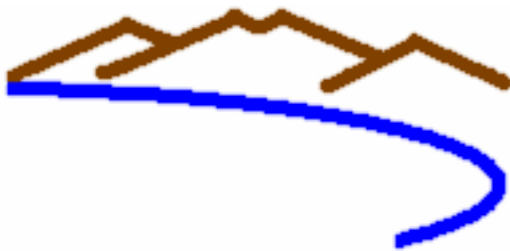


Crescent Bay Software

VAST-F/Altivec

Automatic Fortran Vectorizer for PowerPC Vector Unit



May 2005

Version 1.5

Preface

Document Number V01AVF: VAST-F/AltiVec User's Guide

This is a guide to the use of VAST-F/AltiVec. VAST-F/AltiVec is an automatic vectorizer for Fortran programs on Motorola AltiVec processors. This manual is designed to give programmers an understanding of VAST-F/AltiVec's capabilities and effective use.

Revision Record

Edition	Date	Description
1.0	5/2001	First release of document.
1.1	6/2001	Enhanced description of driver.
1.2	7/2001	Changed driver output object file convention.
1.3	1/2004	Minor revisions and updates.
1.4	2/2004	Minor revisions and updates.
1.5	5/2005	Minor corrections.

Copyright (C) 2001,2005, Crescent Bay Software. No unauthorized use or duplication is permitted. All rights reserved.

VAST and DEEP are registered trademarks of Crescent Bay Software Corp.

AltiVec is a trademark of Motorola, Inc.

Crescent Bay Software Corp., 10950 Washington Blvd., Suite 230, Culver City 90232 USA. Fax: (310)-836-7313 Phone: (310)-836-5183.

Web: <http://www.crescentbaysoftware.com>.

Table of Contents

<i>Preface</i>	<i>i</i>
1. Introduction	1
Overview	1
VAST-F Operation	1
Performance Tips	2
Understanding vectorization progress	2
Double vs. single precision.....	2
Loops and arrays.....	2
Data dependencies.....	3
Inline expansion	3
Optimizations	3
How to use this guide	3
DEEP development environment	4
2. Invoking VAST-F/AltiVec	5
v77 Driver	5
Usage	5
File Extensions.....	6
v77 Options.....	6
Example.....	7
Driver configuration file	7
Specific Layout	7
vastfav Command Line	9
VAST-F/AltiVec switches	10

VAST-F/Altivec options.....	12
3. User directives.....	14
Overview.....	14
VAST directive format.....	14
VAST directive summary.....	14
Transformation directives.....	16
NOVECTOR/VECTOR.....	16
SKIP.....	16
NOASSOC/ASSOC.....	16
Data dependency directives.....	16
NODEPCHK/DEPCHK.....	16
NOEQVCHK/EQVCHK.....	17
PERMUTATION.....	17
RELATION.....	17
Listing directives.....	17
NOLIST/LIST.....	17
SWITCH directive.....	17
Specifying directives on invocation.....	18
4. VAST listing.....	19
Overview.....	19
Input source listing.....	19
Diagnostic messages.....	20
Optimization inhibition messages.....	20
Informative messages.....	20
Error messages.....	20
Output source listing.....	20
Summaries.....	21
Listing control switches.....	21
Listing page length.....	22
5. Altivec code generation.....	23
Overview.....	23
Altivec code generation overview.....	23
Vector code generation sections.....	24
Vector code generation example.....	24
Alignment.....	26
Iteration count cleanup.....	27
Array reductions.....	27
Vector strip unrolling.....	27
6. Loop Optimizations.....	28

Overview	28
Loop fusion	28
Loop rerolling	29
Loop unrolling	29
Store postponement.....	31
Reduction unrolling	31
IF loops to DO loops	32
DO WHILE to DO	33
Loop interchange	33
Outer loop unrolling	33
Loop collapse	34
7. Data dependency analysis	36
Overview	36
Data dependency examples	36
Ambiguous subscript resolution	38
Data dependency directives	38
NODEPCHK - no vector feedback.....	38
NOEQVCHK -- non-recursion in equivalences.....	39
RELATION -- specifying relationship between variables.....	39
PERMUTATION -- declaring safe indirect addressing.....	40
Scalar dependencies	41
Carry-around scalars.....	41
Equivalenced scalars	41
8. Inline expansion	42
Overview	42
Introduction	42
Automatic inline expansion	43
Automatic inlining criteria	43
Avoiding inlining.....	43
Automatic inlining level	43
Explicit inline expansion	44
Source code access	44
Inline expansion directives	44
AUTOEXPAND directive	44
Explicit mode.....	44
EXPAND directive.....	45
NEXPAND directive.....	45
SEARCH directive	45
Where to get the source code	46
Same file	46
Explicitly named file.....	46
Implicitly named file	46

Possible problems	46
Separate compilation.....	46
Code size.....	46
Debugging.....	47
Compilation rate.....	47
Nested expansion	47
Analysis inhibitors	47
Expansion inhibitors.....	47
Inhibitors specific to automatic expansion mode.....	48
Inline expansion abilities	48
Naming conventions.....	48
Common blocks.....	48
Arguments.....	49
Unique names.....	49
Labels.....	49
Returns and return values.....	49
Inlining ENTRYs.....	49
Cleanup of inlined code.....	50
Example.....	50
Expansion of SAVE and DATA.....	51
Inline expansion user messages	52
Inline expansion summary.....	52
User messages.....	52
9. Diagnostic Messages	54
Overview	54
Data dependency conflicts	54
Translation Diagnostics	56
Statement types.....	56
Branches.....	57
External references.....	57
DO statement.....	58
Miscellaneous.....	58
Warnings	59
Potential Errors.....	59
Obsolescent Features.....	59
Syntax errors	59
Internal Errors	59
Directive Errors	60
Notes	60
10. Further Information	61
Crescent Bay Software	61
VAST	61
DEEP	62
Questions or Comments	62

Index..... 63

1. Introduction

Overview

This document describes the features and options of VAST-F/AltiVec, a software tool that vectorizes Fortran programs for execution on AltiVec systems.

VAST-F/AltiVec automatically transforms loops into vector instructions that use the AltiVec unit, which can result in substantially faster code.

Automatic vectorization of existing Fortran programs is a very useful tool, but additional changes by the programmer to the input program can sometimes be helpful in exposing more vectorization opportunities and in getting full performance from the system. VAST-F/AltiVec can generate varying degrees of vectorization in many programs, depending on the algorithms used, coding style, size and layout of arrays, data types, and other factors. To get the most out of VAST, you should familiarize yourself with the switches, options, pragmas, and features it offers.

This version of VAST-F accepts Fortran 77, and Fortran 90 if a Fortran 90 compiler is available (in particular array syntax is vectorized when possible).

VAST-F Operation

VAST is normally invoked via the driver program `v77`. (The driver is named according to the Fortran compiler that will be invoked - `v77` for Absoft `f77` and for `g77`, `v90` for Absoft `f90`, `vlf` for IBM `xlf`, etc. These are all essentially identical in usage; for brevity we'll use `v77` in the examples). Externally, `v77` functions exactly like the compiler. Internally, these are the steps `v77` performs:

1. Applies VAST-F/AltiVec to the input `.f` files. This creates temporary vectorized `V*.f` and `A*.c` files that contain AltiVec vector function references.
2. Compiles the `V*.f` files with a standard Fortran compiler and the `A*.c` files with an AltiVec-aware C compiler, creating vectorized object files (`*.o`). (For any one source file, the separate objects resulting from Fortran and generated C are recombined into a single object.)

3. If requested, invokes the Fortran compiler to link the *.o files to create a vectorized executable (default is a .out).

Performance Tips

VAST-F/AltiVec is intended for use primarily as an automatic tool; at a minimum, you need to know only how to invoke it (see Section 2). However, because of the complexity of the transformations involved and the unavailability of some important data at compile time, the added optimizations VAST-F/AltiVec performs may not significantly decrease the input program's execution time.

Understanding vectorization progress

By default, VAST/F displays the number of loops examined and vectorized for each routine it examines. This gives you a quick idea of the level of vectorization for your program.

To see more detailed vectorization messages, use the **-Vmessages** switch. These messages are not shown by default (as the output can be voluminous) but the messages can be very useful in helping you understand what the vectorizer has discovered.

You can also enable VAST-F/AltiVec's listing and look at the diagnostic messages in this special file. Referring to the relevant sections in this document, you may be able to improve the optimization by switching on or off certain transformations or default assumptions, inserting directives, or making minor code modifications. In some cases, you may be rewarded with dramatically improved performance for a relatively small effort.

Double vs. single precision

Integer and single precision (32 bit) operations can be optimized, but double precision (64 bit) is not supported by the AltiVec hardware. The **-Vforcesingle** switch is provided to make it easy to compile double precision floating point programs as single precision floating point. You can try this switch and see if the answers you get still have acceptable precision.

Loops and arrays

To vectorize efficiently, the program should be operating on arrays of data in reasonably straightforward loops. Programs which chase pointers in complicated data structures at the lowest level are generally difficult to vectorize. Most importantly, only array uses where consecutive array elements are accessed (unit stride) can make efficient use of the AltiVec unit. This means, for example, that you want to have the leftmost subscript vectorized on multiple-dimension arrays. Also, you want to organize your data as structure of arrays rather than an array of structures where possible.

Data dependencies

If you have simple loops traversing arrays with the acceptable data types and these loops are still not being vectorized, then you are probably encountering data dependency problems, where the vectorizer cannot vectorize because there may be overlap among the arrays. Usually, there is no actual overlap, and there are many ways you can inform the vectorizer that your loops are safe to vectorize. The `-vnodepchk` switch (turn off dependency check for potential dependencies) can be useful for programs that have many potential dependencies.

Inline expansion

Most VAST-F/AltiVec features are turned on by default, but some are not. To enable automatic inline expansion of calls to small functions, use `-Wv, -J1` on the driver invocation. (See the chapter on Inlining for further details.)

Optimizations

VAST-F/AltiVec's optimizations include:

- Full analysis of large loop nests, with vectorization at the appropriate level.
- Vectorization of non-aligned arrays.
- Restructuring of loop nests to allow enhanced vector execution.
- Vectorization of loops containing reduction operations (such as global sum functions).

How to use this guide

Section 2 describes how to invoke VAST-F/AltiVec. If you want to use VAST-F/AltiVec as a strictly automatic tool, you can skip the remainder of the guide beyond section 2.

Sections 3 and 4 discuss communication with VAST-F/AltiVec -- VAST-F/AltiVec's listing and messages, and ways to guide VAST-F/AltiVec's action (user directives).

Section 5 has information specific to AltiVec vector code generation, with an example of generated code and a discussion of VAST-F features for AltiVec, and coming enhancements.

Concepts and rules of VAST-F/AltiVec's optimization techniques are discussed in the remaining sections. Examples in these sections illustrate optimizable and unoptimizable loops. Of special importance are the sections on data dependency analysis (Section 7) and subprogram inlining (Section 8).

DEEP development environment

DEEP is a development environment for performance programming that allows you to quickly and easily identify where the performance bottlenecks in your program are (down to the individual loop level) and see what loops have and not been vectorized, and why. Featuring a highly graphical interface, DEEP allows you to rapidly “drill-down” to find optimization opportunities in your code. VAST-F/AltiVec works closely with DEEP, generating information on vectorization and optimization and instrumenting the code to gather run-time performance data. See the DEEP User’s Guide for more information.

2. Invoking VAST-F/AltiVec

v77 Driver

Normally, VAST-F/AltiVec is invoked via the **v77** (alias **v90/v1f** etc.) driver. **v77** is used just like the existing compiler and will accept all the compiler options. The driver takes care of vectorizing the code in the VAST-F precompilation phase, and then compiles this vectorized code with the normal compilation system. For example,

```
v77 myprog.f
```

Will create an `a.out` that is vectorized and ready to run on an AltiVec system (or with an AltiVec simulator).

The driver is actually performing the following steps:

1. Invokes VAST-F/AltiVec on the input files. VAST-F/AltiVec creates two output files: the vectorized loops file (now in C), and the remaining program file (in Fortran). The vectorized loops are extracted into C functions to use the C AltiVec intrinsic functions.
2. Invokes the AltiVec-enhanced C compiler on the vectorized loops file.
3. Invokes the Fortran compiler on the remaining program file.
4. For any one source file, combines the separate Fortran and C objects into a single object.
5. Invokes the linker to produce an executable binary.

Usage

```
v77 [<v77_option>]... [<f77_option>]...  
      [-Wv,<v_option>[,<v_option>]...] [files]
```

where:

`<v77_option>` represents any `v77` driver option (see “v77 Options” below).

<f77_option> represents any Fortran compiler option. All normal compiler options can be used.

<v_option> represents any VAST option.

v77 options and input files may appear in any order.

File Extensions

1. filename with a .f suffix: Fortran source file.
2. filename with a .c suffix: C source file.
3. filename with a .i suffix: preprocessed C source file.
4. filename with a .s suffix: assembler source file.
5. filename with a .o suffix: object file.
6. filename with a .a suffix: archive file.

v77 Options

- c** Suppress the link editing phase of the compilation and do not remove any object files produced. A single object file is produced for each input source file.
- dryrun** Display but do not execute compiler internal commands. v77 will still check for the existence of essential executable files.
- keep** Keep intermediate `vastfav` files. These files are Fortran and C source files. For an input file `file1.f` the intermediate files will be named `vfile1.f` and (if any vectorization was done) `Afile1.c`.
- l<suffix>** Search the library file `lib<suffix>.a`.
- L<dir>** Search directory <dir> for libraries prior to searching the default directories.
- o<name>** Name the final link-edited output file <name> (The default link-edited output file is named `a.out`.) `-o` is ignored if the `-vo` option is used. When used with `-c`, names the object file.
- v** Display v77 internal commands as execution progresses.
- vo** Execute `vastfav` pass only. Certain compiler options included on the command line may be ignored.
- vn** Execute Fortran compiler only. `vastfav` options included on the command line are ignored. Note that the `-vo` and `-vn` options are mutually exclusive.
- w** Suppress warning messages.
- Wv** Hand off arguments to `vastfav`. `vastfav` options must be separated by commas and may not contain embedded blank space characters.

Examples:

-Wv, -Valigned

-Yvf,<path> substitute for f77 an alternate executable whose pathname is specified by <path>.

-Yvv,<path> substitute for vastfav an alternate executable whose pathname is specified by <path>.

V77 invoked without options will generate a brief explanation of usage.

Example

```
v77 -O3 -o file1.vector.exe -Wv,-Vmessages file1.f
```

Executes vastfav and asks for vectorization messages to be displayed (-Vmessages). Compiles (with compiler optimization -O3) the intermediate files Vfile.f and Afile.c using the Fortran compiler and the Altivec-enhanced C compiler, and invokes the linker to produce the executable output file file1.vector.exe.

Driver configuration file

The general layout of the file is that of Windows-style initialization (*.ini) files, also known as the Section, Key - Value style. In general, the file looks like

```
[SectionName]
# <- begins a comment which runs to the end of the line
! <- also begins a comment which runs to the end of the line
KeyName1 = values are strings that do not need quotes
KeyName2 = "But quotes (both single or double) may be used
if the string falls across two or more lines"
KeyName3 = 'Or if the value contains a # or ! character'
KeyName4 = "The Parser" handles double quotes inside single quotes'
KeyName5 = "'and' vice versa"
KeyName6 = strings may also cross lines if the end of line \
is escaped
```

Section names and key names are not case sensitive, but values are passed on without a change in case.

Specific Layout

The driver translates a section-key-value combination in the configuration file into a command line option. Since these options appear first on the command line they can be overridden by options that the user puts on command line. This section lists each section and key recognized and the option that is created when the key is seen in the configuration file. More details on the command line options are given elsewhere. In this section: *{path}* indicates either a partially or fully qualified filename, (for example v77 or /b/develop/vcc2_sun/v77); *{paths}* indicates a series of directories separated by either blanks, commas, or semicolons, (for example /b/develop/v77 /b/develop/vcc is equivalent to

/b/develop/v77,/b/develop/vcc OR /b/develop/v77:/b/develop/vcc), and *{list}* is a series of options again separated by either blanks, commas, or semicolons.

Keys may appear outside of sections to set certain parameters if only one language is supported or if a default language is defined.

```

compiler      = {path}      # <=> -Yvc,{path}
vast          = {path}      # <=> -Yvv,{path}
preprocessor  = {path}      # <=> -Yvp,{path}
linker        = {path}      # <=> -Yvl,{path}
cleanup       = {path}      # <=> -YvCleanup,{path}
INCLUDE_DIRS = {paths}     # <=> --IncludeDirs,{paths}
LIB_DIRS      = {paths}     # <=> --LibraryDirs,{paths}
Defines       = {list}     # <=> --Defines,{list} for the default file type
Libraries     = {list}     # <=> --Libraries,{list} for the default file type
extensions    = {list}     # <=> --extensions[c],{list} or -
extensions[fortran],{list} # depending on default type

[fortran] # keys specific for fortran
vast       = {path}      # <=> -Yvfv,{path}
compiler   = {path}      # <=> -Yvfc,{path}
linker     = {path}      # <=> -Yvfl,{path}
cleanup    = {path}      # <=> -YvCleanup,{path} (same as c or c++)
defines    = {list}     # <=> --Defines[fortran],{list} for fortran files
libraries  = {list}     # <=> --Libraries[fortran],{list} for fortran files
extensions = {list}     # <=> --extensions[fortran],{list}

[c] # keys specific to c
preprocessor = {path}    # <=> -Yvp,{path}
vast        = {path}    # <=> -Yvv,{path}
compiler    = {path}    # <=> -Yvc,{path}
linker      = {path}    # <=> -Yvl,{path}
cleanup     = {path}    # <=> -YvCleanup,{path}
defines     = {list}    # <=> --Defines[c],{list} for c files
libraries   = {list}    # <=> --Libraries[c],{list} for c files
extensions  = {list}    # <=> --extensions[c],{list}

[c++] # keys specific for c++
# note that the first 4 are the same as the c keys.
preprocessor = {path}    # <=> -Yvp,{path}
vast        = {path}    # <=> -Yvv,{path}
compiler    = {path}    # <=> -Yvc,{path}
linker      = {path}    # <=> -Yvl,{path}
cleanup     = {path}    # <=> -YvCleanup,{path}
defines     = {list}    # <=> --Defines[c++],{list} for c++ files
libraries   = {list}    # <=> --Libraries[c++],{list} for c++ files
extensions  = {list}    # <=> --extensions[c++],{list}

[objects]
extensions  = {list}    # <=> --extensions[objects],{list}

[options] # this section gives command line options that are sent directly to
          # the given tool when files of the default file type are processed
vast      = {list}     # <=> -Wv,{list} for the default file type
preprocessor = {list}  # <=> -Wvp,{list} for the default file type
compiler   = {list}   # <=> -Wvc,{list} for the default file type

```

```

linker      = {list}      # <=> -Wvl,{list} for the default file type
cleanup    = {list}      # <=> -WvCleanup,{list} for all file types

[fortran-options] # this section gives options that are sent directly to
                  # the given tool when fortran files are processed
vast = {list}      # <=> -Wv[fortran],{list} for fortran files
preprocessor = {list} # <=> -Wvp[fortran],{list} for fortran files
compiler    = {list} # <=> -Wvc[fortran],{list} for fortran files
linker      = {list} # <=> -Wvl[fortran],{list} for fortran files
cleanup     = {list} # <=> -WvCleanup,{list} for all file types

[c-options] # this section gives command line options that are sent directly to
            # the given tool when c files are processed
vast = {list}      # <=> -Wv[c],{list} for C files
preprocessor = {list} # <=> -Wvp[c],{list} for c files
compiler    = {list} # <=> -Wvc[c],{list} for c files
linker      = {list} # <=> -Wvl[c],{list} for c files
cleanup     = {list} # <=> -WvCleanup,{list} for all file types

[c++-options] # this section gives command line options that are sent directly
              # to the given tool when c++ files are processed
vast = {list}      # <=> -Wv[c++],{list} for c++ files
preprocessor = {list} # <=> -Wvp[c++],{list} for c++ files
compiler    = {list} # <=> -Wvc[c++],{list} for c++ files
linker      = {list} # <=> -Wvl[c++],{list} for c++ files
cleanup     = {list} # <=> -WvCleanup,{list} for all file types

[directories] # this section defines directories the driver searches
includes     = {paths} # <=> --IncludeDirs,{paths}
libraries    = {paths} # <=> --LibraryDirs,{paths}
sources      = {paths} # <=> --SourcesPath,{paths}
executables  = {paths} # <=> --ExecutablesPath,{path}
vast         = {path}  # <=> --VastPath,{paths}

```

vastfav Command Line

You can run VAST-F/AltiVec outside of the v77/v90 driver if you wish. VAST-F/AltiVec is executed by the command:

```

vastfav [-o output] [-l listing] [options]
          input1 [...inputn]

```

output = compilable output file. The default output file names are the input file name prefixed by V and A (Fortran and C, respectively). The extension of the output files are .f and .c. [This feature is disabled in the initial release and the default is always used.] Note that any module information files produced for Fortran 90 module subprograms may be placed in alternative directories via the -L option, below.

listing = VAST-F/AltiVec listing file. A null parameter indicates the listing should go to the terminal. Unless this parameter is used, no listing is produced.

options = VAST-F/Altivec options and switches. Switches (on/off toggles) are passed with lower case parameter names. Options (numbers or names) are passed with upper case parameter names. All switches and option names are a single letter. See following sections for more information on switches and options.

input1...inputn = Fortran source input files.

VAST-F invoked with no arguments will print a short usage summary. VAST-F invoked with the '-v' parameter will print out the version number.

Example 1:

To run the source file `crunch.f` through VAST-F:

```
vastfav crunch.f
```

The optimized output is sent to `Vcrunch.f` and `Acrunch.c`, and the listing to standard output.

Example 2:

To run `sub.f` through VAST-F, save the VAST-F listing in file `sub.lst`, and specify that arrays are aligned (`-Valigned`).

```
vastfav -Valigned -l sub.lst sub.f
```

VAST-F/Altivec switches

To use the VAST-F/Altivec switches, specify "-vswitch". No space is allowed between -V and the switch. You cannot use commas to separate switches. For example, these uses are ILLEGAL:

```
-V novector ILLEGAL, space between -V and switch.  
-Vleastrecent , cachestrip ILLEGAL, only one switch per -V.
```

This would be a legal use (running `vastf` by itself):

```
vastfav -Vinline -Vnpointeroverlap -Valigned myfile.c
```

Example with "v77" driver:

```
v77 -Wv, "-Valigned, -Vnpointeroverlap" -O3 myfile.c
```

These are the switches currently defined for Altivec:

-Valigned

Asserts that start of all arrays is aligned on 16-byte boundary. This can improve performance by up to a factor of two, if you have arranged to have all of your arrays aligned. The average performance gain is about 20% with this option.

-Vnotestalign

Suppress generation of runtime alignment tests (i.e. assume non-aligned data).

-Vcodespace

Request optimization of codespace over time. Turns off vector loop unrolling and runtime testing for alignment, for example. May seriously degrade performance.

-Vforcesingle

Use 4-byte ("single precision") floating point instead of 8- byte ("double precision"). When double precision is not actually required for the application or subprogram (frequently the case), may allow use of AltiVec vectorization (which operates only on 4 byte floating point) and greatly increase performance. All references to double precision will be converted to single precision on output. DOUBLE PRECISION declarations are changed to REAL, double precision constants are changed to single precision, etc.

-Vfixed

Input source is fixed format (overrides default implied by file extension: .f90 = free format).

-Vfree

Input source is free format (overrides default implied by file extension: .f = fixed format).

-Vinline

Request automatic inlining of small "leaf" procedures. May improve performance.

-Vleastrecent

Use least-recently-used version of load/store (so that the vector data does not stay in cache). Default is to use normal (cacheing) versions. We think that this option will only pay off if you have huge data sets.

-Vmessages

Request VAST to display vectorization messages while processing the input program. These messages are sent to the standard output.

-Vnoassoc

Specify that optimizations that could change the last few bits of the result (relative to the scalar original results) not be done. (No associative transformations). Can seriously degrade performance.

-Vnodepchk

Assume potential dependencies are not real dependencies. May improve performance by allowing more loops to vectorize.

-Vnointeroverlap

Assert that pointers in vectorizable loops don't overlap.

-Vnovector

Turns off vectorization. You can turn it back on for selected loops with the "#directive vd_vector" directive. This is useful for selective vectorization.

-Vskip

Skip all transformations. There are some non-vector optimizations that VAST does; this should turn off everything. You can enable specific loops with the CVD\$ NOSKIP directive.

-Vwide_lines

Accept 132-column fixed-format input.

VAST-F/Altivec options

The options can have various parameters, including:

routines = names of Fortran subprograms, separated by commas.

filenames = Unix file names, separated by commas.

nnn =integer constant

The available options are:

[-D *directive*[:*routine,routine...*]]

Directive to process at invocation.

[-F *filename*]

File name to divert command line processing to. The -F parameter allows redirection of command line input to a file. This is useful when many options are specified (e.g. -D parameters), or to insure a uniform set of invocation options over many invocations.

[-H *path*]

Directory in which to search for INCLUDE files and module information files.

[-I *routines*]

Names of subprograms that should be expanded inline.

[-J *nnn*]

Level to automatically expand from bottom of call tree (default 1).

[-L *path*]

Directory in which to put generated module information files.

[-M *nnn*]

Maximum threshold (code lines) for automatic inlining.

[-N *routines*]

List of routines not to inline.

[-Ppage *nnn*]

Page length (lines) for non-terminal listing (use with `-qt`) (default 66).

[-S *filenames*]

Files in which to search for routines to be inlined.

[-Y *routines*]

Names of routines that should be expanded inline (including called routines; nested expansion)

3. User directives

Overview

You may sometimes have information about the structure of a program and about its data that is unavailable to VAST-F through inspection of individual program units. For this reason, a way for you to guide VAST is supplied via user directives. User directives are treated as comments by Fortran compilers, thus preserving code transportability.

VAST directive format

VAST-F user directives have `CVD$` (fixed format) or `!VD$` (free format) in the first four columns of a line. Following the `$` is an optional scope parameter. `F` stands for "file" (meaning the directive applies until the end of all input files), `R` stands for "routine" (directive applies until the end of the current routine), and `L` for "loop" (directive applies to the next loop encountered). A blank following the `$` is equivalent to `L`. Some directives ignore the scope parameter.

Directives affecting `IF` loops must have `R` or `F` scope; directives with `L` scope apply only to `DO` loops.

The body of the directive begins after one or more blanks. Many directives can be preceded by `NO`, thus effecting the reverse operation.

<code>CVD\$</code>	<code>NODEPCHK</code>	<i>(Ignore potential data dependencies in the next loop.)</i>
<code>CVD\$R</code>	<code>SKIP</code>	<i>(Turn off transformation in the rest of this routine.)</i>
<code>CVD\$F</code>	<code>LIST</code>	<i>(Turn on listing for rest of file(s).)</i>

VAST directive summary

The full set of directives is summarized in the table below. The "scope" entry is either `I` for "immediate," meaning that the directive applies immediately; `L`, meaning that it applies to the next loop; `R`, meaning that it applies to the whole

routine; or LRF, which means that any of the loop, routine, or file options can be used to control the scope.

A short description of each of these directives follows the table. In addition, the more important directives are discussed in detail at the appropriate points in the sections on optimization.

VAST-F directives

Directive	Function	Default	Scope
SKIP/ NOSKIP	Disable/reenable transformations	NOSKIP	LRF
VECTOR/ NOVECTOR	Enable/disable vectorization.	VECTOR	LRF
DEPCHK/ NODEPCHK	Don't ignore/ignore potential dependencies.	DEPCHK	LRF
CNCALL	Allow concurrent calls in loop.	n/a	LRF
SWITCH	Pass new global switches.	n/a	I
NOASSOC/ ASSOC	Don't/do perform associative transformations.	ASSOC	LRF
NOEQVCHK/ EQVCHK	Don't/do check EQUIVALENCES to see if they cause data dependencies.	EQVCHK	LRF
PERMUTATION	Pass list of integer arrays that have no repeated values.	n/a	R
RELATION	Specify relationship between two simple variables.	n/a	R
NOLIST/ LIST	Turn off/on listing.	LIST	I
COUNT	Supply iteration count for loop.	n/a	I
ITERATIONS	Supply iteration count for classes of loops.	n/a	R
NOUNROLL/ UNROLL	Unroll loop.	UNROLL	LRF
AUTOEXPAND/ NOAUTOEXPAND	Automatically expand small routines inline.	NOAUTO	R
EXPAND	Expand particular routine(s).	n/a	I
NEXPAND	Nested expansion of particular routine(s).	n/a	I
SEARCH	Supply file/path location(s) for sources for inlined routines.	n/a	I

Transformation directives

These directives are used to change the way VAST-F transforms a loop.

NOVECTOR/VECTOR

NOVECTOR disables conversion of loops to vector form. VECTOR serves only to toggle back from NOVECTOR; it does not force conversion (see SELECT). The `-dv` switch is equivalent to NOVECTOR with file scope. NOVECTOR is a subset of SKIP.

SKIP

SKIP causes VAST-F to avoid transformation for the directed loop or routine. This is the directive to use if you want VAST-F to leave a loop untransformed. NOVECTOR is a subset of SKIP.

NOASSOC/ASSOC

By default, VAST-F transforms certain constructs into vector versions in which the order of operations may be different than the original (they have been associatively transformed). Because of the way numbers are internally represented in computers, this operation reordering may result in answers that differ slightly from the scalar original.

The NOASSOC directive disables all associative transformations, including generating reductions (such as sum or dot product of arrays), and operation reordering when minimizing dependent regions.

The `-d a` switch is equivalent to NOASSOC with file scope.

Data dependency directives

These directives are used to help VAST-F decide whether data dependency conflicts actually exist in a loop. If you know that an operation is not recursive, you can supply one of these directives to inform VAST-F. (Data dependency directives are discussed further in a later chapter.)

NODEPCHK/DEPCHK

When elements of an array are modified within a loop, VAST-F must determine the exact storage relationship of these elements to all other references to the array in the loop. This must be done to ensure that the references do not overlap, and thus can safely be executed in parallel. When the relationships cannot be determined, VAST-F issues a *potential dependency* diagnostic. The NODEPCHK directive asserts that all such potentially recursive relationships are in fact not recursive. It does not, however, force the optimization of operations that are unambiguously recursive. The DEPCHK directive is used only to toggle back to the default state. The `-d d` switch is equivalent to NODEPCHK with file scope.

NOEQVCHK/EQVCHK

NOEQVCHK directs VAST-F to ignore relationships between variables caused by EQUIVALENCE statements, when examining the data dependencies in a loop. The `-d e` switch is equivalent to NOEQVCHK with file scope.

PERMUTATION

The PERMUTATION directive declares an integer array to have no repeated values. This is useful when the integer array is used as a subscript for another array (indirect addressing). If it is known that the integer array is used merely to permute the elements of the subscripted array, then it can often be determined that no feedback exists with that array reference.

RELATION

The RELATION directive advises VAST-F that a specified relationship exists between two integer variables or between an integer variable and an integer constant. This information may be useful to VAST-F in resolving otherwise ambiguous array relationships.

Listing directives

NOLIST/LIST

Listing of the input source can be selectively suppressed with the NOLIST/LIST directive pair. If NOLIST (or the `-q l` switch) is in force when the END statement is encountered, the rest of the listing (messages, translated source, summaries) is suppressed unless specifically enabled via `-p` switches.

SWITCH directive

You can set (or change) global switches with the SWITCH directive. SWITCH directives may be inserted into the source program. The format is:

```
CVD$ SWITCH[, -e ss][, -d tt]  
    [, -g uu][, -f vv]  
    [, -p ll][, -q kk]  
    [, -r zz][, -n xx]  
    [, output format parameters]  
    [, -P nn], -C routines]  
    [, -I routines][, -N routines]  
    [, -Y routines]
```

corresponding to the invocation switch parameters described in the previous chapter.

Specifying directives on invocation

The `-D` invocation parameter can be used to specify directives without inserting them in the actual input source code. The format is:

```
-D directive[:routine,routine,...]
```

Where `directive` is any VAST-F directive, and `routine` is a routine in the input source to which the directive is to be applied. If no routine names are supplied, the directive applies to the entire input source. Multiple `-D` parameters can be supplied. Optionally, `-D` parameters can be placed in a file and invoked with the `-F` command line parameter.

4. VAST listing

Overview

Optimizing Fortran loops for vector execution is a complex task, and to do the best possible job, VAST-F may require some assistance from the user. Two paths of communication are available for this purpose: (1) VAST informs you of the actions it takes on the program (which loops were optimized, which loops were not optimized and the reasons for their rejection); (2) you can pass information and commands to VAST via directives inserted into the program (see section 2), or via global switches on the VAST invocation command (see section 2).

The full VAST listing consists of four parts: a listing of the input source with a graph of the loop structure showing the disposition of each loop; a block of diagnostic messages; a listing of the output transformed source; and summaries of loops and overall statistics. Any part of the listing can be separately enabled or disabled via the listing switches shown in the table below. An example of a full listing is also given below.

Input source listing

The input source lines are numbered on the listing. Source lines coming from `INCLUDEs` are numbered as well. These line numbers are used in the messages, output source listing, and summaries. The codes used for the loop disposition graph are listed below:

Code	Meaning
A	No action requested. (Optimization turned off.)
D	Data dependent. (Vectorizing loop could give wrong answers.)
E	Deleted. (Results of the loop are never used.)
H	Too short, not enough iterations.
I	Inlined. (Routine referenced on this line is expanded.)
N	Not chosen. (Loop is not optimized.)
R	Unrolled. (Several iterations will be calculated each pass.)

- T** Translation problem. (No optimization for this loop.)
- V** Vectorized.

Diagnostic messages

VAST's diagnostic messages appear in a group at the end of the source listing. Each message includes the line number and (if relevant) a variable name. These messages are VAST's means of notifying you of what problems it encountered in optimizing the loops in the source program. There are several different types of diagnostics.

Optimization inhibition messages

Translation Diagnostic. Describes a problem or potential problem in executing a loop in parallel. May prevent loop from being optimized.

Data Dependency Conflict. A real or potential feedback from one loop pass to the next prevents the safe use of vector operations. Potential feedback may result in generation of alternate versions of the loop. Otherwise feedback may prevent at least part of a loop from being optimized.

Informative messages

Warning Message. Some potentially troublesome input has been encountered.

Note Message. Tells about some opportunity or action on the input that might be of interest.

Error messages

Syntax Error. A construct that is not legal in Fortran has been encountered. No translation is done for this program unit.

Internal Error. An internal problem with VAST-F has been detected. No further processing is done for this subprogram; translation is attempted again for the next subprogram in the input file. Please report the error.

You can suppress each of these types of messages independently via the `-q` parameter on the VAST-F command line or on the `SWITCH` directive (see section 3). A later section a list of some of VAST-F's diagnostics, with further explanation of the messages, and tips on avoiding certain problems.

Output source listing

The output source code is listed following the messages. It shows the translated code. The numbers in the column at the right edge correspond to input source line numbers, to help in comparing the transformed source to the original source.

Summaries

Following the listing of the transformed source are two summary tables. One table (Loop Summary) summarizes the action taken for each loop in the routine. %CD is the percentage of code within the loop that is conditional and %DP is the percentage that is dependent. The final table (Event Summary) gives overall counts of errors, diagnostics, and loops transformed.

Listing control switches

The table below shows the switches that control the format of the listing file. These switches can be used either on the invocation (for example, `-q h`) or on the SWITCH directive (e.g., `CVD$ SWITCH, -q h`).

Listing control switches

Switch	Description	Default
b	List input line #s in columns 73-80 of output listing.	on
c	List data dependency conflict messages.	on
e	List event summary at end of routine.	on
f	List fatal error messages.	on
g	List translation diagnostics.	on
h	List input source lines.	on
i	List included lines.	on
l	Produce a listing.	on
n	List translated code.	on
p	List loop summary at end of routine.	on
r	Detect and list potential errors	off
t	Terminal listing: format output for 80 columns.	on
u	Show extent and disposition of loops in source.	on
w	List warning messages.	on
y	List syntax errors.	on

As an example, if you wanted to get a 132-column printer listing with no warning messages and no event summary, you would specify `-q t w e`. If you wanted to get potential errors listed, you would use `-p r`.

Notes on the listing switches:

b: List corresponding input line numbers in columns 73-80 of the listing of the transformed source. This switch is valid only if the `n` switch is on. This listing feature is useful in relating transformed source lines to original source lines.

i: List lines that come from INCLUDED files. When this switch is on, source lines obtained from INCLUDE files are listed. They are identified by a dash following the line number. This switch is valid only if the h switch (list source lines) is on.

l: Produce a listing. `-q 1` suppresses all parts of the listing (except the initial header, if the t switch is on). `-q 1` is equivalent to the NOLIST directive.

r: VAST-F can statically trace all uses and definitions of variables to look for potential programming errors. It generates messages on uses of variables that are undefined and definitions that are unused. This catches many programming errors, such as misspelled variable names, undefined arguments, missing, misplaced or redundant statements, and missing common declarations. This facility is turned on with the `-pr` switch; it is off by default.

t: Format the listing for a terminal. `-q t` results in a wide-format listing file, with printer control, pagination, and page headers, suitable for a 132-column line printer.

u: Show extent and disposition of loops in the input source listing. "Draws" a bracket alongside each loop, indicating how VAST-F treated the loop (parallelized, too short, data dependent, and so forth).

Listing page length

If you want a paginated, wide format listing suitable for a line printer, use the `-q t` option. The page length defaults to 66 lines. To change the number of lines per page, use the `-Ppage` parameter. For example,

```
-Ppage 60
```

changes page length to 60 lines per page, when getting a paginated listing (`-qt`)

5. AltiVec code generation

Overview

VAST-F/AltiVec has several features to take advantage of the capabilities of the AltiVec. These include:

- Dealing with arrays that are not aligned on 16 byte boundaries.
- Dealing with iteration counts that are not an even multiple of the vector register size.
- Vectorizing loops with array reduction operations.
- Unrolling the vector strip loop.
- Handling vectors that have non-unit stride.
- Vectorizing loops that contain indirect vector addressing (gather/scatter).
- Optimization of complex vectors, when the complex values are stored as two-element structures.

AltiVec code generation overview

There are no AltiVec vector extensions defined for Fortran, but there are AltiVec vector extensions defined for C. To take advantage of these, VAST-F/AltiVec translates each Fortran vectorized loop into vector operations in C, in a separate function called from the Fortran source.

Normally, you do not need to examine the vector code generated by VAST-F/AltiVec. This code, while generated at the source language level (in C), actually maps almost one-for-one with the vector instructions of the AltiVec unit; thus, it is fairly low-level and can appear voluminous in the output.

If you have some familiarity with the AltiVec unit, you may wish to examine the VAST/AltiVec code. You can sometimes shuffle the generated vector instructions to achieve slightly better performance - you would do this by editing the VASTed output. This is recommended for expert users only -- most users will not want to do this. To get the VASTed output, use the `-keep` switch

on the v77 driver command. This will keep the VAST intermediate files around (V*.f and A*.c).

Vector code generation sections

If you examine the code generated by VAST-F/AltiVec for a vectorized loop, you will find the following basic structure:

1. Run time test to see if loop should be vectorized. This may be a test to see if the iteration count is greater than zero, for example.
2. Run time test to see if the loop can be executed without need for runtime alignment of operands.
3. Vector declarations.
4. Initializations.
5. Vector strip loop.
6. Main vector calculation.
7. End of vector strip loop.
8. Cleanup for remaining iterations and for non-aligned vectors.
9. End of vector code.

Vector code generation example

Here is an example of VAST/AltiVec code generation. This loop :

```
do 30 i = 1,n
    dy(i) = dy(i) + da*dx(i)
30 continue
```

is translated to this call in the Fortran source:

```
call v_saxpy11 ( n, da, dy, dx )
```

plus this C function (in a separate file):

```
v_saxpy11__( n, da, dy, dx )
float *da;
int *n;
float dy[1], dx[1];
{
    if ( *n > 0 )
    {
        int j1, j2, j3, j4, j5, j6, j7;
        __vector float dylu, dxlu, dalu;
        __vector float dyl8u, dy19u;
        __vector float da2u = (__vector float )(0);
        int j8;
        __vector float dy15u, dy16u, dx9u, dx10u, dy17u;
        __vector float dy12u, dy13u, dx7u, dx8u, dy14u;
        __vector float dy9u, dy10u, dx5u, dx6u, dy11u;
        __vector float dy2u, dy3u;
        __vector unsigned char dy4u = vec_lvsl(0, &dy[0]);
        __vector float dx2u, dx3u;
        __vector unsigned char dx4u = vec_lvsl(0, &dx[0]);
        __vector float dy5u, dy6u;
```

```

__vector unsigned char dy7u = vec_lvsl(0, &dy[0]);
__vector unsigned char dy8u = vec_lvsl(0, &dy[0]);
static __vector unsigned long j1v[3] = { (__vector unsigned long )
(0, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF), (__vector unsigned long )(
0, 0, 0xFFFFFFFF, 0xFFFFFFFF), (__vector unsigned long )(0, 0, 0,
0xFFFFFFFF) } ;
__vector signed short k1v = (__vector signed short )(0, 0, 0, 0, 0,
0, 1, 0);
vec_mtvscr( k1v );
*((float *)&da2u) = *da;
dalu = vec_splat(da2u, 0);
j6 = *n;
dy2u = vec_ld(0, &dy[0]);
dx2u = vec_ld(0, &dx[0]);
dy5u = vec_perm(dy2u, dy2u, dy8u);
for ( j1 = 0; j1 < (j6 - 4 * 4) + 1; j1 += 4 * 4 )
{
    j3 = j1 * sizeof(int );
    j2 = j3 + 4 * sizeof(int );
    dy3u = vec_ld(j2, &dy[0]);
    dx3u = vec_ld(j2, &dx[0]);
    dy9u = vec_ld(j2 + 16, &dy[0]);
    dx5u = vec_ld(j2 + 16, &dx[0]);
    dy12u = vec_ld(j2 + 32, &dy[0]);
    dx7u = vec_ld(j2 + 32, &dx[0]);
    dylu = vec_perm(dy2u, dy3u, dy4u);
    dy2u = vec_ld(j2 + 48, &dy[0]);
    dx1u = vec_perm(dx2u, dx3u, dx4u);
    dx2u = vec_ld(j2 + 48, &dx[0]);
    dylu = vec_madd(dalu, dx1u, dylu);
    dy6u = vec_perm(dy5u, dylu, dy7u);
    vec_st(dy6u, j3, &dy[0]);
    dy10u = vec_perm(dy3u, dy9u, dy4u);
    dx6u = vec_perm(dx3u, dx5u, dx4u);
    dy10u = vec_madd(dalu, dx6u, dy10u);
    dy11u = vec_perm(dylu, dy10u, dy7u);
    vec_st(dy11u, j3 + 16, &dy[0]);
    dy13u = vec_perm(dy9u, dy12u, dy4u);
    dx8u = vec_perm(dx5u, dx7u, dx4u);
    dy13u = vec_madd(dalu, dx8u, dy13u);
    dy14u = vec_perm(dy10u, dy13u, dy7u);
    vec_st(dy14u, j3 + 32, &dy[0]);
    dy16u = vec_perm(dy12u, dy2u, dy4u);
    dx10u = vec_perm(dx7u, dx2u, dx4u);
    dy16u = vec_madd(dalu, dx10u, dy16u);
    dy17u = vec_perm(dy13u, dy16u, dy7u);
    vec_st(dy17u, j3 + 48, &dy[0]);
    dy5u = dy16u;
}
if ( (j6 & (4 * 4 - 1)) == 0 )
    dylu = dy5u;
else
{
    for ( ; j1 < (j6 - 4) + 1; j1 += 4 )
    {
        j3 = j1 * sizeof(int );
        j2 = j3 + 4 * sizeof(int );
        dy3u = vec_ld(j2, &dy[0]);
        dylu = vec_perm(dy2u, dy3u, dy4u);
        dx3u = vec_ld(j2, &dx[0]);
        dx1u = vec_perm(dx2u, dx3u, dx4u);
        dylu = vec_madd(dalu, dx1u, dylu);
        dy6u = vec_perm(dy5u, dylu, dy7u);
        vec_st(dy6u, j3, &dy[0]);
        dy5u = dylu;
        dy2u = dy3u;
        dx2u = dx3u;
    }
}
j8 = ((int )&dy[0] & 15) >> 2;
j4 = j6 & 3;

```

```

if ( j4 > 0 )
{
    j3 = j1 * sizeof(int );
    j2 = j3 + 4 * sizeof(int );
    dy3u = vec_ld(j2, &dy[0]);
    dylu = vec_perm(dy2u, dy3u, dy4u);
    dx3u = vec_ld(j2, &dx[0]);
    dxlu = vec_perm(dx2u, dx3u, dx4u);
    dylu = vec_madd(da1u, dxlu, dylu);
    dy6u = vec_perm(dy5u, dylu, dy7u);
    j7 = j8 + j4;
    if ( j7 >= 4 )
    {
        vec_st(dy6u, j3, &dy[0]);
        if ( j7 > 4 )
        {
            dy18u = vec_ld(j3 + 16, &dy[0]);
            dy6u = vec_perm(dylu, dylu, dy7u);
            dy19u = vec_sel(dy6u, dy18u, j1v[j7-5]);
            vec_st(dy19u, j3 + 16, &dy[0]);
        }
    }
    else
    {
        dy18u = vec_ld(j3, &dy[0]);
        dy19u = vec_sel(dy6u, dy18u, j1v[j7-1]);
        vec_st(dy19u, j3, &dy[0]);
    }
    dy5u = dylu;
}
else
{
    j3 = j1 * sizeof(int );
    if ( j8 )
    {
        dy6u = vec_perm(dylu, dylu, dy7u);
        for ( j5 = 0; j5 < j8; j5++ )
            vec_ste(dy6u, j3 + (j5 - j8) * sizeof(float ), &dy[0]);
    }
}
}
}

```

For compactness the C output has been simplified somewhat (e.g. a runtime test for alignment has been omitted).

Alignment

Normally, the AltiVec instructions deal with data that are aligned on 16-byte boundaries. Fortunately, the AltiVec unit is able to permute non-aligned arrays into aligned vectors. VAST generates code to enable this permute operation on all arrays that are not explicitly declared (with the `ALIGNED` directive) to be aligned. If you know that your arrays are aligned, and you always use the array starting with element zero, then you can use the `ALIGNED` directive to inform VAST of this and avoid the overhead of the non-aligned permutation code.

In the vector code generation example (above), the `vec_lsv1` and `vec_lsvr` instructions set up permutation indices that are used in the vector loop by the `vec_perm` instructions to align the data.

Iteration count cleanup

VAST generates special code to compute the remaining loop iterations, when the iterations desired are not a multiple of the vector size. The final elements are masked or stored element-by-element so that only the appropriate elements in the result arrays are changed.

Array reductions

Reduction operations include the summation of all elements of an array, finding the maximum element in an array, and taking the dot product of two arrays. Reduction operations take array-valued data and reduce it to a scalar value.

Processing reductions takes special code on the AltiVec to create a vector of “partial reductions” that is then reduced into the resulting scalar.

Example:

```
for (i=0;i<n;i++)    (will vectorize)
    s = s + a[i]*b[i]
```

Vector strip unrolling

VAST unrolls the vector strip loop to provide more instructions to overlap for the AltiVec compiler. This can provide substantially improved performance. VAST will automatically unroll the vector strip loop where it may provide a performance benefit, and will unroll it either two or four times.

For aligned operands, loops with four or less unique vector references will be unrolled four times, loops with 24 or less unique vector references will be unrolled twice. For non-aligned operands, the unrolling criteria are stricter, since more registers are required.

In the example on the previous pages, the vector strip loop was automatically unrolled four times.

6. Loop Optimizations

Overview

The transformations in this section all involve the manipulation of single-level loops and/or loop nests to reduce loop overhead or to expose more opportunities for the compiler's instruction optimizations.

Loop fusion

Adjacent loops with identical bounds can often be merged into a single loop. This reduces loop overhead and provides additional instructions for scheduling. There is a maximum of five loops and 50 total lines of fused code; this maximum can be increased with the `-G` parameter which will increase the total lines examined. For example, `-G 100` will examine 100 lines for fusion.

Example:

```
DO 100 I = 1, N
    A(I) = B(I) + C(I)
100 CONTINUE
C
DO 200 I = 1, N
    D(I) = B(I) - C(I)
200 CONTINUE
```

Translation:

```
DO 100 I = 1, N
    A(I) = B(I) + C(I)
    D(I) = B(I) - C(I)
100 CONTINUE
```

Loop rerolling

VAST-F will take loops that have been hand unrolled and put them back into their original state. This may allow the compiler (or VAST-F) to subsequently unroll the loops to a more optimal level for the target system.

Example:

```
DO 20 I = 1, N, 2
    A(I) = B(I) + C(I)
    A(I+1) = B(I+1) + C(I+1)
20 CONTINUE
```

Translation:

```
DO 20 I = 1, (N+1)/2*2
    A(I) = B(I) + C(I)
20 CONTINUE
```

Example:

```
DO 310 I = 1, 96, 5
    S = S + A(I)*B(I) + A(I+1)*B(I+1) +
*         A(I+2)*B(I+2) + A(I+3)*B(I+3) +
*         *A(I+4)*B(I+4)
310 CONTINUE
```

Translation:

```
DO 310 I = 1, 100                (Rerolled into one dot product.)
    S = S + A(I)*B(I)
310 CONTINUE
```

Loop unrolling

Loops can be unrolled automatically under various criteria, or under user direction with the UNROLL directive or -U command line switch.

For example, loops will be unrolled automatically when there is only one operation in the loop (thus not affording enough computation to overlap with the branch); two copies of the body of the loop will be executed in these cases.

Example:

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

Translation:

```
J1S = IAND ( N, 1 )
C
```

```

        DO 100 I = 1, J1S
            C(I) = A(I) + B(I)
100    CONTINUE
C
        DO 101 I = J1S+1, N, 2
            C(I) = A(I) + B(I)
            C(I+1) = A(I+1) + B(I+1)
101    CONTINUE

```

Also, when the loop iteration count is 4 or less and the body of the loop is small, the loop will be completely unrolled and replaced with assignment statements.

Example:

```

        DO 100 J = 1, M
            DO 100 I = 1, 3
                A(I,J) = 0
100    CONTINUE

```

Translation:

```

        DO 100 J = 1, M
            A(1,J) = 0
            A(2,J) = 0
            A(3,J) = 0
100    CONTINUE

```

The `-U` command line option can be used to control the depth of unrolling. The numerical parameter supplied is used by VAST to determine the desired size of the unrolled loop in statements. For example, `-U16` indicates a desired size of 16 statements the loop; a loop with four assignments would be unrolled four times.

To allow you to control the unrolling process, the `UNROLL` directive is provided, as follows:

```
CVD$ [ {L,R,F} ] [NO]UNROLL [ (nn) ]
```

When routine or file scope is specified, automatic unrolling of loops is enabled or disabled over that scope. The optional parameter, which must be a constant, specifies the threshold loop iteration count for automatic unrolling. Loops with an iteration count greater than this value will not be unrolled. By default, VAST-F will automatically unroll loops of length 4 or less.

To force a loop to be explicitly unrolled, use the `UNROLL` directive with local scope immediately preceding the loop. In this case, the optional parameter is taken as the number of times to unroll the loop. If a parameter is not supplied, VAST-F uses an internally calculated function of the loop length, loop complexity, and default threshold.

Short inner loops are automatically unrolled if:

- The iteration count is constant, and below the threshold (default: 4)
- The iteration count times the number of statements in the loop is less than 32.

- The loop contains only assignment statements. No branches, I/O statements, or external references are allowed.
- The DO loop control parameters are integer.
- The last value of the loop index is not required after the loop is executed.

These restrictions do not apply to loops unrolled explicitly by directive. The only inhibitors in this case are assigned GOTOs and I/O keywords other than END=, ERR=, FMT=, and UNIT= . Outer loops may be unrolled by directive also.

Store postponement

By default, array stores in (optimized but non-vectorized) unrolled loops are “postponed”:

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

Translation:

```
DO 101 I = 1, 100, 2
    C1 = A(I) + B(I)
    C2 = A(I+1) + B(I+1)
    C(I) = C1
    C(I+1) = C2
101 CONTINUE
```

Where the right-hand sides of all unrolled assignments corresponding to a single original assignment are computed and temporarily stored into scalars, and then the temp scalars stored into the actual array locations.

In some cases this allows the compiler to overlap more instructions. To disable this feature, use the `-fp` switch.

Reduction unrolling

Reduction unrolling is a special case of inner loop unrolling. Multiple accumulator temporaries are created, to allow overlap of unrolled summation iterations. This transformation is under control of the associative option, as it changes the order of operations from the original.

Example:

```
do i = 1, n
    s = s + a(i)*b(i)
end do
```

Translation:

```
s1 = 0
s2 = 0
```

```

s3 = 0
s4 = 0
do i = 1, n, 4
    s1 = s1 + a(i)*b(i)
    s2 = s2 + a(i+1)*b(i+1)
    s3 = s3 + a(i+2)*b(i+2)
    s4 = s4 + a(i+2)*b(i+2)
end do
s = s + s1 + s2 + s3 + s4

```

IF loops to DO loops

VAST-F will turn IF loops into DO loops. This allows the compiler to optimize more easily certain loops, with the exposure of more regular looping structures. It may also create nested loop situations, which can be further optimized.

Example:

```

330 CONTINUE
    I = I + 1
    IF ( I .GT. N ) GO TO 340
    A(I) = 0.0
    GO TO 330
340 CONTINUE

```

Translation:

```

330 CONTINUE
    I = I + 1
    J1S = I
    IF ( N - J1S + 1 .GT. 0 ) THEN
        DO I = 1, N - J1S + 1
            A(J1S+I-1) = 0.0
        ENDDO
    ENDIF
340 CONTINUE

```

To be converted into a DO loop:

- The loop must have a single entrance and a single exit.
- The iteration count for the loop must be determinable at execution time before the loop is entered. The IF loop may contain other loops.

The `-q1` switch disables conversion of IF loops to DO loops. All directives (such as `NODEPCHK`) affecting IF loops must have routine or file.

DO WHILE to DO

DO WHILE statements can be converted to iterative DO loops when a fixed iteration count can be extracted from the loop.

Example:

```
DO WHILE ( I.GT.0 )
  A(I) = 0.
  I = I - 1
ENDDO
```

Translation:

```
DO I1X = 1, I
  A(I+1-I1X) = 0.
ENDDO
I = 0
```

Loop interchange

An outer loop is pushed inside an inner loop if the outer loop is a better candidate for optimization. Greatest preference is given to minimizing strides on arrays -- long strides can give poor cache performance. The outer loop may be pushed inside more than one inner loop.

Example:

```
DO 100 I = 1, 100
  DO 200 J = 1, 10
    A(I,J) = B(I,J)
  200 CONTINUE
100 CONTINUE
```

Translation:

```
DO J = 1, 10
  DO I = 1, 100
    A(I,J) = B(I,J)
  ENDDO
ENDDO
```

Outer loop unrolling

This transformation allows outer loops to be unrolled inside of inner loops, providing more instructions to optimize in the inner loop without decreasing the iteration count of the inner loop. This optimization will only be done if the inner

loop has only one operation in it or if it allows common expressions along the outer loop to be more easily eliminated.

For example, in the loop below $C(I)$ can be fetched one time and used four times in the unrolled loop.

```
DO 10 J = 1, 100
    DO 20 I = 1, M
        A(I,J) = B(I,J) + C(I)
20    CONTINUE
10    CONTINUE
```

Translation:

```
DO J=1,100,4
    DO 20 I = 1, M
        C1X = C(I)
        A(I,J) = B(I,J) + C1X
        A(I,1+J) = B(I,1+J) + C1X
        A(I,2+J) = B(I,2+J) + C1X
        A(I,3+J) = B(I,3+J) + C1X
20    CONTINUE
    ENDDO
```

Loop collapse

Loop nests that traverse all of the inner dimensions of the arrays in the loop can be automatically collapsed into single loops with larger iteration counts.

Collapse criteria:

- The loops must be tightly nested, with one loop index per array dimension.
- 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.
- All the array references that are indexed by the loops must conform, that is, have the same subscripting.

In the example below, all three dimensions are collapsed. The loops do not have to occur in the order of the subscripts.

```
SUBROUTINE NEST ( L,M,N,A,B )
DIMENSION A(L,M,N), B(L,M,N)
DO 100 K = 2, N-1
    DO 100 J = 1, M
        DO 100 I = 1, L
            A(I,J,K) = B(I,J,K)
100    CONTINUE
```

Translation:

```
      DO 100 K = 1, L*M*(N-2)
        A(K,1,2) = B(K,1,2)
100  CONTINUE
```

7. Data dependency analysis

Overview

VAST-F ensures that the optimized code gives the same answers as the original. For certain loops, parallel or vector execution would result (or could result) in incorrect answers. A loop in which results from one loop pass feed back into a future pass of the same loop is said to have a *data dependency conflict* and may not be completely optimized. (Such a loop is also said to be recursive or to contain recurrences.) In these cases, VAST detects the problem, reports it to the user, and leaves the loop in its original form.

VAST does extensive analysis of the arrays used in each loop nest, and examines the offset and stride of accesses to elements along each dimension of arrays that are both used and stored in a loop. If the uses and stores overlap on different iterations of a loop, or if they could possibly overlap, then there is a data dependency problem. In this case the order of execution of the iterations could change the results, and the order of execution of iterations where a loop is parallelized is not known, so the loop cannot be parallelized. Avoiding dependencies is important in getting the highest performance; this section describes some directives that VAST provides to help you do this.

Data dependency examples

Figure 8.1 demonstrates the concept of data dependency. Four similar loops are displayed. For each loop, the sequences of instructions that would be executed in scalar mode (one at a time) and in vector mode (whole arrays at a time) are also shown. Lowercase variables (such as a) stand for new values set in the current loop, whereas uppercase variables (such as A) stand for old values that were set before the loop started.

```

          a: new value of A                A: old value of A
-----8.1A-----
ORIGINAL LOOP:
  DO 71 I = 2,N
  71 A(I+1) = A(I)*B(I)+C(I)
VECTOR SEQUENCE CORRECT?
No. We are not using updated
values of A.
```

<pre> SCALAR SEQUENCE: a(3) = A(2)*B(2)+C(2) a(4) = a(3)*B(3)+C(3) a(5) = a(4)*B(4)+C(4) a(6) = a(5)*B(5)+C(5) : : </pre>	<pre> VECTOR SEQUENCE: a(3) = A(2)*B(2)+C(2) a(4) = A(3)*B(3)+C(3) a(5) = A(4)*B(4)+C(4) a(6) = A(5)*B(5)+C(5) : : </pre>
-----8.1B-----	
<pre> ORIGINAL LOOP: DO 72 I = 2,N 72 A(I-1) = A(I)*B(I)+C(I) </pre>	<pre> VECTOR SEQUENCE CORRECT? Yes. Sequence is identical. </pre>
<pre> SCALAR SEQUENCE: a(1) = A(2)*B(2)+C(2) a(2) = A(3)*B(3)+C(3) a(3) = A(4)*B(4)+C(4) a(4) = A(5)*B(5)+C(5) : : </pre>	<pre> VECTOR SEQUENCE: a(1) = A(2)*B(2)+C(2) a(2) = A(3)*B(3)+C(3) a(3) = A(4)*B(4)+C(4) a(4) = A(5)*B(5)+C(5) : : </pre>
-----8.1C-----	
<pre> ORIGINAL LOOP: DO 73 I = 2,N 73 A(I+K) = A(I)*B(I)+C(I) </pre>	<pre> VECTOR SEQUENCE CORRECT? ? Depends on K. Here, if 0<K<N-1 then vector is not right. </pre>
<pre> SCALAR SEQUENCE: a(2+K) = A(2)*B(2)+C(2) a(3+K) = ... : Can't show more of sequence because we don't know where A is being changed. </pre>	<pre> VECTOR SEQUENCE: a(2+K) = A(2)*B(2)+C(2) a(3+K) = A(3)*B(3)+C(3) a(4+K) = A(4)*B(4)+C(4) a(5+K) = A(5)*B(5)+C(5) : : </pre>
-----8.1D-----	
<pre> ORIGINAL LOOP: DO 74 I = 2,N,2 74 A(I+1) = A(I)*B(I)+C(I) </pre>	<pre> VECTOR SEQUENCE CORRECT? Yes (Because stride of 2 makes operation non-recursive) </pre>
<pre> SCALAR SEQUENCE: a(3) = A(2)*B(2)+C(2) a(5) = A(4)*B(4)+C(4) a(7) = A(6)*B(6)+C(6) a(9) = A(8)*B(8)+C(8) : : </pre>	<pre> VECTOR SEQUENCE: a(3) = A(2)*B(2)+C(2) a(5) = A(4)*B(4)+C(4) a(7) = A(6)*B(6)+C(6) a(9) = A(8)*B(8)+C(8) : : </pre>

Figure 8.1 Data dependency analysis

It is easy to see that the scalar and vector sequences for Figure 8.1A are not the same; the vector version uses only old values of A, while the scalar version uses new ones. VAST-F detects that this loop is not safe to optimize, puts out a data dependency conflict message, and leaves the loop in its original form (the loop is "rejected"). In contrast, the scalar and vector sequences for Figure 8.1B are identical; no feedback of results from one loop pass to another is occurring here. VAST-F recognizes that this loop is safe to optimize and does so.

The situation is less clear in Figure 8.1C; here the use of the variable K in A's subscript makes the proper scalar sequence impossible to determine at compile time, with some exceptions (see section 8.4). If K is 1, the loop functions like the recursive loop in Figure 8.1A; if K is -1, the loop is safe to optimize, as 9.1B was.

When it is not possible for VAST-F to tell if a loop is recursive or not, the loop is said to have *ambiguous subscripting*. Often the user knows that a loop (or perhaps all the loops in a routine or program) is not recursive, even though VAST-F cannot tell, as in 8.1C. In these cases, you can direct VAST-F to ignore potential recursions, via the -d d switch or the NODEPCHK directive. Examples are below.

Figure 8.1D shows the same loop as in Figure 8.1A, but with a DO increment of 2 instead of 1. VAST-F detects that the loop is now not recursive, because no results feed back into the calculation. (However this loop is not a good candidate for Altivec because the array stride is not one.) These four similar examples point out the sensitivity of data dependency analysis to offset and stride values of arrays that appear on both sides of the equal sign within a loop.

Data dependency analysis extends to more than just single line loops. In the loop shown below, the reference to A at the top of the loop conflicts with the store into A at the bottom. VAST-F prints a message to this effect and does not optimize the loop.

```

DO 1 I = 2,N                               (Not optimized.)
  TEMP = A(I-1)+A(I-2)
  B(I) = TEMP+3.0+A(I)
1  A(I) = SQRT(B(I))-5.0

```

VAST-F uses information from other array dimensions as part of its analysis where possible. The loop in the following example is optimized because VAST-F can see that the second dimension indexes of the A references can never be equal (because N is not equal to N+1) and thus there is no recursion. (The references to A are to two totally different column vectors - they do not share data.)

```

DO 2 I = 2,N                               (Optimized.)
2  A(I,N) = A(I-1,N+1)*B(I)+C(I)

```

Ambiguous subscript resolution

When it is not possible with information contained in the loop to determine the storage relationship between two references to an array, the situation is called *ambiguous subscripting* or *potential feedback*. In these situations, VAST looks for statements outside of the loop that may provide information that clears up the ambiguity. In the loop below, VAST finds the assignment to N2, which makes it clear that N1 is never equal to N2, and recognizes that there is no feedback.

```

N2 = N1 + 1
DO 100 I=1,N                               (Optimized.)
100 A(I+1,N1) = A(I,N2)*B(I)

```

Data dependency directives

NODEPCHK – no vector feedback

NODEPCHK is like NOSYNC but not as strong. NOSYNC specifies that there is no array overlap. NODEPCHK specifies that there is no vector feedback in the loop. For example:

```

DO I = 2, N
  A(I+K) = A(I) *B(I) + C(I)
ENDDO

```

Assume we know that $k=1$. Then, the loop uses only “old” values of A in computing all the new values of A. Thus there is no feedback of values calculated in the loop to future iterations of the loop. Thus the loop can be vectorized safely. We can indicate that the loop has no vector dependencies by adding a NODEPCHK directive in front of the loop.

```
CVD$  NODEPCHK
      DO I = 2, N
          A(I+K) = A(I) *B(I) + C(I)
      ENDDO
```

To summarize: If there is overlap, then you can use the NODEPCHK directive to indicate that there is no feedback of values between the stored and used portions of an array. If there is feedback of values, then you may want to restructure the calculation.

NOEQVCHK -- non-recursion in equivalences

It is very rare in real-world programs that recursion is hidden through the use of EQUIVALENCE statements. However, VAST must assume the worst and not optimize in situations where EQUIVALENCE statements could cause feedback. In programs where many of the variables are EQUIVALENCE d together, this can result in most of the loops being left unoptimized.

The NOEQVCHK directive is provided to tell VAST that EQUIVALENCE statements can be ignored for data dependency analysis. Use of this directive asserts that variables with different names do not overlap in storage. (This is almost always the case, anyway.) Equivalence checking can be suppressed for the entire input file by the `-d e` switch on the VAST command line or the SWITCH directive.

In the example below, several local arrays have been equivalenced to a large array in common (perhaps to save space). If the arrays could overlap (for instance, if the value of the variable N was 1500 in the DO 100 loop), then the DO 100 loop could not be optimized. However, as we know the arrays do not overlap, the NOEQVCHK directive can be applied to the whole routine.

```
COMMON /BIG/ POOL(100000)
DIMENSION A(1),B(1),C(1)
EQUIVALENCE (POOL(1),A(1)),(POOL(1001),B(1)),
1 (POOL(2001),C(1))
CVD$R NOEQVCHK          (Don't worry about equivalences.)
.
.
DO 100 I = 1, N
    A(I+IA) = B(I+IB) + C(I+IC)
100 CONTINUE
```

It is often better to recode the application to avoid EQUIVALENCE entirely.

RELATION -- specifying relationship between variables

You can use the `RELATION` directive to provide additional information to VAST about array subscript ranges, to help in determining if a loop is safe to optimize. The `RELATION` directive has the form:

```
CVD$ RELATION ( simple1 .rel. simple2 )
```

where `simple1` and `simple2` are simple integer variables (one of them can be an integer constant), and `rel` is one of the Fortran relational operators `GT`, `LT`, `GE`, `LE`, `EQ`, `NE`.

When VAST cannot otherwise determine whether the relationship between two uses of an array is recursive, it searches the `RELATIONS` supplied by the user for the current routine to see if they help.

`RELATION` directives are informative only and do not force any action.

`RELATION` directives apply for the whole program unit. If conflicting relations are given, the result is unpredictable. It is up to you to ensure that the relations specified are correct and consistent.

```
CVD$ RELATION ( J.GE.N )
...
DO 100 I = 1, N      (If J .GE. N, no overlap.)
    A(I+J) = A(I) + B(I)
100 CONTINUE
```

The `RELATION` directive is provided for situations where you are unsure if the `NOSYNC` directive (a blanket assertion of non-recursion) is safe, or know it is not, but have some information about relative values of index variables.

PERMUTATION -- declaring safe indirect addressing

When an array with a vector-valued subscript appears on both sides of the equal sign in a loop, feedback is possible even if the subscript is identical. Feedback will occur if there are any repeated elements in the subscripting array.

Sometimes, indirect addressing is used because the elements of interest in an array are sparsely distributed; in this case an integer array is used to point at the elements that are really wanted, and there are no repeated elements in the integer array (see the example below).

This information can be passed to VAST through the `PERMUTATION` directive. `PERMUTATION` asserts that the named integer arrays contain no repeated elements (they serve merely to permute the elements of the arrays they indirectly address).

Syntax:

```
CVD$ PERMUTATION ( ia1, ia2, ... , ian )
```

`PERMUTATION` declares the integer arrays (`ia1`, and so forth) to have no repeated values for the entire routine.

```
CVD$ PERMUTATION (IPNT)    (IPNT has no repeated values.)
...
DO 100 I = 1, N
    A(IPNT(I)) = A(IPNT(I)) + B(I)
```

Scalar dependencies

Scalar variables are unchanging single locations in memory, such as a simple variable (X). Array references whose subscript values are invariant in a loop (and thus represent a single location through all passes of the loop) are called *array constants*. Array constants are treated similarly to simple scalar variables by VAST.

Scalar variables that are modified in a loop can sometimes inhibit optimization by causing data dependencies. Scalar variables that are not modified in the loop do not inhibit optimization.

Carry-around scalars

Scalars that may be used before they are defined in a loop are called carry-around scalars. They may or may not be recursive. Recursive carry-around scalars inhibit optimization. All references to these variables are collected in a scalar loop and split out from the rest of the calculation if possible.

```

DO 3313 I = 1,N           (Not optimized.)
  A(I) = S + 1/S         (S is carried around.)
  B(I) = C(I) - A(I) + S
3313 S = B(I) + D(I)

```

Equivalenced scalars

In some circumstances, scalars that are EQUIVALENCED may inhibit data dependency analysis. The NOEQVCHK directive may be used to allow such operations to optimize, if the equivalencing does not actually create recursion (it almost never does).

```

COMMON /BLOCK/ A(999)
EQUIVALENCE (S,A(100))
...
DO 67 I = M, N           (Not optimized.)
  S = B(I)**2
  A(I) = S + 1.0/S
67 CONTINUE

```

8. Inline expansion

Overview

This section describes the inline routine expansion subsystem of VAST-F. Inline expansion is not invoked by default; you must turn it on explicitly. To get automatic inlining from the same file, you can use the `-e78` or `-J1` options on the command line.

Introduction

Programs can often receive a performance benefit from the expansion of the bodies of certain subroutines and functions into the loops that call them. This allows the calling loop as well as the body of the called routine to be optimized. Application codes sometimes have small external functions that are called from inside many loops; these functions are good candidates for inline expansion. Here is a small example of inline expansion:

Original:

```
DO 100 I = 1, N
    A(I) = CALC (A(I), X+B(I), 2.0)
100 CONTINUE
...
END
FUNCTION CALC (A,B,C)
CALC = A + SQRT( B**2 + C**2 )
IF (CALC.LT.0) CALC = ABS ( B + C )
END
```

Expanding function CALC in line:

```
DO 100 I = 1, N
    TEMP1X = X + B(I)
    CALC1X = A(I) + SQRT( TEMP1X**2 +
```

```

1      2.0**2 )
      IF ( CALC1X.LT.0 ) CALC1X =
1      ABS ( TEMP1X + 2.0 )
      A(I) = CALC1X
100 CONTINUE

```

Inline expansion reduces subroutine calling overhead. It also allows scalar optimizations such as invariant code relocation and common subexpression analysis to extend across the body of the subroutine as well as the body of the calling loop. Even more importantly, inline expansion may enable the parallelization of an outer loop that contains the call.

There are two modes of inlining: automatic and explicit. These modes can be requested by directive or by command line option.

Automatic inline expansion

The objective of automatic inlining is to get rid of "leaves" of the calling tree. There is no programmer intervention needed other than requesting inlining on the command line.

Automatic inlining criteria

When automatic inlining is enabled (via the `-e7` switch, the `AUTOEXPAND` directive or or the `-J` parameter) VAST expands every called subroutine or function which has less than a threshold number of lines of code (default is 50, but the user can change the threshold with the `-M` parameter), no calls inside the expanded routine (no nesting with automatic expansion), and no expansion inhibitors (commons must match, etc.).

Avoiding inlining

If there are routines you want to explicitly exempt from automatic inlining, you can use the `-N` parameter on the command line to specify them.

Automatic inlining level

You can change the level of automatic inlining with the `-J` parameter. This allows you to specify the level of routines to be inlined automatically from the bottom of the calling tree. The default when automatic inlining is requested is one level (routines that do not call anything else). Using `-J1` is equivalent to using `-e7`. Specifying `-J2` requests automatic inlining of routines at the bottom two levels of the calling tree, if they met the other criteria as well. As `-J` implies looking first in the same file, if your code is distributed with one routine in each file you might want to use `-d8` as well after the `-J` switch to save the compile time spent looking in the same file for other routines.

Explicit inline expansion

In explicit inlining, the user lists the routines to be expanded, or directs expansion in the line following a directive. The routines may be listed on the command line when VAST is invoked (-I), or passed on directives such as EXPAND.

Nested expansion can be requested with explicit inlining. The NEXPAND directive or -Y parameter will expand the indicated routines and all routines they call, leaving no external references.

Source code access

VAST can search for inlineable source code in various places:

- same file
- different file
- naming convention

When expanding `call routine`, VAST will by default look for file `routine.f`.

The SEARCH directive or -S parameter informs VAST where to look for source modules. A directory or file can be specified to point VAST at further source files for inlining.

Inline expansion directives

AUTOEXPAND directive

The AUTOEXPAND directive is used to invoke automatic routine expansion.

Format:

```
CVD$ AUTOEXPAND
```

You can use F, R, or L scopes on this directive. NOAUTOEXPAND cancels the action.

The AUTOEXPAND directive with file scope is equivalent to the -e7 switch. See below for a list of inhibitors to automatic inlining.

Explicit mode

In explicit mode, routines listed on a directive (see following section), or on the -I or -Y invocation parameters, are expanded without regard for the automatic mode criteria. If no list is given on the directive, it directs expansion of the single CALL or all function references in the immediately following statement. This reference need not be inside of a loop. See below for a list of inhibitors.

EXPAND directive

The EXPAND directive is supplied for explicit routine expansion. The format is:

```
CVD$ EXPAND [ ( routine1[, routine2[, ...]] ) ]
```

Where *routine1*, *routine2*, ... is a list of routines to expand in this routine. If a list is not supplied, expand the next statement. Scope is ignored on the EXPAND directive.

```
CVD$ EXPAND ( CALC )
    ...
    DO 100 I = 1, N
        A(I) = CALC( A(I), B(I)+1., N )
    100 CONTINUE
    ...
    END
```

You may also supply a list of routines to expand on the command line, via the -I option.

NEXPAND directive

The NEXPAND directive operates in the same manner as the EXPAND directive, except that the named routines are expanded in a nested manner until no calls remain. In this way, you can expand a whole sub-tree of subroutine and function calls. On the command line, you can use the -Y switch.

```
CVD$ NEXPAND [ ( list ) ]
```

SEARCH directive

You can tell VAST where to look for the routines to expand, via the SEARCH directive or -S option on the command line. The SEARCH directive has routine scope by default. It may be used with file scope (CVD\$F) before the first instance of expansion in the input stream.

Directive format:

```
CVD$ SEARCH ( filename [,filename ...] )
```

Invocation parameter format:

```
... -S filename [,filename ...]
```

where *filename* is a character string that identifies a file in which to search for the routines to be expanded. (Note that this directive does not identify which routines are to be expanded, just where to look for routines that have been so identified by directive or by automatic criteria.)

If *filename* is the special entry *.f then, for example, the routine xyz will be looked for in file xyz.f. (This is the default search method.)

Where to get the source code

In order to expand a routine, VAST needs to know where to find its source. The source location depends on programming style and on the operating system user interface.

Same file

Called routines can be searched for in the same file as the calling routine (stacked input). This necessitates an initial pass by VAST through the entire input file (and files INCLUDED therein) to build a directory of the program units in the input file. The switch `-e8` enables this initial pass, which by default is not done.

Explicitly named file

You can supply (via the `SEARCH` or `-S` invocation option directive, described previously) the name of a file in which to search for a particular called routine.

Implicitly named file

Fortran programs are frequently stored such that each routine of the program resides in a separate file with a canonical name (for example, the name of the routine followed by `.f`). This is the default search method.

Possible problems

Separate compilation

Fortran allows program units to be compiled separately and linked together. Because different program units may be compiled at different times, it is not possible to make inline expansion completely foolproof without resorting to "programming environments" which place restrictions on the user. Even if routines are initially compiled from the same input file, an object of a modified version of a routine could be supplied at a later link.

For example, suppose program A has subroutine B, which calls subroutine C. Let's say subroutine C is expanded into subroutine B. Later, we decide to change the calculation in subroutine C, and so we edit it; rather than recompiling the entire program, we just recompile C and link it with the previously compiled routines. Our changes to C are not present in the actual code executed because C is no longer called from B (it was expanded).

Thus, you must be involved in the routine expansion process at least to the point of knowing which routines must be recompiled when a change is made. For this reason, VAST generates both a warning message for every expansion and an expansion event table so that it is clear what has been expanded.

Code size

A problem that may result from inline expansion is larger code size; if too many routines are expanded, or the expanded routines are too large, the size of the compiled code may reach unacceptable proportions. This potential problem can be controlled through judicious application of this transformation.

Debugging

Inline code expansion can complicate user run-time debugging; if the program fails in an expanded section of code, the error is reported in a different routine than the one it originally appeared in.

VAST records the original line number of the routine invocation on all the expanded lines.

Compilation rate

Inline expansion may result in two passes over the entire program, and longer compile times, depending on how much code is brought in line.

Nested expansion

Nested expansion will be done only if specified by user directives or with the `-Y` option, or if the `-J` parameter is used with level greater than one. Nested routines will not be expanded in default autoexpansion mode, thus reducing the possibility of code size mushrooming.

If you want nested routines to be expanded, you must explicitly specify each of them in the chain in an `EXPAND` directive or `-I` parameter, or specify the top routine in the call chain in an `NEXPAND` directive or `-Y` parameter.

Analysis inhibitors

There are several ways in which analysis may be inhibited, thereby prohibiting expansion. An appropriate message informs you of a failed expansion. A full list of messages appears at the end of this section.

Expansion inhibitors

- The routine to be expanded cannot be located.
- Syntax errors are found in the expansion routine.
- The arguments used in the calling sequence do not match the arguments in the expansion routine. (See next section for discussion).
- There is a conflict between common blocks of the calling routine and the expansion routine. (See next section for rules.)
- The routine to be expanded contains `NAMELIST`.
- The routine to be expanded contains `SAVE` statements. (Can be overridden by the `-g6` switch -- see below.)

- The routine to be expanded contains DATA statements for local variables, whose value is changed in the routine. (Can be overridden by the `-g6` switch -- see below.)
- A function is being expanded in a DO WHILE statement or an ELSE IF statement.
- A function name referenced in the expansion routine conflicts with a non-function name used in the calling routine.

Inhibitors specific to automatic expansion mode

In automatic mode, all calls to routines that meet the following criteria are expanded:

- The routine to be expanded has less than the maximum allowed number of non-comment lines. (The default is 50. This number can be changed via the `-M` parameter on the invocation.)
- The routine to be expanded does not call any other external routines. (If the `-J` option is used, it can expand nested references if they are within the number specified from the bottom of the call tree).
- There are no inhibitors to the expansion (common blocks that do not agree, and so forth).

If these parameters are unsatisfactory, you can always resort to explicit mode. The objective of automatic mode is to catch small, simple frequently-called external functions. More demanding cases must be explicitly requested.

Note that although called automatic, this mode still requires you to enable an option or insert a directive. An informational message is issued for each expansion action. This is to remind you that routines that have been expanded into need to be recompiled each time the expanded routine is changed.

Inline expansion abilities

This section presents some of the operations that are performed during inline expansion of subroutines and functions.

Naming conventions

Where necessary, VAST renames all variables and parameters in expanded program units in the following manner. The first four characters of the variable are combined with an integer number and the suffix X to create a unique name. For example, the local variable `I` could become `I1X`, and the next reference to `I` in another expanded routine could become `I2X`.

In this way, VAST keeps a relationship between the original user name and the inlined name; this makes the generated code much more readable.

Common blocks

All common blocks that are used in both routines must agree. COMMON blocks can be just in the caller or just in the callee. Common blocks that appear only in the expanded routine are added to the calling routine. If present in both, the names of objects in the common do not have to match, but sizes of corresponding objects must match. The caller common may be a superset of the callee common (the sizes must match, up to the end of the callee common).

Arguments

Dummy parameters are replaced with their corresponding actual arguments. In the case of expressions passed as actual arguments, VAST will create a temporary variable to hold the expression and use the temporary in each of the places the dummy argument appeared in the called routine.

The number of actual and dummy arguments must match. However, array elements can be passed to scalars, higher dimensioned arrays can be passed to lower dimensioned arrays, and single dimensioned arrays can be passed to multiple dimensioned arrays.

Unique names

Local variables used in the expanded routine are checked against the identifiers defined in the caller, and made unique. If already unique, they are left alone.

Similarly, constant parameter names (from PARAMETER statements) are examined and changed if necessary to avoid conflict with any name from the calling program. If identical in name and value with a constant parameter in the calling program, then no change is made.

Labels

All labels used in the called routine (FORMATS, CONTINUES, DOS, and so forth) are changed so that there is no conflict with labels in the caller.

Returns and return values

RETURN statements are changed into branches to a new label in the caller that represents the end of the called routine. If it is an alternate RETURN statement, the branch corresponding to that RETURN is directed to the specified label.

In addition, references to the function name as a variable in an expanded function are replaced with another name. (Calls to the original function may still exist unexpanded.)

Inlining ENTRYs

VAST inlines calls to ENTRY points themselves. If both the subroutine itself and an entry inside of it are called, they can both be inlined.

Cleanup of inlined code

When constants are passed to routines, there are often opportunities to simplify the inlined code. VAST uses global constant propagation and global expression simplification to eliminate dead code resulting from expansion constants. This can result in dramatic reduction in the size of the expanded routine if many constants are passed.

Example

The code segment below shows two-dimensional arrays passed to one-dimensional arrays, and significant cleanup of the expanded code (dead code elimination) due to the constants passed as arguments.

```
call daxpy(n,t,a(k+1,k),1,a(k+1,j),1)
.
.
.

subroutine daxpy(n,da,dx,incx,dy,incy)
double precision dx(1),dy(1),da
integer i,incx,incy,ix,iy,m,mpl,n
if(n.le.0)return
if (da .eq. 0.0d0) return
if(incx.eq.1.and.incy.eq.1)go to 20 (dead code, as
ix = 1                                both incx and incy
iy = 1                                are passed as one)
if(incx.lt.0)ix = (-n+1)*incx + 1
if(incy.lt.0)iy = (-n+1)*incy + 1
do 10 i = 1,n
    dy(iy) = dy(iy) + da*dx(ix)
    ix = ix + incx
    iy = iy + incy
10 continue
return
20 continue
do 30 i = 1,n (this survives)
    dy(i) = dy(i) + da*dx(i)
30 continue
return
end
```

Translation:

.

```

      .
      .
C***** Code Expanded From Routine:  DAXPY
      IF (N .LE. 0) GO TO 77001          (return)
      IF (T .EQ. 0.0D0) GO TO 77001     (loop 10 deleted)
      DO I = 1, N
          A(I+K,J) = A(I+K,J) + T*A(I+K,K) (2d to 1d array)
      ENDDO
77001 CONTINUE
C***** End of Code Expanded From Routine:  DAXPY
      .
      .
      .

```

Expansion of SAVE and DATA

In some cases it is desirable to inline a routine which has *SAVE*ed local data or *DATA*ed items which get stored into. Without the `-g6` switch specified, such a routine would be rejected for inlining with one of the following messages:

`SAVE stmt. inhibits expansion, change to COMMON stmt.`

or

`Expan. inhibitor - DATA stmt. implies saving of local var.`

If it is desired to expand such a routine, the routine itself must be rewritten. In order to retain the correct values of the *DATA* or *SAVE* variables in the inlined instances of the routine, a *COMMON* block and possibly a *BLOCK DATA* must be created for these items. In most cases VAST has the ability to do this automatically.

To force this type of expansion, in addition to the expansion switches `-g6` must also be specified and the routine (or entry) to be expanded must appear explicitly in the `-I` or `-Y` parameters.

Care must be taken when doing this type of expansion since either the routine must be inlined at *all* the places it is called in the program or the changed routine must be the version which is linked to the program. If an unchanged version of the routine is accidentally linked to the program, wrong answers will result.

For example:

```
vast2 -e78 -g6 -I xp1,xp2 bigsource.f
```

would cause `xp1` and `xp2` to be "forcibly" expanded wherever they are called in `bigsource.f`, and also whichever routines contain the entries `xp1` or `xp2` will be changed so as to be incompatible with the original version.

Inline expansion user messages

All inline expansion messages appear as warnings. VAST does not expand any routine that has caused the generation of one of the following messages, except for the `ROUTINE EXPANDED` message.

Inline expansion summary

VAST notifies you of any routines that have been expanded and also informs you as to why a particular routine was not expanded. If a listing was requested (`-pl`), a summary of the routines and all the locations where a routine was or was not expanded is displayed.

User messages

`SOURCE FOR ROUTINE NOT FOUND`

The expansion routine cannot be located, as specified by the `SEARCH` directive, `-S` option or by the default method.

`SOURCE FOR ROUTINE NOT FOUND IN INPUT FILE`

The input file has been defined as the location for expansion routines/functions, either by default or via switch. The routine/function is not found in the input file.

`EXPANSION ROUTINE IS TOO BIG FOR AUTOMATIC EXPANSION`

The routine has more than the maximum allowed number of non-comment lines (default 50; changeable via the `INLMAX` invocation parameter); use explicit expansion mode to expand.

`EMBEDDED CALL STATEMENT ENCOUNTERED WHILE EXPANDING`

The expansion routine references another subroutine; use explicit expansion mode to expand.

`SYNTAX ERROR ENCOUNTERED IN EXPANSION ROUTINE`

The expansion routine has a syntax error and will not be expanded.

`ARGUMENT MISMATCH BETWEEN CALL AND EXPANSION ROUTINE`

The arguments specified in the calling sequence of a subroutine or in a function reference do not match with arguments of the expansion routine/function. For example, this could result from a mismatch of data types.

`EXCEEDED MAXIMUM NUMBER OF EXPANDED ROUTINES`

The maximum number of routines/functions VAST will expand has been exceeded. Currently this number is 600.

`COMMON BLOCK MISMATCH BETWEEN CALLING AND EXPANSION ROUTINE`

Common blocks found in calling and expansion routine/function do not agree, or a `COMMON` variable in the expansion routine is used as a local variable in the calling routine. The `COMMON` in question is listed with this message.

FUNCTION IN EXPANDED ROUTINE CONFLICTS WITH NON-FUNCTION

A function found in the expanded routine is in conflict with a non-function element in the calling routine. The function name in question is listed with this message.

ROUTINE EXPANDED

This warning message will be issued whenever a routine/function has been expanded.

9. Diagnostic Messages

Overview

This section shows selected VAST-F messages, grouped by category. Most of the messages described here have to do analysis of loops for parallel execution.

Data dependency conflicts

Data dependency messages are always followed by the name of the variable which is causing the problem. If outer loop translation is being attempted, the label and index of the loop causing the problem is given as well.

"Feedback of array elements"

```
DO 4 I=1,N
4 A(I+1) = A(I) + B(I)
```

Feedback of results makes the loop recursive and thus unsafe to translate to partition.

"Feedback of scalar value from one loop pass to another"

```
DO 112 I = 1,N
A(I) = A(I) + SCA
112 SCA = A(I)/C(J)
```

The variable SCA is used in the first line of the DO loop to set $A(I)$, and then is set to a function of $A(I)$ in the last line. This creates feedback of elements of A from one loop pass to the next, which prevents optimization. SCA is called a "carry-around" scalar, as it carries a value around to the next pass of the loop.

"Potential feedback of array elements -- use directive if ok"

```
DO 3 I=1,N
3 A(I+J) = A(I) + B(I)
```

It is not clear whether there is feedback between the two uses of A in this loop or not (it depends on the value of J). Loops of this kind that the user is sure are

safe can be translated by putting the directive `CVD$ NOSYNC` in front of the loop. Here is another case where this message would result:

```
DO 1 I=1,N
1 B(I) = B(IB(I))+A(I)
```

`B(IB(I))` is a gathered array, and as the values in `IB(I)` are unknown, it may conflict with the assignment of elements of `B` on the left side of the equal sign. If the pattern of `B(IB(I))` is known not to overlap `B(I)`, then the `NOSYNC` directive should be used.

As a convenience, the `-dd` option switch or `NOSYNC` directive with routine or global scope may be used instead of loop-by-loop directives.

"Multiple store conflict"

```
DO 100 I = M,N
A(I) = B(I)
100 IF ( C(I) .GT. 0 ) A(I-1) = C(I)
```

Stores into overlapping sections of the same array must be done in the same order as in the scalar loop.

"Potential multiple store conflict -- use directive if ok"

```
DO 100 I=1,N
A(I+J)=B(I)
100 A(I+K)=C(I)
```

The loop above has a potential overlap between the two stores into `A`; usually in these situations there is no real overlap between the two sections of `A` and the `NOSYNC` directive should be used to allow the loop to translate.

"Feedback of array elements (equivalenced arrays)"

Actual feedback between arrays equivalenced together.

"Potential feedback (equivalenced arrays) -- use directive if ok"

```
COMMON /BLOCK/ A(999)
EQUIVALENCE (S,A(100))
...
DO 67 I = M, N
S = B(I)**2
A(I) = S + 1.0/S
67 CONTINUE
```

Either two arrays or an array and a scalar are equivalenced, and their storage relationship in the loop cannot be determined. Use the `NOSYNC` or `NOEQVCHK` directives or the `-dd` or `-de` switches to allow translation, if there is in fact no recursion.

"Equivalence of scalars prevents translation - use directive if ok"

```
COMMON / BLOCK / A(99)
EQUIVALENCE (A(1),S), (A(2),T)
```

```

...
DO 68 J = M, N                (Not translated.)
    S = B(I)**2
    T = C(I)**2
    D(I) = SQRT(S+T)
68 CONTINUE

```

Two scalars are in the same equivalence class and at least one is modified in the loop. The NOSYNC or NEQVCHK directives, or the -dd or -de switches, may be used to allow translation, if in fact there is no recursion.

"Too many data dependency problems"

The loop has exceeded the maximum number of data dependency conflicts allowed (10). Translation of the loop is abandoned at this point to avoid printing out further messages.

Translation Diagnostics

Translation diagnostics point out constructs that prevent a loop from being translated.

Statement types

These messages complain about types of statements that prevent translation.

"ASSIGN prevents loop translation"

```

DO 210 I = 1, N
    IF ( A(I) .GT. 0 ) ASSIGN 225 TO TOGO
210 CONTINUE

```

"ASSIGNed GOTO prevents loop translation"

```

DO 220 I = 1, N
    GOTO TOGO
220 CONTINUE
225 CONTINUE

```

"Computed GOTO prevents loop translation"

```

DO 230 I = 1, N
    GO TO ( 231, 232, 233 ) IGO(I)
231    B(I) = 0
232    B(I) = B(I) + A(I)
233    B(I) = B(I) * C(I)
230 CONTINUE

```

"I/O statements prevent loop translation"

```

DO 240 I = 1, N

```

```

        WRITE ( IOUNIT ) A(I)*B(I)+C(I)
240  CONTINUE

```

"RETURN prevents loop translation"

```

        DO 250 I = 1, N
            IF ( X(I) .LT. 0 ) RETURN
            Y(I) = SQRT(X(I))
250  CONTINUE

```

"STOP prevents loop translation"

```

        DO 260 I = 1, N
            IF ( X(I) .LT. 0 ) STOP 99
            Y(I) = ALOG ( X(I) )
260  CONTINUE

```

Branches

The messages below relate to handling of conditional operations.

"Backward transfers prevent loop translation"

```

        DO 107 I=1,N
104  A(J) = SQRT(B(J-1) + D(I))
        J = J + 1
        IF (B(J) .GT. LIMIT) GO TO 104
107  C(I) = A(J-1)

```

The branch to label 104 is backward, not forward, and prevents translation. In some cases however, backward transfers may be converted into separate DO loops.

"Branches out of the loop prevent translation"

```

        DO 108 I=1,N
            IF (A(I).LT.0) GO TO 777
108  B(I)=6.0

```

The label "777" is not in the loop.

"Branching too complex to translate this loop"

Because of limits on time and table space, conditional constructs with more than six simultaneously active conditions (i.e. IF-THENS, IF-GOTOS, etc.) cannot be optimized.

External references

The messages below deal with external references in loops.

"Subroutine call prevents loop translation"

```

        DO 110 I = 1,N
            A(I) = SQRT(B(I)/C(I))

```

```

        CALL REVAMP(A(I),D(I))
110   D(I) = EXP(A(I)+X)

```

"Reference to function that has no array version"

```

        DO 115 I=1,N
115   A(I) = MYFUNC(B(I))

```

References to non-intrinsic functions in a loop prevent translation of the loop.

DO statement

These messages relate to DO statements.

"DO statement parameters must be integer for array transl."

```

        DO 100 W = 1.0001, 1000000.
           A(I) = W
100   CONTINUE

```

Non-integer variables or constants are not allowed as the loop index or in the start, end or increment fields for translation purposes to array syntax.

"User function references not allowed in iteration count"

```

        DO 210 I = 1, NLEN(J,K)
210   A(I)=0.

```

In this example NLEN is an external user function. Such functions cannot appear in the iteration count for a DO loop (they cannot be in the start, end or increment fields of the DO statement) for the loop to be optimized. Statement functions are allowed, however.

Miscellaneous

These messages fall into none of the previous categories.

"Null loop body"

```

        DO 111 I=1,N
111   CONTINUE

```

Nothing in the loop. (The loop is not eliminated.)

"Character data type inhibits loop translation"

```

        DO 200 I = 1, N
           P(I)(1:2) = Q(I)(2:3)
200   CONTINUE

```

Use of character type data prevents a loop from being considered for translation.

Warnings

Potential Errors

These warning messages relate to potential errors in the input program. Use the switch `-pr` to get this kind of message for your code.

**** Variable used but never defined ****

A local variable is used in executable statements but is never defined.

"Variable defined but never used"

A local variable is defined in executable statements but is never used.

**** Variable used but not defined ****

A local variable is used when it is undefined, although it is defined elsewhere in the program unit.

"Variable defined but not used"

This definition of a local variable is not used, although the variable is used elsewhere in the program unit.

"Variable appears only in argument list"

A local variable appears only once, in the argument list of a subroutine call.

"Dead code"

Flags a section of code which, because of the program's flow of control, can never be executed.

Obsolescent Features

Additional warnings are generated for obsolescent features that are no longer preferred usage.

Syntax errors

There are a very large number of syntax error messages. These messages are for the most part self explanatory, and so are not repeated here.

Internal Errors

Please report any VAST internal errors immediately to your support representative.

"Internal error detected (phase)-- please report"

Directive Errors

These messages describe errors in the way directives to VAST-F have been used.

"Unknown directive -- it is ignored"

```
CVD$ SCALARIZE
```

"Switch input error"

```
CVD$ SWITCH=-1
```

"Excess characters following directive"

```
CVD$ SKIP THIS LOOP, PLEASE
```

In this case, the characters following SKIP are invalid.

Notes

Notes are not generated by default. They can be enabled with the `-po` switch.

"IF loop converted to DO-loop"

An IF loop has been converted to a DO loop. (The resulting DO loop may or may not then be vectorized.)

10. Further Information

Crescent Bay Software

VAST was originally a product of the High Performance Computing Group of Pacific-Sierra Research Corporation. In October of 1998, all of PSR was acquired by Veridian, and the Compiler Tools Department became part of Veridian Systems Division. In April 2003 this group went independent as Crescent Bay Software.

VAST

Work began on the VAST (Vector and Array Syntax Translator) project in 1979, and VAST systems are in use at many high-speed computing sites worldwide. The VAST technology is used in a variety of ways in different systems, from an optimizing prepass of the compiler to a stand-alone translator.

In addition to **VAST-F/Altivec**, the VAST family includes:

VAST-C/Altivec, automatic vectorization of C and C++ programs for the Motorola Altivec architecture.

VAST-F/Parallel, automatic parallelization of Fortran language programs for shared memory parallel systems. Also includes full support for the OpenMP set of directives for user-directed parallelism.

VAST-C/Parallel, automatic parallelization for the C language.

VAST-DPC, Data Parallel C compiler (also compiles C*). Brings the Data Parallel computing model to the ANSI C language.

VAST/77to90, Fortran 77 to Fortran 90 translator. Many people have found it to be a very useful tool for migrating old code to the new language.

VAST/77toOpenMP, Fortran to OpenMP translator. Examines the entire program and creates OpenMP parallelism directives for loop nests that can be executed in parallel.

DEEP

DEEP is a programming environment that provides detailed performance information about programs running on many kinds of parallel systems. DEEP combines VAST compile time optimization information with VAST-profiled run-time information to provide a complete picture of your program's performance and potential bottlenecks, mapped back to the original source code.

Questions or Comments

If you have any questions or comments about VAST-F/AltiVec, please do not hesitate to contact:

Crescent Bay Software
10950 Washington Boulevard, Suite 230
Culver City, CA 90232
Phone: (310) 836-5183
Fax: (310) 836-7313

Index

ALIGNED pragma, 26
aligned switch, -Valigned, 10
alignment, 26
AltiVec, 23
AltiVec vector extensions, 23
ambiguous subscript resolution, 38
array constants, 41
associative, 31
associative transformations, 16
AUTOEXPAND directive, 44
automatic inlining, 43
automatic vectorization, 1
backward transfers, 57
branches out of the loop, 57
carry-around scalars, 41
codespace switch, -Vcodespace, 11
command line, 9
complex vectors, 23
-D invocation parameter, 18
data dependencies, 36
data dependency, 36
Data Dependency Conflict, 20
data dependency messages, 54
data dependency problems, 3
DEEP, 4, 62
diagnostic messages, 19, 20, 54

double precision, 11
double precision floating point, 2
driver, 5
EQUIVALENCE, 41
event summary, 21
EXPAND directive, 45
expansion of SAVE and DATA, 51
explicit inlining, 44
external references, 57
-F command line parameter, 12, 18
feedback of array elements, 54
forcesingle switch, -Vforcesingle, 11
Fortran 77, 1
-G fusion parameter, 28
IF loops into DO loops, 32
INCLUDE files, 22
inline expansion, 3, 12, 42
inlining switch, -Vinline, 11
inner loop unrolling, 29
input files, 10
input line numbers, 22
Internal Error, 20
iteration count, 27
least recent load/store switch, -Vleastrecent, 11
listing, 2, 9, 19
listing control switches, 21
listing, page length, 22
listing, wide format, 22
Loop Collapse, 34
loop disposition graph, 19
loop fusion, 28
loop interchange, 33
loop optimizations, 28
loop rerolling, 29
loop summary, 21

loop unrolling, 29
messages switch, -Vmessages, 11
multiple store conflict, 55
nested expansion, 47
NEXPAND directive, 45
NOASSOC, 16
noassoc switch, -Vnoassoc, 11
NOCONCUR, 16
NODEPCHK directive, 16, 38
nodepchk switch, -Vnodepchk, 11
NOEQVCHK, 17, 41
NOEQVCHK directive, 39
NOLIST, 17
nopointeroverlap switch, -Vnopointeroverlap, 11
Note Message, 20
novector switch, -Vnovector, 12
optimizations, 3
options, 10, 12
outer loop unrolling, 34
output file, 9
Pacific-Sierra Research Corporation, 61
performance tips, 2
PERMUTATION, 17
PERMUTATION directive, 40
potential error detection, 22
potential errors, 59
potential feedback, 38, 54
recurrences, 36
reductions, 27
RELATION, 17
RELATION directive, 40
routine expansion, 42
-S parameter, 44
scalar dependencies, 41
SEARCH directive, 44, 45

single precision, 11
SKIP, 16
skip switch, -Vskip, 12
store postponement, 31
SWITCH directive, 17
switches, 10, 17
Syntax Error, 20
Translation Diagnostic, 20
translation diagnostics, 56
unit stride, 2
UNROLL directive, 29, 30
unrolling, outer loops, 34
unrolling, reductions, 31
unrolling, vector, 27
user directives, 14
v77 driver, 1, 5
vastf command, 10
vector strip loop, 27
vector unrolling, 27
vectorization messages, 2, 11
Veridian, 61
-vforcesingle, 2
-Vnodepchk, 3
-Vswitches, 10
Warning Message, 20