NEC

SX-Aurora TSUBASA

Performance Tuning Guide

SX-Aurora TSUBASA

Proprietary Notice

The information disclosed in this document is the property of NEC Corporation (NEC) and/or its licensors. NEC and/or its licensors, as appropriate, reserve all patent, copyright, and other proprietary rights to this document, including all design, manufacturing, reproduction, use and sales rights thereto, except to the extent said rights are expressly granted to others.

The information in this document is subject to change at any time, without notice.

Remarks

- NEC Fortran Compiler conforms to the following language standards.
  - ISO/IEC 1539-1:2004 Programming languages
  - Fortran-OpenMP Application Program Interface Version 4.5

NEC Fortran compiler also conforms a part of “ISO/IEC 1539-1:2010 Programming languages –Fortran"

- NEC C/C++ Compiler conforms to the following language standards.
  -ISO/IEC 9899:2011 Programming languages
  -C-ISO/IEC 14882:2014 Programming languages
  -C++-OpenMP Application Program Interface Version 4.5

- All product, brand, or trade names in this publication are the trademarks or registered trademarks of their respective owners.
- In this document, the Vector Engine is abbreviated as VE.
- The reader of this document assumes that you have knowledge of software development in Fortran/C/C++ language on Linux.

Preface

Majority of the discussed tuning mechanisms are verified for C/C++ and Fortran90 development only. Also, the optimization and performance tuning avenues have been discussed for only NEC SX-Aurora TSUBASA target. Therefore, the methodologies are limited to the compilers and tools provided with the NEC SX-Aurora TSUBASA development and execution environment.

Some of the tuning methodologies (like loop unrolling, loop-fusion, etc.) are related to the general concepts of code optimization and are independent of the underlying architecture.

This document aims at exploring the architecture of NEC's SX-Aurora TSUBASA Vector Engine and opportunities for source code tuning for the architecture.

Conventions

The following conventions are used throughout this document.

- Names of variables, directives, options are printed in *italics*.
- Syntaxes, commands appear in gray boxes.
- Source code is enclosed in tables and boxes.

Contents

2.1 Format List 9
2.2 Diagnostic List 10
2.3 Program Information (aka PROGINF) 11
2.4 FTRACE – Simple Performance Analysis Function 12
3.1 Analyze the Application 15
3.1.1 Static analysis: Study and Investigation 15
3.1.2 Performance Analysis through Execution 15
3.2 Identify Tuning Candidates 18
4.1 Performance gain through Parallelization 24
4.1.1 OpenMP 24
4.1.2 Message Passing Interface (MPI) 24
4.2 Performance gain through Vectorization 25
4.2.1 Vectorization 25
4.2.2 Basic Conditions for Vectorization 27
4.2.3 Data Dependency Conditions 31
4.2.4 Improving the Vectorization Ratio 36
4.2.5 Improving Vector Instruction Efficiency 39
4.2.6 Lengthening the Loop 39
4.2.7 Improving Array Reference Patterns 40
4.2.8 Removing IF Statements 42
4.2.9 Avoiding Iterative Operations 45
4.2.10 Avoiding Loop Division 45
4.2.11 Avoiding Loop Unrolling for Short Loops 45
4.2.12 Increasing Concurrency 46
4.2.13 Avoiding Arithmetic Division 46
4.2.14 Using Vectorization Options and Directives 46
4.2.15 Other Effective Techniques 49
4.2.16 Vectorization by Statement Replacement 50
4.2.17 Vectorization Using Work Vectors 50
4.2.18 Macro Operations 50
4.2.19 Examples of Vectorization 54

4.2.20 Partial Vectorization 60
4.2.21 Code-Related Optimization 60
4.2.22 Loop Transformations 64
4.2.23 Effects on Arithmetic Results 78
4.2.24 Detection of Vectorization-Caused Errors and Exceptions 78
4.2.25 Boundary of Dummy Array 79
4.2.26 Array Declaration 79
4.2.27 Association of Dummy Arguments 80
4.2.28 High-Speed I/O Techniques 81

List of figures

Figure 1 Types of Vectors 35
Figure 2 Start-Up Time and Cross Length 39
Figure 3 Constant stride vector 40
Figure 4 Vector Mask Generation Instruction 56
Figure 5 Masked Vector Operation Instruction.. 56
Figure 6 Vector Compression and Expansion 58
Figure 7 Vector Gather and Scatter Instructions.................................... 60

Chapter1 What is a Vector Architecture?

Vector architectures grab sets of data elements scattered about memory, place them into large, sequential register files, operate on data in those register files, and then disperse the results back into memory. A single instruction operates on vectors of data, which results in dozens of register–register operations on independent data elements.

A key aspect of vector architecture is the single-instruction-multiple-data (SIMD) execution model. SIMD support results from the type of data supported by the instruction set, and how instructions operate on that data.

In a traditional scalar processor, the basic data type is an n-bit word. The architecture often exposes a register file of words, and the instruction set is composed of instructions that operate on individual words.

In vector architecture, there is support of a vector data type, where a vector is a collection of VL n-bit words (VL is the vector length). Previously, vector machines operated on vectors stored in main memory.

Chapter2　Toolkit on NEC SX-Aurora TSUBASA

Once the basics of vectorization are discussed, we can now move to specific tools for NEC SX-Aurora TSUBASA that help the programmer achieve vectorization.

Below is the list of compilers that are available on NEC SX-Aurora TSUBASA.

Compiler name	Language	Path
ncc	C	/opt/nec/ve/bin/
mpincc	C with MPI support	/opt/nec/ve/bin/
nfort	FORTRAN	/opt/nec/ve/bin/
mpnfort	FORTRAN with MPI support	/opt/nec/ve/bin/

These compilers support some switches which help the programmer obtain clues to perform optimization. They are described below:

2.1　Format List

The format list is an illustrative representation of the current optimization status of the source code. The source lines for each function together with the following information are output to the list.

- Vectorization status of each loop
- Parallelization status of each loop
- Inline expansion status of function calls

The format list can be obtained by using the *-report-format* or *-report-all* compiler switch.

```
ncc -report-format a.c
```

The list is created in the current directory, under the name *source-file-name.L*.

The format list looks like below:

```
NEC C/C++ Compiler (1.0.0) for Vector Engine Wed Jan 17 14:55:16
2018 (a)

FILE NAME: a.c (b)

FUNCTION NAME: func (c)
FORMAT LIST

LINE   LOOP      STATEMENT
(d)  (e)  (f)
  1:            int func(int m, int n)
  2:            {
  3:                int i,j, a[m][n], b[m][n];
  4: +------>       for (i = 0; i < m; i++) {
  5: |V----->          for (j = 0; j < n; j++) {
  6: ||                    a[i][j] = a[i][j] + b[i][j];
  7: |V-----            }
  8: +------    }
  9:            return a[0][0];
  10: }
```

2.2 Diagnostic List

Diagnostics are categorized as follows and output in the list.

- Diagnostics for inline expansion
- Diagnostics for optimization
- Diagnostics for vectorization and parallelization

The diagnostic list gives a detailed optimization status of the source code with line numbers. The explanation helps the developer to strategize tuning.

The diagnostic list can be obtained by using the *-report-diagnostics* or *-report-all* compiler switch.

```
nfort -report-diagnostics fft.f90
```

The list is created in the current directory, under the name *source-file-name.L*.

```
NEC C/C++ Compiler (1.0.0) for Vector Engine Wed Jan 17 14:55:20
2018 (a)

FILE NAME: fft.f90 (b)

FUNCTION NAME: FFT_3D (c)
DIAGNOSTIC LIST

LINE DIAGNOSTIC MESSAGE
(d)     (e)          (f)
 7:   inl(1222): Inlined
 9:   vec( 101): Vectorized loop. 440: vec(  10): Vectorization obstructive
procedure reference.: fft1dA
 448: vec(   1): Vectorized loop.
```

2.3 Program Information (aka PROGINF)

Program information is a report that contains the major execution parameters such as Execution Time, Memory Size, Vec.Op Ratio, Avg Vector Length, etc. The generation of this report is controlled by the environment variable VE_PROGINF. PROGINF can be obtained by setting VE_PROGINF to YES or DETAIL.

```
$ export VE_PROGINF=DETAIL
$ /opt/nec/ve/bin/ve_exec ./a.out
```

This report is generated after the execution of a load module. It is output to *stderr* after its complete execution. The report looks like below:

```
          ********  Program  Information  ********
  Real Time (sec)                         :           204.076110
  User Time (sec)                         :           203.706817
  Vector Time (sec)                       :           197.623752
  Inst. Count                             :          38596814372
  V. Inst. Count                          :          13465836887
  V. Element Count                        :        2957231889428
  V. Load Element Count                   :         997524789907
  FLOP Count                              :        1776569208614
  MOPS                                    :         18087.515129
  MOPS (Real)                             :         18053.533350
  MFLOPS                                  :          8721.924006
  MFLOPS (Real)                           :          8705.537759
  A. V. Length                            :           219.609959
  V. Op. Ratio (%)                        :            99.317880
  L1 Cache Miss (sec)                     :             5.637238
  CPU Port Conf. (sec)                    :             0.125939
  V. Arith. Exec. (sec)                   :            29.765092
  V. Load Exec. (sec)                     :           163.530245
  VLD LLC Hit Element Ratio (%)           :            58.115252
  Power Throttling (sec)                  :             0.000000
  Thermal Throttling (sec)                :             0.000000
  Memory Size Used (MB)                   :           592.000000

  Start Time (date)      :        Tue Feb  5 23:42:11 2019 JST
  End   Time (date)      :        Tue Feb  5 23:45:35 2019 JST
```

2.4 FTRACE – Simple Performance Analysis Function

The compiler kit of SX supports a performance analysis function called FTRACE.

It is used to obtain performance information on the CPU overhead and vectorization of each code region in a program. It can be used to obtain performance information for:

- each function/subroutine of the program
- any programmer-defined region

The ftrace report can be obtained by using the *-ftrace* compiler switch

```
ncc -ftrace source.c
```

Once the source code is compiled using the ftrace switch, an ftrace-compliant binary is generated. Upon execution among other output files, an ftrace report is generated by the name of *ftrace.out*.

ftrace tool-kit provides tools to convert the *ftrace.out* files to text/readable format. The tools are available on the below path:

Tool name	Path	Syntax
ftrace	/opt/nec/ve/bin/	`ftrace -f ftrace.out -fmt1`

Once converted to text, the ftrace report looks like below:

```
*----------------------*
FTRACE ANALYSIS LIST*
----------------------*
Execution Date : Sat Feb 17 12:44:49 2018 JST
Total CPU Time : 0:03'24"569 (204.569 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.%
    1012   49.093( 24.0)    48.511  23317.2  14001.4  96.97  83.2    42.132   5.511    0.000   80.32   funcA
  160640   37.475( 18.3)     0.233  17874.6   9985.9  95.22  52.2    34.223   1.973    2.166   96.84   funcB
  160640   30.515( 14.9)     0.190  22141.8  12263.7  95.50  52.8    29.272   0.191    2.544   93.23   funcC
  160640   23.434( 11.5)     0.146  44919.9  22923.2  97.75  98.5    21.869   0.741    4.590   97.82   funcD
  160640   22.462( 11.0)     0.140  42924.5  21989.6  97.73  99.4    20.951   1.212    4.590   96.91   funcE
53562928   15.371(  7.5)     0.000   1819.0    742.2   0.00   0.0     0.000   1.253    0.000    0.00   funcG
       8   14.266(  7.0)  1783.201   1077.3     55.7   0.00   0.0     0.000   4.480    0.000    0.00   funcH
  642560    5.641(  2.8)     0.009    487.7      0.2  46.45  35.1     1.833   1.609    0.007   91.68   funcF
    2032    2.477(  1.2)     1.219    667.1      0.0  89.97  28.5     2.218   0.041    0.015   70.42   funcI
       8    1.971(  1.0)   246.398  21586.7   7823.4  96.21  79.6     1.650   0.271    0.000    2.58   funcJ
----------------------------------------------------------------------------------------------------------------
54851346  204.569(100.0)     0.004  22508.5  12210.7  95.64  76.5   154.524  17.740   13.916   90.29   total
```

NOTE: Execution time of *–ftrace* execution is longer than no-ftrace execution. Delay is directly proportional to the no. of subroutine calls.

Chapter3　Methodology for Tuning

This is a step-by-step procedure of tuning various applications on NEC SX-Aurora TSUBASA. The tuning process can be broken down to the following major steps:

Step 1: Application analysis

Step 2: Identification of Overall Tuning Strategy

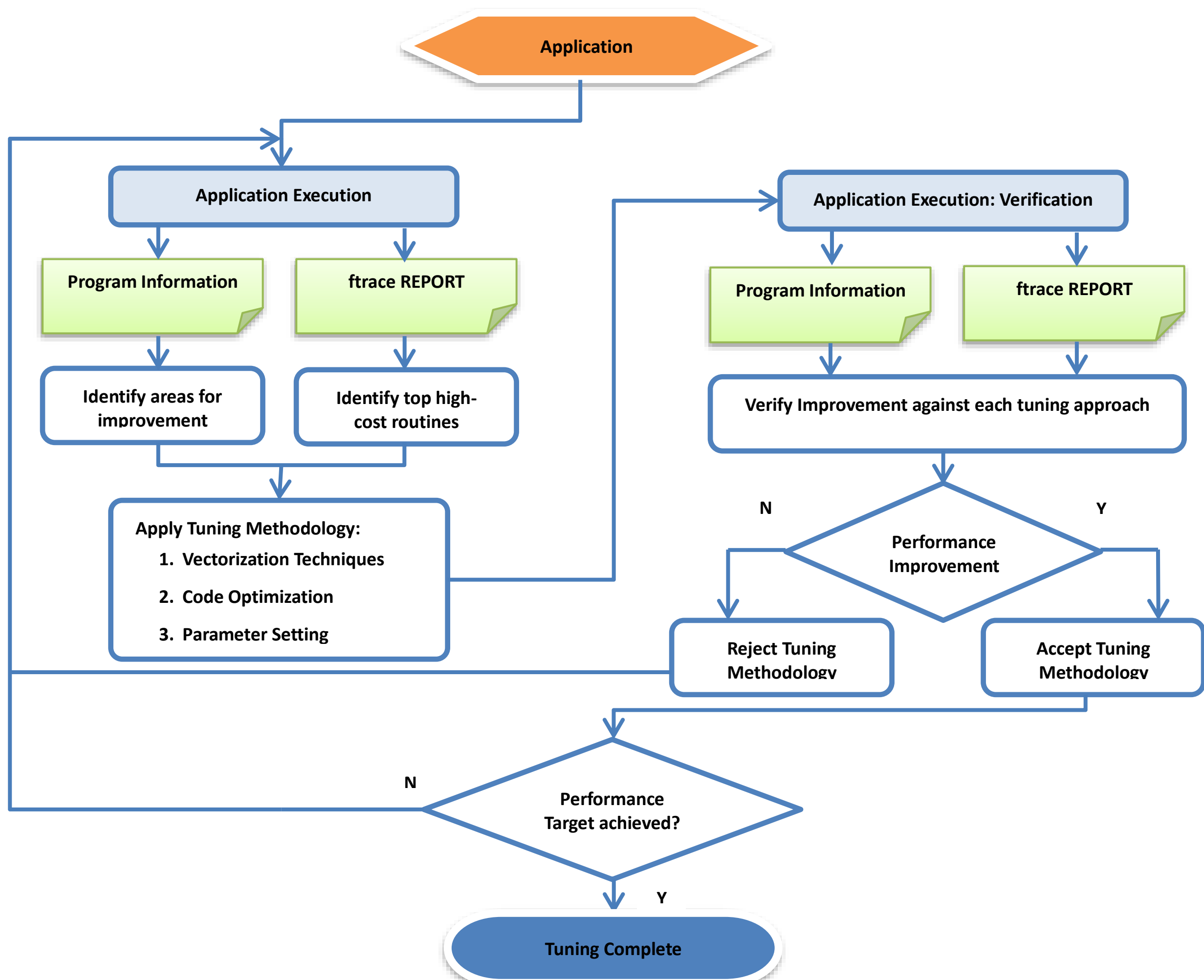
Step 3: Identification of Tuning Candidates

Step 4: Identification of Specific Tuning Strategy

Step 5: Implement Performance Tuning Strategy

Step 6: Re-estimate Performance

Step 7: Result Verification



Let's discuss the detailed procedure for each step mentioned.

3.1 Analyze the Application

The application should be analyzed in two phases:

1. Static Analysis: Study and Investigation
2. Performance Analysis through Execution

3.1.1 Static analysis: Study and Investigation

From this phase, the below information should be obtained:

- Program Overview: Simulation methods, Convergence methods used in the application.
- File Structure of the application:
  - (1) No. of Input/Output/config files
  - (2) Data Structures involved in read/write of files.
  - (3) Details of routines that refer to input/output files
- Application code flow:
  - (1) Internal/External Interfaces
  - (2) List of routines that are a part of the initialization part of the application
  - (3) List of routines that are a part of the main calculation loop of the application.
  - (4) Memory Requirements
- Any specific frameworks used in the source code
  - (1) Parallelization frameworks like MPI
  - (2) Multithreading Frameworks like OpenMP, etc
- Problem topology for parallelization

3.1.2 Performance Analysis through Execution

Performance tuning for an application can also be planned based on performance parameters from the execution reports. It is expected that the execution information meets the peak performance of the underlying architecture.

```
        ********  Program  Information  ********
Real Time (sec)                          :          204.076110
User Time (sec)                          :          203.706817
Vector Time (sec)                        :          197.623752
Inst. Count                              :         38596814372
V. Inst. Count                           :         13465836887
V. Element Count                         :       2957231889428
V. Load Element Count                    :        997524789907
FLOP Count                               :       1776569208614
MOPS                                     :        18087.515129
MOPS (Real)                              :        18053.533350
MFLOPS                                   :         8721.924006
MFLOPS (Real)                            :         8705.537759
A. V. Length                             :          219.609959
V. Op. Ratio (%)                         :           99.317880
L1 Cache Miss (sec)                      :            5.637238
CPU Port Conf. (sec)                     :            0.125939
V. Arith. Exec. (sec)                    :           29.765092
V. Load Exec. (sec)                      :          163.530245
VLD LLC Hit Element Ratio (%)            :           58.115252
Power Throttling (sec)                   :            0.000000
Thermal Throttling (sec)                 :            0.000000
Memory Size Used (MB)                    :          592.000000

Start Time (date)        :        Tue Feb  5 23:42:11 2019 JST
End   Time (date)        :        Tue Feb  5 23:45:35 2019 JST
```

From the above sample program information, the **MFLOPS** parameter is observed as nearly **8.7 GFLOPS**. There is a scope for optimization in the application source that can utilize the peak performance.

Other such parameters are:

Performance Parameter	Recorded Value	Target Value	Remarks
MFLOPS	8721.92	Close to peak	
A. V. Length	219.61	256	
V. Op. Ratio (%)	99.31	99.99	
VLD LLC Hit Element Ratio (%)	58.11	100.00	

Referring to Amdahl's law, it is attempted that every tuning strategy must lead to 99.99% vectorization of the target source code.

There are other parameters that are vital to the overall performance; although target peak

values cannot be fixed. Such parameters are preferred to be improved to as optimized as possible.

Performance Parameter	Recorded Value	Target Value	Remarks
Real Time (sec)	204.08	Lowest	Represents the wall-clock time
CPU Time(sec)	203.70	Lowest	Exclusive time, User Time
Vector Time (sec)	197.62	Close to Real Time	Vector–only time. Depends on B/F.

Through this evaluation step, the areas for performance improvement are prioritized. Once prioritized, it is noted which are the parameters that need improvement first.

When a performance parameter is not optimum, each participating high-cost routine must be tuned with target to improve that performance parameter. The combined effect of tuning of each high-cost routine will reflect on the overall performance of the application.

Note that the performance of an application directly maps with the real time consumed for its execution. The motive behind improving the MFLOPS, average vector length, vectorization ratio, etc. is for faster execution, i.e. total execution time.

Exceptional cases may occur where tuning brings improvement in the targeted parameter, say Vector Op. Ratio, but results in longer execution time. This may happen due to reasons like:

- vectorization of a short loop
- network port conflicts
- high cost inline expansion, etc.

In such cases, it is important to take a call where highest priority lies. General expectation from tuning is faster execution. The tuning measure may be rejected if it ends up slowing down execution. However, attempts must be made to identify the root cause of the exceptional behavior and rectify them so as to improve the execution time.

Once the tuning approaches have been identified, the next step is to identify the tuning candidates.

3.2 Identify Tuning Candidates

We need to execute the ftrace compliant build version in order to move ahead. This ftrace report is then analyzed for identification of tuning avenues.

Below is a sample ftrace report for reference:

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

Execution Date : Tue Feb 19 12:23:39 2019 JST
Total CPU Time : 0:05'31"547 (331.547 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.%

    16008   176.431( 53.2)    11.021   1365.0      0.0  27.44 254.0  4.345  84.385    0.738    95.05 SUB01
 37650816    71.194( 21.5)     0.002  35742.3  11539.4  97.93 246.9 56.919  10.218    0.000    99.97 SUB02
  4706352    47.946( 14.5)     0.010   9169.6   5266.6  89.40 246.9  5.535  23.156    0.000    99.97 SUB03
 18825408    20.172(  6.1)     0.001  10260.1   3914.7  95.00 246.7  5.839   9.834    0.000    99.94 SUB04
```

The routines are first scanned for their percentage contribution to the overall execution time (cost of the routine). Then those routines are selected that are higher-cost as compared to the rest of the participating routines (roughly greater than ~1%).

The individual performance parameters of each shortlisted routine are scanned based on the above thumb rule and strategy is chosen for each routine.

Tuning Approach 1: Inline Expansion

The objective of inline expansion is to get rid of the leaves of the calling tree. By using inline expansion, the frequency of the iterative function calls can be decreased. Functions calls within loops are a common obstruction to vectorization, parallelization and optimization in general. Inline expansion is a tool to get rid of such obstructions.

- A function can identified as an inline expansion candidate, if FREQUENCY is high and AVER. TIME is low. The gauge of high and low is relative to the application. The tuning personnel can judge by analysis if the “FREQUENCY:AVER.TIME” ratio is suitable for performing inline expansion.

  Achievement Expected: Large number of iterative calls to low cost function creates a major overhead. By inline expansion such functions, this overhead may be avoided.

- If there is a high-cost routine and its vectorization/optimization is inhibited by a

function call, the called function can be identified an inline expansion candidate.

Achievement Expected:

a) If the cost of the called routine is low, it may help the caller loop to be vectorized and further optimized.

b) If the cost of the called routine is high, when tuned it may help improve its performance combined with its caller.

c) If the cost of the called routine is high, but not tuned, its high cost will add up to the cost of the caller and may not contribute to overall performance gain.

Inline expansion may cause a prominent re-ordering in the costs of functions of the original program. Hence, it is recommended to perform inline expansion before proceeding to other optimization strategies.

- Automatic Inline expansion

  The compiler chooses the appropriate functions and tries to inline them automatically when corresponding compiler option is specified.

- Explicit Inline expansion

  The compiler tries to automatically inline a function that is called in a statement after the inline expansion directive (that is specified together with the inline expansion directive).

  When the number of calls to the subroutine are very large and the average time per call is relatively small, consider inline expansion. If not, then better not to specify as the cost will be added to the parent/caller routine.

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

Execution Date : Tue Feb 19 12:23:39 2019 JST
Total CPU Time : 0:05'31"547 (331.547 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.%

    16008   176.431( 53.2)    11.021   1365.0      0.0  27.44 254.0  4.345  84.385    0.738    95.05 SUB01
 37650816    71.194( 21.5)     0.002  35742.3  11539.4  97.93 246.9 56.919  10.218    0.000    99.97 SUB02
  4706352    47.946( 14.5)     0.010   9169.6   5266.6  89.40 246.9  5.535  23.156    0.000    99.97 SUB03
 18825408    20.172(  6.1)     0.001  10260.1   3914.7  95.00 246.7  5.839   9.834    0.000    99.94 SUB04
  9412704     8.020(  2.4)     0.001  60934.6  39371.2  98.22 230.9  5.105   1.229    0.000    99.81 SUB05
 10693344     3.760(  1.1)     0.000    972.9      0.0   0.00   0.0  0.000   1.708    0.000     0.00 SUB06
  8132064     2.407(  0.7)     0.000   1162.3      0.0   0.00   0.0  0.000   0.949    0.000     0.00 SUB07
    16008     1.334(  0.4)     0.083  30481.5  23625.6  98.97 252.7  1.198   0.026    0.000    94.41 SUB08
        8     0.118(  0.0)    14.750   1037.0      5.5  10.03 252.3  0.000   0.000    0.000   100.00 SUB09
        1     0.115(  0.0)   114.572    744.8      7.6   0.01 229.0  0.000   0.049    0.000     0.00 SUB10
    32016     0.049(  0.0)     0.002  33744.0  16414.2  98.06 246.9  0.038   0.008    0.000    99.23 SUB11
        8     0.000(  0.0)     0.001  20920.9      0.0  95.19 250.1  0.000   0.000    0.000     0.00 SUB12
        8     0.000(  0.0)     0.000    880.6      0.0   0.00   0.0  0.000   0.000    0.000     0.00 SUB13
-------------------------------------------------------------------------------------------------------------
 89484745   331.547(100.0)     0.004  11973.5   4527.6  92.31 245.1 78.979 131.562    0.738    99.80 total
```

After the inline expansion, the inline expanded routines are now included as part of the caller routines.

```
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*
Execution Date : Tue Feb 23 14:33:32 2019 JST
Total CPU Time : 0:02'09"983 (129.983 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.%

  4706352    94.024( 72.3)     0.020  38555.8  15621.1  98.48 244.9  75.580  13.927    0.000    99.96 SUB03
    16008    34.331( 26.4)     2.145   2815.5      0.0  68.38 254.0   4.423  15.960    0.738    95.16 SUB01
    16008     1.334(  1.0)     0.083  30461.5  23609.6  98.97 252.7   1.198   0.030    0.000    94.55 SUB08
        1     0.123(  0.1)   122.810    695.6      7.1   0.01 229.0   0.000   0.055    0.000     0.00 SUB10
        8     0.120(  0.1)    14.961   1036.3      5.4   9.96 252.2   0.000   0.000    0.000   100.00 SUB09
    32016     0.051(  0.0)     0.002  32922.0  16014.3  98.06 246.9   0.038   0.011    0.000    99.00 SUB11
--------------------------------------------------------------------------------------------------------
  4770393   129.983(100.0)     0.027  28960.4  11548.3  97.66 245.1  81.238  29.982    0.738    99.80 total
```

Tuning Approach 2:

In order to choose the correct optimization approach for a code, one must refer to the diagnostic list to understand which parts of the code are inhibiting optimization.

Eg: Refer to the Diagnostic list below

```
LINE            DIAGNOSTIC MESSAGE
701: vec( 101): Vectorized loop.
727: vec( 102): Partially vectorized loop.
729: opt(1036): Potential feedback - use directive or compiler option if OK.
732: vec( 122): Dependency unknown. Unvectorizable dependency is assumed.: r_arr
```

…corresponding to the below code:

```
LINE   LOOP      STATEMENT

701: V------>       do j = 1, n
702: |                r_arr(j) = 0
703: |                mark(j) = .false.
704: V------        enddo
705:              r_arr(n+1) = 0
 :
727: S------>       do i = 1, n
728: |                j = rows(i) - f + 1
729: |                k = r_arr(j)
730: |                a(k) = ref_array(i)
731: |                c_arr(k) = cols(i)
732: |                r_arr(j) = r_arr(j) + 1
733: S------        enddo
```

Here, diagnostic list helps us to understand that the loop beginning at line no. 727 failed to be auto-vectorized completely by the compiler. The code is then analyzed by the programmer and optimization scheme is chosen. Suitable schemes here:

- Compiler Directive
- Loop-collapse
- Loop-split
- Inline expansion of routine calls

```
LINE   LOOP      STATEMENT

 701: V------>       do j = 1, n
 702: |                r_arr(j) = 0
 703: |                mark(j) = .false.
 704: V------        enddo
 705:              r_arr(n+1) = 0
  :
 727:          cNEC$ ivdep
 728: V------>       do i = 1, n
 729: |                j = rows(i) - f + 1
 730: |      G         k = r_arr(j)
 731: |      C         a(k) = ref_array(i)
 732: |      C         c_arr(k) = cols(i)
 733: |      C         r_arr(j) = r_arr(j) + 1
 734: V------        enddo
```

The effect is confirmed by looking at the diagnostic list. For the example above, the diaglist now looks as below:

```
LINE            DIAGNOSTIC MESSAGE
701: vec( 101): Vectorized loop.
707: vec( 102): Partially vectorized loop.
728: vec( 101): Vectorized loop.
```

Tuning Approach 3:

When a routine has been identified as high cost has several loops, it is important to understand which loop or segment of that function is contributing most to the execution time. In this situation, it is recommended to use the ftrace-region profiler extension. Each loop can be enclosed within a user-defined region. The report can easily specify the highest-cost loop within the high-cost routine.

Refer to the PROGINF/FTRACE User's Guide for detailed instructions.

Chapter4 Performance Optimization

Optimization is a very broad term. In general, it implies transforming the code to make some of its aspects to work more efficiently or use fewer resources or be more robust. For example, a program may be optimized so that it will execute faster or use less memory or in case of NEC SX-Aurora TSUBASA, use the vector resources more efficiently.

The principle is to make a program more efficient and quicker without changing its output or effects. The process of optimization does not necessarily produce a totally optimal system. There's always a trade-off, so only the most lucrative attributes are chosen to be optimized.

Example of code optimization:

Example (C/C++):

ORIGINAL

```
:
   x = y % 32;
   x = y * 8;
   x = y / w + z / w;
   if( a==b &&c==d &&e==f ) {...}
   if( (x &1) || (x &4) ) {...}
   if( x>=0 &&x<8 &&
       y>=0 &&y<8 ) {...}
   if( (x==1) || (x==2)  ||
       (x==4) || (x==8)  || ... )
   if( (x==2) || (x==3)  || (x==5)  ||
       (x==7) || (x==11) || (x==13) ||
       (x==17)|| (x==19) ) {...}
.
.
```

OPTIMIZED

```
:
    x = y & 31;
    x = y <<3;
    x = (y + z) / w;
    if( ((a-b)|(c-d)|(e-f))==0 ) {...}
    if( x & 5 ) {...}
    if( ((unsigned)(x|y))<8 ) {...}

    if( x&(x-1)==0 &&x!=0 )


    if( (1<<x) &
((1<<2)|(1<<3)|(1<<5)|
(1<<7) |(1<<11)|(1<<13)
|(1<<17)|(1<<19)) ) {...}
.
.
```

Example (FORTRAN) :

ORIGINAL

```
do i=1,n               ! column indexing
   do j=1,n
      a(i,j) = a(i,j) + b(i,j)*c(i,j)
   end do
end do
```

OPTIMIZED

```
do j=1,n               ! row indexing
   do i=1,n
      a(i,j) = a(i,j) + b(i,j)*c(i,j)
   end do
end do
```

4.1 Performance gain through Parallelization

Parallel computing techniques can help reduce the time it takes to reach a solution. To derive the full benefits of parallelization, it is important to choose an approach that is appropriate for the optimization problem.

Few frameworks are widely used nowadays to achieve parallelization. Let's discuss some of those:

4.1.1 OpenMP

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs.

In order to parallelize a code, programmers look for regions of code whose instructions can be shared among the processors. Much of the time, they focus on distributing the work in loop nests to the processors. In most programs, the code executed on one processor requires results that have been calculated on another one. In principle, this is not a problem because a value produced by one processor can be stored in main memory and retrieved from there by code running on other processors as needed. However, the programmer needs to ensure that the value is retrieved after it has been produced, that is, that the accesses occur in the required order. Since the processors operate independently of one another, this is a nontrivial difficulty: their clocks are not synchronized, and they can and do execute their portions of the code at slightly different speeds.

A number of compilers from various vendors or open source communities implement the OpenMP API.

In case of NEC SX-Aurora TSUBASA environment, OpenMP is supported by the compilers ncc, nfort by specifying the switch *-fopenmp* at compilation.

Compilation syntax on NEC SX-Aurora TSUBASA:

```
% nfort -fopenmp -c a.f
% ncc   -fopenmp -c b.c
```

Refer to the Fortran Compiler User's Guide for detailed instructions on how to use OpenMP.

4.1.2 Message Passing Interface (MPI)

The Message-Passing Interface (MPI) is a specifications standard that supports coding of distributed memory parallel programs by means of message passing (point-to-point and one-sided) and collective communication operations among processes.

NEC MPI is an implementation of MPI Version 3.1, which uses shared memory feature of a

VH, and InfiniBand functions for communication to achieve high-performance communication. The Fortran compiler (nfort), C compiler (ncc), or C++ compiler (nc++) support compilation and linking of MPI programs. NEC MPI does not support communication in heterogeneous environments (for example, communication among processes that run on different multiple systems).
Refer to the NEC MPI User Guide for detailed instructions on how to use MPI for parallelization.

4.2 Performance gain through Vectorization

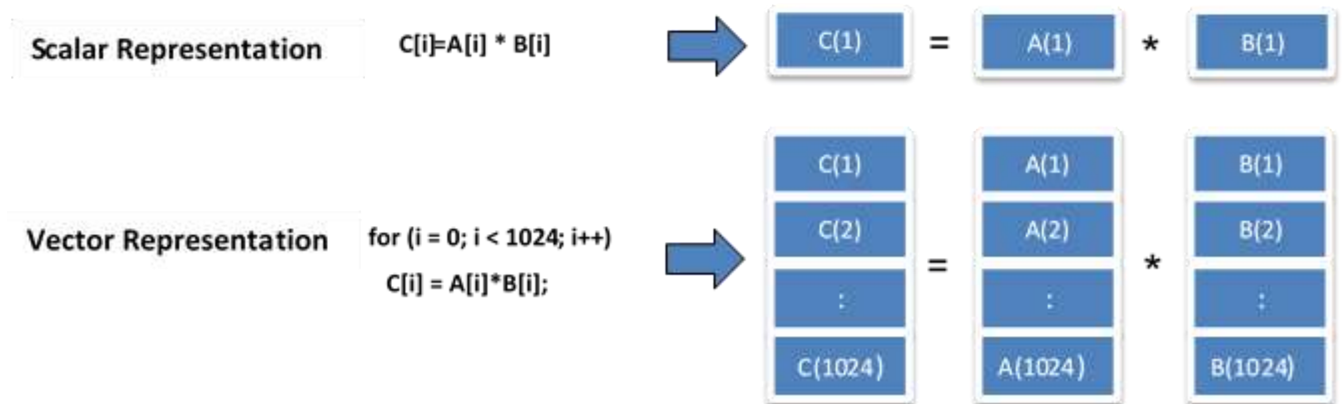
The most important consideration in completely utilizing the high-speed features of the VE is to maximize the vectorization ratio of the part processed by vector instructions followed by improving the efficiency of generated vector instructions.
This section gives suggestions on how to rewrite programs by using these tools to obtain improved performance.

4.2.1 Vectorization

Variables and single elements of an array are examples of scalar data. When arranged in a sequence such as a line, column, or diagonal of a matrix, is known as vector data. An ordinary processor has instructions (called scalar instructions) that can execute operations on only one data item at a time.
The Vector Engine has advanced vector instruction sets that are capable of applying operations (ADD, MUL, etc.) simultaneously to multiple operands (or vector data), while scalar processors can only operate on pairs of operands at once. In order to take full advantage of the features of the Vector Engine, vector instructions are preferable to scalar instructions because they are much faster. Vectorization is the process of replacement of scalar instructions with vector instructions, thereby converting a scalar program to a vector program.
In a typical C/C++ or FORTRAN program, a conversion from scalar to vector is typically targeted at vectorizable subjects like array expressions and loop structures. So, to vectorize a loop means to represent the loop such that it processes LVL elements out of total N elements of the array simultaneously N/LVL times (where LVL is the Vector Length – 256 for VE), instead of processing a single element of an array N times.

Scalar Approach	Vector Approach

for (i = 0; i < 1024; i++)
{
C[i] = A[i]*B[i];
}
	for (i = 0; i < 1024; i++)
	{
	C[i] = A[i]*B[i];
	}



Is Vectorization a similar concept to Loop Unrolling?

The answer is No.

Consider the following very simple loop that adds the elements of two arrays and stores the results to a third array.

Original	Loop Unrolled

for (int i=0; i<1024; ++i)
C[i] = A[i] + B[i];
	for (int i=0; i<1024; i+=256) {
	C[i] = A[i] + B[i];
	C[i+1] = A[i+1] + B[i+1];
	C[i+2] = A[i+2] + B[i+2];
	:
	C[i+255] = A[i+255] + B[i+255];
	}

This same simple loop, however, when vectorized will be interpreted as below:

Original	Loop Vectorized
for (int i=0; i<1024; ++i) C[i] = A[i] + B[i];	for (int i=0; i<1024; ++i) compilerInstrinsic(&C[i], &A[i], &B[i]);

Here, "compilerIntrinsic" is what the compiler uses to specify vector instructions for addition. In case of vectorization, there are more registers available in the hardware to perform the operation and the compiler enables vector instructions for the operation to execute faster than a series of scalar instructions corresponding to the unrolled loop.

Automatic Vectorization is the process of generating vector instructions at compile time for a program by the compiler. This conversion from scalar to vector MUST guarantee the safeguarding of exact program behavior.

Vector Engine has advanced automatic vectorization functions. The programmer need not revise a program using standard language since the compiler automatically analyzes the source program to detect parts that can be executed by vector instructions, and generates the vector version automatically. The programmer also need not be aware of architecture specific instructions in order to make full use of the power of Vector Engine.
To facilitate tuning, compiler switches and compiler directives can be used to give the compiler some information that it cannot obtain from the source program by itself. By using these functions effectively, one can take good advantage of the power of VE.

4.2.2 Basic Conditions for Vectorization

The following sections describe the conditions needed for vectorization.

- Vectorization Subjects

Array expressions and loop structures are vectorization subjects. A loop structure consists of a DO construct or an IF construct and a GOTO statement. There are no constraints on the format of an array expression.

(1) Loop structure must satisfy the following conditions:

(2) The loop must have a single entrance and single exit.

(3) The iteration count for the loop must be determined before the loop is entered.

(4) The compiler determines that the following loop structure satisfy these conditions.

(5) For a DO construct, no statements can pass the control of the program from inside the loop to outside.

The index-variable and the loop control variable:

- Must be a 4-byte, 8-byte, or integer type scalar variable,
- Must not associated with other variables defined in the loop
- Must be defined once in the loop body.

(6) The increment parameter must be a constant and the relational operation must be .LT. / .LE. / .GT. / .GE.

The following are examples of vectorizable DO loops.

Example 1	Example 2

DO I=1,N
  X(I)=Y(I)*Z(I)
ENDDO
	**DO WHILE(I.LT.10)
	  I=I+1
	  X(I)=Y(I)*Z(I)
	ENDDO**

The following are examples of DO loops that are not vectorizable.

Example 1	Example 2	Example 3
Reason: loop control variable is defined twice in the loop body	**Reason: loop control variable is not a type integer**	**Reason: relational operation is does not satisfy the conditions.**

**DO WHILE(I.LT.10)
  I=I+1
  IF(X(I).LT.0.0)
    I=I+1
ENDDO**
	**DO WHILE(R.LT.10.0)
	  R=R+1.0
	ENDDO**
		**DO WHILE(I.NE.10)
		  I=I+1
		ENDDO
		DO WHILE(I.GT.0)
		  I=I+1
		ENDDO**

For an IF construct, no statements can pass the control of the program from inside the loop to outside. The following are examples of vectorizable IF loops.

Example 1	Example 2

**10 IF(I.GT.10) GOTO 20
  I=I+1
  :
  GOTO 10
20 CONTINUE**
	10 J=J+1
	  :
	  IF(J.LT.10)
	    GOTO 10

The following are examples of DO loops that are not vectorizable.

Example 1	Example 2	Example 3
Reason: Scalar logical expression conditions are not satisfied.	**Reason: Control structure does not satisfy conditions**	**Reason: Relationship of increment parameter of loop-control-variable to scalar-logical-expression does not satisfy conditions.**

**10 IF(X(I).LT.0.0) GOTO 20
  I=I+1
  :
  GOTO 10
20 CONTINUE**
	**10 X(I)=Y(I)*Z(I)
	  IF(I.GT.10) GOTO 20
	  I=I+1
	  :
	  GOTO 10
	20 CONTINUE**
		**10 IF(I.NE.10) GOTO 20
		  I=I+3
		  :
		  GOTO 10
		20 CONTINUE**

- Vectorizable Statements

  An array expression is subject to vectorization for all executable statements it appears in. The following executable statements are subject to vectorization when they appear

in a loop structure that is subject to vectorization.

Assignment statements

CONTINUE	**CYCLE**	**EXIT**	**ELSE IF**	**END IF**	**Arithmetic IF**
GOTO	**IF THEN**	**ELSE**	**END SELECT**		**statement**
IF	**CASE**	**CASE**			
SELECT	**ENDDO**				
DO					

The following executable statements are not subject to vectorization.

WHERE	**ENDWHERE**	**ELSEWHERE**
WRITE	**READ**	**PRINT**
RETURN		
CALL		

NOTE: The WHERE, ENDWHERE, and ELSEWHERE statements are vectorized as array expressions. If the following executable statements appear in a loop structure, the loop is not vectorized.

Assigned GOTO statement	**STOP**	**ALLOCATE**
Computed GOTO statements	**PAUSE**	**DEALLOCATE**
Pointer assignment	**OPEN, CLOSE, REWIND,**	**NULLIFY**
statements	**BACKSPACE, ENDFILE, INQUIRE**	

- Vectorizable Types

The following types can be vectorized.

4-byte integer types	**Complex types**
8-byte integer types	**Double-precision complex types**
Real types	**4-byte logical types**
Double-precision real types	**8-byte logical types**

2-byte integer types, quadruple-precision real types, quadruple-precision complex types, single-byte logical types, character types, and derived types are not subject to vectorization.

- Vectorizable Operations and Assignments

The intrinsic operations that are subject to vectorization are:

(1) Numeric

(2) Logical

(3) Numeric relational

An operand must be a constant, scalar variable, structure component, or array element whose type can be vectorized, an argument of an intrinsic function that can be vectorized, or an expression that can be vectorized.

Character intrinsic operations, character relational intrinsic operations, and defined operations cannot be vectorized.
The intrinsic assignments that can be vectorized are:

(1) Numeric

(2) Logical

The left-hand side of the assignment statement must be a variable that can be vectorized. Character intrinsic assignments, pointer assignments, and defined assignments can be vectorized.

- Vectorizable Procedures

(1) Some intrinsic functions referenced in a loop or an array expression are replaced by vector versions.

(2) Some intrinsic functions are unvectorizable and the loop that references them is partially vectorized.

(3) When other procedures appear within a loop, the entire loop is not vectorized.

(4) A SPLIT compiler directive is supplied to partially vectorize a loop which references those procedures.

Refer to Fortran Compiler User's Guide for a list of supported intrinsic functions.

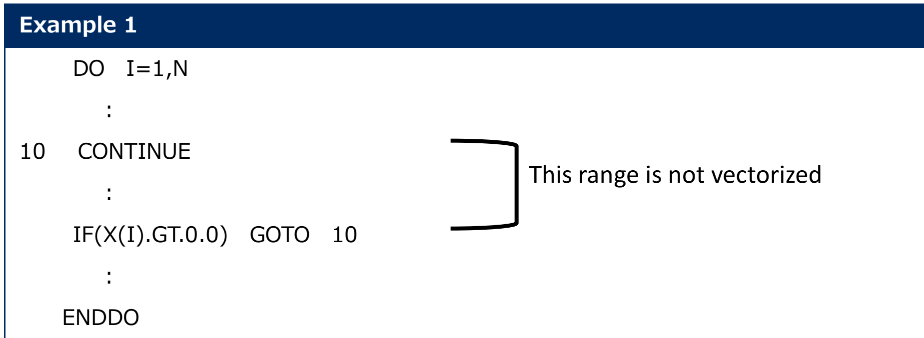
- Vectorizable Control Structures

A WHERE structure is subject to vectorization as a whole. IF constructs and CASE constructs in a loop structure are subject to vectorization. If there is a large number of CASE blocks in a CASE construct, the entire loop may not be subject to vectorization.

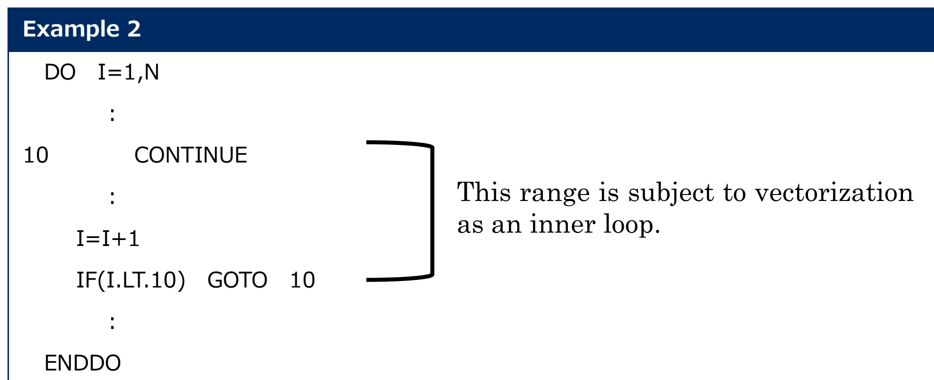
A branch within a loop structure is vectorized if the branch target comes after the branch. If the target comes before the branch (called a backward branch), then all statements in the range from that branch to the branch destination are not subject to vectorization.

If a backward branch constitutes a vectorized loop structure, the branch is subject to vectorization.

The following are examples of backward branching.



Below is a backward branch that constitutes a loop



4.2.3 Data Dependency Conditions

When a loop structure is vectorized, the order in which operations execute after vectorization differs from the order in which they execute before vectorization. If a loop is to be vectorized, the order of defining and referencing variables and array elements that appear within the loop must not be changed even when they are vectorized. The conditions on the relationship between defining and referencing a vectorizable variable are as follows.

If the same variable appears more than once in the loop, then either all the occurrences of the variable must be references, or the definition must precede the reference on all execution paths in the loop. If elements associated by the EQUIVALENCE statement appear more than once in the loop under different names, all of them must appear only as references. A macro operation such as inner product, element sum, maximum value, or minimum value may be vectorized even if the above condition is not satisfied.

Example 1: Reference preceding definition

```
DO I=1 ,N
      A(I)=X
      X=B(I)+C(I)*D(I)
END DO
```

Example 2: Definition always preceding reference
DO I=1,N
X=
Z=
IF( )THEN
Y=
=Y
ELSE
=X
END IF
=Z
END DO

An index variable is an exception to this rule. An index variable is an integer variable that is either:

- A DO variable, or
- An integer variable whose value changes by a constant increment at each loop repetition, and is defined at only one place in the loop.

Example: I, J, and K are index variables
DO I=1 ,N
J=J+1
K=I+3
.
.
.
END DO

If the same array element appears more than once in the loop, the relationship between its definition and reference must be maintained after vectorization. This condition must be satisfied because the execution order of statements is changed by vectorization. The following example shows how vectorization can change the relationship between array definitions and references, producing incorrect results.

Example
DO I=1, 6
A(I)=3 .0
B(I)=A(I+1)
END DO

Execution order when the example loop is not vectorized	Execution order when the example loop is vectorized
A(1)=3.0	**A(1)=3.0**
B(1)=A(2)	**A(2)=3.0**
A(2)=3.0	**A(3)=3.0**
B(2)=A(3)	.
A(3)=3.0	.
B(3)=A(4)	**B(1)=A(2)**
.	**B(2)=A(3)**
.	**B(3)=A(4)**
.	.
.	.

To maintain the correct relationship between array element definitions and references after vectorization, one of the following conditions must be satisfied.

Condition A: The array elements are not defined in the loop and are only referenced.

Condition B: If an array is defined at a point, it is not defined or referenced at any other point. If the array is defined or referenced at another point, elements defined or referenced there are completely different from the former.

Condition C: If an element of the array is defined in each iteration of the loop, or is defined and referenced, the order of the definitions and references must be in an order such that the compiler can resolve them. In other words, if the array element is defined in the pth statement of a loop and referenced in the qth statement of the same loop, then the loop variables i and j should be related as:

$i > j \rightarrow$ when $p > q$ and $i \neq j$

$i < j \rightarrow$ when $p < q$ and $i \neq j$

When the array element is defined and referenced in the same statement, then

$i > j \rightarrow$ when $p = q$ and $i \neq j$

Please note that there is no problem if both array elements are defined, or are defined and referenced in the same iteration (i=j).

Although the compiler can easily determine whether the Condition A is satisfied, it cannot always determine whether the conditions B or C are satisfied. The compiler can determine whether conditions B and C are satisfied as follows:

- For a pair of array elements whose subscript values are different and also constant as the DO loop progresses, condition B is satisfied.
- For array elements having linear subscripts, condition B or C is satisfied under these conditions:

(1) A pair of array elements whose subscript values increment as iteration of the loop progresses.

(2) The subscript value of the array element in the preceding statement is greater than or equal to the subscript value of the array element included in the later statement.

At the second iteration, the subscript value of array A is 2 in the first statement and 3 in the second statement. Accordingly, conditions (2) and (3) are not satisfied.

Example

```
DO I=1,N                !Loop not vectorized
      A(I)=
           =A(I+1)
END DO
```

- For a pair of array elements whose values decrease as iteration of the loop progresses, the reverse is obtained. Since the subscript values of array elements A(I) and A(I+l) both decrement, and the subscript value of the array element included in the later statement is apparently larger, condition (3) is satisfied.

Example

```
DO I=1,N,-1             !Loop vectorized
      A(I)=
           =A(I+1)
END DO
```

- For a pair of array elements where one subscript value increments and the other decrements (or either of the subscript values is constant as the loop progresses), if the relationship of their subscript values does not change as iteration of the loop progresses, condition (2) is satisfied.
- If an array element having a nonlinear subscript is included in the definition and/or reference, conditions (2) and (3) are not satisfied.
- If array elements associated by the EQUIVALENCE statement are included more than once in the loop under different names, and if at least one of them is included in a definition, conditions (2) and (3) are not satisfied. If possible, make the names the same and then check conditions (2) and (3).
- When the compiler cannot determine whether conditions are satisfied, it concludes that the conditions are not satisfied.

  (1) If the compiler cannot determine whether the relationship between definition and

reference is maintained correctly, the user can inform the compiler that the relationship is correctly maintained by using a compiler directive.

(2) A linear subscript is an expression that satisfies the following conditions:

- It is of the integer type.
- It contains only addition, subtraction, multiplication, and exponentiation, in which the base is not an index variable and the exponent is an unsigned integer constant.
- Only one index variable appears in each dimension.
- It does not contain parentheses.
- It contains only constants, index variables, and relative constants. A relative constant is one of the following:

a) A variable or array element that is only referenced in the loop

b) An intrinsic function that has only a constant or relative constant as an actual argument (except for the intrinsic functions that generate random numbers).

NOTE: Vectorization can be done even if the preceding conditions are not satisfied, provided the operation is an iteration type. Automatic modification can also be performed to satisfy the preceding conditions for vectorization by replacing statements or using work vectors.

A vector with a linear subscript is processed as a continuous or constant stride vector. The subscript expression is nonlinear if it does not satisfy these conditions, and the subscript value changes during loop iteration. A vector with a nonlinear subscript is processed as a list vector. Figure 1 shows vector types.

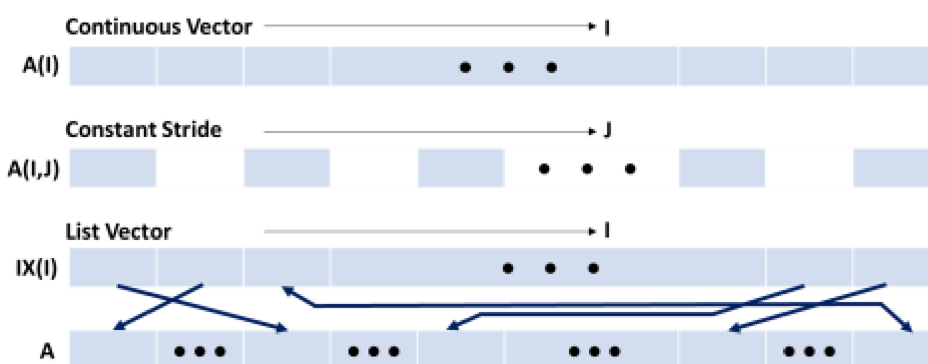


Figure 1 Types of Vectors

The following examples show the relationship between definition and reference when maintained correctly.

Example 1: Only references appear in the loop	Example 2: A(I) and A(I+l) do not overlap	Example 3: A(I) and A(I+1) may overlap	Example 4: A(I) and A(I+1) may overlap
DO 1=1,N	DO 1=1,N,2	DO I=1,N	DO I=N,1,-1
=A(I)	=A(I) !A(1),A(3),..	=A(I+1)	=A(I)
=A(I+1)	A(I+1)= !A(2),A(4),..	A(I)=	A(I+1)=
END DO	END DO	END DO	END DO

In Example 3 and 4, the relationship between definition and reference is maintained correctly after vectorization, despite possible overlap.

The following examples show the relationship between definition and reference when not correctly maintained.

Example 1	Example 2	Example 3
DO I=1,N	DO I=N,1,-1	DO 1=1,N
=A(I)	A(I)=A(I+2)*B(I)+C(I)	A(2*I+3)=
A(I+1)=	END DO	A(3*I-1)=
END DO		END DO

The following examples show how the compiler cannot determine whether the relationship between definition and reference is maintained correctly.

Example 1	Example 2
If K is greater than zero, the relationship is maintained correctly, but the compiler cannot determine this.	If L is equal to or less than zero, the relationship is maintained correctly, but the compiler cannot determine this.
DO I=1,N,K	DO I=1,N
A(I)=A(I+2)+C(I)	A(I) =
END DO	= A(I+L)
	END DO

4.2.4 Improving the Vectorization Ratio

There are three major ways to raise the vectorization ratio:

- Locate loops that were eligible for vectorization, but have not been vectorized or were only partially vectorized. Remove the cause of non-vectorization so that the part can be vectorized.
- Find vectorizable parts that are not eligible for vectorization and rewrite the program so that they can be vectorized keeping strict focus on maintaining the program

behavior.

- Revise the algorithms used for a part of or the entire the program to make them suitable for vectorization.

In the first two techniques, the program is revised to a limited extent while maintaining its structure and algorithms. When that approach is ineffective, the program should be reviewed to see if there are other algorithms more suitable for vectorization. When algorithms used in a program are not suitable for vectorization, the third technique has much greater effect. This section explains the first two techniques from the programming standpoint.

The vectorization ratio can be improved by removing the cause of non-vectorization. When loops are eligible for vectorization, but are not vectorized or are only partially vectorized by the compiler, the cause is indicated in a vectorization diagnostic message. The user may be able to raise the vectorization ratio by removing the cause. The following typical examples show how these conditions can be removed.

- Loops in which the compiler cannot determine whether the correct dependency between definition and reference would be maintained.

Example 1	
ORIGINAL	**VECTORIZED**
	!NEC$ ivdep(X)
DO J=1,N	DO J=1,N
X(J-1)=X(J-1)-X(JW)*Y(J)	X(J-1)=X(J-1)-X(JW)*Y(J)
JW=JW+1	JW=JW+1
END DO	END DO

In this example, the compiler cannot determine whether the correct dependency between definition and reference (X(J-1) on the left side and X(JW) on the right side) would be maintained because the initial parameter of JW is unknown. If the correct dependency can be maintained, the following vectorization directive ‘ivdep’ may be placed immediately before the DO loop.

In the following example, the compiler cannot determine whether the correct dependency between definition and reference of H(IX(I)) would be maintained by vectorization.

Example 2	
ORIGINAL	**VECTORIZED**
	!NEC$ ivdep(H)
DO I=1,N	DO I=1,N
IX(I)=IA(I)-IB(I)*IC(I)	IX(I)=IA(I)-IB(I)*IC(I)
H(IX(I))=H(IX(I))+1.0	H(IX(I))=H(IX(I))+1.0
END DO	END DO

If all N values from IX(1) to IX(N) are different, the correct dependency between definition

and reference of H(IX(I)) is maintained by vectorization, so the entire loop can be vectorized by inserting the following vectorization directive immediately before the loop.

- Loops containing user defined procedure references to one of the following methods can be used to vectorize loops containing external, internal or module procedure calls.

(1) Inline expansion

Expand the procedure directory at the point of reference. The automatic inline expansion function is supplied for this.

Example	
ORIGINAL	**VECTORIZED THROUGH INLINE EXPANSION**
DO I=l,N **CALL MAT(A(I),B(I),C(I),D(I),X,Y)** **ENDDO** **SUBROUTINE MAT(S,T,Y,V,A,B)** **A=S*U+T*V** **B=S*V-U*T** **RETURN** **END**	**DO I=l,N** **X=A(I)*C(I)+B(I)*D(I)** **Y=A(I)*D(I)-B(I)*C(I)** **ENDDO**

(2) SPLIT compiler directive

If the procedure satisfies the following conditions, the SPLIT compiler directive can be applied to the loop to vectorize.

- No STOP, PAUSE or input-output statements.
- Elements in common blocks or variables referenced in the loop are not accessible by the user, host, or pointer association in the procedure.
- Elements in common blocks and variables accessible by the user, host, or pointer association in the procedure are not defined in the loop.
- The dummy arguments that correspond to array elements, specified as actual arguments, are all scalar variable names.
- No pointer associations are changed.
- No function arguments are defined.
- Function does not have a pointer attribute.
- The result of the procedure does not depend on execution time or count.

- Does not have an alternate return asterisk in a dummy argument.

CAUTION: If SPLIT is applied to a loop containing a procedure call that does not satisfy the conditions, the loop will be vectorized, but may produce incorrect results.

4.2.5 Improving Vector Instruction Efficiency

When a DO loop is vectorized, it executes faster, but the speed-up depends on the types of vector instructions generated, the loop iteration count (also called the loop length or vector length), and the array reference patterns.

4.2.6 Lengthening the Loop

Before a vectorized loop is executed, some preparatory processing must be accomplished for each vector instruction before the arithmetic begins. Since the processing time (start-up time) is almost constant regardless of the loop length, if the loop length is small and the actual vector arithmetic operations are not lengthy, the effect of start-up time significantly reduces the efficiency of vectorization.

The cross length is the loop length at which the execution times of the vectorized and unvectorized loops are equal. In case of VE the crossover length usually varies from 5 to 10, depending on conditions.

To maximize the effect of vectorization, therefore, the loop length should be made as long as possible. See the following figure.

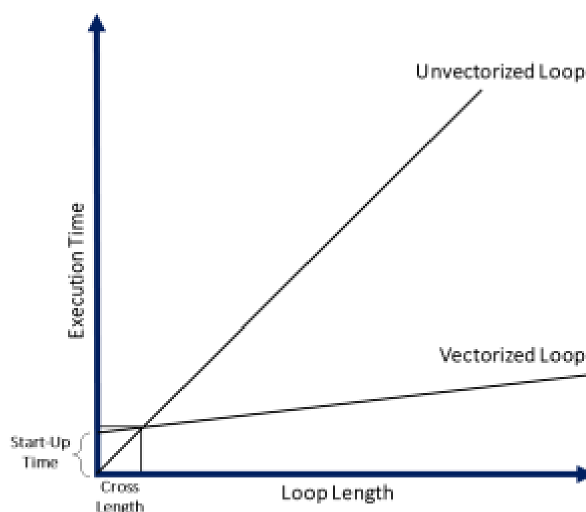


Figure 2 Start-Up Time and Cross Length

- In nested loops, interchange loops by interchanging rows and columns in a matrix, to

maximize the length of the innermost loop.

Example	
ORIGINAL	**LOOPS INTERCHANGED**
DO J=1,N ! N=10000 DO I=1,M ! M=10 A(I,J)=X*B(I,J)+C(I,J) END DO END DO	DO I=1,M ! M=10 DO J=1,N ! N=10000 A(I,J)=X*B(I,J)+C(I,J) END DO END DO

- Collapse a multiple loop to a single loop by converting a multidimensional array to a one-dimensional array. The compiler performs such transformations, when the loop interchange or loop collapsing function is enabled.

4.2.7 Improving Array Reference Patterns

Depending on how vector data is arranged, a vector can be continuous, constant stride, or a list vector. To process data by vector instructions, a vector must be loaded from memory and stored again after processing. It does not always take the same time to load a vector and write it to memory again.

Loading and storing speed is highest for a continuous or a constant stride vector with odd stride (the interval between elements is an odd number).

Figure 3 Constant stride vector

A constant stride vector whose element interval expressed as m*2n is even, is loaded and stored as follows

(m= odd number, n = 0, 1, 2, 3 ...).

- 4-byte data

Load and store speeds are highest when n is 1 (interval: m * two elements) or less. When n is greater than 2, speed slows as n increases.

- 8-byte data

Load and store speeds are highest when n is 0 (interval: m * one element). When n is greater than 1, speed slows as n increases. The fall-off in speed is due to bank contention.

Since the speed of list vector loading and storing varies depending on bank contention, the speed is slower than for contiguous vector processing.

The following points should be noted.

- Array elements in a loop to be vectorized should be referenced so that the index variables, such as the loop index variables, appear in the first dimension wherever possible. The values of subscript expressions should increment or decrement by 1 (or an odd number) at each loop iteration.

Example	
ORIGINAL	**OPTIMIZED**
REAL,DIMENSION (100,100) :: A, B, C : : DO I=1,N DO J=1,N A(I,J)=B(I,J)+X*C(I,J) END DO END DO	REAL, DIMENSION (100,100) :: A, B, C : : DO J=1,N DO I=1,N A(I,J)=B(I,J)+X*C(I,J) END DO END DO

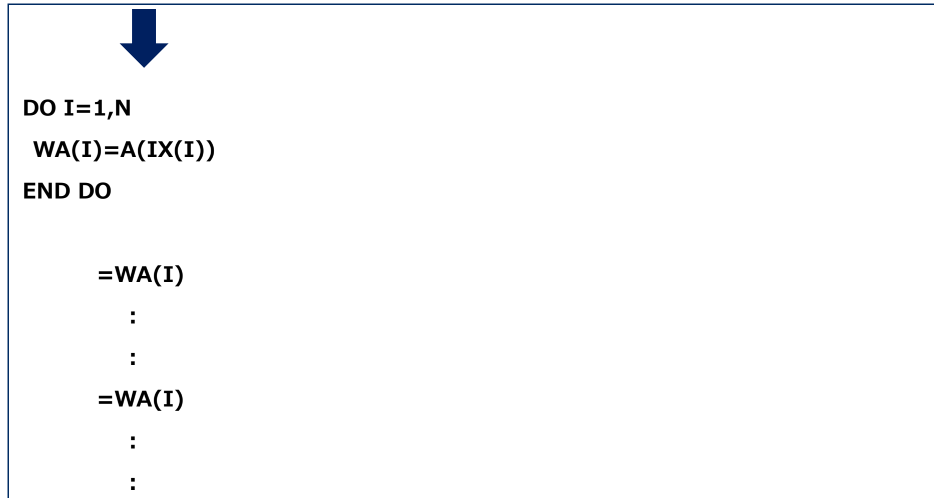
- If an index variable appears in the second dimension of a two-dimensional array (or a higher dimension, in general), the size of the first dimension should be longer in the array declaration.

Example	
ORIGINAL	**OPTIMIZED**
REAL, DIMENSION (1024,1024) :: A, B : : DO K=1,N S=S+A(I,K)*B(K,J) END DO	REAL, DIMENSION (1056,1024) :: A, B : : DO K=1,N S=S+A(I,K)*B(K,J) END DO

- The index variable should appear in the first dimension if possible, and should increment or decrement by 1.

- When an array is used as a list vector and it is referenced by the same subscript several times, the programmer should provide and use a work array, to transfer necessary data in advance. If necessary, data can be returned to the original array after processing.

Example	
= A(IX(I)) : : = A(IX(I)) : :	This array is referenced in the form of A(IX(I)) repeatedly.



In this example, correct results are not obtained if there are definitions for A(IX(I)). Elements with different IXs can have the same value.

4.2.8 Removing IF Statements

By using the vector mask, compress, and expand functions, the compiler is able to vectorize loops that contain IF statements. When a loop is vectorized by masked vector operations, the execution time is the same as when they are not masked. See the following example.

Example 1	Example 2
ORIGINAL	OPTIMIZED

DO I=1,1000
IF(e)THEN
A(I)=B(I)*C(I)
END IF
END DO
	DO I=1,1000
	A(I)=B(I)*C(I)
	END DO

The assignment statements in both examples take the same execution time. Additional time for generating a mask is also required in Example 1. Furthermore, if e is true only 1% of the time, multiplication and assignment are carried out only 10 times when the loop is not vectorized. However, it takes the time required for vector arithmetic on all 1000 elements when the loop is vectorized. This significantly reduces the effect of vectorization.

The compress and expand functions avoid this problem. However, they do not raise the efficiency significantly, because it takes time for compression and expansion. For this reason, the compiler uses these functions only when several operations are carried out, such as when the loop contains intrinsic functions.

Removing IF statements can raise the efficiency of vectorization. The following examples illustrate this.

- Whenever IF statements are used to avoid unnecessary operations as shown, simply remove them.

Example 1	Example 2

DO I=1,N
    IF(A(I).NE.0.0)THEN
        D(I)=A(I)*(B(I)+C(I))
    ELSE
        D(I)=0.0
    END IF
END DO
	DO I =1,N
	    D(I)=A(I)*(B(I)+C(I))
	END DO

- Move special processing, performed only at a particular iteration count, outside the loop.

Example 1		
ORIGINAL	**SIMILAR TO ORIGINAL**	**OPTIMIZED**

DO I=1,N
    α
    IF(I.EQ.K)THEN
    β
    END IF
    γ
END DO
	DO I=1,K-1
	    α
	    γ
	END DO
	I=K
	    α
	    β
	    γ
	DO I=K+1,N
	    α
	    γ
	END DO
		DO I=1,N
		α
		γ
		END DO
		I=K
		α
		β
		γ

- Compression and expansion outside the loop and use of list vectors

When IF statements are used to avoid unnecessary operations on irrelevant data, such as zeros in a sparse matrix, they can be removed. Use compression to do this by gathering only the relevant data in consecutive work array in advance. Arithmetic operations are performed on these work arrays in the loop, then the data from the work array are returned to the original array (expansion).

Example 1	
ORIGINAL	OPTIMIZED

DO I=1,N
IF(A(I).NE.0.0)THEN
C(I)=A(I)*SIN(B(I))
:
:
END IF
END DO
	K=0
	DO I=1,N
	IF(A(I).NE.0.0)THEN
	K=K+1
	IX(K)=I
	AA(K)=A(I) ! Compression
	BB(K)=B(I)
	:
	:
	END IF
	END DO
	
	DO I=1,K
	CC(I)=AA(I)*SIN(BB(I))
	:
	:
	END DO
	
	DO I=1,K
	C(IX(I))=CC(I)
	:
	Expansion
	:
	END DO

Another method is to generate vectors containing only the indexes of the relevant data (list vectors) without performing data compression and expansion. These vectors can then be used as subscripts directly in the loop.

Example 2	
ORIGINAL	OPTIMIZED

DO I=1,N
IF(A(I).NE.0.0)THEN
C(I)=A(I)*SIN(B(I))
:
:
END IF
END DO
	K=0
	DO I=1,N
	! Generation of list vector
	IF(A(I).NE.0.0)THEN
	K=K+1
	IX(K)=I
	END IF
	END DO
	DO I=1,K
	C(IX(I))=A(IX(I))*SIN(B(IX(I)))
	:
	:
	END DO

If the number of arithmetic operations on the compressed elements is significant, compression and expansion is more efficient. If the number is small, using list vectors is more efficient. Data compression and expansion should be performed before and after a block of processing, instead of at each loop.

4.2.9 Avoiding Iterative Operations

Iterative operations are vectorized by using special vector instructions, but these vector instructions, although faster than scalar instructions, are slower than other vector instructions. Therefore, they should be avoided whenever possible.

Example

ORIGINAL	OPTIMIZED
DO I=1,M ! No vector iterative operation	DO J=1,N ! Vector iterative operation
DO J=1,N ! Vector iterative operation	DO I=1,M ! No vector iterative operation
A(I,J+1)=X(I,J)-T(I,J)*A(I,J)	A(I,J+1)=X(I,J)-T(I,J)*A(I,J)
END DO	END DO
END DO	END DO

4.2.10 Avoiding Loop Division

The compiler partially vectorizes loops that contain unvectorizable parts by dividing the loop before and after those parts. Dividing a loop leaves part of the loop unvectorized, lowers efficiency at each division point, and lowers the efficiency of the vectorized part. It is better to remove the cause of nonvectorization, so that the entire loop can be vectorized. If the unvectorizable parts cannot be removed, reduce the number of divisions either by gathering the unvectorizable parts into one block or by moving them at the end of the loop. The efficiency of vectorization is thus improved.

4.2.11 Avoiding Loop Unrolling for Short Loops

Loop unrolling lengthens the body of a loop by a factor of n, in order to reduce the iteration count to 1/n of its value. This technique is frequently used to gain speed. However, unrolling a vectorizable DO loop may lower efficiency by reducing the loop length or converting continuous vectors to noncontinuous vectors. Unrolling vectorizable short DO loops should be avoided.

Example

ORIGINAL	OPTIMIZED
DO I=1,97,3 ! Short Loop Length	DO I=1,99
S=S+A(I)*B(I)+A(I+1)*B(I+1) !Unrolling	S=S+A(I)*B(I) !No unrolling
& +(I+2)*B(I+2)	END DO
END DO	

Unrolling an outer loop that is not vectorized is generally effective, especially if it reduces the number of references to a memory area.

Example	
ORIGINAL	**OPTIMIZED (Outer loop unrolling)**

DO J=1,100
    DO I=1,N
    X(I)=X(I)+A(I,K)*B(K,J)
    END DO
END DO
	DO J=1,99,2
	    DO I=1,N
	        X(I)=X(I)+ A(I,K)*(B(K,J)+ B(K,J+1))
	    END DO
	END DO

In this case, the count of load count and count of store for X(I) and the count of load for A(I,K) are halved by unrolling.

4.2.12 Increasing Concurrency

Vector addition, subtraction, multiplication, vector shift operations (including multiplying a real number by 2 or by 1/2), and logical operations can be executed in parallel. Thus, it is efficient to put as many of these operations together in the same loop as possible.
In general, complex loops result in higher overall performance as compared to relatively simple loops.

Example	
ORIGINAL	**OPTIMIZED**

DO I=1,N
    A(I)=B(I)+C(I)
END DO
DO I=1,N
    X(I)=Y(I)*Z(I)
END DO
	DO I=1,N
	    A(I)=B(I)+C(I)
	    X(I)=Y(I)*Z(I)
	END DO

To make instruction reordering by the compiler efficient, as many arithmetic operations as can be executed in parallel should be performed.

4.2.13 Avoiding Arithmetic Division

Since vector division is slower than other vector arithmetic operations, minimize the number of divisions by converting them to multiplication or use algorithms that do not contain division.

4.2.14 Using Vectorization Options and Directives

Effective use of vectorization options and vectorization directives can raise the efficiency of vectorization.

- novector directive: The novector directive should be used in the following cases.

(1) The loop is so short that vectorization would cause a loss rather than a gain in speed.

(2) Most of the loop is controlled by IF statements and is only rarely executed. Vectorization of such a loop by the masked vector function would cause a loss rather than a gain in performance.

(3) When a loop has an out-of-loop branch and is vectorized, only those arithmetic operations participate in the vectorization that occur before the branch-exit statement. If the branch-statement occurs at an early iteration, the loop practically behaves like a short loop and vectorization may result in reduced performance. novector may be preferred in such cases.

Example

```
!NEC$ NOVECTOR
DO I=1,1000
    IF(A(I)-B(I)LT.1.0E-10) EXIT
    Z(I)=A(I)-B(I)
END DO
```

(4) When a program is executed in scalar mode to test the effect of vectorization on accuracy.

(5) When a program is executed in scalar mode to observe the frequency or locations of exceptions.

- loop_count directive:

In vectorization, the compiler needs to know the iteration count of a loop for the following purposes.

(1) To determine the size of an array.

(2) When a loop is created to replace variables defined (or referenced) that extend across a division point.

(3) When a loop is vectorized by dividing the loop.

(4) To generate efficient vector instructions when the iteration count is less than the vector register length.

(5) To perform efficient register allocation on the basis of the iteration count.

(6) When the iteration count cannot be determined, the compiler makes one of the following assumptions.

(7) If the -floop-count option of the compiler directive specifies the iteration count explicitly, that value is used as the iteration count.

- If the -floop-count option specifies the iteration count explicitly, that value is used as the iteration count. If the value determined by the next method is less than this value, the value determined by the next method is used.

(8) When the upper limit of the iteration count can be inferred from an array declaration appearing in a loop, that value is used. A dummy array, whose highest dimension is declared as 1, is processed as an assumed-size array declarator. If the compile time option or the vectorization directive NOASSUME is used, array declarations are not used in iteration-count assumption.

(9) If the iteration count cannot be determined by these methods, 5000 is assumed.

The iteration count assumed by the compiler is shown in the vectorization diagnostic message. In the following example a loop count of 50 is assumed from the size of the dimension corresponding to the DO variable, I, in the array declaration. However, if a value less than 50 was specified by the –floop-count option, that value would be used.

Example

```
REAL A(100,100),B(200,100),C(50,100)
    :
DO I=1,N
      A(I,J)=B(I,J)*C(I,J)
ENDDO
```

The upper limit of the iteration count cannot be inferred from the array declaration. Therefore, if the -floop-count=n option is specified, that value is used; if not, 5000 is assumed. See the following example.

Example

```
SUBROUTINE SUB (A,B,C,L,M,N)
REAL A (L,M),B(100,*),C(N,1),IX(L)
   :
   :
DO I=1,K
      A(I,J)=B(IX(I),J)*C(J,I)
ENDDO
```

If the actual iteration count is checked during execution and found to be greater than the value assumed by the compiler, error is output.

- verror_check and noverror_check directive:

When the noverror_check option is valid, no check is performed for invalid arguments. As a

result, less time is required for calculating the value of the function. The noverror_check option must be used only when no valid parameter will be passed to the function.

4.2.15 Other Effective Techniques

- Avoiding the use of POINTER attribute

  If the POINTER attribute is used, the compiler cannot fully analyze the dependency between definition and reference and may assume that the dependency cannot be determined, when in fact vectorization is possible. For this reason, the POINTER attribute should be avoided.

- NOOVERLAP compiler directive is supplied to declare that variables do not associate with others.

Example
REAL,DIMENSION(:),POINTER::X
REAL,DIMENSION(100),TARGET::Y
DO I=l,N
X(I)=Y(I)*2.0
ENDDO

Since X may be associated with Y, the compiler assumes data dependency in the loop. Therefore, the loop is unvectorized. If X is never associated with Y, you can specify the following compiler directive:

!NEC$ NOOVERLAP(X,Y)

Then the compiler will vectorize the loop.

- Use variables for work space instead of using arrays.

Example 1	Example 2
DO I=1,N	DO I=1,N
X=A(I)+B(I)	WX(I)=A(I)+B(I)
Y=C(I)-D(I)	WY(I)=C(I)-D(I)
E(I)=S*X+T*Y	E(I)=S*WX(I)+T*WY(I)
F(I)=S*Y+T*X	F(I)=S*WY(I)-T*WX(I)
END DO	END DO

In Example 2, WX and WY are loaded and stored in the loop. On the other hand, in Example 1, vector registers are assigned to X and Y and they are not loaded and stored in the loop. Thus, Example 1 is more efficient than Example 2.

- Additional features

When the basic conditions for vectorization are satisfied, the compiler performs as much vectorization as possible by transforming the source, especially loops and using special vector operations of the Vector Engine.

4.2.16 Vectorization by Statement Replacement

Consider replacement of statements if the conditions for vectorization are not satisfied in a loop structure by variable or array element definitions and references.

Example 1

```
DO  I=1,N
  A(I)=B(I)*C(I)   !definition
  E(I)=D/A(I+1)    !reference
END DO
```

↓

```
DO I=1,N
  E(I)=D/A(I+1)    !definition
  A(I)=B(I)*C(I)   !reference
END DO
```

Example 2

```
DO I=1,N
  B(I)=A(I)*C(I)   !definition
 A(I+1)=D+E(I)     !reference
END DO
```

↓

```
DO I=1,N
  A(I+1)=D+E(I)    !definition
 B(I)=A(I)*C(I)    !reference
END DO
```

Example 3

```
DO I=1,N
  A(I)=B(I)*C(I)   !definition
 A(I+1)=X*Y(I)     !reference
END DO
```

↓

```
DO I=1,N
  A(I+1)=X*Y(I)    !definition
 A(I)=B(I)*C(I)    !reference
END DO
```

4.2.17 Vectorization Using Work Vectors

When defining and referencing a variable or array element that does not satisfy vectorization conditions in a loop structure, if the definition precedes the reference, and the statement replacement (explained in Section 5.1.2.2) is impossible, vectorization conditions are satisfied by saving array values in a work vector. See the following example.

Example

```
DO  I=1,N                !definition
    A(I)=B(I)*C(I)       !reference
    B(I)=T*B(I)
    E(I)=B(I)+A(I+1)
END DO
```

↓

```
DO  I=1,N
    w(I)=A(I+1)         ! work vector w
    A(I)=B(I)*C(I)      ! definition
    B(I)=T*B(I)         ! Reference
    E(I)=B(I)+w(I)
END DO
```

4.2.18 Macro Operations

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

- Sum or inner product

The following shows a sum or inner product example.

S = S ± <exp>

where <exp> is an expression.

A sum or inner product like the following example that consists of multiple statements is also vectorized.

```
t1 = S ± <exp1>
t2 = t1 ± <exp2>
     :
S = tn ± <expn>
```

- Product

The following shows a product example. This example cannot be vectorized if S is a complex type.

S = S* <exp>

A product like the following example that consists of multiple statements can be vectorized.

```
t1 = S  * <exp1>
t2 = t1 * <exp2>
     :
S=tn * <expn>
```

- Iteration

Iterations shown in the following examples are vectorized unless X is a complex type.

```
X(I)=<exp> ± X(I-1)
X(I)=<exp> * X(I-1)
X(I)=<exp1> ± X(I-1) * <exp2>
X(I)=(<exp1> ± X(I-1)) * <exp2>
```

An iteration like the following example consists of multiple statements can be vectorized.

```
t=<exp1> ± X(I-1)
X(I)=t * <exp2>
```

- Function type

Example

```
DO  I=1,N
      XMAX=MAX(XMAX,X(I))
END DO

DO  I=1,N
      XMIN=MIN(XMIN,X(I))
END DO
```

- IF type

(1) Finds the maximum or minimum value only.

Example

```
DO  I=1,N
      IF(XMAX.LT.X(I))   THEN
         XMAX=X(I)
      END IF
END DO
```

(2) Finds the maximum or minimum value and its index.

Example

```
DO  I=1,N
      IF(XMIN.GT.X(I)) THEN
         XMIN=X(I)
         IX=I
      ENDIF
END DO
```

(3) Finds the index only.

Example

```
DO  I=1,N
      IF(X(IX).LT.X(I)) THEN
           IX=I
      END IF
END DO
```

(4) Finds the maximum or minimum value, and its index.

Example

```
DO  I=1,N
    IF(XMIN.GT.X(I, J) ) THEN
        XMIN=X(I, J)
        IX=I
        IY=J
    END IF
END DO
```

(5) Compares absolute values.

Example

```
DO  I=1,N
    IF(ABS(XMIN).GT.ABS(X(I)))  THEN
        XMIN=X(I)
    END IF
END DO
```

- Search

(1) A loop that searches for an element that satisfies a given condition is vectorized.

Example

```
DO  I=1,N
    IF(X(I).EQ.0.0)  THEN
        EXIT
    END IF
END DO
```

All of the following conditions must be satisfied.

- This is the innermost loop.
- There is just one branch out of the loop.
- The condition for branching out of the loop depends on repetition of the loop.
- There must not be an assignment statement to an array element before the branch out of the loop.
- All basic conditions for vectorization are satisfied except for not branching out of the loop.

- Compression

A loop for compressing elements that satisfy a given condition is vectorized.

Example

```
J=0
DO  I=1,N
     IF(X(I).GT.0.0)  THEN
          J=J+1
          Y(J)=Z(I)
     END IF
END DO
```

- Expansion

  A loop for expanding values to elements that satisfy a given condition is vectorized.

Example

```
J=0
DO  I=1,N
     IF(X(I).GT.0.0)  THEN
          J=J+1
          Z(I)=Y(J)
     END IF
END DO
```

4.2.19 Examples of Vectorization

- Simple array expressions and loops

Example 1

```
A(1:N)=B(1:N)+X*C(1:N)
```

Example 2

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

The array expression and DO loop in the examples are expanded using a vector multiplication instruction and a vector addition instruction.

$T_i \leftarrow X * C_i \quad (i=1, 2,\cdots, N)$

$A_i \leftarrow B_i + T_i \quad (i=1, 2,\cdots, N)$

- Multidimensional array expression

The first dimension of an array expression of two or more dimensions is vectorized.

Example

```
A(1:N, 1:M) = B(1:N, 1:M)+X*C(1:N, 1:M)
    ⬇
DO  j=1, M
     Tij ← X * Cij          (i=1, 2,⋯, N)
     Aij ← Bij+ Tij         (i=1, 2,⋯, N)
END DO
```

- Masked array assignment

A masked array assignment is vectorized using a vector mask generation instruction that generates a bit vector (mask vector) whose values are 1 or 0 depending on the truth of a condition expression for each element. A masked vector operation instruction is also generated that executes an operation only on elements corresponding to a 1 bit in the mask vector.

Example 1

```
WHERE(C(1:N).NE.0.0)   A(1:N)=B(1:N)/C(1:N)

Mi ← { 1  (if  Ci ≠ 0.0)
     { 0  (if  Ci = 0.0)      (i=1, 2,⋯, N)

Ai  ←  Bi / Ci  (if Mi = 1)   (i=1, 2, ⋯, N)
```

Mi is a bit vector used in vector operation mask control.

Example 2

```
WHERE(C(1:N).EQ.0.0)
     A(1:N)=B(1:N)+D(1:N)
ELSEWHERE
     A(1:N)=B(1:N)*C(1:N)+D(1:N)
ENDWHERE
```

```
M1i ← { 1 (if Ci ≠ 0.0)
      { 0 (if Ci = 0.0)       (i=1, 2,⋯, N)
Ai  ←  Bi * Ci (if M1i = 1)   (i=1, 2,⋯, N)
M2i ← NOT M1i                 (i=1, 2,⋯, N)
Ti  ← Bi * Ci (if M2i = 1)    (i=1, 2,⋯, N)
Ai  ← Ti + Di (if M2i = 1)    (i=1, 2,⋯, N)
```

M1i and M2i are bit vectors used in vector operation mask control.

Mi = 1 (if Ai ≥ 0)

     0 (if Ai < 0)

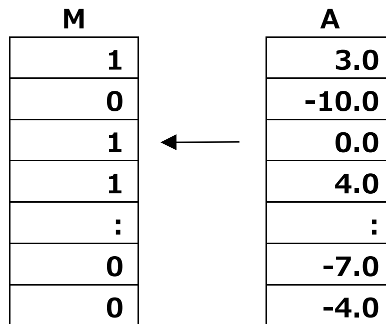


Figure 4　Vector Mask Generation Instruction

$C_i \leftarrow A_i + B_i$ (if $M_i = 1$)

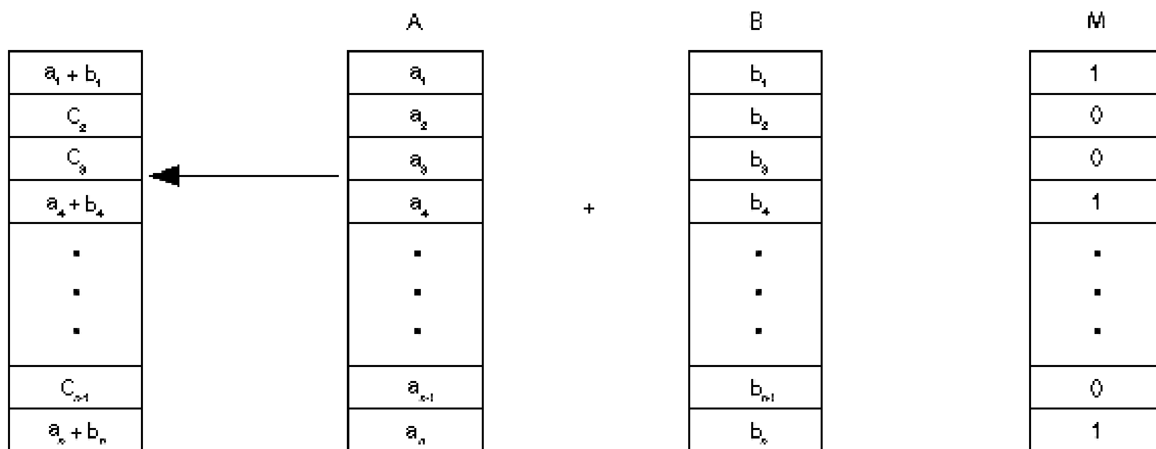


Figure 5　Masked Vector Operation Instruction

- Loops containing IF constructs

The method of vectorizing a loop that contains an IF construct varies depending on the form of the condition expression in the IF statement. If the value of the condition expression is invariant within the loop, the IF statement is expanded without change.

Example 1

```
DO   I=1, 100
      A(I)=B(I)*C(I)
      IF(ISW.EQ.1) THEN
            C(I)=D(I)+E(I)
      ENDIF
END DO


Ai ← Bi * Ci              (i=1, 2,…, N)
if   ISW.EQ.1   then
      Ci ← Di + Ei          (i=1, 2,…, N)
end if
```

When the value of the condition expression depends on the loop iteration, vectorization

uses a vector mask generation instruction and a masked vector operation instruction. See masked assignment previously shown.

Example 2

```
DO  I=1,N
     IF(C(I).NE.0.0)  THEN
          A(I)=B(I)/C(I)
     END IF
END DO
```

Mi ← 1 (if Ci ≠ 0.0)
 0 (if Ci = 0.0) (i=1, 2,⋯, N)
Ai ← Bi / Ci (if Mi = 1) (i=1, 2,⋯, N)
Mi is a bit vector used in vector operation mask control.

Vectorization is also possible when the IF construct is nested.

Example 3

```
DO  I=1,N
     IF(X(I).NE.0.0)  THEN
          IF(Y(I).GE.0.0)  THEN
               Z(I)=Y(I)/X(I)
          ELSE
               Z(I)=0.0
          ENDIF
     ENDIF
END DO
```

M1i ← 1 (if Xi ≠ 0.0)
 0 (if Xi = 0.0) (i=1, 2,⋯, N)
M2i ← 1 (if Yi ≥ 0.0 and M1i = 1)
 0 (if Yi < 0.0 or M1i = 0) (i=1, 2,⋯, N)
M3i ← M2i, AND M1i (i=1, 2,⋯, N)
Zi ← Yi / Xi (if M2i = 1) (i=1, 2,⋯, N)
Zi ← 0.0(if M3i = 1) (i=1, 2,⋯, N)

M1i, M2i, and M3i are bit vectors used in vector operation mask control

Array expressions and loops containing intrinsic function references

When there is a reference to an intrinsic function that is subject to vectorization in an array expression or loop, it is expanded into a series of instructions that reference a vector version intrinsic function that takes vector data as arguments and returns function values using vector data.

Example 1
A(1:N)=SQRT(X(1:N)*X(1:N)+Y(1:N))

Example 2
DO I=1, N **A(I)=SQRT(X(I)*X(I)+Y(I)*Y(I))** **ENDDO**

Either of the two previous examples can be expanded into the following series of instructions.

Ti ← Xi * Xi +Yi * Yi (i=1, 2,⋯, N)

Ai ← VSQRT (Ti) (i=1, 2,⋯, N)

VSQRT is the vector version SQRT function.

If the intrinsic function reference is within an IF condition, a series of instructions (that compress the vector data of the arguments using a vector compression instruction) call the vector function using the compressed vector data as arguments, and expand the vector data of the function value to vector data of the original size using a vector expansion instruction.

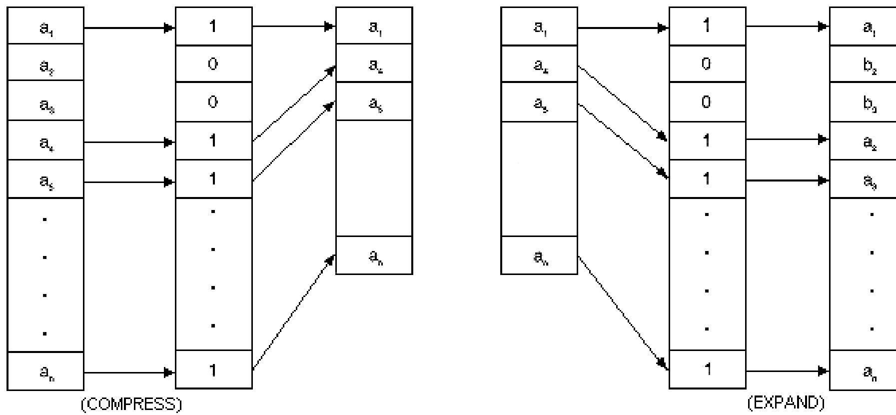


Figure 6 Vector Compression and Expansion

Some intrinsic functions are expanded into a series of vector instructions that compute direct function values.

Example 1
IX(1:N)= IAND (IY(1:N),IZ(I:N))

Example 2
DO I=1,N
IX(I)= IAND (IY(I), IZ(I))
END DO

Either of the previous two examples can be expanded into the following sequence of instructions.

Example 3	
IXi ← IYi & IZi	(i=1, 2,···, N)

- Loops containing index variables

  If an index variable is used anywhere within a loop except in an array subscript expression, it is vectorized by generating vector data for a number sequence.

Example	
DO I=1,N	
X(I)=I	
ENDDO	
Ti ← Ii	(i=1, 2,···, N)
Xi ← float(Ti)	(i=1, 2,···, N)
Ii is vector data in which the value of element i is i.	

- Vector subscripts

A reference in an array expression to a whole array or an array section (in which section subscripts are subscript triplets) or a reference inside a loop to an array element with linear subscripts is treated as a sequence vector or a constant stride vector.

A reference to an array section in an array expression (in which the section subscripts are vector subscripts) or a reference to an array element in a loop with subscripts that are nonlinear subscripts is treated as an indirect index vector. It is expanded into a series of vector instructions in which vector gather and scatter instructions are used.

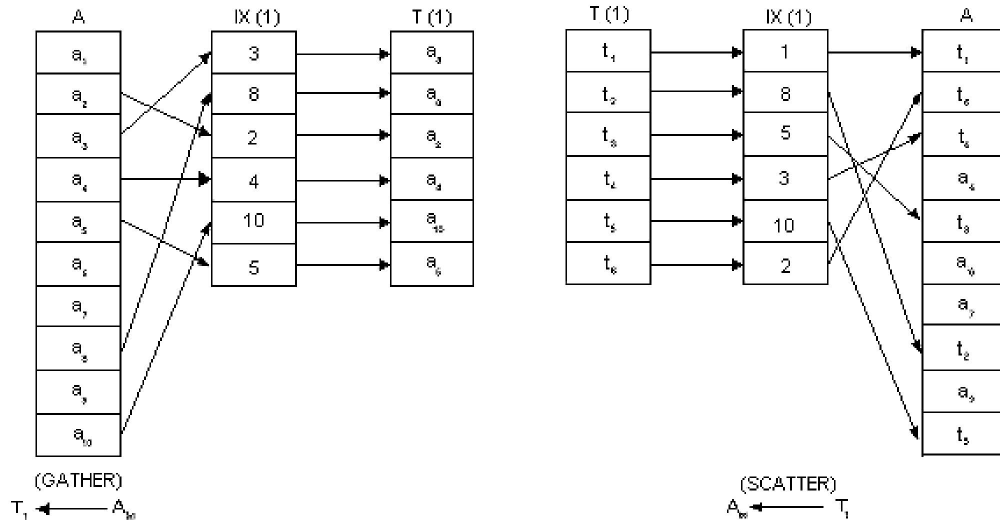


Figure 7 Vector Gather and Scatter Instructions

The following are examples of vector and nonlinear subscripts.

Example 1: vector subscripts

X(IX(1:N))=Y(IX(1:N)) * Z(1:N)

T1i ← Y(IXi) (i=1, 2,⋯, N)

T2i ← T1i * Zi (i=1, 2,⋯, N)

X(IXi) ← T2i (i=1, 2,⋯, N)

Example 2: nonlinear subscripts

DO I=I, N

X(I * I)=Y(I * I) + Z(I)

END DO

T1i ← Ii (i=1, 2,⋯, N)

ITi ← T1i * T1i (i=1, 2,⋯, N)

T2i ← YITi (i=1, 2,⋯, N)

T3i ← T2i * Zi (i=1, 2,⋯, N)

XITi ← T3i (i=1, 2,⋯, N)

Ii is vector data in which the value of element i is i.

4.2.20 Partial Vectorization

4.2.21 Code-Related Optimization

Code-related optimizations are designed to eliminate operations that are not needed as much as possible by analyzing the flow of control and the flow of data in a program. They shorten the execution time of a program by keeping the operations in a loop to the minimum

necessary and replacing an operation with an equivalent faster operation whenever possible. Optimizations related to vectorized code are shown as follows.

- Making multiplication and division special operations

  The operation processor provides instructions that execute doubling and halving for vector data, and using these effectively makes fast execution possible.

Example		
ORIGINAL	**VECTORIZATION**	**HALF INSTRUCTION**
DO I=1,N	T1i ← Ai * 0.5	T1i ← Half (Ai)
X (I) = A (I) * 0.5*B	T2i ← Bi * Ci	T2i ← Bi * Ci
(I) * C (I)	Xi ← T1i * T2i	Xi ← T1i * T2i
END DO		

  In this example, in order to use the multiplication/logic pipeline with each operation, Xi ← T1i*T2i cannot be executed until T1i← Ai*0.5 or T2i← Bi*Ci ends (the addition/shift and multiplication/logic pipelines each are two at a time). However, by replacing T1i ← Ai*0.5 with T1i ← Half(Ai), it is possible to begin at the same time through Xi ←T1i *T2i.

  This is because the operation pipeline used in the half instruction and the multiplication pipeline are executed in parallel.

- Deleting common expressions

If a given computation has already been performed in the same program, that computation is removed and replaced by a reference to the result of the previous computation in order to avoid an unnecessary computation. This is called common expression deletion.

Common expression deletion is also performed on vector operations.

Example		
ORIGINAL	**VECTORIZATION**	**HALF INSTRUCTION**
DO I=1,N	T1i ← B+SIN (Ci)	T1i ← B+SIN(Ci)
X(I)=A(I) * (B + SIN (C(I)))	Xi ← Ai * T1i	Xi ← Ai * T1i
Y(I)=D(I) + (SIN (C(I)) + B)	T2i ← SIN (Ci) + B	Yi ← Di + T1i
END DO	Yi ← Di + T2i	

Common expression deletion decreases the computation volume and number of references to intrinsic functions and shortens execution time.

- Extracting scalar operations

Although the operation processor can operate at high speed using operations between scalar data and vector data and vector instructions, operating on scalar data is more efficient than operating on scalar data and vector data. When an expression is evaluated using the normal evaluation sequence, a vector operation is sometimes required where a scalar operation could be used. The compiler is designed to shorten execution time by extracting such scalar operations and causing them to operate at the same time.

Example:		
ORIGINAL	**SCALAR OPERATION EXTRACTION**	**VECTORIZATION**
DO I=1, N **X(I) = A * Y (I) * B** **END DO**	**DO I=1, N** **X(I) = A * B * Y (I)** **END DO**	**T = A * B (scalar operation)** **Xi ← T *Yi (vector operation)**
Normal vectorization Ti←A*Yi (vector operation) Xi←Ti*B (vector operation)		

In this example, when normal vectorization is performed, vector multiplication of scalar data A and vector data Y and vector multiplication of the resulting vector data T and scalar data B are executed. However, when scalar operation extraction is performed, first scalar multiplication of scalar data A and B is executed and then vector multiplication of the resulting scalar data T and vector data Y is executed. Because of this, the number of vector operations is one less and the execution time is shortened.

This optimization is not performed when the compile time option -floop-nointerchange is specified.

- Making division multiplication

Since vector multiplication instructions are faster than vector division instructions, making division multiplication shortens execution time.

Example 1	
ORIGINAL	**VECTORIZATION**
DO I=1, N **A(I)=B(I)/X** **ENDDO**	T ← 1.0/X Ai ← Bi * T

Example 2	
ORIGINAL	**VECTORIZATION**
DO I=1, N	Ti ← Bi * Ci
X(I)=A(I)/B(I)/C(I)	Xi ← Ai / Ti
ENDDO	

This optimization is not performed when the compile time option -novwork is specified.

- Optimizing mask operations

Using masked operations makes vectorization possible for a DO loop containing an IF statement. However, if IF statements are nested to make a complex condition, identical operations may arise between masks, lowering execution efficiency. In order to avoid this, optimization is performed as follows for mask operations.

(1) Process identical operations as common expressions

In this example, A(I).GT.0.0 is processed as a common expression.

Example	
ORIGINAL	**VECTORIZATION**
DO I=1, N	M1i ← 0: if Ai > 0.0
IF (A(I).LE.0.0) THEN	1: if Ai ≤ 0.0
X(I)=A(I) * B(I)	
END IF	Xi ← Ai * Bi (if M1i = 1)
Y(I)=A(I) +B(I)	Yi ← Ai + Bi
IF (A(I).GT.0.0 AND. B(I). EQ.0.0) THEN	
Z(I)=A(I)	M2i ← 0: if Bi ≠ 0.0
END IF	1: if Bi = 0.0
END DO	
	M3i ← M1i AND M2i
	Zi ← Ai (if M3i =1)

(2) Common expression processing for nested IF statements

In this example, Y(I).GT.0.0 is processed as a common expression.

Example	
ORIGINAL	**VECTORIZATION**
DO I=1, N	M1i ← 0: if Xi > 0.0
IF (X(I). GT. 0.0) THEN	1: if Xi ≤ 0.0
IF (Y(I). GT. 0.0) THEN	
Z (I) = Y(I)/X (I)	M2i ← 0: if Yi > 0.0
ELSE	1: if Yi ≤ 0.0
Z (I) = 0.0	
END IF	M3i ← M1i AND M2i
ELSE	Zi ← Yi / Xi (if M3i = 1)

```
        IF (Y(I). GT. 0.0) THEN         M4i  ←  M1i  AND M2i
            Z (I) = X(I)/Y(I)           Zi   ←  0. 0 (if M4i = 1)
        END IF                          M5i  ←  M1i  AND M2i
    END IF                              Zi   ← Xi / Yi (if M5i = 1)
END DO
```

(3) Eliminating unneeded last value saves

When vectorizing the following DO loop, in order to process the variable X by considering it as an array, the value of X for I=N (last value) must be moved from the area that made an array to the area of X. However, if X is not referenced after execution of the DO loop, the last value of X need not be saved. In such cases, the instruction to save the last value is deleted to shorten execution time.

Example

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

(4) Code clean-up

Deletion of simple assignments and deletion of unneeded code are also performed for vectorized code.

4.2.22 Loop Transformations

Based on program structures, the most time consuming parts of a program are loops. Most high performance application work iteratively, giving rise to the need of optimization of the loop performance. Depending on the target architecture, the goal of loops transformations are:

- improve data reuse and data locality
- efficient use of memory hierarchy
- reducing overheads associated with executing loops
- instructions pipeline
- maximize parallelism

Understanding of the underlying system becomes necessary in order to identify the attributes for optimization. However, some general optimization techniques are presented below. Loop transformations can be performed at different levels by the programmer, the compiler, or specialized tools. At high level, some well-known transformations are commonly considered.

The compiler performs various loop transformations in order to improve performance of the vector code.

- Loop Collapse

  A loop collapse is effective in not only reducing the execution time for controlling iteration of the outer loop but also in increasing the loop iteration count and improving the efficiency of vector instructions.

ORIGINAL	COLLAPSED

DO N = 1, NMAX
  DO K = 1, KMAX
    ARR(N,K) = 0.D0
  END DO
END DO
	DO N = 1, NMAX*KMAX
	  ARR(N,0) = 0.D0
	END DO

ORIGINAL	COLLAPSED
Example 1 (in C)	

int a[100][300];
 	
for (i = 0; i < 300; i++)
    for (j = 0; j < 100; j++)
        a[j][i] = 0;
	int a[100][300];
	int *p = &a[0][0];
	
	for (i = 0; i < 30000; i++)
	    *p++ = 0;

Example 2 (in C)	

float a[100][100][100], b[100][100][100];
for (i = 1; i < n-2; i++) {
    for (j = 0; j < 100; j++) {
        for (k = 0; k < 100; k++) {
            a[i][j][k] = b[i][j][k];
        }
    }
}
	float a[100][100][100], b[100][100][100];
	
	for (i =0;i <n*10000-30000;i++) {
	      a[1][0][i] = b[1][0][i];
	}

ORIGINAL	COLLAPSED
Example 1 (in FORTRAN)	

REAL,DIMENSION(10,30)::A,B,C
DO J=1,30
  DO I=1,10
    A(I,J)=B(I,J)+C(I,J)
  END DO
END DO
	REAL,DIMENSION(10,30)::A,B,C
	DO IJ=1,30*10
	  A(IJ,1) = B(IJ,1) + C(IJ,1)
	ENDDO

Example 2 (in FORTRAN)	

REAL,DIMENSION(10,30)::A,B,C
do l = 2, lmax
  do k = 1, lmax - 1
    do i = 1, imax
      ARR(i, k, l) = max(ARR(i, k, l), 0.d0)
      ARR(i, k, l) = min(ARR(i, k, l), 1.d0)
    end do
  end do
end do
	do l = 2, lmax
	  do k = 1, lmax*imax - imax
	    temp = ARR(k,1,l)
	    temp = max(temp, 0.d0)
	    ARR(k,1,l) = min(temp, 1.d0)
	  enddo
	end do

The compiler automatically collapses a loop if:

(1) The loops are too tightly nested, i.e. each DO loop and the DO loop nested immediately within it must look as shown in Example 1.

Example 1: loop nest is tightly nested	Example 2: loop nest is not tightly nested	Example 3: loop nest is not tightly nested

DO K=1,10
  DO J=1,20
    DO I=1,30
      A(I,J,K)=B(I,J,K)*C(I,J,K)
    ENDDO
  ENDDO
ENDDO
	DO K=1,10
	  D(K)=0.0
	  DO J=1,20
	    DO I=1,30
	      A(I,J,K)=B(I,J,K)*C(I,J,K)
	    ENDDO
	    X(K,J)=Y(K,J)+Z(K,J)
	  ENDDO
	ENDDO
		DO K=1,10
		  DO J=1,20
		    DO I=1,10
		      S(I,J,K)=T(I,J,K)*U(I,J,K)
		    ENDDO
		    DO I=1,30
		      A(I,J,K)=B(I,J,K)*C(I,J,K)
		    ENDDO
		  ENDDO
		ENDDO

(2) The loop bounds must be identical to the array bounds for the first N-1 dimensions, where N is the number of dimensions to be collapsed. When the iteration count of a loop depends on the iteration count of its outer loop, the loop is not collapsed.

Example: This loop nest is not collapsed.

DO K=1,20
  DO J=K,100
    DO I=1,100
      A(I,J,K)=B(I,J)
    ENDDO
  ENDDO
ENDDO

(3) In the subscription of the array reference, each index of loops is appeared with the same form and the same order.

Example: Data dependencies are unchanged after the loop collapsed.	
ORIGINAL	**LOOP COLLAPSED**

REAL,DIMENSION(10,30)::A,B,C
DO J=1,30
    DO I=1,10
        A(I,J)=B(I,J)+C(I,J)
    ENDDO
ENDDO
	REAL,DIMENSION(10,30)::A,B,C
	REAL,DIMENSION(300)::aa,bb,cc
	EQUIVALENCE (A,aa),(B,bb),(C,cc)
	DO ii=1,300
	  aa(ii)=bb(ii)+cc(ii)
	ENDDO

(4) All arrays that are indexed by the loops are neither a pointer nor an assumed shape array.

When a pointer or an assumed shape array is indexed by the loop, the compiler does not automatically collapse the loops, and can neither automatically collapse the loops containing an automatic array nor an allocatable array. If you specify the compiler directive COLLAPSE, then the compiler collapses the loops even if such arrays exist in the loops.

- Loop Interchange

Loop interchange is performed in order to remove data dependency or improve performance of vector instructions.

Loop Interchange is the process of interchanging the loop indexes of inner and outer loops in the case of nested loops. This is mostly used to improve cache behavior. In practice, the innermost loop should (only) index the right-most array index expression in case of row-major storage like in C.

Loop interchange can also expose parallelism. If an inner-loop does not carry a dependency (entry in direction vector equals '='), this loop can be executed in parallel. The granularity of the parallel loop can be increased by moving the inner loop outward.

ORIGINAL	INTERCHANGED
Example 1 (in C)	

for (i=0; i<N; i++)
 for (j=0; j<M; j++)   ! Unvectorized Loop
   B[i][j] = f(A[j],B[i][j-1]);
	for (j=0; j<M; j++)
	 for (i=0; i<N; i++) ! Vectorized Loop
	   B[i][j] = f(A[j],B[i][j-1]);

Example 2 (in C)	

for (i=0; i <n; i++) {
  for (j=0; j <n; j++) {
    a[j][i+1] = 2.0*a[j][i-1];
  }
	for (j=0; j<n; j++){
	  for (i=0; i<n; i++){
	    a[j][i+1] = 2.0*a[j][i-1];
	  }

}	}

ORIGINAL	INTERCHANGED
Example 1 (in FORTRAN)	
DO J=1,M **DO I=1,N** **A(I+1,J) = A(I,J) + B(I,J)** **ENDDO** **ENDDO**	**DO I=1,N** **DO J=1,M** **A(I+1,J) = A(I,J) + B(I,J)** **ENDDO** **ENDDO**
Example 2 (in FORTRAN)	
DO I=1,N **DO J=1,M** **A(I,J)=B(I,J)+C(I,J)** **ENDDO** **ENDDO**	**DO J=1,M** **DO I=1,N** **A(I,J)=B(I,J)+ C(I,J)** **ENDDO** **ENDDO**

The compiler automatically exchanges the outer loop with the inner if:

The loops are tightly nested.

Interchanging loops would enable the loop to be vectorized, increase the loop length, or shorten the stride of array references.

Data dependencies are unchanged after the loop interchange.

Example: Data dependency is removed	
ORIGINAL	**LOOP INTERCHANGED**
DO J=l,100 **DO I=1,50** **A(I+l,J)=A(I,J)*B(I,J)** **ENDDO** **ENDDO**	**DO I=1,50** **DO J=1,100** **A(I+l,J)=A(I,J)*B(I,J)** **ENDDO** **ENDDO**

Example: Longer loop length and shorter stride of array references.	
ORIGINAL	**LOOP INTERCHANGED**
DO I=1,100 **DO J=1,10** **A(I,J)=A(I,J)*B(I,J)** **ENDDO** **ENDDO**	**DO J=1,10** **DO I=1,100** **A(I,J)=A(I,J)*B(I,J)** **ENDDO** **ENDDO**

- Alternate code generation

When there is a choice on vectorized loop, the compiler generates two versions of the loop together with a run-time test to choose between them.

Vectorization threshold length run-time testing

When the compiler cannot determine a loop length, it generates two versions of the loop

together with a run-time test. If the loop length is greater than or equal to the vectorization threshold length, the vector version executes. Otherwise, the scalar version executes. Vectorization threshold length can be specified by compiler option -mvector-threshold.

Example: Longer loop length and shorter stride of array references.	
ORIGINAL	**LOOP TRANSFORMED**

DO I=1,N
    A(I)=B(I)+C(I)
ENDDO
	IF(N.LT.5) THEN
	    DO I=1,N
	        A(I)=B(I)+C(I) !Scalar
	    ENDDO
	ELSE
	    A(1:N)=B(1:N)+C(1:N) !Vector
	ENDIF

Data dependency run-time testing

When the data dependency is unclear because of variables in array subscripts, the compiler generates two versions of the loop together with run-time test. If the array has no data dependency, the vector version will execute; otherwise, the scalar version or the partial vectorized version, i.e., a part of the loop body which references the array is not vectorized, will execute.

Example:

Example: If K> 0 or K<-10, A(I) does not conflict with A(I+K).	
ORIGINAL	**LOOP TRANSFORMED**

DO I=N,N+10
    A(I)=A(I+K)+B(I)
ENDDO
	IF(K.GE.0 OR. K.LE.-11) THEN
	    A(N:N+10)=A(N+K:N+10+K)+B(N:N+10) !
	Vector
	ELSE
	    DO I=N,N+10
	      A(I)=A(I+K)+B(I) ! Scalar
	    ENDDO
	ENDIF

Short reduction loop run-time testing

When a loop length is less than or equal to the maximum vector register length (short loop), the compiler is able to generate a more simple and efficient vector code for a reduction macro operation such as sum, inner product, product or maximum/minimum value.

If the iteration count of the loop including those macro operations is unknown, the compiler generates two versions of the loop together with a run-time test. If the loop length is less than or equal to the maximum vector length, the vector code optimized for a short loop executes; otherwise, the normal vector code executes.

Example	
ORIGINAL	**LOOP TRANSFORMED**

DO I=1,N
S = S + X(I)
ENDDO
	! Is N less or equal to maximum vector length?
	IF(N.LE.MaxVL) THEN
	!CDIR SHORTLOOP
	DO I=1,N
	! Optimized vector code for short loop
	S = S + X(I)
	ENDDO
	ELSE
	DO I=1,N
	S = S + X(I) ! Normal vector code
	ENDDO
	ENDIF

When a loop or an array expression is vectorized with the extended vectorization function, an alternate code generation function does not apply the loop or the array expression even if the -mvector-dependency-test option or directive is specified.

- Unrolling outer loops

The compiler automatically unrolls outer loops, if outer loop unrolling improves the opportunities for overlapping vector instructions or reduces the number of loads and stores in the inner loops. Loop Unrolling (also referred to as Loop Unwinding) is a method of optimizing time-critical/ performance-critical loops. It is achieved by reducing its overhead through reduction of the number of iterations in that loop. This iteration reduction is performed by replicating the functionality within the same loop.

ORIGINAL	UNROLLED
for (i = 0; i < 100; i++) **{** **func();** **}**	**for (i = 0; i < 100; i += 2)** **{** **func();** **func();** **}**

Loop unrolling is effective when you can break any dependency chains within the loop.

ORIGINAL	**UNROLLED**

for (int i=0; i<n; i++)
{
sum += data[i];
}
	for (int i=0; i<n; i+=4)
	{
	sum1 += data[i+0];
	sum2 += data[i+1];
	sum3 += data[i+2];
	sum4 += data[i+3];
	}
	sum = sum1 + sum2 + sum3 + sum4;

In principle, the target is to improve the speed of the program by elimination/reduction of instructions that control the loop. Through loop-unrolling, below benefits can be achieved:

Reduction in branching

Hiding read/write latencies

Some illustrated examples:

ORIGINAL	UNROLLED
Example 1 (in C)	

**for (i=0; i<50; i++)
{
   a[i] = b[i];
}**
	**for (i =0; i<50; i+=2)
	{
	   a[i] = b[i];
	   a[i+1] = b[i+1];
	}**

Example 2 (in C)	

**int countbit(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}**
	**int countbit(unsigned int n)
	{
	    int bits = 0;
	    while (n != 0)
	    {
	        if (n & 1) bits++;
	        if (n & 2) bits++;
	        if (n & 4) bits++;
	        if (n & 8) bits++;
	        n >>= 4;
	    }
	    return bits;
	}**

ORIGINAL	UNROLLED
Example 1 (in FORTRAN)	

**DO I=1, N
        A(I)=B(I)
END DO**
	**DO I=1, N-1, 2
	   A(I)=B(I)
	   A(I+1)=B(I+1)
	END DO**

Example 2 (in FORTRAN)	

**DO I = 1,   IMAX
    DO J = 1, JMAX
        S(J)=S(J)+A(I,J)*B(I,J)
    END DO
END DO**
	**if (IMAX .gt. 0)then
	  TEMP = and(IMAX,3)
	  DO I = 1, TEMP
	    DO J = 1, JMAX
	      S(J)=S(J)+A(I,J)*B(I,J)
	    END DO
	  END DO**

```
DO I = TEMP+1, IMAX, 4
  DO J = 1, JMAX
    S(J)=S(J)+A(I,J)*B(I,J) &
    & +A(I+1,J)*B(I+1,J) &
    & +A(I+2,J)*B(I+2,J) &
    & +A(I+3,J)*B(I+3,J)
  END DO
END DO
endif
```

Example: The number of loads and stores of S(J) are reduced.	
ORIGINAL	**LOOP UNROLLED**

DO I=1,10
    DO J=l,N
        S(J)=S(J)+A(I,J)*B(I,J)
    ENDDO
ENDDO
	DO I=1,10,2
	    DO J=l,N
	        S(J)=S(J)+A(I,J)*
	            & B(I,J)+A(I+l,J)*B(I+l,J)
	    ENDDO
	ENDDO

- Loop rerolling

  Unrolling a vectorizable loop may lower efficiency because it reduces loop length or converting continuous vectors to non-continuous vectors.

  The compiler recognizes unrolled loops and rerolls them.

Example:	
ORIGINAL	**LOOP REROLLED**

DO I=1,100,2
    A(I)=B(I)+C(I)
    A(I+l)=B(I+l)+C(I+l)
ENDDO
	DO I=l, 100
	    A(I)=B(I)+C(I)
	ENDDO

- Outer loop strip-mining

When an iteration count of a loop is greater than the maximum vector register length, the compiler puts a loop around the vector loop which splits the total vector operation into "strips" so that the vector length will not be exceeded.

This is the method of converting a single loop into two nested loops for a specified “block” size.

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as providing a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

ORIGINAL	STRIP-MINED
Example 1 (in C)	
i = 1 **do while (i<=n)** **a(i) = b(i) + c(i)** **i = i + 1** **end do**	**// when n is a multiple of 4** **i = 1** **do while (i < (n - mod(n,4)))** **a(i:i+3) = b(i:i+3) + c(i:i+3)** **i = i + 4** **end do**
Example 2 (in C)	
do i=1,N **A[i] = x + B[i] * 2** **enddo**	**do ii=1,N,B** **do i=ii, min(ii+B-1, N), 1** **A[i] = x + B[i] * 2** **enddo** **enddo**

Example:	
ORIGINAL	**LOOP STRIP-MINED**
DO I=1,1000 **A(I)=B(I)+C(I)** **ENDDO**	**DO i=1,1000,maxvl** **l=MIN(1000-i,maxvl-1)** **A(i:i+l)=B(i:i+l)+C(i:i+l)** **ENDDO** **maxvl: maximum vector register length**

For a loop nest that has invariable array references on the outer loop inside the inner loop, the inner loop is split into a strip loop and the strip loop is moved outside of the outer loop so that invariants can be kept in the vector register.

Example:		
ORIGINAL	**OUTER LOOP STRIP-MINED**	**Load and store of S(j:j+l) are moved to outside.**
DO I=1,10 **DO J=1,1000** **S(J)=S(J)+X(J,I)*Y(J,I)** **ENDDO** **ENDDO**	**DO j=1,1000,maxvl** **l=MIN(1000-j,maxvl-1)** **DO I=1,10** **S(j:j+l)=S(j:j+l)+X(j:j+l,I)*Y(j:j+l,I)** **ENDDO** **ENDDO**	**DO j=1,1000,maxvl** **l=MIN(1000-j,maxvl-1)** **vr(1:l)=S(j:j+l)** **DO I=1,10** **vr(1:l)=vr(1:l)+X(j:j+l,I)*Y(j:j+l,I)** **ENDDO** **S(j:j+l)=vr(1:l)** **ENDDO**

maxvl: maximum vector register length

vr: vector register

- Recognizing matrix multiply loop

The compiler recognizes matrix-matrix or matrix-vector multiplication loops, and replaces them to a turned internal library call.

Matrix-Vector Multiplication

Example:

```
do j = 0,N2-1
  do i = 0,N1-1
    C(i*NC+1) = C(i*NC+1) + B(j*NB+1) * A(i+1,j+1)
  enddo
enddo
```

NB and NC should be integer constants.

Example:

```
do j = 0,N2-1
  do i = 0,N1-1
    C(i*NC+1) = C(i*NC+1) - B(j*NB+1) * A(i+1,j+1)
  enddo
enddo
```

NB and NC should be integer constants.

Matrix-Matrix Multiplication

Example 1:

```
do k = 1, N3
   do j = 1, N2
     do i = 1, N1
       C(i, j) = C(i, j) + B(k, j) * A(i, k)
     enddo
   enddo
 enddo
```

Example 2:

```
do k = 1, N3
  do j = 1, N2
    do i = 1, N1
      C(i, j) = C(i, j) - B(k, j) * A(i, k)
    enddo
  enddo
enddo
```

Example 3:

```
do j = 1, N2
   do i = 1, N1
      C(i, j)=0
   enddo
 enddo
 do k = 1, N3
   do j = 1, N2
     do i = 1, N1
       C(i,j) = C(i,j) + B(k,j) * A(i,k)
     enddo
   enddo
 enddo
```

```
do i = 1, N1
  do j = 1, N2
    C(i, j)=0
    do k = 1, N3
      C(i,j) = C(i,j) + B(k,j) * A(i,k)
    enddo
  enddo
enddo
```

Example 4:

```
do j = 1, N2
  do i = 1, N1
     C(i, j)=0
  enddo
enddo
do k = 1, N3
  do j = 1, N2
    do i = 1, N1
      C(i,j) = C(i,j) - B(k,j) * A(i,k)
    enddo
  enddo
enddo
```

```
do i = 1, N1
  do j = 1, N2
    C(i, j)=0
    do k = 1, N3
      C(i,j) = C(i,j) - B(k,j) * A(i,k)
    enddo
  enddo
enddo
```

- Loop expansion

The compiler expands a loop if all of the following conditions exist.

The loop is an innermost loop.

The loop does not contain IF statement.

The detailed option -floop-unroll is effective.

The loop length of the loop can be determined at compile time.

Example

ORIGINAL	LOOP EXPANDED
DO J=1,3 X(I)=Y(I) END DO	X(1)=Y(1) X(2)=Y(2) X(3)=Y(3)

Loop expanding is done before vectorization, therefore when all the loops in an outer loop is expanded, the outer loop is vectorized as an innermost loop.

Example	
ORIGINAL	**LOOP EXPANDED**
DO I=1,N	**DO I=1,N**
DO J=1,3	**X(I,1)=X(I,4)**
X(I,J) = X(I,4)	**X(I,2)=X(I,4)**
END DO	**X(I,3)=X(I,4)**
END DO	**ENDDO**

Loops that have a small iteration count and contain only a few lines of code may be expanded into the equivalent statements, so that the loop no longer exists. This may enable other optimizations.

Loop and array assignment expanding take precedence over loop collapsing and the COLLAPSE directive and the user should specify the detailed option -noexpand to make collapsing occur in these cases.

- Loop fusion

As the name suggests, it is the mechanism of fusing two adjacent loops of similar lengths/functionality into one loop. This is a very effective method of reducing the loop overhead and improving run-time performance of the program.

Although loop fusion reduces loop overhead, it does not always improve run-time performance, and may in some cases, reduce run-time performance. For example, the memory architecture may provide better performance if two arrays are initialized in separate loops, rather than initializing both arrays simultaneously in one loop.

<table>
<tr><th>ORIGINAL</th><th>FUSED</th></tr>
<tr><td colspan="2">Example 1 (in C)</td></tr>
<tr><td>

```
/* L1: short parallel loop */
for (i=0; i < 100; i++)
{
	a[i] = a[i] + b[i];
}
/* L2: another short parallel loop */
for (i=0; i < 100; i++)
{
	b[i] = a[i] * d[i];
}
```

</td><td>

```
/* L3: a larger parallel loop */
for (i=0; i < 100; i++)
{
  a[i] = a[i] + b[i];
b[i] = a[i] * d[i];
}
```

</td></tr>
</table>

ORIGINAL	FUSED

Example 1 (in FORTRAN)

ORIGINAL

```
psx(:)=0.0D0
do i=1,imax
        psx(i)=ps(i)*lnpsx(i)
end do

psy(:)=0.0D0
do i=1,imax
        psy(i)=ps(i)*lnpsy(i)
end do
```

FUSED

```
psx(:)=0.0D0
psy(:)=0.0D0
do i=1,imax
        psx(i)=ps(i)*lnpsx(i)
        psy(i)=ps(i)*lnpsy(i)
end do
```

Example2 (in FORTRAN)

ORIGINAL

```
do i=1, imax
  do k=1, NUM
    if ( cvr(i,k) > 1.D-30 ) then
      cbase(i) = p(i,k)
      exit
    end if
  end do

  do k=NUM, 1, -1
    if ( cvr(i,k) > 1.D-30 ) then
      ctop(i) = p(i,k)
      exit
    end if
  end do
end do
```

FUSED

```
do i=1, imax
  FBase=0
  FTop=0
  do k=1, NUM
    if (( cvr(i,k) > 1.D-30 ) .and. FBase.eq.0) then
      FBase = k
    end if
    if (( cvr(i,((NUM + 1) - k)) > 1.D-30 ) .and.
FTop.eq.0) then
      FTop = k
    end if
  end do
  if (FBase.ne.0) then
    cbase(i) = p(i,FBase)
  end if
  if (FTop.ne.0) then
    ctop(i) = p(i,((NUM + 1) - FTop))
  end if
end do
```

Example 3 (in FORTRAN)

ORIGINAL

```
l = kmax - 1

DO i = 1, imax
  WORK(i, l) =DX(i, l) * DY(i, l)
  WSUM(i, l) = WORK(i, l)
  ARR(i, l, l) = 1.0d0
  ARR(i, l+1, l) = ARR(i,l,l) - WORK(i,l)
  ARR(i, l, l + 1) = ARR(i, l + 1, l)
  ARR(i,l+2,l)= ARR(i,l+1,l) - WORK(i,l+ 1)
  ARR(i, l, l + 2) = ARR(i, l + 2, l)
END DO
```

FUSED

```
l = kmax - 1

DO i = 1, imax
  WORK(i, l) = DX(i, l) * DY(i, l)
  WSUM(i, l) = WORK(i, l)
  ARR(i, l, l) = 1.0d0
  ARR(i, l+1,l) = ARR(i, l, l) - WORK(i, l)
  ARR(i, l, l + 1) = ARR(i, l + 1, l)
  ARR(i,l+2,l)= ARR(i,l+1,l) - WORK(i,l+1)
  ARR(i, l, l + 2) = ARR(i, l + 2, l)

  !REPLACED ALL 'l' WITH 'l+1'
```

```
l = kmax
do i = 1, imax
  WORK(i, l) = DX(i, l) * DY(i, l)
  WSUM(i, l) = WORK(i, l)
  ARR(i, l, l) = 1.0d0
  ARR(i, l+1,l) = ARR(i, l, l) - WORK(i, l)
  ARR(i, l, l + 1) = ARR(i, l + 1, l)
end do
```

```
  WORK(i, l+1) = DX(i, l+1) * DY(i, l+1)
  WSUM(i, l+1) = WORK(i, l+1)
  ARR(i, l+1, l+1) = 1.0d0
  ARR(i,l+2,l+1)=ctau(i,l+1,l+1)-WORK(i, l+1)
  ARR(i,l+1,l+2) = ARR(i,l+2, l+1)
end do
```

4.2.23 Effects on Arithmetic Results

Execution results may differ before and after vectorization for the following reasons.

The order of operation may differ before and after vectorization.

Example: The operation order of a summation operation containing eight elements.	
BEFORE VECTORIZATION	**AFTER VECTORIZATION**
s=s+a1	t1=a1+a5
s=s+a2	t2=a2+a6
:	t3=a3+a7
:	t4=a4+a8
s=s+a8	t5=t1+t3
	t6=t2+t4
	t7=t5+t6
	s=s+t7

To increase speed, the vector versions of intrinsic functions do not always use the same algorithms as the scalar versions.

Optimization techniques, such as conversion of division to multiplication, are applied differently.

Optimization techniques, such as reordering of arithmetic operations, are applied differently.

Integer iteration macro operation is vectorized by using a floating-point instruction. So when the result exceeds 52 bits or when a floating overflow occurs, the result differs from that of scalar execution.

4.2.24 Detection of Vectorization-Caused Errors and Exceptions

Detection of errors and arithmetic exceptions by intrinsic functions may differ before and after vectorization. A difference in the order of detection is shown as follows:

```
DO I=1,100
     X(I)=SQRT(A(I))
     Y(I)=ALOG(B(I))
END DO
```

On the assumption that A(2) and A(5) are negative and other elements of A are positive, and B(3) and B(4) are zero and other elements of B are positive, the order of error detection before vectorization is as follows:

Error for A(2)<0 in SQRT
Error for B(3)=0 in ALOG
Error for B(4)=0 in ALOG
Error for A(5)<0 in SQRT

The order of error detection after vectorization is as follows:

Error for A(2)<0 in SQRT
Error for A(5)<0 in SQRT
Error for B(3)=0 in ALOG
Error for B(4)=0 in ALOG

When a loop containing intrinsic functions is vectorized, and the vector version is referenced, no error check is made on the values of arguments. The NOVERRCHK compiler directive is enabled.

4.2.25 Boundary of Dummy Array

Data which is an operand of the vector operation must be aligned on a memory boundary corresponding to its type. On vectorization, the compiler checks whether each operand is aligned on a vectorizable memory boundary or not. The compiler assumes that the dummy array is aligned on a vectorizable memory boundary, since the alignment of the dummy array is unknown at compilation. Then, if an actual array corresponding to the dummy array is not aligned on a vectorizable memory boundary, an execution exception occurs. In this case, each operand of vectorized operations must be aligned on the correct memory boundary.

4.2.26 Array Declaration

When the compiler checks whether vectorization would preserve the proper dependency between array definitions and references, it assumes that all values of subscript expressions are within the upper and lower limits of the corresponding size in the array declaration. If a loop violating this condition is vectorized, correct results are not guaranteed.

When a loop containing IF statements is vectorized, arithmetic operations are carried out only for the part that conditionally requires them, but arrays are referenced as many times as the iteration count called for by the DO statement and array elements that should not be

referenced are referenced. Unless the arrays have enough area reserved to satisfy the iteration count, memory access exceptions can occur as a result.

Example

```
DIMENSION A(10)
 :
 :
DO I=1,50
   IF(I.LE.10)THEN
     X=A(I)
   ELSE
   END IF
 :
 :
ENDDO
```

When a loop containing a branch out of the loop is vectorized, arithmetic operations are carried out unconditionally for the part before the branch point, as many times as the iteration count called for by the DO statement. Therefore, arithmetic operations that should not be carried out are carried out, or data that should not be referenced are referenced. These events can cause errors or exceptions.

Example

```
DO I=1,30
      X=SQRT(A(I))
      IF(X.LE.0.01)EXIT
ENDDO
```

If a branch occurs when I=10 and A(20)<0, an error occurs that should not occur.

4.2.27 Association of Dummy Arguments

When the compiler checks definition-reference dependency, it assumes that dummy arguments with different names identify different elements. Therefore, if different dummy arguments are associated with the same actual argument, and either of them is defined, correct results are not guaranteed. If dummy arguments are associated with a common block element, the same problem occurs. Such programming violates the Fortran standard.

Example

```
DIMENSION A(100)
     :
CALL SUB   (A,A,100)
     :
END
```

```
SUBROUTINE SUB (A,B,N)
DIMENSION A(N),B(N)
DO I=1,N-1
      A(I+1)=X*B(I)
ENDDO
END
```

4.2.28 High-Speed I/O Techniques

There are two techniques for speeding up I/O operations.

- One involves program coding to reduce unnecessary I/O overhead
- The other involves file access to reduce the number of times I/O accesses external files.

This section explains the techniques for speeding up I/O operations in terms of these approaches.

The compiler provides the file I/O analysis information output function (F_FILEINF) as a support function for determining whether I/O operations are performed at a satisfactorily high speed. This function is also explained here.

- Programming Techniques
  - Unformatted I/O is recommended when possible. Format conversion is not done and resulting precision errors are avoided.
  - Avoid implied-DO lists in an I/O list, and use array names whenever possible. When array names are used, the transfer of data between the I/O buffer and the user area is done with one instruction. When an implied-DO list is used, I/O speed is reduced by additional loop overhead.

Example

```
DIMENSION A(M,N)
WRITE(1)((A(I,J), I=1, M), J=1, N)

The above is less efficient than
WRITE(1)A
```

When an assumed-size dummy array is used, or when an implied-DO list must be used because of programming considerations, the implied-DO list should be arranged so that the implied loop processing is done only once. The information of the implied-DO list may be collectively passed to the I/O routine. The overhead of I/O at execution time is reduced, thereby increasing processing speed.

When array element names are specified in an implied-DO list, the values of subscripts

should be written in the following format:

A([+a*] I [+b])

where:

I DO variable

a,b Unsigned variable or constant

[ ] Optional

No expression should be written in the implied-DO list for the WRITE statement.

Example	
ORIGINAL	**OPTIMIZED**
DIMENSION A(10) **WRITE(1)(A(I)+X, I=1, 10)**	**DIMENSION A(10), B(10)** **DO 10 I=1, 10** **10 B(I)=A(I)+X** **WRITE(1)(B(I), I=1, 10)**

The initial, terminal, and increment parameter should not be computed in the implied-DO list.

Example	
ORIGINAL	**OPTIMIZED**
WRITE(1)(A(I), I=1, J*K)	**JK=J*K** **WRITE(1)(A(I), I=1, JK)**

Since the implied-DO list is processed in batches, for an array of two or more dimensions the outer loop is expanded. To reduce the overhead in expanding a DO loop, reduce the number of calls to the runtime I/O routines as follows.

Make the DO loop such that the list of elements to be processed by the implied-DO list is contiguous in the storage area. The data transfer between the I/O buffer and the user area in the runtime I/O routine can be done with one instruction.

By exchanging the subscripts of the inner DO loop and that of the outer DO loop, the DO loop becomes contiguous.

Example	
ORIGINAL	**OPTIMIZED**
DIMENSION A(M,N) **WRITE(1)((A(I,J), J=1, N), I=1,M)**	**DIMENSION A(M,N)** **WRITE(1)((A(I,J), I=1, M), J=1, N)**

Specify the variable of the innermost DO loop as one dimensional and avoid increment parameter other than 1.

Make the DO loop so that only one I/O list may exist in one implied-DO list. If more

than one I/O list is desired, use two or more implied-DO lists.

Example	
ORIGINAL	**OPTIMIZED**
WRITE(1)(A(I), B(I), I=1, N)	**WRITE(1)(A(I), I=1, N), (B(I), I=1, N)**

- Asynchronous I/O Functions

Asynchronous I/O functions are provided to increase speed by conducting I/O processing and arithmetic processing simultaneously.

The asynchronous I/O function consists of an asynchronous READ statement and an asynchronous WRITE statement that initiate data transfer between main memory and secondary memory (such as a magnetic disk) and a WAIT statement. Application programs can be executed more rapidly because the data transfer is processed in parallel with the executable statements that follow the asynchronous READ or asynchronous WRITE statement.

BUFFER IN/BUFFER OUT statements and UNIT/LENGTH functions are added for the same purpose.

- Techniques for Effective File Access

To perform high-speed file access, the user should:

(1) Reduce the number of data transfers between the I/O buffer used by the compiler and the user area.

(2) Improve efficiency in buffering (processing to store data in a buffer) using the I/O buffer.

(3) Reduce the number of I/O operations on external files.

(4) Use optimum types of records.

The second and subsequent items are closely related to specifications of various runtime options. This section describes these relationships.

The user can perform faster file access by satisfying the conditions of these items.

- I/O Buffer

This section describes the I/O buffer used by the compiler.

During execution of I/O statements, data is generally transferred between the user data area and the system area via the I/O buffer. I/O routines perform various processing to effectively use the I/O buffer. I/O routine processing affects file access.

The size of the I/O buffer can be altered by specifying the runtime option VE_FORT_SETBUF. The default depends on file organization and other factors as follows:

(1) Sequential file (all file system types): 512 KB

(2) Direct file (when the value of the RECL specifier in the OPEN statement is 4096 bytes or less): 4 KB

(3) Direct file (when the value of the RECL specifier in the OPEN statement is 2,048,000,000 bytes or more): 2,000,000 KB

(4) Direct file (when the value of the RECL specifier in other than the above): Raise fractions of record length to unit (KB).

- Management for I/O Buffer

This section describes the I/O buffer used by the compiler.

(1) Sequential access I/O

I/O routines examine the length of each element of an I/O list. When the list is small (I/O-buffer-size*2), data is buffered. When the list is large, data is transferred directly between the user area and the system area without using the I/O buffer (bufferless I/O). If I/O mode is output, the contents of the I/O buffer are output before direct I/O. If I/O mode is input, the contents of the I/O buffer are input before direct I/O.
The above processing can speed up I/O operations because it effectively uses the I/O buffer only for I/O statements that process small amounts of data.
The contents of the I/O buffer are output in the following cases:

- When the I/O buffer becomes full.
- When the CLOSE statement is executed.
- When an I/O statement which causes file positioning beyond the range of the I/O buffer is executed.
- When the REWIND statement is executed after the WRITE statement.

(2) Direct access I/O

This processing is basically the same as for unformatted sequential access I/O. Unlike sequential access, however, the contents of the I/O buffer are output (switched) in the following cases:

- When the system detects a nonconsecutive record number (for example, when

record #3 is the next record processed after record #1) in the I/O statement.

- When the I/O mode is changed from output to input or vice versa.

Record numbers are normally nonconsecutive in a direct access file. In this case, the contents of the I/O buffer are output (switched) frequently. Note that this type of processing gains little benefit from effective use of the I/O buffer.

- Efficient Techniques

  After it has been decided to perform processing using the I/O buffer, examine the following measures for performing high-speed unformatted file access. An explanation is given for each type (sequential/direct access) of file that is transferred.

(1) Sequential file

Run-time option VE_FORT_SETBUF is effective in the following cases:

- When a large number of I/O statements are used for a small amount of data transfer.
- When the total size of the file is known. (In this case, when the size is specified using this runtime option, data is not transferred to or from the file and all I/O operations are performed using the I/O buffer.)

(2) Direct file

Run-time option VE_FORT_SETBUF is effective in the following cases:

When the record numbers to be processed by an I/O statement are contiguous (record #1, record #2, record #3, etc.). Record numbers in a direct file are normally nonconsecutive. When a value greater than the default is specified for runtime option VE_FORT_SETBUF in cases other than the above, the size of data transferred at one time becomes larger. This may degrade performance.

Chapter5　Conclusion

The tuning methodology specified in this document requires a skillful perspective of code analysis to understand which part of the source code is high cost and how performance must be extracted from it.

This procedure document must be used as the guide for understanding basics of tuning applications for the NEC SX-Aurora TSUBASA.

Fine-tuning an application is a skill that may not necessarily be achieved through a specific tuning process. Performance tuning is an act of experiment. It is necessary to understand that there is no measure of highest performance. The application must be constantly scrutinized iteratively to look for that one detail that may result in a higher performance.

The tuning personnel may encounter various examples of un-optimized:

- line-of-code
- code structure
- code flow structure; OR
- algorithm

A process can not completely define what to tune in what situation. It is a deep study of the algorithm and code structure that can help the tuning personnel to extract best performance from the code.