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

1. Introduction to Vector Architecture
2. SDK and compiler features
3. Vectorization on NEC Vector Engine
4. Demonstration of Examples
5. OpenMP and Automatic Thread Parallelization
6. MPI Parallelization
7. Case Study
8. Q & A
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

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

```fortran
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 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.
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)
    - \( C = A + B \)
    - \( C = A - B \)
    - \( C = A \times B \)
  - FMA (Fused Multiply-Add)
    - \( D = A + B \times C \)
  - Division
    - \( C = A / B \)
  - Square root
    - \( B = \sqrt{A} \)
  - Other operations are combinations of standard operations
    - \( \sin, \cos, \tan, \arctan, \ldots \)
    - \( A^{**B}, \exp, \log, \ldots \)
  - \ldots
## Vector Length of VE vs modern SIMD

<table>
<thead>
<tr>
<th></th>
<th>512bit SIMD</th>
<th>Vector (256 vector length)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data size/operation</strong></td>
<td>512bit</td>
<td>16384bit 32x larger</td>
</tr>
<tr>
<td></td>
<td>8B x 8 elements</td>
<td>256 elements</td>
</tr>
<tr>
<td></td>
<td>8 elements</td>
<td>8B x 32 elements</td>
</tr>
<tr>
<td><strong>Cycles/operation</strong></td>
<td>1 cycle</td>
<td>8 cycle</td>
</tr>
<tr>
<td><strong>Processing speed</strong></td>
<td>512bit/cycle</td>
<td>2048bit/cycle 4x larger</td>
</tr>
</tbody>
</table>
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

**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!
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++

- **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

- **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)

◆ Hybrid programming
  ■ VE offloading
  ■ Reverse offloading
  ■ Hybrid MPI
  ■ OpenMP target (coming soon)

◆ Tools
  ■ GNU profiler
  ■ GNU debugger
  ■ TAU tool
  ■ NEC profilers : FtraceViewer, PROGINF, vftrace
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.

- **BLAS / CBLAS**
  - Basic linear algebra subprograms

- **LAPACK**
  - Linear algebra package

- **ScaLAPACK**
  - Scalable linear algebra package for distributed memory parallel programs

- **SBLAS**
  - Sparse BLAS

- **Stencil Code Accelerator**
  - Stencil Code Acceleration (SCA)

- **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
```

Options to control the level of automatic vectorization and optimization.

- `-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

Compilers from the NEC SDK for the Vector Engine.

- `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
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

- Executing a compiled program:
  ```
  $ nfort a.f90 b.cf90
  $ ./a.out
  ```

- Executing a program getting input file and parameter from command line:
  ```
  $ ./b.out data1.in
  ```

- Executing with redirecting an input file instead of standard input file:
  ```
  $ ./c.out < data2.in
  ```

- Executing with specifying the index of VE:
  ```
  $ env VE_NODE_NUMBER=1 ./a.out
  ```

- Executing a parallelized program with specifying the number of threads:
  ```
  $ nfort -mparallel -O3 a.f90 b.f90
  $ export OMP_NUM_THREADS=4
  $ ./a.out
  ```

- Executing a compiled MPI program:
  ```
  $ 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 on different number of VE cards:
  ```
  -- 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**

<table>
<thead>
<tr>
<th>A(1)</th>
<th>= B(1) + C(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(2)</td>
<td>= B(2) + C(2)</td>
</tr>
<tr>
<td>A(3)</td>
<td>= B(3) + C(3)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>A(100)</td>
<td>= B(100) + C(100)</td>
</tr>
</tbody>
</table>

Execute one calculation 100 times

**Execution image of vector instructions**

<table>
<thead>
<tr>
<th>DO I = 1, 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(I) = B(I) + C(I)</td>
</tr>
</tbody>
</table>

Execute 100 calculation at once

At most 256 calculation at once
Order of Hardware Instructions

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

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

Array “b”

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

Note that the order of addition has changed.
("b[1]+c[1]" is added faster than "e[0]+f[0]")
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.

**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.

**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.

```
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
Calculation order in vector

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

```
a[0] = s ;
a[1] = s ;
: 
s = b[0] + c[0] ;
s = b[1] + c[1] ;
: 
```
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 \( p = \&a[3], \ q = \&a[2] \)

Ex 2: Can be vectorized when \( p=\&a[1], \ q=\&a[2] \)

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 dependencies.

```c
for (i = 2 ; i < n; i++) {
    *p = *q + *r;
    p++, q++, r++;
}
```
Conditional branches (if statements) are also vectorized.

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

**Execute with vector operations**

```c
... ...
mask[100] = a[100] < b[100]
```

```c
... ...
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

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

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

```
DO i = 1, n
   A(i) = inlinable_fkt(B(i))
   A(i) = SQRT(A(i))
END DO
```

![This does not vectorize](red)

```
DO i = 1, n
   CALL very_long_routine(A(i))
   A(i) = SQRT(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 vectorizable data structures (e.g. strings)

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

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

**Source Program**
```c
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**
```c
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];
```
Loop Interchange

Source Program

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

```c
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

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

```c
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.
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

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.

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

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

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

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

Sum

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

Iteration

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

Maximum or minimum values

```c
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

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.

Source Program

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

Transformation Image

```c
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.
Loop Fusion

- 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

When the `-finline-functions` option is specified, the compiler expands 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.

Source Program

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

Transformation Image

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

<table>
<thead>
<tr>
<th>LINE</th>
<th>DIAGNOSTIC MESSAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1181</td>
<td>vec(103): Unvectorized loop.</td>
</tr>
<tr>
<td>1181</td>
<td>vec(113): Unvectorizable dependency is assumed.: *(p)</td>
</tr>
<tr>
<td>1234</td>
<td>vec(102): Partially vectorized loop.</td>
</tr>
<tr>
<td>1485</td>
<td>vec(101): Vectorized loop.</td>
</tr>
</tbody>
</table>
...
```

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

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++) {
    for (i=0; i<m; i++) {
        idx = j*m+i;
    }
}

For (i = 0; i < n; i++) {
    : :
    : :
    : :
}

For (j=0; j<n; j++) {
    for (i=0; i<m; i++) {
        : :
        : :
        : :
    }
}

For (i=0; i<n; i++) {
    : :
    : :
}

For (j=0; j<n; j++) {
    : :
    : :
    : :
}

For (i=0; i<m; i++) {
    : :
    : :
    : :
}

$ncc \text{ -report-format abc.c}$
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

```bash
$ ncc -O4 a.c b.c c.c
$ ls a.out
a.out
$ export VE_PROGINF=DETAIL
$ ./a.out

******** Program Information ********

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Time (sec)</td>
<td>11.329254</td>
</tr>
<tr>
<td>User Time (sec)</td>
<td>11.323691</td>
</tr>
<tr>
<td>Vector Time (sec)</td>
<td>11.012581</td>
</tr>
<tr>
<td>Inst. Count</td>
<td>6206113403</td>
</tr>
<tr>
<td>V. Inst. Count</td>
<td>2653887022</td>
</tr>
<tr>
<td>V. Element Count</td>
<td>619700067996</td>
</tr>
<tr>
<td>V. Load Element Count</td>
<td>53789940198</td>
</tr>
<tr>
<td>FLOP count</td>
<td>576929115066</td>
</tr>
<tr>
<td>MOPS</td>
<td>73492.138481</td>
</tr>
<tr>
<td>MOPS (Real)</td>
<td>73417.293683</td>
</tr>
<tr>
<td>MFLOPS</td>
<td>50976.512081</td>
</tr>
<tr>
<td>MFLOPS (Real)</td>
<td>50924.597321</td>
</tr>
<tr>
<td>A. V. Length</td>
<td>233.506575</td>
</tr>
<tr>
<td>V. Op. Ratio (%)</td>
<td>99.572922</td>
</tr>
<tr>
<td>L1 Cache Miss (sec)</td>
<td>0.010847</td>
</tr>
<tr>
<td>CPU Port Conf. (sec)</td>
<td>0.000000</td>
</tr>
<tr>
<td>V. Arith. Exec. (sec)</td>
<td>8.406444</td>
</tr>
<tr>
<td>V. Load Exec. (sec)</td>
<td>1.384491</td>
</tr>
<tr>
<td>VLD LLC Hit Element Ratio (%)</td>
<td>100.000000</td>
</tr>
<tr>
<td>Power Throttling (sec)</td>
<td>0.000000</td>
</tr>
<tr>
<td>Thermal Throttling (sec)</td>
<td>0.000000</td>
</tr>
<tr>
<td>Max Active Threads</td>
<td>1</td>
</tr>
<tr>
<td>Available CPU Cores</td>
<td>8</td>
</tr>
<tr>
<td>Average CPU Cores Used</td>
<td>0.999509</td>
</tr>
<tr>
<td>Memory Size Used (MB)</td>
<td>204.000000</td>
</tr>
</tbody>
</table>
```

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

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

### Time information

- Real Time
- User Time
- Vector Time
- Instruction Count
- Vector Instruction Count
- Vector Element Count
- Vector Load Element Count
- FLOP count
- MOPS
- MOPS (Real)
- MFLOPS
- MFLOPS (Real)

### Number of instruction executions

- Instruction Count
- Vector Instruction Count
- Vector Element Count
- Vector Load Element Count
- FLOP count
- MOPS
- MOPS (Real)
- MFLOPS
- MFLOPS (Real)

### Vectorization, memory and parallelization information

- A. V. Length
- V. Op. Ratio (%)
FTRACE

Performance information of each function

<table>
<thead>
<tr>
<th>FREQUENCY</th>
<th>EXCLUSIVE</th>
<th>AVER.TIME</th>
<th>MOPS</th>
<th>MFLOPS</th>
<th>V.OP</th>
<th>AVER.TIME</th>
<th>VECTOR</th>
<th>L1CACHE</th>
<th>CPU</th>
<th>PORT</th>
<th>VLD LLC</th>
<th>PROC.NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>15000</td>
<td>4.762(42.7)</td>
<td>0.317</td>
<td>77117.2</td>
<td>62034.6</td>
<td>99.45</td>
<td>251.0</td>
<td>4.605</td>
<td>0.002</td>
<td>0.000</td>
<td>100.0</td>
<td></td>
<td>funcA</td>
</tr>
<tr>
<td>15000</td>
<td>3.541(31.7)</td>
<td>0.236</td>
<td>73510.3</td>
<td>56944.5</td>
<td>99.46</td>
<td>216.0</td>
<td>3.554</td>
<td>0.000</td>
<td>0.000</td>
<td>100.0</td>
<td></td>
<td>funcB</td>
</tr>
<tr>
<td>15000</td>
<td>2.726(24.4)</td>
<td>0.182</td>
<td>71930.2</td>
<td>27556.5</td>
<td>99.43</td>
<td>230.8</td>
<td>2.725</td>
<td>0.000</td>
<td>0.000</td>
<td>100.0</td>
<td></td>
<td>funcC</td>
</tr>
<tr>
<td>1</td>
<td>0.134(1.2)</td>
<td>133.700</td>
<td>60368.8</td>
<td>35641.2</td>
<td>98.53</td>
<td>214.9</td>
<td>0.118</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>45001</td>
<td></td>
<td>11.163(100.0)</td>
<td>0.248</td>
<td>74505.7</td>
<td>51683.9</td>
<td>94.44</td>
<td>233.5</td>
<td>11.002</td>
<td>0.002</td>
<td>0.000</td>
<td>100.0</td>
<td>total</td>
</tr>
</tbody>
</table>

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

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

α: Vectorization ratio
Ts: Scalar execution time
Tv: Vector execution time

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

**Average vector length**

The average number of data 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 low, 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
**Sample Report**

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

- **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.

<table>
<thead>
<tr>
<th>FREQUENCY</th>
<th>EXCLUSIVE TIME<a href="%25">sec</a></th>
<th>AVER.TIME [msec]</th>
<th>MOPS</th>
<th>MFLOPS</th>
<th>V.OP RATIO</th>
<th>AVER. V.LEN</th>
<th>VECTOR</th>
<th>L1CACHE TIME</th>
<th>MISS</th>
<th>CPU PORT</th>
<th>VLD</th>
<th>LLC</th>
<th>PROC.NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>15000</td>
<td>4.767( 42.7)</td>
<td>0.318</td>
<td>77030.2</td>
<td>61964.6</td>
<td>99.45</td>
<td>251.0</td>
<td>4.610</td>
<td>0.002</td>
<td>0.000</td>
<td>100.00</td>
<td></td>
<td></td>
<td><code>funcA</code></td>
</tr>
<tr>
<td>15000</td>
<td>3.541( 31.7)</td>
<td>0.236</td>
<td>73505.6</td>
<td>56940.8</td>
<td>99.46</td>
<td>216.0</td>
<td>3.555</td>
<td>0.000</td>
<td>0.000</td>
<td>100.00</td>
<td></td>
<td></td>
<td><code>funcB</code></td>
</tr>
<tr>
<td>15000</td>
<td>2.726( 24.4)</td>
<td>0.182</td>
<td>71930.1</td>
<td>27556.5</td>
<td>99.43</td>
<td>230.8</td>
<td>2.725</td>
<td>0.000</td>
<td>0.000</td>
<td>100.00</td>
<td></td>
<td></td>
<td><code>funcC</code></td>
</tr>
<tr>
<td>1</td>
<td>0.134(  1.2)</td>
<td>133.700</td>
<td>60368.9</td>
<td>35641.3</td>
<td>98.53</td>
<td>214.9</td>
<td>0.118</td>
<td>0.000</td>
<td>0.000</td>
<td>0.00</td>
<td></td>
<td></td>
<td><code>main</code></td>
</tr>
<tr>
<td>45001</td>
<td>11.168(100.0)</td>
<td>0.248</td>
<td>74468.3</td>
<td>51657.9</td>
<td>99.44</td>
<td>233.5</td>
<td>11.008</td>
<td>0.002</td>
<td>0.000</td>
<td>100.00</td>
<td></td>
<td></td>
<td><code>total</code></td>
</tr>
</tbody>
</table>
Debugging
Traceback Information

Compile and link with `-traceback`.
Set the environment variable “VE_TRACEBACK” to “FULL” or “ALL” at execution.

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

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”.

Occur “divide-by-zero”

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 ...

- 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

```bash
$ /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
...```

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

Optimal memory access.

Example: 
$$A(i) = B(i)$$
Memory Access in Vector Computers

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

Not optimal due to partially used cache lines.

Example:

\[
\text{DO } i = 1, n, 2 \\
A(i) = B(i)
\]
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

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

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

Vector Register
X X X X X X

Cache & Memory

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

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

Example: \[ A(i) = B(idx(i)) \]
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: 
\[ A(\text{idx}(i)) = B(i) \]

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

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

Example:

\[
A = A + B(i)
\]

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(1,1)
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(1,2) = 2.0*A(1,2)
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

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 nested loop:

\[
\begin{align*}
&\text{DO } i = 1, n*m \\
&A(i,1) = 2.0*A(i,1) \\
&\text{END DO}
\end{align*}
\]

◆ 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

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

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!
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.
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

- 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.

```fortran
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
```
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 \texttt{#pragma}.

- The compiler directive format is as follows.
  \texttt{#pragma _NEC directive-name [clause]}

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

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

- 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

Unvectorized Loop

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

Unvectorizable Loop

It cannot be vectorized. Because compiler cannot recognizes the variable “t” is defined or not.

Unvectoritized Loop

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

Unvectorized Loop

Compiler cannot recognizes sum type macro operation

Vectorized Loop

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

Vectorized Loop

Modified so that variable “t” is always defined.

Vectorization as a sum type macro operation.

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

Sum type macro operation is vectorized using special HW instruction

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

<Diagnostic message after vectorization>

```
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.

Sum type macro operation is vectorized using special HW instruction

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

Raising Vectorization Ratio
Dealing with Unvectorizable Dependencies

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

```c
#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

```c
#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

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

```c
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

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**

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

The condition is not satisfied when n is an odd number.

**Vectorized Loop**

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

Fix to “i < n”

**Unvectorized Loop**

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

C++ iterator type array

**Vectorized Loop**

```c++
double *first, *last, *p;
......
for (p=first; p < last; p++)
{
    ...... 
}
```
Logical AND/OR Operator in Loop-termination-expression

- 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.

```c
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;
}
```

```c
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

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++) {
    a[i] = fun(b[i], c[i]) + d[i];
}
```

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

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

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

Raising Vectorization Ratio
Outer Loop Unrolling

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.

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

Program after unrolling the outer loop 4 times.

```cpp
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];
    }
}
```

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

Specifying \texttt{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.

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

If \texttt{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, if there are no duplicate values in “ix[i]”, \textit{more efficient vector instructions can be generated by specifying \texttt{ivdep} instead of \texttt{list_vector}}.

<Message after vectorization by \texttt{ivdep}>

\begin{verbatim}
ncc: vec( 101): a.c, line 8: Vectorized loop.
\end{verbatim}
OpenMP and Automatic Parallelization
OpenMP Parallelization

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.

Specify “-fopenmp” also when linking

$ ncc -fopenmp a.c b.c
Parallelize function "sub" of Example 1 with OpenMP C/C++

```
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;
}
```

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

- `#pragma omp parallel for`:
  - `parallel`: Specify start of parallelization region
  - `for`: Specify parallelization of for loop

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.
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

```c
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;
}
```

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.

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.

Also specify `-mparallel` for linking.

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

<table>
<thead>
<tr>
<th>Programming method</th>
<th>Select loops / blocks</th>
<th>Insert directives</th>
<th>Program modification</th>
<th>Difficulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP C/C++ (-fopenmp)</td>
<td>O</td>
<td>O</td>
<td>–</td>
<td>High</td>
</tr>
<tr>
<td>Automatic parallelization (-mparallel)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>Low</td>
</tr>
</tbody>
</table>

O : 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

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`.

```c
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;
}
```

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

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.

Specify #pragma _NEC concurrent schedule(dynamic, 4) right before an outermost loop
MPI Parallelization
Compiling and Linking MPI Programs

◆ It is possible to compile and link MPI programs with the MPI compilation commands corresponding to each programming 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
```
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.

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

**NEC Compiler:**

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

**GNU Compiler:**

(setup the GNU compiler (e.g., PATH, LD_LIBRARY_PATH))

```bash
$ 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.

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int my_rank, name1;
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Get_processor_name(name, &name1);
    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.

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int my_rank, name1;
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Get_processor_name(name, &name1);
    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.

```plaintext
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.

<table>
<thead>
<tr>
<th>Process 0</th>
<th>is running on node0, VH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>is running on node0, VH</td>
</tr>
<tr>
<td>Process 2</td>
<td>is running on node0, VH</td>
</tr>
<tr>
<td>Process 3</td>
<td>is running on node0, VH</td>
</tr>
<tr>
<td>Process 4</td>
<td>is running on node0, VH</td>
</tr>
<tr>
<td>Process 5</td>
<td>is running on node0, VH</td>
</tr>
</tbody>
</table>

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

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

$ mpirun -vh -nnp 3 ./mpi_VH

Output of the above execution command.

<table>
<thead>
<tr>
<th>Process</th>
<th>Status</th>
<th>Node</th>
<th>VH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>running on node0</td>
<td>VH0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>running on node0</td>
<td>VH0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>running on node0</td>
<td>VH0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>running on node1</td>
<td>VH1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>running on node1</td>
<td>VH1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>running on node1</td>
<td>VH1</td>
<td></td>
</tr>
</tbody>
</table>

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

```sh
$ 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.

<table>
<thead>
<tr>
<th>Process</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>running on node0, VH</td>
</tr>
<tr>
<td>1</td>
<td>running on node0, VH</td>
</tr>
<tr>
<td>2</td>
<td>running on node0, ve id 0</td>
</tr>
<tr>
<td>3</td>
<td>running on node0, ve id 1</td>
</tr>
<tr>
<td>4</td>
<td>running on node0, ve id 2</td>
</tr>
<tr>
<td>5</td>
<td>running on node0, ve id 2</td>
</tr>
<tr>
<td>6</td>
<td>running on node0, ve id 3</td>
</tr>
<tr>
<td>7</td>
<td>running on node0, ve id 3</td>
</tr>
<tr>
<td>8</td>
<td>running on node0, VH</td>
</tr>
</tbody>
</table>

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:

$ 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
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
$ ./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

```bash
$ 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:

```bash
$ 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:

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

◆ To run the benchmark, we need access to `pw.x` and a run script.

```bash
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
  ▪ 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 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.