimization Techniques

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.

Vectorization Optimization Techniques

1. Compiler-driven optimization
   - Optimization flags
   - Compiler Directives
   - Function Inlining

2. Advanced Vectorization Concepts
   - Index Lists
   - Direct Vector Register Usage

Compiler-driven optimization

Optimization Flags

Collective Optimization Flags

The -O flag can be used to switch on/off a set of optimizations

-OX	Specifies optimization level by X. (default: -O2)
-O0	Disables any optimizations.
-O1	Optimizations without any side-effects.
-O2	Optimizations which may cause side-effects.
-O3	Optimizations which cause side-effects.
-O4	Aggressive optimization (violates language standard).

Use the compiler options listing from -report-all
to find out details on the individual activated options

Extensive lists of compiler options are available in the compiler user guides.

Collective Optimization Flags

The -O flag can be used to switch on/off a set of optimizations

Optimization Level and Options' Defaults

Option Name	*-O4*	*-O3*	*-O2*	*-O1*	*-O0*
-faddress-taken-across-func (*1)	no	no	no	no	yes
-fassociative-math	yes	yes	yes	no	no
-ffast-math	yes	yes	yes	yes	no
-finline-copy-arguments	no	yes	yes	yes	yes
-fipa	yes	yes	no	no	no
-fignore-volatile	yes	no	no	no	no
-floop-collapse	yes	yes	no	no	no
-floop-interchange	yes	yes	no	no	no
-floop-normalize	yes	yes	no	no	no
-floop-strip-mine	yes	yes	no	no	no
-floop-unroll	yes	yes	yes	no	no
-fmatrix-multiply	yes	yes	no	no	no

[…]

Extensive lists of compiler options are available in the compiler user guides.

Compiler Directives

Compiler Directives

Directives are hints on how specific code passages should be compiled.
C uses pragmas for directives.
Fortran uses comments with specific beginnings

```
C/C++: #pragma _NEC directive-name [ clause] ...
```

```
Fortran:       !NEC$ directive-name [ clause] ... (Free form)
               *NEC$ directive-name [ clause] ... (Fixed form)
               cNEC$ directive-name [ clause] ... (Fixed form)
```

For examples on how to use specific directives in
various circumstances refer to the NEC vectorization training.

Extensive lists of compiler directives are available in the compiler user guides.

Example: Prevent loop exchange

```
MODULE testmod
   PUBLIC
CONTAINS
SUBROUTINE init_mat(ndim, nelem, a)
   IMPLICIT NONE
   INTEGER :: ndim, nelem
   REAL(8) :: a(ndim, nelem)
   INTEGER :: idim, ielem
   DO idim = 1, ndim !ndim=3
      DO ielem = 1, nelem
         a(idim, ielem) = &
         SIN(2.0*idim+ielem)
      END DO
   END DO
END SUBROUTINE init_mat
END MODULE testmod
```

```
 4:            SUBROUTINE init_mat(ndim, nelem, a)
 5:               IMPLICIT NONE
 6:               INTEGER :: ndim, nelem
 7:               REAL(8) :: a(ndim, nelem)
 8:               INTEGER :: idim, ielem
 9: X------>      DO idim = 1, ndim !ndim=3
10: |U----->         DO ielem = 1, nelem
11: ||      F           a(idim, ielem) = &
12: ||                  SIN(2.0*idim+ielem)
13: |U-----          END DO
14: X------       END DO
15:            END SUBROUTINE init_mat
```

```
$ nfort -c -report-all test.F90
```

- Exchanging the loops would only be beneficial if ndim was larger.

Example: Prevent loop exchange

```
MODULE testmod
   PUBLIC
CONTAINS
SUBROUTINE init_mat(ndim, nelem, a)
   IMPLICIT NONE
   INTEGER :: ndim, nelem
   REAL(8) :: a(ndim, nelem)
   INTEGER :: idim, ielem
!NEC$ NOINTERCHANGE
   DO idim = 1, ndim !ndim=3
      DO ielem = 1, nelem
         a(idim, ielem) = &
         SIN(2.0*idim+ielem)
      END DO
   END DO
END SUBROUTINE init_mat
END MODULE testmod
```

```
 4:            SUBROUTINE init_mat(ndim, nelem, a)
 5:               IMPLICIT NONE
 6:               INTEGER :: ndim, nelem
 7:               REAL(8) :: a(ndim, nelem)
 8:               INTEGER :: idim, ielem
 9:            !NEC$ NOINTERCHANGE
10: +------>      DO idim = 1, ndim
11: |V----->         DO ielem = 1, nelem
12: ||                  a(idim, ielem) = &
13: ||                  SIN(2.0*idim+ielem)
14: |V-----         END DO
15: +------      END DO
16:            END SUBROUTINE init_mat
```

```
$ nfort -c -report-all test.F90
```

- Exchanging the loops would only be beneficial if ndim was larger.
- Structure appears once => Loop directive.

Options Directive

The OPTIONS directive allows to define file specific command line compiler options within the source file.

```
!NEC$ OPTIONS “-fno-loop-interchange”
MODULE testmod
   PUBLIC
CONTAINS
SUBROUTINE init_mat(ndim, nelem, a)
   IMPLICIT NONE
   INTEGER :: ndim, nelem
   REAL(8) :: a(ndim, nelem)
   INTEGER :: idim, ielem
   DO idim = 1, ndim !ndim=3
      DO ielem = 1, nelem
         a(idim, ielem) = &
         SIN(2.0*idim+ielem)
      END DO
   END DO
END SUBROUTINE init_mat
END MODULE testmod
```

```
 5:            SUBROUTINE init_mat(ndim, nelem, a)
 6:               IMPLICIT NONE
 7:               INTEGER :: ndim, nelem
 8:               REAL(8) :: a(ndim, nelem)
 9:               INTEGER :: idim, ielem
10: +------>      DO idim = 1, ndim
11: |V----->         DO ielem = 1, nelem
12: ||                  a(idim, ielem) = &
13: ||                  SIN(2.0*idim+ielem)
14: |V-----         END DO
15: +------       END DO
16:            END SUBROUTINE init_mat
```

```
$ nfort –c -report-all test.F90
```

- Exchanging the loops would only be beneficial if ndim was larger.
- Structure appears once => Loop directive.
- Structure appears multiple times => File wide via OPTIONS directive.
- The OPTIONS directive overwrites command line options.

Options Directive

Rules:

- The options directive must be specified at the top of your source program.
- Two or more options directives can be specified in succession.
- Blank lines, comment lines, #line and #ident can be written before and between options directives.
- The options directive can be specified in the file included by #include at the top of your source file.

Restrictions:

- An option directive line cannot be continued.
- The directory specified by -I option in options directive is not searched for reading options directive.
- The upper limits of nesting level of files included by #include is 1000.
- The options directive cannot be specified in a file included by INCLUDE lines (Fortran)
- The compiler options that control linking or the compiler environment cannot be specified.

Function Inlining

Function Inlining

- In general, function inlining eliminates the call overhead of a function.

```
function inc (x)
   integer :: inc
   integer :: x
   inc = x + 1
end function inc

program test
   integer :: sum
   integer :: i, j
   do i = 1, 100000
       sum = 0
       do j = 1, 100000
          sum = sum + inc(i)
       end do
   end do
end program test
```

```
function inc (x)
   integer :: inc
   integer :: x
   inc = x + 1
end function inc

program test
   integer :: sum
   integer :: i, j
   do i = 1, 100000
       sum = 0
       do j = 1, 100000
          sum = sum + i + 1
       end do
   end do
end program test
```

```
$ gfortran-o test.x test.F90
```

t: 24.7 s

t: 15.2 s

Function Inlining

- On SX Aurora, function calls prohibit vectorization!

```
REAL(8) FUNCTION avg(a,b)
   REAL(8) :: a, b
   avg = 0.5*(a+b)
END FUNCTION avg

PROGRAM test
   IMPLICIT NONE
   REAL(8) :: avg
   INTEGER, PARAMETER :: n = 65536
   REAL(8) :: a(n), b(n), c
   INTEGER :: i
   a(:) = 1.0
   b(:) = 1.0
   DO i = 1, n
      c = c + avg(a(i), b(i))
   END DO
   WRITE(*,*) c
END PROGRAM test
```

```
 1:            REAL(8) FUNCTION avg(a,b)
 2:               REAL(8) :: a, b
 3:               avg = 0.5*(a+b)
 4:            END FUNCTION avg
 5:
 6:            PROGRAM test
 7:               IMPLICIT NONE
 8:               REAL(8) :: avg
 9:               INTEGER, PARAMETER :: n = 65536
10:               REAL(8) :: a(n), b(n), c
11:               INTEGER :: i
12: V======>      a(:) = 1.0
13: V======>      b(:) = 1.0
14: +------>      DO i = 1, n
15: |                c = c + avg(a(i), b(i))
16: +------       END DO
17:               WRITE(*,*) c
18:            END PROGRAM test
```

```
$ nfort -report-all -o test.x test.F90
```

Function Inlining

- On SX Aurora, function calls prohibit vectorization!

```
REAL(8) FUNCTION avg(a,b)
   REAL(8) :: a, b
   avg = 0.5*(a+b)
END FUNCTION avg

PROGRAM test
   IMPLICIT NONE
   REAL(8) :: avg
   INTEGER, PARAMETER :: n = 65536
   REAL(8) :: a(n), b(n), c
   INTEGER :: i
   a(:) = 1.0
   b(:) = 1.0
   DO i = 1, n
      c = c + avg(a(i), b(i))
   END DO
   WRITE(*,*) c
END PROGRAM test
```

```
 1:            REAL(8) FUNCTION avg(a,b)
 2:               REAL(8) :: a, b
 3:               avg = 0.5*(a+b)
 4:            END FUNCTION avg
 5:
 6:            PROGRAM test
 7:               IMPLICIT NONE
 8:               REAL(8) :: avg
 9:               INTEGER, PARAMETER :: n = 65536
10:               REAL(8) :: a(n), b(n), c
11:               INTEGER :: i
12: V======>      a(:) = 1.0
13: V======>      b(:) = 1.0
14: V------>      DO i = 1, n
15: |       I        c = c + avg(a(i), b(i))
16: V------       END DO
17:               WRITE(*,*) c
18:            END PROGRAM test
```

```
$ nfort -finline-functions -report-all -o test.x test.F90
```

- The flag -finline-functions allows vectorization of the loop by inlining avg.

Function Inlining

- On SX Aurora, function calls prohibit vectorization!

```
REAL(8) FUNCTION avg(a,b)
   REAL(8) :: a, b
   avg = 0.5*(a+b)
END FUNCTION avg

PROGRAM test
   IMPLICIT NONE
   REAL(8) :: avg
   INTEGER, PARAMETER :: n = 65536
   REAL(8) :: a(n), b(n), c
   INTEGER :: i
   a(:) = 1.0
   b(:) = 1.0
   DO i = 1, n
      c = c + avg(a(i), b(i))
   END DO
   WRITE(*,*) c
END PROGRAM test
```

```
 1:            REAL(8) FUNCTION avg(a,b)
 2:               REAL(8) :: a, b
 3:               avg = 0.5*(a+b)
 4:            END FUNCTION avg
 5:
 6:            PROGRAM test
 7:               IMPLICIT NONE
 8:               REAL(8) :: avg
 9:               INTEGER, PARAMETER :: n = 65536
10:               REAL(8) :: a(n), b(n), c
11:               INTEGER :: i
12: V======>      a(:) = 1.0
13: V======>      b(:) = 1.0
14: V------>      DO i = 1, n
15: |          !NEC$ INLINE
16: |        I        c = c + avg(a(i), b(i))
17: V------       END DO
18:               WRITE(*,*) c
19:            END PROGRAM test
```

```
$ nfort -report-all -o test.x test.F90
```

- The flag -finline-functions allows vectorization of the loop by inlining avg.
- Alternatively, the INLINE directive can be used.

Crossfile Function Inlining

```
MODULE testmod
   PUBLIC :: avg
CONTAINS
   REAL(8) FUNCTION avg(a,b)
      REAL(8) :: a, b
      avg = 0.5*(a+b)
   END FUNCTION avg
END MODULE testmod
```

```
 4:                REAL(8) FUNCTION avg(a,b)
 5:                   REAL(8) :: a, b
 6:                   avg = 0.5*(a+b)
 7:                END FUNCTION avg
```

```
PROGRAM test
   USE testmod, ONLY : avg
   IMPLICIT NONE
   INTEGER, PARAMETER :: n = 65536
   REAL(8) :: a(n), b(n), c
   INTEGER :: i
   a(:) = 1.0
   b(:) = 1.0
   DO i = 1, n
      c = c + avg(a(i), b(i))
   END DO
   WRITE(*,*) c
END PROGRAM test
```

```
 1:            PROGRAM test
 2:               USE testmod, ONLY : avg
 3:               IMPLICIT NONE
 4:               INTEGER, PARAMETER :: n = 65536
 5:               REAL(8) :: a(n), b(n), c
 6:               INTEGER :: i
 7: V------>      a(:) = 1.0
 8: V------>      b(:) = 1.0
 9: +------>      DO i = 1, n
10: |                c = c + avg(a(i), b(i))
11: +------       END DO
12:               WRITE(*,*) c
13:            END PROGRAM test
```

```
$ nfort -report-all -c testmod.F90
$ nfort -finline-functions -report-all -c test.F90
$ nfort -report-all -o test.x test.o testmod.o
```

- The function avg is located in a different file and cannot be inlined.

Crossfile Function Inlining

```
MODULE testmod
   PUBLIC :: avg
CONTAINS
   REAL(8) FUNCTION avg(a,b)
      REAL(8) :: a, b
      avg = 0.5*(a+b)
   END FUNCTION avg
END MODULE testmod
```

```
 4:                REAL(8) FUNCTION avg(a,b)
 5:                   REAL(8) :: a, b
 6:                   avg = 0.5*(a+b)
 7:                END FUNCTION avg
```

```
PROGRAM test
   USE testmod, ONLY : avg
   IMPLICIT NONE
   INTEGER, PARAMETER :: n = 65536
   REAL(8) :: a(n), b(n), c
   INTEGER :: i
   a(:) = 1.0
   b(:) = 1.0
   DO i = 1, n
      c = c + avg(a(i), b(i))
   END DO
   WRITE(*,*) c
END PROGRAM test
```

```
 1:             PROGRAM test
 2:                USE testmod, ONLY : avg
 3:                IMPLICIT NONE
 4:                INTEGER, PARAMETER :: n = 65536
 5:                REAL(8) :: a(n), b(n), c
 6:                INTEGER :: i
 7: V------>       a(:) = 1.0
 8: V------>       b(:) = 1.0
 9: V------>       DO i = 1, n
10: |       I         c = c + avg(a(i), b(i))
11: V------        END DO
12:                WRITE(*,*) c
13:             END PROGRAM test
```

```
$ nfort -report-all -c testmod.F90
$ nfort –finline-functions -finline-file=testmod.F90 -report-all -c test.F90
$ nfort -report-all -o test.x test.o testmod.o
```

- The function avg is located in a different file and cannot be inlined directly.
- Specify the functions location with -finline-file or -finline-dicrectory to allow inlining.

Crossfile Function Inlining

```
MODULE testmod
   PUBLIC :: avg
CONTAINS
   REAL(8) FUNCTION avg(a,b)
      REAL(8) :: a, b
      avg = 0.5*(a+b)
   END FUNCTION avg
END MODULE testmod
```

```
 4:                REAL(8) FUNCTION avg(a,b)
 5:                   REAL(8) :: a, b
 6:                   avg = 0.5*(a+b)
 7:                END FUNCTION avg
```

```
PROGRAM test
   USE testmod, ONLY : avg
   IMPLICIT NONE
   INTEGER, PARAMETER :: n = 65536
   REAL(8) :: a(n), b(n), c
   INTEGER :: i
   a(:) = 1.0
   b(:) = 1.0
   DO i = 1, n
      c = c + avg(a(i), b(i))
   END DO
   WRITE(*,*) c
END PROGRAM test
```

```
 1:             PROGRAM test
 2:                USE testmod, ONLY : avg
 3:                IMPLICIT NONE
 4:                INTEGER, PARAMETER :: n = 65536
 5:                REAL(8) :: a(n), b(n), c
 6:                INTEGER :: i
 7: V------>       a(:) = 1.0
 8: V------>       b(:) = 1.0
 9: V------>       DO i = 1, n
10: |       I         c = c + avg(a(i), b(i))
11: V------        END DO
12:                WRITE(*,*) c
13:             END PROGRAM test
```

```
$ nfort -report-all -c testmod.F90
$ nfort –finline-functions -finline-file=testmod.F90 -report-all -c test.F90
$ nfort -report-all -o test.x test.o testmod.o
```

- Crossfile inlining can drastically increase the compilation time.

Limits of Function Inlining

```
PROGRAM test
   IMPLICIT NONE
   REAL(8) :: long_function
   INTEGER, PARAMETER :: n = 65536
   REAL(8) :: a(n), c
   INTEGER :: i
   a(:) = 1.0
   DO i = 1, n
      c = c + long_function(a(i))
   END DO
   WRITE(*,*) c
END PROGRAM test
```

```
 1:            PROGRAM test
 2:               IMPLICIT NONE
 3:               REAL(8) :: long_function
 4:               INTEGER, PARAMETER :: n = 65536
 5:               REAL(8) :: a(n), c
 6:               INTEGER :: i
 7: V------>      a(:) = 1.0
 8: +------>      DO i = 1, n
 9: |                c = c + long_function(a(i))
10: +------       END DO
11:               WRITE(*,*) c
12:            END PROGRAM test
```

- Under certain circumstances functions are not inlined.
  - The function is too long.
    The Limit can be extended with -finline-max-function-size=<n>
    “size” is not measured in C/Fortran-Code lines,
    but in lines of an internal representation, thus hard to estimate.

Limits of Function Inlining

```
PROGRAM test
   IMPLICIT NONE
   REAL(8) :: deep_function
   INTEGER, PARAMETER :: n = 65536
   REAL(8) :: a(n), c
   INTEGER :: i
   a(:) = 1.0
   DO i = 1, n
      c = c + deep_function(a(i))
   END DO
   WRITE(*,*) c
END PROGRAM test
```

```
 1:            PROGRAM test
 2:               IMPLICIT NONE
 3:               REAL(8) :: deep_function
 4:               INTEGER, PARAMETER :: n = 65536
 5:               REAL(8) :: a(n), c
 6:               INTEGER :: i
 7: V------>      a(:) = 1.0
 8: +------>      DO i = 1, n
 9: |                c = c + deep_function(a(i))
10: +------       END DO
11:               WRITE(*,*) c
12:            END PROGRAM test
```

- Under certain circumstances functions/routines are not inlined.
  - The function is too long.
    The Limit can be extended with -finline-max-function-size=<n>
    “size” is not measured in C/Fortran-Code lines,
    but in lines of an internal representation, thus hard to estimate.
  - Call depth is to large (A calls B calls C calls …).
    The depth limit can be extended with -finline-max-depth=<n>.

Limits of Function Inlining

```
PROGRAM test
   IMPLICIT NONE
   REAL(8) :: save_function
   INTEGER, PARAMETER :: n = 65536
   REAL(8) :: a(n), c
   INTEGER :: i
   a(:) = 1.0
   DO i = 1, n
      c = c + save_function(a(i))
   END DO
   WRITE(*,*) c
END PROGRAM test
```

```
 1:            PROGRAM test
 2:               IMPLICIT NONE
 3:               REAL(8) :: save_function
 4:               INTEGER, PARAMETER :: n = 65536
 5:               REAL(8) :: a(n), c
 6:               INTEGER :: i
 7: V------>      a(:) = 1.0
 8: +------>      DO i = 1, n
 9: |                c = c + save_function(a(i))
10: +------       END DO
11:               WRITE(*,*) c
12:            END PROGRAM test
```

- Under certain circumstances functions/routines are not inlined.
  - The function is too long.
    The Limit can be extended with -finline-max-function-size=<n>
    “size” is not measured in C/Fortran-Code lines,
    but in lines of an internal representation, thus hard to estimate.
  - Call depth is to large (A calls B calls C calls …).
    The depth limit can be extended with -finline-max-depth=<n>.
  - The function has save-variables (Fortran) / static-variables (C).
    In Fortran inlining can be used, if save variables
    can be replaced by module variables.
    In C this cannot be circumvented.

Limits of Function Inlining

```
REAL(8) FUNCTION long_function(a)
   REAL(8) :: a
!NEC$ ALWAYS_INLINE
   …
END FUNCTION long_function
```

- To overcome limits like -finline-max-function-size or -finline-max-depth the directive ALWAYS_INLINE can be utilized in the function to be inlined.
- This will allow the compiler to always inline the function, even in places where it is not desirable!

Traceback Information

Traceback Information

```
 1 SUBROUTINE init(n, a)
 2    INTEGER :: n, i
 3    REAL(8) :: a(n)
 4    a(:) = [(i, i=1, 2*n)]
 5 END SUBROUTINE init
 6
 7 PROGRAM test
 8    INTEGER, PARAMETER :: n = 4096
 9    REAL(8) :: a(n)
10    CALL init(n, a)
11 END PROGRAM test
```

```
$ nfort -o test.x test.F90
$ ./test.x
Segmentation fault
```

- Traceback information gives information about the current callstack in the event of an error driven program termination.

Traceback Information

```
 1 SUBROUTINE init(n, a)
 2    INTEGER :: n, i
 3    REAL(8) :: a(n)
 4    a(:) = [(i, i=1, 2*n)]
 5 END SUBROUTINE init
 6
 7 PROGRAM test
 8    INTEGER, PARAMETER :: n = 4096
 9    REAL(8) :: a(n)
10    CALL init(n, a)
11 END PROGRAM test
```

```
$ nfort -g -traceback=verbose -o test.x test.F90
$ export VE_TRACEBACK=VERBOSE
$ ./test.x
Segmentation fault: Address not mapped to object at 0x60000001cca0
[ 0] 0x60000001cca0 init_            traceback.F90:4
[ 1] 0x60000001ce38 MAIN__           traceback.F90:10
[ 2] 0x40b003ffffffff8 ?                 ?:?
Segmentation fault
```

- Traceback information gives information about the current callstack in the event of an error driven program termination.
- The traceback options should always be set to “verbose”. Otherwise only the address without the source code information is given and needs to be converted manually with “naddr2line”.
- For MPI programs additionally use NMPI_VE_TRACEBACK.

Advanced Vectorization Concepts

\Orchestrating a brighter world NEC

Index Lists

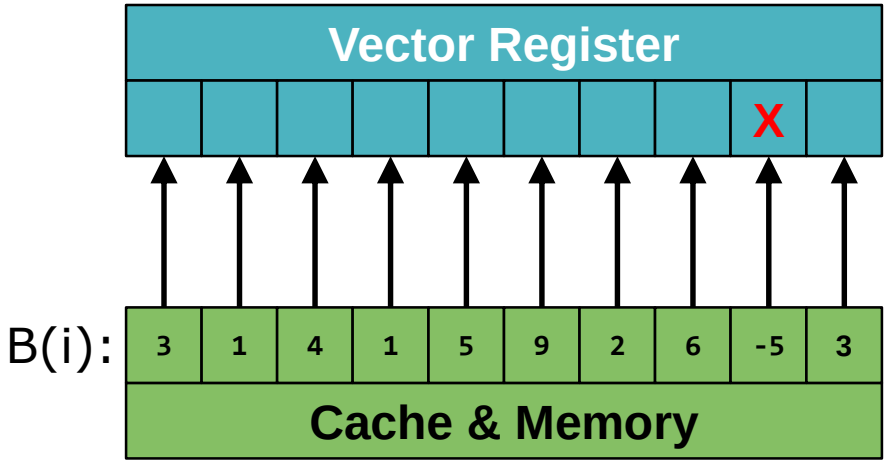
Index Lists

Consider the following loop with a condition:

```
DO i = 1, m
   IF (B(i) > 0.0) THEN
      A(i) = SQRT(B(i))
   END IF
END DO
```

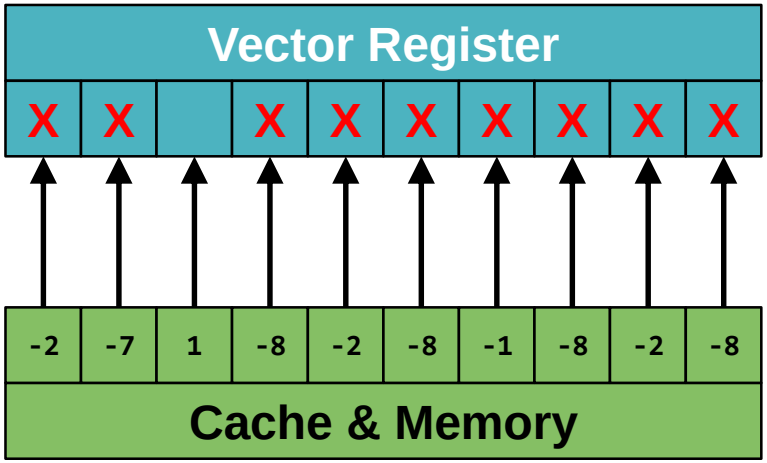
Loads and computations are performed for every element. When storing the result the mask is applied.

Condition is almost always true



Wasting very few loads and computations → Almost no loss in performance

Condition is almost never true



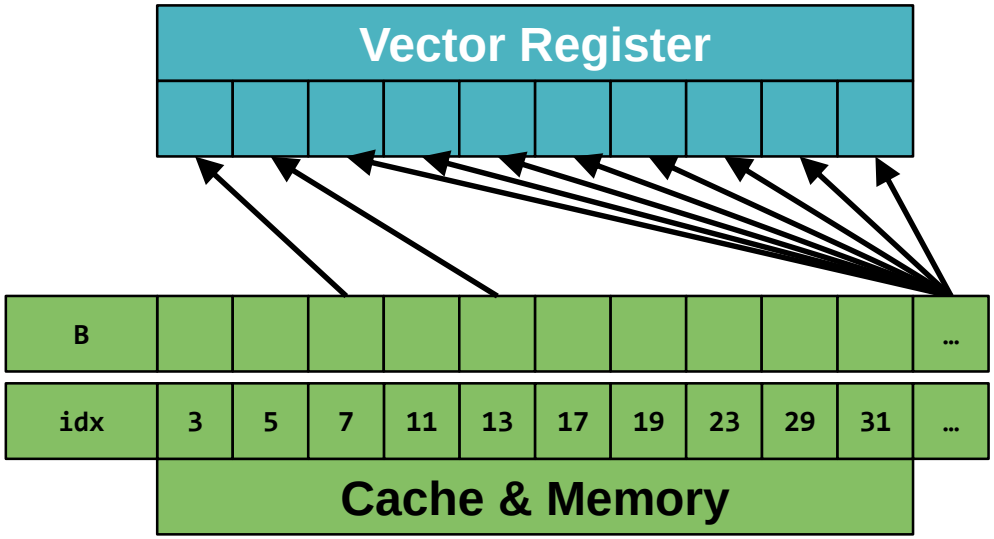
Wasting a lot of loads and computations → Significant decrease in performance

Index Lists

Use a list containing the indices of cases where the condition applies

```
DO j = 1, maxidx
   i = idx_list(j)
   A(i) = SQRT(B(i))
END DO
```

Condition is almost never true,
work only on memory where required



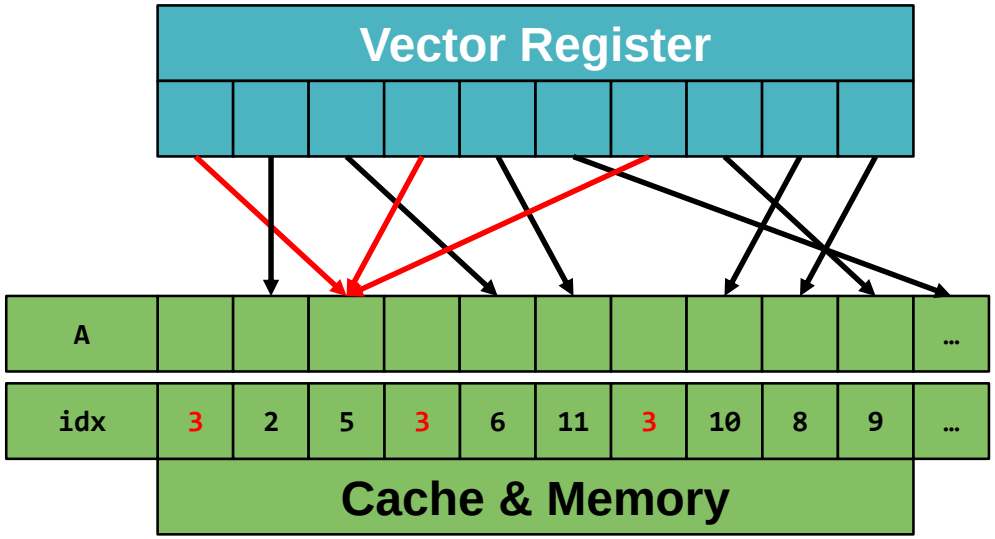
Wasting no loads
→ Maximum performance

Index Lists

An index list needs to be injective: Every index must only appear once in the list.

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

```
A(3) = A(3) + B(1)
A(2) = A(2) + B(2)
A(5) = A(5) + B(3)
A(3) = A(3) + B(4)
A(6) = A(6) + B(5)
```



- Execution in serial is fine.
- Simultaneous execution leads to an undefined result in A(3).

Index Lists

An index list needs to be injective: Every index must only appear once in the list.

```
do i = 1, n
   j = idx_list(i)
   A(j) = A(j) + B(i)
end do
```

```
[…]
9: vec( 122): Dependency unknown. Unvectorizable
dependency is assumed.: A
[…]
 7: S------>    do i = 1, n
     8: |             j = idx_list(i)
     9: |             A(j) = A(j) + B(i)
    10: S------     end do

[…]
```

- The compiler often cannot infer if a list is injective.
- In that case, the compiler is “better safe than sorry” and does not vectorize.

Index Lists

An index list needs to be injective: Every index must only appear once in the list.

```
!NEC$ ivdep
do i = 1, n
   j = idx_list(i)
   A(j) = A(j) + B(i)
end do
```

```
[…]
 8: vec( 101): Vectorized loop.
[…]
 7:            !NEC$ ivdep
     8: V------>    do i = 1, n
     9: |             j = idx_list(i)
    10: |       C     A(j) = A(j) + B(i)
    11: V------     end do
[…]
```

- Ivdep directive: Tell the compiler that the given list does not have dependencies.
- Global option –fivdep can be applied to the entire file.

Index Lists

Rewriting a loop with an index list needs two steps

```
! Original Loop

DO i = 1, n
   IF (B(i) > 0.0) THEN
      A(i) = SQRT(B(i))
   END IF
END DO
```

- Note: Building and using index lists is expensive.
- More complex load behavior Gather/Scatter patterns.
- Potentially more bank conflicts.
- They should only be used if everything else fails.

```
INTEGER :: idx, maxidx
INTEGER :: idx_list(n)

! 1. Setup index list
maxidx = 0
DO i = 1, n
   IF (B(i) > 0.0) THEN
      maxidx = maxidx + 1
      idx_list(maxidx) = i
   END IF
END DO

!2. Use index list
!NEC$ ivdep
DO idx = 1, maxidx
   i = idx_list(idx)
   A(i) = SQRT(B(i))
END DO
```

Index Lists – Vectorizing while loops

"While loops", or any loop whose iteration count is not known upon entering the loop, are impossible to vectorize automatically, but are a necessary programming technique. Index lists can be used to rewrite a while loop such that it is vectorizable.

```
DO i = 1, n
   DO WHILE (B(i) > C(i))
      B(i) = B(i) / A(i)
   END DO
END DO
```

```
12: vec( 103): Unvectorized loop.
12: vec( 113): Overhead of loop division is too large.
13: opt(1082): Backward transfers inhibit loop optimization.
13: vec( 103): Unvectorized loop.
13: vec( 108): Unvectorizable loop structure.

12: +------>     DO i = 1, n
13: |+----->        DO WHILE (B(i) > C(i))
14: ||                 B(i) = B(i) / A(i)
15: |+-----         END DO
16: +------      END DO
```

\Orchestrating a brighter world NEC

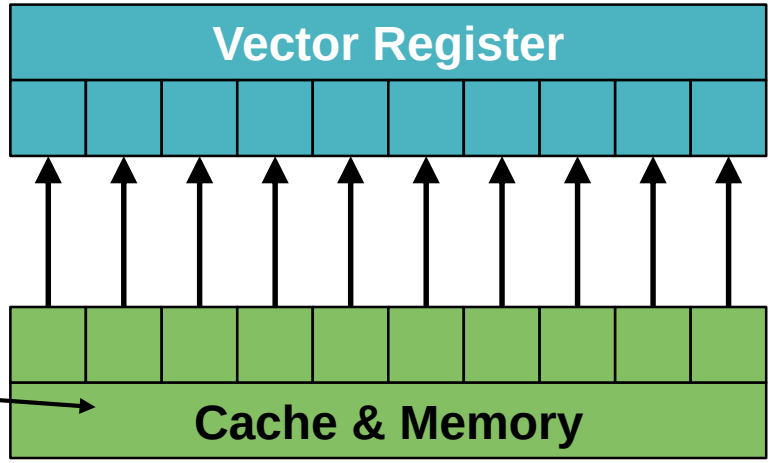
Direct Vector Register Usage

Vector Registers

Reminder: Loop unrolling optimizes cache reusage.

```
!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
```



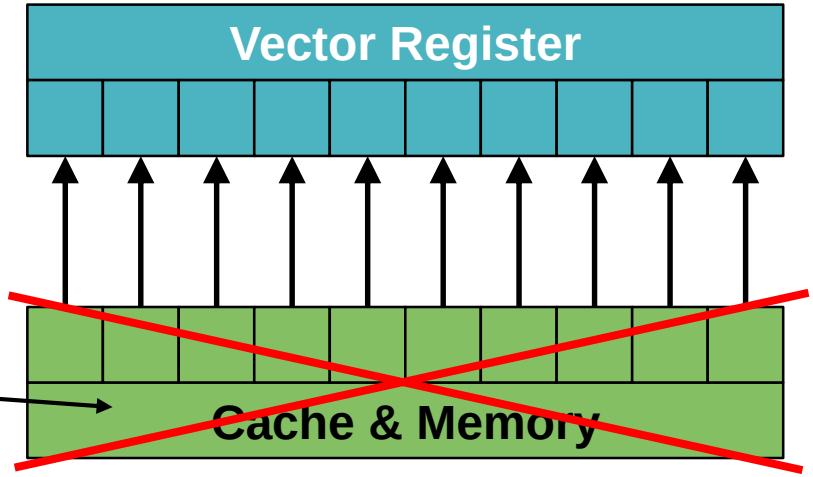
But: Time is still spent in the vector register transfer.

Vector Registers

Reminder: Loop unrolling optimizes cache reusage.

```
!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
```



But: Time is still spent in the vector register transfer.

SX Aurora allows for vector register reusage.

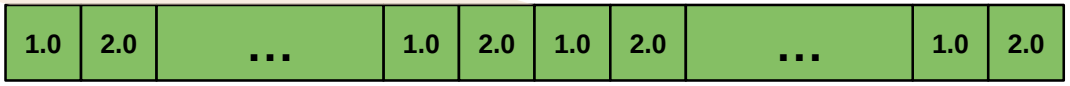
Vector Registers

```
DO j = 1, m
   DO i = 1, n
      C(i) = 0.1*C(i) + A(i)
   END DO
END DO
```

Vreg(A) (256 elements)

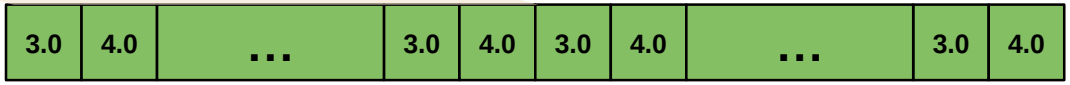
Apply stripmining to load cache data into vector registers

A



Vreg(C) (256 elements)

C



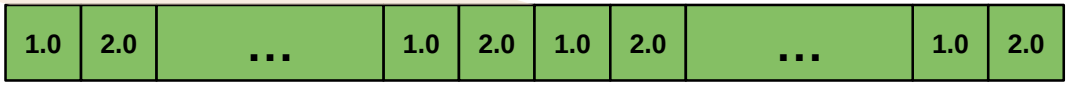
Vector Registers

```
DO j = 1, m
   DO i = 1, n
      C(i) = 0.1*C(i) + A(i)
   END DO
END DO
```

Vreg(A) (256 elements)

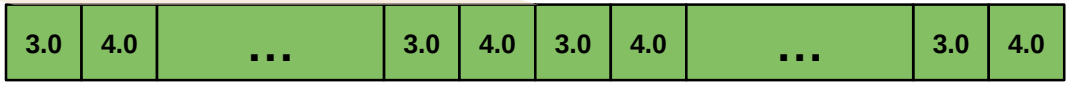
Apply stripmining to load cache data into vector registers

A



Vreg(C) (256 elements)

C



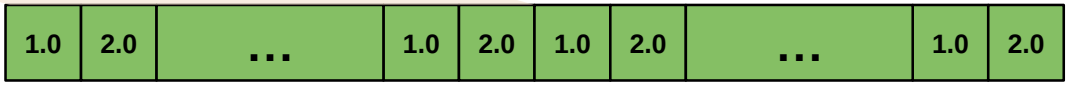
Vector Registers

```
DO j = 1, m
   DO i = 1, n
      C(i) = 0.1*C(i) + A(i)
   END DO
END DO
```

Vreg(A) (256 elements)

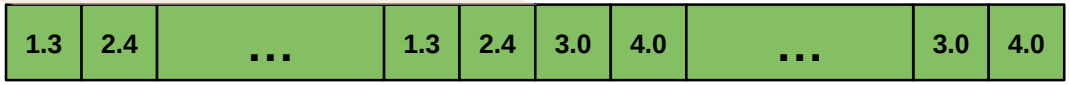
Apply stripmining to load cache data into vector registers

A



Vreg(C) (256 elements)

C



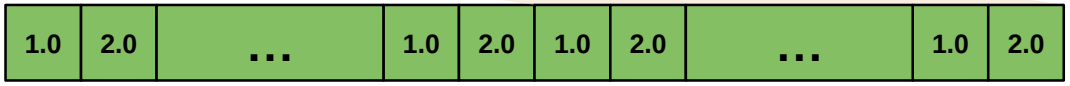
Vector Registers

```
DO j = 1, m
   DO i = 1, n
      C(i) = 0.1*C(i) + A(i)
   END DO
END DO
```

Vreg(A) (256 elements)

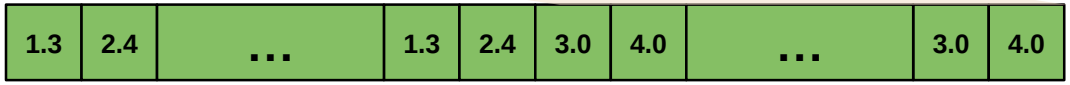
Apply stripmining to load cache data into vector registers

A



Vreg(C) (256 elements)

C



Vector Registers

Assign arrays to vector registers

```
REAL(8) :: vrega(256), vregc(256)
!NEC$ vreg(vrega)
!NEC$ vreg(vregc)
DO is = 1, n, 256
   ie = MIN(is-1+256, n)
   DO i = is, ie
      vrega(i-is+1) = A(i)
      vregc(i-is+1) = C(i)
   END DO
   DO j = 1, m
      DO i = is, ie
         vregc(i-is+1) = 0.1*vregc(i-is+1) + vrega(i-is+1)
      END DO
   END DO
   DO i = is, ie
      C(i) = vregc(i-is+1)
   END DO
END DO
```

Vector Registers

Outer loop over batches of vector length 256

```
REAL(8) :: vrega(256), vregc(256)
!NEC$ vreg(vrega)
!NEC$ vreg(vregc)
DO is = 1, n, 256
   ie = MIN(is-1+256, n)
   DO i = is, ie
      vrega(i-is+1) = A(i)
      vregc(i-is+1) = C(i)
   END DO
   DO j = 1, m
      DO i = is, ie
         vregc(i-is+1) = 0.1*vregc(i-is+1) + vrega(i-is+1)
      END DO
   END DO
   DO i = is, ie
      C(i) = vregc(i-is+1)
   END DO
END DO
```

Vector Registers

Determine the upper limit of each batch

```
REAL(8) :: vrega(256), vregc(256)
!NEC$ vreg(vrega)
!NEC$ vreg(vregc)
DO is = 1, n, 256
   ie = MIN(is-1+256, n)
   DO i = is, ie
      vrega(i-is+1) = A(i)
      vregc(i-is+1) = C(i)
   END DO
   DO j = 1, m
      DO i = is, ie
         vregc(i-is+1) = 0.1*vregc(i-is+1) + vrega(i-is+1)
      END DO
   END DO
   DO i = is, ie
      C(i) = vregc(i-is+1)
   END DO
END DO
```

Vector Registers

Load data into vector registers.

```
REAL(8) :: vrega(256), vregc(256)
!NEC$ vreg(vrega)
!NEC$ vreg(vregc)
DO is = 1, n, 256
   ie = MIN(is-1+256, n)
   DO i = is, ie
      vrega(i-is+1) = A(i)
      vregc(i-is+1) = C(i)
   END DO
   DO j = 1, m
      DO i = is, ie
         vregc(i-is+1) = 0.1*vregc(i-is+1) + vrega(i-is+1)
      END DO
   END DO
   DO i = is, ie
      C(i) = vregc(i-is+1)
   END DO
END DO
```

Vector Registers

```
REAL(8) :: vrega(256), vregc(256)
!NEC$ vreg(vrega)
!NEC$ vreg(vregc)
DO is = 1, n, 256
   ie = MIN(is-1+256, n)
   DO i = is, ie
      vrega(i-is+1) = A(i)
      vregc(i-is+1) = C(i)
   END DO
   DO j = 1, m
      DO i = is, ie
         vregc(i-is+1) = 0.1*vregc(i-is+1) + vrega(i-is+1)
      END DO
   END DO
   DO i = is, ie
      C(i) = vregc(i-is+1)
   END DO
END DO
```

Perform the calculation on the vector registers.

Vector Registers

```
REAL(8) :: vrega(256), vregc(256)
!NEC$ vreg(vrega)
!NEC$ vreg(vregc)
DO is = 1, n, 256
   ie = MIN(is-1+256, n)
   DO i = is, ie
      vrega(i-is+1) = A(i)
      vregc(i-is+1) = C(i)
   END DO
   DO j = 1, m
      DO i = is, ie
         vregc(i-is+1) = 0.1*vregc(i-is+1) + vrega(i-is+1)
      END DO
   END DO
   DO i = is, ie
      C(i) = vregc(i-is+1)
   END DO
END DO
```

Load the result of the batch back into the cache.

Vector Registers

```
15: +------>  DO is = 1, n, 256
16: |            ie = MIN(is-1+256, n)
17: |V----->     DO i = is, ie
18: ||      V       vrega(i-is+1) = A(i)
19: ||      V       vregc(i-is+1) = C(i)
20: |V-----      END DO
21: |+----->     DO j = 1, m
22: ||V---->        DO i = is, ie
23: |||     V          vregc(i-is+1) = 0.1*vregc(i-is+1)
                                        + vrega(i-is+1)
24: ||V----         END DO
25: |+-----      END DO
26: |V----->     DO i = is, ie
27: ||      V       C(i) = vregc(i-is+1)
28: |V-----      END DO
29: +------   END DO
```

Vector registers

- The Aurora has vector registers of length 256.
- Only for high end tuning.
  Often the quick solution, a simple loop unroll of length 8, gives you 80% of the performance.
- Vector registers are very fragile in their usage.
- Only use them for arithmetic operators (+,-,*,/).
- Usage of functions or calls in expression with vector registers
  e.g. `vreg(i) = SQRT(vreg(i))`
  can give massively wrong results.
- Use as few vector register assignments as possible to avoid side effects.

Orchestrating a brighter world

NEC