\Orchestrating a brighter world

NEC

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

Programming on Vector Machines

https://www.hpc.nec/forums/

\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.

How to …?

Welcome to the self-study Aurora vector training of NEC.

Download the training exercises from the Aurora Forum.

The training consists of 17 folders/exercises containing a README file with detailed instructions.

You can compare your solution with the reference solution of the NEC Deutschland team available in a separate archive in the Aurora Forum.

This presentation explains the theoretical background and techniques required to do the exercises.

A blue slide will show you when it is time for a specific exercise.

The NEC team wishes you an informative and fun time with the Aurora vector training.

Vector Programming Training

1. Introduction to Vector Computing
   - Exercise 01 – Operation Performance
   - Exercise 02 – Expensive Operations
   - Exercise 03 – Vector Memory Access Performance
2. NEC compiler
3. Performance Analysis
   - Exercise 04 – Simple Inhibitors
4. Vectorization Techniques
   - Exercise 05 – Collapsing
   - Exercise 06 – Loop Pushing
   - Exercises 07-09 – Index Lists

Vector Programming Training

5. Special Loop Structures
   - Exercise 10 – While Loop
   - Exercise 11 – Inner K-Loop
   - Exercise 12 – Search Loops
6. Data Reusage (Load/Store Optimization)
   - Exercise 13 – Loop Combination
   - Exercise 14 – Unrolling
   - Exercise 15 – Vector Registers
7. I/O Operation Optimization
   - Exercise 16 – Small Block IO

Vector Programming Training

8. [Conflicting Memory Access]
   - Exercise 17 – Hyperplane Ordering
9. [Short Loop Vectorization]
   - Exercise 18 – Short Loop Reduction

Introduction to 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:

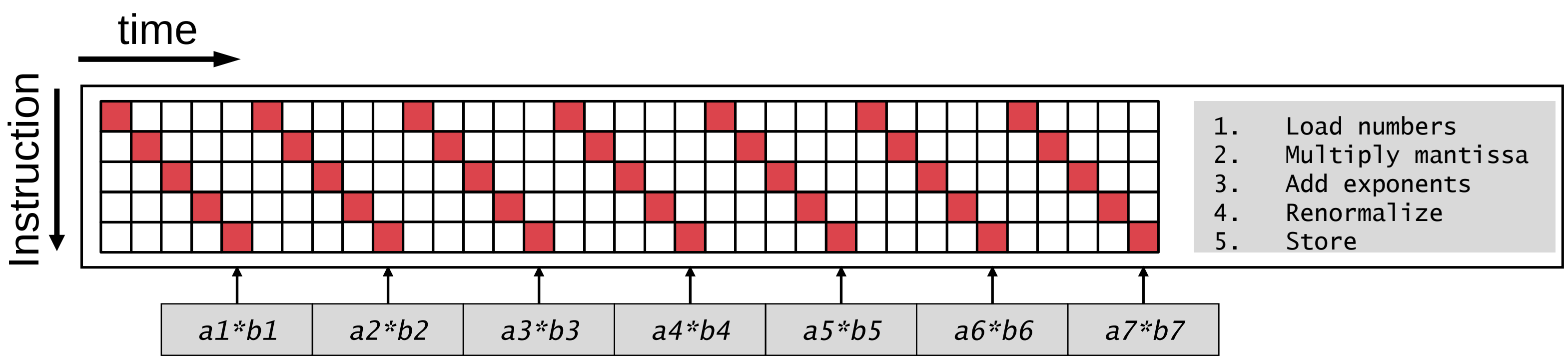
```
read vector instruction
decode vector instruction
fetch vector data
perform operation on data
   simultaneously
store vector results
```

“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.

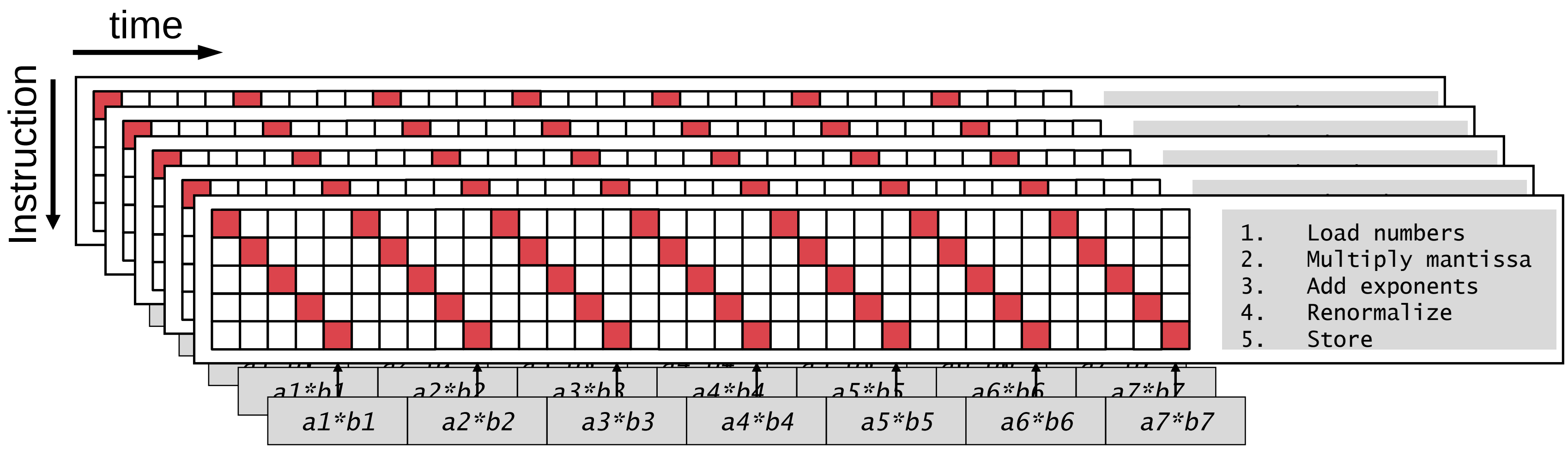
Scalar Processing

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



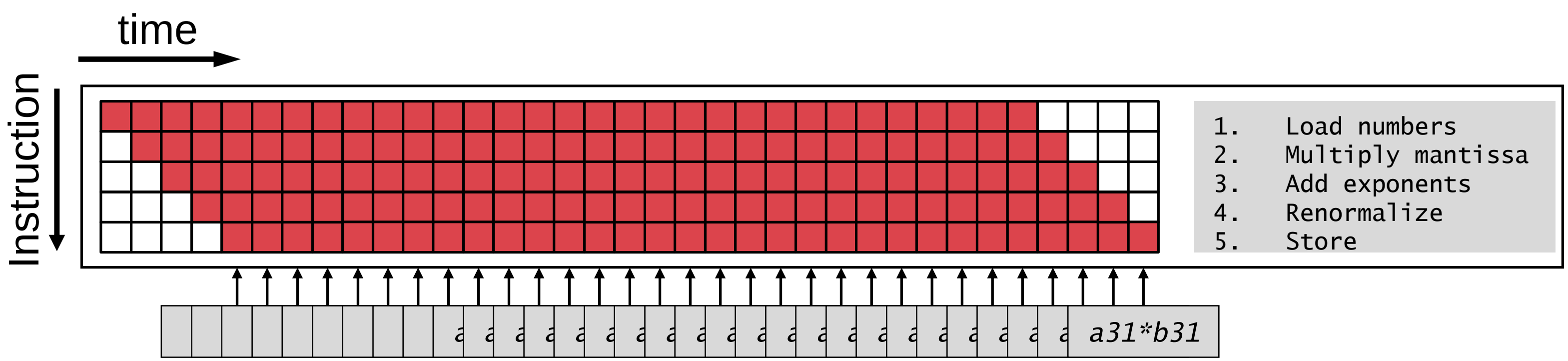
SIMD Processing (Modern Scalar)

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



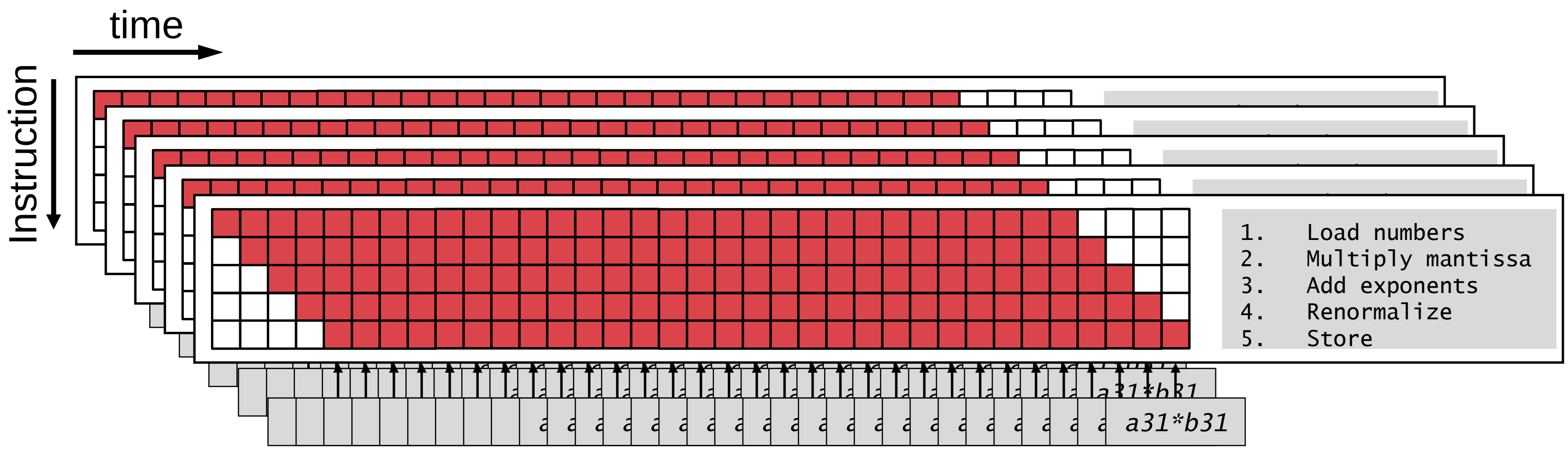
Vector Processing

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



Parallel Vector Processing (NEC Aurora)

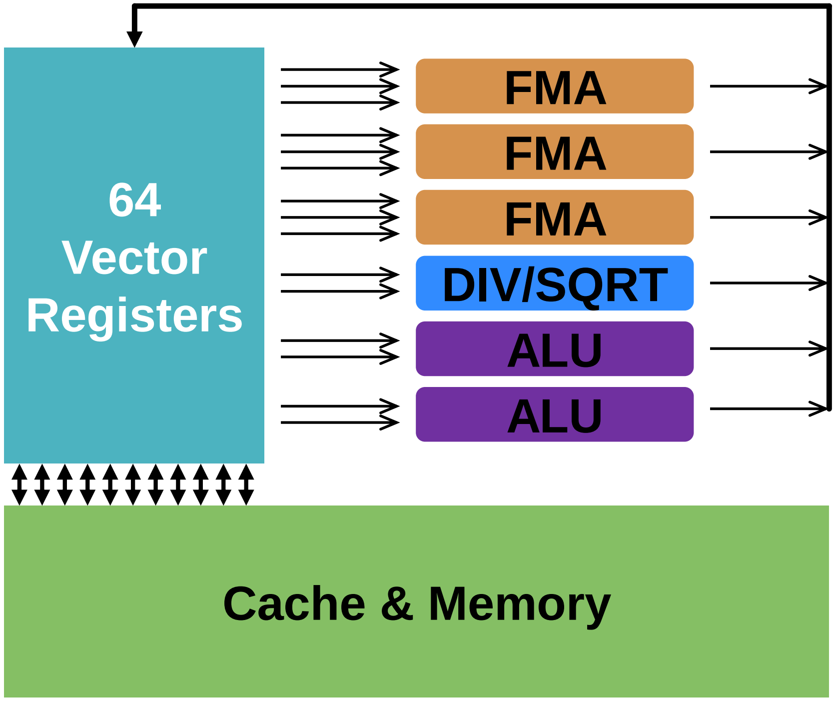
With Pipelining + Multiple Pipes:
Execute instructions in parallel to hide latency.
Parallel execution via multiple pipes



Cost of Instructions

- Pipes exist for various operations:
  - Arithmetic operations (ALU pipe):
    C = A ± B
    C = A * B
  - FMA (fused multiply add):
    D = A + B * C
  - Division (in Aurora):
    C = A/B
  - Square root (in Aurora):
    B = SQRT(A)
  - …
- Other operations are combinations of standard operations:
  - SIN, COS, TAN, ATAN,…
  - A**B, EXP, LOG,…
  - …

Aurora architecture scheme with available pipes



FTRACE - Runtime

```
$ 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)

Exercise 01 – Operation Performance

Exercise 02 – Expensive Operations

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
```

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
```

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
```

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
```

Vectorizable Structures

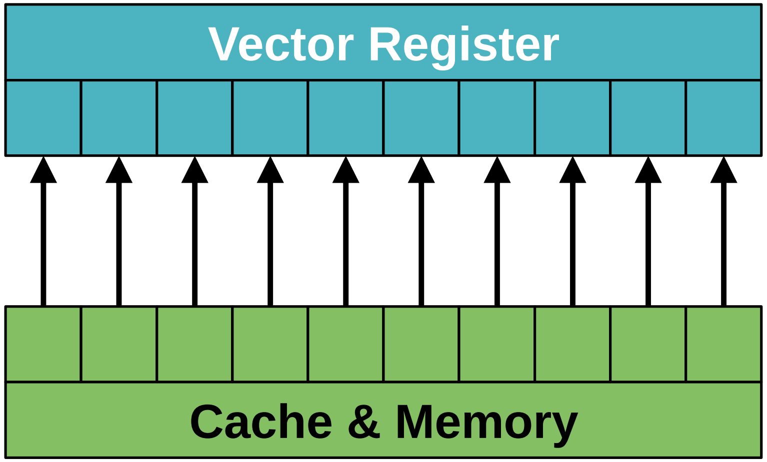
- In order to be (automatically) vectorizable a loop structure needs to fulfill certain criteria:
  - Loop count needs to be known upon entering the loop.
  - No I/O operations inside the loop.
  - Data needs to be parallel. Order of operation must not matter. (Exception for scatter instructions)
  -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

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)
```

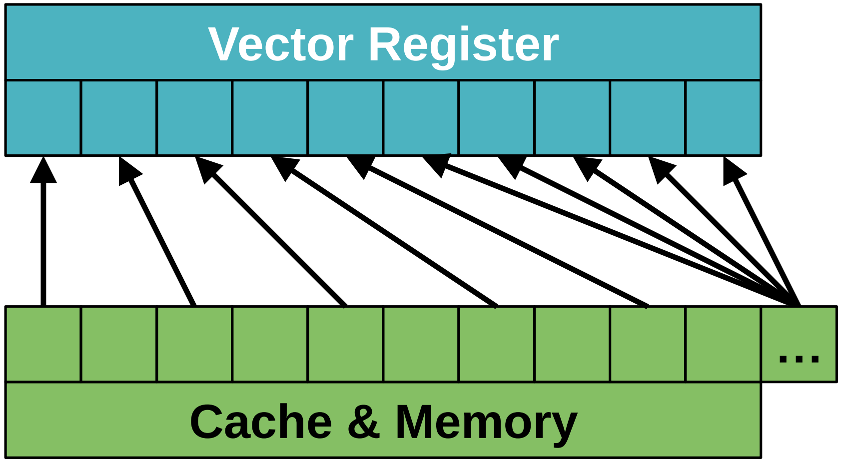


Optimal memory access.

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)
```

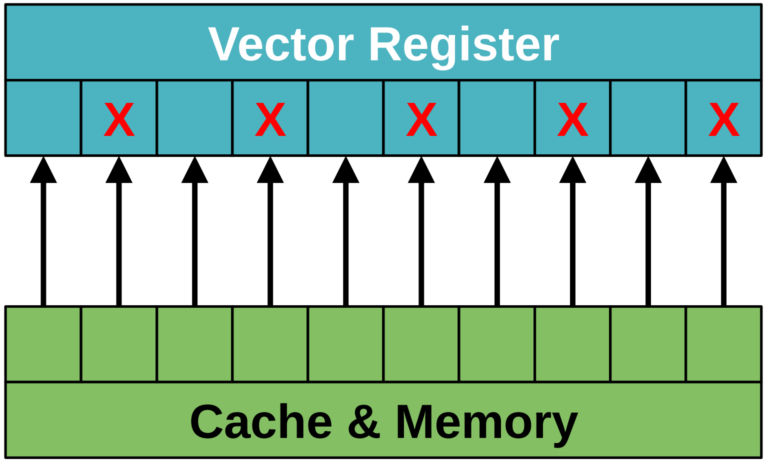


Not optimal due to only partially used cache lines.

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.

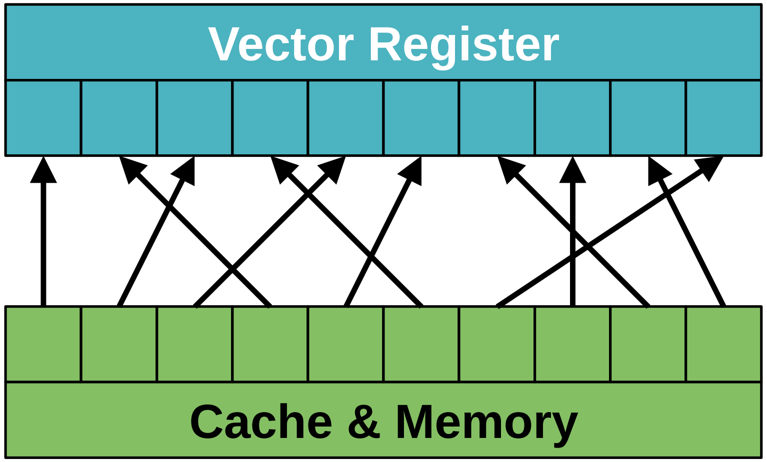
Note that all elements are loaded into the vector registers and operated on, but the write back is only performed if the condition applies.

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))
```

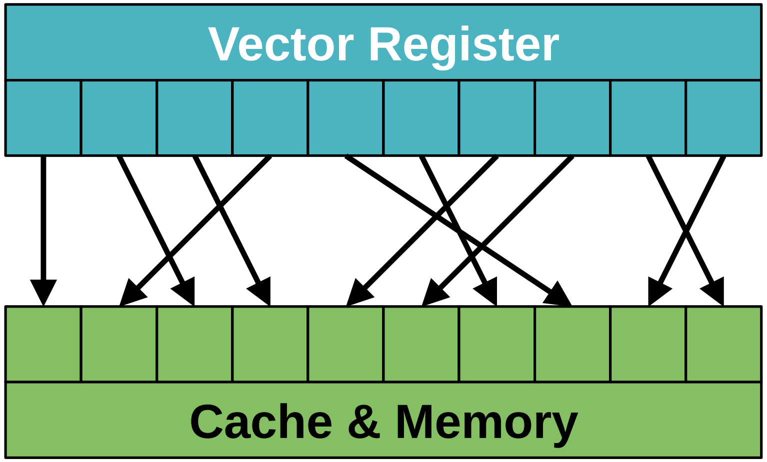


Inefficient due to random memory access, potential bank conflicts, partially used cache lines.

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)
```



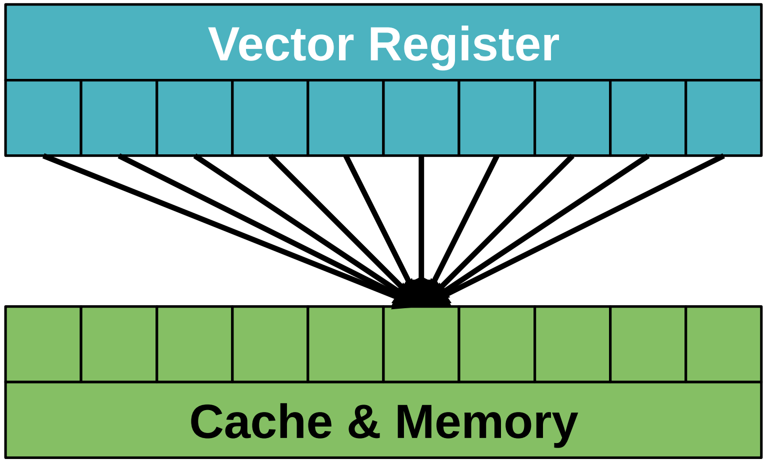
Inefficient due to random memory access, potential bank conflicts, partially used cache lines.

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/….

Exercise 03 – Vector Memory Access Performance

Aurora Compiler

Aurora Compilers

ncc	C Compiler
nc++	C++ Compiler
nfort	Fortran 2008 Compiler
nar	xar Archiver
mpincc	MPI C Compiler
mpinc++	MPI C++ Compiler
mpinfort	MPI Fortran 2008 Compiler

Syntax:

```
$ <compiler> <cmd line options> \
             -Wp,<preprocessor options> \
             -Wa,<assembler options> \
             -Wl,<linker options> … <source>
```

Example:

```
$ nfort -O3 -report-all -Wall -o test.x test.f90
```

Note that often the stack limit needs to be increased for successful compilations: `ulimit –s unlimited`

Running Aurora Programs

- In order to run Aurora programs it is sufficient to run the executable

```
./test.x
```

- To run on a specific vector engine use ve_exec
  The vector engines are specified by the –N option

```
ve_exec –N 0 ./test.x
```

Fortran Compiler Options

Use compiler flags to tell the compiler what it should, and can do during compile time.

`-c`	Create object file
`-o`	Specify output file
`-fcheck=<list>`	Check for “bounds”,…,”all”
`-g`	Produces debug symbols
`-traceback`	Enable traceback (Runtime: export VE_TRACEBACK=FULL)
`-Wall`	Enable all warnings
`-O[0-4]`	Optimization level
`-finline-functions`	Allow inlining
`-report-all`	Generate listing file
`-fdiag-vector=[0-3]`	Detail of vector diagnostics
`-ftrace`	Enable ftrace
…	

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

Compiler Directives

Use compiler directives to tell the compiler what is allowed and should be done at specific places in the source code.

```
!NEC$ ivdep                   Ignore vector dependencies
!NEC$ shortloop               Loop shorter than hardware
                                 vector length
!NEC$ outerloop_unroll(n)     Unroll outer loop n times
!NEC$ unroll(n)               Unroll loop n times
!NEC$ [no]vector              [Dis]allows vectorization
!NEC$ vreg(array-name)        Assigns array “array-name”
                                 to vector registers
!NEC$ loop_count_test         Allow conditional
                                 vectorization based
                                 on loop count
…
```

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

PROGINF – Vector Performance

```
********  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.

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 !

Format List File

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

```
NEC Fortran Compiler (2.1.24) for Vector
Engine   Fri Mar 29 07:01:25 2019
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 (2.1.24) for Vector
Engine   Fri Mar 29 07:01:25 2019
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
```

Format List File

```
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”).

Format List File

```
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.

Format List File

```
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.

Format List File

```
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.

Format List File

```
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("+").

Format List File

```
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.

Format List File – Loop Transformations

```
+------>                Loop is not vectorized

V------>                Loop is vectorized

S------>                Loop is partially vectorized

C------>                Loop is conditionally vectorized

U------>                Loop is unrolled

*------>                Loop is expanded

V======>                Array instruction is vectorized

W------>                Nested loops are collapsed
|*----->                   and vectorized

X------>                Nested loops are interchanged
|*----->                   and vectorized
```

Extensive lists of format lists symbols are available in the compiler user guides.

Format List File – Special instructions

I	A function call is inlined
M	Nested loop is replaced by matrix-multiply routine
F	Fused multiply-add instruction
G	Vector gather memory operation
C	Vector scatter memory operation
V	One or more local arrays are assigned to the vector registers

Extensive lists of format lists symbols are available in the compiler user guides.

Format List File

The *.LL file gives information about the compilation and optimization. Generated with "-report-all"

```
NEC Fortran Compiler (2.4.20) for Vector Engine
Tue Oct 15 09:37:35 2019
FILE NAME : test.F90

COMPILER OPTIONS : -fpp -O3 -floop-interchange
                   -o test.x -report-all

  OPTIMIZATION PARAMETER :
    -On                                    : 3
    -fargument-alias                       : disable
    -fargument-noalias                     : enable
    -fassociative-math                     : enable
    -fassume-contiguous                    : disable
    -fcopyin-intent-out                    : enable
    -fcse-after-vectorization              : disable
    -ffast-formatted-io                    : enable
    -ffast-math                            : enable
    -fignore-asynchronous                  : disable
    -fignore-volatile                      : disable
    -fivdep                                : disable
    -floop-collapse                        : enable
    -floop-count                           : 5000
    -floop-fusion                          : enable
    -floop-interchange                     : enable
    -floop-normalize                       : enable
...
```

- Used compiler version

Format List File

The *.LL file gives information about the compilation and optimization.
Generated with "-report-all"

```
NEC Fortran Compiler (2.4.20) for Vector Engine
Tue Oct 15 09:37:35 2019
FILE NAME : test.F90

COMPILER OPTIONS : -fpp -O3 -floop-interchange
                   -o test.x -report-all

  OPTIMIZATION PARAMETER :
    -On                                      : 3
    -fargument-alias                         : disable
    -fargument-noalias                       : enable
    -fassociative-math                       : enable
    -fassume-contiguous                      : disable
    -fcopyin-intent-out                      : enable
    -fcse-after-vectorization                : disable
    -ffast-formatted-io                      : enable
    -ffast-math                              : enable
    -fignore-asynchronous                    : disable
    -fignore-volatile                        : disable
    -fivdep                                  : disable
    -floop-collapse                          : enable
    -floop-count                             : 5000
    -floop-fusion                            : enable
    -floop-interchange                       : enable
    -floop-normalize                         : enable
...
```

- Used compiler version
- Compile time

Format List File

The *.LL file gives information about the compilation and optimization. Generated with "-report-all"

```
NEC Fortran Compiler (2.4.20) for Vector Engine
Tue Oct 15 09:37:35 2019
FILE NAME : test.F90

COMPILER OPTIONS : -fpp -O3 -floop-interchange
                   -o test.x -report-all

  OPTIMIZATION PARAMETER :
    -On                                  : 3
    -fargument-alias                     : disable
    -fargument-noalias                   : enable
    -fassociative-math                   : enable
    -fassume-contiguous                  : disable
    -fcopyin-intent-out                  : enable
    -fcse-after-vectorization            : disable
    -ffast-formatted-io                  : enable
    -ffast-math                          : enable
    -fignore-asynchronous                : disable
    -fignore-volatile                    : disable
    -fivdep                              : disable
    -floop-collapse                      : enable
    -floop-count                         : 5000
    -floop-fusion                        : enable
    -floop-interchange                   : enable
    -floop-normalize                     : enable
...
```

- Used compiler version
- Compile time
- Compiled source file

Format List File

The *.LL file gives information about the compilation and optimization.
Generated with "-report-all"

```
NEC Fortran Compiler (2.4.20) for Vector Engine
Tue Oct 15 09:37:35 2019
FILE NAME : test.F90

COMPILER OPTIONS : -fpp -O3 -floop-interchange
                   -o test.x -report-all

  OPTIMIZATION PARAMETER :
    -On                                  : 3
    -fargument-alias                     : disable
    -fargument-noalias                   : enable
    -fassociative-math                   : enable
    -fassume-contiguous                  : disable
    -fcopyin-intent-out                  : enable
    -fcse-after-vectorization            : disable
    -ffast-formatted-io                  : enable
    -ffast-math                          : enable
    -fignore-asynchronous                : disable
    -fignore-volatile                    : disable
    -fivdep                              : disable
    -floop-collapse                      : enable
    -floop-count                         : 5000
    -floop-fusion                        : enable
    -floop-interchange                   : enable
    -floop-normalize                     : enable
...
```

- Used compiler version
- Compile time
- Compiled source file
- Used compiler options

Format List File

The *.LL file gives information about the compilation and optimization.
Generated with "-report-all"

```
NEC Fortran Compiler (2.4.20) for Vector Engine
Tue Oct 15 09:37:35 2019
FILE NAME : test.F90

COMPILER OPTIONS : -fpp -O3 -floop-interchange
                   -o test.x -report-all

  OPTIMIZATION PARAMETER :
    -On                                   : 3
    -fargument-alias                      : disable
    -fargument-noalias                    : enable
    -fassociative-math                    : enable
    -fassume-contiguous                   : disable
    -fcopyin-intent-out                   : enable
    -fcse-after-vectorization             : disable
    -ffast-formatted-io                   : enable
    -ffast-math                           : enable
    -fignore-asynchronous                 : disable
    -fignore-volatile                     : disable
    -fivdep                               : disable
    -floop-collapse                       : enable
    -floop-count                          : 5000
    -floop-fusion                         : enable
    -floop-interchange                    : enable
    -floop-normalize                      : enable
...
```

- Used compiler version
- Compile time
- Compiled source file
- Used compiler options
- Explicitly set values/options

Format List File

The *.LL file gives information about the compilation and optimization. Generated with "-report-all"

```
NEC Fortran Compiler (2.4.20) for Vector Engine
Tue Oct 15 09:37:35 2019
FILE NAME : test.F90

COMPILER OPTIONS : -fpp -O3 -floop-interchange
                   -o test.x -report-all

  OPTIMIZATION PARAMETER :
    -On                                  : 3
    -fargument-alias                     : disable
    -fargument-noalias                   : enable
    -fassociative-math                   : enable
    -fassume-contiguous                  : disable
    -fcopyin-intent-out                  : enable
    -fcse-after-vectorization            : disable
    -ffast-formatted-io                  : enable
    -ffast-math                          : enable
    -fignore-asynchronous                : disable
    -fignore-volatile                    : disable
    -fivdep                              : disable
    -floop-collapse                      : enable
    -floop-count                         : 5000
    -floop-fusion                        : enable
    -floop-interchange                   : enable
    -floop-normalize                     : enable
...
```

- Used compiler version
- Compile time
- Compiled source file
- Used compiler options
- Explicitly set values/options
- Implicitly set values/options (e.g. through -O3)
- Values/options set by default

Format List File

The *.LL file gives information about the compilation and optimization. Generated with “-report-all”

```
...
    -mvector-dependency-test                    : enable
    -mvector-fma                                : enable
    -mvector-intrinsic-check                    : disable
    -mvector-iteration                          : enable
    -mvector-iteration-unsafe                   : enable
    -mvector-loop-count-test                    : disable
    -mvector-merge-conditional                  : enable
    -mvector-packed                             : disable
    -mvector-reduction                          : enable
    -mvector-shortloop-reduction                : disable
    -mvector-threshold                          : 5
    -mwork-vector-kind=none                     : disable

 PROCEDURE NAME: TEST

    REPORT FROM: VECTORIZATION

    LOOP BEGIN: (test.F90:7)
      <Vectorized loop.>
      *** The number of VGT,   VSC.    :  0,  0.
(test.F90:7)
      *** The number of VLOAD, VSTORE. :  0,  0.
(test.F90:7)
      *** Idiom detected. : SUM (test.F90:8)
    LOOP END
```

- Used compiler version
- Compile time
- Compiled source file
- Used compiler options
- Explicitly set values/options
- Implicitly set values/options (e.g. through -O3)
- Values/options set by default
- Summary of loop transformations/vectorizations without source code (see *.L for that)

Exercise 04 – Simple Inhibitors

Vectorization Techniques

Collapsing – Increasing the Vector Length

Consider the following nested loop:

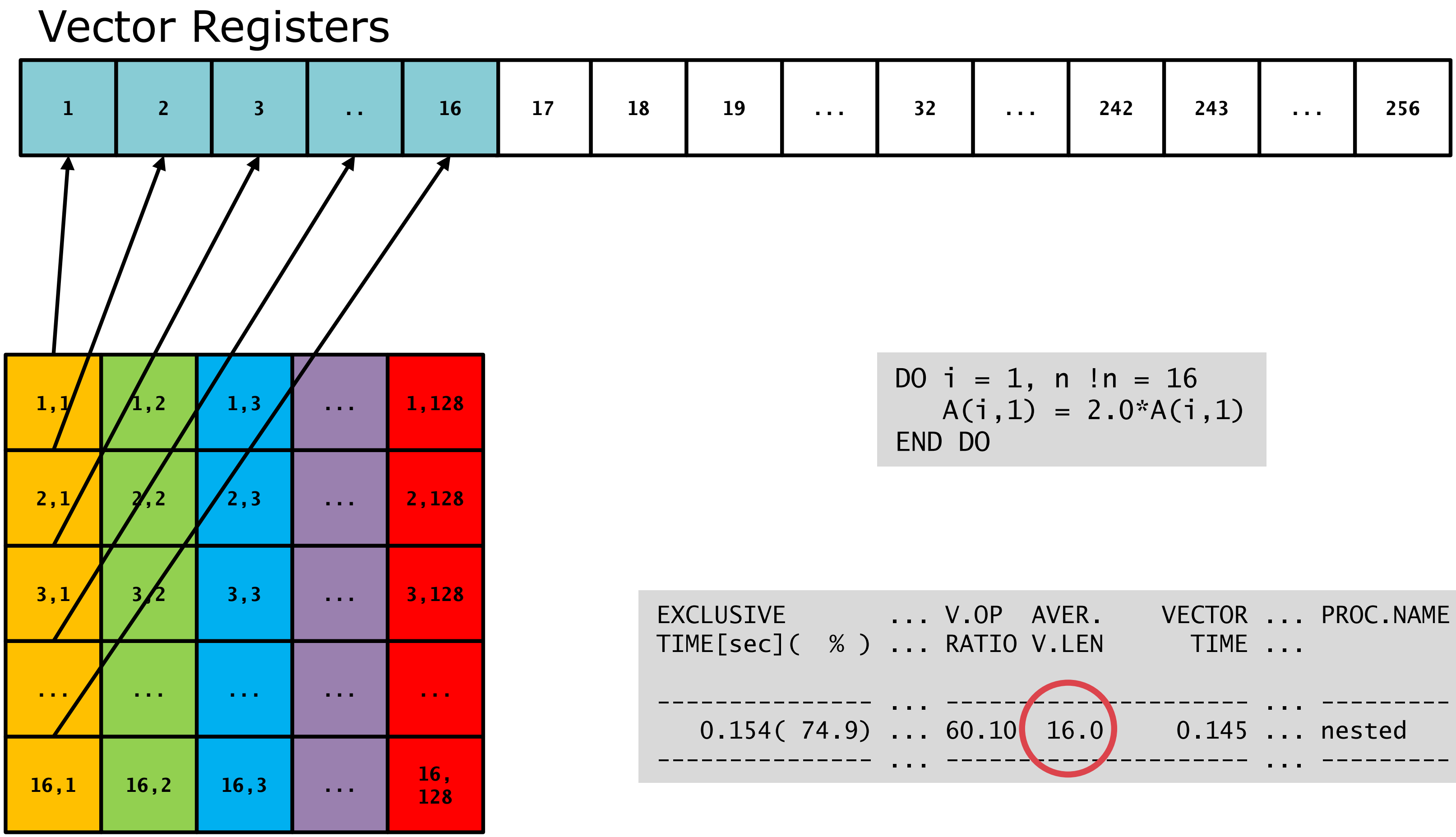
```
DO j = 1, m
   DO i = 1, n
      A(i,j) = 2.0*A(i,j)
   END DO
END DO
```

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`

Collapsing – Increasing the Vector Length

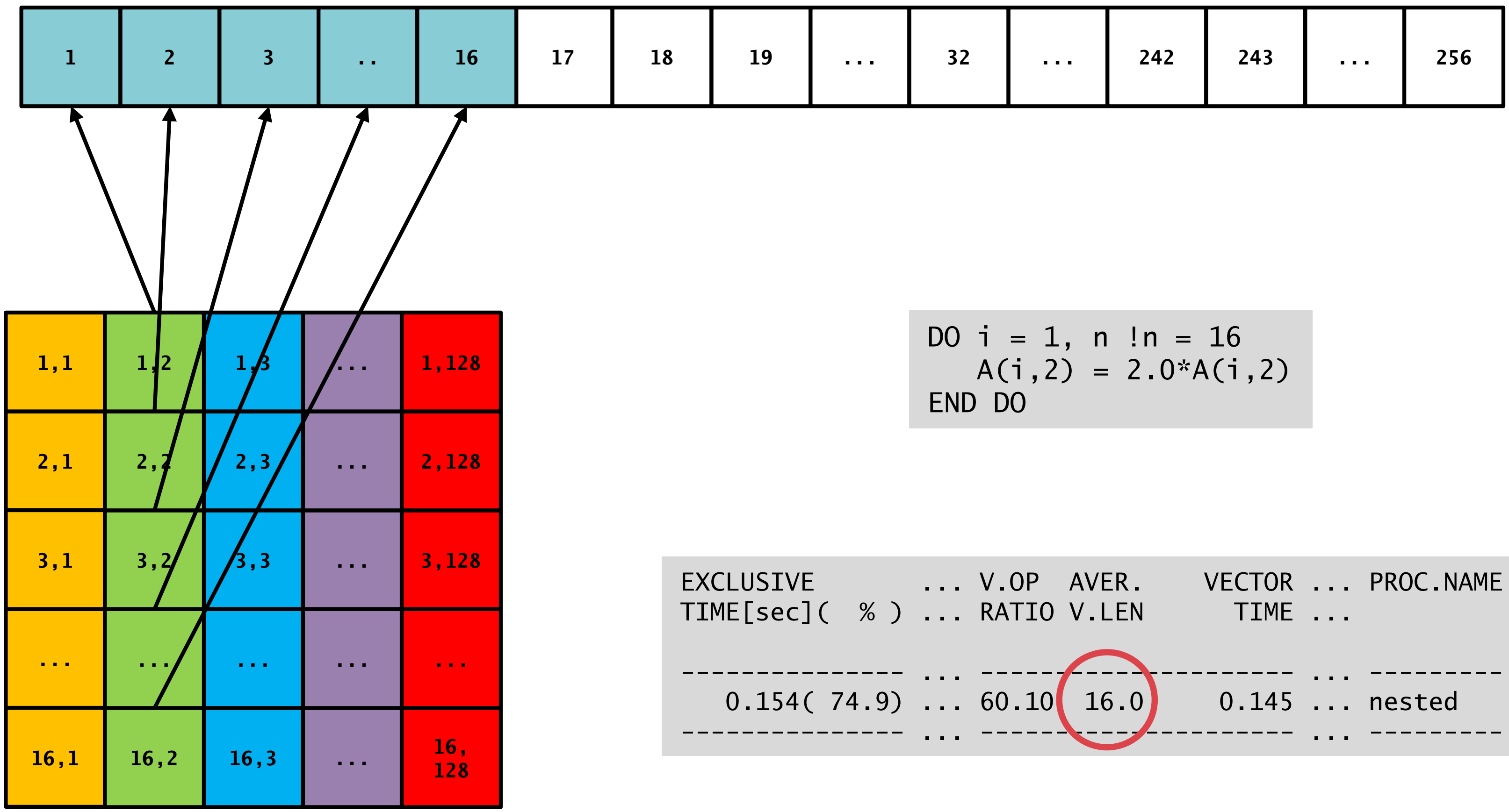



```
DO i = 1, n !n = 16
   A(i,1) = 2.0*A(i,1)
END DO
```

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

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

Collapsing – Increasing the Vector Length

Vector Registers




```
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
---------------  ... -------------------- ... ---------
```

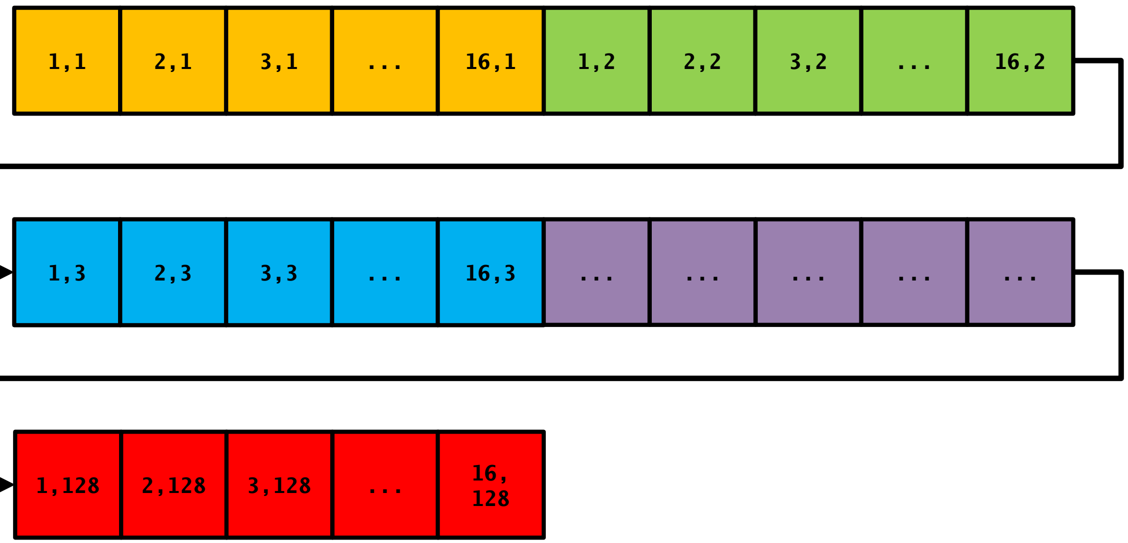
Collapsing – Increasing the Vector Length

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)`!

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
```

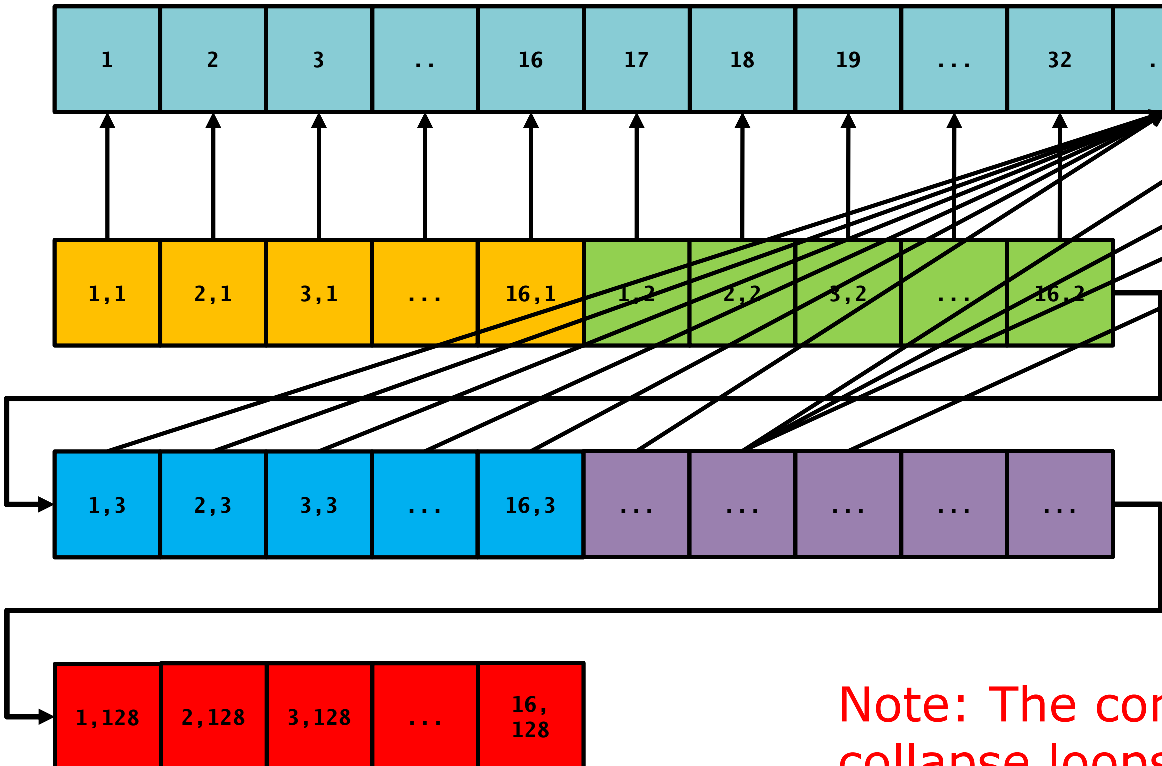
Innermost (only) loop is vectorized:

```
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

Collapsing – Increasing the Vector Length

Vector Registers



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
---------------  ... --------------------- ... ---------
```

Exercise 05 – Collapsing

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

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
```

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



      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
```

Loop pushing

1. Promote variables to arrays

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



      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
```

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,ij,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



      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
```

Loop pushing

3. Push loops into subroutine

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

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
```

Loop pushing

4. Change loop order

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

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
```

Exercise 06 – Loop Pushing

Index Lists

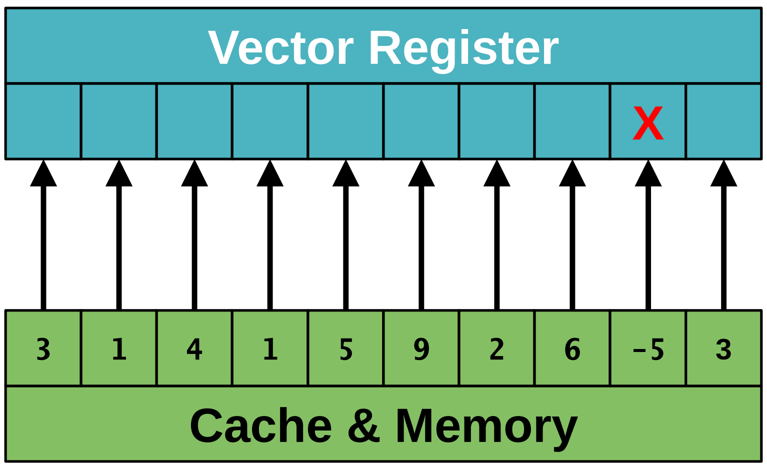
Consider the following loop with a condition:

```
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.

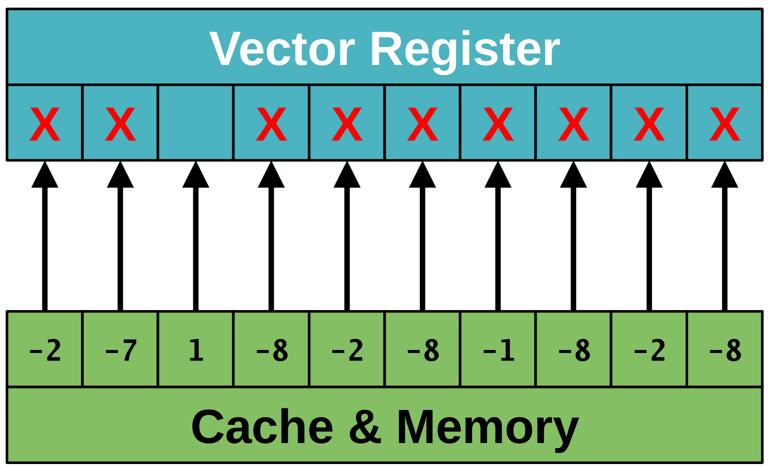
Condition is almost always true

B(i):



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

Condition is almost never true



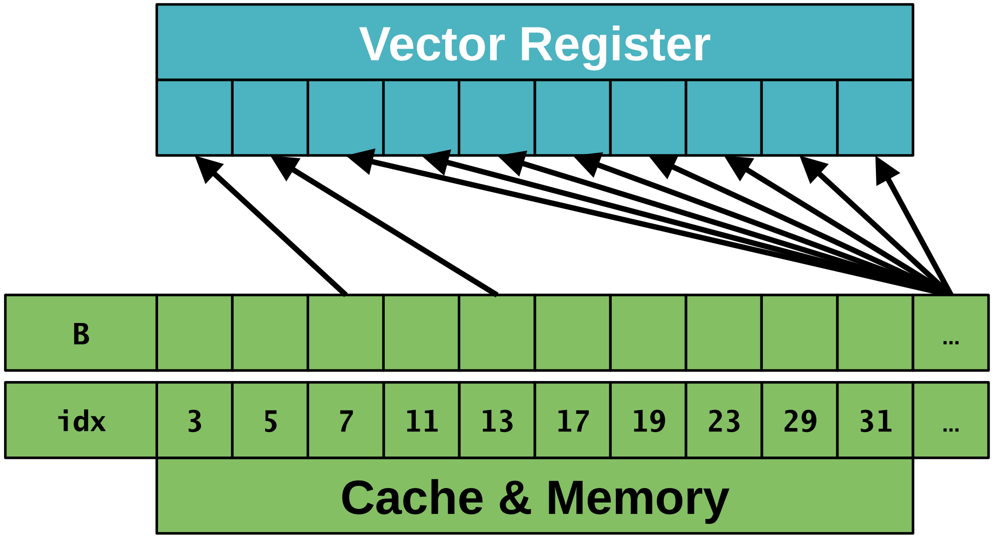
Wasting a lot of loads and computations → Significant decrease in performance

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

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
```

```
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
```

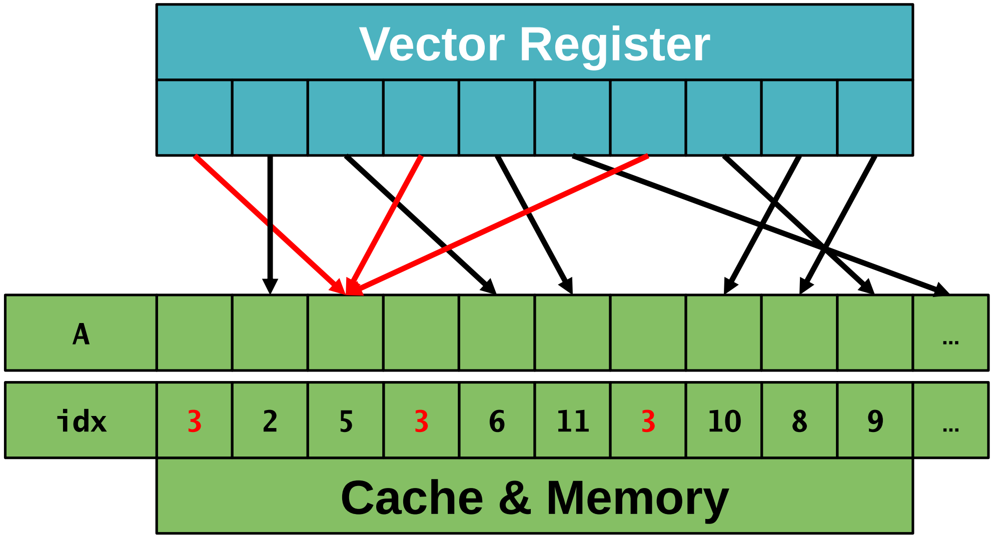
- Note: Building and using index lists is expensive.
- More complex load behavior Gather/Scatter patterns.
- Potentially more bank conflicts.
- They should only be used if everything else fails.

Index Lists

An index list often 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)
```



- Executed in serial is fine.
- Executed simultaneously leads to an undefined result in A(3).
- Scatter only is fine.
- Gather only is fine.
- Gather + scatter can lead to problems.

\Orchestrating a brighter world NEC

Exercise 07 – Index Lists

Exercise 08 – Index Lists II

\Orchestrating a brighter world NEC

Exercise 09 – Index Lists III

Special Loop Structures

While Loops – Boolean Masks

“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. Utilizing a Boolean mask or an index list are two techniques to rewrite the 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
```

While Loops – Boolean Masks

Goal: Generate vectorizable innermost loop using Boolean mask

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

   END DO
END DO
```

While Loops – Boolean Masks

1. Push while loop outside
2. replace “while-condition” with “if-condition”
3. Pull operation out of “if-condition”

```
DO WHILE (there is still work)

   DO i = 1, n
      B(i) = B(i) / A(i)
      IF (B(i) > C(i)) THEN

      END IF
   END DO
END DO
```

While Loops – Boolean Masks

4. Introduce counter on how much work is to be done

```
INTEGER :: todo
todo = n

DO WHILE (todo > 0)
   todo = 0
   DO i = 1, n
      B(i) = B(i) / A(i)
      IF (B(i) > C(i)) THEN
         todo = todo + 1
      END IF
   END DO
END DO
```

While Loops – Boolean Masks

The inner most loop is now vectorizable.
This solution works best if a similar amount of iterations is expected for every `i`.

```
17: err( 504): The number of VLOAD, VSTORE.:  3,  1.
17: err( 505): The number of VGT,   VSC.   :  0,  0.
17: vec( 101): Vectorized loop.

13:              todo = n
14:
15: +------->  DO WHILE (todo > 0)
16: |             todo = 0
17: |V----->      DO i = 1, n
18: ||               B(i) = B(i) / A(i)
19: ||               IF (B(i) > C(i)) THEN
20: ||                  todo = todo + 1
21: ||               END IF
22: |V-----       END DO
23: +------     END DO
```

While Loops – Index List

Goal: Generate vectorizable innermost loop using index list.

```
DO i = 1, n

   DO WHILE (B(i) > C(i)) THEN
      B(i) = B(i) / A(i)


   END DO
END DO
```

While Loops – Index List

Setup initial index list

Usage and update of index list

1. Pull out "while loop", replace by if
2. pull operation out of the if

```
DO WHILE (index list has entries)



   DO i = 1, n

      B(i) = B(i) / A(i)
      IF (B(i) > C(i)) THEN


      END IF
   END DO

END DO
```

While Loops – Index List

Setup initial index list

3. Build initial index list

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

maxidx = 0
DO i = 1, n
   IF (B(i) > C(i)) THEN
      maxidx = maxidx + 1
      idx_list(maxidx) = i
   END IF
END DO
```

Usage and update of index list

```
DO WHILE (index list has entries)



   DO i = 1, n

      B(i) = B(i) / A(i)
      IF (B(i) > C(i)) THEN


      END IF
   END DO

END DO
```

While Loops – Index List

Setup initial index list

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

maxidx = 0
DO i = 1, n
   IF (B(i) > C(i)) THEN
      maxidx = maxidx + 1
      idx_list(maxidx) = i
   END IF
END DO
```

Usage and update of index list

4. Replace do loop with index loop

```
DO WHILE (maxidx > 0)

   !NEC$ ivdep

   DO idx = 1, maxidx
      i = idx_list(idx)
      B(i) = B(i) / A(i)
      IF (B(i) > C(i)) THEN


      END IF
   END DO

END DO
```

While Loops – Index List

Setup initial index list

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

maxidx = 0
DO i = 1, n
   IF (B(i) > C(i)) THEN
      maxidx = maxidx + 1
      idx_list(maxidx) = i
   END IF
END DO
```

Usage and update of index list

5. Update index list and maxidx

```
DO WHILE (maxidx > 0)
   maxidxnew = 0
   !NEC$ ivdep
   !NEC$ loop_count_test
   DO idx = 1, maxidx
      i = idx_list(idx)
      B(i) = B(i) / A(i)
      IF (B(i) > C(i)) THEN
         maxidxnew = maxidxnew + 1
         idx_list(maxidxnew) = i
      END IF
   END DO
   maxidx = maxidxnew
END DO
```

Note that the index list can only get shorter, thus overwriting it does not destroy needed information

While Loops – Index List

```
17:              maxidx = 0
18: V------>     DO i = 1, n
19: |               IF (B(i) > C(i)) THEN
20: |                  maxidx = maxidx + 1
21: |                  idx_list(maxidx) = i
22: |               END IF
23: V------      END DO
14:
25: +------>     DO WHILE (maxidx > 0)
26: |               maxidxnew = 0
27: |               !NEC$ ivdep
28: |               !NEC$ loop_count_test
29: |V----->        DO idx = 1, maxidx
30: ||                 i = idx_list(idx)
31: ||      G          B(i) = B(i) / A(i)
32: ||      G          IF (B(i) > C(i)) THEN
33: ||                    maxidxnew = maxidxnew + 1
34: ||                    idx_list(maxidxnew) = i
35: ||                 END IF
36: |V-----         END DO
37: |               maxidx = maxidxnew
38: +------      END DO
```

Exercise 10 – While Loop

Inner K-Loop

- The innermost loop (here with index k) depends on the outer loop indices i and j.
- It is possible to vectorize the k-loop.
- Depending on L(i,j) the average vector length can turn out to be disadvantageous
- Three approaches to optimize this:
  1. Boolean mask (good if all k-loops are of similar length)
  2. Static index list (good if k-loop lengths vary a lot, but memory intensive)
  3. Dynamic index list (good if k-loop lengths vary a lot, more index list creation over head)

```
+------>   DO j = 1, n
|+----->      DO i = 1, m
||V---->         DO k = 1, L(i,j)
|||                 B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
||V----          END DO
|+-----       END DO
+------    END DO
```

Inner K-Loop – Boolean Mask

Goal: Generate vectorizable innermost loop using Boolean mask

```
DO j = 1, n








   DO i = 1, m
      DO k = 1, L(i,j)

            B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)

      END DO
   END DO
END DO
```

Inner K-Loop – Boolean Mask

1. Exchange i and k-loop and change loop length of k
2. Introduce if-construct (ensuring functionality of k-loop

```
DO j = 1, n

   maxk = MAXVAL(L(:,j))






   DO k =      1, maxk
      DO i = 1, m
         IF (k <= L(i,j)) THEN
            B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
         END IF
      END DO
   END DO
END DO
```

Inner K-Loop – Boolean Mask

3. Split k-loop into a part that works on all points and a part where a condition is needed to increase performance

```
DO j = 1, n
   mink = MINVAL(L(:,j))
   maxk = MAXVAL(L(:,j))
   DO k = 1, mink
      DO i = 1, m
         B(i,j) = B(i,j) + B(i,j) / A(i,j,k)
      END DO
   END DO
   DO k = mink+1, maxk
      DO i = 1, m
         IF (k <= L(i,j)) THEN
            B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
         END IF
      END DO
   END DO
END DO
```

Inner K-Loop – Boolean Mask

The innermost loop is now vectorizable

```
+------>  DO j = 1, n
|V====>      mink = MINVAL(L(:,j))
|V====>      maxk = MAXVAL(L(:,j))
|+----->     DO k = 1, mink
||V---->        DO i = 1, m
|||                B(i,j) = B(i,j) + B(i,j) / A(i,j,k)
||V----         END DO
|+-----      END DO
|+----->     DO k = mink+1, maxk
||V---->        DO i = 1, m
|||                IF (k <= L(i,j)) THEN
|||                   B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
|||                END IF
||V----         END DO
|+-----      END DO
+------   END DO
```

Note that this is only efficient if all k-loops have similar length!

Inner K-Loop – Static 2D-Index List

Goal: Generate vectorizable innermost loop using index list

```
DO j = 1, n







   DO i = 1, m

      DO k = 1, L(i,j)

         B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
      END DO
   END DO
END DO
```

Inner K-Loop – Static 2D-Index List

1. Build an index list in i for every k

```
INTEGER :: maxk, max_idx(MAXVAL(L)), max_idx_tmp
INTEGER :: idx_lst(m,MAXVAL(L))

maxk = MAXVAL(L)
DO j = 1, n
   DO k = 1, maxk
      max_idx_tmp = 0
      DO i = 1, m
         IF (k <= L(i,j)) THEN
            max_idx_tmp = max_idx_tmp + 1
            idx_lst(max_idx_tmp,k) = i
         END IF
      END DO
      max_idx(k) = max_idx_tmp
   END DO
   DO i = 1, m

      DO k = 1, L(i,j)

         B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
      END DO
   END DO
END DO
```

Inner K-Loop – Static 2D-Index List

2. Exchange order of i and k-loop

```
INTEGER :: maxk, max_idx(MAXVAL(L)), max_idx_tmp
INTEGER :: idx_lst(m,MAXVAL(L))

maxk = MAXVAL(L)
DO j = 1, n
   DO k = 1, maxk
      max_idx_tmp = 0
      DO i = 1, m
         IF (k <= L(i,j)) THEN
            max_idx_tmp = max_idx_tmp + 1
            idx_lst(max_idx_tmp,k) = i
         END IF
      END DO
      max_idx(k) = max_idx_tmp
   END DO
   DO k = 1, maxk

      DO i = 1, m

         B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
      END DO
   END DO
END DO
```

Inner K-Loop – Static 2D-Index List

3. Apply index list

```
INTEGER :: maxk, max_idx(MAXVAL(L)), max_idx_tmp
INTEGER :: idx_lst(m,MAXVAL(L)), idx

maxk = MAXVAL(L)
DO j = 1, n
   DO k = 1, maxk
      max_idx_tmp = 0
      DO i = 1, m
         IF (k <= L(i,j)) THEN
            max_idx_tmp = maxidx_tmp + 1
            idx_lst(max_idx_tmp,k) = i
         END IF
      END DO
      max_idx(k) = max_idx_tmp
   END DO
   DO k = 1, maxk
      !NEC$ ivdep
      DO idx = 1, max_idx(k)
         i = idx_lst(idx, k)
         B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
      END DO
   END DO
END DO
```

Inner K-Loop – Static Index List

The innermost loop is now vectorizable

```
+------>  DO j = 1, n
|+----->     DO k = 1, maxk
||              max_idx_tmp = 0
||V---->        DO i = 1, m
|||                IF (k <= L(i,j)) THEN
|||                   max_idx_tmp = max_idx_tmp + 1
|||                   i_lst(max_idx_tmp,k) = i
|||                END IF
||V----         END DO
||              max_idx(k) = max_idx_tmp
|+-----      END DO
|+----->     DO k = 1, maxk
||              !NEC$ ivdep
||V---->        DO idx = 1, max_idx(k)
|||                i = i_lst(idx, k)
|||     G          B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
||V----         END DO
|+-----      END DO
+------   END DO
```

Note that this is best used if the k-loops have strongly varying length!

Note that the performance can be increased by separating the part, where work is done for every k (1…mink).

Inner K-Loop – Dynamic Index List

Goal: Generate vectorizable innermost loop using index list

```
DO j = 1, n





   DO i = 1, m


      DO k = 1, L(i,j)

         B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)


      END DO

   END DO
END DO
```

Inner K-Loop – Dynamic Index List

1. Initialize index list for k=1

```
INTEGER :: maxidx, idx_lst(m), idx
k = 1
DO j = 1, n
   maxidx = 0
   DO i = 1, m
      IF (k <= L(i,j)) THEN
         maxidx = maxidx + 1
         idx_lst(maxidx) = i
      END IF
   END DO
   DO i = 1, m



      DO k = 1, L(i,j)


         B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)



      END DO

   END DO
END DO
```

Inner K-Loop – Dynamic Index List

2. Replace i and k-loop by while loop with index loop

```
INTEGER :: maxidx, idx_lst(m), idx
k = 1
DO j = 1, n
   maxidx = 0
   DO i = 1, m
      IF (k <= L(i,j)) THEN
         maxidx = maxidx + 1
         idx_lst(maxidx) = i
      END IF
   END DO
   DO WHILE (maxidx > 0)


      !NEC$ ivdep
      DO idx = 1, maxidx
         i = idx_lst(idx)

         B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)



      END DO

   END DO
END DO
```

Inner K-Loop – Dynamic Index List

3. Conditionally update index list

```
INTEGER :: maxidx, idx_lst(m), idx, nmaxidx
k = 1
DO j = 1, n
   maxidx = 0
   DO i = 1, m
      IF (k <= L(i,j)) THEN
         maxidx = maxidx + 1
         idx_lst(maxidx) = i
      END IF
   END DO
   DO WHILE (maxidx > 0)
      nmaxidx = 0
      k = k + 1
      !NEC$ ivdep
      DO idx = 1, maxidx
         i = idx_lst(idx)
         IF (k <= L(i,j)) THEN
            B(i,j) = B(i,j) + 1.0d0 / A(i,j,k)
            nmaxidx = nmaxidx + 1
            idx_lst(nmaxidx) = i
         END IF
      END DO
      maxidx = nmaxidx
   END DO
END DO
```

Inner K-Loop – Dynamic Index List

The innermost loop is now better vectorizable

```
               k = 1
    V======>   maxk = MAXVAL(L)
    +------>   DO j = 1, n
    |             maxidx = 0
    |V----->      DO i = 1, m
    ||               IF (1 <= L(i,j)) THEN
    ||                  maxidx = maxidx + 1
    ||                  idx_lst(maxidx) = i
    ||               END IF
    |V-----       END DO
    |+----->      DO WHILE (maxidx > 0)
    ||               nmaxidx = 0
    ||               k = k + 1
    ||               !NEC$ ivdep
    ||V---->         DO idx = 1, maxidx
    |||                 i = idx_lst(idx)
    |||     G           IF (k <= L(i,j)) THEN
    |||     G              B(i,j) = B(i,j)+1.0d0/A(i,j,k)
    |||                    nmaxidx = nmaxidx + 1
    |||                    idx_lst(nmaxidx) = i
    |||                 END IF
    ||V----          END DO
    ||               maxidx = nmaxidx
    |+-----       END DO
    +------    END DO
```

Note that this is only efficient because the index list creation can be “hidden” behind the computation by the compiler.

Exercise 11 – Inner K-Loop

Search Loops

There are two types of search loops

1. Extensive searches, where every element has to be read at least once, in order to find the smallest/largest/… element.
2. Restricted searches, where only elements are read and compared until a first match is found (Used here as an example).

```
DO i = 1, m
   DO k = 1, o
      IF (A(k,i) < L(k)) THEN
         k_s = k
         EXIT
      END IF
   END DO
   B(i) = B(i) * A(k_s,i)
END DO
```

```
 8: S------>  DO i = 1, m
 9: |V----->     DO k = 1, o
10: ||              IF (A(k,i,j) < L(k)) THEN
11: ||                 k_s = k
12: ||                 EXIT
13: ||              END IF
14: |V-----      END DO
15: |       G    B(i,j) = B(i,j) * A(k_s,i,j)
16: S------   END DO
```

- Compiler detects search pattern and introduces a vectorized version that is still inefficient due to the short vector length.
- Two approaches (Boolean mask and index lists) to better vectorize searches.

Search Loops – Boolean Mask

Goal: Generate vectorizable innermost loop using Boolean mask

```
DO i = 1, m
   DO k = 1, o

         IF (A(k,i) < L(k)) THEN
            k_s    = k
            EXIT
         END IF

   END DO


   B(i) = B(i) * A(k_s   ,i)
END DO
```

Search Loops – Boolean Mask

1. Promote k_s and separate search from computation

```
DO i = 1, m
   DO k = 1, o

         IF (A(k,i) < L(k)) THEN
            k_s(i) = k
            EXIT
         END IF

   END DO
END DO
DO i = 1, m
   B(i) = B(i) * A(k_s(i),i)
END DO
```

Search Loops – Boolean Mask

2. Eliminate the EXIT statement and replace it by another condition

```
k_s(:) = 0
DO i = 1, m
   DO k = 1, o
      IF (k_s(i) == 0) THEN
         IF (A(k,i) < L(k)) THEN
            k_s(i) = k
            EXIT
         END IF
      END IF
   END DO
END DO
DO i = 1, m
   B(i) = B(i) * A(k_s(i),i)
END DO
```

Search Loops – Boolean Mask

3. Exchange i and k loop

```
k_s(:) = 0
DO k = 1, o
   DO i = 1, m
      IF (k_s(i) == 0) THEN
         IF (A(k,i) < L(k)) THEN
            k_s(i) = k

         END IF
      END IF
   END DO
END DO
DO i = 1, m
   B(i) = B(i) * A(k_s(i),i)
END DO
```

Search Loops – Boolean Mask

```
V=====>   k_s(:) = 0
+------>  DO k = 1, o
|V----->     DO i = 1, m
||              IF (k_s(i) == 0) THEN
||                 IF (A(k,i) < L(k)) THEN
||                    k_s(i) = k
||
||                 END IF
||              END IF
|V-----      END DO
+------   END DO
V------>  DO i = 1, m
|       G    B(i) = B(i) * A(k_s(i),i)
V------   END DO
```

Note that this works best if all searches are expected to be of similar length

Search Loops – Index List

Goal: Generate vectorizable innermost loop using index list

```
DO i = 1, m

   DO k = 1, o

      IF (A(k,i) < L(k)) THEN
         k_s    = k
         EXIT


      END IF
   END DO



   B(i) = B(i) * A(k_s   ,i)
END DO
```

Search Loops – Index List

1. Promote k_s
2. Initialize index list to every element

```
DO i = 1, m
   k_s(i) = 0
   idx_lst(i) = i
END DO

DO i = 1, m

   DO k = 1, o

      IF (A(k,i) < L(k)) THEN
         k_s(i) = k
         EXIT


      END IF
   END DO




   B(i) = B(i) * A(k_s(i),i)
END DO
```

Search Loops – Index List

3. Separate search and computation

```
DO i = 1, m
   k_s(i) = 0
   idx_lst(i) = i
END DO

DO i = 1, m

   DO k = 1, o

      IF (A(k,i) < L(k)) THEN
         k_s(i) = k
         EXIT


      END IF
   END DO

END DO
DO i = 1, m
   B(i) = B(i) * A(k_s(i),i)
END DO
```

Search Loops – Index List

4. Apply and update index list

```
DO i = 1, m
   k_s(i) = 0
   idx_lst(i) = i
END DO
maxidx = m
DO k = 1, o
   nmaxidx = 0
   DO idx = 1, maxidx
      i = idx_lst(idx)
      IF (A(k,i) < L(k)) THEN
         k_s(i) = k
      ELSE
         nmaxidx = nmaxidx + 1
         idx_lst(nmaxidx) = i
      END IF
   END DO
   maxidx = nmaxidx
END DO
DO i = 1, m
   B(i) = B(i) * A(k_s(i),i)
END DO
```

Search Loops – Index List

```
V------>  DO i = 1, m
|            k_s(i) = 0
|            idx_lst(i) = i
V------   END DO
          maxidx = m
+------>  DO k = 1, o
|            nmaxidx = 0
|V----->     DO idx = 1, maxidx
||              i = idx_lst(idx)
||              IF (A(k,i) < L(k)) THEN
||                 k_s(i) = k
||              ELSE
||                 nmaxidx = nmaxidx + 1
||                 idx_lst(nmaxidx) = i
||              END IF
|V-----      END DO
|            maxidx = nmaxidx
+------   END DO
V------>  DO i = 1, m
|       G    B(i) = B(i) * A(k_s(i),i)
V------   END DO
```

Note that this approach works best if the searches are expected to have strongly varying length

Exercise 12 – Search Loops

Data Reusage

Load/Store Optimizations

Loop Combination

Consider the following successive loops

```
11: vec( 101): Vectorized loop.
11: err( 504): The number of VLOAD, VSTORE.:  2,  1.
11: err( 505): The number of VGT,   VSC.   :  0,  0.
12: vec( 128): Fused multiply-add operation applied.
17: vec( 101): Vectorized loop.
17: err( 504): The number of VLOAD, VSTORE.:  2,  1.
17: err( 505): The number of VGT,   VSC.   :  0,  0.
 ...
10: +------>  DO j = 1, 1024
11: |V----->     DO i = 2, 2048
12: ||      F       B(i,j) = B(i,j) + A(i,j)**2
13: |V-----      END DO
14: +------   END DO
15:
16: +------>  DO j = 2, 1024
17: |V----->     DO i = 2, 2048
18: ||              B(i,j) = B(i,j) * A(i,j)
19: |V-----      END DO
20: +------   END DO
```

- The diagnostics list shows two loads (A(i,j), B(i,j)) and one store (B(i,j)) for both loops.
- The loads and stores are for the same variables in both loops.
- Combining the loops can save a lot of loads thus increasing performance.
- For easy cases the compiler can and will combine loops.

Loop Combination

Split of iterations only done by one loop.
Combine operations for the rest.

```
11: vec( 101): Vectorized loop.
11: err( 504): The number of VLOAD, VSTORE.:  2,  1.
11: err( 505): The number of VGT,   VSC.   :  0,  0.
12: vec( 128): Fused multiply-add operation applied.
16: vec( 101): Vectorized loop.
16: err( 504): The number of VLOAD, VSTORE.:  2,  1.
16: err( 505): The number of VGT,   VSC.   :  0,  0.
17: vec( 128): Fused multiply-add operation applied.
...
10:           j = 1
11: V------>  DO i = 2, 2048
12: |       F    B(i,j) = B(i,j) + A(i,j)**2
13: V------   END DO
14:
15: +------>  DO j = 2, 1024
16: |V----->     DO i = 2, 2048
17: ||      F       B(i,j) = B(i,j) + A(i,j)**2
18: ||              B(i,j) = B(i,j) * A(i,j)
19: |V-----      END DO
20: +------   END DO
```

- The diagnostics list still shows two loads (A(i,j), B(i,j)) and one store (B(i,j)) for both loops.
- The first loop is only for a corner case and negligible.
- All important work is done in the second loop where the length did not change .
- The total number of loads and stores is nearly cut in half by simple rearrangement of the loops.

\Orchestrating a brighter world NEC

Exercise 13 – Loop Combination

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.

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

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

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)

Exercise 14 – Loop Unrolling

Vector Registers

- Vector registers can be used in order to eliminate load and store operations. Like a perfect loop unroll of length of the hardware vector length.

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

- `C(i)` and `A(i)` are needed again in the next iteration of j, thus need to be loaded again.
- Dividing the i loop in chunks such that `C(i)` and `A(i)` can stay loaded would increase performance.
- The !NEC$ vreg(varname) directive can be used to assign local arrays directly to the vector registers to eliminate additional loads. (No allocation in memory is done)
- The vector registers on the Aurora have a length of 256.

Vector Registers

Goal: Divide the i loop into chunks of 256 (stripmining) and store `A(i)` and `C(i)` in a vector register arrays.
For the sake of simplicity only the loads of A are optimized for now.

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

Vector Registers

1. Imagine the i loop was only 256 iterations long (corrected later)
2. Create a local work array of length 256 and copy `A(i)` in

```
REAL(8) :: vrega(256)




   DO i = 1, 256
      vrega(i      ) = A(i)
   END DO
   DO j = 1, m
      DO i = 1, 256
         C(i) = 0.1*C(i) + vrega(i      )
      END DO
   END DO
```

Vector Registers

3. Assign the local array to the vector registers using the directive.
   If the array A was only 256 elements long this would be the result for A

```
REAL(8) :: vrega(256)
!NEC$ vreg(vrega)
!This directive must be
!after the declaration block

   DO i = 1, 256
      vrega(i     ) = A(i)
   END DO
   DO j = 1, m
      DO i = 1, 256
         C(i) = 0.1*C(i) + vrega(i     )
      END DO
   END DO
```

Vector Registers

4. Create an outer loop for iterating over chunks of length 256 `is` is the start and `ie` is the end index of the chunk.

```
REAL(8) :: vrega(256)
!NEC$ vreg(vrega)

DO is = 1, n, 256
   ie = MIN(is-1+256, n)
   DO i = is, ie
      vrega(i-is+1) = A(i)
   END DO
   DO j = 1, m
      DO i = 1, 256
         C(i) = 0.1*C(i) + vrega(i-is+1)
      END DO
   END DO
END DO
```

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-----      END DO
20: |+----->     DO j = 1, m
21: ||V---->        DO i = is, ie
22: |||     V          C(i) = 0.1*C(i) + vrega(i-is+1)
23: ||V----         END DO
24: |+-----      END DO
25: +------   END DO
```

V (lines 18 and 22) shows that the compiler utilizes vector registers to store our local array.

Vector Registers

5. Create a local array for C
and assign it to the 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




END DO
```

Vector Registers

6. Copy the vector register back to array C

```
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
```

```
+------>  DO is = 1, n, 256
|            ie = MIN(is-1+256, n)
|V----->     DO i = is, ie
||      V       vrega(i-is+1) = A(i)
||      V       vregc(i-is+1) = C(i)
|V-----      END DO
|+----->     DO j = 1, m
||V---->        DO i = is, ie
|||     V          vregc(i-is+1) = 0.1*vregc(i-is+1)
                                     + vrega(i-is+1)
||V----         END DO
|+-----      END DO
|V----->     DO i = is, ie
||      V       C(i) = vregc(i-is+1)
|V-----      END DO
+------   END DO
```

Vector Registers

```
DO j = 2, n-1
  DO i = 2, n-1
    C(i,j)=A(i,j)-A(i,j-1)+B(i,j)-B(i,j+1)
  END DO
END DO
```

Vector Registers

```
DO j = 2, n-1

   DO i = 2, n-1
      C(i,j)=A(i,j)-A    (i  ,j-1)+B    (i    ,j)-B(i,j+1)


   END DO
END DO
```

Vector Registers

1. Stripmine the innermost loop

```
DO is = 2, n-1, 256
   ie = is, min(is-1+256,n-1)




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


      END DO
   END DO
END DO
```

Vector Registers

2. Introduce two local arrays of length 256

```
REAL, DIMENSION(256)::vrega,vregb


DO is = 2, n-1, 256
   ie = is, min(is-1+256,n-1)
   !NEC$ shortloop
   DO i = is, ie
      vrega(i+1-ii)=a(i,1)
      vregb(i+1-ii)=b(i,2)
   END DO
   DO j = 2, n-1
      !NEC$ ivdep
      DO i = is, ie
         C(i,j)=A(i,j)-vrega(i+1-is)+vregb(i+1-is)-B(i,j+1)
         vrega(i+1-ii)=A(i,j)
         vregb(i+1-ii)=B(i,j+1)
      END DO
   END DO
END DO
```

Vector Registers

3. Assign the local arrays to vector registers

```
REAL, DIMENSION(256)::vrega,vregb
!NEC$ vreg(vrega)
!NEC$ vreg(vregb)
DO is = 2, n-1, 256
   ie = is, min(is-1+256,n-1)
   !NEC$ shortloop
   DO i = is, ie
      vrega(i+1-ii)=a(i,1)
      vregb(i+1-ii)=b(i,2)
   END DO
   DO j = 2, n-1
      !NEC$ ivdep
      DO i = is, ie
         C(i,j)=A(i,j)-vrega(i+1-is)+vregb(i+1-is)-B(i,j+1)
         vrega(i+1-ii)=A(i,j)
         vregb(i+1-ii)=B(i,j+1)
      END DO
   END DO
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.

Exercise 15 – Vector Registers

I/O Operation Optimization

I/O Operations

- As could be seen in Exercise 03 – Simple Inhibitors, I/O operations obstruct automatic vectorization.

- To achieve maximum of performance write rarely and big chunks instead of often and small chunks.

- Try to separate computation and I/O.

- I/O operations are especially problematic for vector machines.
  The next exercise was timed on the Aurora 10B' with 304.14s and on an Intel Skylake 6148 with 2666 MHz with 8.72s
  The reference solution for the next exercise was timed with 0.33s on the Aurora and with 0.46s on Skylake.

Exercise 16 – Small Block IO

Conflicting Memory Access

Memory Bank Conflicts

Clock Cycle: 00

Stride = 1
Elements read = 0

Banks			
1	2	3	4
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)

Stride = 2
Elements read = 0

Banks			
1	2	3	4
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)

Stride = 3
Elements read = 0

Banks			
1	2	3	4
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)

Ready for access

Accessing

Blocked from access (Bank busy time = 2 cycles)

Memory Bank Conflicts

Clock Cycle: 01

Stride = 1
Elements read = 1

Banks			
1	2	3	4

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)

Stride = 2
Elements read = 1

Banks			
1	2	3	4

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)

Stride = 3
Elements read = 1

Banks			
1	2	3	4

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)

Ready for access

Accessing

Blocked from access (Bank busy time = 2 cycles)

Memory Bank Conflicts

Clock Cycle: 02

Stride = 1
Elements read = 2

Banks			
1	2	3	4
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)

Stride = 2
Elements read = 2

Banks			
1	2	3	4
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)

Stride = 3
Elements read = 2

Banks			
1	2	3	4
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)

Ready for access

Accessing

Blocked from access (Bank busy time = 2 cycles)

Clock Cycle: 03

Stride = 1
Elements read = 3

Banks			
1	2	3	4

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)

Stride = 2
Elements read = 2

Banks			
1	2	3	4

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)

Bank Conflict!

Stride = 3
Elements read = 3

Banks			
1	2	3	4

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)

Ready for access

Accessing

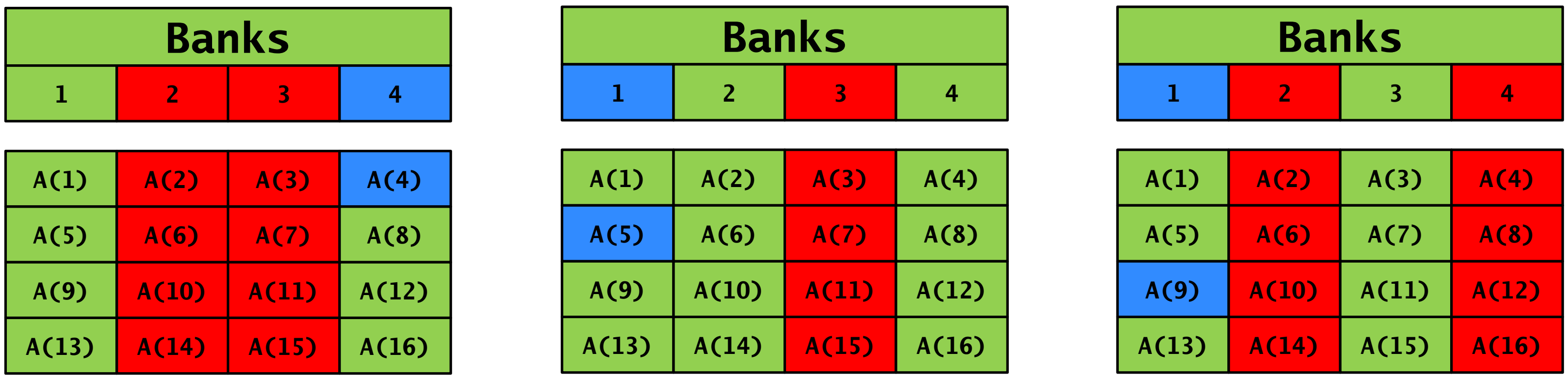
Blocked from access (Bank busy time = 2 cycles)

Clock Cycle: 04

Stride = 1
Elements read = 4

Stride = 2
Elements read = 3

Stride = 3
Elements read = 4



Ready for access

Accessing

Blocked from access (Bank busy time = 2 cycles)

Memory Bank Conflicts

Clock Cycle: 05

Stride = 1
Elements read = 5

Banks			
1	2	3	4
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)

Stride = 2
Elements read = 4

Banks			
1	2	3	4
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)

Stride = 3
Elements read = 5

Banks			
1	2	3	4
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)

Ready for access

Accessing

Blocked from access (Bank busy time = 2 cycles)

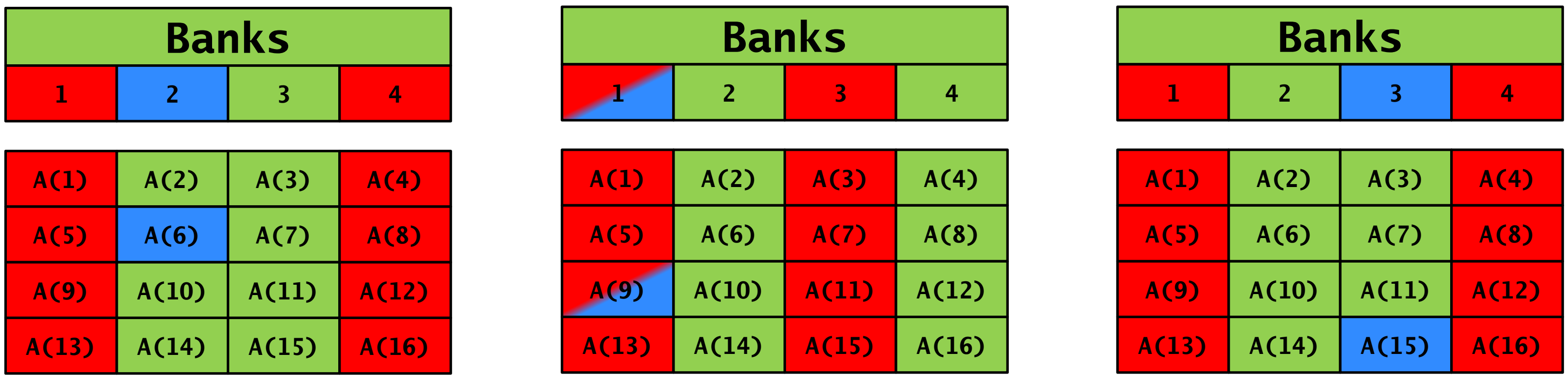
Memory Bank Conflicts

Clock Cycle: 06

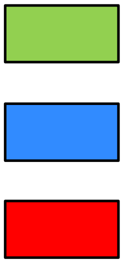
Stride = 1
Elements read = 6

Stride = 2
Elements read = 4

Stride = 3
Elements read = 6



Bank Conflict!



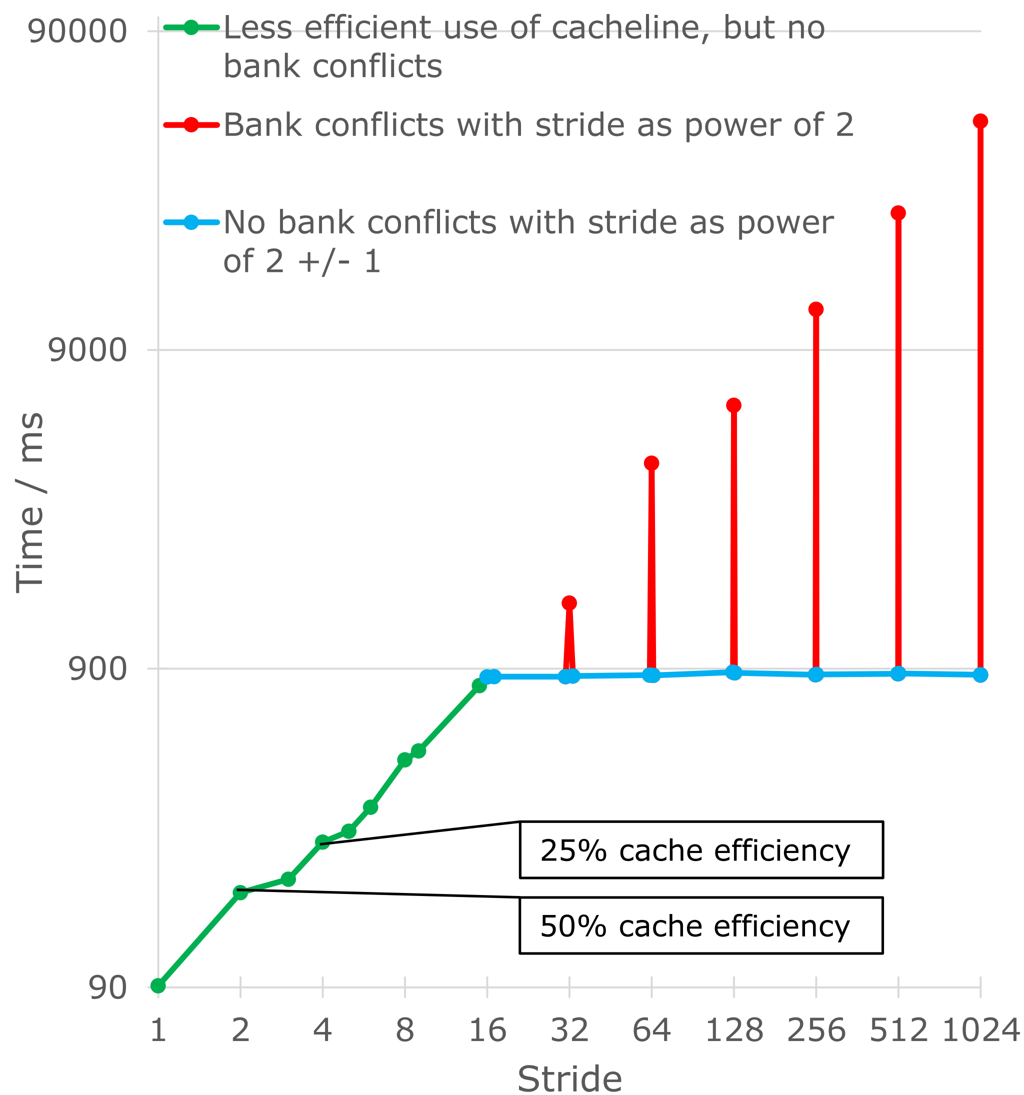
Ready for access

Accessing

Blocked from access (Bank busy time = 2 cycles)

Performance Decrease with Bank Conflicts

```
PROGRAM bankconflicts

   IMPLICIT NONE

   INTEGER(KIND=8), PARAMETER :: &
      n = 2147483647
   REAL(KIND=8), PARAMETER :: &
      s = 0.5d0

   REAL(KIND=8) :: A(n), B(n)
   INTEGER(KIND=8) :: i, j, stride

   READ(*,*) stride
   A=0.0d0
   B=0.5d0

   CALL ftrace_region_begin("loop")
   DO j = 1, stride
      DO i = j, n, stride
         A(i) = s + B(i)
      END DO
   END DO
   CALL ftrace_region_end("loop")

END PROGRAM bankconflicts
```



Note: To predict bank conflicts is complicated, but as general rule try to avoid strides which are multiples of 16 and/or 24.

Hyperplane Ordering

- Consider the following summation:

```
DO j = 2, m
   DO i = 2, n
      A(i,j) = A(i,j) + A(i, j-1) + A(i-1, j)
   END DO
END DO
```

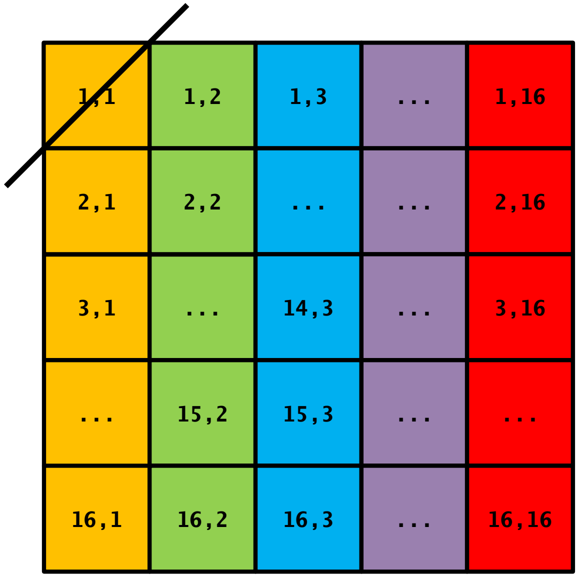
1,1	1,2	1,3	...	1,16
2,1	2,2	...	...	2,16
3,1	...	14,3	...	3,16
...	15,2	15,3	...	...
16,1	16,2	16,3	...	16,16

The elements `A(i, j) + A(i, j-1)`
Are always accessed with a stride of `n`, `thus potentially creating bank conflicts.`

Hyperplane Ordering

- Hyperplane ordering is a memory access procedure designed to minimize bank conflicts on vector computers without sacrificing generality.
- Instead of looping over rows or columns
  loop over skew diagonals where the offset is always (n-1) thus reducing the possibility of bank conflicts.

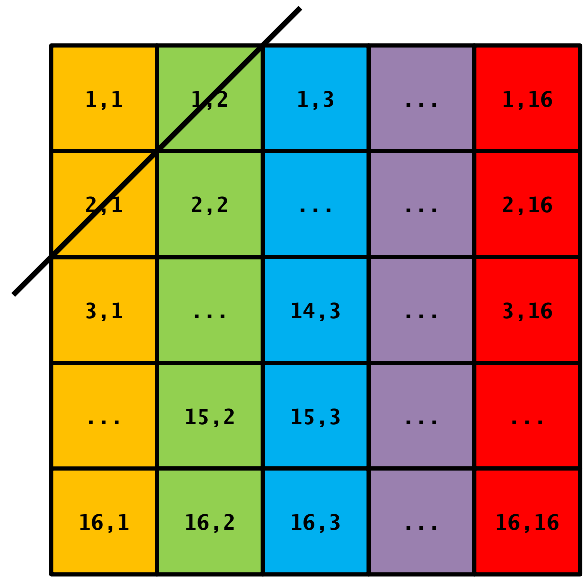
```
Iteration 1:

A(1,1)
```



Note that the sum of the matrix indices (i,j)
of one element is equal to the skew diagonal index -1

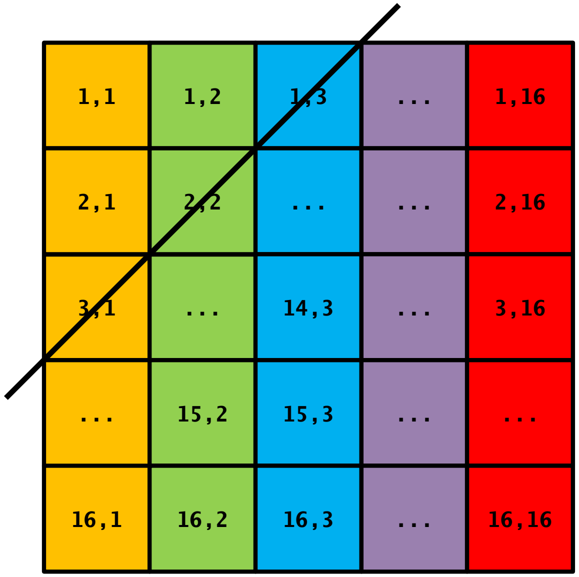
Hyperplane Ordering

```
Iteration 2:

A(2,1),
A(1,2)
```



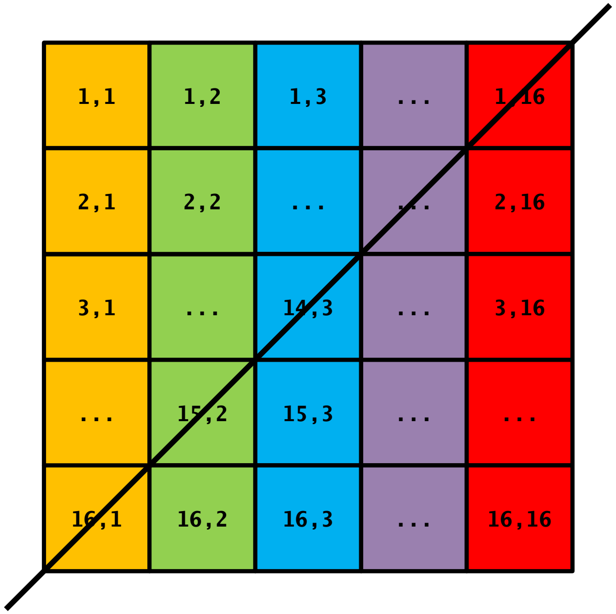
Hyperplane Ordering

```
Iteration 3:

A(3,1),
A(2,2),
A(1,3)
```



Hyperplane Ordering

```
Iteration n:

A(n,1),
A(n-1,2),
A(n-2,3),
...
A(3,n-2),
A(2,n-1),
A(1,n)
```



Hyperplane Ordering

```
Iteration 2n-1:

A(n,n)
```

1,1	1,2	1,3	...	1,16
2,1	2,2	...	...	2,16
3,1	...	14,3	...	3,16
...	15,2	15,3	...	...
16,1	16,2	16,3	...	16,16

\Orchestrating a brighter world NEC

Exercise 17 – Hyperplane Ordering

Short Loop Vectorization

Short Loops

- If the length of a loop is not known during compile time the compiler assumes the loop length to be larger than the hardware vector length.
- Note that reductions of short loops are extremly costly.

```
DO j = 1, n
    DO i = 1, m
      ! o not known during compile time
      ! but expected to be small
      DO k = 1, o
         B(i,j) = B(i,j) + A(k,i,j)
      END DO
    END DO
  END DO
```

Short Loops

- To aid the compiler with optimizing short loops several steps can be attempted
  1. Use !NEC$ shortloop directive to indicate that a loop is shorter than hardware vector length.
  2. Additionally exchange loop order to bring the short loop outside to unroll it.
  3. (High end solution) Optimize the unrolling using vector registers.

Exercise 18 – Short Loop Reduction

Thank You

Thank you for participating in the NEC training on vector computer programming.

We from NEC hope that it was an informative and fun experience for you.

If you encountered any problems with the training slides or the programming exercises please let us know in the Aurora forum.

If you have further questions regarding vector computing or programming our NEC team is always happy to help in the Aurora forum.

Your NEC Deutschland team

\Orchestrating a brighter world

NEC