# NEC SX-Aurora TSUBASA
## Tutorial for NEC Vector Engine

July 14, 2023

Presented by:
Raghunandan Mathur (raghunandan.mathur@necam.com)

# Before we begin…

- This presentation is available on the path:

```
/scratch/training/nec/hpc
```

- The hands-on source codes are available on the path

```
/scratch/training/nec/hpc/nec-aces-codes
```

- Copy the example codes to your user spaces using the following command

```
$ cp -r /scratch/training/nec/hpc/nec-aces-codes ~
```

…or ask our friends from Texas A&M how to do it using GUI.

# Table of Contents

# Introduction to Vector Architecture

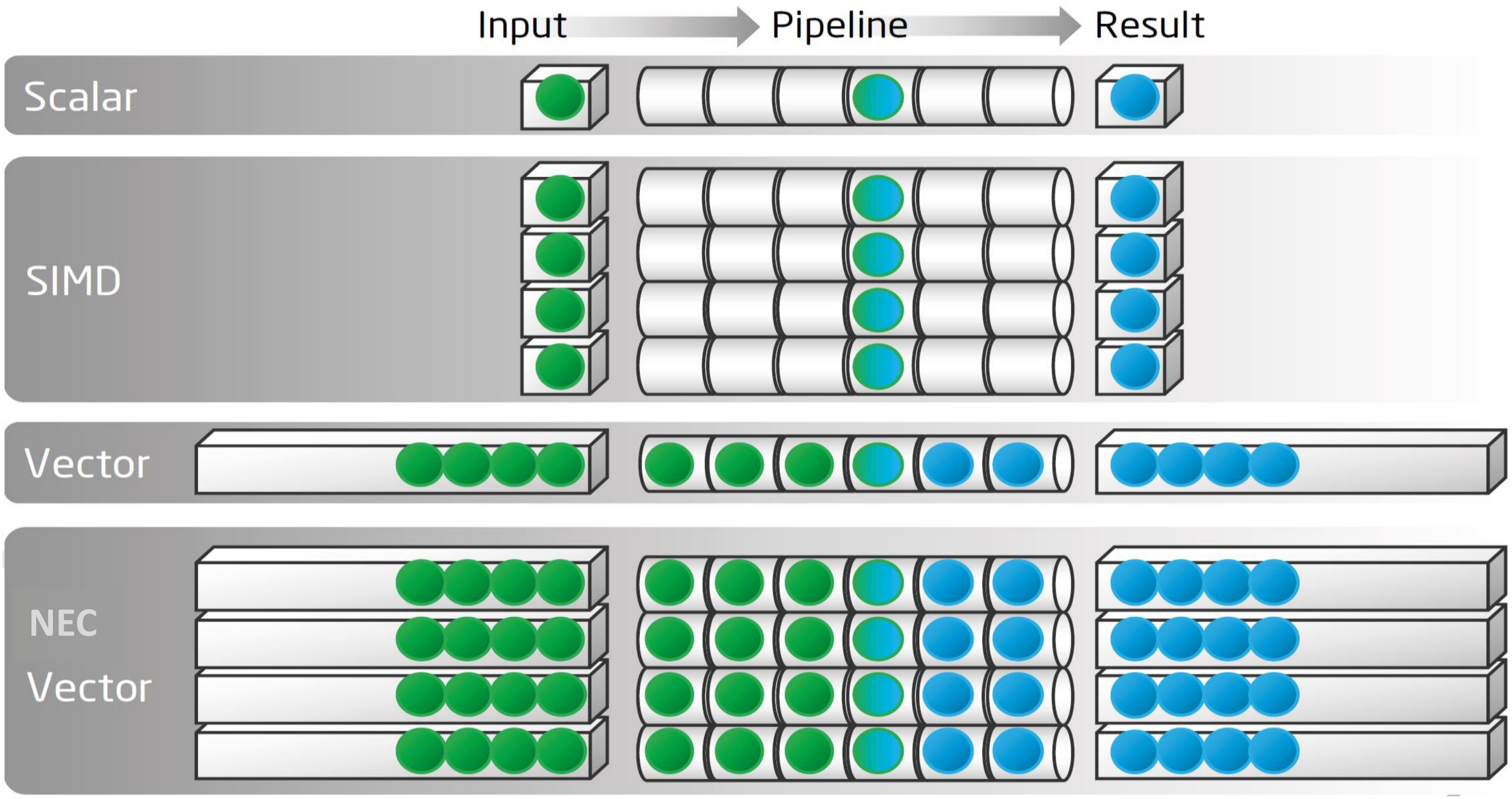
# An overview of processor architectures

No Pipelining | Single pipe →

No Pipelining | Multiple pipes →

With Pipelining | Single pipe →

With Pipelining | Multiple pipes →



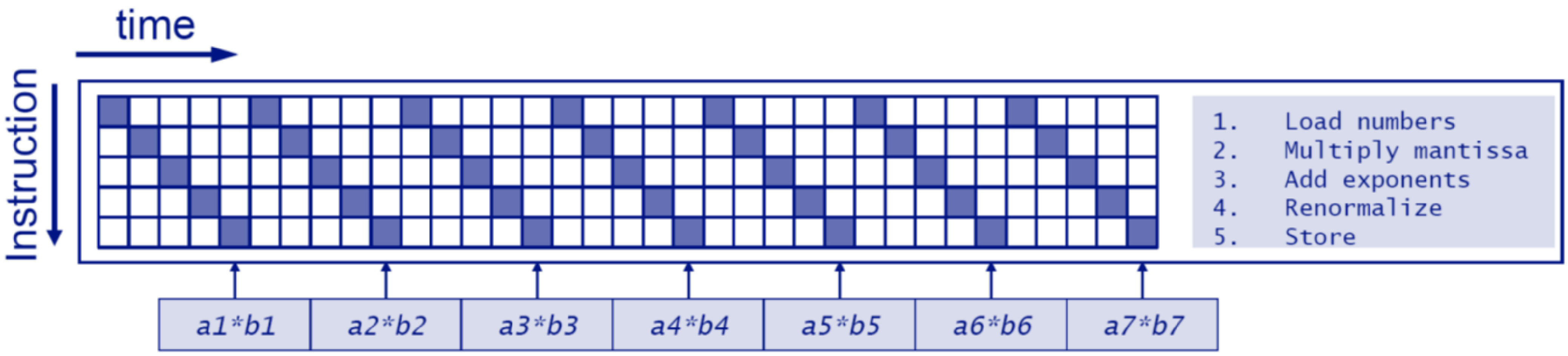
# Scalar processing

**Sample Program**

```
for (i = 0; i < 100; i++)
{
  c[i] = a[i] * b[i];
}
```

```
DO I = 1, 100
   C(I) = A(I) * B(I)
END DO
```

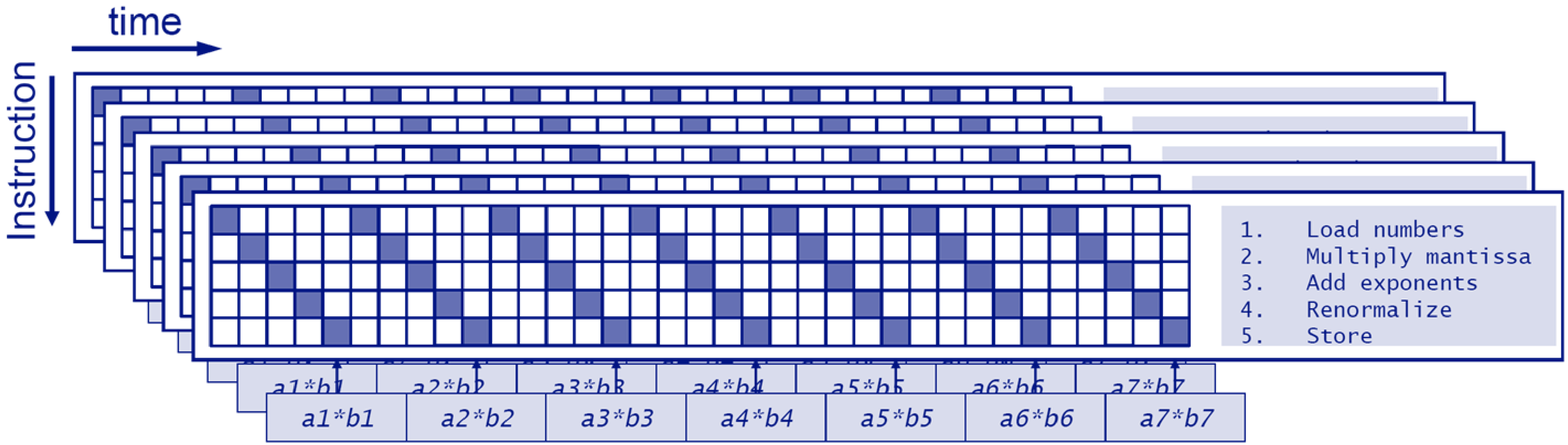
- Without pipelining + Single Pipe.




- One instruction is executed at a time.

# SIMD Processing (modern scalar)

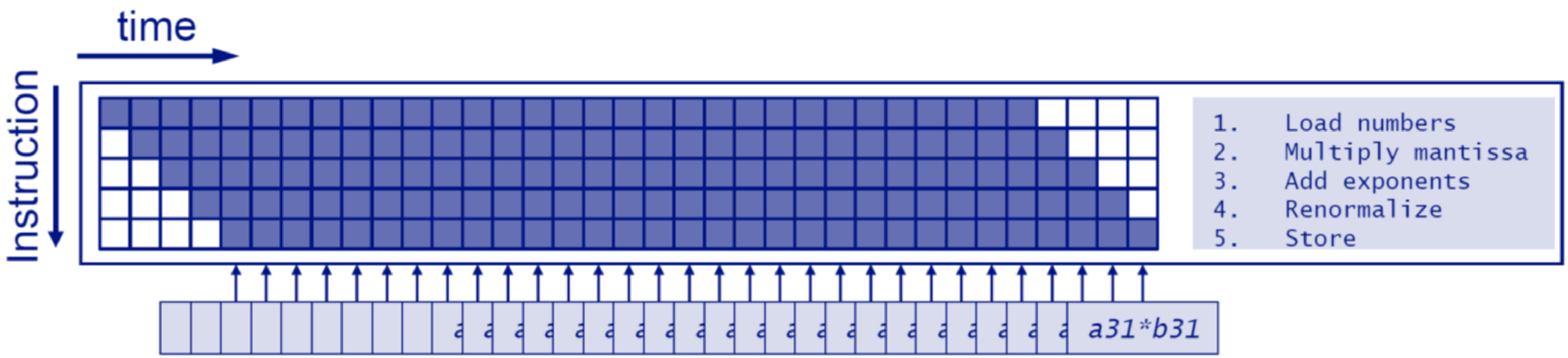
- Without Pipelining + Multiple Pipes




- Only one instruction is executed at a time.
- Parallel execution via multiple pipes.

# Vector Processing

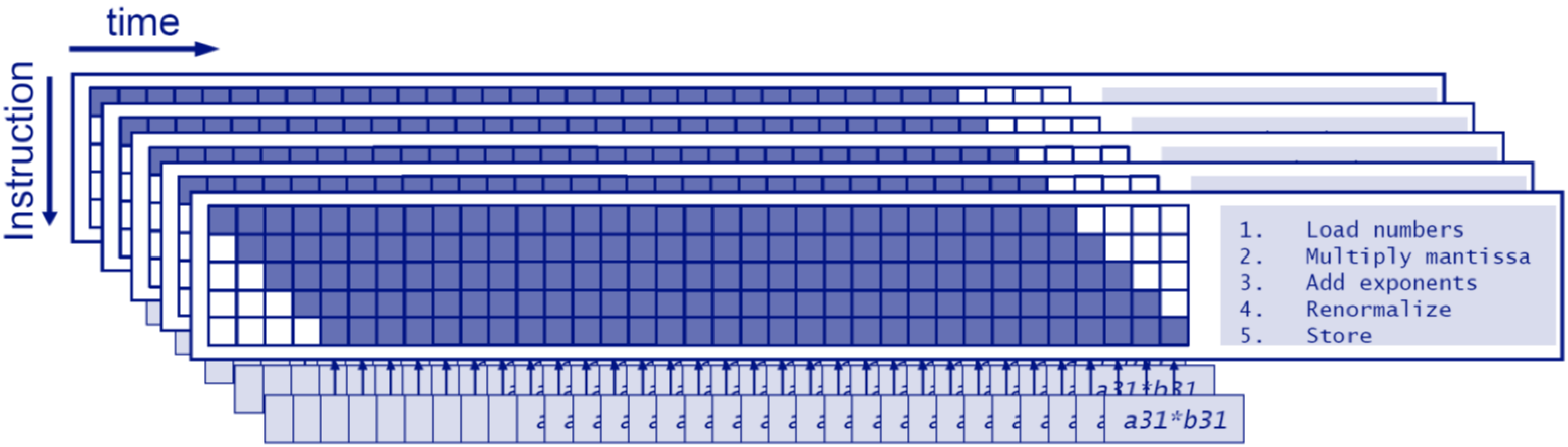
- With Pipelining + Single Pipe




- Execute instructions in parallel to hide latency.

# NEC Vector Engine

- With Pipelining + Multiple Pipes




- Execute instructions in parallel to hide latency.
- Parallel execution via multiple pipes.

# Programmer’s approach

## ◆ Scalar Approach

```
For all the 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?”

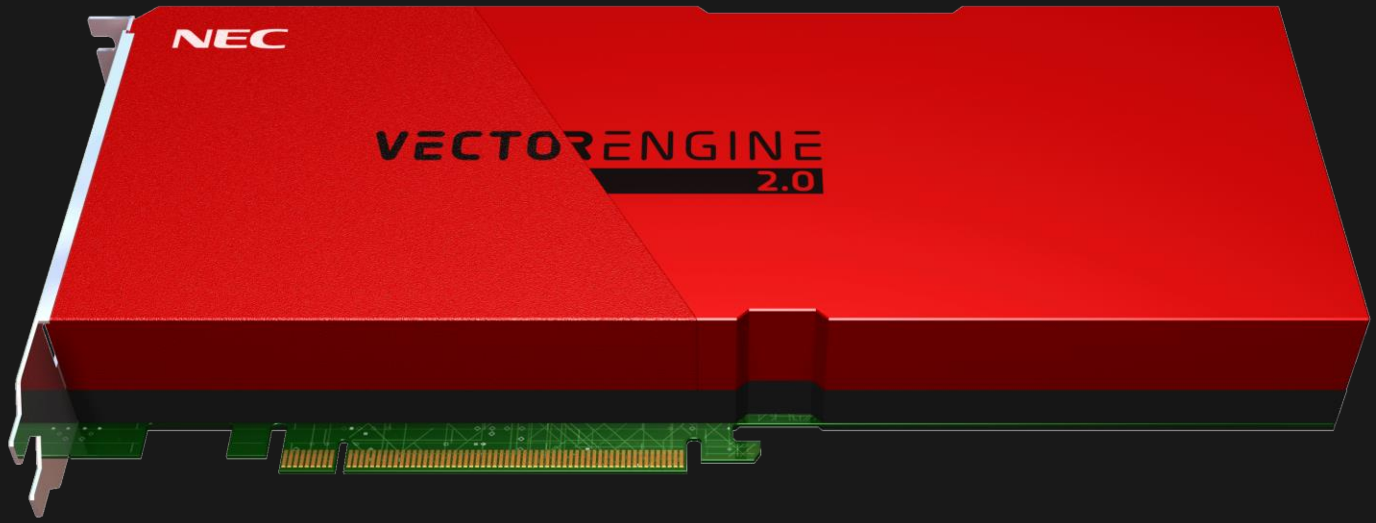
## ◆ Vector Approach

```
→ read vector instruction
→ decode vector instruction
→ fetch vectordata
→ 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?”

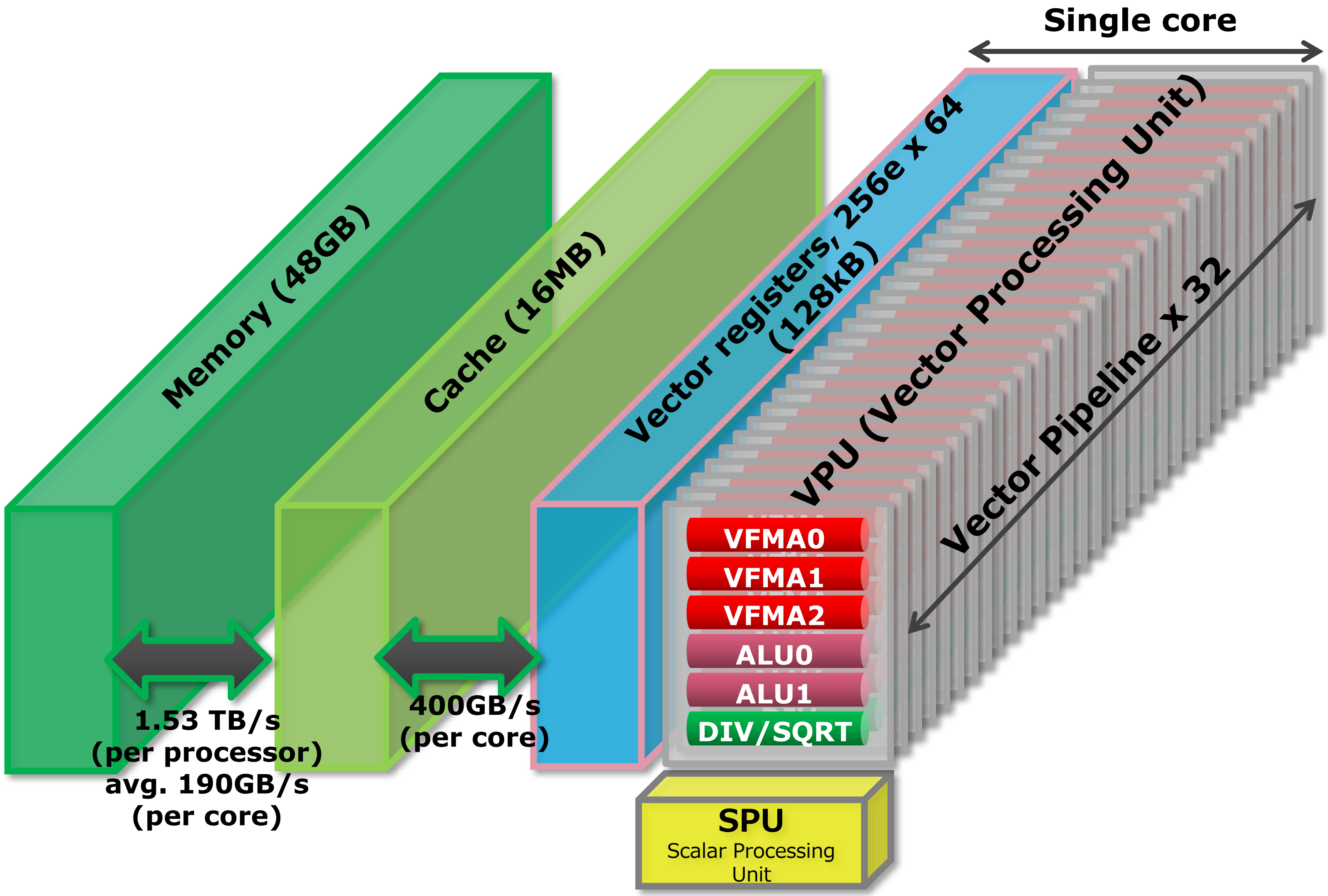
Instead of constantly reading/decoding instructions and fetching data, a vector computer reads one instruction and applies it to a set of vector data.

# Vector Processor on PCIe Card
## (High Memory Capacity & Bandwidth Processor)



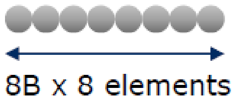
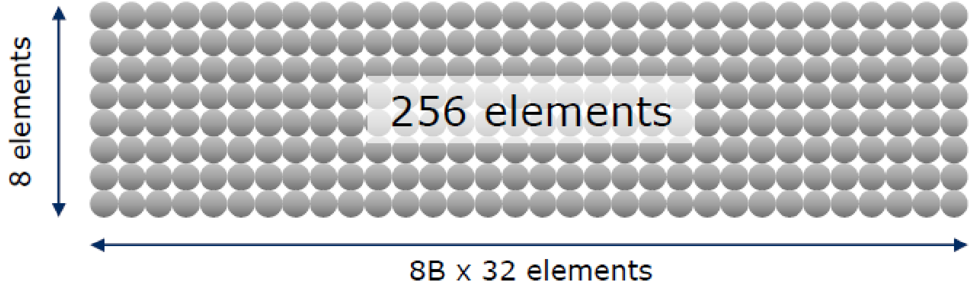

- 8 cores per processor
- 1.53 TB/s memory bandwidth
- 48GB on-chip HBM2 memory
  → Very High Memory Bandwidth
- 2.45TF performance (double precision)
- Low power consumption of under 300W
  → Operational power consumption around 200W
- Standard programming with Fortran/C/C++
  → No Special Programming Model Required

# Core Architecture and Operations




- Dedicated pipelines
  - Arithmetic operations (ALU pipe)
    - C = A + B
    - C = A – B
    - C = A * B
  - FMA (Fused Multiply-Add)
    - D = A + B * C
  - Division
    - C = A / B
  - Square root
    - B = SQRT (A)
  - …
- Other operations are combinations of standard operations
  - SIN, COS, TAN, ATAN, …
  - A**B, EXP, LOG, …
  - …

# Vector Length of VE vs modern SIMD

| | 512bit SIMD | Vector(256 vector length) |
|---|---|---|
| Data size/ operation | 512bit   | 16384bit **32x larger**   |
| Cycles/ operation | 1cycle | 8cycle |
| Processing speed | 512bit/cycle | 2048bit/cycle **4x larger** |

# Let’s do a quick hands-on!

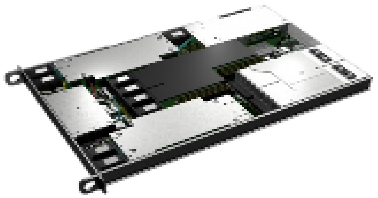
# Architecture of SX-Aurora TSUBASA

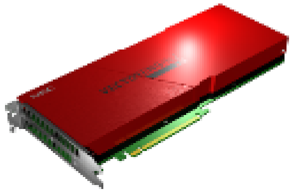
- SX-Aurora TSUBASA = VH + VE
- Linux + standard language (C/C++/Fortran/Python)
- Enjoy high performance with easy programming

**SX-Aurora TSUBASA Architecture**

Linux OS | Application

x86 server (VH) — PCIe — Vector Engine(VE)





## Hardware

- VH(Standard x86 server) + Vector Engine

## Software

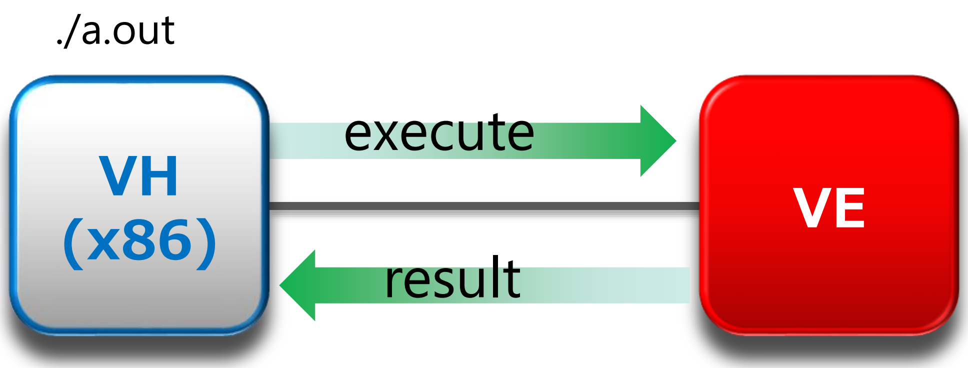
- Linux OS
- C/C++/Fortran/Python
- Automatic vectorization compiler

## Interconnect

- InfiniBand for MPI
  - ✓ VE-VE direct communication support

Easy programming (standard language) ▶ Automatic vectorization compiler ▶ Enjoy high Performance!

# Transparent execution




```
$ vi sample.c
$ gcc sample.c
$ ./a.out
Hello World !
$
$ ncc sample.c
ncc: opt(1135): sample.c, line 10: Outer loop conditionally
executes inner loop.
ncc: vec( 101): sample.c, line 13: Vectorized loop.
$ ./a.out
Hello World !
```

Program on VH

Compile using ncc

Compiler message

Execute on VE

Result shown on VH

# NEC SDK and compiler features

# NEC Compilers for Vector Engine

- NEC **C/C++ Compiler** for Vector Engine conforms to the following language standards:
  - ISO/IEC 9899:**2011** Programming languages - C
  - ISO/IEC 14882:**2014** Programming languages - C++
  - ISO/IEC 14882:**2017** Programming languages - C++
  - ISO/IEC 14882:**2020** Programming languages - C++ (partial – work in progress)

- NEC **Fortran Compiler** conforms to the following language standards.
  - ISO/IEC 1539-1:**2004** Programming languages – Fortran
  - ISO/IEC 1539-1:**2010** Programming languages – Fortran
  - ISO/IEC 1539-1:**2018** Programming languages – Fortran (partial – work in progress)

- NEC C/C++ and Fortran compilers conform to the following standards.
  - OpenMP Version **4.5**
  - OpenMP Application Program Interface Version **5.0** (partial – work in progress)

- Major Features
  - Automatic Vectorization
  - Automatic Parallelization and OpenMP C/C++
  - Automatic Inline Expansion

# Software

- Fortran
  - Fortran 2003
  - Fortran 2008
  - Fortran 2018
- C/C++
  - C11
  - C++14 / C++17 / C++20
- Python
  - NLCPy
- OpenMP
  - Version 4.5
  - Version 5.0
- Libraries
  - glibc
  - MPI version 3.1
  - Numerical libraries : BLAS, FFT, Lapack, Stencil, etc.
  - AI libraries : Frovedis (Apache Spark and Scikit-learn clone)
- Tools
  - GNU profiler
  - GNU debugger
  - TAU tool
  - NEC profilers : FtraceViewer, PROGINF, vftrace
- Hybrid programming
  - VE offloading
  - Reverse offloading
  - Hybrid MPI
  - OpenMP target (coming soon)

# NEC Numerical Library Collection (NLC)

- NLC is a collection of mathematical libraries that powerfully supports the development of numerical simulation programs.

**ASL Unified Interface**

Fourier transforms and Random number generators

**FFTW3 Interface**

Interface library to use Fourier Transform functions of ASL with FFTW (version 3.x) API

**ASL**

Scientific library with a wide variety of algorithms for numerical/statistical calculations:
Linear algebra, Fourier transforms, Spline functions, Special functions, Approximation and interpolation, Numerical differentials and integration, Roots of equations, Basic statistics, etc.

**Stencil Code Accelerator**

Stencil Code Acceleration (SCA)

**BLAS / CBLAS**

Basic linear algebra subprograms

**LAPACK**

Linear algebra package

**ScaLAPACK**

Scalable linear algebra package for distributed memory parallel programs

**SBLAS**

Sparse BLAS

**HeteroSolver**

Direct sparse solver

# Usage of the compilers

```
$ ncc               -O3 a.c b.c     … Compile and link C program
$ ncc   -report-all -O3 a.c b.c     … Compile and link C program
$ nfort -report-all -O3 a.f90 b.f90 … Compile and link Fortran program(a.f90 b.f90)
$ nc++              -O4 x.cpp y.cpp … Compile and link C++ program
```

| | |
|---|---|
| **-O4** | … Automatic vectorization with the highest level optimization |
| **-O3** | … Automatic vectorization with high level optimization |
| **-O2** | … Automatic vectorization with default level optimization |
| **-O1** | … Automatic vectorization with optimization without side-effects |
| **-O0** | … No vectorization and optimization |

High

Low

Options to control the level of automatic vectorization and optimization.

| | |
|---|---|
| **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 |

Compilers from the NEC SDK for the Vector Engine.

# Example of Typical Compiler Option Specification

| | |
| --- | --- |
| `$ nfort a.f90` | Compiling and linking with the default vectorization and optimization. |
| `$ nfort –O4 a.f90 b.f90` | Compiling and linking with the highest vectorization and optimization. |
| `$ nfort –fopenmp –O3 a.f90 b.f90` | Compiling and linking using OpenMP parallelization with the advanced vectorization and optimization. |
| `$ nfort –O4 –finline-functions a.f90 b.f90` | Compiling and linking using automatic inlining with the highest vectorization and optimization. |
| `$ ncc –O0 -g a.c b.c` | Compiling and linking with generating debugging information in DWARF without vectorization and optimization. |
| `$ ncc –g a.c b.c` | Compiling and linking with generating debugging information in DWARF with the default vectorization and optimization. |
| `$ ncc –E a.c b.c` | Performing preprocessing only and outputting the preprocessed text to the standard output. |
| `$ nc++ –fsyntax-only a.cpp b.cpp` | Performing only grammar analysis. |

# Program Execution

```
$ nfort a.f90 b.cf90
$ ./a.out
```

Executing a compiled program.

```
$ ./b.out data1.in
```

Executing a program getting input file and parameter from command line.

```
$ ./c.out < data2.in
```

Executing with redirecting an input file instead of standard input file.

```
$ env VE_NODE_NUMBER=1 ./a.out
```

Executing with specifying the index of VE.

```
$ nfort -mparallel -O3 a.f90 b.f90
$ export OMP_NUM_THREADS=4
$ ./a.out
```

Executing a parallelized program with specifying the number of threads.

```
$ mpincc a.c b.c
$ mpirun -ve 0    -np 8 ./a.out
$ mpirun -ve 0-1 -np 16 ./a.out
$ mpirun -ve 0-7 -np 64 ./a.out
```

Executing a compiled MPI program,

-- on a single VE card

-- on two VE cards

-- on eight VE cards

# Vectorization on NEC Vector Engine

# Vectorization Features

- An orderly arranged scalar data sequence such as a line, column, or diagonal of a matrix is called vector data. Vectorization is the replacement of scalar instructions with vector instructions.

**Execution image of scalar instructions**

```
A(1)   = B(1) + C(1)
A(2)   = B(2) + C(2)
A(3)   = B(3) + C(3)
…
A(100)= B(100) + C(100)
```

A(1) = B(1) + C(1)
A(2) = B(2) + C(2)
…
A(100) = B(100) + C(100)

Execute one calculation 100 times

**Execution image of vector instructions**

```
DO I = 1, 100
   A(I) = B(I) + C(I)
END DO
```

| A | | B | | C |
|---|---|---|---|---|
| A(1) | | B(1) | | C(1) |
| A(2) | = | B(2) | + | C(2) |
| … | | … | | … |
| A(100) | | B(100) | | C(100) |

Execute 100 calculation at once

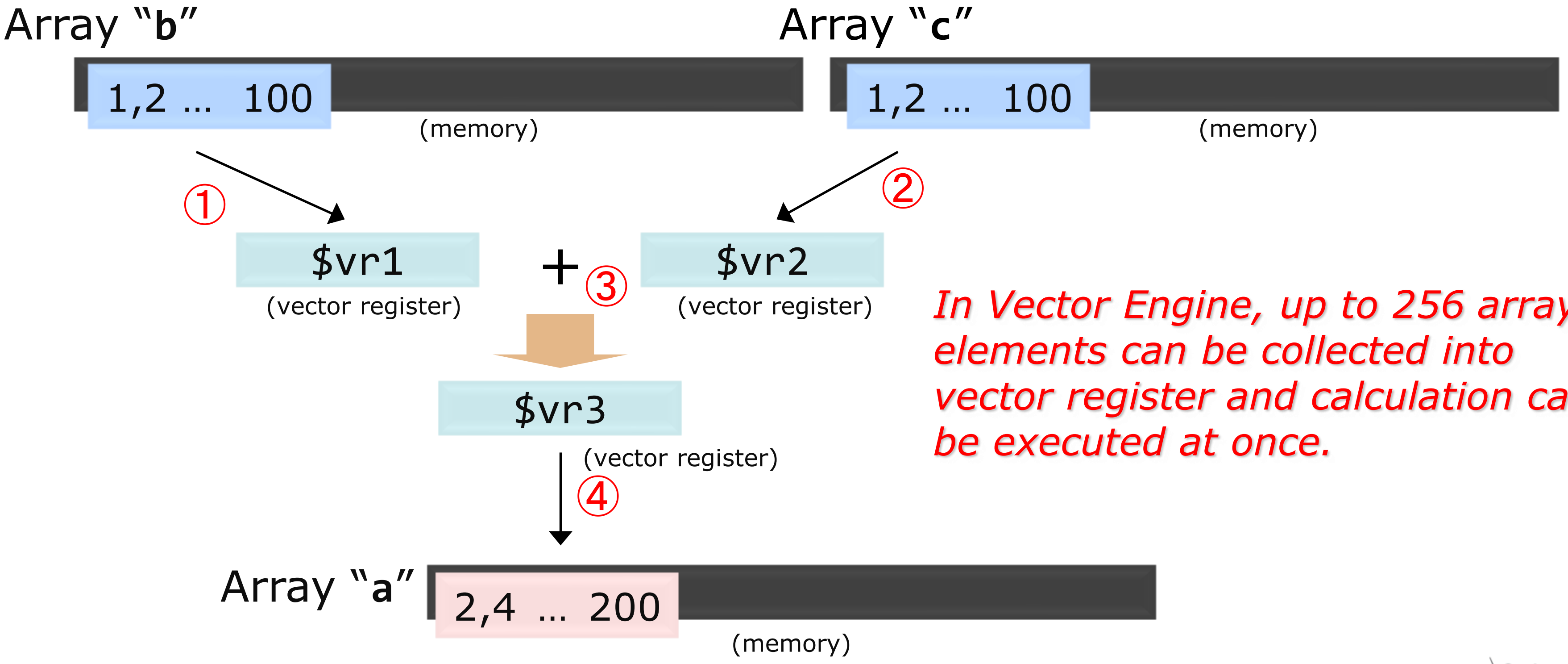
At most 256 calculation at once

# Order of Hardware Instructions

In the case of a scalar machine, these four instruction sequences must be repeated 100 times.

```
A(1)  = B(1)   + C(1)
A(2)  = B(2)   + C(2)
…
A(100)= B(100) + C(100)
  ④      ①     ③    ②
```

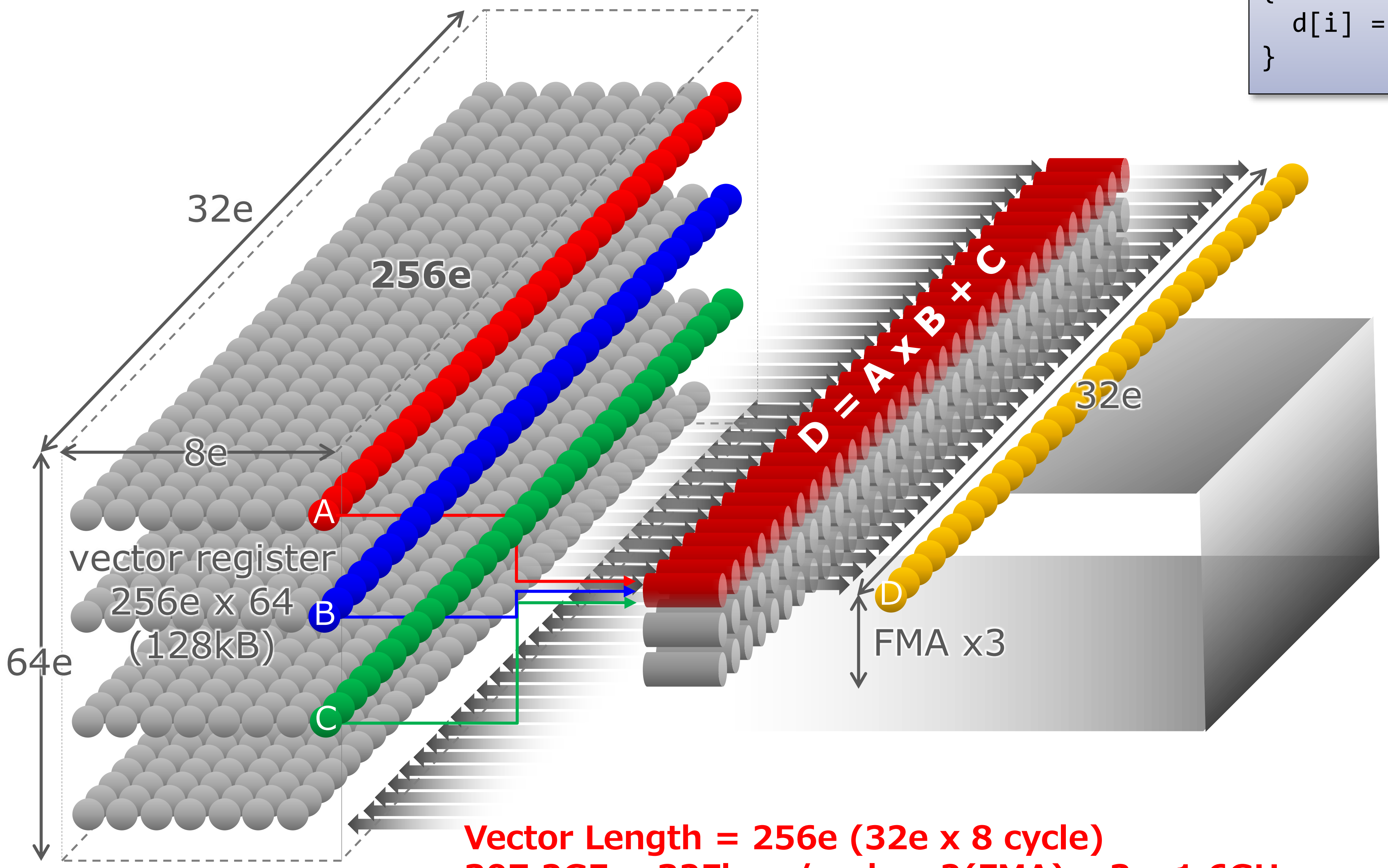
```
① VLoad  $vr1, B[1:100]
② VLoad  $vr2, C[1:100]
③ VAdd   $vr3, $vr1, $vr2
④ VStore $vr3, A[1:100]
```




*In Vector Engine, up to 256 array elements can be collected into vector register and calculation can be executed at once.*

# Vector Execution

```
for (i = 0; i < 256; i++)
{
  d[i] = a[i] * b[i] + c[i];
}
```

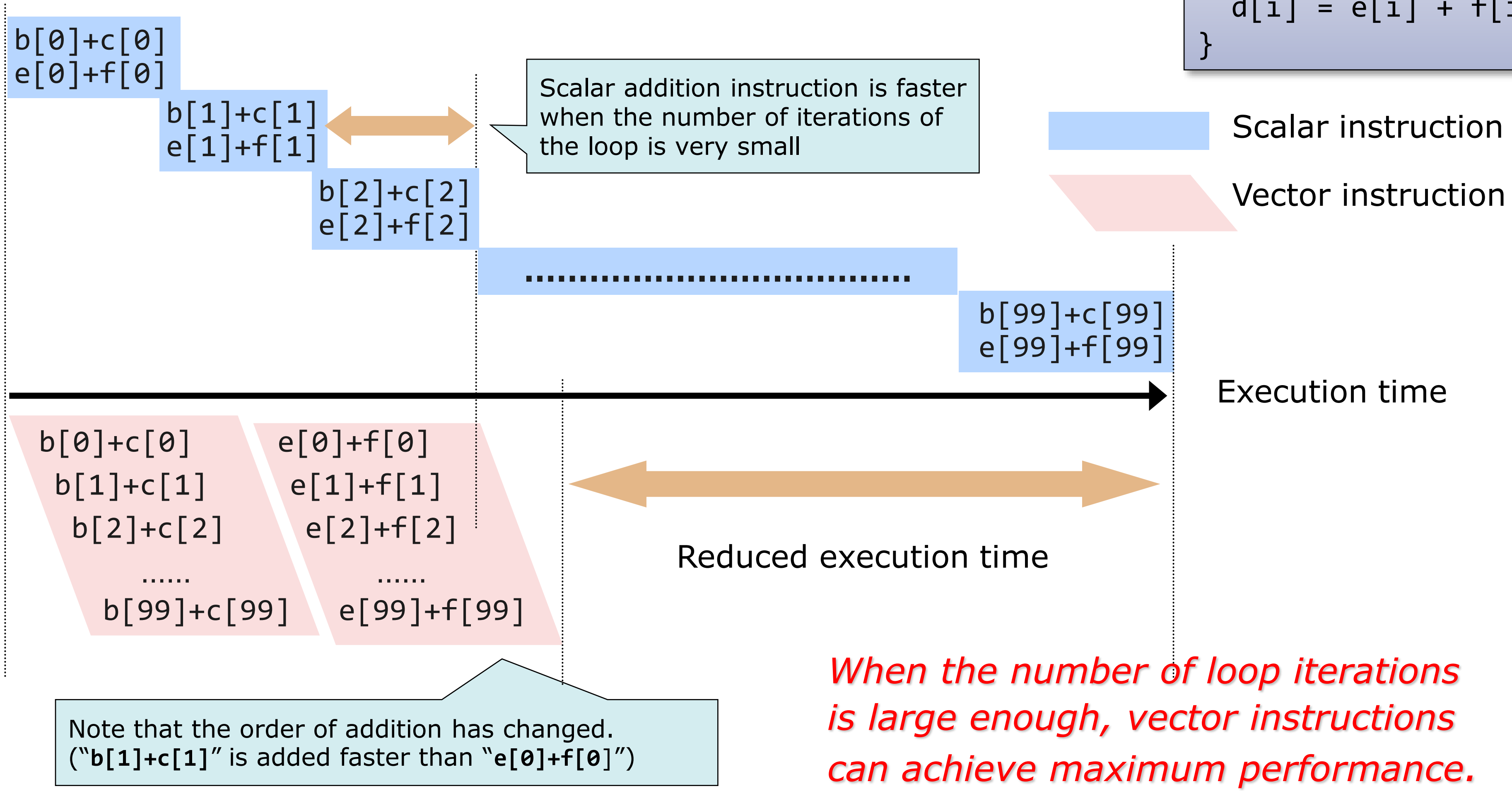



**Vector Length = 256e (32e x 8 cycle)**
**307.2GF = 32Flops/cycle x 2(FMA) x 3 x 1.6GHz**

# Order of instruction execution with time

Execution image of scalar addition instruction
(when two instructions are simultaneously executed)

```
for (i = 0; i < 100; i++)
{
  a[i] = b[i] + c[i];
  d[i] = e[i] + f[i];
}
```




*When the number of loop iterations is large enough, vector instructions can achieve maximum performance.*

Execution image of vector addition instruction

# Unvectorizable Dependencies

The calculation order cannot be changed, when array elements or variables which defined in the previous iteration are referred in the later iteration.

**Example 1**

```
for (i=2; i < n; i++)
  a[i+1] = a[i] * b[i] + c[i];
```

Unvectorizable, because the updated "**a**" value cannot be referenced.

Calculation order in scalar

```
a[3] = a[2] * b[2] + c[2];
a[4] = a[3] * b[3] + c[3];
a[5] = a[4] * b[4] + c[4];
a[6] = a[5] * b[5] + c[5];
  :         :
```

Calculation order in vector

```
a[3] = a[2] * b[2] + c[2];
a[4] = a[3] * b[3] + c[3];
a[5] = a[4] * b[4] + c[4];
a[6] = a[5] * b[5] + c[5];
```

a[n] : Updated "a" value

before update

**Example 2**

```
for (i=2; i < n; i++)
  a[i-1] = a[i] * b[i] + c[i];
```

Vectorizable, because the order of calculation does not change.

Calculation order in scalar

```
a[1] = a[2] * b[2] + c[2];
a[2] = a[3] * b[3] + c[3];
a[3] = a[4] * b[4] + c[4];
a[4] = a[5] * b[5] + c[5];
  :         :
```

Calculation order in vector

```
a[1] = a[2] * b[2] + c[2];
a[2] = a[3] * b[3] + c[3];
a[3] = a[4] * b[4] + c[4];
a[4] = a[5] * b[5] + c[5];
```

*Notice that there is no dependency between loop iterations.*

# Unvectorizable Dependencies

**Example 3**

```
for (i = 0; i < n; i++) {
    a[i] = s;
    s = b[i] + c[i];
}
```

Unvectorizable, because the reference of "S" appears before its definition in a loop.

Calculation order in scalar

```
a[0] = s ;
s = b[0] + c[0] ;
a[1] = s ;
s = b[1] + c[1] ;
      :
```

Calculation order in vector

```
a[0] = s ;
a[1] = s ;
       :
a[n-1] = s ;
s = b[0] + c[0] ;
s = b[1] + c[1] ;
       :
```

```
a[0] = s
for (i = 1; i < n; i++) {
    s = b[i-1] + c[i-1];
    a[i] = s;
}
s = b[n-1] + c[n-1];
```

It can be vectorized by transforming the program.

Calculation order in scalar

```
a[0] = s ;
s = b[0] + c[0] ;
a[1] = s ;
s = b[1] + c[1] ;
      :
```

Calculation order in vector

```
a[0] = s ;
s = b[0] + c[0] ;
s = b[1] + c[1] ;
       :
a[1] = s ;
a[2] = s ;
       :
```

# Unvectorizable Dependencies

**Example 4**

```
s = 1.0;
for (i=0; i < n; i++) {
   if (a[i] < 0.0)
      s = a[i];
   b[i] = s + c[i];
}
```

Cannot be vectorized when a variable definition may not be executed, even if its definition precedes its reference.

**Example 5**

```
for (i=0; i < n; i++) {
   if (a[i] < 0.0)
      s = a[i];
   else
      s = d[i];
   b[i] = s + c[i];
}
```

Can be vectorized, because there is always a definition of "**s**" before its reference.

**Example 6**

```
for (i=1; i < n; i++) {
   a[i] = a[i+k] + b[i];
}
```

Cannot be vectorized. It is not possible to determine whether there is a dependency or not, because the value of "**k**" is unknown at compilation.

*Unknown pattern in Example 1 or 2*

# Unvectorizable Dependencies

**Example 7**

```
for (i=0; i < n; i++) {
  d[i] = a[i] * b[i] + c[i];
  printf ("Calculating");
}
```

Cannot be vectorized due to a function call within the computational loop.

**Example 8**

```
for (i=0; i < n; i++) {
  b[i] = a[i] * func(a[i]);
}
…
double func (double x)
{
   return (x*x);
}
```

Cannot be vectorized originally due to a function call within the computational loop.

Inline expansion of func() can however help vectorize this loop easily and automatically.

**Example 9**

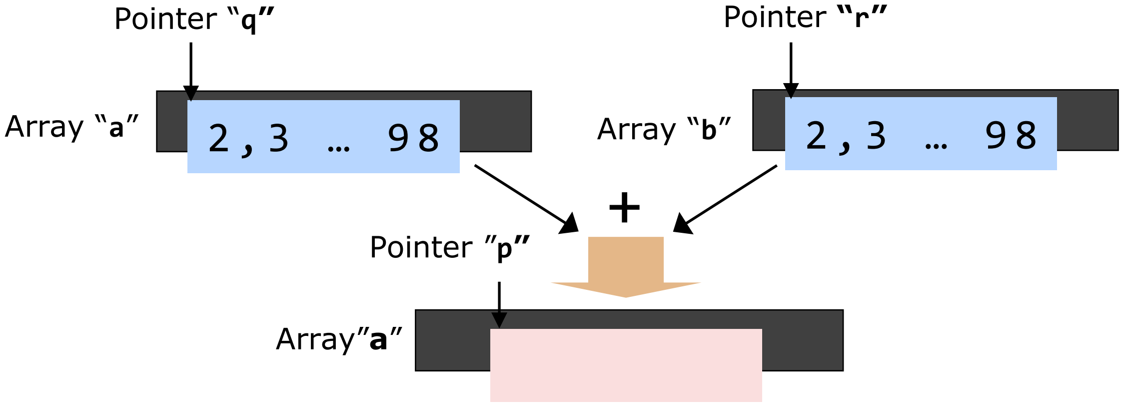
```
for (i=0; i < n; i++) {
  b[i] = a[i] * sin(a[i]);
}
```

Can be vectorized despite a function call within the computational loop because a few mathematical library functions are tuned within the SDK.

# C/C++ Pointer and Vectorization

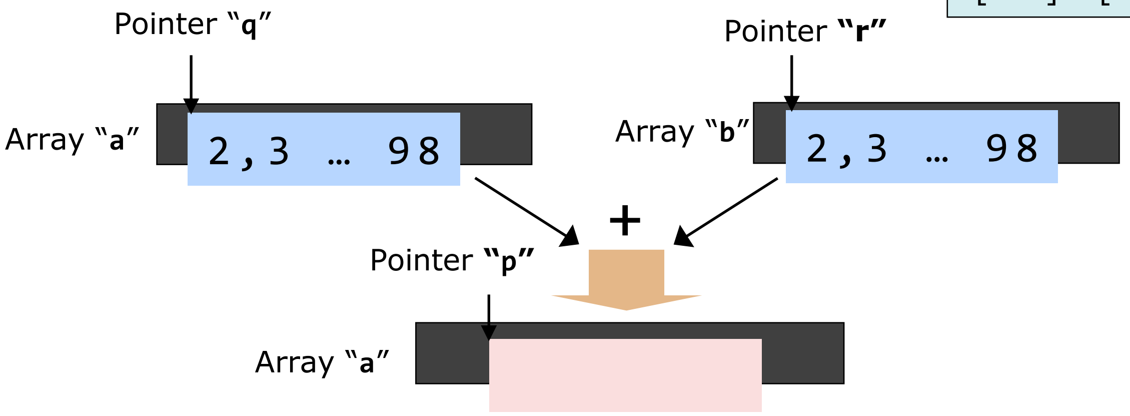
**Ex 1**: Cannot be vectorized when `p = &a[3], q = &a[2]`

Pattern of
a[i+1]=a[i]+…




**Ex 2**: Can be vectorized when `p=&a[1], q=&a[2]`

Pattern of
a[i-1]=a[i]+…




```
for (i = 2 ; i < n; i++) {
  *p = *q + *r;
  p++, q++, r++;
}
```

The pointer value is determined when program is executed.

It is regarded as unvectorizable dependency and not vectorized to avoid generating incorrect results, unless it is clear that there are no dependencies.

*Specifying the compiler option or **#pragma** to indicate that there are no dependecies.*

# Vectorization of IF Statement

Conditional branches (**`if`** statements) are also vectorized.

```
for (i = 0, i < 100; i++) {
  if (a[i] < b[i]) {
    a[i] = b[i] + c[i];
  }
}
```

**Execute with vector operations**

```
mask[1]   = a[1]  <  b[1]
mask[2]   = a[2]  <  b[2]
  :           :        :
mask[100] = a[100] < b[100]
```

```
if (mask[1] == true)   a[1] = b[1] + c[1]
if (mask[2] == true)   a[2] = b[2] + c[2]
 :           :              :
if (mask[100] == true) a[100] = b[100] + c[100]
```

# Vectorizable Loop Structure

- In order to be (automatically) vectorizable, a loop structure must fulfil certain criteria:

  - Loop count needs to be known upon entering the loop

```
! This vectorizes
DO i= 1, n
   do stuff
END DO
```

```
! This does not vectorize in general
DO WHILE (stuff to do)
   doing stuff
END DO
```

# Vectorizable Loop Structure

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

```
! This does not vectorize in general
DO WHILE (stuff to do)
   WRITE(*,*) stuff
END DO
```

# Vectorizable Loop Structure

- In order to be (automatically) vectorizable, a loop structure must fulfil 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 vectorizes
DO i= 1, n
   A(i) = A(i) + B(i)
END DO
```

```
! This does not vectorize
DO WHILE (stuff to do)
   A(i) = A(i-1) + B(i)
END DO
```

NOTE: The compiler is able to build a slower pseudo vectorized version of this.

# Vectorizable Loop Structure

- In order to be (automatically) vectorizable, a loop structure must fulfil 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 expanded inline
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 Loop Structure

- In order to be (automatically) vectorizable, a loop structure must fulfil 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 vectorizabledata structures (e.g. strings)

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

# NEC Compiler and automatic vectorization

- When the basic conditions for vectorization are not satisfied, the compiler performs as much vectorization as possible by transforming the program and using the special vector operations.

- Statement Replacement
- Loop Collapse
- Loop Interchange
- Partial Vectorization
- Conditional Vectorization
- Macro Operations
- Outer Loop Vectorization
- Loop Fusion
- Inline Expansion

# Statement Replacement

Source Program

```
for (i = 0; i < 99; i++) {
    a[i] = 2.0;
    b[i] = a[i+1];
}
```

Transformation Image

```
for (i = 0; i < 99; i++) {
    b[i] = a[i+1];
    a[i] = 2.0;
}
```

When this loop is vectorized, all the value from b[0] to b[98] will be 2.0. This loop do not satisfy the vectorization conditions.

The compiler replaces the statements in the loop to satisfy the vectorization conditions.

# Loop Collapse

Source Program

```
double a[M][N], b[M][N], c[M][N];
for (i = 0; i < M; i++)
   for (j = 0; j < N; j++)
      a[i][j] = b[i][j] + c[i][j];
```

Transformation Image

```
double a[M][N], b[M][N], c[M][N];
for (ij = 0; ij < M*N; ij++)
    a[0][ij] = b[0][ij] + c[0][ij];
```

A loop collapse is effective in increasing the loop iteration count and improving the efficiency of vector instructions.

# Loop Interchange

Source Program

```
for (j = 0; j < M; j++) {
  for (i = 0; i < N; i++) {
    a[i+1][j] = a[i][j] +  b[i][j];
   }
}
```

```
a[1][0] = a[0][0] + b[0][0];
a[2][0] = a[1][0] + b[1][0];
a[3][0] = a[2][0] + b[2][0];
a[4][0] = a[3][0] + b[3][0];
```

The loop "`for (i=0; i<N; i++)`" has unvectorizable dependency about the array `a`.

Transformation Image

```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      a[i+1][j] = a[i][j] + b[i][j];
   }
}
```

```
a[1][0] = a[0][0] + b[0][0];
a[1][1] = a[0][1] + b[0][1];
a[1][2] = a[0][2] + b[0][2];
a[1][3] = a[0][3] + b[0][3];
```

Interchanging loops removes unvectorizable dependency, and enable the loop "`for (j=0; j<M; j++)`" to be vectorized.

# Partial Vectorization

Source Program

```
for (i = 0; i < N; i++) {
    x = a[i] + b[i];
    y = c[i] + d[i];
    func(x, y);
}
```

Transformation Image

```
for (i = 0; i < N; i++) {
    wx[i] = a[i] + b[i];
    wy[i] = c[i] + d[i];
}
for (i = 0; i < N; i++) {
   func(wx[i], wy[i]);
}
```

Vectorizable

Unvectorizable

If a vectorizable part and an unvectorizable part exist together in a loop, the compiler divides the loop into vectorizable and unvectorizable parts and vectorizes just the vectorizable part.

To do this, work vectors (the array `wx` and `wy` in above example) are generated if necessary.

# Conditional Vectorization

Source Program

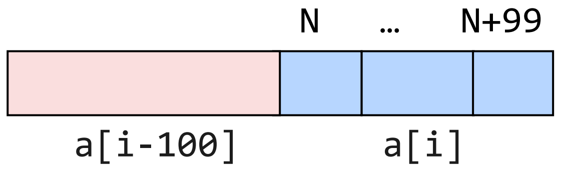
```
for (i = N; i < N+100; i++) {
    a[i] = a[i+k] + b[i];
}
```

Transformation Image

```
if (k >= 0 || k < -99) {
    // Vectorized Code
}
else {
    // Unvectorized Code
}
```

The compiler generates a variety of codes for a loop, including vectorized codes and scalar codes, as well as special codes and normal codes. The type of code is selected by run-time testing at execution when conditional vectorization is performed.

```
(When k=-1)
    a[i] = a[i-1]+b[i];

(When k=-100)
    a[i] = a[i-100]+b[i];
```



# Macro Operations

Sum

```
for (i = 0; i < N; i++)
    s = s + a[i];
```

Iteration

```
for (i = 0; i < N; i++)
    a[i] = a[i-1]*b[i]+c[i];
```

Maximum or minimum values

```
for (i = 0; i < N; i++) {
    if (xmax < x[i])
        xmax = x[i];
}
```

Although patterns like these do not satisfy the vectorization conditions for definitions and references, the compiler recognizes them to be special patterns and performs vectorization by using proprietary vector instructions.

# Outer Loop Vectorization

Source Program

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
        a[i][j] = 0.0;
    b[i] = 1.0;
}
```

Transformation Image

```
for (i = 0; i < N; i++) {
    for (j = 0; j <N; j++)
        a[i][j] = 0.0;
}
for (i = 0; i < N; i++)
    b[i] = 1.0;
```

*In this case, these loops are collapsed.*

The compiler basically vectorizes the innermost loop.
If a statement which is contained only in the outer loop exists, the compiler divides the loop and vectorizes the divided outer loop.

# Loop Fusion

Source Program

```
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
for (j = 0; j < N; j++)
    d[j] = e[j] * f[j];
```

Transformation Image

```
for (i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
    d[i] = e[i] * f[i];
}
```

- The compiler fuses consecutive loops which have the same iteration count and vectorizes the fused loop.
- If the same loop structure are continuous, they can be fused. But if there are the different loop structures, and other sentences, they cannot be fused.
- In order to increase speed, it is better to make same loop structures continuous as much as possible.

# Vectorization with Inlining

Source Program

```
for (i = 0; i < N; i++) {
    b[i] = func(a[i]);
    c[i] = b[i];
}
…
double func(double x)
{
    return x*x;
}
```

Transformation Image

```
for (i = 0; i < N; i++) {
    b[i] = a[i] * a[i];
    c[i] = b[i];
}
…
double func(double x)
{
    return x*x;
}
```

When the **`-finline-functions`** option is specified, the compiler expand the function directory at the point of calling it if possible. If the function is called in a loop, the compiler tries to vectorize the loop after inlining the function.

# Diagnostic Messages

- You can check the vectorization status from output messages and lists of the compiler.
  - Standard error … **-fdiag-vector=2** (detail)
  - Outputs diagnostic list … **-report-diagnostics**

```
$ ncc –fdiag-vector=2 abc.c
…
ncc: vec( 103): abc.c, line 1181: Unvectorized loop.
ncc: vec( 113): abc.c, line 1181: Unvectorizable dependency is assumed.: *(p)
ncc: vec( 102): abc.c, line 1234: Partially vectorized loop.
ncc: vec( 101): abc.c, line 1485: Vectorized loop.
…
$ ncc –report-diagnostics abc.c
…
$ less abc.L
FILE NAME: abc.c
…
FUNCTION NAME: func
DIAGNOSTIC LIST

 LINE             DIAGNOSTIC MESSAGE

  1181  vec( 103): Unvectorized loop.
  1181  vec( 113): Unvectorizable dependency is assumed.: *(p)
  1234  vec( 102): Partially vectorized loop.
  1485  vec( 101): Vectorized loop.
…
```

A message indicating that pointer **p** is considered to have a dependency that cannot be vectorized and has not been vectorized

List file name is "source file name".L

# Format List notations

```
$ ncc -report-format abc.c
```

Loop Mark
---------
C - Conditionally Vectorized
P - Parallelized
S - Partially Vectorized
U - Unrolled
V - Vectorized
W - Collapsed and Vectorized
Y - Parallelized and Vectorized
X - Interchanged and Vectorized
+ - Not Vectorized
* - Expanded

Line Mark
---------
C - Vector Scatter
F - Fused-multiply-add
G - Vector Gather
I - Inlined
M - Vector Matrix Multiply
R - Retain
V - Vreg

```
+------> for (j=0; j<n; j++) {
|V----->  for (i=0; i<m; i++) {
||          idx = j*m+i;
||     F    D[idx] = A[idx]+B[idx]*C[idx];
|V-----   }
+------  }
```

```
W------> for (i = 0; i < n; i++) {
|*-----> for (j = 0; j < m; j++) {
||                :
||                :
|*-----  }
W------ }
```

```
P------>    for (j=0; j<n; j++) {
|V----->      for (i=0; i<m; i++) {
||               :
||               :
|V-----        }
P------      }
```

# Performance Analysis Tools

# Performance Information of Vector Engine

## ◆ PROGINF

- Performance information of the whole program.
- The overhead to obtain the performance information is low.

## ◆ FTRACE

- Performance information of each function.
- It is necessary to re-compile and re-link the program.
- When frequencies for function calls high, the overhead to get performance information and the execution time may increase.

# PROGINF

**Performance information of the whole program**

```
$ ncc -O4  a.c b.c c.c
$ ls a.out
a.out
$ export VE_PROGINF=DETAIL
$ ./a.out
            ********  Program  Information  ********
  Real Time (sec)                        :             11.329254
  User Time (sec)                        :             11.323691
  Vector Time (sec)                      :             11.012581
  Inst. Count                            :            6206113403
  V. Inst. Count                         :            2653887022
  V. Element Count                       :          619700067996
  V. Load Element Count                  :           53789940198
  FLOP count                             :          576929115066
  MOPS                                   :          73492.138481
  MOPS (Real)                            :          73417.293683
  MFLOPS                                 :          50976.512081
  MFLOPS (Real)                          :          50924.597321
  A. V. Length                           :            233.506575
  V. Op. Ratio (%)                       :             99.572922
  L1 Cache Miss (sec)                    :              0.010847
  CPU Port Conf. (sec)                   :              0.000000
  V. Arith. Exec. (sec)                  :              8.406444
  V. Load Exec. (sec)                    :              1.384491
  VLD LLC Hit Element Ratio (%)          :            100.000000
  Power Throttling (sec)                 :              0.000000
  Thermal Throttling (sec)               :              0.000000
  Max Active Threads                     :                     1
  Available CPU Cores                    :                     8
  Average CPU Cores Used                 :              0.999509
  Memory Size Used (MB)                  :            204.000000
```

Time information

Number of instruction executions

Vectorization, memory and parallelization information

Set the environment variable "**VE_PROGINF**" to "**YES**" or "**DETAIL**" and run the executable file.

"**YES**" … Basic information.
"**DETAIL**" … Basic and memory information.

# FTRACE

## Performance information of each function

```
$ ncc -ftrace a.c b.c c.c       (Compile and link a program with -ftrace to an executable file)
$ ./a.out
$ ls ftrace.out
ftrace.out                      (At the end of execution, ftrace.out file is generated in a working directory)
$ ftrace                        (Type ftrace command and output analysis list to the standard output)
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*

Execution Date : Thu Mar 22 17:32:54 2018 JST
Total CPU Time : 0:00'11"163 (11.163 sec.)


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS     CONF HIT E.%

    15000     4.762( 42.7)     0.317  77117.2  62034.6  99.45 251.0     4.605   0.002    0.000  100.00 funcA
    15000     3.541( 31.7)     0.236  73510.3  56944.5  99.46 216.0     3.554   0.000    0.000  100.00 funcB
    15000     2.726( 24.4)     0.182  71930.2  27556.5  99.43 230.8     2.725   0.000    0.000  100.00 funcC
        1     0.134(  1.2)   133.700  60368.8  35641.2  98.53 214.9     0.118   0.000    0.000    0.00 main
------------------------------------------------------------------------------------------------------------------
    45001    11.163(100.0)     0.248  74505.7  51683.9  99.44 233.5    11.002   0.002    0.000  100.00 total
```

For an MPI program, multiple **ftrace.out** files are generated. Specify them by **-f** option.

```
$ ls ftrace.out.*
ftrace.out.0.0   ftrace.out.0.1   ftrace.out.0.2   ftrace.out.0.3
$ ftrace -f ftrace.out.0.0 ftrace.out.0.1 ftrace.out.0.2 ftrace.out.0.3
```

# Objectives of Program Tuning

# Objectives of Program Tuning

## ◆ Raising the Vectorization Ratio

- The vectorization ratio is the ratio of the portion processed by vector instructions in the whole program.
- The vectorization ratio can be improved by removing the cause of non-vectorization.
  - Increase the part processed by vector instructions.

## ◆ Improving Vector Instruction Efficiency

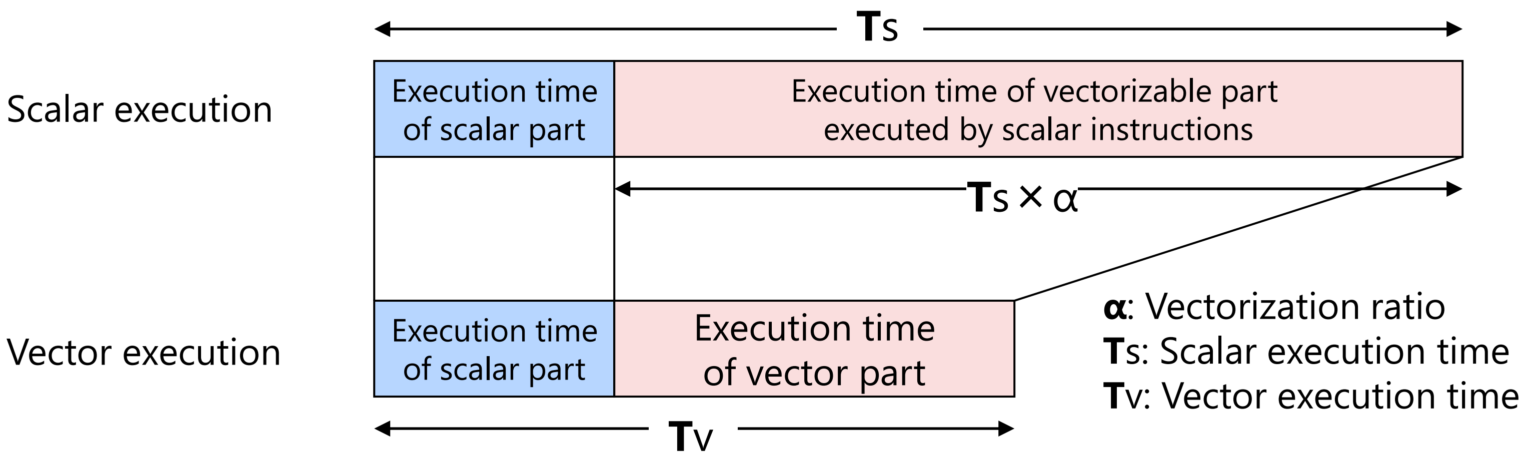
- Increase the amount of data processed by one vector instruction.
  - Make the iteration count of  a loop (loop length) as long as possible.
- Avoid vectorization when the length of the loop is short.

## ◆ Improving Memory Access Efficiency

- Avoid using a list vector.

# Vectorization Ratio or Vector Operation Ratio

- The ratio of the part processed by vector instructions in whole program



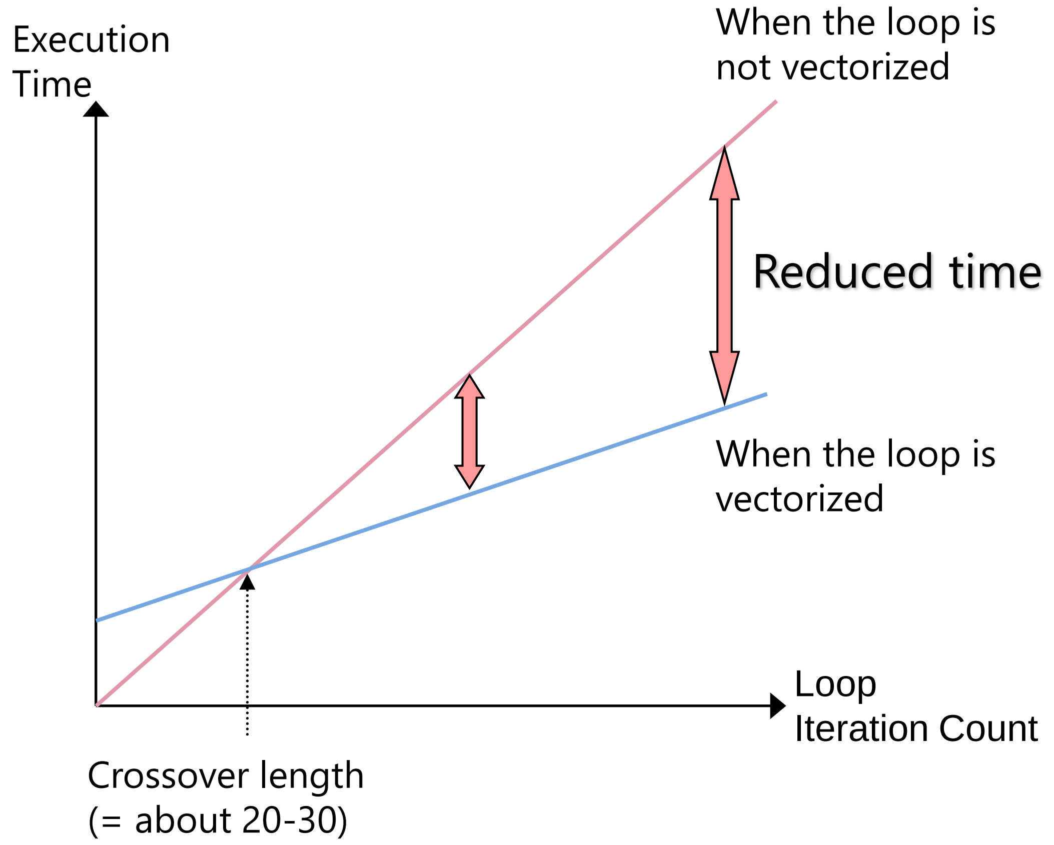

- The vector operation ratio is used instead of the vectorization ratio

$$\textbf{Vector operation ratio} = 100 \times \frac{\text{Number of vector instruction execution elements}}{\text{Execution count of all instructions} - \text{Execution count of vector instructions} + \text{Number of vector instruction execution elements}}$$

# Loop Iteration Count and Execution Time

**Improving vector instruction efficiency**

- To maximize the effect of vectorization,  the loop iteration count should be made as long as possible
  - Increase the amount of data processed by one vector instruction.




It is difficult to analyze iteration count for each loops.

Analyze **average vector length**.

*The average number of date processed by one vector instruction.*
*The maximum number is 256.*

# Process of Tuning

- Finding the function whose execution time is long, vector operation ratio is law and average vector length is short from the performance analysis information
  - PROGINF
    - Execution time, vector operation ratio and average vector length of the whole program.
  - FTRACE
    - Execution time, execution count, vector operation ratio and average vector length of each function.

↓

- Finding unvectorized loops in the function from diagnostics for vectorization

↓

- Improving vectorization by specifying compiler options and **`#pragma`** directives

# PROGINF

**Sample Report**

```
  ********  Program  Information  ********
 Real Time (sec)                  :            11.336602
 User Time (sec)                  :            11.330778
 Vector Time (sec)                :            11.018179
 Inst. Count                      :           6206113403
 V. Inst. Count                   :           2653887022
 V. Element Count                 :         619700067996
 V. Load Element Count            :          53789940198
 FLOP count                       :         576929115066
 MOPS                             :         73455.206067
 MOPS (Real)                      :         73370.001718
 MFLOPS                           :         50950.894570
 MFLOPS (Real)                    :         50891.794092
 A. V. Length                     :           233.506575
 V. Op. Ratio (%)                 :            99.572922
 L1 Cache Miss (sec)              :             0.010855
 CPU Port Conf. (sec)             :             0.000000
 V. Arith. Exec. (sec)            :             8.410951
 V. Load Exec. (sec)              :             1.386046
 VLD LLC Hit Element Ratio (%)    :           100.000000
 Power Throttling (sec)           :             0.000000
 Thermal Throttling (sec)         :             0.000000
 Max Active Threads               :                    1
 Available CPU Cores              :                    8
 Average CPU Cores Used           :             0.999486
 Memory Size Used (MB)            :           204.000000
```

- **A.V.Length** (Average vector length)
  - Indicator of vector instruction efficiency.
  - The longer, the better (Maximum length: 256).
  - If this value is short, the iteration count of the vectorized loops is insufficient.
- **V.Op.Ratio** (Vector operation ratio)
  - Ratio of data processed by vector instructions.
  - The larger, the better (Maximum rate: 100).
  - If this value is small, the number of vectorized loops is small or there are few loops in the program.

# FTRACE

- A feature used to obtain performance information of each function
  - Focus on V.OP.RATIO (Vector operation ratio) and AVER.V.LEN (Average vector length) as well as PROGINF, and analyze the performance of each function.

```
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*

Execution Date : Thu Mar 22 15:47:42 2018 JST
Total CPU Time : 0:00'11"168 (11.168 sec.)


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS      CONF HIT E.%

    15000     4.767( 42.7)     0.318  77030.2  61964.6  99.45 251.0     4.610   0.002     0.000  100.00 funcA
    15000     3.541( 31.7)     0.236  73505.6  56940.8  99.46 216.0     3.555   0.000     0.000  100.00 funcB
    15000     2.726( 24.4)     0.182  71930.1  27556.5  99.43 230.8     2.725   0.000     0.000  100.00 funcC
        1     0.134(  1.2)   133.700  60368.9  35641.3  98.53 214.9     0.118   0.000     0.000    0.00 main
---------------------------------------------------------------------------------------------------------------------
    45001    11.168(100.0)     0.248  74468.3  51657.9  99.44 233.5    11.008   0.002     0.000  100.00 total
```

# Debugging

# Traceback Information

Compile and link with **-traceback**.

Set the environment variable "**VE_TRACEBACK**" to "**FULL**" or "**ALL**" at execution.

Set the environment variable "**VE_FPE_ENABLE**" to catch arithmetic exceptions.

- "**DIV**" … Divide-by-zero exception
- "**INV**" … Invalid operation exception
- "**DIV,INV**" … Both exceptions

Note: "**VE_FPE_ENABLE**" can be set to other values but traceback basically uses "**DIV**" or "**INV**".

```
#include <stdio.h>
int main(void) {
    printf("%f\n",1.0/0.0);
}
```

Occur "divide-by-zero"

```
$ ncc -traceback main.c
$ export VE_TRACEBACK=FULL
$ export VE_ADVANCEOFF=YES
$ export VE_FPE_ENABLE=DIV
$ ./a.out
Runtime Error: Divide by zero at 0x600000000cc0
[ 1] Called from 0x7f5ca0062f60
[ 2] Called from 0x600000000b70
Floating point exception
$ naddr2line -e a.out -a 0x600000000cc0
0x0000600000000cc0
/.../main.c:3
```

- Compile and link with **-traceback**
- Use traceback information
- Advance-mode is off
- Catch exception of "divide-by-zero"
- Traceback information
- Specify where the exception occurs
- Notice that divide-by-zero is occurring in the 3rd line in the main.c file

# Using GDB

Specify **-g** to the files including the functions which you want to debug, in order to minimize performance degradation

```
$ ncc -O0 -g -c a.c
$ ncc -O4 -c b.c c.c
$ ncc a.o b.o c.o
$ gdb a.out
(gdb) break func
Breakpoint 1 at func
(gdb) run
Breakpoint 1 at func
(gdb) continue
...
```

Only a.c is compiled with **-O0 -g**

The others are compiled without **-g**

Run GDB

- When debugging without **-O0**, compiler optimization may delete or move code or variables, so the debugger may not be able to reference variables or set breakpoints.
- The exception occurrence point output by traceback information can be incorrect by the advance control of HW. The advance control can be stopped to set the environment variable **VE_ADVANCEOFF=YES**. The execution time may increase substantially to stop the advance control. Please take care it.

# Strace: Trace of system call

```
$ /opt/nec/ve/bin/strace ./a.out
...
write(2, "delt=0.0251953, TSTEP".., 27)           = 27
open(“MULNET.DAT", O_WRONLY|O_CREAT|O_TRUNC, 0666)= 5
ioctl(5, TCGETA, 0x8000000CC0)                    Err#25 ENOTTY
fxstat(5, 0x8000000AB0)                           = 0
write(5, "1 2 66 65", 4095)                       = 4095
write(5, "343 342", 4096)                         = 4096
write(5, "603 602", 4096)                         = 4096
write(5, "863 862", 4094)                         = 4094
write(5, "1105 1104", 4095)                       = 4095
write(5, "1249 1313 1312", 4095)                  = 4095
write(5, "1456 1457 1521 1520", 4095)             = 4095
write(5, "1727", 4095)                            = 4095
...
```

System call arguments　　　　System call return values

## Arguments and return values of system calls are output

- You can check if the system library has been called properly.
- You should carefully select system calls to be traced by **`-e`** of **`strace`**, because the output would be so many.

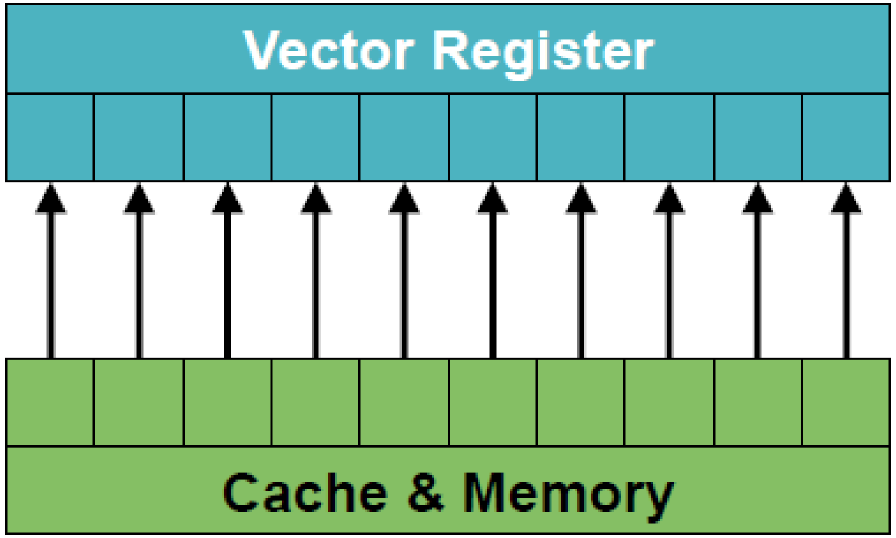
# Hands-on Example : Memory Access

# 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)
```

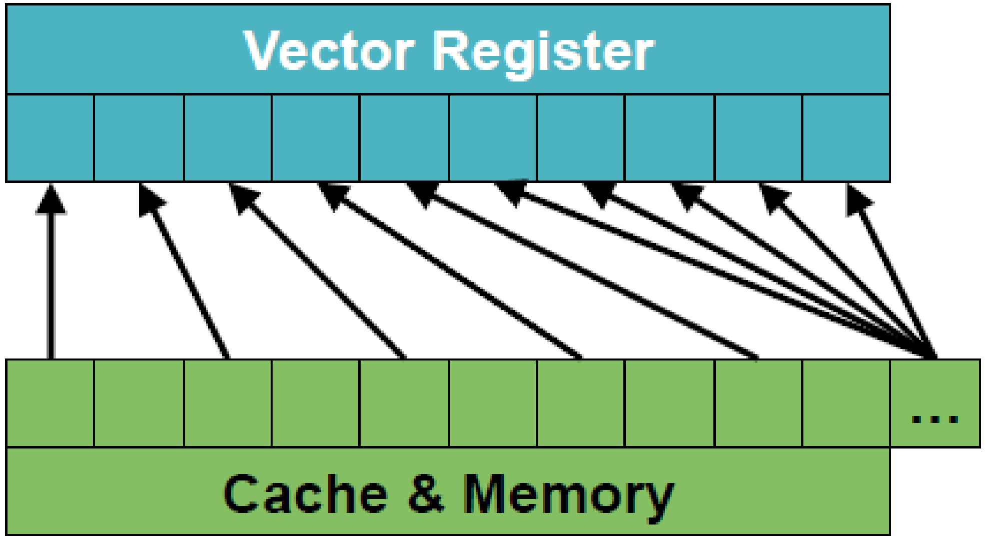



Optimal memory access.

# 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)
```



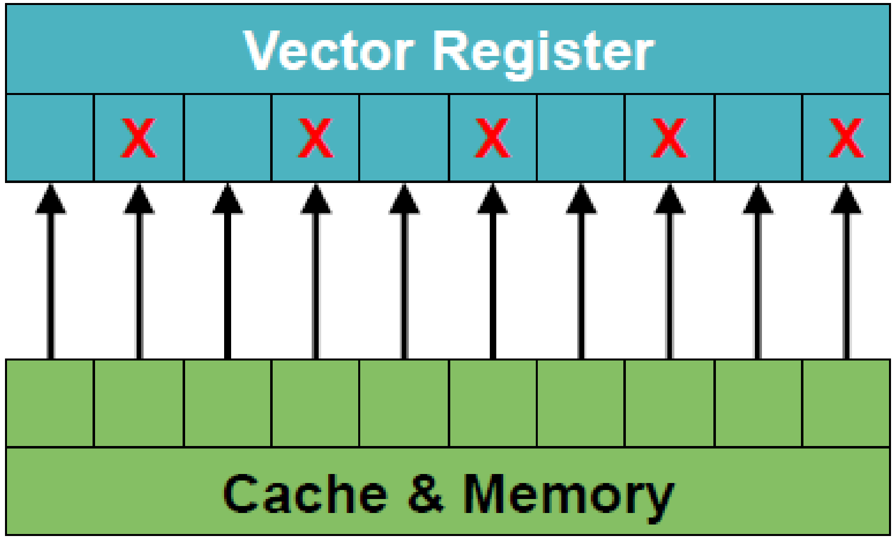

Not optimal due to partially used cache lines.

# 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.

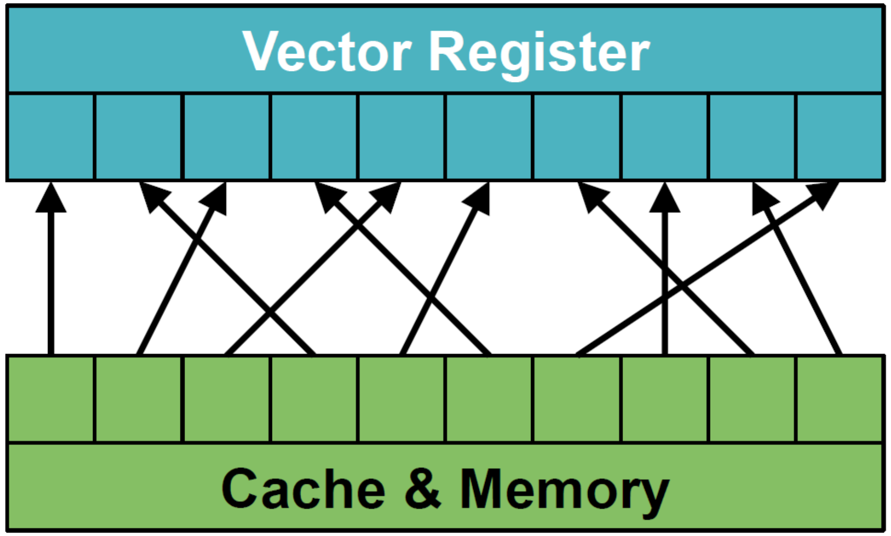
Note that all elements are loaded into the vector registers and operated on, but the write back is only performed if the condition applies.

# 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))
```

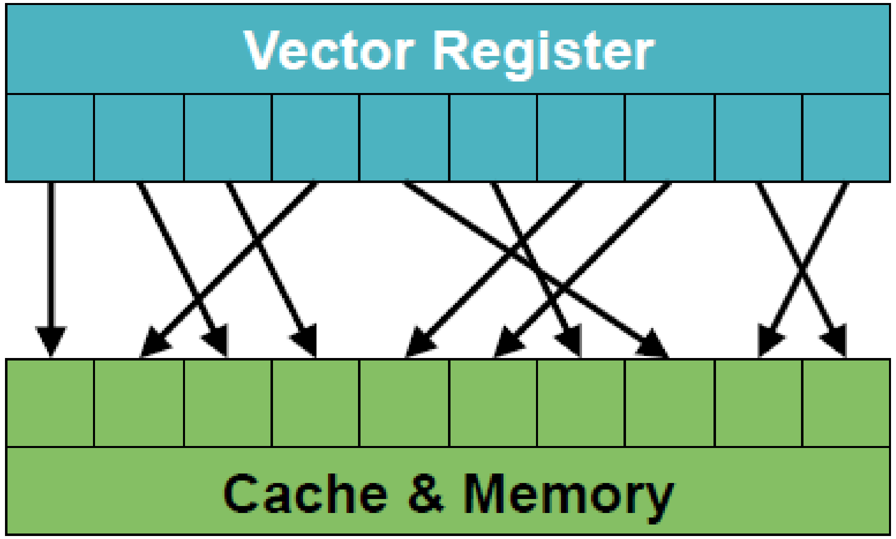



Insufficient due to random memory access, potential bank conflicts, partially used cache lines.

# 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)
```



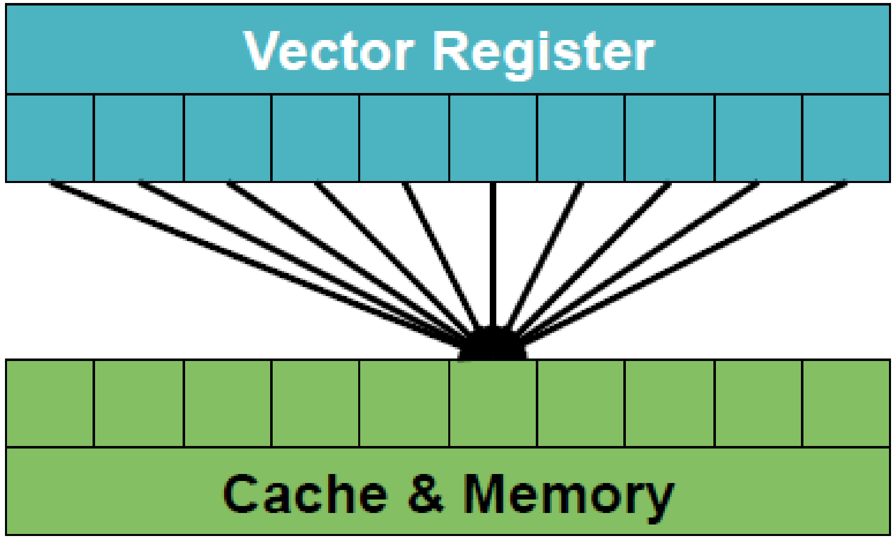

Inefficient due to random memory access, potential bank conflicts, partially used cache lines.

# 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/….

# Hands-on Example: Loop Collapse

# 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
```

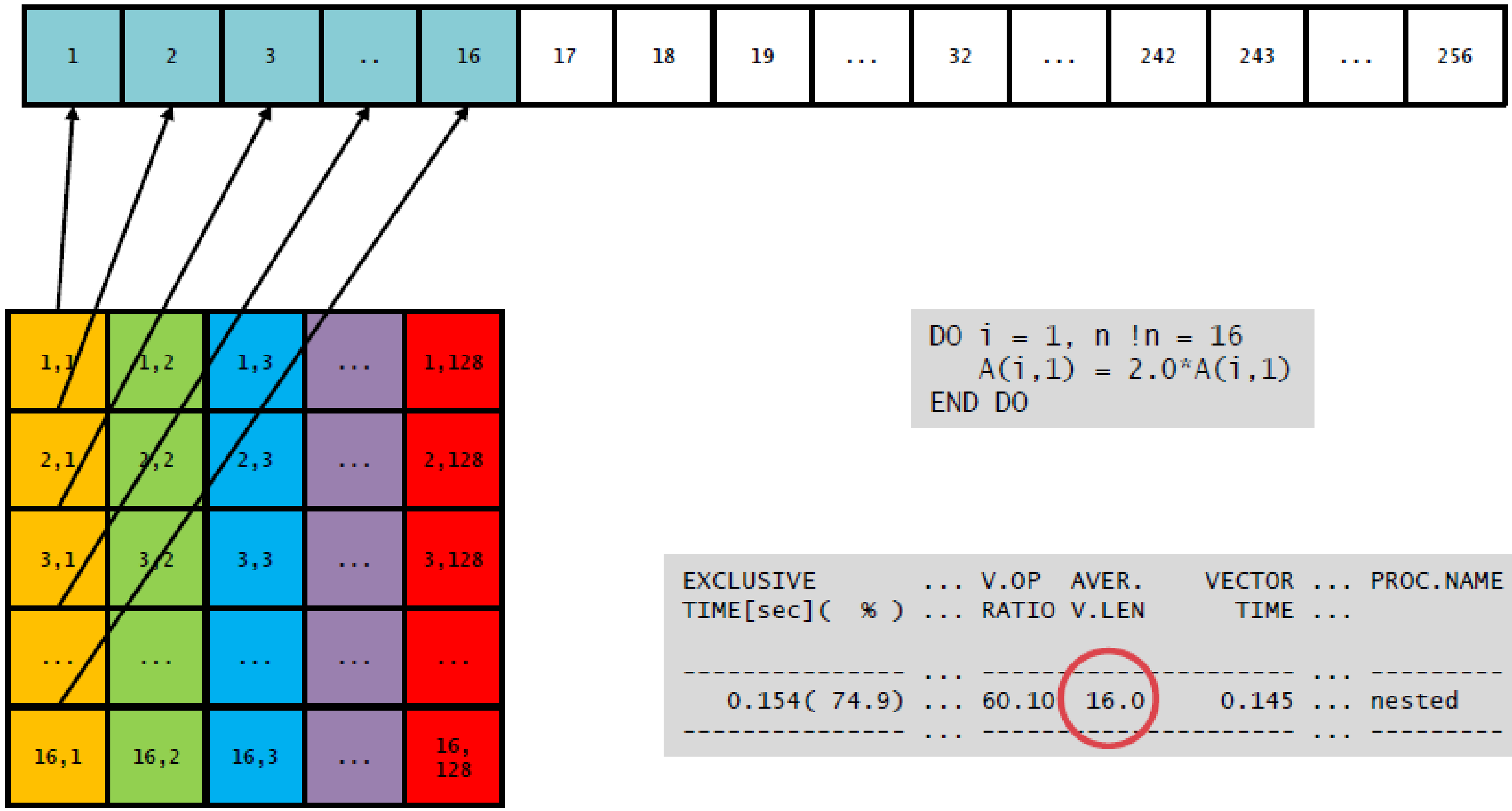
- Innermost Loop is automatically vectorized by the compiler:

```
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’s assume n = 16 ; m = 128

# Collapsing – Increasing the Vector Length

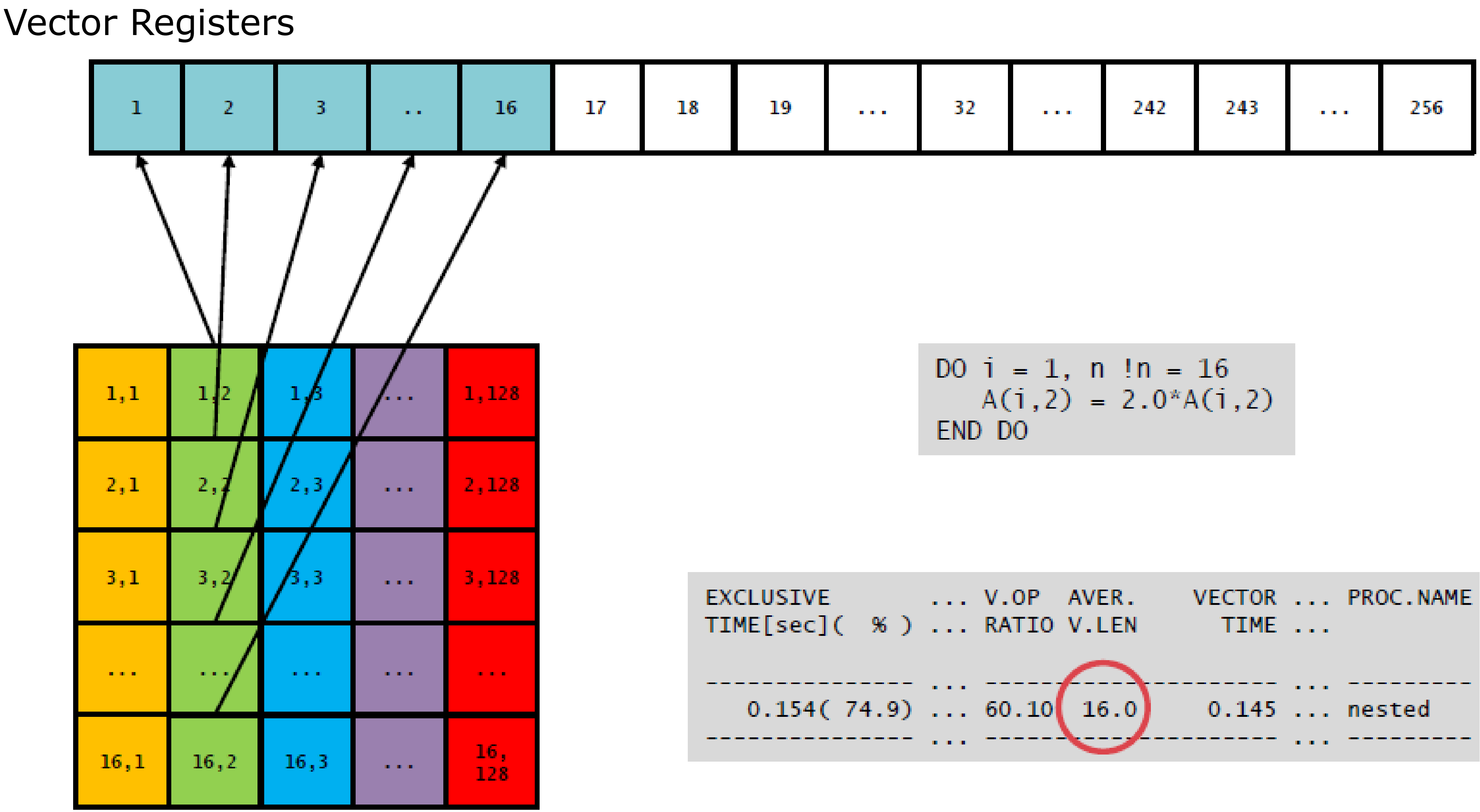
Vector Registers




```
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

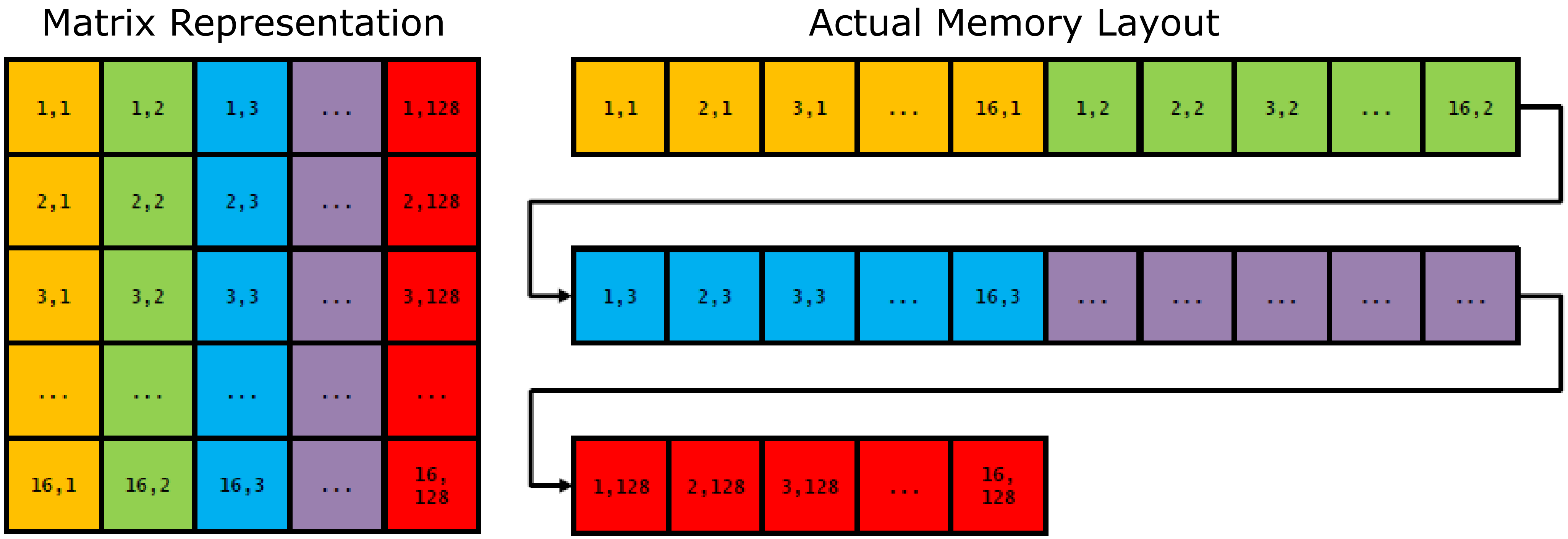



```
DO i = 1, n !n = 16
   A(i,2) = 2.0*A(i,2)
END DO
```

```
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 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)`**..!



\Orchestrating a brighter world

# Collapsing – Increasing the Vector Length

- Consider the following nested loop:

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

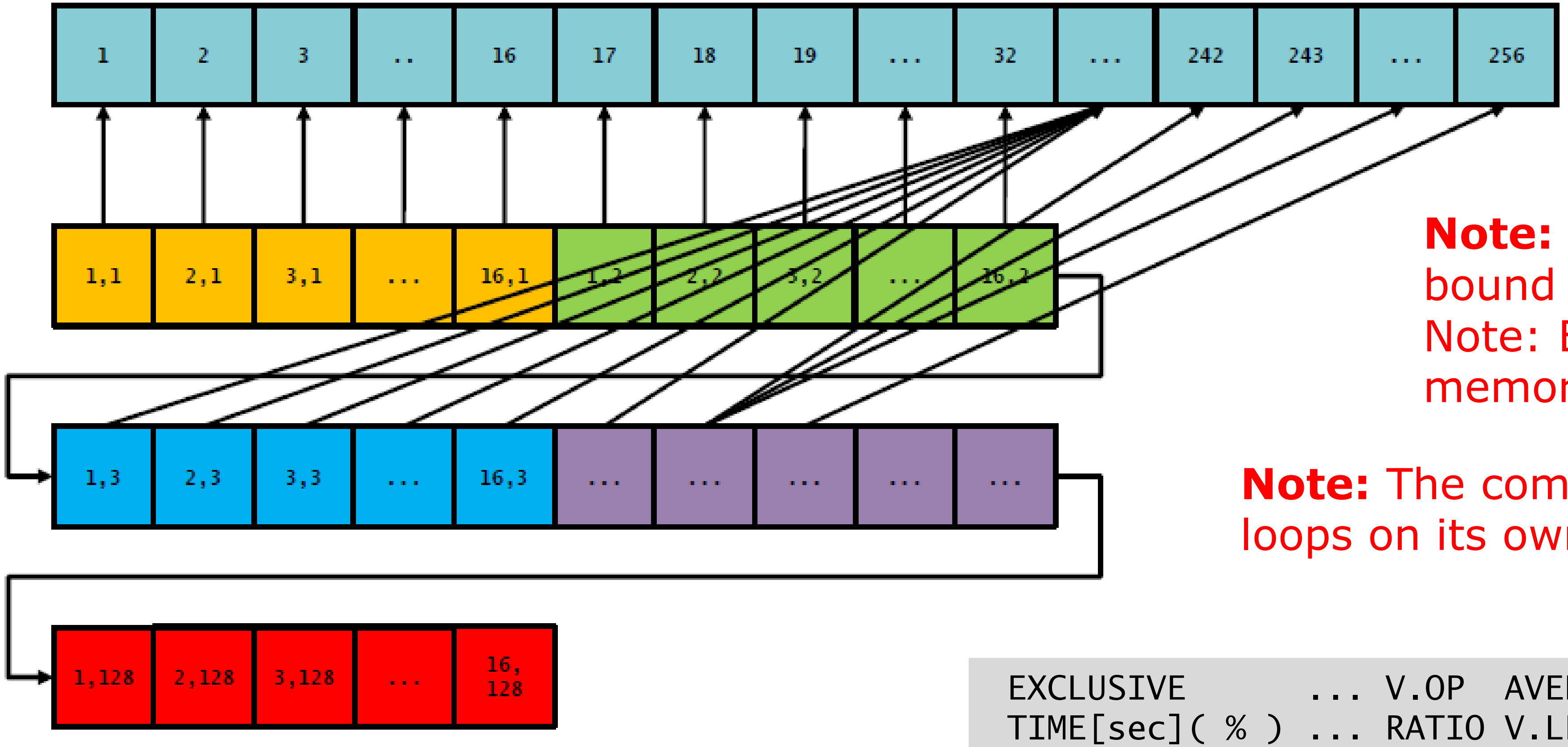
- Innermost (only) loop is vectorized:

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

Let’s assume n = 16 ; m = 128 $\rightarrow$ n*m = 2048

# Collapsing – Increasing the Vector Length

Vector Registers




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

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!

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

# Hands-on Example: Loop Unrolling

# Unrolling – Balancing the number of loads

```
!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.

# Unrolling – Balancing the number of loads

```
!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.

# Unrolling – Balancing the number of loads

```
!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.

# Unrolling – Balancing the number of loads

```
!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).

# Program Tuning Techniques

# Compiler Directives

◆ The compiler directive is to give the compiler the information that it cannot obtain from source code analysis alone to further the effects of the vectorization and parallelization, writing **`#pragma`**.

- The compiler directive format is as follows.

  **`#pragma _NEC`** *directive-name* [*clause*]

- Major vectorized compiler directives.
  - **vector/novector** : Allows [Disallows] automatic vectorization of the following loop
  - **ivdep** : Regards the unknown dependency as vectorizable dependency during the automatic vectorization.

```
#pragma _NEC ivdep
  for (i = 2 ; i < n; i++)
   {
    *p = *q + *r;
    p++, q++, r++;
  }
```

- Specify the vectorization directive option just before the loop by delimiting with the specified space.
- It works only for the loop immediately after the directive.

# Dealing with Unvectorizable Dependencies



```
ncc: vec( 103): a.c, line 16: Unvectorized loop.
ncc: vec( 113): a.c, line 16: Overhead of loop division is too large.
ncc: vec( 121): a.c, line 18: Unvectorizable dependency.
```

Such messages may be displayed to attempt partial vectorization.

It cannot be vectorized. Because compiler cannot recognizes the variable "**t**" is defined or not.

Unvectorized Loop

```
for (i=0; i<N; i++) {
   if (x[i] < s)
      t = x[i];
   else if (x[i] >= s)
      t = -x[i];
   y[i] = t;
}
```

Vectorized Loop

```
for (i=0; i<N; i++) {
   if (x[i] < s)
      t = x[i];
   else
      t = -x[i];
   y[i] = t;
}
```

Modified so that variable "**t**" is always defined.

Compiler cannot recognizes sum type macro operation

Unvectorized Loop

```
for (i=0; i<N; i++) {
   if (a[i] < 0.0)
      s = s + b[i];
   else
      s = s + c[i];
}
```

Vectorized Loop

```
for (i=0; i<N; i++) {
   if (a[i] < 0.0)
      t = b[i];
   else
      t = c[i];
   s = s + t;
}
```

Vectorization as a sum type macro operation.

<Diagnostic message after vectorization>

```
ncc: vec( 101): a.c, line 16: Vectorized loop.
ncc: vec( 126): a.c, line 21: Idiom detected.: Sum.
```

*Sum type macro operation is vectorized using special HW instruction*

# Dealing with Unvectorizable Dependencies

Raising Vectorization Ratio

```
ncc: vec( 103): vec_dep2.c, line 7: Unvectorized loop.
ncc: vec( 113): vec_dep2.c, line 7: Overhead of loop division is too large.
ncc: vec( 122): vec_dep2.c, line 8: Dependency unknown. Unvectorizable dependency is assumed.: a
```

- Specify “**ivdep**” if you know that there are no unvectorizable data dependencies in the loops, even when the compiler assumed that some unvectorizable dependencies exit.

Unvectorized Loop

```
#define N 1024
double a[N],b[N],c[N];
void func(int k, int n)
{
    int i;

    for (i=1; i < n; i++)
        a[i+k] = a[i] + b[i];
}
```

It is not vectorized because it is unknown whether the pattern of **a[i-1] = a [i]** or the pattern of **a[i + 1] = a [i]**

Vectorized Loop

```
#define N 1024
double a[N],b[N],c[N];
void func(int k, int n)
{
    int i;
#pragma _NEC ivdep
    for (i=1; i < n; i++)
        a[i+k] = a[i] + b[i];
}
```

When it is clear that the pattern is **a[i-1] = a[i]**, specify “**ivdep**” to vectorized.

<Diagnostic message after vectorization>

```
ncc: vec( 101): a.c, line 7: Vectorized loop.
```

# Dealing with Pointer Dependencies

Raising Vectorization Ratio

```
ncc: vec( 103): a.c, line 12: Unvectorized loop.
ncc: vec( 122): a.c, line 13: Dependency unknown. Unvectorizable dependency is assumed.: *(p)
```

- Specify "**ivdep**" if you know that there are no unvectorizable data dependencies in the loops, even when the compiler assumed that some unvectorizable dependencies exist.

**Vectorized Loop**

```
main() {
double *p = (double *) malloc(8*N);
double *q = (double *) malloc(8*N);
…
func(p,q);
…
}
void func(double *p, double *q) {
  …
#pragma _NEC ivdep
  for (int i = 0; i < n; i++) {
      p[i] = q[i];
  }
}
```

There is no unvectorizable dependency between **p[i]** and **q[i]** because it is an area secured separately by **malloc(3C)**, but it is not known in function "**func ()**"

It is clear to the programmer that there is no unvectorizable dependency, so you can specify "**ivdep**".

Even if "**ivdep**" is specified, the compiler ignores it and does not vectorize the loop when there is a clearly unvectorizable dependency.

***NOTE: Specifying ivdep may result in invalid results when there is a dependency that cannot be vectorized in practice***

# Equality Operator in Loop-termination-expression

**Raising Vectorization Ratio**

- When the equality operator (`==`) or the inequality operator (`!=`) appears in a loop-termination-expression, it cannot be determined whether the expression becomes true or not during the loop execution.
  - Use the relational operators `<`, `>`, `<=` or `>=` in the loop-termination-expression to vectorize the loop.

**Unvectorized Loop**

```
for (i=0; i != n; i+=2) {
    ……
}
```

The condition is not satisfied when n is an odd number

**Vectorized Loop**

```
for (i=0; i < n; i+=2) {
    ……
}
```

Fix to "i <n"

**Unvectorized Loop**

```
double *first, *last, *p;
……
for (p=first; p != last; p++)
{
    ……
}
```

C ++ iterator type array

**Vectorized Loop**

```
double *first, *last, *p;
……
for (p=first; p < last; p++)
{
    ……
}
```

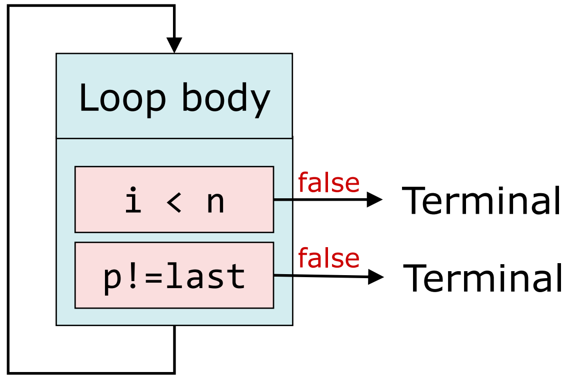
# Logical AND/OR Operator in Loop-termination-expression

**Raising Vectorization Ratio**

When a logical AND operator (**&&**) or a logical OR operator (**||**) appears in a loop-termination-expression, two branches are generated for the expression and the loop cannot be vectorized.

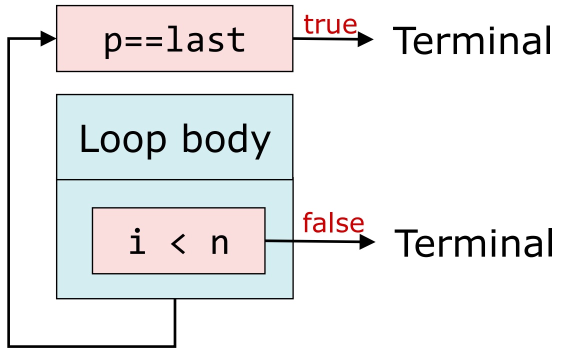
- Modify the source code so as to avoid using (**&&**) or (**||**) the loop-termination-expression.
- Part of the loop-termination-expression is moved into the loop body to remove the branch from the loop-termination-expression.

```
double func(double *first, double *last, double *a, int n)
{
    double *p = first;
    double sum = 0.0;
        /* Unvectorizable loop structure */
    for (int i = 0; i < n && p != last; i++, p++) {
        sum += a[i] * (*p);
    }
    return sum;
}
```




**Processing of loop-termination-expression**

```
double func(double *first, double *last, double *a, int n)
{
    double *p = first;
    double sum = 0.0;
        /* Vectorizable */
    for (i = 0; i < n; i++, p++) {
        if (p == last) break;
        sum += a[i] * (*p);
    }
    return sum;
}
```



# Inline Expansion: Improving Vectorization

**Raising Vectorization Ratio**

```
ncc: vec( 103): a.c, line 9: Unvectorized loop.
ncc: vec( 110): a.c, line 10: Vectorization obstructive procedure reference.: fun
```

- When a function call prevents vectorization, above messages are output
- Try to inlining with either of the following
  - Specify "**`-finline-functions`**" option
  - Specified as inline function at function declaration

```
#include <math.h>
double fun(double x, double y)
{
  return sqrt(x)*y;
}
…
for (i=0; i<N; i++) { // Unvectorized
    a[i] = fun(b[i], c[i]) + d[i];
  }
…
```

"**`double sqrt (double)`**" is vectorizable function, so it does not prevent vectorization

<When specifying inline function>

```
#include <math.h>
inline double fun(double x, double y)
{
  return sqrt(x)*y;
}
…
for (i=0; i<N; i++) {  // Vectorized
    a[i] = fun(b[i], c[i]) + d[i];
  }
…
```

<When specifying compiler option>

```
$ ncc -finline-functions a.c
```

# Outer Loop Unrolling

Raising Vectorization Ratio

## Outer loop unrolling will reduce the number of load and store operations in the inner loops.

- Unrolling the outer loop when there are multiple loop nests reduces the number of loads and stores that use only the inner loop's induction variable.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = b[i][j] + c[j];
    }
}
```

Insert **outerloop_unroll(4)** directive

```
#pragma _NEC outerloop_unroll(4)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = b[i][j] + c[j];
    }
}
```

specify $2^x$ times unrolling in parentheses.

**Program after unrolling the outer loop 4 times.**

```
for (int i = 0; i < (n%3); i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = b[i][j] + c[j];
    }
}

for (int i = (n%3); i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j]   = b[i][j]   + c[j];
        a[i+1][j] = b[i+1][j] + c[j];
        a[i+2][j] = b[i+2][j] + c[j];
        a[i+3][j] = b[i+3][j] + c[j];
    }
}
```

4 times vector operations can be performed per one vector load in array "c"

Specifying "**outerloop_unroll**" directive or "**-fouterloop-unroll**" option shortens the loop length of the outer loop (induction variable "i") and reduces the number of vector loads of the array "c".

<Message after outer loop unroll by "**outerloop_unroll**" directives>

```
ncc: opt(1592): a.c, line 3: Outer loop unrolled inside inner loop.: I
ncc: vec( 101): a.c, line 4: Vectorized loop.
```

# A Loop Contains an Array with a Vector Subscript Expression

**Raising Vectorization Ratio**

```
ncc: vec( 103): a.c, line 8: Vectorized loop.
ncc: vec( 126): a.c, line 9: Idiom detected.: List Vector
```

## Specifying **ivdep** for the list vector further improves performance

- List vector is an array with a vector subscript expression.
- When the same list vector appears on both the left and right sides of an assignment operator, it cannot be vectorized because its dependency is unknown.

Vectorized Loop (“**list_vector**” Directives)

```
#pragma _NEC list_vector
for (i=0; i < n; i++) {
    a[ix[i]] = a[ix[i]] + b[i];
}
```

Vectorized Loop (**“ivdev”** Directives)

```
#pragma _NEC ivdep
for (i = 0; i < n; i++) {
    a[ix[i]] = a[ix[i]] + b[i];
}
```

If **list_vector** is specified, the loop can be vectorized.
If the same element of array “a” is not defined twice or more in the loop, in other words, <u>if there are no duplicate values in “ix[i]”</u>, *more efficient vector instructions can be generated by specifying **ivdep** instead of **list_vector***.

<Message after vectorization by **ivdep**>

```
ncc: vec( 101): a.c, line 8: Vectorized loop.
```

# OpenMP and Automatic Parallelization

# OpenMP Parallelization

```
$ ncc -fopenmp a.c b.c
```

Specify “**`-fopenmp`**” also when linking

- International standards of directives and libraries for shared memory parallel processing
  - “NEC C/C++ Compiler for Vector Engine” supports some features up to “OpenMP Version 4.5”.
- Programming method
  - The programmer extracts a set of loops and statements that can be executed in parallel, and specifies OpenMP directives indicating how to parallelize them.
  - The compiler modifies the program based on the instruction and inserts processing for parallel processing control.
  - Compile and link with “**`-fopenmp`**”.
- Feature
  - Higher performance improvement than automatic parallelization is expected because the programmer can select and specify the parallelization part.
  - Easy to program because the compiler performs program transformation involving extraction of parallelized part, barrier synchronization and shared attribute of variables.

# Example: Writing in OpenMP C/C++

*<u>Parallelize function "**sub**" of Example 1 with OpenMP C/C ++</u>*

```
double sub (double *a, int n)
{
  int i, j;
  double b[n];
  double sum = 1.0;
#pragma omp parallel for
  for (j=0; j<n; j++)  {
     for (i=0; i<n; i++)
        sum += a[j] + b[i];
  }
  …
  return sum;
}
```

Insert OpenMP directives

Search loops that can be execute in parallel

Specifying with"-**fopenmp**". And OpenMP directives is enable.

```
$ ncc -fopenmp a.c
ncc: par(1801): ex1_omp.c, line 5: Parallel routine generated.: sub$1
ncc: par(1803): ex1_omp.c, line 6: Parallelized by "for".
ncc: vec( 101): ex1_omp.c, line 7: Vectorized loop.
```

The Compiler modifies the program so that the compiler can execute in parallel.

## The OpenMP directives follows "**#pragma omp**" to specify the parallelization method.

#pragma omp parallel for

**<u>parallel</u>**

Specify start of parallelization region

**<u>for</u>**

Specify parallelization of for loop

# Automatic Parallelization on NEC Compilers

## Program to execute in parallel in multiple threads

- Select loops and statements and extract code that can be execute in parallel.
- Generate executable code to execute in parallel with automatic parallelization or OpenMP.

*Example 1: Parallelization by automatic parallelization*

```
double sub (double *a, int n)
{
  int i, j;
  double b[n];
  double sum = 1.0;

  for (j=0; j<n; j++)  {
     for (i=0; i<n; i++)
        sum += a[j] + b[i];
  }

  return sum;
}
```

Specify "**-mparallel**" to enable automatic parallelization.

```
$ ncc -mparallel a.c
ncc: par(1801): ex1.c, line 6: Parallel routine generated.: sub$1
ncc: par(1803): ex1.c, line 6: Parallelized by "for".
ncc: vec( 101): ex1.c, line 7: Vectorized loop.
```

Vectorize the inner loop.

Extract as another function to execute the loop in parallel.

Search loops that can be execute in parallel.

Remark: Other part of loop is regarded as impossible to execute in parallel.

# Automatic Parallelization

In automatic parallelization, compiler does everything as a typical OpenMP program would do.

```
$ ncc -mparallel a.c b.c
```

Also specify **`-mparallel`** for linking.

## Compile and link with **`-mparallel`**.

- Compiler finds and parallelizes parallelizable loops and statements.
  - Automatically select loops without factors inhibiting parallelization.
  - Automatically select outermost loops in multiple loops.
    - Innermost loops should be increased speed with vectorization.

## Compiler directives to control automatic parallelization.

- Compiler directive format

  **`#pragma _NEC`** ***`directive-option`***

- Major directive options
  - **`concurrent/noconcurrent`** … parallelize/not-parallelize a loop right after this.
  - **`cncall`** … parallelize a loop including function calls.

# Parallelization Programming Available on Vector Engine

## OpenMP C/C++

- The programmer selects a set of loops and statement blocks that can be executed in parallel, and specifies OpenMP directives indicating how to parallelize them.
- The compiler transforms the program based on the instruction and inserts a directives for parallel processing control.

## Automatic parallelization

- The compiler selects loops and statement blocks that can be executed in parallel and transforms the program into parallel processing control.
- The compiler automatically performs all the work of loop detection and program modification and directives insertion of "Example 1" on the previous page.

| Programming method | Select loops / blocks | Insert directives | Program modification | Difficulty |
|---|---|---|---|---|
| OpenMP C/C++ (**-fopenmp**) | ○ | ○ | ― | High |
| Automatic parallelization (**-mparallel**) | ― | ― | ― | Low |

**○ : Manual work is needed.**
**― : Manual work is not needed because the compiler automatically executes it.**

Remark: Manual work may be needed at the time of tuning.

# Apply Both OpenMP and Automatic Parallelization

```
$ ncc –fopenmp –mparallel a.c b.c
```

## Compile and link with both **–fopenmp** and **-mparallel**.

- Automatic parallelization is applied to the loops outside of OpenMP parallel regions.
- If you don’t want to apply automatic parallelization to a routine containing OpenMP directives, specify **-mno-parallel-omp-routine**.

```
double sub (double *a, int n)
{
  int i, j;
  double b[n][n];
  double sum = 1.0;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      b[i][j] = i * j;

#pragma omp parallel for
  for (j=0; j<n; j++) {
    for (i=0; i<n; i++)
      sum += a[j] + b[i][j];
  }

  return sum;
}
```

Automatic parallelized

OpenMP parallelized

```
$ ncc –fopenmp –mparallel t.c
ncc: par(1801): t.c, line 7: Parallel routine generated.: sub$1
ncc: par(1803): t.c, line 7: Parallelized by "for".
ncc: par(1801): t.c, line 11: Parallel routine generated.: sub$2
ncc: vec( 101): t.c, line 8: Vectorized loop.
ncc: par(1803): t.c, line 12: Parallelized by "for".
ncc: vec( 101): t.c, line 13: Vectorized loop.
```

# FTRACE for parallelized programs

- Load balance in functions are shown in information for each thread.

```
REQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS      CONF  HIT E.%

    60000    62.177( 73.1)      1.036 100641.4  79931.0  99.55 248.5     62.134   0.023     0.000  100.00 funcX$1
    15000     4.467(  5.3)      0.298 107076.2  83033.3  99.47 248.4      4.455   0.005     0.000  100.00  -thread0
    15000    11.552( 13.6)      0.770 104082.7  82404.6  99.54 248.5     11.542   0.006     0.000  100.00  -thread1
    15000    19.000( 22.3)      1.267 101390.4  80683.3  99.55 248.6     18.990   0.006     0.000  100.00  -thread2
    15000    27.157( 31.9)      1.810  97595.1  77842.2  99.56 248.6     27.147   0.006     0.000  100.00  -thread3
    15000    22.711( 26.7)      1.514   1426.9      0.0   0.00   0.0      0.000   0.015     0.000    0.00 funcX
...
------------------------------------------------------------------------------------------------------------------
    79001    85.034(100.0)      1.076  74062.7  58500.4  98.89 248.5     62.249   0.043     0.000  100.00 total
```

Specify **`#pragma _NEC concurrent schedule(dynamic, 4)`** right before an outermost loop

```
REQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS      CONF  HIT E.%

    60000    66.872( 99.6)      1.115  93599.2  74318.7  99.52 248.5     64.077   1.418     0.000  100.00 funcX$1
    15000    16.766( 25.0)      1.118  92992.0  73842.7  99.52 248.5     16.022   0.409     0.000  100.00  -thread0
    15000    16.697( 24.9)      1.113  91671.0  72790.7  99.52 248.5     16.000   0.397     0.000  100.00  -thread1
    15000    16.714( 24.9)      1.114  94854.7  75312.8  99.52 248.5     16.040   0.305     0.000  100.00  -thread2
    15000    16.695( 24.9)      1.113  94880.7  75329.6  99.51 248.5     16.014   0.307     0.000  100.00  -thread3
    15000     0.129(  0.2)      0.009   1284.5      0.1   0.00   0.0      0.000   0.010     0.000    0.00 funcX
...
------------------------------------------------------------------------------------------------------------------
    79001    67.148(100.0)      0.850  93334.5  74082.8  99.51 248.5     64.192   1.430     0.000  100.00 total
```

Before :EXCLUSIVE TIME are ununiform for **`-thread0`** to **`-thread3`** of **`funcX$1`**.(Load imbalance)
After :EXCLUSIVE TIME are uniform for each threads and that of **`funcX`** is shorter(time for barrier sync and so on reduced) although that of **`funcX$1`** increases because of time to control threads.

# MPI Parallelization

# Compiling and Linking MPI Programs

- It is possible to compile and link MPI programs with the MPI compilation commands corresponding to each programing language.

```
$ source /opt/nec/ve/mpi/x.x.x/bin/necmpivars.sh
```

```
$ mpincc   a.c
```

```
$ mpinc++  a.cpp
```

```
$ mpinfort a.f90
```

- Use the option –*compiler* to specify a specific version of the C/C++ or Fortran compiler for compilation

```
$ mpinfort -compiler /opt/nec/ve/bin/nfort-5.0.0 program.f90
```

# Compiling and Linking Hybrid MPI Programs

- By using the NEC MPI/Scalar-Vector Hybrid, you can perform a communication among processes on VH or scalar nodes and those on VE nodes

```
$ source /opt/nec/ve/mpi/x.x.x/bin/necmpivars.sh
```

**NEC Compiler:**

```
$ mpincc   a.c
$ mpinc++  a.cpp
$ mpinfort a.f90
```

**GNU Compiler:**

```
(setup the GNU compiler (e.g., PATH, LD_LIBRARY_PATH)
$ mpincc   -vh a.c
$ mpinc++  -vh a.cpp
$ mpinfort -vh a.f90
```

# Compiling and Linking MPI Programs

- Using the NEC MPI compiler wrappers, it is easy to compile simple MPI code.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
   MPI_Init(&argc, &argv);
   int my_rank, namel;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Get_processor_name(name, &namel);
   printf("Process %2d is running on %s\n",
          my_rank, name);
   MPI_Finalize();
   return 0;
}
```

```
$ mpincc -o mpi_VE mpi.c
```

# Compiling and Linking MPI Programs

- We can use the same compiler wrappers to compile the program for the vector host.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
   MPI_Init(&argc, &argv);
   int my_rank, namel;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Get_processor_name(name, &namel);
   printf("Process %2d is running on %s\n",
          my_rank, name);
   MPI_Finalize();
   return 0;
}
```

```
$ export NMPI_CC_H=gcc
$ mpincc -vh -o mpi_VH mpi.c
```

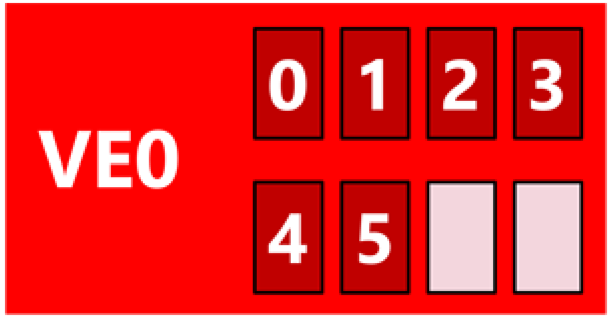
# Executing the MPI Programs

- -np, -n, -c determine the number of processes to create, -v prints out process placement

```
$ mpirun -np 6 -v ./mpi_VE
```

- Output of the above execution command.

```
mpid: Creating 6 processes of './mpi_VE' on VE 0 of local host
node0
Process  0 is running on node0, ve id 0
Process  1 is running on node0, ve id 0
Process  2 is running on node0, ve id 0
Process  3 is running on node0, ve id 0
Process  4 is running on node0, ve id 0
Process  5 is running on node0, ve id 0
```




Processes are parallelized over 6 cores of the same Vector Engine card.

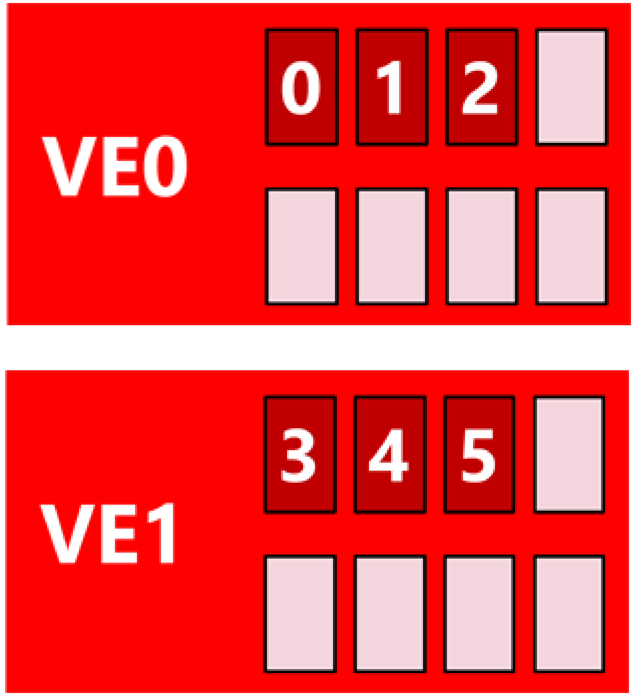
# Executing the MPI Programs

- -vennp, -ve_nnp –nnp_ve determine the number of processes per VE

```
$ mpirun –ve 0-1 –vennp 3 ./mpi_VE
```

- Output of the above execution command.

```
Process  0 is running on node0, ve id 0
Process  1 is running on node0, ve id 0
Process  2 is running on node0, ve id 0
Process  3 is running on node0, ve id 1
Process  4 is running on node0, ve id 1
Process  5 is running on node0, ve id 1
```




Processes are parallelized over 3 cores of two Vector Engine cards each.

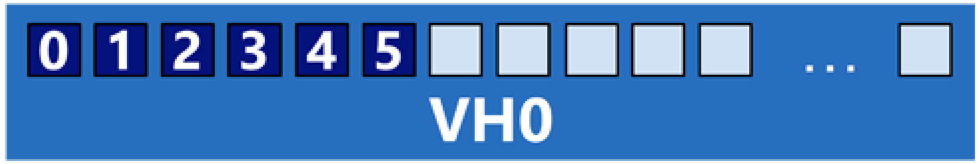
# Executing the MPI Programs

- -vh executes the program on the VH. It needs to be compiled for the target architecture

```
$ mpirun -vh 0-1 -np 6 ./mpi_VH
```

- Output of the above execution command.

```
Process  0 is running on node0, VH
Process  1 is running on node0, VH
Process  2 is running on node0, VH
Process  3 is running on node0, VH
Process  4 is running on node0, VH
Process  5 is running on node0, VH
```




Processes are parallelized over 6 cores of the Vector Host CPU.

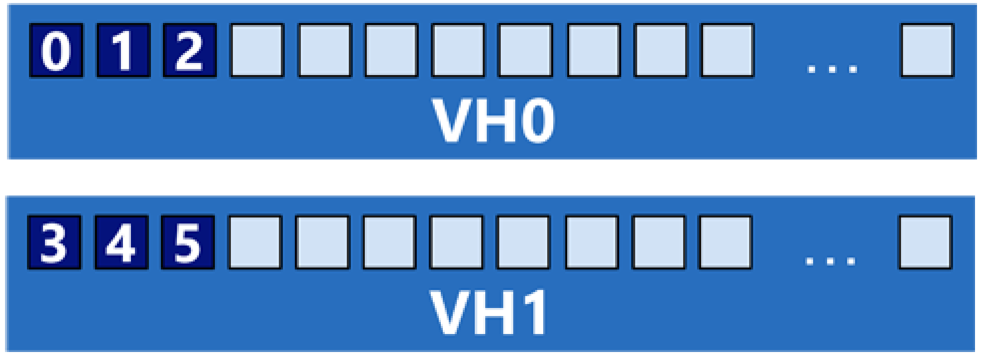
# Executing the MPI Programs

- -nnp, -ppn –npernode, -N determine the number of processes per VH

```
$ mpirun -vh -nnp 3 ./mpi_VH
```

- Output of the above execution command.

```
Process  0 is running on node0, VH
Process  1 is running on node0, VH
Process  2 is running on node0, VH
Process  3 is running on node1, VH
Process  4 is running on node1, VH
Process  5 is running on node1, VH
```




Processes are parallelized across CPUs of two Vector Hosts over 3 cores of each host.

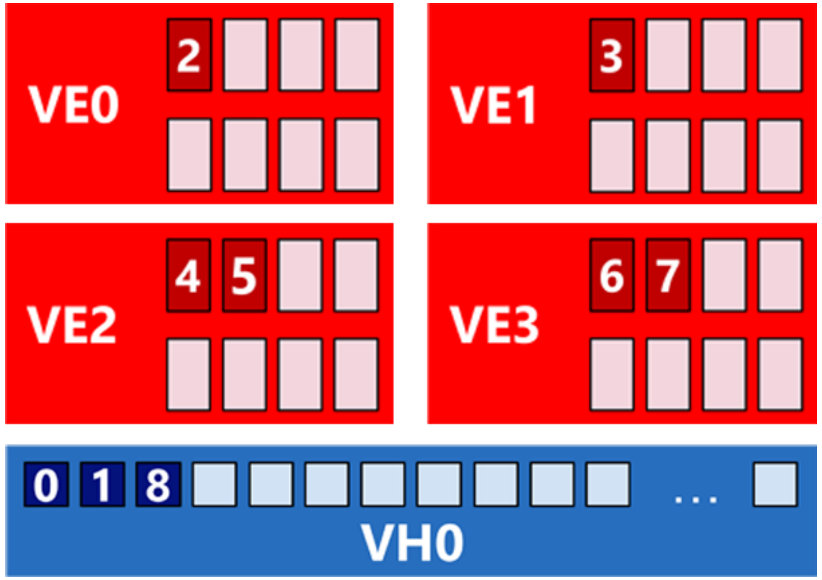
# Executing the MPI Programs

- VE/VH hybrid execution is easily achieved by chaining VE and VH commands for the corresponding executables

```
$ mpirun  -vh          -np 2     ./mpi_VH : \
              -ve 0-1 -vennp 1  ./mpi_VE : \
              -ve 2-3 -vennp 2  ./mpi_VE : \
          -vh          -np 1     ./mpi_VH
```

- Output of the above execution command.

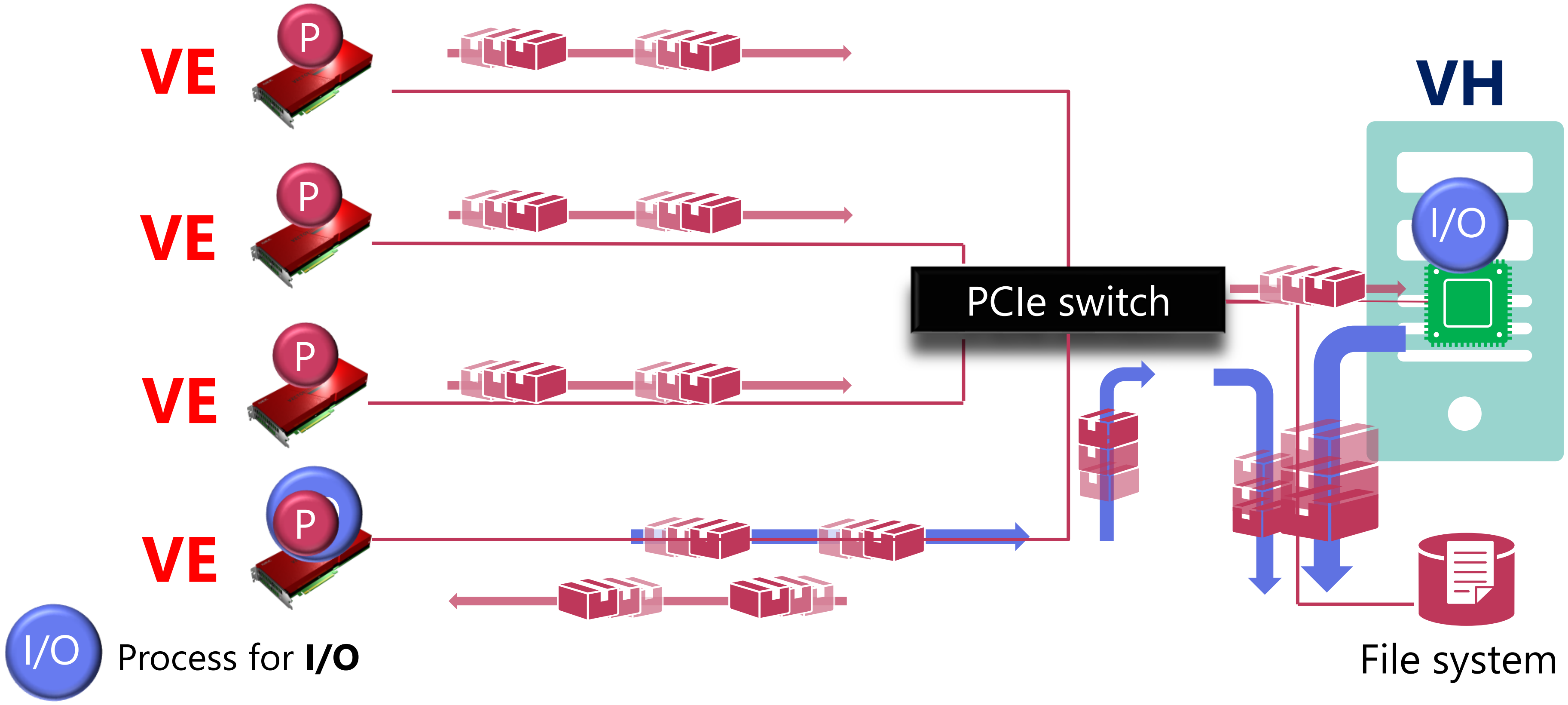
```
Process  0 is running on node0, VH
Process  1 is running on node0, VH
Process  2 is running on node0, ve id 0
Process  3 is running on node0, ve id 1
Process  4 is running on node0, ve id 2

Process  5 is running on node0, ve id 2
Process  6 is running on node0, ve id 3
Process  7 is running on node0, ve id 3
Process  8 is running on node0, VH
```




A hybrid execution over CPU and VE architectures under the same MPI execution.

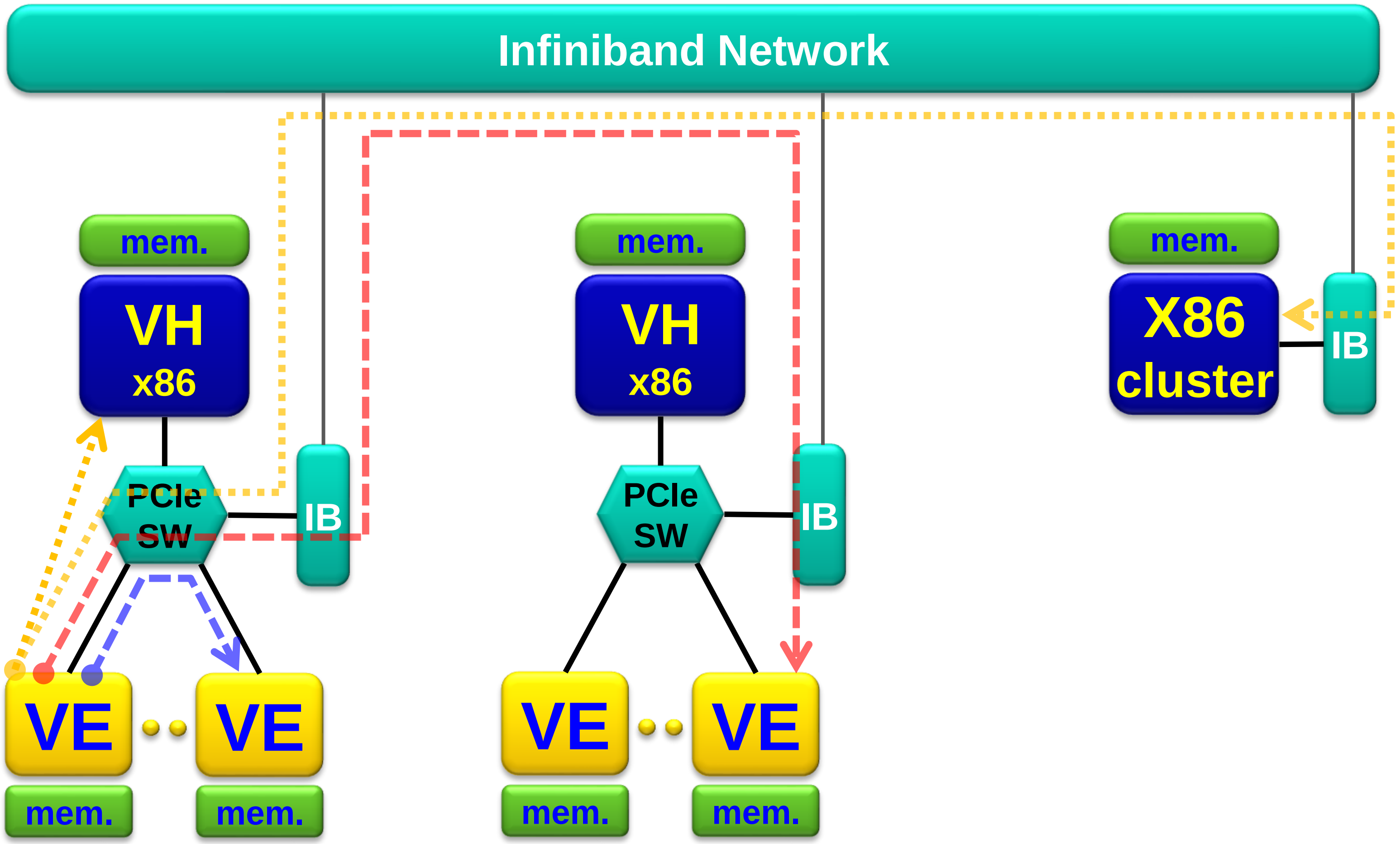
# Offload I/O using Hybrid MPI

- Offload I/O processes on VH using Hybrid MPI and continue the computations on VE



# MPI communication

- Direct communications between VEs (no x86 involved and RDMA)
- Hybrid mode with processes on host CPU and Vector Engine processors



# Running the MPI Programs

- Execution on one VE.
  - Execution of an MPI program on VE#3 on local VH using 4 processes

```
$ mpirun -ve 3 -np 4 ./ve.out
```

- Execution on multiple VEs on a VH
  - Execution of an MPI program on from VE#0 through VE#7 on local VH using 16 processes in total (2 processes per VE).

```
$ mpirun -ve 0-7 -np 16 ./ve.out
```

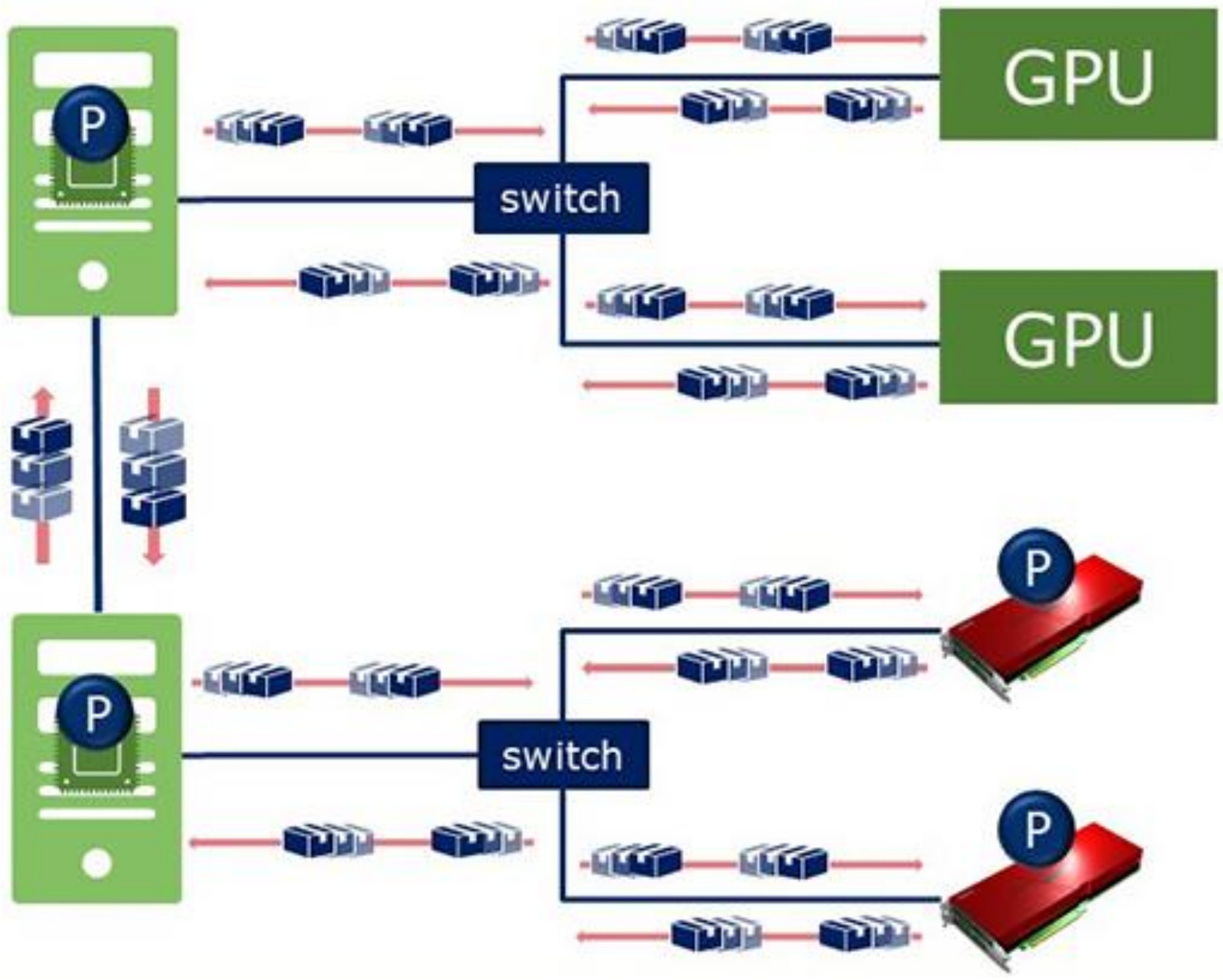
- Execution on multiple VEs on multiple VHs
  - Execution of an MPI program on VE#0 and VE#1 on each of two VHs (host1 and host2), using 32 processes in total (8 processes per VE).

```
$ mpirun -hosts host1,host2 -ve 0-1 -np 32 ./ve.out
```

- Hybrid Execution on a VH and on multiple VEs
  - Hybrid Execution of vh.out on VH host1 using 8 processes and ve.out on VE#0 and VE#1 on VH host1 using 16 processes in total (8 processes per VE).

```
$ mpirun -vh -host host1 -np 8 vh.out : -host host1 -ve 0-1 -np 16 ./ve.out
```

# Hybrid MPI: GPGPU




- MPI communication between GPU cluster and Aurora cluster is also possible.
- Application performance is maximized by allocating appropriate resources (VH [CPU], VE and GPU), based on the compute.
- By using Hybrid MPI, all computational resources can be utilized even for hybrid systems with a mix of different architectures.

# Examples

# NEC Optimized Quantum Espresso v7.1

- **NEC SX-Aurora TSUBASA Optimized QE**
  - OSS code: Quantum ESPRESSO
  - Quantum ESPRESSO is one of the major applications in materials science.
  - QE is widely used as a first-principle calculation application
  - This version can be downloaded from: [GitHub - SX-Aurora/qe-ve: QuantumEspresso electronic structure calculations and materials modeling optimized for SX-Aurora TSUBASA Vector Engine]

```
$ git clone https://github.com/SX-Aurora/qe-ve.git
```

- **Download Quantum Espresso from:**

  https://www.quantum-espresso.org/ | Release package: qe-7.1-ReleasePack.tgz

```
$ wget https://www.quantum-espresso.org/rdm-download/488/v7-
1/468ef2db4d26294ab85c0d299d0dab3f/qe-7.1-ReleasePack.tar.gz
```

- **Use ELPA (Eigenvalue soLvers for Petaflop Applications)**

  https://elpa.mpcdf.mpg.de

```
$ wget https://gitlab.mpcdf.mpg.de/elpa/elpa/-
/archive/new_release_2022.05.001/elpa-new_release_2022.05.001.tar.gz
```

# NEC Optimized Quantum Espresso v7.1

- A naïve download script is available on:

```
/scratch/training/nec/hpc/demo/download-packages.sh
```

- All packages can be downloaded with the following commands:

```
$ cd $SCRATCH
$ mkdir nec-qe-demo
$ cd nec-qe-demo
$ cp /scratch/training/nec/hpc/demo/download-packages.sh .
$ ./download-packages.sh
```

- Alternately, all packages have been downloaded and available on:

```
/scratch/training/nec/hpc/demo/packages
```

- Bring them to your user directory:

```
$ cd $SCRATCH
$ mkdir nec-qe-demo
$ cd nec-qe-demo
$ cp -r /scratch/training/nec/hpc/demo/packages/* .
```

# Steps to Build Quantum Espresso (pw.x)

## ◆ Setup ELPA library

```
$ tar -zxvf elpa-new_release_2022.05.001.tar.gz
$ mv elpa-new_release_2022.05.001 elpa
$ cd elpa
$ mkdir elpa-install
$ export ELPADIR=<current ELPA directory>/elpa-install (use `pwd` command to find the path)
$ ./autogen.sh
$ ./conf_ELPA.sh
$ make
$ make install
$ cp modules/*.mod private_modules/*.mod ${ELPADIR}/include/elpa-2022.05.001/modules
```

# Steps to Build Quantum Espresso (pw.x)

## ◆ Build Quantum Espresso

```
$ tar -zxvf qe-7.1-ReleasePack.tar.gz
$ mv build_qe-7.1.sh patch_qe-7.1 qe-7.1/
$ cd qe-7.1/external/
$ ./initialize_external_repos.sh
$ cd ../
$ patch -p 1 < patch_qe-7.1
$ ./build_qe-7.1.sh
```

# Steps to run Quantum Espresso (pw.x)

- Benchmark datasets can be downloaded from github as:

```
$ git clone https://github.com/QEF/benchmarks
```

  while the downloaded package has several datasets, we will use AUSURF112 for the demo.

- For ease, it has been downloaded and kept on the path:

```
/scratch/training/nec/hpc/demo/packages/qe-ve/AUSURF112
```

- To run the benchmark, we need access to **pw.x** and a run script.

```
export VE_PROGINF=DETAIL
export MPIPROGINF=DETAIL
export MPISEPSELECT=4
export VE_TRACEBACK=FULL
export OMP_NUM_THREADS=1

mpirun -ve 0 -np 8 /opt/nec/ve/bin/mpisep.sh ./pw.x -npool 2 -nband 1 -ntg 1 -
ndiag 4 -input ausurf.in
```

- Review the output files and logs in std:0.* files.

# NEC Vector Engine Knowledge base

## NEC Compiler user manuals

- C/C++ Compiler: https://www.hpc.nec/documents/sdk/pdfs/g2af01e-C++UsersGuide-019.pdf
- Fortran Compiler: https://www.hpc.nec/documents/sdk/pdfs/g2af02e-FortranUsersGuide-019.pdf

## Detailed tuning guide for the Vector Engine

- https://www.hpc.nec/forums/topic?id=pwdcB9

## Vectorization training with examples

- https://www.hpc.nec/forums/topic?id=p8kc9Z

\Orchestrating a brighter world

NEC

## \Orchestrating a brighter world

NEC creates the social values of safety, security, fairness and efficiency to promote a more sustainable world where everyone has the chance to reach their full potential.