

# CONVEX FORTRAN Optimization Guide

*Third Edition*



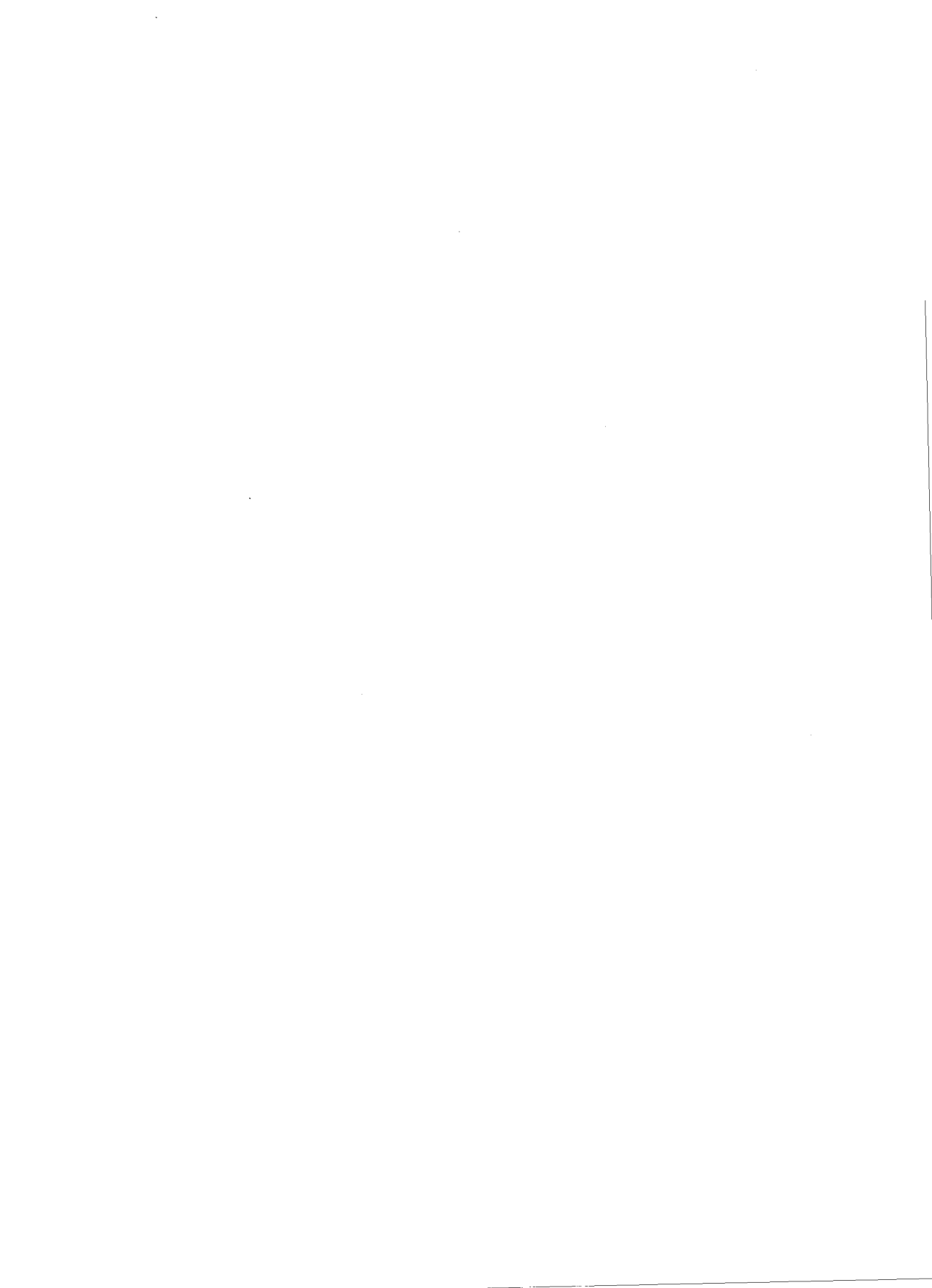
CONVEX

CONVEX COMPUTER CORPORATION



609

**CONVEX Computer Corporation**  
3000 Waterview Parkway  
P.O. Box 833851  
Richardson, TX 75083-3851  
United States of America  
(214)497-4000



---

# CONVEX FORTRAN Optimization Guide



---

Order No. DSW-034

Third Edition  
November 1991

CONVEX Press  
Richardson, Texas  
United States of America

---

# CONVEX FORTRAN Optimization Guide

Order No. DSW-034

Copyright ©1990, 1991 CONVEX Computer Corporation  
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OR ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.

COVUE is a trademark of CONVEX Computer Corporation. COVUE products consist of COVUEbatch, COVUEbinary, COVUEedt, COVUElib, COVUEnet, and COVUEshell.

CONVEX C Series architecture, C100, C200, C3, VECLIB, CXpa, and ASAP are trademarks of CONVEX Computer Corporation.

Cray is a registered trademark of Cray Research, Inc.

UNIX is a trademark of UNIX System Laboratories, Inc.

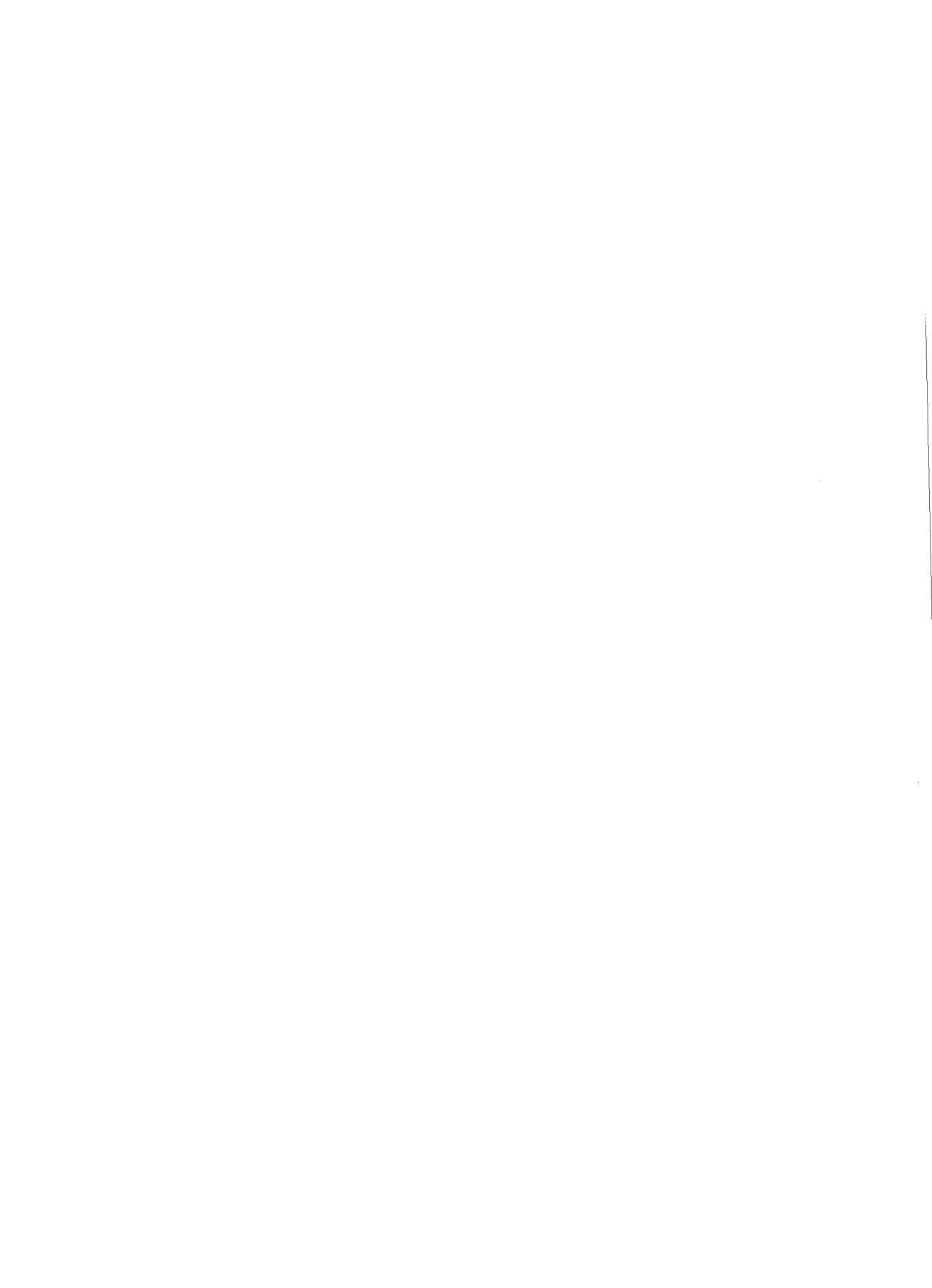
Printed in the United States of America

---

## Revision information for

# CONVEX FORTRAN Optimization Guide

Edition	Document No.	Description
Third	720-000930-203	<p data-bbox="542 538 1178 596">Released with CONVEX FORTRAN V7.0, November 1991. The following changes have occurred:</p> <ol data-bbox="542 617 1178 1049" style="list-style-type: none"><li data-bbox="542 617 1178 802">1. Added documentation on the NO_PEEL, NO_PROMOTE_TEST, PEEL, PEEL_ALL, PROMOTE_TEST, and PROMOTE_TEST_ALL optimization directives to Chapter 3, "Vector optimization," and Appendix B, "Compiler directives."</li><li data-bbox="542 820 1178 882">2. Added the section, "Test replacement," to Chapter 9, "Limits of optimization."</li><li data-bbox="542 899 1178 961">3. Added the section, "IF-DO optimizations," to Chapter 3, "Vector optimization."</li><li data-bbox="542 979 1178 1049">4. Added the section, "Short-circuit evaluation of conditionals," to Chapter 3, "Vector optimization."</li></ol>
Second	720-000930-201	Released with CONVEX FORTRAN V6.0, March 1990.
First	720-000930-200	May 1989. Initial release.



---

# Contents

---

<b>How to use this guide</b> .....	<b>xi</b>
Organization .....	xii
Notational conventions .....	xiii
Command syntax .....	xiii
General conventions .....	xiii
Associated documents .....	xiv
Ordering documentation .....	xv
Technical assistance .....	xv

---

<b>1 The basics</b> .....	<b>1</b>
Optimization options .....	1
Scalar optimization .....	2
Machine-dependent scalar optimization .....	2
Machine-independent scalar optimization .....	2
Vector optimization .....	3
Parallel optimization .....	3
Optimization tools .....	4

---

<b>2 Scalar optimization</b> .....	<b>5</b>
Optimizations performed at -no .....	6
Instruction scheduling .....	6
Span-dependent instructions .....	7
Register allocation .....	7
Tree-height reduction .....	7
Short-circuit evaluation of conditionals .....	9
Optimizations performed at -O0 .....	9
Instruction scheduling .....	10
Redundant-assignment elimination .....	10
Assignment substitution .....	11
Common-subexpression elimination .....	11
Redundant-use elimination .....	12
Constant propagation and folding .....	12
Algebraic and trigonometric simplification .....	13
Optimizations performed at -O1 .....	13

---

Constant propagation and folding .....	14
Redundant-assignment elimination .....	14
Dead-code elimination .....	16
Hoisting and sinking scalar and array references .....	16
Copy propagation .....	17
Common subexpression elimination .....	17
Code motion .....	19
Strength reduction .....	20
Strength reduction of induction variables and constants .....	21
<hr/>	
<b>3 Vector optimization .....</b>	<b>23</b>
Basic operation .....	23
Transformations the compiler performs .....	24
Strip mining .....	24
Loop distribution .....	25
Loop interchange .....	26
Paired hoist and sink .....	27
IF-DO optimizations .....	28
Redundant test elimination .....	28
Loop boundary-value peeling .....	29
Test promotion .....	30
Pattern matching .....	31
Conditional induction variables .....	32
Inhibitors of vectorization .....	32
Recurrence .....	33
Loop-carried dependency .....	33
Loop-independent dependency .....	35
Apparent recurrences .....	36
Reduction .....	37
Optimization report .....	38
<hr/>	
<b>4 Parallel optimization .....</b>	<b>43</b>
Basic operation .....	43
Inhibitors of parallelization .....	47
Loops with subroutine calls .....	47
Loop-carried dependency .....	49
Parallelizing code outside of loops .....	50
<hr/>	
<b>5 Optimizing FORTRAN applications.....</b>	<b>53</b>
Step 1. Compile the program .....	53
Step 2. Add scalar optimizations .....	54
Step 3. Add vectorization .....	55
Step 3a. Add vectorization in one step .....	55
Step 3b. Add selective vectorization .....	55

Step 4. Enhance vector optimization .....	56
Step 5. Add parallelization .....	57
Step 5a. Add parallelization in one step .....	58
Step 5b. Add selective parallelization .....	58
Step 6. Enhance parallel optimization .....	59
Step 7. Wrap up .....	59
<hr/>	
<b>6 Efficient programming constructs.....</b>	<b>61</b>
Data type in calculations .....	61
Writing efficient loops .....	62
Optimizing memory accesses .....	67
Memory interleaving .....	68
Multidimensional arrays .....	71
Partial word accesses .....	72
<hr/>	
<b>7 Manual optimization techniques .....</b>	<b>73</b>
Eliminate unnecessary strip mines .....	73
Do not vectorize loops with small trip counts .....	74
Promote an array .....	75
Remove a conditional from a loop .....	78
<hr/>	
<b>8 Inline substitution .....</b>	<b>81</b>
When to use inlining .....	81
How to use inlining .....	82
Creating .fil files .....	82
Using the -is option .....	83
Limits of inline substitution .....	84
<hr/>	
<b>9 Limits of optimization.....</b>	<b>85</b>
Incorrect results .....	85
Erroneous code .....	85
Hidden aliases .....	86
Invalid subscripts .....	89
Floating-point imprecision .....	89
Roundoff error .....	90
Vector reductions .....	90
Misused directives and options .....	90
Compiler limitations .....	92
Reductions .....	93
Evaluation order .....	94
Iterating by zero .....	94
Nondeterminism of parallel execution .....	95
Conditional vectorization .....	95
Test replacement .....	95

Slower code .....	97
Misused directives .....	97
Short vector length .....	98
Complicated conditionals .....	98
<hr/>	
<b>A The -uo option .....</b>	<b>99</b>
Simple strength reduction .....	99
Code motion .....	99
Conversion elimination .....	99
<hr/>	
<b>B Compiler directives .....</b>	<b>101</b>
ASSIGN_LOCK .....	
FREE_LOCK .....	103
BEGIN_ORDER .....	
END_ORDER .....	103
BEGIN_SECTION .....	
END_SECTION .....	104
BEGIN_TASKS .....	
NEXT_TASK .....	
END_TASKS .....	105
FORCE_PARALLEL .....	106
FORCE_PARALLEL_EXT .....	106
FORCE_VECTOR .....	107
MAX_TRIPS .....	107
NO_PARALLEL .....	108
NO_PEEEL .....	108
NO_PROMOTE_TEST .....	108
NO_RECURRENCE .....	108
NO_SIDE_EFFECTS .....	109
NO_VECTOR .....	110
PEEL .....	110
PEEL_ALL .....	110
PREFER_PARALLEL .....	110
PREFER_PARALLEL_EXT .....	110
PREFER_VECTOR .....	111
PROMOTE_TEST .....	111
PROMOTE_TEST_ALL .....	111
PSTRIP .....	111
ROW_WISE .....	112
SCALAR .....	113
SELECT .....	114
SYNCH_PARALLEL .....	114
UNROLL .....	115
VSTRIP .....	115

---

<b>C Vector operations.....</b>	<b>117</b>
Vector hardware .....	117
Vector accumulator register .....	117
Vector-length register .....	117
Vector-stride register .....	117
Vector-merge register .....	118
How the CONVEX architecture works .....	118
Vector instruction set .....	120
Vector load .....	120
Vector store .....	121
Binary vector operators .....	122
Vector reductions .....	123
Chaining .....	124
Vector comparisons .....	125
Vector operations under mask—C2 and C3 .....	125
Vector-merge register operations .....	127
Merge and mask .....	127
Compress .....	127
Expand .....	127
Examples .....	128
Examples of vector operations .....	129
Embedded if statement .....	129
Indirect array addressing .....	130

---

<b>Bibliography .....</b>	<b>133</b>
---------------------------	------------

---

<b>Glossary .....</b>	<b>135</b>
-----------------------	------------



---

# How to use this guide

This guide describes methods for optimizing FORTRAN programs. Background information and concepts presented in the first few chapters form a foundation for methods presented later in the book. Examples show the use of command-line options, compiler directives, and various tricks and tips to control and enhance scalar, vector, and parallel optimization.

Producing an efficient program requires efficient algorithms and efficient implementation. The techniques of writing an efficient algorithm are beyond the scope of this guide. The guide assumes you have chosen the best possible algorithm for your problem and helps you obtain the best possible performance from that algorithm.

The *CONVEX FORTRAN Optimization Guide* is for experienced FORTRAN programmers. Readers need not be familiar with the CONVEX implementation of scalar, vector, and parallel optimization. Although intended primarily for users of CONVEX FORTRAN, the methods described in this book have potential application to other FORTRAN compilers.

---

## Organization

This document consists of these chapters:

- Chapter 1 introduces CONVEX's approach to program optimization. Chapter 1 defines the terms and concepts you need to understand how the CONVEX FORTRAN compiler works with CONVEX C Series architectures.
- In Chapter 2, you learn the basics of scalar optimization and how the compiler transforms programs compiled for scalar optimization (command line options `-no`, `-O0`, and `-O1`).
- In Chapter 3, you learn the basics of vector optimization and how the compiler transforms programs compiled for vector optimization (command line option `-O2`).
- In Chapter 4, you learn the basics of parallel optimization and how the compiler transforms programs compiled for parallel optimization (command line option `-O3`).
- Chapter 5 presents a strategy for developing your programs to enhance optimization and provides you with examples of using compiler options and directives and their effects on optimization.
- Chapter 6 discusses programming constructs that can aid or hinder optimization.
- Chapter 7 presents some tricks and tips for optimizing your programs to run on CONVEX C Series supercomputers.
- Chapter 8 tells how and when to use inline substitution to enhance optimization.
- Chapter 9 discusses common optimization problems you can encounter and presents some possible solutions.
- Appendix A covers some details about using the `-uo` compiler option.
- Appendix B explains how to use CONVEX FORTRAN compiler directives.
- Appendix C describes vector operations on the assembly-language level and presents examples of some assembly-language instructions.

### Command syntax

Consider this example:

```
COMMAND input_file [...] {a | b} [output_file]
```

①            ②            ③            ④            ⑤

1. **COMMAND** must be typed as it appears.
2. *input\_file* indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets indicates that additional input file names may be supplied.
4. Either a or b must be supplied.
5. *output\_file* indicates an optional file name.

---

### General conventions

In general, the following conventions are used in this guide:

- *Italic*
  - Designates user-supplied variables in a command-line example
  - Introduces new and important terms
  - Identifies variables in mathematical equations
  - Indicates document titles
- Constant-width font designates input and output, including
  - Command names and options
  - System calls
  - Data structures and types
  - Directives, program statements, display examples, printout examples, and error messages returned
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipsis shows that lines of code have been left out of an example.

References to the FORTRAN man pages appear in the form intro(1F), where the name of the man page is followed by its section number enclosed in parentheses.

---

**Note**

---

A note highlights supplemental information.

---

**Caution**

---

A caution highlights procedures or information necessary to avoid damage to equipment, software, or data.

---

**Associated documents**

Using the CONVEX FORTRAN compiler successfully often requires information not described in this document. CONVEX Computer Corporation provides these documents to help you use the compiler.

- For more information about the compiler, see the *CONVEX FORTRAN Guide* (DSW-038), and *Release Notice, CONVEX FORTRAN Compiler V7.0*.
- For more information about CXpa, see the *CONVEX Performance Analyzer (CXpa) User's Guide* (DSW-251) and the *CONVEX Performance Analyzer (CXpa) Reference Manual* (DSW-254).
- For more information on *csd*, the source-level debugger, see the *CONVEX Consultant User's Guide* (DSW-025).
- For more information on parallel programming in assembly language, see the *CONVEX Compiler Utilities User's Guide* (DSW-096) and the *CONVEX C Series Architecture Reference Manual* (DHW-300).
- For information on CONVEX's interprocedural optimization compiler, see the *CONVEX Application Compiler User's Guide* (DSW-401).

---

## Ordering documentation

To order this document or any other CONVEX document, send requests to

CONVEX Computer Corporation  
Customer Service  
P.O. Box 833851  
Richardson, Texas 75083-3851 U.S.A.

Order documents by title, requesting the most recent edition. In some situations, you may not want the current edition. To receive a specific edition of a manual, contact the local CONVEX office or call the Technical Assistance Center (TAC).

---

## Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC):

- Within the continental U.S., use (800)952-0379.
- Outside continental U.S., contact your local CONVEX office.



Optimization improves the performance of programs. To optimize programs, the CONVEX FORTRAN compiler performs these functions:

- Eliminates unnecessary operations
- Arranges operations in the most efficient order
- Replaces slow operations with faster equivalents
- Takes full advantage of CONVEX architectures

---

## Optimization options

The CONVEX FORTRAN compiler offers five optimization options, which are specified on the `fc` command line. The compiler transforms code according to the optimization option you specify. These transformations are cumulative: each higher-level option retains the transformations of the previous option. The optimization options are summarized below.

---

Option	Description
-no	Machine-dependent scalar optimization. This option is the default.
-O0	Basic block machine-independent scalar optimization
-O1	Basic block and program unit machine-independent scalar optimization
-O2	Vector optimization
-O3	Parallel optimization

---

---

## Scalar optimization

A scalar value is a single value or entity. A scalar instruction operates on one or a pair of scalar values. There are two types of scalar optimization: machine-dependent and machine-independent.

---

### Machine-dependent scalar optimization

At the lowest option ( -no ), the compiler does machine-dependent scalar optimization, which fully exploits the machine's scalar functional units and registers. Because machine-dependent scalar optimization works at the machine-instruction level, you cannot disable it.

---

### Machine-independent scalar optimization

While machine-dependent scalar optimization works at the machine-instruction level, machine-independent scalar optimization works at two levels:

- Local (basic-block) level
- Global (program-unit) level

A basic block is a sequence of statements ending with a conditional or unconditional branch. Branches do not exist within the body of a basic block. At level -O0, optimization is local to a basic block. The compiler does machine-independent optimizations within the scope of a basic block.

A program unit is a subroutine, function, or main section. At level -O1, optimization is local to a program unit and is global with respect to basic blocks. This means that the compiler does machine-independent optimizations across multiple blocks in a program unit at one time.

To improve performance, machine-independent optimizations

- Reduce the number of times memory is accessed
- Simplify expressions
- Eliminate redundant operations
- Replace variables with constants
- Replace slow operations with faster equivalents

---

## Vector optimization

Vector optimization, or vectorization, typically improves the performance of programs that manipulate arrays. For example, suppose you write a loop to add the corresponding elements of two arrays. With vector optimization, the CPU can add up to 128 elements of each array with a single instruction.

The compiler also transforms many loops that it cannot vectorize into loops that it can vectorize. This increases the number of loops that the compiler can optimize, which minimizes execution time dramatically.

The `-O2` option allows vector optimization. It also performs scalar optimization on loops that it cannot vectorize and on loops that are not profitable to vectorize.

---

## Parallel optimization

Parallel optimization reduces time to solution by spreading work across multiple CPUs.

The actual savings you can achieve with parallel optimization depend on the application, the load on the system when the application is run, and how well suited your algorithm is to parallel optimization. At best, parallelization can improve time to solution by a factor of  $N$ , where  $N$  is the number of CPUs on your system. Limitations imposed by algorithms prevent some programs from realizing all of this theoretical improvement.

Every program has at least one *thread* or sequence of instructions that can execute on a single CPU. Parallel programs have more than one thread. On CONVEX C2 and C3 Series computers, threads can execute on multiple CPUs, which are allocated by the *Automatic Self-Allocating Processors (ASAP)* mechanism. ASAP is a way of getting the most work from multiple CPUs, which gives you the benefits of multiprocessing and parallel processing.

The compiler divides a job into tasks that the processors execute as efficiently as possible, using ASAP technology. The compiler does the first step, which is to look for regions of code it can parallelize. The compiler then generates an instruction that causes a request to be posted in a set of registers called communication registers. During execution, idle CPUs check the communication registers for requests. If a CPU finds a request, it begins executing that thread of parallel code. At this point, two or more CPUs are working on different threads of the same job.

When you specify `-O3` on the `fc` command line, the compiler performs parallel and vector optimization. It also performs scalar optimization on loops that it cannot parallelize or vectorize.

With the `-O3` option, the compiler automatically performs parallel and vector optimization at the loop level. The compiler divides loop iterations into separate threads and generates code that is independent of the number of available CPUs.

To parallelize constructs other than loops, you can use tasking directives. For more information about tasking directives, see the section, "Parallelizing code outside loops," in Chapter 4, and Appendix B, "Compiler directives."

---

## Optimization tools

Part of the CONVEX Consultant package, the `csd` source-level debugger can set process breakpoints, examine machine registers, and display traces of the stack. For more information on using `csd`, see the *CONVEX Consultant User's Guide*.

The CONVEX Performance Analyzer, `CXpa`, is a tool for examining your program's performance at routine, loop, and basic-block level. You can use `CXpa` or one of the profilers in the CONVEX Consultant to track the effects of optimizations. For more information on how to use `CXpa`, see the *CONVEX Performance Analyzer User's Guide*.

This chapter describes how the compiler transforms code compiled for scalar optimization. The compiler optimizes scalar code automatically, so there is no need to rewrite code to achieve the gains described here.

A scalar value is one value or entity. A scalar instruction operates on one or a pair of scalar values, as in the FORTRAN statement

$$\text{SCALAR1} = \text{SCALAR2} + \text{SCALAR3}$$

The CONVEX FORTRAN compiler performs two types of optimizations on scalar instructions:

- Machine-dependent
- Machine-independent

At optimization level `-no`, the compiler performs machine-dependent scalar optimizations, which occur at the machine-instruction level. You cannot disable this optimization. At optimization level `-O0`, the compiler performs machine-dependent and machine-independent optimizations. The compiler optimizes one basic block at a time at this level. At level `-O1`, the compiler optimizes multiple basic blocks within a program unit.

---

## Note

---

You can identify basic blocks in the final, optimized assembly code by looking for jump statements and labels in the assembly-language listings produced by the compiler's `-s` option. At optimization level `-no`, there is a one-to-one correspondence between these basic blocks and the statements in the original FORTRAN code. At higher optimization levels, this one-to-one correspondence may not exist. If a basic block is dead code, such as an unreachable alternative in an `IF` statement, the compiler can eliminate the basic block at higher optimization levels. The number of basic blocks in the assembly-language output (or output of block-level profilers `bprof` and `CXpa`) typically decreases as the optimization level is increased.

---

## Optimizations performed at -no

At optimization level -no, the compiler performs machine-dependent optimizations only. These optimizations take place at the machine-instruction level. They create object code that fully uses the scalar features of the CONVEX architecture.

---

### Instruction scheduling

Instruction scheduling rearranges machine instructions to use the computer's functional units most effectively. Each CPU on a CONVEX supercomputer has multiple functional units on which operations execute simultaneously. On a CONVEX C Series processor, operations such as add, multiply, and store execute simultaneously on separate functional units.

At optimization level -no, the compiler rearranges instructions derived from a single FORTRAN source statement to maximize use of the functional units. Compare the equivalent assembly pseudocodes for the typical FORTRAN source statement shown below.

**FORTRAN Source:**  $A = (B + C * D) / E - F$

Original Code	Optimized Code
ld.w D, s0	ld.w D, s0
ld.w C, s1	ld.w C, s1
mul.s s1,s0	ld.w B, s2
ld.w B, s1	mul.s s0,s1
add.s s1,s0	ld.w E, s0
ld.w E, s1	ld.w F, s3
div.s s1,s0	add.s s1,s2
ld.w F, s1	div.s s0,s2
sub.s s1,s0	sub.s s3,s2
st.w s0, A	st.w s2, A

In the original code, operations execute one at a time. In the optimized code, groups of registers used by different functional units are loaded. All registers are loaded before arithmetic operation begins, if possible. Operations, such as multiply and load, that use different functional units also execute simultaneously.

Concurrent execution of machine instructions on multiple functional units, within a single CPU, is distinct from parallel processing, which occurs on multiple CPUs.

For more information on functional units, see the *CONVEX C Series Architecture Reference Manual*.

---

## Span-dependent instructions

When possible, the compiler generates short-form instructions for conditional and unconditional jumps and branches. Short-form instructions, which are two bytes long, are generated when the span between the jump or branch instruction and its target is within defined limits for these instructions. Short-form instructions conserve memory and increase execution speed.

For more information on jump and branch instructions, see the *CONVEX C Series Architecture Reference Manual* and *CONVEX Compiler Utilities User's Guide*.

---

## Register allocation

CONVEX FORTRAN uses a technique for allocating registers that fully exploits the CONVEX register set. This allows grouping of register loads, concurrent execution of instructions (*pipelining*), and reduces register conflicts.

---

## Tree-height reduction

The compiler represents expressions internally as trees. These trees are optimized by *tree-height reduction* or *balancing*. For example, consider this real expression:

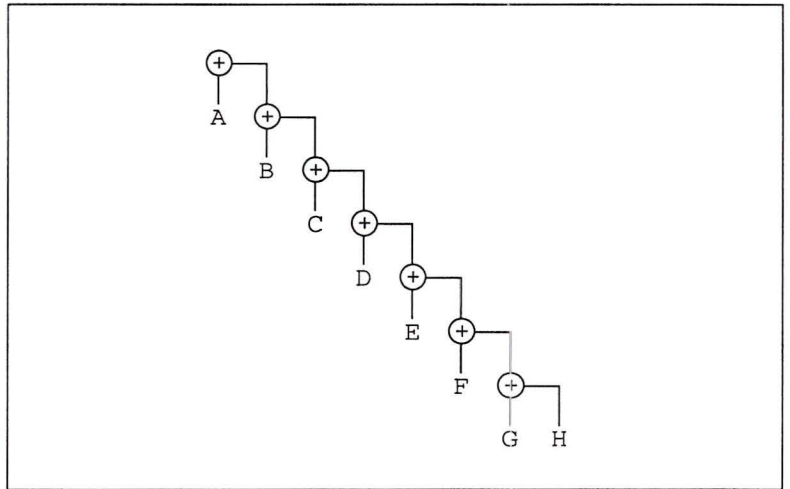
$$A + B + C + D + E + F + G + H$$

The expression can be evaluated as follows:

$$(A + (B + (C + (D + (E + (F + (G + H)))))))$$

(G+H) is evaluated first. No two additions can be carried out simultaneously because each addition depends on the result of the addition to the right. Figure 1 shows how the compiler represents this order internally.

**Figure 1**  
Unbalanced tree  
representation

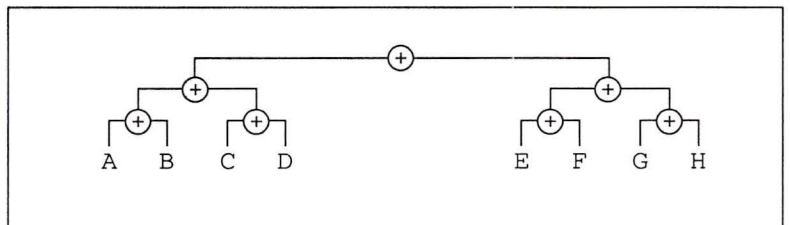


Another way to evaluate the expression is

$$(((A + B) + (C + D)) + ((E + F) + (G + H)))$$

Because none of the four additions in the innermost parentheses requires the result of another addition, the additions can be done simultaneously on several functional units.  $((A+B) + (C+D))$  and  $((E+F) + (G+H))$  are then evaluated. The compiler represents this order internally as a balanced tree, as shown in Figure 2.

**Figure 2**  
Balanced tree  
representation



In Figure 1, the depth of the tree is seven; in Figure 2, the depth of the tree is three. The machine instructions generated for the tree in Figure 1 execute slower than the instructions generated for the tree in Figure 2.

The deeper the tree representing the expression, the more time is required to evaluate the expression. The compiler chooses an evaluation order that minimizes the depth of the expression and maximizes instruction pipelining. Of course, the compiler preserves all execution-order rules as specified in the ANSI standard. Because the compiler chooses evaluation order to ensure the most efficient execution, you can write expressions in any order.

If your application depends on a specific order of evaluation, you must use parentheses to specify that order.

---

## Short-circuit evaluation of conditionals

Short-circuiting the evaluation of conditionals increases the efficiency of IF statements by skipping irrelevant tests when logical operators are involved in the conditional. CONVEX FORTRAN short-circuits evaluation of IF statements that contain .AND. and .OR. operators which have logical operands and are used in a logical context. Take, for example, the following IF statement:

```
IF ((A .EQ. B) .OR. F(G)) THEN
```

If (A .EQ. B) evaluates to true, the evaluation of F(G) is skipped, and the THEN portion of the statement is evaluated.

Similarly, given the code

```
IF ((A .EQ. B) .AND. F(G)) THEN
```

if (A .EQ. B) evaluates to false, the evaluation of F(G) and the THEN portion of the statement is skipped.

Short-circuit evaluation works with all types of IF statements (arithmetic, logical, and block). Performing arithmetic (+, -, \*, /) on a logical expression disables short circuiting within that expression. Logical-valued expressions used as arguments to function calls within an IF statement's conditional expression are not short circuited. Note that the binary operators .EQ., .NE., .LT., .LE., .GT., and .GE. always produce a logical result.

The compiler short-circuits the evaluation of conditionals by default. You can disable short-circuiting by specifying the -nosc flag on the compiler command line.

---

## Optimizations performed at -O0

At optimization level -O0, the compiler performs machine-independent scalar optimizations within a basic block. The compiler continues to perform the machine-dependent optimizations performed at -no.

---

## Instruction scheduling

At optimization level -O0 and above, instructions from *multiple* statements, as well as those from single statements, are scheduled as a group. To see how this works, compare the assembly code for the two FORTRAN statements as shown below.

**FORTRAN source:**  $T = B + C * D$   
 $A = (B + C * D) / E - F$

Original code	Optimized code
ld.w D, s0	ld.w D, s0
ld.w C, s1	ld.w C, s1
ld.w B, s2	ld.w B, s2
mul.s s0,s1	mul.s s0,s1
add.s s1,s2	ld.w E, s0
st.w s2, T	ld.w F, s3
	add.s s1,s2
ld.w D, s0	st.w s2, T
ld.w C, s1	div.s s0,s2
ld.w B, s2	sub.s s3,s2
mul.s s0,s1	st.w s2, A
ld.w E, s0	
ld.w F, s3	
add.s s1,s2	
div.s s0,s2	
sub.s s3,s2	
st.w s2, A	

In the original code, which was generated at -no, instructions from each statement are scheduled independently. Instructions generated from the first statement execute first, followed by instructions generated from the second statement.

In the optimized code, instructions from the two statements are scheduled together, as if derived from a single statement. Instructions are generated and scheduled in an order that optimizes performance.

---

## Redundant-assignment elimination

Redundant-assignment elimination removes unnecessary assignments to a variable. When a variable is not used between two assignments, the first assignment is eliminated. The code in the following example contains a redundant assignment,  $X=Y+C$ , which the compiler removes.

Original code	Optimized code
X = Y + C	!(statement eliminated)
!(X not used)	...
X = 3.1416	X = 3.1416
...	...
Y = (X + 7) * 2.15	Y = (X + 7) * 2.15

## Assignment substitution

Assignment substitution eliminates redundant loads. The compiler “remembers” the value assigned to a variable and replaces subsequent references to that variable with the assigned value. An example appears below.

Original code	Optimized code
X = Y + C	REG = Y + C
X = X * 4.4	REG = REG * 4.4
T = X * B + 12.4	T = REG * B + 12.4
X = 4.179	X = 4.179

After the machine instructions for the first statement execute, the value of  $Y+C$  remains in a register. The compiler replaces subsequent references to  $X$  with references to this register until the value of  $X$  changes or until the end of the basic block is reached. This optimization eliminates repeated loading and storing of  $X$  into a register, which increases performance and provides opportunities for further optimization. In this example, assignment substitution makes the first assignment to  $X$  redundant, so the compiler eliminates the assignment.

Because the compiler substitutes assignments, you rarely need to optimize a program by replacing a variable reference with a constant in the source code.

## Common-subexpression elimination

The compiler recognizes subexpressions that repeat within a basic block. The compiler retains the value of the subexpression in a register, which eliminates redundant computations and register loads. For example, the compiler recognizes  $B+C$  as a common subexpression of  $A+B+C+D$  and  $B+E+C$ , and calculates the subexpression only once.

The compiler also eliminates redundant array address calculations. As with assignment substitution, you do not need to manually create a temporary variable in which to store the value of a common subexpression. The compiler performs that function automatically.

---

## Redundant-use elimination

This optimization is a special case of common subexpression elimination where the subexpression is a variable. The compiler detects multiple references to a variable between assignments and retains the value of the variable in a register. This action helps eliminate redundant register loads.

---

## Constant propagation and folding

After assigning a constant to a variable, the compiler replaces subsequent references to the variable with the constant. For example, if you write  $X=5$ , the compiler replaces  $X$  with 5 within that basic block or until a new value is assigned to the variable. This is known as *constant propagation*, which is a form of assignment substitution.

An example of constant propagation and folding appears below.

Original code	Optimized code
$I = 5$	$I = 5$
$J = 0$	!(assignment eliminated)
...	...
$J = J + 2$	$J = 2$
...	...
$K = K + I * J$	$K + K + 10$

The compiler also replaces operations on constants with the result of the operation. This is known as *constant folding*. For example, it replaces  $Y=5+7$  with  $Y=12$ . It then propagates the constant value to replace future references to  $Y$  within the basic block. The compiler also propagates and folds values assigned to names in `PARAMETER` statements.

The compiler folds the most frequently used intrinsics when they are applied to constant arguments. For example, `SIN(0.0)` becomes 0.0. The compiler also folds exponentiation involving constants. For example,  $3**3$  becomes 27.

The compiler type-converts constants, if necessary, before propagating and folding them. If a program contains the expression  $X=1$ , where  $X$  is `REAL`, the compiler converts 1 to 1.0 before propagating it.

If an integer overflow occurs as a result of constant folding, the compiler reports "Integer constant truncation." If a floating-point overflow occurs, the compiler reports "Real constant either too large or too small." Floating-point under-flow always results in zero. If any of these messages or conditions occur, eliminate the offending operation or bring the value of the constant within acceptable bounds.

---

## Algebraic and trigonometric simplification

The compiler simplifies algebraic and trigonometric expressions, as shown below.

Original expression	Optimized expression
$X + 0$	$X$
$X * 1$	$X$
$X * 0$	$0$
$K \text{ .AND. } -1$	$K$
$K \text{ .AND. } 0$	$0$
$K \text{ .OR. } -1$	$-1$
$K \text{ .OR. } 0$	$K$
$-1 * X$	$-X$
$X - X$	$0$
$X / -1$	$-X$
$(-1) ** K$	$1 - ((K \text{ .AND. } 1) * 2)$
$X ** 0.5$	$\text{SQRT}(X)$
$X ** 0$	$1$
$1 ** X$	$1$
$X / X$	$1$
$0 - X$	$-X$
$0 / X$	$0$
$\text{SIN}(X) * \text{COS}(X)$	$0.5 * \text{SIN}(2X)$
$\text{SIN}(X) / \text{COS}(X)$	$\text{TAN}(X)$

The compiler performs obvious variations of these operations for the commutative operators. For example, the compiler converts  $X + (0 + Y)$  to  $X + Y$ .

---

## Optimizations performed at -O1

Global optimization is done across a group of basic blocks but within a single program unit or subroutine. The -O1 option performs global, basic-block, and machine-dependent optimizations.

---

## Constant propagation and folding

Propagating and folding constants at the global level is analogous to performing the same operations at the basic-block level. The scope of the optimization is now a function, subroutine, or program main section.

An example of constant propagation and folding appears below.

Original code	Optimized code
INTEGER A,B,C	INTEGER A,B,C
A = 5	A = 5
B = 15	B = 15
READ *, I	READ *, I
IF (I) 10,10,15	IF (I) 10,10,15
10 A = 6	10 A = 6
C = A	C = 6 !A=6
GOTO 20	GOTO 20
15 C = A + B	15 C = 20 !A=5,B=15
GOTO 25	GOTO 25
20 B = A + C	20 B = 12 !A=6,C=6
GOTO 30	GOTO 30
25 B = A + B + C	25 B = 40 !A=5,B=15,C=20
30 PRINT *,A,B,C	30 PRINT *,A,B,C
END	END

The compiler propagates and folds constants globally at optimization level -O1 and higher, which eliminates the need to propagate constants by hand in programs compiled at these levels.

---

## Redundant-assignment elimination

At optimization level -O1, the compiler eliminates assignments to variables that do not have subsequent references within the program unit. The following example shows how the compiler eliminates redundant assignments to the variable A.

### Original code

```
SUBROUTINE FOO
C A is local
...
X = Y * Z
IF (A .GT. 0) THEN
...
    A = X * Y + 3.1416

ELSE
...
    X = (X + 7) * Z + 3.1416
ENDIF
...
Comment: A is not used later in this routine
END
```

As shown in the optimized code below, the compiler does not eliminate ASSIGN statements and assignments to dummy arguments, function names, and common variables.

### Optimized code

```
SUBROUTINE FOO
...
X = Y * Z
IF (A .GT. 0) THEN
...
ELSE
...
    X = (X + 7) * Z + 3.1416
ENDIF
...
END
```

If the right side of a redundant assignment statement contains a function or subroutine call, the compiler eliminates the assignment and retains the call, as in the following example.

Original code	Optimized code
SUBROUTINE FOO	SUBROUTINE FOO
...	...
I = INTFUN(X)	<NULL> = INTFUN(X)
...	...
Comment: I not used	

If a function or subroutine has no side effects, the compiler eliminates the function or subroutine call, as well as the assignment, saving much more time. Functions and subroutines that do not modify the value of an argument or common variable, perform input and output, or call another function or subroutine have no side effects.

Existing compilers cannot automatically determine whether a side effect exists and eliminate function or subroutine calls only if you explicitly request it with the `NO_SIDE_EFFECTS` directive.

The form of this directive is

```
C$DIR NO_SIDE_EFFECTS (func_list)
```

where *func\_list* is a list of function and subroutine names separated by commas. The directive must precede the function or subroutine call that does not contain side effects.

---

## Caution

---

Do not use the `NO_SIDE_EFFECTS` directive on a call to a function or subroutine that

- Changes the value of an argument
- Changes the value of a `COMMON` variable
- Performs input or output
- Calls another function or subroutine that performs one of these operations

For more information about the `NO_SIDE_EFFECTS` directive, see Appendix B, “Compiler directives.”

---

## Dead-code elimination

If, as a result of constant propagation and folding, the compiler can fold an arithmetic or logical expression in an `IF` statement to `.TRUE.` or `.FALSE.`, the compiler eliminates the unreachable code.

---

## Hoisting and sinking scalar and array references

The compiler can *hoist* some scalar or array references out of a loop. Hoisting moves an operation from within a loop to a basic block preceding the loop. *Sinking* moves a store operation from a loop to a basic block succeeding the loop. Hoisting and sinking eliminate redundant loads and stores by moving a reference to a location where it is executed only once instead of many times. Hoisting can occur with or without sinking, but sinking never occurs without hoisting.

Hoisting occurs without sinking in the following cases:

- At optimization level -O1, when the value of a scalar variable or array reference is unchanged within the loop
- At optimization level -O2, if the array is indexed only by loop constants and the loop-control variable

Hoisting and sinking can be applied together

- At optimization level -O1, to a scalar variable that can be kept in a scalar register during the loop's execution
- At optimization level -O2, to a section of an array that can be kept in a vector register during the loop's execution

---

## Copy propagation

The compiler can replace a variable with another variable to which it has been equated. This is called *copy propagation*. For example, after evaluating the statement  $X=Y$ , the compiler replaces later occurrences of  $X$  with  $Y$ .

In the following example, if the compiler determines that  $X$  and  $Y$  are unchanged between the assignment and the reference, it replaces  $X$  with  $Y$ .

```
X = Y
...
W = Z - X
```

becomes

```
...
W = Z - Y
```

---

## Common subexpression elimination

The compiler eliminates common subexpressions at the global level. The compiler retains the value of the common subexpression in a register if one is available; otherwise, the compiler assigns the value to a temporary variable. The compiler then replaces subsequent occurrences of the common subexpression with references to the register or temporary variable.

The code in the following example contains a common subexpression that can be eliminated.

### Original code

```
SUBROUTINE GCSE1
...
A = B + C / (-J * B + SQRT(C))
IF (K .LT. 1) THEN
    L = 5
ENDIF
F = E - C / (-(J * B) + SQRT(C))
...
END
```

The compiler recognizes that a common subexpression,  $C / (-J * B + \text{SQRT}(C))$ , is used before and after the IF statement. The compiler saves the value of the subexpression in the temporary variable T1 before the IF statement and uses this variable later to compute the value of F, as shown below.

### Optimized code

```
SUBROUTINE GCSE1
...
T1 = C / (-J * B + SQRT(C))
A = B + T1
IF (K .LT. 1) THEN
    L = 5
ENDIF
F = E - T1
...
END
```

In the following example, the compiler determines that the subexpression must be calculated whether the condition associated with the IF statement evaluates to .TRUE. or .FALSE.

### Original code

```
SUBROUTINE GCSE2
...
IF (K .LT. L) THEN
    A = (C * 4) / -(J * B + SQRT(C))
ELSE
    E = (E * 4) / -(J * B + SQRT(C))
ENDIF
F = (B * 4) / -(J * B + SQRT(C))
...
END
```

The compiler saves the value of the common subexpression in the temporary variable T1 and uses the variable to compute the value for assignment to A, E, and F, as follows.

### Optimized code

```
SUBROUTINE GCSE2
...
T1 = -(J * B + SQRT(C))
IF (K .LT. L) THEN
  A = (C * 4) / T1
ELSE
  E = (E * 4) / T1
ENDIF
F = (B * 4) / T1
...
END
```

---

### Code motion

Code motion is the movement of invariant expressions out of loops. An invariant expression yields the same result on every iteration of a loop.

In the following example, all variables used in the assignment to A remain invariant within the loop. The compiler recognizes this and moves the calculations and assignments out of the loop, performing these costly calculations only once.

### Original code

```
SUBROUTINE GCM
REAL AR(10)
...
DO I = 1, 10
  A = C / (-(E * B) + SQRT(C))
  AR(I) = A + B * C
ENDDO
...
END
```

At higher optimization levels, the compiler can vectorize the loop

### Optimized code

```
SUBROUTINE GCM
REAL AR(10)
...
A = C / ( - (E * B) + SQRT(C) )
T1 = A + B * C
DO I = 1, 10
    AR(I) = T1
ENDDO
...
END
```

If an invariant expression does not lie on a path to all loop exits, the compiler does not move the invariant expression unless you use the `-uo` (unsafe optimizations) compiler option. For more information about using the `-uo` option, see Appendix A, "The `-uo` option."

---

## Strength reduction

In some cases, the compiler can replace an arithmetic operation with an equivalent operation that executes more quickly. Such replacements are called strength reductions. On the C Series machines, for example, the compiler transforms integer multiplication by 2, 4, 8, and 16 into integer shifts:

$J * 2$  becomes `IISHFT(J, 1)`

$J * 4$  becomes `IISHFT(J, 2)`

The strength of integer divisions is not reduced with integer shifting because the CONVEX architecture provides a logical shift instruction, but not an arithmetic shift instruction. (Logical shifts do not sign-extend.)

$A / 2$  remains  $A / 2$

Multiplication involving integer constants is reduced to addition:

$X * 2$  becomes  $X + X$

When the `-uo` (unsafe optimizations) command line option is specified, division by a constant is reduced to multiplication:

$X / C$  becomes  $D * X$  where  $D = 1 / C$

Because  $C$  is a constant,  $D$  also is a constant, which can be computed at compile time.

---

## Strength reduction of induction variables and constants

The compiler can reduce the strength of operations to optimize loop induction variables and loop constants. Multiplications within a loop that calculate the address of a subscripted variable are often candidates for strength reduction.

The compiler does not reduce operations that only involve REAL variables. Because floating-point arithmetic is imprecise, reduced operations do not always yield equivalent results. If an expression does not lie on a path to all loop exits, the compiler does not reduce the expression unless you use the `-uo` option.

In the following example, the compiler recognizes that `I` is incremented by 2 on each iteration and that `X` is incremented by  $2 * C$ , a loop constant.

### Original code

```
SUBROUTINE GSR
  I = 1 !induction var
10  X = I * C !loop induction value
    ...
    I = I + 2
    IF(I .LE. 100) GOTO 10
    ...
  END
```

As shown below, the compiler produces code that calculates  $2 * C$  only once and increments `X` by the value saved in `T2` instead of calculating `I * C` on every iteration.

### Optimized code

```
SUBROUTINE GSR
  I = 1
  T1 = C
  T2 = 2 * C
10  X = T1
    ...
    T1 = T1 + T2
    I = I + 2
    IF(I .LE. 100) GOTO 10
    ...
  END
```



Appropriate use of vector instructions is the key to high performance on CONVEX C Series architectures. Vectorization converts scalar operations in loops on array elements into equivalent vector operations. The `-O2` compiler option instructs the compiler to vectorize loops in a program. For loops that cannot be vectorized, the compiler carries out the global transformations performed at `-O1`.

Vector operations use vector registers to perform operations on up to 128 pairs of array elements with a single machine instruction. For vector operations on arrays longer than 128 elements, the compiler partitions the operation into groups of no more than 128 elements. This is called *strip mining*.

---

## Basic operation

Loops typically perform repetitive operations on multiple elements of arrays. The following loop involves at least 700 instruction executions: load an element of B and an element of C; add them, and store the result in the corresponding element of A; load, increment, and store I; and repeat for each of the next 99 elements.

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

At optimization level `-O2`, the compiler generates vector code to load 100 elements of B and 100 elements of C into vector registers, add them simultaneously, and store the 100 resulting elements in A.

Think of the vector code as a pseudocode statement involving only four instructions:

```
A(1:100) = B(1:100) + C(1:100)
```

where `A(1:100)` means `A(1)` through `A(100)`.

---

## Transformations the compiler performs

The compiler reorders the statements and instructions of a program to make the program easier to vectorize. The following subsections explain the most important of these transformations.

---

### Strip mining

The vector registers hold up to 128 elements. When the iterations of a vectorizable loop are unknown or exceed 128, the compiler strip mines the loop before vectorizing it. Strip mining replaces the original loop with two loops. The inner loop has an iteration count that never exceeds 128. The outer loop controls the number of times the inner loop is executed. The following example shows a loop that can be strip mined.

#### Original loop

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

In the vectorized loop code shown below, IOU<sub>T</sub> is a variable that the compiler uses to count the number of elements remaining to be processed, and the vector operations are shown using the section notation described above. I is the starting index for each vector operation.

#### Vectorized loop

```
I = 1
DO IOUT = N, 0, -128
  K = I + MIN(127, IOUT - 1)
  A(I:K:1) = B(I:K:1) + C(I:K:1)
  I = I + 128
ENDDO
```

If N equals 300, IOU<sub>T</sub> is tested four times. For each comparison of IOU<sub>T</sub> to zero, the table below shows values of I and IOU<sub>T</sub> and the elements of array A that are calculated.

---

I	IOU <sub>T</sub>	Elements processed
1	300	1...128
129	172	129...256
257	44	257...300
385	-84	

---

The fourth test of IOUT fails, so the loop is not executed and no elements of A are processed.

---

## Loop distribution

Vectorization is only done on simple loop nests. A simple loop nest is one in which all calculations are done in the innermost loop. Nested loops, however, can be vectorized by distributing the outermost loop and vectorizing each of the resulting loops or loop nests. Consider the loop in the following example.

### Original loop

```
DO I = 1, N
  B(I, 1) = 0
  DO J = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
  ENDDO
  D(I) = E(I) + A(I)
ENDDO
```

Three copies of the I loop are created, separating the nested J loop from the assignments to arrays B and D. In this way, all three assignments become vector operations, as shown below.

### Vectorized loop

```
DO I = 1, N
  B(I, 1) = 0
ENDDO
DO I = 1, N
  DO J = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
  ENDDO
ENDDO
DO I = 1, N
  D(I) = E(I) + A(I)
ENDDO
```

---

## Loop interchange

The compiler interchanges nested loops for the following reasons:

- To make the loop that is the most profitable to vectorize the innermost loop
- To make the loop that is the most profitable to parallelize the outermost loop
- To make memory accesses to consecutive words in memory
- To bring a loop with long vector length (iteration count) inside a loop with short vector length

For vectorization, profitability is the improvement in execution time.

Consider the matrix addition shown below.

### Original loop

```
DO I = 1, N
  DO J = 1, M
    A(I, J) = B(I, J) + C(I, J)
  ENDDO
ENDDO
```

To vectorize the original loop, the compiler interchanges the I and J loops so that contiguous elements of B and C are loaded into vector registers. This optimization, shown in the following example, substantially improves performance over the row-by-row approach of the source code.

### Vectorized loop

```
DO J = 1, M
  DO I = 1, N
    A(I, J) = B(I, J) + C(I, J)
  ENDDO
ENDDO
```

---

## Paired hoist and sink

A vector register can sometimes be used as an accumulator, making it possible for the compiler to move loads and stores of the register outside the vector loop. As noted in Chapter 2, hoisting is the movement of an operation (such as loading a register) out of a loop to a basic block preceding the loop. Sinking is the complement of hoisting. The compiler moves an operation, such as a register store, out of a loop to a basic block following the loop. The following example shows a loop nest that is a candidate for hoisting and sinking.

```
DO I = 1, N
  DO J = 1, N
    A(I) = A(I) * B(I, J)
  ENDDO
ENDDO
```

When this program fragment is compiled at optimization level -O2, the I loop is interchanged to innermost and is vectorized. Additionally, the load of vector A is hoisted above the loop and the store of vector A is sunk below the loop. This optimization eliminates the need for repeated vector loads and stores and makes the loop even faster.

The following code shows an example of vector hoisting and sinking.

```
I = 1
DO IOUT = N, 0, -128
  K = I + MIN(127, IOUT - 1)
  V0 = A(I:K:1)
  DO J = 1, N
    V0 = V0 + B(I:K:1, J)
  ENDDO
  A(I:K:1) = V0
  I = I + 128
ENDDO
```

Vector loads and stores are hoisted and sunk only under these conditions:

- The array reference and array assignment have the same subscripts.
- All subscripts of the array are the induction variable of a vectorized loop or loop constants.

The compiler sometimes interchanges loops to make a subscript a loop constant so that sinking and hoisting is possible.

---

## IF - DO optimizations

IF-DO optimizations modify loops containing tests to improve vector performance. Tests can be promoted out of the loops or eliminated completely. By minimizing the number of tests within a loop, the compiler reduces the number of masked vector instructions that must be executed, thereby improving performance.

There are three types of IF-DO optimizations: redundant test elimination, loop peeling and test promotion. Each of these is described in detail below.

### Redundant-test elimination

Redundant-test elimination is the simplest of the IF-DO optimizations. The compiler recognizes when a test against some index variable is evaluated more than once and eliminates that test as well as any accompanying redundant code.

This optimization is especially relevant when you are optimizing FORTRAN 66 programs that contain DO loops surrounded by IF tests, as shown in the following example.

#### Original loop

```
DO I = 1, N
  IF (I .GT. 0) THEN
    DO J = 1, I
      A(I,J) = 0
    ENDDO
  ENDF
ENDDO
```

#### Optimized loop

```
DO I = 1, N
  DO J = 1, I
    A(I,J) = 0
  ENDDO
ENDDO
```

Here the explicit test IF (I .GT. 0) is redundant, since the test is implicit in the DO loop. It is therefore removed during redundant-test elimination.

Redundant-test elimination is always performed at optimization levels -O2 and above.

## Loop boundary-value peeling

Loop boundary-value peeling involves removing the first iteration, last iteration, or first and last iterations of a loop to remove conditional tests from the loop. This is done when the loop contains a test involving an explicit reference to the loop index variable that always evaluates to `.TRUE.` or `.FALSE.` for the first iteration, last iteration, or first and last iterations.

With the code shown below, the compiler automatically peels off the first and last tests and rewrites the loop to cover the remaining indexes.

### Original loop

```
DO I = 1, 100
  IF (I .EQ. 1) THEN
    A(I) = B(I)
  ELSE IF (I .EQ. 100) THEN
    A(I) = C(I)
  ELSE
    A(I) = -A(I)
  ENDIF
ENDDO
```

### Peeled loop

```
A(1) = B(1)
DO I = 2, 99
  A(I) = -A(I)
ENDDO
A(100) = C(100)
```

In some cases, boundary-value peeling requires replicating large amounts of code and can greatly increase the size of the executable file. By default, the compiler peels boundary values and expands the code up to a predetermined conservative limit; you can increase this limit by using the `-peel` compiler option or, if you wish to do so on a loop-by-loop basis, the `PEEL` compiler directive.

You can allow the compiler to expand code without bound by using the `-peelall` compiler option or the `PEEL_ALL` directive. In codes containing large numbers of boundary-value operations, allowing code expansion without bound can greatly lengthen compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

Boundary-value peeling can be disabled completely with the `-nopeel` compiler option. Similarly, you can disable peeling on a loop-by-loop basis with the `NO_PEEL` compiler directives. See Appendix B, "Compiler directives," for more information.

---

## Note

---

**Loop boundary-value peeling is not performed on loops that have no tests on boundary values. In other words, the compiler does not try to peel unpeelable loops.**

### Test promotion

Test promotion involves promoting a test out of the loop that encloses it by replicating the containing loop(s) for each branch of the test. The replicated loops contain fewer tests than the originals or no tests at all, so the loops execute much faster. Multiple tests can be promoted, and copies of the loop are made for each test.

IF-DO interchange is an important special case of test promotion that is performed on perfectly nested DO loops that contain IF statements. In this case, the IF statements are interchanged out of the DO loops, and the DO loops are replicated if necessary. An example is shown below.

Original loop	Interchanged loop
DO I = 1, 100	IF (G) THEN
IF (G) THEN	DO I = 1, 100
A(I) = B(I)	A(I) = B(I)
ELSE	ENDDO
A(I) = C(I)	ELSE
ENDIF	DO I = 1, 100
ENDDO	A(I) = C(I)
	ENDDO
	ENDIF

For loops containing large numbers of tests, loop replication can greatly increase the size of the code.

You can control the amount of code replication and test promotion with compiler options and directives. By default, the compiler promotes tests and replicates code up to a predetermined, conservative limit.

The `-ptst` compiler option increases this limit and can cause a noticeable increase in compile time.

The `-ptstall` option promotes all tests regardless of code replication. This can cause a large increase in compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

The `-noptst` option disables test promotion.

The `PROMOTE_TEST`, `PROMOTE_TEST_ALL` and `NO_PROMOTE_TEST` compiler directives provide similar functionality on a loop-by-loop basis. See Appendix B, "Compiler directives," for more information about these directives.

At optimization levels `-O2` and above, the CONVEX FORTRAN compiler automatically performs IF-DO optimizations on DO and hand-rolled loops that contain logical and arithmetic IF statements, IF-THEN-ELSE statements, and computed GOTO statements. Simple and nested loops, and loops with exits are handled.

---

## Pattern matching

Pattern matching allows the compiler to vectorize certain loops that it cannot otherwise vectorize. The compiler recognizes loops that use an IF test to determine a maximum (or minimum) value stored in an array. The following example shows such a loop.

```
CM = 1
XM = A(1)
DO I = 2, N
  IF (A(I) .GT. XM) THEN
    CM = I
    XM = A(I)
  ENDIF
ENDDO
```

The compiler recognizes loops containing recurrences that can be implemented with a special sequence of vector instructions. The following code shows examples of patterns the compiler matches.

```
DO I = 1, N
  X(I) = X(I - 1) + Y(J)
ENDDO

DO I = 1, N
  IF (X(I) .GE. X(M)) M = I
ENDDO

DO I = 1, N
  IF (X(I) .EQ. Y) K = I
ENDDO
```

---

## Conditional induction variables

A *loop induction variable* is a variable whose value is incremented by a constant amount on every iteration of a loop. Loop induction variables that do not change on every iteration are called *conditional induction variables*. The compiler frequently recognizes these variables and generates vector code for expressions involving them.

In the following example, *K* is a conditional induction variable, not an upper limit on the vector.

```
K = 0
DO I = 1, 100
  IF (COND(I) .EQ. .TRUE.) THEN
    K = K + 1
    A(I) = B(K)
    C(K) = D(I)
  ENDIF
ENDDO
```

The compiler generates machine instructions that do the following:

- Save values of *I* for which *COND(I)* is *.TRUE.*
- Count the number of those values.
- Load the vector strip of *B*.
- Expand the vector strip of *B* to the appropriate indexes according to the saved truth values.
- Store the expanded vector in *A(1:100)*.
- Load the vector *D(1:100)*.
- Compress the vector according to the saved truth values.
- Store the vector in *C*.

---

## Inhibitors of vectorization

Any of these conditions can inhibit or prevent vectorization:

- Computed or assigned *GOTO* statements
- Multiple loop entries or exits
- Function or subroutine calls
- I/O statements
- Equivalenced scalar or array variables
- Recurrences

Of these conditions, you are likely to be unfamiliar with recurrence and its variations. The following section defines recurrence and describes its effect on vector optimization.

---

## Recurrence

A value calculated in one iteration of a loop might be referenced in another iteration. When this happens, the value recurs and a *recurrence* exists. (Recurrences are sometimes incorrectly referred to as recursions. To avoid confusion, the term recursion is not used in discussions about loops. Instead, the term recursion is used only to mean subroutine or function-call recursion.)

Recurrence is closely related to *data dependency*. A data dependency is a relationship between two operations such that one operation depends on the results of the other. This implies a definite chronology of operations: execution of one operation must always precede execution of the other, and the execution order cannot be changed without affecting the results.

Dependencies may be either loop-carried or loop-independent. There must be at least one *loop-carried dependency (LCD)* for a recurrence to exist. Any number of *loop-independent dependencies (LIDs)* can occur in a loop, but a recurrence does not exist unless that loop contains at least one loop-carried dependency.

Some loops are written in such a way that the compiler cannot determine whether or not a recurrence exists. A possible recurrence that does not actually exist is called an apparent recurrence. The compiler does not automatically vectorize a loop that contains a real or apparent recurrence.

### Loop-carried dependency

A loop-carried dependency (LCD) exists when one iteration of a loop computes a value that is referenced on another iteration. The loop below contains an LCD.

```
DO I = 1, N
  A(I + 1) = A(I) + 3.14
ENDDO
```

The dependency is carried by the loop from one iteration to the next.

Dependencies can be backward or forward. A backward LCD exists when one iteration references a variable whose value is assigned on a previous iteration. The previous example shows a backward LCD. The first iteration of the loop assigns a value to  $A(2)$ , the second iteration references that value and assigns a new value to  $A(3)$ , and so on. The iterations of the loop are serial, and the loop cannot be vectorized.

A forward LCD exists when one iteration references a variable whose value is assigned on a later iteration. The loop below contains a forward loop-carried dependency.

```
DO I = 1, N
  A(I) = A(I + 1) + 3.14
ENDDO
```

In this example, the first iteration assigns a value to  $A(1)$  and references  $A(2)$ . The second iteration assigns a value to  $A(2)$  and references  $A(3)$ . The reference to  $A(I)$  depends on the fact that the  $I+1$ th iteration, which assigns a new value to  $A(I)$ , has not yet executed. A forward dependency, therefore, does not prevent vectorization of a loop.

The compiler can vectorize some loops containing backward LCDs. The following loop contains an LCD that points backward from  $B(I+1)$  to  $B(I)$ .

```
DO I = 1, N - 1
  A(I) = B(I) + C(I)
  B(I + 1) = D(I) * 3.14
ENDDO
```

In this loop, the assignment to  $A(2)$  on the second iteration depends on the value assigned to  $B(2)$  on the first iteration. The compiler interchanges the statements within the loop so that the assignment to  $B$  occurs before the assignment to  $A$ , as shown below.

```
DO I = 1, N - 1
  B(I + 1) = D(I) * 3.14
  A(I) = B(I) + C(I)
ENDDO
```

When a scalar variable causes an LCD, the compiler eliminates the recurrence with a transformation called *scalar spreading*. Within the body of the loop, the compiler replaces all occurrences of a scalar variable that cause a recurrence with a temporary vector variable. The correct value is assigned to the scalar variable when the loop ends. In the following example, there is an LCD on the variable  $x$ .

### Original loop

```
DO I = 1, 10
  X = A(I)
  = ... X ...
ENDDO
```

### Vectorized loop

```
DO I = 1, 10
  TA(I) = A(I)
  = ... TA(I) ...
ENDDO
X = TA(10)
```

In this example, the temporary vector TA replaces all references to the scalar variable X in the loop. When the loop ends, the value of A(10) is assigned to X.

A backward LCD that cannot be eliminated might not stop vectorization completely. Using temporary vectors, the compiler can sometimes vectorize part of a loop that contains an LCD. The code below shows an example.

### Original loop

```
DO I = 1, N
  A(I) = A(I - 1) + B(I) * C(I)
ENDDO
```

The assignment to A(I) depends on the value of A(I - 1), which is computed on the previous iteration. The vectorized loop below shows that the compiler isolates the dependency by distributing the loop and vectorizes the first distributed part. The second distributed part is executed with scalar instructions. This transformation is called partial vectorization because it distributes a loop into vector and scalar parts.

### Vectorized loop

```
DO I = 1, N
  T(I) = B(I) * C(I)
ENDDO
DO I = 1, N ! Scalar
  A(I) = A(I - 1) + T(I)
ENDDO
```

### Loop-independent dependency

A loop-independent dependency (LID) exists when two operations in a single iteration must be executed in a specific order to produce correct results. The loop below produces two LIDs.

```
DO I = 1, N
  A(I) = B(I) + D(I) ! Statement 1
  B(I) = 0.0 ! Statement 2
  D(I) = D(I) + 1.0 ! Statement 3
ENDDO
```

Here, the proper evaluation of statement 1, which assigns a value to A, prevents statements 2 and 3, which assign new values to B and D, from being evaluated first. Statement 1 is anti-dependent on statements 2 and 3. A forward LID exists between statements 1 and 2; another exists between statements 1 and 3.

---

## Note

---

**LIDs do not normally prevent loop vectorization. LCDs, which cause recurrences, can prevent vectorization. Vectorization is inhibited when an LCD between an assignment and a reference to an array prevents the compiler from generating correct vector code.**

An LID can stop vectorization by preventing the compiler from eliminating an LCD. In the following example, the loop cannot be vectorized.

```
DO I = 1, N - 1
  A(I) = B(I) - C(I)      ! Statement 1
  B(I + 1) = A(I) + D(I) ! Statement 2
ENDDO
```

Interchanging the statements would remove the backward LCD that exists between the assignment to B(I+1) in statement 2 and the reference to B(I) in statement 1. The LID between the assignment to A(I) in statement 1 and the reference to A(I) in statement 2 prevents this interchange.

### Apparent recurrences

An apparent recurrence exists when the compiler lacks sufficient information to prove that an actual recurrence does not exist. Apparent recurrences usually result from using a loop constant of unknown sign or an array reference in an array subscript. The following loop cannot be vectorized because the sign of K is unknown.

```
DO I = M, N
  A(I + K) = 2.0
  A(I) = 0.0
ENDDO
```

If K is positive or zero, the final value of each element of A(M:N) is 0.0. The compiler cannot interchange the statements because the assignment to A(I) must follow the assignment to A(I+K). If K equals -1, the final value of A(M:N-1) is 2.0; only A(N) is 0.0. The compiler must interchange the statements so the assignment to A(I+K) follows the assignment to A(I). Because these conditions are contradictory, neither operation can be performed.

The loop below cannot be vectorized because the compiler cannot determine whether a recurrence exists.

```
DO I = 1, N
  A(J(I)) = A(K(I)) + 1
ENDDO
```

The value assigned to A(J(I)) in one iteration might be used in a subsequent iteration, so the compiler assumes that the references to A(K(I)) form a recurrence.

---

## Reduction

The compiler vectorizes a special recurrence known as reduction. In general, a reduction has the form:

$$X = X \text{ operator } Y$$

where X is a variable not assigned or used elsewhere in the loop, Y is a loop constant expression not involving X, and operator is +, -, \*, .AND., .OR., .EQV., or .NEQV.

The compiler also recognizes reductions of the form

$$X = \text{function}(X, Y)$$

where X is a variable not assigned or referenced elsewhere in the loop, Y is a loop constant expression not involving X, and function is the intrinsic MAX function or intrinsic MIN function.

The loop below computes the sum of the elements of A(1:N) and notes the value of the greatest element. The compiler vectorizes both reductions.

Original loop	Vectorized loop
SUM = 0.0	SUM = VSUM(A(1:100))
X = A(1)	X = VMAX(A(1:100))
DO I = 1, 100	
SUM = SUM + A(I)	
X = MAX(X, A(I))	
ENDDO	

In the optimized code, VSUM and VMAX are single vector machine instructions that return the sum and the greatest value, respectively, of up to 128 elements.

When invoked with the `-uo` option (see Appendix A), the compiler recognizes loops that use an `IF` test to determine a maximum (or minimum) array element. The following example shows such a loop.

```

IM = MAXINT
DO I = 1, N
  IF (A(I) .LT. IM) IM = A(I)
ENDDO

```

## Optimization report

When you compile a program with the `-O2` option, the compiler generates an optimization report for each program unit. The `-or` option determines the report's contents, as shown in the following table.

<code>-or</code> option	Report contents
<code>all</code>	Loop table and array table
<code>loop</code>	Loop table only (default)
<code>array</code>	Array table only
<code>none</code>	No report

For example, consider the matrix multiplication algorithm below. (Line numbers are provided as a reference.)

```

1 PROGRAM EXAMPLE1
2 REAL A(200, 200), B(200, 200), C(200, 200)
3
4 DO I = 1, 200
5   DO J = 1, 200
6     C(I, J) = 0
7     DO K = 1, 200
8       C(I, J) = C(I, J) + A(I, K) * B(K, J)
9     ENDDO
10  ENDDO
11 ENDDO
12
13 END

```

At line 8, individual elements `C(I, J)` are summed directly rather than stored in a temporary scalar variable. Introducing a temporary scalar later assigned to `C(I, J)` would inhibit vectorization. Figure 3 shows the optimization report generated by compiling the program `EXAMPLE1` at optimization level `-O2`.

**Figure 3**  
Optimization report

```
%fc -O2 -or all example1.f
```

```
Optimization by Loop for Routine EXAMPLE1
```

Line Num.	Iter Var.	Reordering Transformation	Optimizing/Special Transformation	Exec. Mode
4	I	Dist		
4-1	I	FULL VECTOR	Inter	
4-2	I	FULL VECTOR	Inter	
5-1	J	Scalar		
5-2	J	Scalar		
7-2	K	Scalar		

```
Line Iter. Analysis
Num. Var.
```

4-1	I	Interchanged to innermost
4-2	I	Interchanged to innermost

```
Array References for Routine EXAMPLE1
```

Line Num.	Var. Name	Optimization	Dependencies
8	A	Hoist	
8	C	Sunk	
8	C	Hoist	

Some numbers in the Line Num. column are actually pairs, indicating that at least one loop was distributed. In this example, loop distribution creates two loop nests, called *distributed parts*. In each pair, the first number is the source file line number and the second number is the distributed part number.

The loop table, which lists the optimizations performed on each loop, has two parts. The first part shows that these transformations were performed:

- The I loop at line 4 was distributed and (in both distributed parts) interchanged and fully vectorized.
- The J and K loops were not vectorized.

The second part of the loop table provides additional information about the transformations performed. In the example, the compiler reports that the I loop in each distributed part was interchanged to innermost, to allow hoist and sink.

Following is an example of other transformations the compiler performs. (Line numbers are provided as a reference.)

```

1 SUBROUTINE EXAMPLE2 (A, N, ZERO, NEGATE, SUM)
2 REAL A(N), SUM
3 LOGICAL ZERO, NEGATE
4
5 SUM = 0.0
6 DO I = 1, N
7   SUM = SUM + A(I)
8   IF (ZERO) THEN
9     A(I) = 0
10  ELSE IF (NEGATE) THEN
11    A(I) = -A(I)
12  ENDIF
13  IF (I .EQ. 1 .OR. I .EQ. N) THEN
14    A(I) = -1
15  ENDIF
16 ENDDO
17 END

```

Figure 4 shows the optimization report generated by compiling the subroutine EXAMPLE2 above for vectorization. The `-c` option suppresses loading because no main program is included.

**Figure 4**  
Optimization report

```
%fc -O2 -c example2.f
```

Optimization by Loop for Routine EXAMPLE2

Line Num.	Iter. Var.	Reordering Transformation	Optimizing / Special Transformation	Exec. Mode
6	I	FULL VECTOR	Peel Reduction	
6	I	FULL VECTOR	Peel Reduction	
6	I	FULL VECTOR	Peel Reduction	

Line Num.	Col. Num.	Test Transformation	Analysis
8	14	TEST PROMOTED	Test promoted out of loop on 6
10	19	TEST PROMOTED	Test promoted out of loop on 6
13	16	TEST REMOVED	Peeled first iteration of loop on 6
13	30	TEST REMOVED	Peeled last iteration of loop on 6

The `Peel` entry indicates that the compiler recognizes peelable tests and removes them from the loop. Note that line number 6 appears three times: this signifies that the loop is replicated three times. The `Test Transformation` section indicates that the tests at lines 8 and 10 were promoted out of the loop, resulting in two replications; the third replication is created in case both `IF` tests fail.

The other two entries in the `Test Transformation` section indicate that the `IF` test at line 13 was peeled out of the loop and the test removed.



At optimization level -O3, the CONVEX FORTRAN compiler performs vector and parallel optimization to enhance program performance. Unlike vector optimization, parallel optimization does not reduce CPU time. Instead, processing of a single program is spread across multiple CPUs, reducing the program's time to solution.

---

## Basic operation

Parallel optimization divides a program into *threads*. A thread is a sequence of instructions that must execute on a single CPU.

The CONVEX FORTRAN compiler finds parallelism at the loop level. The compiler vectorizes inner loops and parallelizes outer loops. Often, the outer loops are the strip-mine loops that the compiler creates when it vectorizes an inner loop.

As with vector optimization, the compiler distributes and interchanges loops to produce the most efficient parallel code. The compiler can parallelize most scalar reductions and assignments with the addition of synchronization code.

As an example of the transformations the compiler performs at optimization level -O3, consider the matrix multiplication shown below.

```
DO I = 1, N
  DO J = 1, N
    C(I,J) = 0.0
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

The compiler processes this loop nest by distributing the loop nest containing the I and J loops, as shown in the following example.

```
DO I = 1, N
  DO J = 1, N
    C(I,J) = 0.0
  ENDDO
ENDDO
```

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

The compiler moves the I loop to the innermost position in each nest, as shown below, so that it can retrieve contiguous elements on successive iterations.

```
DO J = 1, N
  DO I = 1, N
    C(I,J) = 0.0
  ENDDO
ENDDO
```

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

The compiler strip mines both I loops to the optimal vector length, a function of the loop upper bound (N). In the following examples, MVSL represents that function, and V0 and V1 represent vector registers that can contain up to 128 64-bit elements.

```

M = MVSL(N)
DO J = 1, N
  DO IOUTER = 1, N, M
    C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
  ENDDO
ENDDO
DO J = 1, N
  DO K = 1, N
    DO IOUTER = 1, N, M
      V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
      V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
      V0 = V0 + V1 * B(K, J)
      C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
    ENDDO
  ENDDO
ENDDO

```

In the second nest, the compiler interchanges the IOUTER strip-mine loop outside of the K loop.

```

M = MVSL(N)
DO J = 1, N
  DO IOUTER = 1, N, M
    C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
  ENDDO
ENDDO
DO J = 1, N
  DO IOUTER = 1, N, M
    DO K = 1, N
      V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
      V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
      V0 = V0 + V1 * B(K, J)
      C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
    ENDDO
  ENDDO
ENDDO

```

In the following example, PARALLEL DO represents a loop that can be processed by multiple CPUs.

```
M = MVSL(N)
PARALLEL DO J = 1, N
  DO IOUTER = 1, N, M
    C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
  ENDDO
ENDDO

PARALLEL DO J = 1, N
  DO IOUTER = 1, N, M
    DO K = 1, N
      V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
      V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
      V0 = V0 + V1 * B(K, J)
      C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
    ENDDO
  ENDDO
ENDDO
```

As shown below, the compiler hoists a vector load and sinks a vector store out of the K loop. The remaining reference to vector V1 chains with the vector addition and vector multiplication in the next statement, resulting in even faster execution.

```
M = MVSL(N)
PARALLEL DO J = 1, N
  DO IOUTER = 1, N, M
    C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
  ENDDO
ENDDO

PARALLEL DO J = 1, N
  DO IOUTER = 1, N, M
    V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
    DO K = 1, N
      V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
      V0 = V0 + V1 * B(K, J)
    ENDDO
    C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
  ENDDO
ENDDO
```

The combination of these optimizations results in generated code that performs at a level similar to that of hand-tuned assembly code.

---

## Inhibitors of parallelization

Parallelization and vectorization are so closely related that most things that prevent vectorization can prevent parallelization. Specific factors that can inhibit or prevent automatic parallel optimization are

- Multiple entries or exits
- Function or subroutine calls
- I/O statements
- Equivalenced scalar or array variables
- Nondeterminism of parallel execution
- Loop-carried dependencies (LCDs)

---

## Loops with subroutine calls

The compiler does not automatically parallelize a loop containing a subroutine call. You can force it to parallelize such a loop by inserting the `FORCE_PARALLEL` directive before the loop. This directive allows parallelization regardless of potential dependencies that the compiler detects. Certain actual dependencies, such as from one scalar to another, cause the compiler to ignore the directive.

If you use `FORCE_PARALLEL`, you must recompile the called subroutine (or any routines called indirectly) for re-entrancy with the `-re` option. Each invocation of a subroutine compiled with `-re` maintains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary variables. For more information about compiler directives, see Appendix B.

The call to SUB in the following example prevents the compiler from automatically parallelizing the loop. FORCE\_PARALLEL overrides the compiler's decision, and the compiler generates parallel code for the loop.

```
...
C$DIR FORCE_PARALLEL
DO I = 1, N
    CALL SUB(A, B, I, N)
ENDDO
...
END
OPTIONS -re ! compile for reentrancy
SUBROUTINE SUB(A, B, I, N)
REAL A(N), B(N)
A(I) = B(I) * 3.14
RETURN
END
```

The way the code is written guarantees that SUB does not contain any operations violating data independence, so the code can execute safely in parallel.

If a subroutine is called only from within a parallelized loop, compile the subroutine at a level lower than -O3. Only one loop at a time can be run in parallel. Code that can be parallelized within the subroutine cannot execute in parallel. Additional code generated to parallelize the called routine is useless overhead. There is one exception: if a routine called from a parallel loop uses directives such as BEGIN\_SECTION or BEGIN\_ORDER to synchronize operations, it must be compiled at -O3 to produce correct results.

---

## Caution

---

If you use FORCE\_PARALLEL to parallelize a loop containing an actual recurrence, the behavior of the loop may change from one execution to the next. Errors may result at runtime, but no amount of testing can guarantee that an error will be revealed. Analyze your data and algorithms to ensure that your code can be safely parallelized before using this directive. A good test is to run the loop with iterations in reverse order, for example, DO I=N, 1, -1.

For more information about compiler directives, see Appendix B.

Another way to allow loops containing subroutine calls to parallelize (or vectorize) is to inline the subroutine. See Chapter 8, "Inline substitution," for more information.

---

## Loop-carried dependency

Chapter 3 discusses how recurrence and dependency affect vectorization. While only backward dependencies interfere with vectorization, forward and backward dependencies affect parallelization.

The loop in the following example has no dependencies. The compiler can strip mine and vectorize the inner loop and parallelize the strip-mine loop.

```
DO I = 1, N
  A(I) = A(I) + 3.14
ENDDO
```

The compiler transforms the outer strip-mine loop so that it runs in parallel on a multiprocessor machine. The result is a *parallel vector loop*.

The loop below has a backward loop-carried dependency (LCD) caused by the assignment to  $A(I+1)$ . The loop cannot be vectorized or parallelized by the compiler. The loop remains in scalar form.

```
DO I = 1, N - 1
  A(I + 1) = A(I) + 3.14
ENDDO
```

The following loop has a forward LCD. Because forward LCDs do not interfere with vectorization, the compiler strip mines and vectorizes the loop. It is not safe to parallelize a loop that has an LCD, however. The result is a strip-mine vector instead of a parallel vector loop:

```
DO I = 1, N - 1
  A(I) = A(I + 1) + 3.14
ENDDO
```

If a loop has dependencies that prevent the compiler from automatically parallelizing it, you can instruct the compiler to insert *synchronization* code to honor the dependencies. The compiler can then parallelize the loop. Synchronization code causes execution of a thread to halt momentarily, if necessary, until an operation in another thread, on which the halting thread depends, has been performed. The `SYNCH_PARALLEL` directive instructs the compiler to generate synchronization code. More information about CONVEX FORTRAN directives appears in Appendix B.

The overhead of synchronization code often outweighs performance gains from parallelization. Synchronized parallel loops are advantageous only if the amount of code that contains dependencies is small compared to the amount of code that does not contain dependencies.

The compiler can handle most scalar assignments and reductions within parallel loops. For example, the compiler can generate parallel code for the following loop.

```
DO I = 1, N
  IF (A(I) .LE. 0.0) THEN
    S = S + B(I) * C(I)
    X = B(I)
  ENDIF
ENDDO
```

---

## Parallelizing code outside of loops

The compiler does not automatically parallelize code outside a loop. You can use tasking directives to instruct the compiler to parallelize such code. The `BEGIN_TASKS` directive tells the compiler to begin parallelizing a series of tasks. The `NEXT_TASK` directive marks the end of a task and the start of the next task. The `END_TASKS` directive marks the end of a series of tasks to be parallelized. For more information about tasking directives, see Appendix B.

The following example shows how to insert tasking directives into a section of code containing three tasks that can be run in parallel.

```
C$DIR BEGIN_TASKS
  <statement 1>
C$DIR NEXT_TASK
  <statement 2>
  <statement 3>
C$DIR NEXT_TASK
  <statement 4>
C$DIR END_TASKS
```

The compiler transforms the above code into a parallel loop and creates machine code equivalent to that shown below.

```
C$DIR FORCE_PARALLEL
      DO I = 1,3
          GOTO (10,20,30) I
10     <statement 1>
          GOTO 40
20     <statement 2>
          GOTO 40
30     <statement 3>
          GOTO 40
      ENDDO
40    CONTINUE
```



This chapter describes a strategy for optimizing FORTRAN programs. The same principles apply to developing new applications, but the examples address the more common need to optimize existing code.

For programs that manipulate arrays, vectorization usually provides the greatest performance gains of any possible optimization. Focus your efforts first on vectorizing the loops in subprograms that account for the major part of your program's execution time. When you obtain the best vector performance, you can frequently achieve additional gains through parallelization.

---

## Note

---

When you are optimizing code, it is easy to produce a fast program that no longer gives correct results. The goal of optimization is to make a program run fast without adversely affecting results. Test your code at each stage of the optimization process to make sure the optimized program still gives correct results.

---

## Step 1. Compile the program

1. Compile the program with minimal optimizations (-no).
2. Run the resulting program and check the output. If you are porting a program from another machine, compare the new output with output from the old machine. If you are compiling a new application, compare the output with expected values. If the output does not match the expected results, allowing for roundoff error, use `csd` to pinpoint and fix the logic error that is causing the problem. See Chapter 9, "Limits of optimization," for possible causes of such errors.

If you are certain there is no logic error, check for violations of ANSI standards (see Chapter 9). If the code does not violate ANSI standards, use the `contact` utility to report a possible compiler bug.

Do not skip this first step. Optimizations performed at higher levels make debugging much more difficult. Be sure your program produces correct results before you start to add optimizations.

---

## Step 2. Add scalar optimizations

1. Compile the program with scalar optimization (`-O1`). Use the `-pa` option to include instrumentation for profiling with CXpa. If you use one of the profilers contained in the CONVEX Consultant instead of the CONVEX Performance Analyzer (CXpa), you can still perform most of the steps in this chapter. You cannot analyze the performance of individual loops, however. See the *CONVEX Consultant User's Guide* to determine the appropriate options and commands for using the Consultant profilers.

Code rarely slows down at `-O1`. If you do not obtain the expected results at higher optimization levels, you may need to recompile part of your program at `-O0`. This problem is the only reason to compile a program at `-O0`.

2. Compare the output of your program with the output produced in **Step 1**. Because scalar optimization rarely affects the output, the results, allowing for differences in floating-point roundoff, should be the same.

If the output is significantly changed, use a binary search to isolate the subprogram responsible for the change. Compile half the subprograms at `-no` and the other half at `-O1`. Run the program and check the output to determine which half contains the offending routine. Then, split the suspect group of subprograms in half. Compile half of the suspect routines at `-no` and the other half at `-O1`. Continue this process until you isolate the routine containing the error.

When you have isolated the erroneous routine, check its source code and fix any errors that you find. If you do not find logic errors, recompile that subprogram at `-no` and continue optimizing the rest of the program.

3. Run the program under the same profiler you used in **Step 1**. Note the program's total execution time and which routines consume the most time. Concentrate your optimization efforts on these routines.

---

## Step 3. Add vectorization

You can approach vectorization in one of two ways.

### Step 3a. Add vectorization in one step

The more common approach is to compile the entire program at `-O2`. Nothing is wrong with this approach, except that it may not be safe or desirable in all cases. If a program has hidden dependencies, misuses directives, encroaches on the limits of floating-point precision, or violates certain restrictions of the ANSI standard, the code may no longer produce the same output after it has been vectorized. It is also possible, although rare, that code will slow down due to vectorization. The reasons for these phenomena are discussed in Chapter 9, "Limits of optimization."

### Step 3b. Add selective vectorization

This step represents an alternative approach. Its advantage is that, if unexpected results occur, it allows you to isolate the cause of the problem more quickly. Although safer, this approach can take more time. If you have compiled complete programs at `-O2` in the past and achieved good results, there is no reason not to continue with that approach. If your code slows down or gives incorrect answers at `-O2`, then backtrack and carry out the steps outlined below. Otherwise, go on to Step 4. If you have had problems with vectorization in the past, however, you might want to begin with the procedures outlined below.

Do not try to vectorize a program unit that produces incorrect results at `-O1`. The compiler continues to perform scalar optimizations at `-O2`, so any problems that you encounter at `-O1` are sure to recur when you add vectorization.

1. Look at the CXpa output from Step 2 to determine which routines consume the most CPU time. Compile the most time-consuming routines for vectorization. To do this, place the `OPTIONS -O2` statement above the subprogram in the source code or use the `-O2` option on the `fc` command line. Compile the rest of the program for program-unit optimization and CXpa profiling.
2. Compare the output of your program with the output produced in Step 2. The results, allowing for floating-point roundoff, should be unchanged.

If the output is significantly changed, use the binary search procedure described in Step 2 to isolate the offending routine. Check the source code and fix any errors that you

find. If you do not find any logic errors, recompile the affected subprogram at `-O1` and continue optimizing the rest of the program.

3. Run the program under `CXpa`. Take note of the program's total execution time and the most time-consuming routines. Compare this `CXpa` output with the `CXpa` output from **Step 2** and determine the effect of vectorization on your program's performance.
4. Repeat **Step 3a**, vectorizing routines that consume a significant amount of CPU time in the new `CXpa` output and have not been vectorized. Continue until you have vectorized all time-consuming routines that can be properly vectorized; proceed to **Step 4**.

---

## Step 4. Enhance vector optimization

1. Run the vectorized program under `CXpa` to produce a loop-level profile of the most time-consuming routines.
2. Study the `CXpa` profile and the optimization report. Look for loops that are not vectorized and consume significant amounts of CPU time. Note which of these loops are inner loops, which are candidates for vectorization.

The goal is to increase the number of vectorized loops. Look for apparent recurrences that prevent the compiler from vectorizing time-consuming loops. If you find loops with apparent recurrences that do not contain actual recurrences, use the `NO_RECURRENCE` directive to tell the compiler it is okay to vectorize the loop.

Complicated conditional structures can prevent the compiler from vectorizing a loop. If a loop containing a conditional does not vectorize, try to rewrite the code to remove the conditional from the loop.

3. When you are satisfied that no more loops can be vectorized or the loops that can be vectorized do not consume a significant amount of time, you may still be able to improve the efficiency of your code. Try the following techniques:
  - Simplify conditionals. Even if a loop is vectorized, an embedded conditional can slow it down.
  - Simplify array subscripts. Array subscripts that require many operations to evaluate can slow down the execution of a loop.
  - Look for loops with short vector lengths (small trip counts). If the trip count is small, the loop probably runs faster in scalar form than it does in vector form. On the

C100 and C200 Series, this slowdown occurs when the trip count is less than five. Use the `SCALAR` directive to stop the compiler from vectorizing such a loop.

- Look for unnecessary strip mines and inefficient strip-mine lengths. Use `CXpa` to determine whether a vector loop is strip mined. Use the `MAX_TRIPS` directive to stop the compiler from creating unnecessary strip mines around a vector loop.
- Consider inlining any short routines that are called frequently or consume a large amount of CPU time. See Chapter 8 for information about inlining.

For more examples of how to tune your code for better vector performance, refer to Chapter 7, “Manual optimization techniques.”

4. When you finish modifying your code, recompile it and run the program under `CXpa`. Check your program’s output to make sure the output has not changed. If it has changed, locate the directive that is causing the problem and remove it.

When your program’s output is correct, compare the `CXpa` profile with the profile obtained in part 1 of Step 4. Note the effect of the changes you made on each routine’s CPU time. Some changes may cause your code to slow down. Remove those changes.

---

## Note

---

Automatic vectorization typically reduces CPU time by up to 90%. If your machine has two or more CPUs and the program is the only compute-intensive application running on it at a given time, consider optimizing the program for parallel processing. If not, go to Step 7, “Wrapping up.”

---

## Step 5. Add parallelization

You can approach parallelization in two ways. The comments made about vectorization in Step 3 apply to parallelization. Performance gains from parallelization are usually smaller than those from vectorization, and your chances of running into problems can be greater.

### Step 5a. Add parallelization in one step

Based on your own experience, you can begin by compiling your entire program at `-O3`, or you can follow the step-by-step approach outlined in **Step 5b**. Parallelization requires additional effort to ensure that results remain correct. The best approach is to add parallelism selectively. If you choose the “all at once” approach and run into trouble, backtrack and begin down the other path.

### Step 5b. Add selective parallelization

Unlike vectorization, parallelization does not reduce a program’s CPU time. In fact, CPU time may increase slightly when a program is parallelized. By spreading work across multiple CPUs, however, parallelization can reduce a program’s time to solution. If your program is going to run on a machine with multiple CPUs, and turn-around time is more important than CPU time, consider parallelizing your program. Otherwise, go to **Step 7**.

To achieve the best performance gains from parallelization, your program must run on a lightly or moderately loaded machine, where CPUs are available for parallel execution. If your program is to run in a heavily loaded environment, it may not benefit from parallel optimization. If this is the case, go to **Step 7**.

At best, parallelization can reduce a program’s turnaround time by a factor of  $N$ , where  $N$  is the number of CPUs on your machine. The improvement depends on your program’s algorithm. Follow the procedures in this section to obtain the best parallel performance out of your program’s algorithm.

1. Look at the CXpa output from **Step 4**. Determine which routines consume the most CPU time and compile them for parallelization. To do this, place the `OPTIONS -O3` statement above the subprogram in the source code or use the `-O3` option on the `fc` command line. Compile the rest of the program for vectorization and CXpa profiling.
2. Compare the output of your program with the output produced in 4) of **Step 4**. The results, allowing for floating-point roundoff, should be unchanged.

If the output is significantly changed, use the binary search procedure described in 2) of **Step 2** to isolate the offending routine. Check the source code and fix any errors that you find. If you do not find logic errors, recompile the affected subprogram at `-O2` and continue optimizing the rest of the program.

3. Run the program under CXpa. Note the process virtual times in each routine. See the *CONVEX Performance Analyzer (CXpa) User's Guide* for procedures to calculate the parallel efficiency of your code. If most of the regions in a routine have an efficiency less than or equal to one, parallelization of the routine is probably counter-productive and should be removed. See the *CONVEX Performance Analyzer CXpa User's Guide* for information on interpreting process virtual time.
4. Repeat Step 5a, parallelizing those routines that consume a significant amount of process virtual time in the new CXpa output and have not been parallelized. Continue until you have parallelized all routines that can be safely and productively parallelized; then proceed to Step 6.

---

## Step 6. Enhance parallel optimization

1. Run the parallelized program under CXpa to produce a loop-level profile of the most time-consuming routines.
2. Study the CXpa profile and the optimization report. Look for loops that failed to parallelize. A scalar loop that consumes significant CPU time is a candidate for parallelization. Inner loops are less likely candidates.
3. Look for apparent dependencies that stop the compiler from parallelizing a scalar or vector loop. Remove an apparent dependency by inlining the subprogram call or by applying the `NO_RECURRENCE` or `FORCE_PARALLEL` directive. If you find a real dependency, consider replacing the routine with a call to a VECLIB subprogram that performs the same function in parallel. For more information about VECLIB, see the *CONVEX VECLIB User's Guide*.
4. When you finish modifying your code, recompile it and run the program under CXpa. Check your program's output to make sure it has not changed. If it has changed, locate the directive causing the problem and remove it.

When your program produces correct output, compare the CXpa profile with the profile obtained in 3) of Step 5. Note the effect of the changes you have made on the process virtual time of each region. Some changes may cause your code to slow down. Remove those changes.

---

## Step 7. Wrap up

The `-pa` option causes the compiler to insert special code and data, called *instrumentation*, into your program. When your program is completely optimized, recompile it without the `-pa` option to remove the instrumentation overhead.



---

# Efficient programming constructs

# 6

FORTRAN has long been the language of choice for advanced scientific and engineering applications. It provides a set of simple and effective programming constructs that are readily optimized by advanced compilers such as the CONVEX FORTRAN compiler. By carefully choosing programming constructs, you can easily create programs that make best use of the CONVEX system.

---

## Data type in calculations

In CONVEX FORTRAN, floating-point variables and constants can be declared to be REAL (REAL\*4), DOUBLE PRECISION (REAL\*8), or REAL\*16. Using lower precision reduces your program's memory requirements and usually increases performance. However, if your code requires conversion of operands from one precision to another when evaluating an expression, the performance benefit may be lost because of the extra time required to do the conversion.

Integer operations are usually faster than floating-point operations. For vector operations, the difference can be quite small. When integer and floating-point operations are combined in the same expression, the overhead caused by type conversions usually outweighs any performance benefit that can be gained by using integers. Avoid writing mixed-mode expressions, especially within vectorized loops.

---

## Writing efficient loops

When you are writing loops, the most important performance consideration is whether the loop will vectorize. The compiler vectorizes only loops that are counted. A counted loop is one whose iteration value can be determined at runtime before the loop is executed. The iteration value, or iteration count, is required to determine the number and length of the vector strips.

A counted loop has at least one induction variable and a fixed stop value. An induction variable is one whose value is incremented or decremented by a fixed amount on every iteration. If the incrementing or decrementing may take place only if some condition is true, then the induction variable is conditional.

Counted loops can be DO loops or DO WHILE loops, or loops written with IF and GOTO statements. The following example shows four typical counted loops.

```
DO 10 I = 1, 1000
  A(I) = A(I) * B(I)
10 CONTINUE

DO I = 1000, 1, -1
  A(I) = B(I) / 4.16
ENDDO

I = 1
DO WHILE (I .LT. 1000)
  A(I) = A(I) * B(I)
  I = I + 4
ENDDO

I = 1
5 A(I) = B(I) + C(I)
  I = I + 1
  IF (I .LT. 10) GOTO 5
```

I is the induction variable for each of these loops. I is assigned a value at the beginning of each loop and is incremented or decremented by a constant integer value on every iteration. Each loop terminates when I reaches a predetermined stop value. The compiler determines the iteration count for each loop and sets up the vector registers and functional unit for vectorization.

If a loop uses an iteration variable that is not incremented or decremented by a constant nonzero integer value, the loop has no induction variable and the compiler cannot vectorize it. The following example shows a loop that has no induction variable.

```
I = 1
DO WHILE (I .LT. 1000)
  A(I) = B(I) * C(I)
  I = I * 2
ENDDO
```

When this loop executes, it increments the value of *I* by one on the first iteration, two on the second iteration, four on the third iteration, and so on. Because *I* is not incremented by a constant value, the loop has no induction variable, and the compiler cannot vectorize it.

The loop can be unrolled by hand, as shown below. Because the loop overhead is eliminated, the unrolled code runs faster than the original loop.

```
A( 1) = B( 1) * C( 1)
A( 2) = B( 2) * C( 2)
A( 4) = B( 4) * C( 4)
A( 8) = B( 8) * C( 8)
A(16) = B(16) * C(16)
A(32) = B(32) * C(32)
A(64) = B(64) * C(64)
A(128) = B(128) * C(128)
A(256) = B(256) * C(256)
A(512) = B(512) * C(512)
```

If the iteration variable of a loop is incremented by a non-integer constant, the loop has no induction variable and the compiler cannot vectorize it. The loop in the following example increments *I* by a REAL value, which prevents vectorization of the loop.

```
Z = 4.0
I = 1
DO WHILE (I .LT. 1000)
  A(I) = A(I) * Z
  I = I + Z
ENDDO
```

---

## Caution

---

If the start, stop, or iteration value of a loop falls outside the range of `INTEGER*4` (31 bits), the compiler may truncate the value to 31 bits when it vectorizes the loop. Avoid using start, stop, or iteration values that exceed the range of `INTEGER*4`.

For a DO WHILE loop to vectorize, the WHILE test must compare the induction variable to a fixed stop value. The test can use any of the following comparison operators: .GT., .LT., .GE., .LE., and .NE.

More complicated iteration tests, such as the one shown in the following example, often prevent the compiler from vectorizing a loop.

```
I = 1
J = 0
DO WHILE ((I .LT. N) .AND. (J .LT. N))
  A(I) = A(I + J)
  I = I + 1
  J = J + M
ENDDO
```

The complexity of the WHILE test prevents the compiler from generating code to determine the loop's iteration count at runtime. As a result, the compiler cannot vectorize the loop.

A stop value can be a variable or a constant, but its value must be determined at runtime prior to the execution of the loop and cannot change within the loop. The example below shows a loop whose stop value changes within the loop.

```
DO WHILE (I .LT. N)
  A(I) = B(I)
  IF (A(I + 1) .GT. 0) N = A(I + 1)
  I = I + 1
ENDDO
```

If the array A contains a positive value within the range of 0 to N, the value of N is altered. The compiler cannot predict what the contents of A might be; therefore, it cannot predict how the value of the stop variable, N, might change within the loop. This makes it impossible to determine the number of iterations the loop will make. The loop is uncounted and cannot be vectorized.

If a loop has more than one exit, the compiler cannot predict which sections of code within the loop will be executed at runtime. This prevents the compiler from generating equivalent vector instructions. Loops that have alternate exits, such as the loop below, do not vectorize.

```

DO I = 1, N
  A(I) = C(I) + D(I)
  IF (A(I) .LT. 0.0) THEN
    GOTO 30
  ENDIF
  A(I) = A(I) / 2.0
ENDDO
30 CONTINUE

```

The compiler can vectorize most loops that contain IF tests. Embedded conditionals, however, reduce the efficiency of vector loops. Remove conditionals from loops when possible. Check boundary conditions before or after, rather than within, the loop.

The following example shows a series of conditionals embedded within a DO loop. The conditionals do not prevent vectorization of the loop, but they do slow it down.

```

DO I = 1, 10000
  IF (I .LT. 2000) THEN
    C(I) = A(I) * 2000.0 + COS(A(I))
    B(I) = B(I) * C(I) ** 4 / A(I)
  ENDIF
  IF ((I .GE. 2000) .AND. (I .LT. 4000)) THEN
    C(I) = A(I) + COS(A(I))
    B(I) = B(I) + C(I)
  ENDIF
  IF ((I .GE. 4000) .AND. (I .LT. 6000)) THEN
    C(I) = A(I) + 2000.0
    B(I) = B(I) ** 3
  ENDIF
  IF (I .GE. 6000) THEN
    C(I) = A(I)
    B(I) = 1.0
  ENDIF
ENDDO

```

Remove the conditional by splitting the single DO loop into four separate loops, as shown below. This change to the source code improves performance dramatically.

```
DO I = 1, 1999
  C(I) = A(I) * 2000.0 + COS(A(I))
  B(I) = B(I) * C(I)**4 / A(I)
ENDDO
DO I = 2000, 3999
  C(I) = A(I) + COS(A(I))
  B(I) = B(I) + C(I)
ENDDO
DO I = 4000, 5999
  C(I) = A(I) + 2000.0
  B(I) = B(I)**3
ENDDO
DO I = 6000, 10000
  C(I) = A(I)
  B(I) = 1.0
ENDDO
```

The following example shows boundary tests that can be removed from a loop.

```
DO I = 1, 1000
  DO J = 1, 1000
    IF ((I .EQ. 1) .OR. (I .EQ. 1000)) THEN
      IF ((J .EQ. 1) .OR. (J .EQ. 1000)) THEN
        A(I, J) = 0.0
      ELSE
        A(I, J) = B(I, J)
      ENDIF
    ELSE
      A(I, J) = B(I, J)
    ENDIF
  ENDDO
```

When boundary values are set outside the loop, this code fragment runs several times faster:

```
DO I = 2, 999
  DO J = 2, 999
    A(I, J) = B(I, J)
  ENDDO
ENDDO
A( 1, 1) = 0.0
A( 1, 1000) = 0.0
A(1000, 1) = 0.0
A(1000, 1000) = 0.0
DO I = 2, 999
  A( 1, I) = B( 1, I)
  A(1000, I) = B(1000, I)
  A( I, 1) = B( I, 1)
  A( I, 1000) = B( I, 1000)
ENDDO
```

Most loops that are hand coded using GOTO statements do not vectorize. A hand-coded loop usually lacks a fixed stop value and a recognizable induction variable. If a hand-coded loop has these characteristics, however, it can be vectorized.

---

## Optimizing memory accesses

In FORTRAN, arrays are stored in column-major order. As a result, using innermost loops that vary the leading, or leftmost, dimension is faster than using innermost loops that vary the trailing, or rightmost, dimension. Write inner loops so that most of the accesses are to the leading dimension. If this is not possible, use the ROW\_WISE directive to store arrays in row-major order.

CONVEX FORTRAN automatically interchanges many loops to optimize the efficiency of array accesses. Vector stride and memory interleaving also affect a loop's efficiency. These issues are discussed later in this chapter.

The following example shows three loops in order of increasing efficiency.

```
DO J = 1, N ! least efficient
  A(1, 1, J) = 4.0
ENDDO
```

```
DO J = 1, N
  A(1, J, 1) = 4.0
ENDDO
```

```
DO J = 1, N ! most efficient
  A(J, 1, 1) = 4.0
ENDDO
```

In this example, the compiler interchanges the J and I loops:

Original code	Optimized code
DO I = 1, N	DO J = 1, N
DO J = 1, N	DO I = 1, N
A(I, J, 1) = 4.0	A(I, J, 1) = 4.0
ENDDO	ENDDO
ENDDO	ENDDO

If the trip count of an outer loop is much smaller than that of the inner loop, the compiler may not interchange the loops even though it could achieve more efficient memory accesses by doing so. The compiler realizes that a few slow memory accesses can be faster than many fast accesses. If the compiler cannot determine the trip count, the compiler might interchange two loops to achieve fast memory accesses even though this results in a much larger average trip count on the outer loop. If you write most loops to access the leading dimension of an array, you can minimize the number of compromises the compiler must make.

---

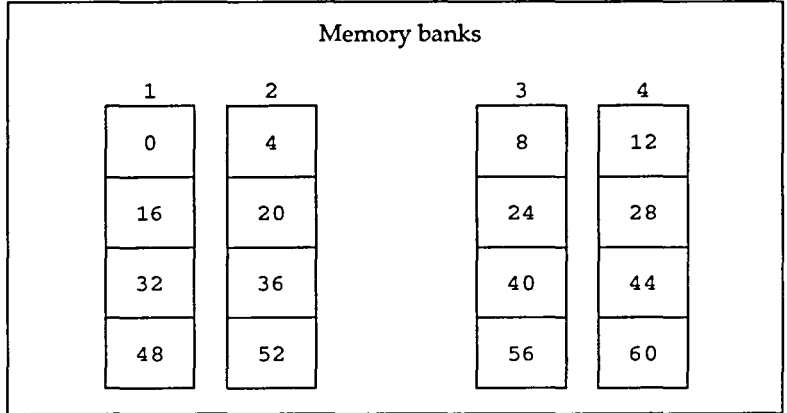
## Memory interleaving

The CONVEX C200 Series supercomputer requires eight clock cycles to access data stored in main memory. To speed up memory accesses, the CPU posts requests for data before the data is needed.

Main memory comprises at least four banks of dynamic RAM. The memory system stores data so that contiguous words are in separate memory banks. This is called *memory interleaving*. One memory bank is accessed on each clock cycle. As a result, sequential requests to ascending banks proceed at full speed.

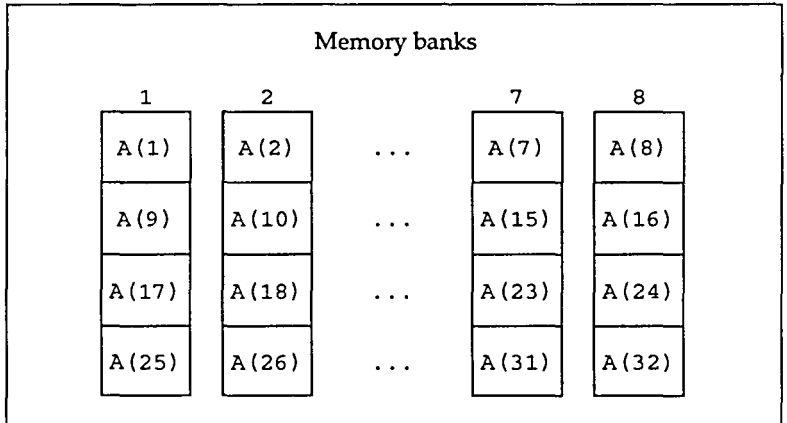
Figure 5 shows the configuration on a four-bank machine. Byte addresses are expressed in decimal notation.

Figure 5  
Four-way  
interleaved  
memory



Eight memory banks are needed to return data at the rate of one word per clock cycle. A load instruction, for example, takes eight cycles to return data. If a program makes eight load requests, at a rate of one request per clock cycle, each to a separate bank, data returns at a rate of one per clock cycle, beginning eight clock cycles after the first request. Memory interleaving directly affects efficient array accesses. Figure 6 shows a one-dimensional array in eight-way interleaved memory.

Figure 6  
One-dimensional  
array in eight-way  
interleaved  
memory



The loop below processes an array sequentially. After an initial wait of eight clock cycles, the CPU receives one data word of data per clock cycle.

```

DO 10 I = 1, 32
  A(I) = A(I) + 1
10 CONTINUE

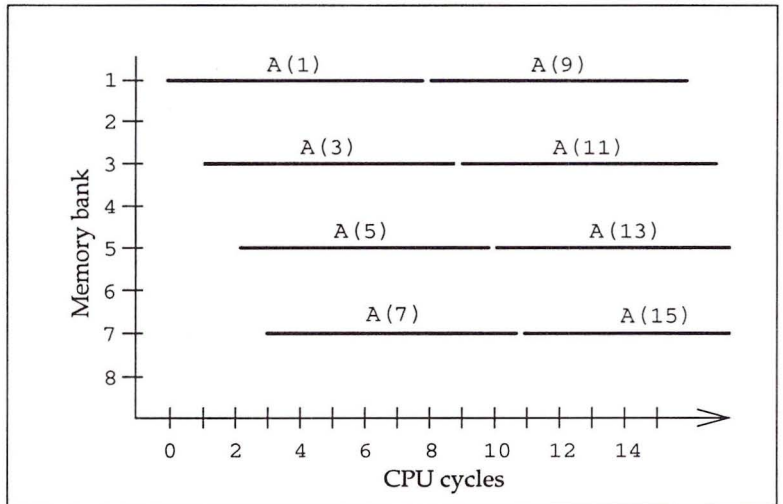
```

The following loop, however, causes memory bank conflicts. The CPU must wait for memory requests to be filled.

```
DO 10 I = 1, 32, 2
  A(I) = A(I) + 1
10 CONTINUE
```

Figure 7 shows the timing relationships that cause these bank conflicts.

**Figure 7**  
Bank conflict



Load requests occur each clock cycle. The first request, for A(1), keeps bank 1 occupied for eight clock cycles. The CPU cannot access the data in A(9) until this first access is satisfied. This results in a delay of four clock cycles.

Memory bank conflicts occur when an array's stride does not efficiently use the memory of the computer. An array's stride is the difference in the index value between two successive iterations. In the loop below, array A has a stride of one.

```
DO 10 I = 1, 32, 1
  A(I) = A(I) + 1
10 CONTINUE
```

The following loop, which accesses every other element of the array, has a stride of two.

```
DO 10 I = 1, 32, 2
  A(I) = A(I) + 1
10 CONTINUE
```

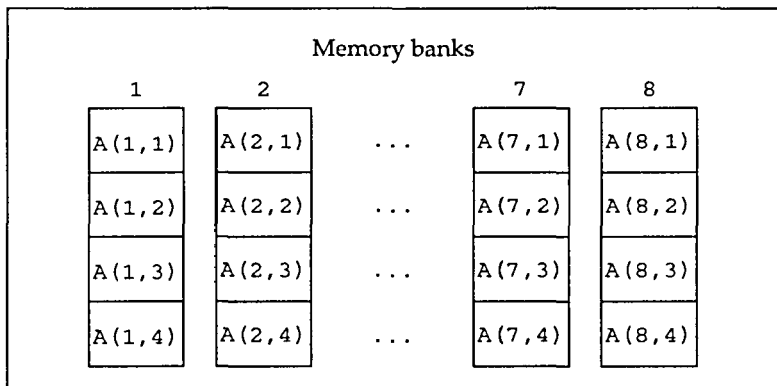
Arrays with a stride of two use only half the memory banks. Arrays with a stride of four use one bank in four. Whenever possible, avoid writing loops with a stride that is a multiple of a power of two. Odd strides give better performance than even strides.

## Multidimensional arrays

FORTRAN arrays are stored in column-major order:  $A(1, 1)$ ,  $A(2, 1)$ ,  $A(3, 1)$ , and  $A(4, 1)$  are stored in contiguous memory locations. In other languages, for example C and Ada, arrays are stored in row-major order:  $A(1, 1)$ ,  $A(1, 2)$ ,  $A(1, 3)$ , and  $A(1, 4)$  are stored contiguously.

Figure 8 shows how a two-dimensional, eight-by-four column-major array is stored in memory with eight-way interleave.

**Figure 8**  
Two-dimensional array stored in eight-way interleaved memory



The loop in the following example processes a two-dimensional array.

```

REAL A(8, 4)
DO 20 I = 1, 8
  DO 10 J = 1, 4
    A(I, J) = A(I, J) + 1
  10 CONTINUE
20 CONTINUE

```

When the inner loop is vectorized, the vector register load and vector store have a stride of eight. Only one memory bank is used in the inner loop, as shown in Figure 8, and eight clock cycles are required to load each element into the vector register.

To avoid bank conflicts, declare the leading index of A to be an odd number, as shown in the following loop.

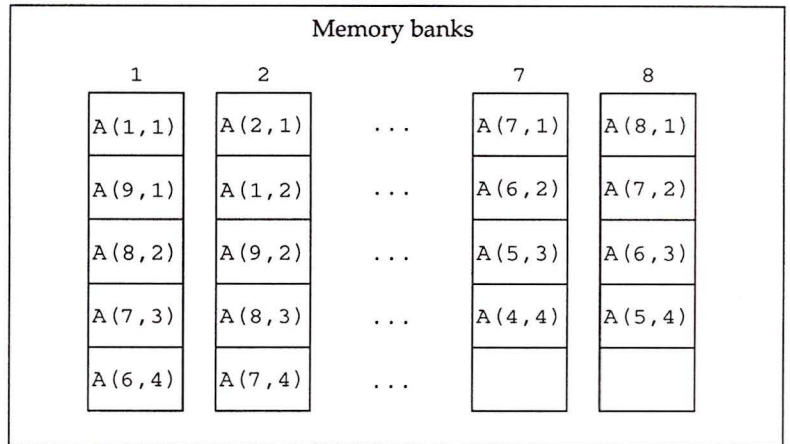
```

REAL A(9, 4)
DO 20 I = 1, 8
  DO 10 J = 1, 4
    A(I, J) = A(I, J) + 1
  10 CONTINUE
20 CONTINUE

```

Figure 9 shows how this array is accessed and arranged in memory. The elements of the ninth row are never used, but they force each column to start in a different memory bank, which resolves bank conflicts.

**Figure 9**  
Leading  
dimension odd:  
no bank conflict



## Partial word accesses

A partial word access requires less than a full word of data. Reading or writing data types such `INTEGER*2`, which occupies a half word, and `CHARACTER`, which occupies a single byte, causes partial memory accesses.

Partial word accesses are inefficient because extra time is required to access the individual bytes of a word. If an array is accessed sequentially, bank conflicts also occur. An `INTEGER*2` array incurs bank conflicts on every other memory access. A `CHARACTER` array incurs bank conflicts on three out of every four memory accesses. Avoid using `CHARACTER` data types in a loop whenever possible.

No matter how sophisticated the compiler is, optimization remains more an art than a science. This chapter presents optimization tricks that FORTRAN programmers have accumulated for optimizing programs to run on the CONVEX C Series supercomputers. The chapter explains underlying principles and offers tips on how to apply these principles to your FORTRAN programs.

## Eliminate unnecessary strip mines

If the compiler determines that the iteration or trip count of a loop is less than or equal to 128, the loop can be executed with a single set of vector operations. In this case, the compiler does not strip mine the loop. Loops are often written with variable trip counts. Unless the compiler can determine the value of the trip-count variable (through constant propagation, for example), the compiler must strip mine the loop to allow for a possible trip count greater than 128. The following example shows a loop with a trip count that varies between 1 and 50.

```
N = GETVAL(N) !returns a value from 1 to 50
DO I = 1, N
  A(I) = B(I) * C(I)
ENDDO
```

In this case, strip mining produces unnecessary overhead. If you know that GETVAL never returns a value greater than 50, you can use the MAX\_TRIPS directive to prevent strip mining the loop, as shown below.

```
N = GETVAL(N)
C$DIR MAX_TRIPS(50)
DO I = 1, N
  A(I) = B(I) * C(I)
ENDDO
```

A value of `MAX_TRIPS` up to 128 stops the compiler from strip-mining a loop. Because you know the trip count cannot exceed 50, use that value. This value permits the compiler to generate a more efficient loop.

---

## Do not vectorize loops with small trip counts

Look for loops with small trip counts. On the C100 and C200 machines, a loop with a trip count less than five is usually not worth vectorizing. The compiler vectorizes loops with trip counts greater than two. For loops with variable trip counts or trip counts between three and five, you can use the `SCALAR` directive to prevent vectorization or the `UNROLL` directive to unroll the loop. The following example shows such a loop.

```
C$DIR SCALAR
      DO I = 1, 4
        A(I) = B(I) * C(I)
      ENDDO
```

The compiler usually vectorizes and strip mines loops with variable trip counts. The compiler strip mines the loop in the following example because it cannot determine the trip count.

```
N = GETVAL(N) ! returns a value of 1, 2, 4,
              ! 8, 16, or 32
      DO I = 1, N
        A(I) = B(I) * C(I)
      ENDDO
```

You can use the `MAX_TRIPS` directive to prevent the compiler from strip mining the loop, but half the time this loop has a trip count so small that it should not be vectorized. You can distribute this loop by hand and use the `SCALAR` directive to eliminate the overhead of a vectorized loop when the trip count is less than or equal to five. An example appears below.

```
      N = GETVAL(N)
      IF (N .GT. 4) THEN
C$DIR MAX_TRIPS(32)
        DO I = 1, N
          A(I) = B(I) * C(I)
        ENDDO
      ELSE
C$DIR SCALAR
        DO I = 1, N
          A(I) = B(I) * C(I)
        ENDDO
      ENDIF
```

Instead of distributing the loop by hand and using the scalar directive, you can use the `SELECT` directive, which tells the compiler to create multiple versions of the loop. The following example shows the use of the `SELECT` directive.

```

N = GETVAL(N)
C$DIR SELECT(4, *, *)
DO I = 1, N
    A(I) = B(I) * C(I)
ENDDO

```

`SELECT` tells the compiler to create multiple versions of the loop, one of which the generated code selects at runtime. The first argument selects the vectorized version if the trip count is greater than or equal to four. The asterisks tell the compiler not to create parallel and vector-parallel versions of the loop.

Because the `SELECT` directive does not require rewriting code, this approach is usually safer and easier. In this case, however, you lose the benefit of the `MAX_TRIPS` directive.

Scalar loops with small constant trip counts can be more efficient if the loops are unrolled. Unrolling replaces a loop with a linear sequence of statements. The example below shows such a loop and how it is unrolled.

Original loop	Loop unrolled
<code>C\$DIR UNROLL</code>	<code>A(1) = A(1) + 1</code>
<code>DO I = 1, 4</code>	<code>A(2) = A(2) + 1</code>
<code>    A(I) = A(I) + 1</code>	<code>A(3) = A(3) + 1</code>
<code>ENDDO</code>	<code>A(4) = A(4) + 1</code>

The `UNROLL` directive works only if the compiler can determine that the trip count is less than five. If a loop has a variable trip count, you can partially unroll it by hand.

## Promote an array

Sometimes it is necessary to promote an array to a higher dimension to vectorize a loop. In the following example, only the `J` loop vectorizes. The compiler is unable to vectorize the `I` loop because of a recurrence. Values assigned to `Q` within the `J` loop depend on values assigned to array `B` by the four preceding statements. Those values of array `B` exist only until the next iteration of the `I` loop. There is a cycle of anti-dependencies between assignments to `B(N)`. This cycle prevents the compiler from distributing the `I` loop.

```

DOUBLE PRECISION GLS(62510)
INTEGER I, J
DOUBLE PRECISION B(4), P(4), Q(4)

DO I = 1, 62500 ! SCALAR
  B(1) = GLS(I+ 1) * P(1) + GLS(I+ 5) * P(2)
>      + GLS(I+ 8) * P(3) + GLS(I+10) * P(4)
  B(2) = GLS(I+ 5) * P(1) + GLS(I+ 2) * P(2)
>      + GLS(I+ 6) * P(3) + GLS(I+ 9) * P(4)
  B(3) = GLS(I+ 8) * P(1) + GLS(I+ 6) * P(2)
>      + GLS(I+ 3) * P(3) + GLS(I+ 7) * P(4)
  B(4) = GLS(I+10) * P(1) + GLS(I + 9) * P(2)
>      + GLS(I+ 7) * P(3) + GLS(I+ 4) * P(4)
  DO J = 1, 4 ! VECTOR
    Q(J) = Q(J) + B(J)
  ENDDO
ENDDO

```

To eliminate the recurrence, promote B to a two-dimensional array, as shown below.

```

DOUBLE PRECISION GLS(62510)
INTEGER I, J
DOUBLE PRECISION B(4, 62510), P(4), Q(4)

DO I = 1, 62500
  B(1,I) = GLS(I+ 1) * P(1) + GLS(I+ 5) *
>      P(2) + GLS(I+ 8) * P(3) +
>      GLS(I+10) * P(4)
  B(2,I) = GLS(I+ 5) * P(1) + GLS(I+ 2) *
>      P(2) + GLS(I+ 6) * P(3) +
>      GLS(I+ 9) * P(4)
  B(3,I) = GLS(I+ 8) * P(1) + GLS(I+ 6) *
>      P(2) + GLS(I+ 3) * P(3) +
>      GLS(I+ 7) * P(4)
  B(4,I) = GLS(I+10) * P(1) + GLS(I+ 9) *
>      P(2) + GLS(I+ 7) * P(3) +
>      GLS(I+ 4) * P(4)

  DO J = 1, 4
    Q(J) = Q(J) + B(J, I)
  ENDDO
ENDDO

```

In the modified code, the calculation of  $B(N, I)$  is independent of the calculation of  $B(N, I+1)$ . These independent calculations permit the compiler to distribute the  $I$  loop, as shown in the following example.

```

DO I = 1, 62500 ! VECTOR
  B(1,I) = GLS(I+ 1) * P(1) + GLS(I+ 5) *
>         P(2) + GLS(I+ 8) * P(3) +
>         GLS(I+10) * P(4)
  B(2,I) = GLS(I+ 5) * P(1) + GLS(I+ 2) *
>         P(2) + GLS(I+ 6) * P(3) +
>         GLS(I+ 9) * P(4)
  B(3,I) = GLS(I+ 8) * P(1) + GLS(I+ 6) *
>         P(2) + GLS(I+ 3) * P(3) +
>         GLS(I+ 7) * P(4)
  B(4,I) = GLS(I+10) * P(1) + GLS(I+ 9) *
>         P(2) + GLS(I+ 7) * P(3) +
>         GLS(I+ 4) * P(4)
ENDDO
DO J = 1, 4          ! Interchanged - SCALAR
  S0 = Q(J)         ! Hoisted register load
  DO I = 1, 62500 ! Interchanged VECTOR
                    ! reduction
    S0 = S0 + B(J, I)
  ENDDO
  Q(J) = V0 ! Sunken register store
ENDDO

```

The compiler vectorizes both distributed parts of the  $I$  loop. The second distributed part is interchanged with the  $J$  loop, which allows the compiler to hoist the load of  $Q(J)$  and sink the corresponding store. These optimizations dramatically reduce the time required for each call to this routine.

---

## Remove a conditional from a loop

A loop with an embedded conditional usually runs slower than a loop without a conditional, even if both loops are vectorized. Some types of conditionals can prevent the compiler from vectorizing a loop. Remove conditional tests from loops whenever possible.

The compiler vectorizes the I loop in the following example. The loop has a series of embedded IF tests that slow it down.

```
DO I = 1, 10000
  IF (I .LE. 2000) THEN
    C1(I) = A1(I) * 2000.0 + COS(A1(I))
    B1(I) = B1(I) * C1(I) * C1(I) * C1(I) *
           C1(I) / A1(I)
  ENDIF
  IF ((I .GT. 2000) .AND. (I .LE. 4000)) THEN
    C1(I) = A1(I) + COS(A1(I))
    B1(I) = B1(I) + C1(I)
  ENDIF
  IF ((I .GT. 4000) .AND. (I .LE. 6000)) THEN
    C1(I) = A1(I) + 2000.0
    B1(I) = B1(I) * B1(I) * B1(I) * B1(I)
  ENDIF
  IF (I .GT. 6000) THEN
    C1(I) = A1(I)
    B1(I) = 1.0
  ENDIF
ENDDO
```

To improve execution speed, remove the conditional by distributing the loop. This produces four distributed parts, shown below.

```
DO I = 1, 2000
  C1(I) = A1(I) * 2000.0 + COS(A1(I))
  B1(I) = B1(I) * C1(I) * C1(I) * C1(I) *
          C1(I) / A1(I)
ENDDO
DO I = 2001, 4000
  C1(I) = A1(I) + COS(A1(I))
  B1(I) = B1(I) + C1(I)
ENDDO
DO I = 4001, 6000
  C1(I) = A1(I) + 2000.0
  B1(I) = B1(I) * B1(I) * B1(I) * B1(I)
ENDDO
DO I = 6001, 10000
  C1(I) = A1(I)
  B1(I) = 1.0
ENDDO
```

The compiler vectorizes each distributed part. The resulting code runs dramatically faster than the original loop.



Inline substitution, or inlining, is the replacement of a subprogram (subroutine or function) call with a copy of the subprogram. Inlining replaces dummy arguments with actual arguments and gives local identifiers unique names.

Inlining can improve performance by eliminating the overhead of a subprogram call and allowing additional optimization. When a subprogram is inlined, the scope of global optimization expands to include both the calling subprogram and the inlined subprogram. The vectorization of loops containing subprogram calls is enhanced, and dead code is eliminated from inlined subprograms.

You can nest inlined subprograms. An inlined subprogram can have another subprogram inlined within it, and this nesting can be carried to any depth. Recursion is not permitted. An inlined function cannot call itself, either directly or indirectly.

---

## When to use inlining

It is seldom advantageous to inline every subroutine in your program. Run your program using CXpa or another profiler. Look for subprograms that are short and frequently called. Inline these subprograms and profile your program again to observe results. Because subprograms within loops inhibit vectorization, inlining them will often allow the loop to vectorize.

Inlining increases a program's compilation time and memory requirements. Avoid inlining large subprograms, no matter how frequently they are called. Inlining large subprograms may prevent the compiler from carrying out other optimizations, negating the advantage of inlining.

---

## How to use inlining

Inlining is done in two steps:

- Create an intermediate language (.fil) file for each subprogram to be inlined.
- Compile the main program using the `-is` option to tell the compiler where to find the .fil files.

---

### Creating .fil files

To create .fil files, follow these steps:

- Place the subprograms to be inlined in individual source files separate from the program MAIN section and other subprograms. You can use the `fsplit` function to do this; see the `fsplit(1F)` man page for details.
- Use the `-il` option when you compile the files containing subprograms to be inlined.

You can use the `-il` option to create .fil files for more than one source file. The compiler generates a separate .fil file for each subprogram in the specified source files. You cannot generate .fil files for a source file containing the main program.

The compiler assigns a name to each .fil file. This name is the name of the subprogram, plus a .fil extension. If a file contains subprograms SUB1, FUNC2, and SUB3, compiling it with the `-il` option creates files `sub1.fil`, `func2.fil`, and `sub3.fil`. The compiler cannot generate a .fil file for a subprogram that has any of the following characteristics:

- Is also compiled with the `-cs` (check subscript) option
- Has a CHARACTER dummy argument
- Uses an adjustable array
- Contains a DATA, SAVE, or NAMELIST statement
- Contains a type statement with initial values
- Contains alternate entry points
- Contains Cray POINTER declarations
- Returns a CHARACTER value
- Contains a statement function

If the compiler cannot generate a .fil file, it issues an error message explaining the reason. The `-il` option cannot be used with the `-c`, `-cs`, or `-S` options. Optimization flags are ignored.

---

## Using the `-is` option

The `-is` option on the `fc` command line tells the compiler to inline subprograms for which `.fil` files exist. The format of this option is

```
-is dir [ -is dir ... ]
```

where *dir* is the name of a directory containing `.fil` files. Use the `-is` option in front of each directory name. Directories are searched in the order specified, and you can specify any number of directories.

The compiler attempts to inline every `.fil` file found in the specified directories. If you do not want to inline specific `.fil` files, delete them or move them to a different directory.

If the compiler cannot inline a `.fil` file, compilation continues. The compiler issues a message explaining why it cannot inline the file and retains the original subprogram call in the finished code.

The compiler cannot inline a subprogram in the following cases:

- A name in a `COMMON` block conflicts with a name in the calling program.
- Data types and sizes in `COMMON` do not match.
- The actual arguments passed to the subprogram do not agree in number or type with the corresponding dummy arguments.
- An array passed to the subprogram does not agree in dimension, lower bound, or upper bound with the corresponding dummy argument.
- A dummy argument is used as a subroutine, but the corresponding actual argument is not a subroutine name.
- A dummy argument is used as a function, but the corresponding actual argument is not a function name.
- A function passed to the subprogram does not agree in type with the dummy argument.
- An intrinsic function passed to the subprogram requires arguments inconsistent with the arguments used in a reference to the corresponding dummy function.

---

## Limits of inline substitution

When using the CONVEX FORTRAN compiler, remember that local variables are static by default. They retain their values between calls. If you compile for re-entrancy, using the `-re` command line option, local variables do not retain their values between calls. In this case, local variables are allocated on the stack. The subprogram gets a fresh copy of the variables on each call, but you can use `SAVE` statements to override this effect for specific variables.

`SAVE` statements prevent inlining. To inline a subprogram that contains `SAVE` statements, put the saved variables into a `COMMON` block and remove the `SAVE` statements.

In CONVEX FORTRAN, variables of inlined subprograms are global only to subprograms in the same source code compilation unit. To make these variables global, place them in a `COMMON` block or place the subprograms that contain them in a single compilation unit.

If you use language-compatibility options when compiling a subprogram for inline substitution, you must use the same options when compiling the program unit that calls it. If you use options that affect data size and layout, you must use the same options when compiling the program unit that calls it.

The source-level debugger, `csd`, and the CONVEX performance analyzer, `CXpa`, do not reflect inlined code in their output. With `csd`, you cannot set breakpoints in inlined code, nor can you access the local symbols of inlined subprograms. You can still run your program under `CXpa` to observe the effects of inline substitution on overall performance.

Optimization can remove instructions, replace them, and change the order in which they execute. In some cases, however, improper optimizations can be performed that produce these effects:

- Different, unexpected, or incorrect results (results that differ from those produced at lower optimization levels or by the original code)
- Code that slows down at higher optimization

If you encounter either of these problems, use this chapter as a guide for troubleshooting.

---

## Note

---

The compiler performs optimizations assuming that the compiled program is valid FORTRAN source. Optimizations done on source that violates certain ANSI standard rules can cause the compiler to generate incorrect code.

---

## Incorrect results

When a program produces different answers at higher optimization levels, look for the following possible causes:

- Erroneous (nonstandard) code
- Floating-point imprecision (roundoff error)
- Misused directives and options
- Compiler limitations

---

### Erroneous code

The most common causes of answers that change with optimization are hidden aliases and invalid subscripts.

## Hidden aliases

Optimizing FORTRAN compilers must assume that subroutine arguments are independent. Page 15-20 of the *American National Standard Programming Language FORTRAN* says,

If a subroutine reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument, neither dummy argument may become defined during the execution of that subprogram.

If a subroutine reference causes a dummy argument to become associated with an entity in a common block in the referenced subprogram or in a subprogram referenced by the referenced subprogram, neither the dummy argument nor the entity in the common block may become defined within the subprogram or within a subprogram referenced by the referenced subprogram.

For example, if a subroutine contains the statements:

```
SUBROUTINE XYZ (A)
COMMON C
```

and is referenced by a program unit that contains the statements:

```
COMMON B
CALL XYZ (B)
```

then the dummy argument A becomes associated with the actual argument B, which is associated with C, which is in a common block. Neither A nor C may become defined during execution of the subroutine XYZ or by any procedures referenced by XYZ.

To interpret the *American National Standard Programming Language FORTRAN* quote properly, it is helpful to understand what the *Standard* means by the phrase, "may become defined."

Variables, array elements, and substrings become defined as follows:

- (1) Execution of an arithmetic, logical, or character assignment statement causes the entity that precedes the equals to become defined.
- (2) As execution of an input statement proceeds, each entity that is assigned a value of its corresponding

- type from the input medium becomes defined at the time of such assignment.
- (3) Execution of a DO statement causes the DO variable to become defined.
  - (4) Beginning of execution of action specified by an implied-DO list in an input/output statement causes the implied-DO-variable to become defined.
  - (5) A DATA statement causes entities to become initially defined at the beginning of execution of an executable program.
  - (6) Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.
  - (7) When an entity of a given type becomes defined, all totally associated entities of the same type become defined except that entities totally associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed.
  - (8) A reference to a subprogram causes a dummy argument to become defined if the corresponding actual argument is defined with a value that is not a statement label value. Note that there must be agreement between the actual argument and the dummy argument.
  - (9) Execution of an input-output statement containing an input/output status specifier causes the specified integer variable or array element to become defined.
  - (10) Execution of an INQUIRE statement causes any entity that is assigned a value during the execution of the statement to become defined if no error condition exists.
  - (11) When a complex entity becomes defined, all partially associated real entities become defined.
  - (12) When both parts of a complex entity become defined as a result of partially associated real or complex entities becoming defined, the complex entity becomes defined.
  - (13) When all characters of a character entity become defined, the character entity becomes defined.

The program below contains hidden aliases that are harder to find than those in the *ANSI Standard* example.

```
PROGRAM ALIAS
INTEGER I
COMMON /DATA/I
I = 666
CALL CONFUSED(I)
END

SUBROUTINE CONFUSED(N1)
DO I = 1, 2
  N2 = 3 * (N1 + 1)
  CALL CALC(N2)
  WRITE(*,*)'Iteration:', I, ', n1 = ', N1
  WRITE(*,*)'Iteration:', I, ', n2 = ', N2
ENDDO
RETURN
END

SUBROUTINE CALC(N)
INTEGER K, N
COMMON /DATA/ K
K = N + 1
RETURN
END
```

In the subroutine `CONFUSED`, the compiler assumes that `N1` is invariant. The right side of the assignment to `N2` appears to be invariant, so the compiler moves the assignment to `N2` out of the loop. When compiled at `-O1` or above, the program produces incorrect answers.

The results of this program compiled and run at optimization levels `-O0` and `-O1` are shown below. Note that the answers are changed at optimization level `-O1`.

```
% fc -O0 alias.f -o O0.out
% O0.out
Iteration: 1, n1 = 2002
Iteration: 1, n2 = 2001
Iteration: 2, n1 = 6010
Iteration: 2, n2 = 6009

% fc -O1 alias.f -o O1.out
% O1.out
Iteration: 1, n1 = 2002
Iteration: 1, n2 = 2001
Iteration: 2, n1 = 2002
Iteration: 2, n2 = 2001
```

The following code shows another example of the hidden alias problem.

```
PROGRAM ALIAS
PARAMETER (N = 500)
REAL A(N) , B(N)

CALL CONFUSED (A, B, A, N)
END

SUBROUTINE CONFUSED (X, Y, Z, N)
INTEGER N
REAL X(N) , Y(N) , Z(N)

DO I = 2, N
  Z(I) = Y(I - 1) + X(I - 1)
ENDDO
RETURN
END
```

In subroutine `CONFUSED`, the compiler assumes that `X` and `Z` are independent. In fact, they are not, and if this erroneous program is compiled at `-O2`, the compiler improperly vectorizes the `DO` loop, producing a faulty executable.

### Invalid subscripts

An array reference in which *any* subscript falls outside declared bounds for that dimension is called an invalid subscript.

Page 5-5 of the *American National Standard programming language FORTRAN* says,

Within a program unit, the value of each subscript expression must be greater than or equal to the corresponding lower dimension bound in the array declarator for the array. The value of each subscript expression must not exceed the corresponding upper dimension bound declared for the array in the program unit. If the upper dimension is an asterisk, the value of the corresponding subscript expression must be such that the subscript value does not exceed the size of the dummy array.

Invalid subscripts are a common cause of wrong answers at higher optimization levels. Invalid subscripts can cause a program to abort.

---

## Floating-point imprecision

When floating-point numbers are rounded off for internal representation or used in vector reductions, incorrect answers may result.

### Roundoff error

The compiler applies normal arithmetic rules to real numbers. It assumes that two arithmetically equivalent expressions produce the same numerical result. Page 6-17 of the *American National Standard Programming Language FORTRAN* says,

Two arithmetic expressions are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions may produce different computational results.

Most real numbers cannot be represented exactly in digital computers. Instead, these numbers are rounded to a floating-point value that can be represented. When optimization changes the evaluation order of a floating-point expression, the results can change. Possible consequences of floating-point roundoff include program aborts, division by zero, address errors, and incorrect results.

Problems with floating-point precision can occur when a program tests the value of a variable without allowing enough tolerance for roundoff errors. To solve the problem, adjust the tolerances to allow for greater roundoff errors or declare the variables to be `DOUBLE PRECISION` instead of `REAL`.

### Vector reductions

Reductions change the order in which an operator is applied to values in a vector. Reductions can change results, particularly if the values in the vector have greatly different magnitudes. If this causes a problem, run the reduction loop as a `SCALAR` loop. Or, try modifying your algorithm.

---

## Misused directives and options

Misused directives are a common cause of wrong answers. Parallelizing a loop that contains a call is safe only if the called routine contains no dependencies that could cause a recurrence.

Do not assume that it is always safe to parallelize a loop that is safe to vectorize. You can safely vectorize any loop that does not contain a backward loop-carried dependency (LCD). You cannot safely parallelize a loop that contains backward or forward LCDs. For more information about LCDs and LIDs, see the "Recurrence" section in Chapter 3.

The MAIN section of the program below initializes A, calls CALC, and displays the new array values. In subroutine CALC, the apparent recurrence on A(I+N) prevents the compiler from vectorizing the I loop.

```
PROGRAM MAIN
REAL A(1025), B(1025)
COMMON /DATA/ A, B
DO J = 1, 1025
  A(J) =J
ENDDO
CALL CALC(1)
DO J = 1, 1025
  WRITE(*,*) J, A(J)
ENDDO
END

OPTIONS -O2
SUBROUTINE CALC(N)
REAL A(1025), B(1025)
COMMON /DATA/ A, B
DO I = 1, 1024
  A(I) = A(I + N) + B(I)
ENDDO
RETURN
END
```

Because you know the value of N is 1, you can use the NO\_RECURRENCE directive, as shown below. This directive tells the compiler to ignore the apparent recurrence and vectorize the I loop.

```
OPTIONS -O2
SUBROUTINE CALC(N)
REAL A(1025), B(1025)
COMMON /DATA/ A, B
C$DIR NO_RECURRENCE
DO I = 1, 1024
  A(I) = A(I + N) + B(I)
ENDDO
RETURN
END
```

Obtaining correct results with vectorization does not imply that correct results will be obtained with parallelization. Using the `FORCE_PARALLEL` directive on this loop, as shown in the following example, is inappropriate. The compiler warns you of the dependency but parallelizes the loop. Because of the forward dependency, the parallel code can produce incorrect results.

```
OPTIONS -O3
SUBROUTINE CALC(N)
REAL A(1025), B(1025)
COMMON /DATA/ A, B
C$DIR FORCE_PARALLEL
DO I = 1, 1024
    A(I) = A(I + N) + B(I)
ENDDO
RETURN
END
```

Routines called by a parallel loop must be compiled for re-entrancy with the `-re` option. Do not assume that variables in a routine compiled with `-re` have been initialized. Local variables in a re-entrant routine must be set to their initial values during each execution of that routine.

---

## Compiler limitations

Compiler limitations can produce faulty optimized code when the source code contains

- Reductions
- Different possible evaluation orderings
- Iterations by zero
- Nondeterminism of parallel execution
- Conditional vectorization
- Replaceable loop test variables

## Reductions

Reductions, which are discussed more fully in Chapter 3, are a special class of recurrence that the compiler knows how to vectorize. An apparent recurrence can prevent the compiler from vectorizing a loop containing a reduction. The loop in the following figure is not vectorized because of an apparent dependency between the reference to  $A(I)$  on line 4 and the assignment to  $A(JA(J))$  on line 5.

```
DATA JA /6, 7, 8, 9, 10/  
DO I = 1, 5  
  DO J = I, 5  
    A(I) = A(I) + B(J) * C(J) !line 4  
    A(JA(J)) = B(J) + C(J)      !line 5  
  ENDDO  
ENDDO
```

A `NO_RECURRENCE` directive placed before the  $J$  loop tells the compiler that the indirect subscript does not cause a true recurrence. This directive also tells the compiler to ignore the reduction on  $A(I)$ . The compiler generates normal vector load, add, and store instructions for the first statement. The resulting code runs fast but produces incorrect answers.

To solve this problem, distribute the  $J$  loop, isolating the reduction from the other statements, as shown in the following example.

```
DATA JA /6, 7, 8, 9, 10/  
DO I = 1, 5  
  DO J = I, 5  
    A(I) = A(I) + B(J) * C(J)  
  ENDDO  
ENDDO  
  
DO J = 1, 5  
  A(JA(J)) = B(J) + C(J)  
ENDDO
```

The apparent recurrence is removed, and both loops vectorize. This problem occurs only if the reduction and the apparent recurrence involve the same variable. If the reduction and the apparent recurrence involve different variables, as in the following example, both reduction and recurrence are handled correctly without your intervention.

```

DATA JD /6, 7, 8, 9, 10/
DO I = 1, 5
C$DIR NO_RECURRENCE
DO J = I, 5
A(I) = A(I) + B(J) * C(J)
D(JD(J)) = D(I) + B(J) + C(J)
ENDDO
ENDDO

```

### Evaluation order

Assumptions the compiler makes about reordering code can sometimes cause answers to change at higher optimization levels. If this happens, use parentheses to force a specific order of evaluation.

### Iterating by zero

If the compiler vectorizes a loop that iterates a variable by zero on each trip, the loop can produce incorrect answers or cause the program to abort. This error can occur when a variable used as an iteration value is accidentally set to zero. If the compiler detects that the variable has been set to zero, the compiler does not vectorize the loop. If the compiler cannot detect the assignment, however, the previously described symptoms occur. The following example shows three loops that iterate by zero.

```

CALL SUB1(0)
...
SUBROUTINE SUB1(ZR)

J = 1
DO I = 1, N
  B(I) = A(J)
  A(J) = C(I)
  J = J + ZR
ENDDO

DO I = 1, N, ZR
  A(I) = B(I)
ENDDO

DO I = 1, N
  B(I) = A(J)
  J = J + ZR
  A(J) = C(I)
ENDDO

```

Because ZR is an argument passed to SUB1, the compiler does not detect that ZR has been set to zero. All three loops vectorize at -O2. The first loop runs, even when vectorized, but produces wrong answers. The other two loops run infinitely when compiled at -O1, but cause the program to abort at -O2.

### **Nondeterminism of parallel execution**

In a parallel program, threads do not execute in a predictable or determined order. If you force the compiler to parallelize a loop when a dependency exists, the results are unpredictable and can vary from one execution to the next.

Because the results depend on the order in which statements execute, the errors are nondeterministic. Unless you are sure that no loop carried dependency exists, it is safer to let the compiler choose which loops to parallelize.

### **Conditional vectorization**

A vectorized loop may fail if the indexes for a conditionally referenced array fall outside the array's bounds. The following code shows an example.

```
DIMENSION A(10000), B(10000), C(10)
DATA A/10*-5, 9990*0/
DO 10 I = 1, 10000
    IF (A(I) .LT. 0) B(I) = A(I) + C(I)
10 CONTINUE
```

### **Test replacement**

When optimizing loops, the compiler often disregards the original induction variable, using instead a variable or value that is referenced more often within the loop. This reduces the execution time of the loop by reducing the number of variables the compiler must track.

The subroutine below contains an example of a loop in which the induction variable is replaced.

```

SUBROUTINE LOOP(N)
  NGD = 0
  IZERO = 0

  DO 8 IRES = 1, N

5     IF(1.GT.NGD) GO TO 8
       IPACK = ((IRES*1024)*64)*64      !LINE 6
       IF((IPACK-IZERO).GE.0) GO TO 5 !LINE 7

8     CONTINUE

  END

```

The IF condition at line 5 in this loop always evaluates to true, but because the test involves a variable (NGD), the compiler assumes that lines 6 and 7 can execute. These lines use the variable IPACK, so the compiler replaces references to IRES, the original induction variable, with suitably equivalent references to IPACK, because it is referenced more often in the loop. The value by which IPACK increases ( $1024*64*64=2^{22}$ ) on each iteration is then said to be the loop's *stride*. The number of times the loop executes is called the *trip count* (N in the example), and the initial value of the induction variable is the *start* value.

Test replacement, a standard optimization at levels -O1 and above, normally does not cause problems. However, when the loop stride is very large, as in the example above, a large trip count can cause the loop limit value ( $stride*trip+start$ ) to overflow.

In the example above, the induction variable is a default (4-byte) integer, which occupies 32 bits in memory. That means if  $stride*trip+start$  ( $2^{22}*N+1$ ) is greater than  $2^{31}-1$ , the value overflows into the sign bit and the computer treats it as a negative number. (If the stride value is negative, the absolute value of  $stride*trip+start$  must be not exceed  $2^{31}$ .) When a loop has a positive stride and the trip count overflows its memory location, the loop executes only once because the limit is now negative (assuming a positive stride) and the termination test fails.

When the trip count is a constant, the compiler can check  $stride*trip+start$  for overflow at compile time and catch this error. However, if the trip count is a variable, no compile-time checking is done, and so large trip and stride combinations can cause the loop to terminate prematurely.

Because the largest allowable value for  $stride * trip + start$  is  $2^{31}-1$ , the start value is 1, and the stride is  $2^{22}$ , the maximum trip count for the loop can be found.

The stride, trip, and start values for a loop must satisfy the following inequality:

$$(stride * trip + start) < 2^{31}$$

The start value is 1, so  $trip$  can be solved for as follows:

$$stride * trip + start < 2^{31}$$

$$2^{22} * trip + 1 < 2^{31}$$

$$trip < 2^9 - 2^{-21}$$

$$trip < 512$$

If you have problems with test replacement and still want to optimize at -O1 or above, restructure the loop to force the compiler to choose a different induction variable.

---

## Slower code

When your program slows down at a higher optimization levels, look for the following causes:

- Misused compiler directives
- Short vector length (small trip count)
- Complicated conditionals in a loop nest

---

### Misused directives

The `SYNCH_PARALLEL` directive tells the compiler to parallelize a loop and insert synchronization code to ensure that dependencies are honored. Synchronization code results in some loss of efficiency. Consequently, using `SYNCH_PARALLEL` is not always profitable. Usually, the compiler can generate more efficient code automatically than with `SYNCH_PARALLEL`. Synchronized code is profitable only if the independent (parallel) part of the code is much larger than the dependent (sequential or synchronized) part.

At -O3, the compiler calculates the optimum strip lengths based on the number of CPUs detected on the compiling machine or the number of CPUs specified by the `-ep` option. The `VSTRIP` and `PSTRIP` directives override the compiler's choice of strip lengths. If you select the wrong strip length, your code may slow down.

---

## Short vector length

When possible, the compiler vectorizes a loop that has more than two iterations. The compiler also vectorizes loops whose iteration count cannot be determined at compile time. A loop that iterates only a few times (three or four, on the CONVEX C Series machines) usually runs faster if the loop is not vectorized. The `SCALAR` directive can prevent the compiler from vectorizing such loops. The `SELECT` directive tells the compiler to generate multiple versions of a loop and code to allow dynamic (runtime) selection of the best version. Using the `SELECT` directive, you can specify optimum cutoff points for scalar, vector, and parallel processing.

---

## Complicated conditionals

Loops containing elaborate conditionals can slow down when they are vectorized.

When the compiler vectorizes a loop containing an `IF - ELSE` construct, the compiler creates a separate vector loop for each clause. Instead of choosing one of these clauses, the program executes both. Results from these two clauses are merged using a conditional mask to produce the final result. If there is an imbalance between the amount of code in each clause, evaluating the smaller clause can result in significant overhead. This overhead is even higher if the smaller clause is branched to more frequently than the larger clause.

A short vector length (small trip count) makes a loop containing complicated conditionals less efficient. Simplify conditionals, remove them from the loop, or use the `SCALAR` directive to prevent vectorization.

---

# The -uo option



The `-uo` option on the `fc` command line instructs the compiler to try potentially unsafe optimizations. The `-uo` option enables the compiler to perform these optimizations:

- Simple strength reductions
- Code motion
- Elimination of type conversions

---

## Simple strength reduction

Chapter 2, “Scalar operations,” describes how the compiler replaces slow operations with faster ones on the assumption that arithmetically equivalent expressions always yield the same results. However, reducing an expression such as  $X/C$  to  $(1/C) * X$  can be unsafe because it can increase roundoff error.

When you use the `-uo` option, the compiler replaces division operations with multiplication. If a possibility of overflow exists, however, the compiler does not perform this optimization.

---

## Code motion

The compiler normally moves an invariant expression out of a loop if the expression is located on a path to all loop exits. When you use `-uo`, the compiler can move an invariant expression out of a loop if the expression does not lie on all paths to all loop exits.

---

## Conversion elimination

When you use the `-uo` option, the compiler eliminates costly type conversions by creating `REAL` induction variables. Consider the loop below.

### Original loop

```
DO I = 1, 100000
  A(I) = I
ENDDO
```

At optimization level -O2, the compiler vectorizes the optimized loop:

### Optimized loop

```
REAL_I = 1.0  
I = 1  
10 A(I) = REAL_I  
REAL_I = REAL_I + 1.0  
I = I + 1  
IF (I .LE. 100000) GOTO 10
```

This appendix describes CONVEX FORTRAN compiler directives. Some directives provide the compiler with information that it cannot deduce on its own. Other directives instruct the compiler to override default conditions that control optimization, vectorization, or parallelization. A directive line has the format

`C$DIR directive [, directive]`

If you specify more than one directive, separate them with commas. A directive must fit on one line; it cannot be continued. A directive can be surrounded by any number of comment lines.

CONVEX FORTRAN supports these optimization directives:

- ASSIGN\_LOCK, FREE\_LOCK
- BEGIN\_ORDER, END\_ORDER
- BEGIN\_SECTION, END\_SECTION
- BEGIN\_TASKS, NEXT\_TASK, END\_TASKS
- FORCE\_PARALLEL
- FORCE\_PARALLEL\_EXT
- FORCE\_VECTOR
- MAX\_TRIPS
- NO\_PARALLEL
- NO\_PEEL
- NO\_PROMOTE\_TEST
- NO\_RECURRENCE
- NO\_SIDE\_EFFECTS
- NO\_VECTOR
- PEEL
- PEEL\_ALL
- PREFER\_PARALLEL
- PREFER\_PARALLEL\_EXT
- PREFER\_VECTOR
- PROMOTE\_TEST
- PROMOTE\_TEST\_ALL
- PSTRIIP
- ROW\_WISE
- SCALAR
- SELECT
- SYNCH\_PARALLEL
- UNROLL
- VSTRIP

Certain combinations of directives are invalid when used within the same program unit or loop, and cause the compiler to issue a warning. Incompatibilities are noted in each directive's description.

A directive associated with a loop affects the loop that immediately follows the directive and does not affect nested loops.

The remaining sections in this appendix describe the directives. A directive's format is shown when it has associated arguments.

---

## **ASSIGN\_LOCK**

### **FREE\_LOCK**

The `ASSIGN_LOCK` directive designates an integer scalar variable or array element to be used as a lock to control ordered and unordered critical regions during parallel optimization. `FREE_LOCK` releases a lock assigned previously by `ASSIGN_LOCK`. Both directives must be in the same program unit as the loop to be parallelized.

The formats of these directives are

```
ASSIGN_LOCK (lockname [, lockname] ...)  
FREE_LOCK (lockname [, lockname] ...)
```

where *lockname* is the name of the variable being assigned or freed. The locking variable must be declared as either `INTEGER*4` or `INTEGER*8`.

The `ASSIGN_LOCK` directive places the lock in the unlocked state. The lock is set or reset as it is used during the execution of critical regions.

---

## **BEGIN\_ORDER**

### **END\_ORDER**

The `BEGIN_ORDER` directive identifies an ordered critical region. An ordered critical region begins with `BEGIN_ORDER` and ends with `END_ORDER`.

An ordered critical region is a section of code in which execution of one instance of the section must be delayed until previous instances have been executed. Loop iterations must be performed in correct sequence.

The `BEGIN_ORDER` and `END_ORDER` directives must be placed in the same program unit as the `ASSIGN_LOCK` and `FREE_LOCK` directives.

The formats of these directives are

```
BEGIN_ORDER (lockname)
END_ORDER
```

where *lockname* is the name of a locking variable that has been assigned with ASSIGN\_LOCK. The following example shows how to use these directives.

```
C$DIR ASSIGN_LOCK (ILOCK)
C$DIR FORCE_PARALLEL
      DO I = 1, N
          A(I) = B(I) * C(I)
C$DIR BEGIN_ORDER (ILOCK)
          IF ( A(I) .LT. 0 ) THEN
              D(I) = D(I - 1) + E(I)
          ENDIF
C$DIR END_ORDER
      ENDDO
C$DIR FREE_LOCK (ILOCK)
```

---

## **BEGIN\_SECTION**

## **END\_SECTION**

The BEGIN\_SECTION directive identifies an unordered critical region. An unordered critical region begins with a BEGIN\_SECTION directive and ends with an END\_SECTION directive.

An unordered critical region is a section of code in which execution of only one instance of the section is allowed at any one time. Loop iterations need not be performed in sequence. The formats are

```
BEGIN_SECTION (lockname)
END_SECTION
```

where *lockname* is the name of a locking variable that has previously been assigned by the ASSIGN\_LOCK directive.

An example of how to use this pair of directives appears below.

```
C$DIR ASSIGN_LOCK (JLOCK)
C$DIR FORCE_PARALLEL
      DO I = 1, N
          A(I) = B(I) * C(I)
C$DIR BEGIN_SECTION (JLOCK)
      IF ( A(I) .LT. 0 ) THEN
          K = K + 1
      ENDIF
C$DIR END_SECTION
      ENDDO
C$DIR FREE_LOCK (JLOCK)
```

The `BEGIN_SECTION` and `END_SECTION` directives must be placed in the same program unit as the `ASSIGN_LOCK` and `FREE_LOCK` directives.

---

## **BEGIN\_TASKS**

### **NEXT\_TASK**

## **END\_TASKS**

A task is a sequence of linear code that can be executed in parallel with other tasks. The `BEGIN_TASKS` directive identifies a sequence of tasks for independent, parallel execution. The sequence of tasks ends with `END_TASKS`. `NEXT_TASK` precedes each task except the first.

The following example shows how to use tasking directives.

```
C$DIR BEGIN_TASKS
      statement
      ...
C$DIR NEXT_TASK
      statement
      ...
C$DIR NEXT_TASK
      statement
      ...
C$DIR END_TASKS
```

You can specify a maximum of 255 tasks between a `BEGIN_TASKS` and an `END_TASKS` directive.

---

## FORCE\_PARALLEL

The `FORCE_PARALLEL` directive is effective only if you specify the `-O3` compiler option. The directive tells the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. You can use this directive on a loop whether or not the loop contains calls, but it may not be safe to do so.

Certain actual dependencies, such as from one scalar to another, cause the compiler to ignore the `FORCE_PARALLEL` directive.

`FORCE_PARALLEL` does not allow the compiler to interchange or distribute outer loops for vectorization. To enable those optimizations, use `FORCE_PARALLEL_EXT`.

---

### Caution

---

**This directive causes the compiler to ignore any apparent dependencies between iterations. When you use this directive on a loop, you may not get correct results. Check answers generated by the parallelized code.**

If you use this directive with `SCALAR` or `NO_PARALLEL`, a warning is issued. Also, an error occurs when you use `FORCE_PARALLEL` and another parallelizing directive in the same loop nest.

An example of how to use `FORCE_PARALLEL` appears below.

```
C$DIR FORCE_PARALLEL
      DO I = 1, N
        CALL SUB (A, I)
      ENDDO
```

---

## FORCE\_PARALLEL\_EXT

The `FORCE_PARALLEL_EXT` directive is effective only if you specify the `-O3` compiler option. This directive forces the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. You can use this directive on a loop whether or not the loop contains calls.

If you specify `FORCE_PARALLEL_EXT` and `FORCE_VECTOR` for the same loop, the compiler first vectorizes the loop and then parallelizes the resulting strip-mine loop.

`FORCE_PARALLEL_EXT` allows the compiler to interchange outer loops for vectorization.

---

**Caution**

---

This directive causes the compiler to ignore any apparent dependencies between iterations. When you use this directive on a loop, you may not get correct results. Check answers generated by the parallelized code.

If you use this directive with `SCALAR` or `NO_RECURRENCE`, an error occurs. Also, an error occurs when you use `FORCE_PARALLEL_EXT` and another parallelizing directive in the same loop nest.

---

**FORCE\_VECTOR**

The `FORCE_VECTOR` directive forces the compiler to vectorize the loop that follows, regardless of apparent recurrences. It is possible to use `FORCE_VECTOR` on a loop that the compiler would not fully vectorize without the directive and get incorrect answers because the directive causes the compiler to ignore dependencies.

Use this directive only with fully vectorizable loops. If you specify `FORCE_VECTOR` and `FORCE_PARALLEL_EXT` for the same loop, the compiler first vectorizes the loop and then parallelizes the resulting strip-mine loop.

---

**Caution**

---

This directive causes the compiler to ignore any apparent dependencies between iterations. When you use this directive on a loop, you may not get correct results. Check answers generated by the vectorized code.

If you use `FORCE_VECTOR` with `NO_RECURRENCE` or `SCALAR`, a warning is issued. An error occurs when you try to use `FORCE_VECTOR` and another vectorizing directive in the same loop nest.

---

**MAX\_TRIPS**

The `MAX_TRIPS` directive tells the compiler that the loop will execute no more than the specified number of times. The format of this directive is

`MAX_TRIPS (n)`

where the value of  $n$  is less than or equal to the vector register length of 128. You can use this directive to prevent the compiler from strip mining the loop. Eliminating strip mining results in more efficient code generation when the maximum trip count is less than or equal to 128.

---

## NO\_PARALLEL

The `NO_PARALLEL` directive tells the compiler not to parallelize the loop immediately following the directive. The directive does not prevent vectorization of the loop.

If `NO_PARALLEL` and `NO_VECTOR` both precede a loop, the result is the same as if `SCALAR` were used.

---

## NO\_PEEL

The `NO_PEEL` directive prevents the compiler from applying loop boundary value peeling to the loop that immediately follows. This directive overrides boundary level peeling at all levels—default, `-peel`, and `-peelall`. See Chapter 3, “Vector optimization,” for more information.

---

## NO\_PROMOTE\_TEST

The `NO_PROMOTE_TEST` directive prevents the compiler from applying test promotion to the loop that immediately follows. This directive overrides test promotion at all levels—default, `-ptst`, and `-ptstall`. See Chapter 3, “Vector optimization,” for more information.

---

## NO\_RECURRENCE

The `NO_RECURRENCE` directive instructs the compiler to disregard any recurrence in a loop. If nothing else impedes vectorization, the compiler vectorizes the loop.

Place the directive immediately before a `DO` statement or a labeled statement that begins a loop. You can include comment lines between the directive and the beginning of the loop.

`NO_RECURRENCE` does not affect recurrences caused by a nested `DO` loop. You can, however, use the directive on each loop in a nest to provide maximum opportunity to improve the nest’s performance.

When you use `NO_RECURRENCE` and the compiler finds a recurrence, the compiler breaks the recurrence by removing one or more dependencies of the cycle. In the following example, if `J` is positive, no recurrence exists.

```
C$DIR NO_RECURRENCE
      DO 10 I = 1, N
10      A(I) = A(I + J)
```

The compiler always accepts a `NO_RECURRENCE` directive on an apparent recurrence involving an array element; the compiler always ignores a `NO_RECURRENCE` directive on an apparent recurrence involving a scalar. In the latter case, the compiler knows that a recurrence exists.

---

## Caution

---

**Incorrect results can occur if you mistake a real recurrence for an apparent one. Always test vector results against scalar results to determine whether a recurrence is real or apparent.**

For more information about recurrence, see Chapter 9, "Limits of optimization."

---

## NO\_SIDE\_EFFECTS

The `NO_SIDE_EFFECTS` directive tells the compiler that the specified functions do not modify the value of an argument, or common variable, read or write, or call another routine. The format of this directive is

```
NO_SIDE_EFFECTS [( func [, func ] ... )]
```

The argument *func* specifies one or more user-defined functions.

This directive allows the compiler to remove a function call during scalar optimization if the call occurs in an expression assigned to an unused scalar variable. The compiler removes the function call because the function has no side effects. Such optimization opportunities usually arise after the compiler performs other optimizations. These opportunities rarely occur in the original source text.

Place the directive before the call to the named function. Use the directive if the compiler gives the advisory message "More optimization is possible if this function call has no side effects," and the function is really one with no side effects. If there are no arguments, the directive applies to all functions following the directive. An example of how to use this directive appears below.

```
C$DIR NO_SIDE_EFFECTS (F1, F2)
...
X = Y * F1(5, Z) - W
...
```

A function call with no side effects is invariant with respect to a loop under these conditions:

- The function call's arguments do not vary within the loop and the function call can be moved out of the loop.
- The function call does not modify a common variable.
- The function call does not perform I/O.

---

## **NO\_VECTOR**

The `NO_VECTOR` directive tells the compiler not to vectorize the loop immediately following the directive. This directive does not prevent parallelization.

If `NO_PARALLEL` and `NO_VECTOR` both precede a loop, the result is the same as if `SCALAR` were used.

---

## **PEEL**

The `PEEL` directive allows the compiler to peel the loop immediately following the directive, expanding the code without bound. See Chapter 3, "Vector optimization," for more information.

---

## **PEEL\_ALL**

The `PEEL_ALL` directive allows the compiler to peel the loop immediately following the directive, expanding the code without bound. See Chapter 3, "Vector optimization," for more information.

---

## **PREFER\_PARALLEL**

The `PREFER_PARALLEL` directive tells the compiler to parallelize the loop immediately following the directive only if it appears safe. The compiler checks for actual loop-carried dependencies. If the compiler finds no dependencies, the compiler parallelizes the loop.

This directive prevents the compiler from interchanging and distributing outer loops for vectorization, whereas `PREFER_PARALLEL_EXT` does not.

---

## **PREFER\_PARALLEL\_EXT**

The `PREFER_PARALLEL_EXT` directive tells the compiler to parallelize the loop immediately following the directive only if it appears safe. The compiler checks for actual loop-carried dependencies. If the compiler finds no dependencies, the compiler parallelizes the loop.

This directive allows the compiler to interchange outer loops for vectorization. To vectorize a loop and parallelize the resulting strip-mine loop, use `PREFER_PARALLEL_EXT` and `PREFER_VECTOR` at optimization level `-O3`.

---

## PREFER\_VECTOR

The `PREFER_VECTOR` directive tells the compiler to vectorize the loop immediately following the directive only if it appears safe. The compiler checks for actual recurrences. If the compiler finds no recurrences, the compiler tries to interchange the loop so that it is the innermost loop and then it tries to vectorize the interchanged loop.

---

## PROMOTE\_TEST

The `PROMOTE_TEST` directive allows the compiler to promote tests out of the loop immediately following the directive, replicating code beyond the default conservative limit, but not without bound. See Chapter 3, “Vector optimization,” for more information.

---

## PROMOTE\_TEST\_ALL

The `PROMOTE_TEST_ALL` directive allows the compiler to promote tests out of the loop immediately following the directive, replicating code without bound. See Chapter 3, “Vector optimization,” for more information.

---

## PSTRIP

The `PSTRIP` directive tells the compiler to strip mine the parallel loop immediately following the directive. The compiler strip mines the loop according to the strip-mine length you specify. The format of this directive is

`PSTRIP (integer_constant)`

where *integer\_constant* specifies the strip-mine length.

Parallel strip mining combines loop iterations into groups of  $n / (2 * ep)$ , where *ep* is the argument to the directive and *n* is the actual loop trip count. A single thread executes each group. Parallel strip mining occurs only at optimization level `-O3`. If you do not use `PSTRIP`, the compiler selects a default strip-mine length appropriate for the architecture of the machine for which you are compiling.

The default number of loop iterations in a group is one. The number of loop iterations in a group is also one when the argument to `-ep` is one. At `-O3` when the argument to `-ep` is more than one, the compiler determines the maximum strip-mine length with the formula

$$\max(\min((n + ep - 1) / ep), 128), 8)$$

You cannot use `PSTRIP` with vector loops.

When the number of iterations is small (less than 32), a `PSTRIP` value of one usually gives the best results.

---

## ROW\_WISE

FORTTRAN stores arrays in column-major order. Reversing the order of subscripts so that the array is accessed through contiguous rather than noncontiguous memory can improve the efficiency of memory accesses. `ROW_WISE` tells the compiler to reverse the dimensions of the designated arrays. The format of this directive is

```
ROW_WISE (array_name [, array_name ... ])
```

The following rules apply to using the `ROW_WISE` directive:

- Implicit array I/O, such as `READ (5, *) A`, is not allowed for arrays that appear in a `ROW_WISE` directive.
- The array appears transposed when viewed in the debugger.
- If you apply the `ROW_WISE` directive to a dummy argument, the actual argument must appear in a `ROW_WISE` directive within the caller.

The following example shows a code segment containing a candidate for the `ROW_WISE` directive.

```
DIMENSION A(4, 1000)
DO I = 1, 4
  DO J = 1, 1000
    A(I, J) = 0
  ENDDO
ENDDO
```

Although the example vectorizes, it runs slowly because the array is being accessed through noncontiguous memory. The array is now being accessed from contiguous memory, increasing the execution speed. Using `ROW_WISE` makes the accesses contiguous.

---

## SCALAR

The `SCALAR` directive prevents the `DO` loop following the directive from being vectorized or parallelized. The compiler can vectorize or parallelize the body of the loop if it can interchange an outer loop with the `SCALAR` loop.

The `SCALAR` directive is useful when the loop's iteration count is too low for the overhead involved in setting up vectorization or when you must obtain numerical results identical to those of a scalar loop. You can also use this directive to prevent the compiler from interchanging loops. In some cases, the compiler cannot determine the iteration counts of loops and might not choose the best loops to interchange.

The results of a vectorized loop can differ from its scalar equivalent. For example, floating-point sum and product reduction operators can give different answers because the underlying hardware does not process the operands in sequential order.

In the following example, the compiler normally interchanges the `I` and `J` loop so that elements of `A`, `B`, and `C` are accessed contiguously. The `SCALAR` directive ensures that the loop of greater iteration count is retained as the innermost loop.

```
C$DIR SCALAR
      DO 10 I = 1, N      ! (where N = 2)
          DO 10 J = 1, M ! (where M = 1000)
10     A(I,J) = B(I, J) + C(I, J)
```

In the following example, neither iteration count is sufficient to warrant vectorizing the loops.

```
C$DIR SCALAR
      DO 10 I = 1, N      ! (where N = 2)
C$DIR SCALAR
          DO 10 J = 1, M ! (where M = 2)
10     A(I, J) = B(I, J) + C(I, J)
```

---

## SELECT

The `SELECT` directive tells the compiler to generate multiple versions of a loop that will select runtime code based on specified trip, or iteration, counts. The compiler can generate up to four versions of a loop: scalar, vector, parallel, and parallel-vector. The format of this directive is

```
SELECT (vtrip, ptrip, pvtrip)
```

The arguments *vtrip*, *ptrip*, and *pvtrip* specify the trip count at which to select vector, parallel, or parallel-vector execution, respectively, for the loop following the directive. Parallel-vector execution implies that the loop is vectorized and the resulting strip-mine loop is parallelized.

If you omit a trip count by using two adjacent commas, the compiler uses a default value. If you place an asterisk in place of a trip count, the compiler does not generate code for the corresponding mode.

If the actual trip count is less than or equal to the smallest specified trip count in the directive, the loop runs scalar. If the actual trip count is greater than the largest trip count, the loop runs in the mode of the largest trip count. For example, suppose you precede a loop with this statement:

```
C$DIR SELECT (10,4,200)
```

The loop runs scalar if the actual trip count is one to four, and parallel if the trip count is five to 10. The loop runs vector if the trip count is 11 to 200, and parallel-vector if the trip count is greater than 200.

The statement

```
C$DIR SELECT (*,*,*)
```

is equivalent to

```
C$DIR SCALAR
```

---

## SYNCH\_PARALLEL

The `SYNCH_PARALLEL` directive is effective only if you specify the `-O3` compiler option. This directive tells the compiler to generate code that executes the loop that follows in parallel. However, instead of ignoring dependencies, the compiler inserts synchronization code that causes the dependencies to be honored at runtime.

Without specific directives, the compiler vectorizes any dependency-free part of the loop; this normally produces superior results. However, if a loop contains much code that is conditionally executed, you might want to parallelize the loop with the `SYNCH_PARALLEL` directive, particularly if all the dependencies are in seldom executed branches.

On a machine with four processors, the loop in the following example might run faster parallelized and synchronized than if it were partially vectorized and the recurrence placed in a scalar, nonparallel loop.

```
C$DIR SYNCH_PARALLEL
DO I = 1, 32
  IF (A(I) .LT. 0) THEN
    A(I) = A(I - 1) + B(I)
    D(I) = E(I) * F(I)
  ENDIF
ENDDO
```

---

## UNROLL

The `UNROLL` directive is effective only if you specify the `-O2` or `-O3` compiler option. `UNROLL` reduces loop overhead by replicating the body of the loop following the directive. Complete unrolling is performed only on loops that can be vectorized. Partial unrolling is performed on loops that cannot be vectorized.

To be eligible for unrolling, a loop must contain no internal branching and have an iteration count that the compiler can determine. The compiler unrolls a loop completely only if it knows that the iteration count is less than five; otherwise, the compiler partially unrolls the loop. Complete unrolling occurs before vectorization. Partial unrolling occurs after vectorization.

---

## VSTRIP

The `VSTRIP` directive tells the compiler to strip mine the vector loops immediately following the directive. This directive is especially useful for automatically parallelized vector loops (loops that are vectorized and run with the outer strip parallel).

The format of this directive is

```
VSTRIP (integer_constant)
```

where *integer\_constant* specifies the strip mine length, which must be less than or equal to 128.

Vector strip mining executes a loop in strips of 128 elements by default, and the parallel outer loop runs iterations of the vector loop in parallel. If you do not specify `VSTRIP`, the compiler selects a default value of 128 for the strip-mine length.

`VSTRIP` overrides the compiler default and specifies a shorter strip-mine length. The shorter strip creates more iterations of the strip-mine loop so that the loop can be effectively parallelized. To determine the maximum strip-mine length when the argument to `-ep` is more than one, the compiler uses the formula

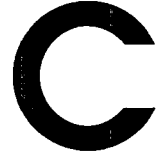
$$\max(\min((n + ep - 1) / ep), 128), 8)$$

where `ep` is the argument to the directive and `n` is the actual loop trip count.

The actual strip length is the smaller of the number of iterations remaining to be processed or the maximum length of a strip determined with the formula (either the default or from the directive).

---

# Vector operations



This appendix describes the vector instruction set on CONVEX computers. These descriptions can help you create efficient code. You do not need to know assembly language to read and understand this chapter. For more detailed information about vector operations and hardware, see the *CONVEX C Series Architecture Reference Manual*.

---

## Vector hardware

Four types of registers are used in vector operations:

- Vector-accumulator (V) register
- Vector-length (VL) register
- Vector-stride (VS) register
- Vector-merge (VM) register

---

### Vector-accumulator register

A vector register can hold up to 128 64-bit elements. These elements can be integer or floating-point data. Data must be of uniform size and precision. The vector register is used to store arrays of operands. CONVEX C Series machines have eight vector registers.

---

### Vector-length register

The CONVEX C Series machines have one vector-length register. The value in a VL register is the number of elements used in subsequent vector operations.

---

### Vector-stride register

The 32-bit vector-stride register is used by the load and store instructions. The value in the VS register is the number of bytes from one element of an array in memory to the next sequential element. Strides can be either positive or negative.

---

## Vector-merge register

The vector-merge (VM) register holds a 128-bit mask used for compress, expand, operate-under-mask, and merge instructions. The VM register also stores the results of a vector comparison. If the comparison of corresponding elements in two vector registers is true, the corresponding bit in the VM register is set. Otherwise, the corresponding bit is cleared.

The VM register is often used for these operations:

- Population count (number of successful compares)
- Sparse vector manipulation
- Array compression, expansion, and merging
- Vector clipping

---

## How the CONVEX architecture works

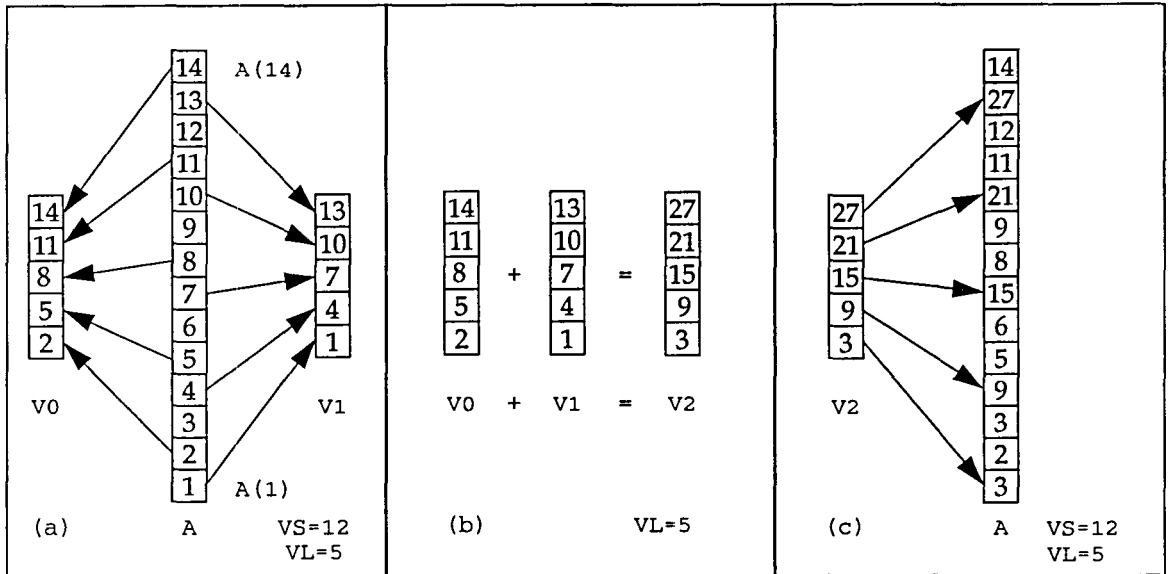
To see how the vector hardware works, consider the vector operation in the following example.

```
INTEGER A(14), I
DO I = 1, 14, 3
  A(I) = A(I + 1) + A(I)
ENDDO
```

The code increments every third element of the array and uses the VS, VL, and V registers. Figure 10 shows the vector operations on array A.

Figure 10

Vector operations for  $A(I) = A(I + 1) + A(I)$



In (a), the CPU sets the vector-stride register (VS) to 12 (the number of bytes between elements of the array). In (b), the CPU has set the vector-length register (VL) controlling the operation to 5. The VL register controls the loading of elements from array A into vector registers V0 and V1.

The CPU adds the contents of vector registers V0 and V1 and stores the result in V2.

In (c), the CPU stores elements of V2 back into array A.

---

## Vector instruction set

This section describes some of the assembly language instructions used in vector operations. Assembly-language listings are provided to show how certain FORTRAN statements are vectorized in assembly language. For a complete list of the vector instruction set, see the *CONVEX C Series Architecture Reference Manual*.

---

### Vector load

The vector load instruction loads the contents of an array stored in memory into a vector register. The data types are byte, half-word, word, and long-word. The VS register contains the byte separation of each element that is loaded into the vector register, and the VL register contains the number of array elements to be loaded.

For example, suppose the FORTRAN code on the left in the following example is vectorized.

FORTRAN	Assembly language
INTEGER*4 A(25)	ld.w #25,VL
	ld.w #4,VS
DO I = 1, 25	ld.w A,v0
A(I) = A(I) + 4	
ENDDO	

The assembly-language code required to load A into a vector register appears on the right.

The first statement loads the length of array A, which is 25, into VL. The second statement loads 4 into VS because each element in the array requires four bytes for storage and the loop's stride is one. The last statement loads the contents of array A into vector register v0.

The following example shows another example of vector load.

FORTRAN	Assembly language
REAL*8 B(100)	ld.w #5,VL
	ld.w #160,VS
DO I = 1, 100, 20	ld.l B,v0
B(I) = B(I) + 8.0	
ENDDO	

The FORTRAN code on the left generates the vectorized assembly-language code on the right.

VL contains 5 because only five elements of array B are modified. It is unnecessary to load all the elements of B into the vector. Similarly, VS contains 160, because each element requires eight bytes of storage and the loop's stride is 20. The last statement loads five elements of array B into vector register V0.

Some operations that appear to require a load statement use other instructions instead, as shown in the example below.

<b>FORTRAN</b>	<b>Assembly language</b>
INTEGER*4 C(100)	ld.w #5,s0
	ld.w #100,VL
DO I = 1, 100	ld.w #4,VS
C(I) = 5	ste.w s0,C
ENDDO	

The result is a repeated store of a scalar register. The assembly-language instruction for store scalar extended is *ste*. This instruction uses the VS and VL registers in the same way the vector load instruction does: VL specifies the length of the array, and VS specifies the number of bytes between each array element that is accessed.

## Vector store

The vector store instruction stores the contents of a vector register into an array in memory. The VS register contains the byte separation of each element stored, and the VL register contains the number of array elements in the vector register.

Following is an example of vector store.

<b>FORTRAN</b>	<b>Assembly language</b>
INTEGER*4 B(100) ,C(100)	ld.w #100,VL
	ld.w #4,VS
DO I = 1, 100	ld.w B,v0
C(I) = B(I)	st.w v0,C
ENDDO	

The VL register contains 100 because 100 elements are loaded and stored. The VS register contains 4 because each element requires four bytes for storage and the loop's stride is one. In the example, the vector store and vector load operations use the same VL and VS values.

The following example shows another vector store.

<b>FORTRAN</b>	<b>Assembly language</b>
INTEGER*4 B(100), C(100)	ld.w #50,VL
	ld.w #4,VS
DO I = 1, 50	ld.w B,v0
C(I * 2) = B(I)	ld.w #8,VS
ENDDO	st.w v0,C+4

In this example, the VS register is increased to 8 because only every other element of array C is modified. The loop's stride is 2, and each element of array C requires four bytes for storage. Because the destination of the vector store operation starts at the second element of array C, its base address is increased by one element, or four bytes.

---

## Binary vector operators

Four binary operators used in vector arithmetic are addition, division, multiplication, and subtraction. Additional binary operators used for logical operations are `and`, `or`, and `xor`. Both operands of these operators can be vectors, or one can be a scalar and the other a vector. All operators use the VL register to determine the number of vector elements to use in computations.

The following example shows the use of the vector add operator.

<b>FORTRAN</b>	<b>Assembly language</b>
INTEGER*4 B(100), C(100)	ld.w #50,VL
	ld.w #4,VS
DO I = 1, 50	ld.w C,v0
C(I) = C(I) + B(I)	ld.w B,v1
ENDDO	add.w v0,v1,v2
	st.w v2,C

Arrays B and C are loaded into vector registers, which are added together. The result is stored in a third vector register. The fourth and fifth or fifth and sixth statements can be chained together because they map to different functional units.

The FORTRAN code in the following example computes the product of a vector and a scalar.

FORTRAN	Assembly language
INTEGER*4 C(100)	ld.w #100,VL
	ld.w #8,s0
DO I = 1, 100	ld.w #4,VS
C(I) = C(I) * 8	ld.w C,v0
ENDDO	mul.w v0,s0,v1
	st.w v1,C

Array C is loaded into v0, and v0 is multiplied by the scalar register s0. The result is stored in v1 and then returned to array C.

---

## Vector reductions

Reduction operations reduce a vector to a scalar. A reduction operation requires two inputs: a scalar register and a vector register. A scalar input is provided so that reduction operators can be performed for vectors greater than 128 elements.

Mathematically, reduction operations are the sum reduction (sum) and multiply or product reduction (prod). Additional reduction operations are provided to implement the FORTRAN MAX and MIN intrinsics, as well as reduction using logical operators such as .AND., .OR., and .NEQV.

The example below generates a sum reduction.

FORTRAN	Assembly language
INTEGER*4 C(100)	ld.w #0,s0
	ld.w #100,VL
ISUM = 0	ld.w #4,VS
DO I = 1, 100	ld.w C,v0
ISUM = ISUM + C(I)	sum.w v0
ENDDO	st.w s0,ISUM

During a vector reduction, a vector register is paired with a scalar register (Vi is paired with Si). In this example, s0 is the scalar register that corresponds to ISUM, and v0 is the vector that is reduced. This statement

```
sum.w v0
```

can be replaced with

```
sum.w s0
```

Both statements produce the same results.

The FORTRAN code in the following example computes a vector's maximum.

FORTRAN	Assembly language
INTEGER*4 C(100), CMAX	ld.w C(1), s0
	ld.w #100, VL
CMAX = C(1)	ld.w #4, VS
DO I = 1, 100	ld.w C, v0
CMAX = JMAX0(CMAX, C(I))	max.w v0
ENDDO	st.w s0, CMAX

This code performs the way the code in the sum reduction shown above does, except the vector's maximum is returned.

---

## Chaining

By chaining vector operations, the CPU can use the output of one vector instruction as input for the next. Addition and multiplication can be chained so that an addition begins while the products of two vectors are being computed. These concurrent, or pipelined, events greatly improve performance.

In the following example, a dot-product operation requires the sum of a series of products.

```
INTEGER D(100), N(100), A(100), I, SUM
SUM = 0
DO I = 1, 100
    SUM = SUM + D(I) * A(I)
ENDDO
```

The assembly language for the dot-product operation is shown below.

```
ld.w #0, s0
ld.w #100, VL
ld.w #4, VS
ld.w A, v1
ld.w N, v2
mul.w v2, v1, v0
st.w v0, D
sum.w v0
st.w s0, SUM
```

In this example, the summation is chained with the multiplication. Pipelining uses multiple functional units of the CPU to perform a specific set of operations, and the functional units allow the multiplication and addition operations to overlap.

---

## Vector comparisons

The three vector comparison instructions are less-than, less-than-or-equal, and equal. All other logical operators are obtained by taking the complement of these three instructions. For example, greater-than is the complement of less-than-or-equal.

The result of a vector comparison is stored in the vector-merge register. This register has 128 bits, each one corresponding to an element in a vector register. If the comparison of two elements is true, the corresponding bit in the VM register is set; otherwise the bit is cleared. The VM register controls other vector operations as described below in the section, "VM operations under mask— C2 and C3."

Consider the vector comparison in the following example.

<b>FORTRAN</b>	<b>Assembly language</b>
INTEGER*4 A(100), B(100)	ld.w #100,VL
	ld.w #4,VS
DO I = 1, 100	ld.w A,v0
IF(A(I).LE.B(I))J = I	ld.w B,v1
ENDDO	le.w v0,v1

The arrays are loaded into vectors, and the vectors are compared.

---

## Vector operations under mask—C2 and C3

The CONVEX C2 and C3 Series computers can perform vector operations under mask. The CONVEX C1 Series computers can perform vector operations and mask operations, but multiple vector instructions must replace an individual vector operation under mask on the CONVEX C2 and C3 Series computers.

Most vector operations can operate under mask. A vector-merge register bit is associated with each vector register element. When an operation is performed under mask, each element is either included or excluded from the operation based on the state of its corresponding VM bit.

In this mode, the bit of the VM register corresponding to each vector element is examined to either enable or disable that vector element from the operation.

The two forms of vector operations under mask are

- **True**—Elements with VM bit equal to one are included. Instructions of this type have a `.t` suffix, such as `add.w.t`.
- **False**—Elements with VM bit equal to zero are included. Instructions of this type have a `.f` suffix, such as `div.b.f`.

The statement

```
add.w    v0,v1,v2
```

adds all elements (restricted by vector length) of `v0` and `v1`, placing the results in `v2`.

The statement

```
add.w.t  v0,v1,v2
```

adds only elements whose corresponding VM bits are one. Elements of `v2` whose corresponding VM bits are zero remain unmodified.

The complement of VM bits is used for `.f`, as in the statement

```
add.w.f  v0,v1,v2
```

This version operates only on vector elements whose corresponding VM bits are 0.

For the remaining examples of operations under mask, assume these values before instructions are executed:

```
v0 = 0 1 2 3 4 5    VL = 6
v1 = 6 7 8 9 2 3    VM = 0 1 1 0 0 1
v2 = 5 5 5 5 5 5
```

The statement

```
add.w.t  v0,v1,v2
```

produces

```
v2 = 5 8 10 5 5 8
```

The statement

```
add.w.f  v0,v1,v2
```

produces

```
v2 = 6 5 5 12 6 5
```

The following FORTRAN code is an example of using operations under mask.

FORTRAN	Assembly language
DO I = 1, 100	ld.w A, v0
IF (A(I) .EQ. B(I)) THEN	ld.w B, v1
C(I) = D(I)	eq.w v0, v1
ENDIF	ld.w D, v0
ENDDO	st.w.t v0, C

Ignoring the length and stride setup, the code on the left can be vectorized with the assembly-language code shown on the right.

---

## Vector-merge register operations

The merge, mask, compress, and expand operations use the vector-merge register to control the selection of elements in the vector operands.

### Merge and mask

The merge and mask instructions take two operands and produce a vector as the result. The two operands can be two vectors or a vector and a scalar. The merge and mask instructions differ only in the way the indexes of the operands are used to create the result vector. For merge, the indexes of the operands are incremented only if that particular register is selected by VM. For mask, element  $n$  of the result vector is element  $n$  of either the left or the right operand.

### Compress

The compress instruction uses the VM register to extract elements selectively from one vector register and place the elements in another vector register. Either zeros or ones of VM can be used by specifying the instruction's `.f` (false) or `.t` (true) version, respectively. Only elements with the corresponding VM bit set (clear for `.f`) are moved from the source vector to the destination vector. This creates a destination vector with a number of elements equal to the number of bits set (or cleared) in VM.

### Expand

The expand instruction is only available on the CONVEX C2 and C3 Series computers. This instruction uses the VM register to extract elements from one vector register and selectively place the elements in another vector register. Either zeros or ones of VM can be used by specifying the instruction's `.f` or `.t` (false or true) version, respectively. Only elements with the

corresponding VM bit set (clear for . f) are loaded into the destination vector. Other elements in the destination vector corresponding to clear VM bits (set for . f) are skipped over. The expand instruction creates a destination vector with VL elements, including a number of elements of the source vector equal to the number of bits set (or clear) in the VM register.

## Examples

The examples below show how these instructions work.

The vector mask, merge, compress, and expand instructions have either a single true version, or both . t and . f (true and false) versions. You can use either the ones or the zeros (. t or . f) of VM. If you use . t, when the appropriate bit of VM is one, the second operand is selected.

Assume these values before the instructions of each example are executed:

```
V0 = 1 2 3 4 5 6      V5 = 7 7 7 7 7 7      VL = 6
V1 = a b c d e f      VM = 0 1 1 0 0 1      S1 = 8
```

Compressing V0 produces

```
cprs.t V0,V5 = 2 3 6 7 7 7
cprs.f V0,V5 = 1 4 5 7 7 7
```

Expanding V0 produces

```
xpnd.t V0,V5 = 7 1 2 7 7 3
xpnd.f V0,V5 = 1 7 7 2 3 7
```

Masking V0 and V1 produces

```
mask.t V0,V1,V5 = 1 b c 4 5 f
mask.t V1,V0,V5 = a 2 3 d e 6
mask.t V0,S1,V5 = 1 8 8 4 5 8
```

Merging V0 and V1 produces

```
VL = 12      VM = 0 1 1 0 0 1 0 0 0 1 1 1
merg.t V0,V1,V5 = 1 a b 2 3 c 4 5 6 d e f
```

Merging V0 and S1 with the previous VL and VM produces

```
merg.t V0,S1,V5 = 1 8 8 2 3 8 4 5 6 8 8 8
merg.f V0,S1,V5 = 8 1 2 8 8 3 8 8 8 4 5 6
```

---

## Examples of vector operations

This section shows examples of common vector operations. You do not need to understand assembly language to read the examples. To obtain the assembly-language listings of the examples, compile the source code with the `-O2`, `-tm C1`, and `-S` options. The `-tm` option is specified because the CONVEX C Series computers may produce differing assembly listings.

In the examples, if `a` is an array, then `Va` is the vector in which `a` is stored; `Vb(5)` is the fifth element of vector `Vb`; and `VM<6>` is the sixth bit of the VM register.

---

### Embedded if statement

The vector operations used in this example are conditional test and masking.

Vector operations often use conditional tests. The logical tests are `and`, `equal`, `less-than-or-equal`, `less-than`, `greater-than-or-equal`, `greater-than`, `or`, and `exclusive-or`. The CPU places the results of a vector comparison in the VM register, with the corresponding bit set if the result is true. If the comparison is `equal` and `V0(5)` is the same as `V1(5)`, then `VM<5>` equals 1.

The vector mask operation restricts the elements altered by a vector assignment operation to those specified by bits set in the VM register. In the vector mask sum `V1=V2+V3`, for example, `V1(5)` is assigned a value only if `VM<5>` is set.

The results of the conditional in the loop in the following example cannot be determined until the program is executed.

```
SUBROUTINE EMBED(A, B, C)
  INTEGER A(100), B(100), C(100), D(100), I

  DO I = 1, 100
    D(I) = I ! initialize array
  ENDDO

  DO I = 1, 100
    IF(A(I) .EQ. B(I)) D(I) = C(I)
  ENDDO

  PRINT *, (D(I), I = 1, 100)
END
```

Array `A` is loaded into `Va`, and array `B` is loaded into `Vb`. The two vectors are compared, and the result is stored in `VM`. The VM register controls the assignment operation. `Vc(i)` is assigned to `Vd(i)`. Finally, when `VM<i>` is one, `Vd` is stored in `D`.

---

## Indirect array addressing

The vector operations used in this example are gather and scatter.

*Gather* loads values from an array into a vector register. The operands come from various locations in the array. For example, if random gather moves elements from A to Va, A(5) may be placed in Va(10) while A(10) is copied into Va(2). *Scatter* copies elements from a vector register into various locations in an array.

The gather and scatter vector operations are used when the elements of an array are indirectly addressed. The code in the following example uses indirect addressing.

```
SUBROUTINE GATHER (A, IA, B, IB)
  INTEGER I, A(100), IA(100), B(100), IB(100)

  DO I = 1, 100
    A(IA(I)) = B(IB(I)) + 1
  ENDDO

  PRINT *, (A(I), I = 1, 100)
  RETURN
END
```

The assembly language that performs this function is shown in the following example.

```

ld.w    #1, s3
ld.w    #-4, a3
L3:
st.w    s3, LU+404
st.w    s3, -536(fp)
st.w    a3, -524(fp)
mov     ap, a5
ldea   LC+4, ap
calls  @12(a5)
ld.w   12(fp), ap
ld.w   -524(fp), a3
mov    a3, a5
add.w  fp, a5
add.w  #4, a3
st.w   s0, -508(a5)
ld.w   -536(fp), s0
add.w  #1, s0
le.w   #396, a3
mov.w  s0, s3
jbra.f L3

ld.w   #100, VL
ld.w   #4, s0
ld.w   8(ap), a5
ld.w   4(ap), a1
add.w  #-4, a5
ld.w   0(ap), a2
ld.w   #1, s1
add.w  #-4, a2
ld.w   #4, VS
ld.w   -512(fp), v0
mul.w  v0, s0, v1
ldvi.w v1, v2
ld.w   0(a1), v0
mul.w  v0, s0, v1
add.w  v2, s1, v3
mov    a2, a5
stvi.w v3, VL

```

The three steps to this operation are

1. Load the indirectly addressed elements of array B into a vector.
2. Increment each element by one.
3. Store the values into the indirectly addressed elements of array A.

The compiler accomplishes step 1 by loading the contents of array  $IB$  into a vector register (for example,  $V_{ib}$ ), incrementing each address by  $B$ 's base address, and storing the address in a vector register. For example, if  $IB(5)$  is 10,  $V_{ib}(5)$  is the value of  $B(10)$ . The compiler then increases each element of  $V_{ib}$  by one.

The compiler then loads the contents of array  $IA$  into vector register  $V_{ia}$ , increments each element by the base address of array  $A$ , and scatters the contents of  $V_{ib}$  using the addresses in  $V_{ia}$ . For example, if  $IA(5)$  is 17 and  $IB(5)$  is 10,  $B(10) + 1$  is stored in  $A(17)$ .

---

# Bibliography

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1987.

American National Standards Institute. *American National Standard Programming Language FORTRAN*. New York, New York: American National Standards Institute, 1978.

Bentley, Jon Louis. *Writing Efficient Programs*. Englewood Cliffs, NJ: Prentice Hall, 1982.

Fischer, Charles N. and Richard J. LeBlanc Jr. *Crafting a Compiler*. Menlo Park, CA: Benjamin/Cummings, 1988.

Levesque, John M. and Joel W. Williamson. *A Guidebook to FORTRAN on Supercomputers*. San Diego: Academic Press, Inc., 1989.

Padua, David A, and Michael J. Wolfe, "Advanced Compiler Optimizations for Supercomputers." *Communications of the ACM* (December 1986).

Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge, MA: Cambridge University Press, 1986.

Schofield, C. F. *Optimising FORTRAN Programs*. England: Ellis Horwood Limited, 1989.

Sedgewick, Robert. *Algorithms in C*. Reading, MA: Addison-Wesley, 1990.

Stone, Harold S. *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley, 1987.

Wolfe, Michael Joseph. *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: The MIT Press, 1989.



---

# Glossary

---

## A

### aliases

Multiple names for a single memory location. A typical alias arises in a subroutine to which an element in `COMMON` has been passed, if that memory location is also referred to within the subroutine as an element of `COMMON`. In the following example, `Z` is an alias for `Y` in this invocation of subroutine `SUB`:

```
...  
COMMON /DATA/X  
...  
CALL SUB (X)  
...  
END  
  
SUBROUTINE SUB (Y)  
COMMON /DATA/Z  
...  
RETURN  
END
```

Another kind of alias occurs across subroutine calls. In the following example, `B` is an alias for `C` in this invocation of subroutine `SUB`:

```
...  
CALL SUB (A, A)  
...  
END  
  
SUBROUTINE SUB (B, C)  
...
```

`EQUIVALENCE` statements create explicit aliases. Aliases complicate dependency analysis and can inhibit program optimization.

## ASAP

Automatic Self-Allocating Processors, a unique architecture designed by CONVEX. A cornerstone of ASAP is the communication register, which allows CPUs to seek out and execute the next piece of work as soon as possible.

---

## B

### bank conflict

An attempt to load two elements concurrently from the same memory bank. On CONVEX C2 and C3 Series machines, each memory board is divided into four 64-bit memory banks. Arrays are stored in main memory across all available banks.

Loading each array element takes eight clock cycles, during which time no other element can be retrieved from the same bank. Storing contiguous array elements across four memory banks allows each of the three intervening clock cycles to be used for loading another element.

### basic block

A linear sequence of statements that ends with a conditional or unconditional branch. A basic block is the optimization unit considered at optimization level -O0. A subprogram contains at least one basic block and typically contains many. The following subroutine is divided into basic blocks:

```
        SUBROUTINE SUB(A, B, N)
        REAL A, B(N), TMP
    Comment:  Begin basic block 1
            TMP = A
            IF (A .GE. 0) GOTO 10
    Comment:  Begin basic block 2
            A = -A
10       RETURN
        END
```

### balancing

See *tree-height reduction*.

---

## C

### chaining

See *vector chaining*.

**chime**

A chained vector time. The time required to perform the simultaneous instructions of one vector chain. On the CONVEX C Series machines, this is equal to the vector length plus 10 clock cycles.

**column-major order**

Memory representation of an array such that the columns of an array are stored contiguously. For example, given a two-dimensional array  $A(3, 4)$ ,  $A(3, 1)$  immediately precedes  $A(1, 2)$  in memory. This is the default storage method for arrays in FORTRAN.

**communication register**

A high-speed register used for communication among the threads of a process. Threads communicate by sending and receiving data through the communication registers. A hardware-maintained lock bit is associated with each communication register. The lock bit guarantees mutually exclusive access to the register.

**compress**

A vector operation that uses the vector-merge register to filter values in a vector. The operation copies elements from one vector into another vector only if the bit in the vector-merge register that corresponds with the index of the vector's element is set to the same truth suffix value as that of the instruction.

**concurrent**

In parallel processing, threads that can execute at the same time are called concurrent threads.

**conditional induction variable**

Loop induction variables that are not incremented on every iteration

**constant folding**

Replacement of an operation on a constant with the result of the operation

**constant propagation**

Replacement of a variable with a constant. For example, if you assign  $X=5$ , the compiler can replace  $X$  with 5 within that basic block or until a new value is assigned to the variable.

### **copy propagation**

Replacement of a variable with another variable to which it has been equated. For example, if you assign  $X=Y$ , the compiler can replace later occurrences of  $X$  with  $Y$  if doing so eliminates a load from memory.

### **CPU**

Central processing unit

### **CPU time**

The amount of time the CPU requires to execute a program. Because programs share access to a CPU, the wall-clock time of a program may not be the same as its CPU time. If a program can use multiple processors, the CPU time may be greater than the wall-clock time. (See *wall-clock time*.)

### **critical region**

A segment of code that must be executed by only one CPU at a time

---

## **D**

### **data dependency**

A relationship between two statements, such that one statement must precede the other to produce the intended result. (See also *loop-carried dependency* and *loop-independent dependency*.)

### **distributed part**

A loop generated by the compiler in the process of loop distribution

---

## **E**

### **execution stream**

A series of instructions executed by a CPU

---

## **F**

### **functional unit**

A part of the CPU that performs a set of operations on quantities stored in registers

---

**G****gather**

A vector operation that loads values from an array into a vector register. The operands of this operation come from various locations in an array.

---

**H****hoist**

An optimization process that moves a load from within a loop to the basic block preceding the loop

---

**I****interleaved memory**

Memory that is divided into multiple banks to permit concurrent memory accesses

---

**L****loop-carried dependency (LCD)**

A dependency between two operations executed on different iterations of a given loop and on the same iteration of all enclosing loops. A loop carries a dependency from an indexed assignment to an indexed use if, for some iteration of the loop, the assignment stores a value that is referred to on a later iteration of the loop. For example, an LCD from  $A(I+1)$  to  $A(I)$  exists in the following loop:

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

An LCD from  $B(I+1)$  to  $B(I)$  exists in the following loop:

```
DO I = 1, 100
  A(I) = B(I) + C(I)
  B(I + 1) = D(I) * 3.14
ENDDO
```

**loop constant**

A constant or expression whose value does not change within the loop

---

## loop distribution

The restructuring of a loop nest to create additional innermost loops and to enhance opportunities for loop interchange. Loop distribution creates two or more loops, called distributed parts, isolating code that must run serially from parallelizable or vectorizable code.

## loop-independent dependency (LID)

A dependency between two operations executed on the same iteration of all enclosing loops such that one operation must precede the other to produce correct results. For example, an LID from the use of  $B(I)$  to the assignment to  $B(I)$  exists in the following loop:

```
DO I = 1, 100
  A(I) = B(I) + C(I)
  B(I) = 0.0
ENDDO
```

An LID from  $B(100)$  to  $B(I)$  exists in the following loop, though only on the hundredth iteration:

```
DO I = 1, 100
  A(I) = B(100) + C(I)
  B(I) = 0.0
ENDDO
```

## loop induction variable

A variable that changes linearly within the loop, that is, whose value is incremented by a constant amount. For example, in the following loop,  $J$  and  $K$  are induction variables, but  $L$  is not.

```
DO I = 1, N
  J = J + 2
  K = K + N
  L = L + I
ENDDO
```

## loop interchange

The reordering of nested loops to increase the granularity of the parallelizable outer loop, to increase the iteration count of the vectorizable inner loop, or to achieve the most efficient vector stride in the inner loop.

## loop invariant

See *loop constant*.

## loop invariant computation

An operation that yields the same result on every iteration of a loop

---

## M

### mask

See *vector mask*.

### memory bank conflict

See *bank conflict*.

### merge, vector

See *vector merge*.

### mutual exclusion

A protocol that prevents access to a given resource by more than one thread at a time

---

## O

### oversubscript

An array reference that falls outside declared bounds

---

## P

### parallel vector loop

A nested loop structure such that the innermost loop is vectorized and the outer strip-mine loop can run in parallel if a CPU is available

### parallelization

The act of creating code that enables sections of code to run simultaneously on multiple CPUs. At optimization level -O3, the CONVEX FORTRAN compiler automatically parallelizes your program and recognizes compiler directives with which you can specify parallelization.

### pipelining

Grouping register loads together for concurrent execution

### population count

A vector operation that counts the number of bits that are set, or not set, in the vector-merge register

---

**process**

A collection of one or more execution streams within a single logical address space; an executable program. A process is made up of one or more threads.

**program unit**

A subroutine, function, or main section

---

**R****recurrence**

A cycle of dependencies among the operations within a loop. (See also *data dependency*.)

**re-entrancy**

The ability of a program unit to have multiple versions in existence that may execute in parallel. Each version maintains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary variables.

**row-major order**

Memory representation of an array such that the rows of an array are stored contiguously. For example, given a two dimensional array  $A(3, 4)$ ,  $A(1, 4)$  immediately precedes  $A(2, 1)$  in memory.

---

**S****scalar spreading**

The substitution of a temporary vector for a scalar

**scatter**

A vector operation that stores values from a vector into an array in memory. The destinations of this operation are various locations in the array.

**sinking**

An optimization process that moves a store from within a loop to the basic block following the loop

**span**

The distance between a jump or branch instruction and its target

---

**stack**

Storage automatically allocated on entry to a block of code by instructions that the compiler generates

**strip length, parallel**

The amount by which the induction variable of the inner loop is advanced on each iteration of the outer loop

**strip length, vector**

The number of array elements processed in a given vector operation

**strip mining**

The transformation of a single loop into two nested loops. CONVEX compilers perform parallel and vector strip-mine optimizations.

In a parallel strip-mine optimization, the outer loop (the parallel strip-mine loop) advances the initial value of the inner loop's induction variable by the parallel strip length. When more than one processor is detected (or specified with the `-ep` option), the parallel strip length is based on the trip count of the loop and the amount of code in the loop body.

In a vector strip-mine optimization, the inner loop is vectorized, and the outer loop iterates over blocks of arrays in steps equal to the vector length of the target machine. When more than one processor is detected (or specified with the `-ep` option), the vector strip length is based on the trip count of the loop and the amount of code in the loop body.

**synchronization**

A way to keep two threads from accessing the same critical region simultaneously. You can synchronize programs using compiler directives or assembly-language instructions. You do so, however, at the cost of additional overhead; synchronization may cause at least one CPU to wait for another.

---

**T****thread**

An independent execution stream that is fetched and executed by a CPU. One or more threads, each of which can execute on a different CPU, make up each process. Memory, files, signals, and other process attributes are generally shared among threads in a given process, enabling the threads to cooperate in solving the common problem. Threads are created and terminated by instructions that can be automatically generated by CONVEX compilers, inserted by adding compiler directives to source code, or coded explicitly in assembly-language programs.

**thread-private or thread-specific**

Data that is accessible by a single thread only (not shared among the threads constituting a process). Thread-specific data allows the same virtual address to refer to different physical memory locations.

**tree-height reduction**

Expressions are represented internally as trees whose height corresponds to the depth of the expression. These trees are optimized by tree-height reduction or balancing. For example, the height of  $A+B+C+D+E+F+G+H$  could be seven:  $(((((A+B)+C)+D)+E)+F)+G)+H$ . However, the compiler orders this expression so that more than one addition can occur at the same time:  $((A+B)+(C+D))+((E+F)+(G+H))$ . The height of this tree is three. Shorter heights mean faster execution. Tree height reduction occurs only for floating-point expressions.

---

**V****vector accumulator register (V)**

A vector register that can contain up to 128 64-bit operands called elements. It is used in high-speed calculations.

**vector chaining**

The overlapping of vector operations in the CPU. For instance, in the case of a vector load followed by a vector add, the add may be started as soon as the first operands are available.

**vector mask**

A vector operation that restricts the assignments that are computed in a vector assignment. The assignments are determined by the bits in the vector-merge register.

**vector merge**

A vector operation that merges either two vectors or a vector and a scalar into one vector. The assignments are determined by the vector-merge register.

**vector merge register (VM)**

A vector register that holds the status of element-by-element array comparisons and controls certain vector operations

**vector spill**

A situation in which more vectors are used in a calculation than can be stored in vector registers. The overflow must be stored and retrieved, as needed.

**vector stride (VS)**

The distance in bytes between adjacent array elements. This figure is used with arrays to load them into vector accumulators or transfer them to memory from a vector accumulator.

---

**W****wall-clock time**

The time an application requires to complete its processing. If an application starts running at 1:00 p.m. and finishes at 5:00 a.m., its wall-clock time is sixteen hours. See *CPU time*.



---

# Index

.fil file 82, 83

---

## A

aborts  
  program 89, 94, 95  
accesses  
  memory 68  
  partial memory 72  
accessing arrays 68  
actual arguments 83  
adjustable arrays 82  
algebraic simplification 13  
algorithms  
  parallelism of 3, 58  
aliases  
  defined 135  
  hidden 86, 88  
allocation  
  of registers 7  
alternate entry  
  routine 82  
alternate exits  
  loop 65  
*American National Standard Programming Language FORTRAN 90*  
ANSI FORTRAN 77 Standard 9, 86, 89, 90  
anti-dependency 75  
apparent dependency 47, 59, 93  
apparent recurrence 33, 91, 93, 109  
arguments  
  actual 83  
  CHARACTER 82  
  dummy 81, 82, 83, 86, 112  
array I/O  
  implicit 112  
array index  
  odd leading 72  
array stride  
  even 71  
  odd 71  
arrays  
  accessing 68  
  adjustable 82  
  compression 118  
  dummy 89  
  expansion 118

  in EQUIVALENCE 32, 47  
  merging 118  
  promoting 75  
  storage of 67  
ASAP 3  
  defined 136  
ASSIGN\_LOCK directive 103, 104  
ASSIGN statement 15  
assigned GOTO statements 32  
assignment substitution 11  
assignments  
  elimination of redundant 10, 14  
assistance  
  technical xv  
Automatic Self-Allocating Processors (ASAP)  
  3, 136  
  defined 136

---

## B

backward dependency 34, 35, 36, 49  
balanced tree 8  
balancing  
  defined 136  
  trees 7  
bank conflicts 70, 72  
  defined 136  
banks  
  memory 68, 69, 72  
basic 12  
basic block 12  
  defined 136  
basic-block optimizations 2, 13  
BEGIN\_ORDER directive 103  
BEGIN\_SECTION directive 104  
BEGIN\_TASKS directive 50, 105  
binary search procedure 54  
binary vector operators 122  
boundary value tests 66, 67  
bprof 5  
branches  
  span-dependent 7  
breakpoints  
  in inlined code 84

---

## C

- c option 40, 82
- calls
  - subprogram 47
- caution
  - on NO\_SIDE\_EFFECTS 16
  - start, stop, and iteration values 63
- chained vector time 137
- chaining 46, 124
  - defined 136
- CHARACTER
  - accessing 72
  - arguments 82
- chime
  - defined 137
- code
  - erroneous 85
  - nonstandard 85
- code motion 19, 99
- column-major order 67, 71, 112
  - defined 137
- COMMON block 83, 84, 86
- common subexpressions
  - elimination of 11, 17
- communication registers 3
  - defined 137
- comparison operators 64
- comparisons
  - vector 125
- compilation time 81
- compiler directives 101
- compiling a new application 53
- complicated conditionals 97
- complicated iteration tests 64
- complicated subscripts 56
- compress
  - defined 137
- computed statements 32
- concurrent
  - defined 137
- concurrent execution 6
- conditional induction variables 32
  - defined 137
- conditional test operation 129
- conditionals 98
  - complicated 97
  - embedded 56, 65, 78, 79
  - removing 78
  - short-circuit evaluation 9

- conflicts
  - bank 70, 72
  - bank, defined 136
- constant folding 12, 14
  - defined 137
- constant propagation 12, 14, 73
  - defined 137
- constant trip count 96
- constants
  - floating-point 61
  - strength reduction of 21
  - type conversion 12
- constructs
  - effective 61
- contact utility 53
- conversions
  - precision 61
  - type 61
- CONVEX Consultant 4
- CONVEX Performance Analyzer (CXpa) 4,  
54, 55, 56, 57, 59, 84
- copy propagation 17
  - defined 138
- count
  - iteration 62, 97
  - trip 26, 56, 68, 73, 74, 96, 97
- counted loop
  - defined 62
- CPU
  - defined 138
- CPU time 43, 55, 56, 57, 58, 59
  - defined 138
- Cray POINTER declarations 82
- critical region
  - defined 138
- cs option 82
- csd 4, 53, 84
- customer support xv
- CXpa 4, 5, 54, 55, 56, 57, 59, 81, 84

---

## D

- data
  - thread-private 47
- data dependency
  - defined 33, 138
- data requests 68
- DATA statement 82
- dead code
  - eliminating 5, 16, 81

- debugger
  - source-level 4
  - symbolic 4, 53, 84
- dependency 33, 90
  - apparent 47, 59, 93
  - backward 34, 35, 36, 49
  - defined 33, 138
  - forward 34, 49, 92
  - hidden 55
  - loop-carried (LCD) 33, 34, 35, 36, 47, 49, 91
  - loop-independent (LID) 33, 35, 36
- determined order of execution 95
- directives 50
  - ASSIGN\_LOCK 103
  - BEGIN\_ORDER 103
  - BEGIN\_SECTION 104
  - BEGIN\_TASKS 50, 105
  - END\_ORDER 103
  - END\_SECTION 104
  - END\_TASKS 50, 105
  - FORCE\_PARALLEL 47, 59, 92, 106
  - FORCE\_PARALLEL\_EXT 106
  - FORCE\_VECTOR 106, 107
  - FREE\_LOCK 103
  - MAX\_TRIPS 57, 73, 74, 75, 107
  - misused 55, 85, 90, 97
  - NEXT\_TASK 50, 105
  - NO\_PARALLEL 106, 108
  - NO\_PEEL 30, 108
  - NO\_PROMOTE\_TEST 31, 108
  - NO\_RECURRENCE 56, 59, 91, 93, 107, 108, 109
  - NO\_SIDE\_EFFECTS 16, 109
  - NO\_VECTOR 110
  - PEEL 29, 30, 110
  - PEEL\_ALL 29, 110
  - PREFER\_PARALLEL 110
  - PREFER\_PARALLEL\_EXT 110
  - PREFER\_VECTOR 111
  - PROMOTE\_TEST 31, 111
  - PROMOTE\_TEST\_ALL 31, 111
  - PSTRIP 97, 111
  - ROW\_WISE 67, 112
  - SCALAR 57, 74, 98, 106, 107, 108, 113
  - SELECT 75, 98, 114
  - SYNCH\_PARALLEL 49, 97, 114
  - tasking 4, 50
  - UNROLL 75, 115
  - VSTRIP 97, 115
- distributed parts 39, 77, 79
  - defined 138
- distribution
  - loop 25, 39, 44, 74, 75, 77, 79

- DO loops 62
- DO WHILE loops 62
  - vectorizing 64
- documentation
  - ordering xv
- dot product 124
- DOUBLE PRECISION
  - effect of 61, 90
- dummy arguments 81, 82, 83, 86, 112
- dummy array 89
- dynamic RAM 68
- dynamic selection 98

---

## E

- eliminating type conversions 99
- elimination
  - of common subexpressions 11, 17
  - of dead code 5, 16
  - of function assignments 15
  - of redundant assignments 10, 14
  - of redundant loads 11
  - of redundant uses 12
- embedded conditionals 56, 65, 78, 79
- END\_ORDER directive 103
- END\_SECTION directive 104
- END\_TASKS directive 50, 105
- entries
  - multiple routine 32, 47, 82
- EQUIVALENCE 32, 47
- equivalence
  - mathematical 90
- equivalent expressions 90
- erroneous code 85
- error message
  - overflow 13
- errors
  - logic 53, 56, 58
  - roundoff 99
- evaluation order 94
- even stride 71
- execution stream
  - defined 138
- exits
  - multiple loop 65
  - multiple routine 32, 47
- expressions
  - equivalent 90, 99
  - invariant 99
  - mixed-mode 61

---

## F

floating-point  
  imprecision 21, 55, 85, 90  
  operations 61  
  roundoff 54, 58  
  variables and constants 61  
folding  
  constant 12, 14, 137  
FORCE\_PARALLEL directive 47, 59, 92, 106  
FORCE\_VECTOR directive 106, 107  
FORTRAN Standard, ANSI 90  
forward dependency 34, 49, 92  
FREE\_LOCK directive 103  
function calls 47  
functional units 6, 124  
  defined 138  
functions  
  intrinsic 37, 83

---

## G

gather 130  
  defined 139  
global optimizations 2, 13  
  scope 81  
GOTO statements 67

---

## H

half-word data  
  accessing 72  
hand unrolling 63  
hand-coded loops 67  
hidden aliases 86, 88  
hidden dependency 55  
hoisting 16, 17, 27, 39, 46  
  defined 16, 139

---

## I

I/O  
  implicit array 112  
IF tests  
  embedded 78  
  vectorizing 65  
IF-DO interchange 30, 31  
IF-DO optimizations 28  
IF-ELSE 98  
-il option 82

implicit array I/O 112  
imprecision  
  floating-point 21, 55, 90  
induction variables 21, 32, 62, 67, 95, 96  
  replacement 95  
inline substitution 81  
inlined subprograms  
  nesting 81  
inlining 81  
instruction scheduling 6, 10  
instruction span  
  defined 142  
instructions  
  span-dependent 7  
INTEGER operations 61  
INTEGER\*2  
  accessing 72  
interchange  
  IF-DO 30, 31  
  loop 26, 27, 39, 43, 45, 67, 68, 77, 113  
interleaved memory 69  
  defined 139  
intermediate language (.fil) file 82  
intrinsic functions 37, 83  
invariant expressions 99  
  -is option 82, 83  
iterating by zero 94  
iteration count 26, 56, 62, 68, 73, 74, 97  
iteration tests  
  complicated 64  
iteration value 62, 63, 94  
iteration variables 63, 94

---

## J

jumps  
  span-dependent 7

---

## L

language-compatibility options 84  
LCD 33, 34, 35, 36, 49, 91  
  defined 139  
leading index  
  odd 72  
LID 33, 35, 36  
  defined 140  
loading  
  system 58  
logic errors 53, 56, 58

- loop constants 21
  - defined 139
- loop distribution 25, 39, 44, 74, 75, 77, 79
  - defined 140
- loop exits
  - multiple 65
- loop induction variable
  - defined 140
- loop interchange 26, 27, 39, 43, 45, 67, 68, 77, 113
  - defined 140
- loop invariant
  - defined 140
- loop invariant computation
  - defined 141
- loop limit value 96
- loop peeling 29
- loop start value 96
- loop stride 96
- loop termination test 96
- loop unrolling 63, 75
- loop-carried dependency (LCD) 33, 35, 36, 47, 49, 91
  - defined 139
- loop-independent dependency (LID) 33, 35, 36
  - defined 140
- loops
  - counted 62
  - DO 62
  - DO WHILE 62
  - hand-coded 67

---

## M

- machine-dependent optimizations 2, 6, 9, 13
- machine-dependent scalar optimizations 2, 5
- machine-independent optimizations 9
- machine-independent scalar optimizations 2, 5
- MAIN program 82
- masks
  - vector operations under 125
- matching patterns 31
- mathematical equivalence 90
- matrix multiplication 26, 38, 43
- MAX\_TRIPS directive 57, 73, 74, 75, 107
- maximum trip count 97
  - equation 97
- memory
  - interleaving 69, 139

- memory access 68
  - partial 72
- memory banks 68, 69
  - conflict 72, 136
- memory interleaving 68
- memory requirements 81
- message
  - overflow error 13
- misused directives 55, 85, 90, 97
- mixed-mode expressions 61
- moving code 19, 99
- multiple loop exits 65
- multiple routine entries 32, 47
- multiple routine exits 32, 47
- multiprocessing 3
- multithreaded programs 3
- mutual exclusion
  - defined 141

---

## N

- NAMELIST statement 82
- nesting of inlined subprograms 81
- NEXT\_TASK directive 50, 105
  - no option 1, 2, 5, 6, 9
- NO\_PARALLEL directive 106, 108
- NO\_PEEEL directive 30, 108
- NO\_PROMOTE\_TEST directive 31, 108
- NO\_RECURRENCE directive 56, 59, 91, 93, 107, 108, 109
- NO\_SIDE\_EFFECTS directive 16, 109
  - caution 16
- NO\_VECTOR directive 110
- nondeterminism
  - parallel 47, 95
- nonstandard code 85
  - nopeel option 30
  - noptst option 31
  - nosc option 9
- notational conventions xiii

---

## O

- O0 option 1, 2, 5, 9
- O1 option 1, 2, 5, 13, 23
- O2 option 1, 3, 23, 28
- O3 option 1, 4
- odd leading index 72
- odd stride 71

- operations
  - floating point 61
  - integer 61
- operators
  - comparison 64
- optimization options 1
- optimization report 38, 40
- optimization strategy 53
- optimizations
  - basic-block 13
  - basics of 1
  - global 13, 81
  - IF-DO 28
  - local level 2
  - machine-dependent 2, 5, 9, 13
  - machine-independent 2, 5, 9
    - no option 2
  - parallel 3
  - scalar 5
- options
  - c 40, 82
  - cs 82
  - il 82
  - is 82, 83
  - language-compatibility 84
  - misused 85
    - no 1, 2, 5, 6, 9
    - nopeel 30
    - noptst 31
    - nosc 9
    - O0 1, 2, 5, 9
    - O1 1, 2, 5, 13, 23
    - O2 1, 3, 23, 28
    - O3 1, 4
  - optimization 1
    - or 38
    - pa 54, 59
    - peel 29
    - ptst 30
    - ptstall 30
    - re 47, 84, 92
    - s 5, 82
    - uo 20, 21, 38, 99
- OPTIONS statement 55, 58
  - or option 38
- order of evaluation 94
- ordering documentation xv
- overflow 99
  - error message 13
- overhead
  - strip-mine 73
- oversubscripting 89
  - defined 141

---

## P

- pa option 54, 59
- paired hoist and sink 27, 46
- parallel calls 92
- parallel optimizations 3
- parallel processing 3
- parallel strip length
  - defined 143
- parallel strip-mine
  - defined 143
- parallel vector loop 49
  - defined 141
- parallelization
  - defined 141
- parentheses
  - use of 94
- partial memory access 72
- pattern matching 31
- PEEL directive 29, 30, 110
  - peel option 29
  - PEEL\_ALL directive 29, 110
- peeling
  - loop 29
- performance analyzer 4, 54, 55, 56, 57, 59, 84
- pipelining 7, 124
  - defined 141
- POINTER declarations
  - Cray 82
- population count 118
  - defined 141
- porting an application 53
- precision
  - conversion of 61
  - effect of floating-point 61
- PREFER\_PARALLEL directive 110
- PREFER\_PARALLEL\_EXT directive 110
- PREFER\_VECTOR directive 111
- premature loop termination 96
- process
  - defined 142
- process virtual time 59
- processing
  - parallel 3
- processor functional units 6, 124
- profiler 4, 5, 54, 55, 56, 57, 59, 81
- program aborts 95
- program unit
  - defined 142
- programming constructs 61
- programs
  - multithreaded 3

program-unit optimizations 2  
PROMOTE\_TEST directive 31, 111  
PROMOTE\_TEST\_ALL directive 31, 111  
promoting arrays 75  
promotion  
  test 30  
propagating constants 12, 14, 73  
propagating copies 17  
PSTRIP directive 97, 111  
-ptst option 30  
-ptstall option 30

---

## R

RAM 68  
-re option 47, 84, 92  
read requests 70  
READ statement 112  
REAL  
  effect of 61, 90  
REAL variables  
  effect of 21  
REAL\*16  
  effect of 61  
recurrence 32, 33, 34, 36, 49, 75, 90, 95, 108  
  apparent 33, 91, 93, 109  
  defined 33, 142  
recursion 33, 81  
reduction  
  strength 20, 21, 99  
  tree height 7, 8  
  vector 37, 90, 93, 123  
redundant loads  
  elimination of 11  
redundant-assignment elimination 10, 14  
redundant-test elimination 28  
redundant-use elimination 12  
re-entrancy 47, 84, 92  
  defined 142  
register allocation 7  
removing conditionals 78  
requests  
  data 68  
  read 70  
roundoff  
  floating-point 54, 58  
roundoff error 85, 90, 99  
routine entry  
  alternate 82  
ROW\_WISE directive 67, 112  
row-major order 67, 71  
  defined 142

runtime selection 98

---

## S

-S option 5, 82  
SAVE statement 82, 84  
scalar (S) register 123  
SCALAR directive 57, 74, 98, 106, 107, 108, 113  
scalar instruction  
  defined 2, 5  
scalar optimizations 5  
  basics of 2  
  machine-dependent 2  
  machine-independent 2  
scalar spreading 34  
  defined 142  
scalar value  
  defined 2, 5  
scatter 130  
  defined 142  
scheduling  
  instructions 10  
search procedure  
  binary 54  
SELECT directive 75, 98, 114  
selection  
  dynamic 98  
short-circuit evaluation of conditionals 9  
short-form instructions 7  
  at -no level 7  
side effects 109  
single-byte data  
  accessing 72  
sinking 16, 17, 27, 39, 46  
  defined 16, 27, 142  
source-level debugger 4  
span  
  defined 142  
span-dependent instructions 7  
sparse vector manipulation 118  
stack 84  
  defined 143  
start value 63  
  loop 96  
statement function 82  
stop value 62, 63, 64, 67  
storage of arrays 67  
strategy  
  optimization 53  
strength reduction 99  
  at -O1 20  
  of induction variables 21

- stride
  - array 70
  - even 71
  - loop 96
  - odd 71
- strip length
  - defined 143
  - determining 62
- strip mines
  - defined 143
  - unnecessary 57, 73
- strip mining 24, 43, 45, 49, 107
  - defined 23, 143
  - with SELECT directive 114
- strip-mine
  - length with PSTRIP directive 111
  - overhead 73
- subexpressions
  - elimination of 11, 17
- subprogram calls 47
- subscripts
  - invalid 89
- substitution
  - inline 81
  - of assignments 11
- support
  - technical xv
- symbolic debugger 4, 53
- SYNCH\_PARALLEL directive 49, 97, 114
- synchronization
  - defined 143
- synchronization code 43
  - defined 49
- system loading 58

---

## T

- tasking directives 4, 50
- technical assistance xv
- test elimination
  - redundant 28
- test promotion 30
- test replacement 95, 96, 97
- thread
  - defined 3, 43, 144
- thread-private
  - defined 144
- thread-private data 47
- thread-specific
  - defined 144
- time to solution 43
- transformations 24

- tree
  - balanced 8
  - balancing 7
  - tree-height reduction 7, 8
    - defined 144
  - trigonometric simplification 13
  - trip count 26, 56, 68, 73, 74, 96, 97
    - constant 96
    - variable 96
  - trip-count variable 73
  - type conversions 61
    - eliminating 99
    - of constants 12
  - type statement 82
  - typographic conventions xiii

---

## U

- unbalanced tree 7
- UNROLL directive 75, 115
- unrolling 63
  - loop 75
- unsafe optimizations
  - potential 21
- uo option 20, 21, 38, 99

---

## V

- value
  - iteration 63, 94
  - start 63
  - stop 62, 63, 64, 67
- variable trip count 96
- variables
  - conditional induction 32
  - floating-point 61
  - in EQUIVALENCE 32, 47
  - induction 21, 62, 67, 95, 96
  - iteration 63, 94
  - of inlined subprograms 84
  - static 84
  - trip-count 73
- VECLIB 59
- vector
  - accumulator (V) register 144
  - chained time 137
  - merge (VM) register 145
  - stride (VS) register 145
- vector (V) register 117, 119, 120, 121, 123
- vector accumulator (V) register 144
- vector addition operator 122

vector chaining 46, 124  
    defined 144  
vector clipping 118  
vector comparisons 125  
vector compress 137  
vector division operator 122  
vector length 26, 56, 97, 116  
    optimal 45  
vector load instruction 120  
vector mask  
    defined 144  
vector mask operation 127, 129  
vector merge  
    defined 145  
vector merge operation 127  
vector multiplication operator 122  
vector operations under mask 125  
vector operators  
    binary 122  
vector reduction 123  
vector register  
    as accumulator 27  
vector spill  
    defined 145  
vector store 121  
vector strip length  
    defined 143  
vector strip-mine  
    defined 143  
vector subtraction operator 122  
vector-accumulator (V) register 117  
vectorization 3  
vector-length (VL) register 117, 119, 120, 121,  
    122, 128  
vector-merge (VM) register 117, 118, 125, 127,  
    128, 129, 145  
vector-stride (VS) register 117, 119, 120, 121,  
    145  
VSTRIP directive 97, 115

---

## W

wall-clock time  
    defined 145  
WHILE test  
    complicated 64

---

## Z

zero  
    iterating by 94  
zero stride 94









**Order Number**  
**DSW-034**



**Document Number**  
**720-000930-203**