CONVEX FORTRAN Optimization Guide

Fourth Edition

CONVEX

CONVEX COMPUTER CORPORATION

CONVEX FORTRAN Optimization Guide

Order No. DSW-034

Fourth Edition
November 1992

CONVEX Press
Richardson, Texas
United States of America

vision Information for

ONVEX FORTRAN ptimization Guide

dition	Document No.	Description
ourth	720-000930-205	Released with CONVEX FORTRAN software V8.0, November 1992. Created new Chapter 9, "Unsafe optimizations," to cover material covered in Appendix A of former editions. Created Chapter 10, "Limits of optimization;" this information was previously covered in Chapter 9. Created a new Appendix A, "Optimization options." Added Appendix D, "Optimization report," which expands on the optimization report information formerly contained in Chapter 3, and updates it to explain the new optimization report. Added `DO_PRIVATE` and `TASK_PRIVATE` directives to and removed `ASSIGN_LOCK`, `FREE_LOCK`, `BEGIN_ORDER`, `END_ORDER`, `BEGIN_SECTION`, and `END_SECTION` directives from Appendix B, "Compiler directives."
Third	720-000930-203	Released with CONVEX FORTRAN software V7.0, November 1991.
Second	720-000930-201	Released with CONVEX FORTRAN software V6.0, March 1990.
First	720-000930-200	Released with CONVEX FORTRAN software V5.1, May 1989. Initial release of the manual.

Contents

Figures

Tables

How to use this guide

Purpose and audience

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.

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.

Scope

This guide covers the optimization of CONVEX FORTRAN Version 8.0, which runs under ConvexOS Version 10.0 or higher. CONVEX FORTRAN Version 8.0 also requires the CONVEX Assembler, Loader and Libraries (ALL) Version 2.0 or higher. The CONVEX FORTRAN compiler runs on all CONVEX hardware platforms, incluing C1, C2 and C3 Series architectures.

This guide is concerned with producing highly-efficient, optimized programs. 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.

Organization

This document consists of the following 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.
- Chapter 2 presents the basics of scalar optimization and explains how the compiler transforms programs compiled for scalar optimization (command line options `-no`, `-O0`, and `-O1`).
- Chapter 3 presents the basics of vector optimization and explains how the compiler transforms programs compiled for vector optimization (command line option `-O2`).
- Chapter 4 presents the basics of parallel optimization and explains 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 discusses inline substitution and how to use it to enhance optimization.
- Chapter 9 discusses potentially unsafe optimizations and the benefits and risks associated with them.
- Chapter 10 discusses common optimization problems you can encounter and presents some possible solutions.
- Appendix A lists and describes compiler options that relate to optimization.
- 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.
- Appendix D explains the optimization report.

Notational conventions

This section discusses notational conventions used in this book.

Command syntax

Consider this example:

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

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 adb(1), 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 User's Guide* (DSW-038), *CONVEX FORTRAN Language Reference Manual* (DSW-037), and *Release Notice, CONVEX FORTRAN Compiler V8.0.*
- 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 the CXdb debugger, see the *CONVEX CXdb Reference* (DSW-472) or *CONVEX CXdb Concepts* (DSW-471).
- 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 Architecture Reference Manual (C Series)* (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.

The basics 1

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 in Table 1.

Table 1
Optimization options

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 enables 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 of loops," in Chapter 4, and Appendix B, "Compiler directives."

Optimization tools

CONVEX CXdb is an optional window-based debugger that includes all the functionality of regular debuggers and is capable of debugging optimized code. CXdb is an optional product; refer to *CONVEX CXdb Concepts* for additional information.

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, refer to 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, refer to the *CONVEX Performance Analyzer User's Guide.*

CONVEX CXmetrics is an optional window-based tool that provides analytical data about the relative complexity of C and FORTRAN programs. These data, called *software metrics*, consist of numerical quantities that measure particular characteristics of a program. Using software metrics can help you reduce complexity and improve quality in your programs. For more information on CXmetrics, refer to the *CONVEX CXmetrics User's Guide.*

The CONVEX Application Compiler is an interprocedural analyzer that tracks the flow of data and control between procedures. The information generated by this analysis removes scope restrictions on optimization, which allows the Application Compiler to generate more efficient code by taking the entire program, with all its dependencies, into account. The database of program information that the interprocedural analyzer builds up also allows the Application Compiler to perform better error checking, leading to more robust and reliable programs. Many of the optimizations discussed in this book are performed automatically by the Aplication Compiler, with little or no user intervention.

The CONVEX Application Compiler is an optional product. For more information, refer to the *CONVEX Application Compiler User's Guide*, or contact your CONVEX sales representative.

Scalar optimization 2

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 following FORTRAN statement:

```
SCALAR1 = SCALAR2 + 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  E, s2
ld.w B, s1           mul.s s1, s0
add.s s1, s0         ld.w  F, s1
ld.w  E, s1          ld.w  B, s3
ld.w  F, s2          mul.s s2, s1
mul.s s2, s1         add.s s3, s0
sub.s s1, s0         sub.s s1, s0
st.w  s0, A          st.w  s0, A
```

In the original code, many operations must wait until a previous operation finishes. In the optimized code, the instructions are arranged to that data is not demanded before it is ready. Loads are moved ahead of arithmetic operations so that the data will arrive before the arithmetic operation starts. Operations that use different functional units, such as multiply and load, 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 Architecture Reference Manual (C Series).*

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 Architecture Reference Manual (C Series)* 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:

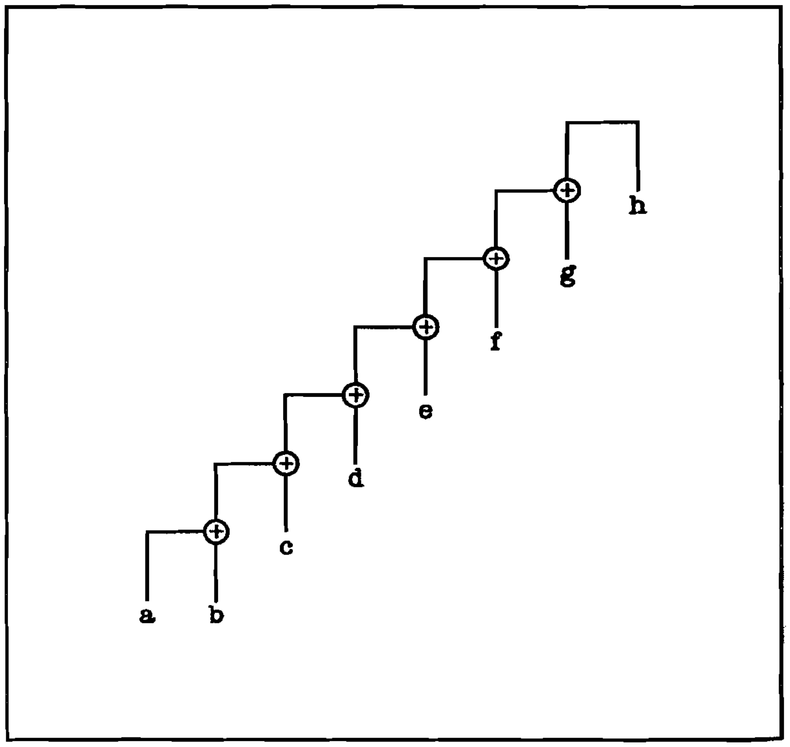
```
A + B + C + D + E + F + G + H
```

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 3 shows how the compiler represents this order internally.

Figure 1
Unbalanced tree representation



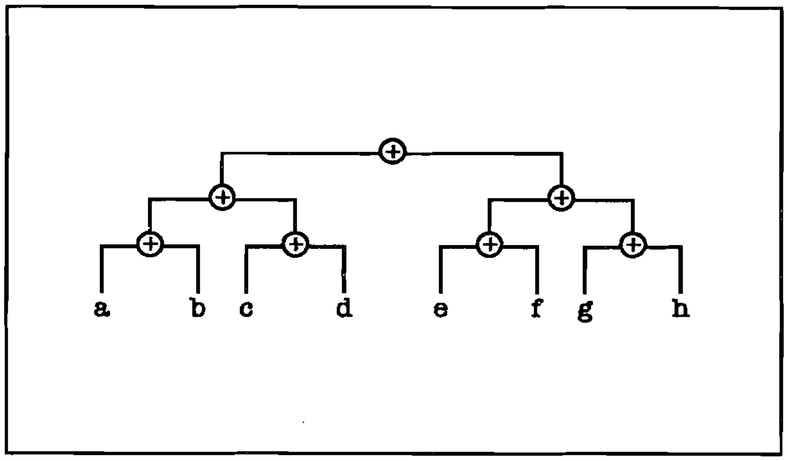
Another way to evaluate the expression is

```
(((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.

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.

Figure 2
Balanced tree representation



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 that 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 shown in the following example:

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 follows.

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 the following example.

Original expression	Optimized expression
`X + 0`	`X`
`X * 1`	`X`
`X * 0`	`0`
`K .AND. -1`	`K`
`K .AND. 0`	`0`
`K .OR. -1`	`-1`
`K .OR. 0`	`K`
`-1 * X`	`-X`
`X - X`	`0`
`X / -1`	`-X`
`(-1) ** K`	`1-((K .AND. 1) * 2)`
`X ** 0.5`	`SQRT(X)`
`X ** 0`	`1`
`1 ** X`	`1`
`X / X`	`1`
`0 - X`	`-X`
`0 / X`	`0`
`SIN(X) * COS(X)`	`0.5 * SIN(2X)`
`SIN(X) / COS(X)`	`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 follows.

```
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=2

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 t 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
      .
      .
      .
C     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 procedure compilers cannot automatically determine whether a side effect exists. The CONVEX FORTRAN compiler eliminates 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."

The CONVEX Application Compiler is capable of recognizing side effects in function and subroutine calls. Refer to the *CONVEX Application Compiler User's Guide* for more information.

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+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, refer to Chapter 9, "Potentially unsafe optimizations."

Strength reduction

In some cases, the compiler can replace an arithmetic operation with an equivalent operation (possibly non-arithmetic) that executes more quickly. Such replacements are called strength reductions.

Explicit arithmetic reductions

The compiler can reduce the strength of various arithmetic operations. On C Series machines, for example, the compiler transforms integer multiplication on positive numbers 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.

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
```

Vector optimization 3

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; increment I and the addresses of A, B, and C; and repeat for each of the next 99 elements.

Example

```
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, `IOUT` 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, IOUT is tested four times. For each comparison of IOUT to zero, the table below shows values of I and IOUT and the elements of array A that are calculated.

I	IOUT	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.

Caution

Loops with runtime trip counts greater than 2^{32} - 1 will yield incorrect results unless they are manually strip mined. Refer to the "Large trip counts at -O2 and above" section of Chapter 10 for more information.

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 in the following loop.

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.

Example

```
DO J = 1, N
  DO I = 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 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.

Example

```
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 thes 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 subscr a loop constant so that sinking and hoisting is possible.

IF-DO optimizations

IF-DO optimizations modify loops containing tests to improv vector performance. Tests can be promoted out of the loops o 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 eliminat 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
  ENDIF
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 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 la amounts of code and can greatly increase the size of the executable file. By default, the compiler peels boundary value and expands the code up to a predetermined conservative lin you can increase this limit by using the `-peel` compiler optio 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` directiv In codes containing large numbers of boundary-value operations, allowing code expansion without bound can grea lengthen compile time and can increase the size of the code enough to exceed the limits of some of the compiler's interna tables.

Boundary-value peeling can be disabled completely with the `-nopeel` compiler option. Similarly, you can disable peeling a loop-by-loop basis with the `NO_PEEL` compiler directives. S 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

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

Interchanged loop

```
IF (G) THEN
  DO I = 1, 100
    A(I) = B(I)
  ENDDO
ELSE
  DO I = 1
    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 an 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. At optimization levels -O2 and above, the compiler will recognize a loop that uses an IF test to determine a maximum (or minimum) value stored in an array and replace it with a call to a vectorized subroutine that performs the same task. The following example shows such a loop.

Example

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

Similarly, 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.

Example

```
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(I)) 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.

Example

```
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 the following 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 may be unfamiliar with recurrence an 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 reference 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 recursio 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 tha one operation depends on the results of the other. This implies 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-independen There must be at least one loop-carried dependency (LCD) fo 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.

Example

```
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.

Example

```
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)`.

Example

```
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. Th compiler interchanges the statements within the loop so that th assignment to `B` occurs before the assignment to `A`, as shown below.

Example

```
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              Vectorized loop

DO I = 1, 10               DO I = 1, 10

  X = A(I)                   TA(I) = A(I)

    = ... X ...                    = ... TA(I) ...

ENDDO                      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.

Example

```
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.

Caution

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.

Example

```
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.

Example

```
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-1: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.

Example

```
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 *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 = *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 instrinsic MIN function.

The following loop 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

```
SUM = 0.0
X = A(1)
DO I = 1, 100,
  SUM = SUM + A(I)
  X = MAX(X, A(I))
ENDDO
```

Vectorized loop

```
SUM = VSUM(A(1:100))
X = VMAX(A(1:100))
```

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.

Parallel optimization 4

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.

Example

```
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.

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.

Example

```
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.

Example

```
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.

Example

```
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.

Example

```
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.

Example

```
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. The following specific factors can inhibit or prevent automatic parallel optimization:

- 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.

Example

```
      .
      .
      .
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.

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.

Example

```
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 (LCI caused by the assignment to A(I+1). The loop cannot be vectorized or parallelized by the compiler. The loop remains in scalar form.

Example

```
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 an vectorizes the loop. It is not safe to parallelize a loop that has ar LCD, however. The result is a strip-mine vector instead of a parallel vector loop.

Example

```
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 threa 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 doe 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.

Example

```
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.

Example

```
C$DIR BEGIN_TASKS
       statement 1
C$DIR NEXT_TASK
       statement 2
C$DIR NEXT_TASK
       statement 3
C$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
```

Note

If the task contains a subroutine call and variable passed to the subroutine is referenced within the task, the compiler will issue a warning and fail to parallelize the task. If possible, moving the variable reference to before the BEGIN_TASKS directive will allow parallelization.

Optimizing FORTRAN applications 5

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 a debugger, such as csd or CXdb 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 acceptable 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 C1, C2, and C3200 Series machines, this slowdown occurs when the trip count is around 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.
   - Consider using CONVEX VECLIB routines where possible. CONVEX VECLIB is an optional library of highly-optimized mathematical routines. Refer to the *CONVEX VECLIB User's Guide* for more information.

   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.

Caution

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, refer to 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.

Example

```
      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.

Example

```
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; nor could the compiler unroll the loop in the presence of the `UNROLL` directive or the `-ur` flag.

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

Example

```
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.

Example

```
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.

Example

```
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.

Example

```
    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.

Example

```
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.

Example

```
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.

Example

```
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
ENDDO
```

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

Example

```
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.

Example

```
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, C3200 and C3400 Series supercomputers require eight clock cycles between main memory bank accesses; C3800 Series supercomputers, which have a faster clock, require up to 12 clock cycles. To speed up memory accesses, the CPU posts requests for data before the data is needed.

On C200, C3200, and C3400 Series supercomputers, main memory comprises at least 8 banks of dynamic RAM; on C3800 Series supercomputers, main memory comprises at least 16 banks. 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 3 shows the configuration of an eight-bank design such as that used on the C3200 Series. Bank numbers are indicated above the banks, and byte addresses are expressed in decimal notation within the banks.

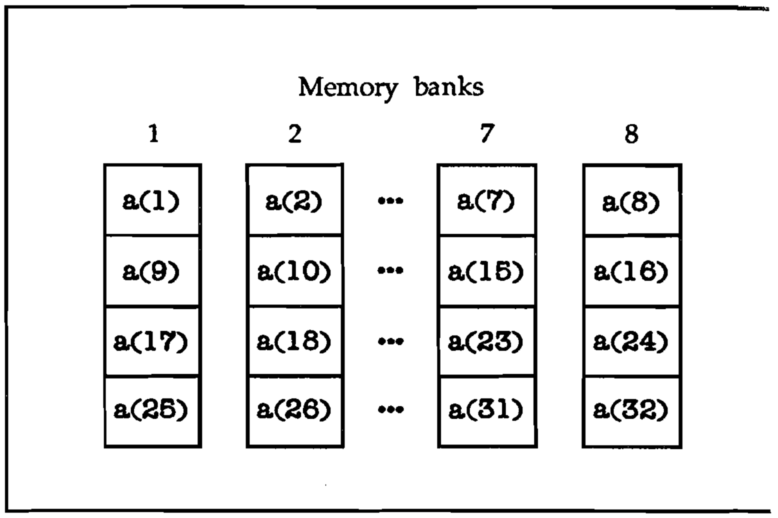
Figure 3
Eight-way interleaved memory

Memory banks

1	2	3	4	5	6	7	8
0	4	8	12	16	20	24	28
32	36	40	44	48	52	56	60
64	68	72	76	80	84	88	92
96	100	104	108	112	116	120	124

On a C3200, 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 4 shows a one-dimensional array in eight-way interleaved memory.

Figure 4
One-dimensional array in eight-way interleaved memory



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

Example

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

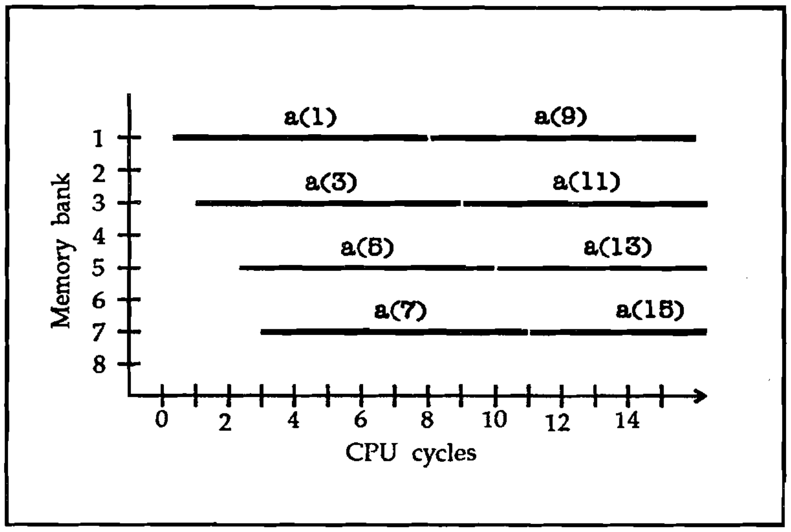
The following loop, however, causes memory bank conflicts, since the stride through A is 2. The CPU must wait for memory requests to be filled.

Example

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

Figure 5 shows the timing relationships that cause these bank conflicts.

Figure 5
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.

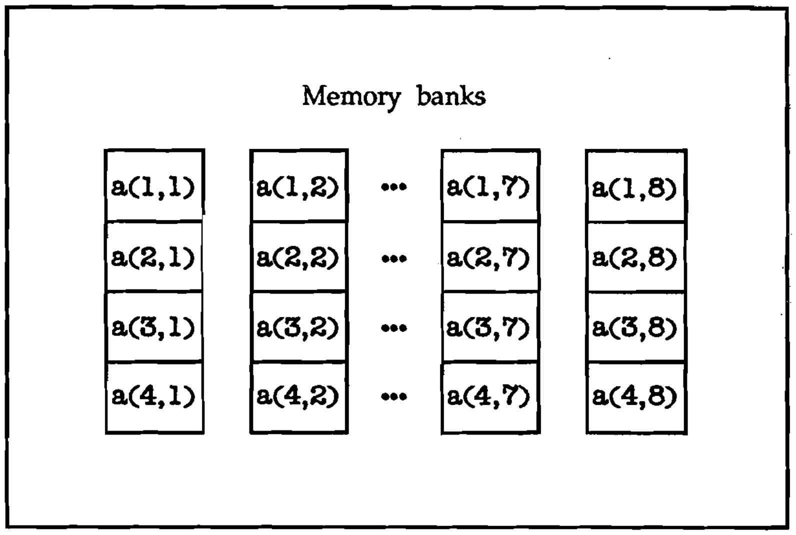
Arrays with a stride of two use only half the memory banks. Arrays with a stride of four use one bank in four. Whenever possible, avoid writing loops with a stride that is a multiple of a power of two. Odd strides give better performance than even strides.

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

Figure 6
Two-dimensional array stored in eight-way interleaved memory



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

Example

```
      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.

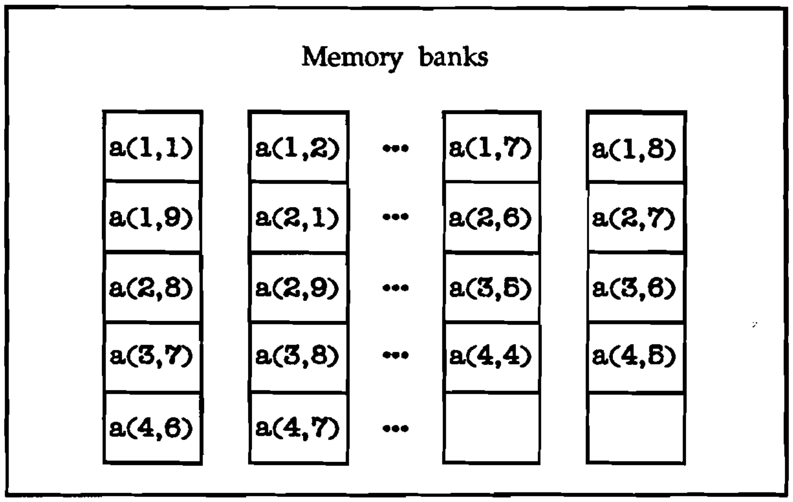
To avoid bank conflicts, declare the leading index of `A` to be an odd number, as shown in the following loop.

Example

```
      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 7 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 7
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.

Manual optimization techniques 7

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.

Example

```
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.

Example

```
       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. In the C-Series 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.

Example

```
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, as shown in the following example.

Example

```
       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.

Example

```
       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	**Unrolled loop**
`C$DIR UNROLL`	`A(1) = A(1) + 1`
`      DO I = 1, 4`	`A(2) = A(2) + 1`
`        A(I) = A(I) + 1`	`A(3) = A(3) + 1`
`      ENDDO`	`A(4) = A(4) + 1`

The UNROLL directive, which must be used at optimization level -O2 or higher, completely unrolls loops only if the compiler can determine that the trip count is less than five. For constant trip counts of five or more, UNROLL can partially unroll the loop if an attempted vectorization fails. If a loop has a variable trip count, you can partially unroll it by hand. Refer to Chapter 6, "Efficient programming constructs," for more information on manual loop unrolling. Refer to Appendix B, "Optimization options," for more information on the UNROLL directive.

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 the assignment to Q(J) in the J loop depends on the assignments to B(1), B(2), B(3), and B(4) in the I loop. Those values of array B exist only until the next iteration of the I loop; therefore they must be used by the J loop before they are overwritten in the next iteration of the I loop. The compiler cannot distribute the I loop because doing so would prevent Q from accumulating all values of B.

Example

```
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 dependency, 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
```

This insures that every value assigned to the array B is stored and available to the array Q later. With the dependency between B and Q eliminated, the compiler can distribute the I loop as shown in the following example.

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.

Example

```
   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.

Example

```
 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 8

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:

1. Create an intermediate language (`.fil`) file for each subprogram to be inlined.
2. 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 the 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.

Potentially unsafe optimizations 9

By default, the CONVEX FORTRAN compiler avoids performing optimizations that can potentially generate incorrect results. These optimizations can be enabled through use of the -uo option.

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 optimization," 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 only 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 a path to all loop exits.

In the following example, the invariant expression A=B/X is relocated only when the program is compiled with the -uo option.

Example

```
   DO I = 1, 100
     IF (X .NE. 0) THEN
       A= B/X
       AR(I) = A*C
     ELSE
       AR(I) = D*C
     ENDIF
   ENDDO
```

Conversion elimination

Type conversions are costly in terms of machine cycles, and they can inhibit vectorization. When you use the -uo option, the compiler eliminates costly type conversions by creating REAL induction variables that it then increments concurrently with the loop's INTEGER induction variables. Consider the loop below.

Original program

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

Here, in absence of the -uo option, I must be converted to type REAL on every iteration of the loop. With the -uo option the compiler avoids this costly operation by copying I into a REAL induction variable before entering the loop, then incrementing this REAL variable by 1.0 on every iteration of the loop. At optimization level -O2, the compiler can then vectorize the following optimized DO 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 optimization is considered potentially unsafe because the internal representation of real numbers is inexact, and this can lead to a significant accumulated error when `REAL_I` is incremented over the course of the loop.

Limits of optimization 10

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." From pages 17-3 and 17-4 of the standard:

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.

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.

Example

```
% 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.

Example

```
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.

Example

```
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.

Example

```
      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.

Example

```
         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
- Trip counts greater than $2^{31} - 1$ at optimization levels -O2 and -O3

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 example 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.

Example

```
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.

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.

Example

```
      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.

Example

```
CALL SUB1(0)
.
.
.
SUBROUTINE SUB1(IZR)
DIMENSION A(100), B(100), C(100)

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

DO I = 1, N, IZR  ! ITERATION VALUE OF 0 IS
                  ! NON-STANDARD
  A(I) = B(I)
ENDDO

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

Because IZR is an argument passed to SUB1, the compiler does not detect that IZR has been set to zero. All three loops vectorize at -O2, but because of the zero increments, their runtime behavior cannot be reliably predicted. All three loops compile at -O1, but the second loop, which specifies the step as part of the DO statement, will cause a runtime error. Runtime behavior of the other two loops cannot be predicted at -O1.

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.

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) GOTO 8
          IPACK = ((IRES*1024)*64)*64   !LINE 6
          IF((IPACK-IZERO).GE.0) GOTO 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:

$$\begin{aligned} stride * trip + start &< 2^{31} \\ 2^{22} * trip + 1 &< 2^{31} \\ trip &< 2^{9} - 2^{-21} \\ trip &< 512 \end{aligned}$$

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

Large trip counts at -O2 and above

When a loop is vectorized or parallelized, its trip count must fit in a signed 32-bit vector register. The largest positive value that can fit in a such a register is 2^{31} - 1 (2,147,483,647). If the compiler can determine that the trip count is larger than this at compile time, it will issue a warning. Loops with trip counts that cannot be determined at compile time but that exceed $2^{32} - 1$ at runtime will yield wrong answers.

This limitation only applies at optimization levels `-O2` and `-O3`.

Loops with trip counts that overflow 32 bits can be made to vectorize or parallelize by manually strip mining the loop.

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 ca 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 vector loop which executes both clauses on every vector iteration, rather than branching over instructions. On C1 Series machines, the scalar iterations that should not be executed in each clause will be executed in vector mode by explicitly inserting identity elements as operands so that no values are changed, but the operation is sti performed. On C2 and C3 Series machines, the scalar iterations that should not be executed in each clause are usually executed by vector masked instructions, where the hardware performs n operation for the scalar iterations that should not be executed. On all machines, significant overhead can result if one clause contains substantially more code than the other and is seldom executed. Balancing the amount of work in each clause and the percentage of time each clause executes will improve performance.

Note

Floating point exceptions can occur when arithmetic is performed on uninitialized variables. Since arithmetic (using identity elements) can be performed even in the false clause of an IF-ELSE construct, variables and arrays used therein should always be initialized.

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.

Optimization options A

This appendix provides a list of the optimization options available in CONVEX FORTRAN and a brief description of each.

Optimization level options

The options listed in this section specify the level of optimization allowed.

`-no`

Specifies that the compiler is to perform only machine-dependent scalar optimization. This option is the default if one of the `-O`*n* options is not specified. Refer to Chapter 2, "Scalar optimization."

`-O0`

Basic block machine-independent scalar optimization and machine-dependent scalar optimization. Refer to Chapter 2, "Scalar optimization."

`-O1`

`-O0` optimizations plus program unit machine-independent scalar optimization. Refer to Chapter 2, "Scalar optimization."

`-O2`

`-O1` plus vectorization. Refer to Chapter 3, "Vector optimization."

`-O3`

`-O2` plus parallelization. Refer to Chapter 4, "Parallel optimization."

Cross compilation options

The options listed in this section allow a program to be optimized for a machine other than the machine the program is being compiled on.

`-ep` *n*

Specifies the expected number of processors (*n*) on which the program is going to run. Must be used with the `-O3` option. The value of *n* should be an integer from 1 to 4 for C2 and C3200 Series machines, and from 1 to 8 for C3400 and C3800 Series machines. You can find out how many processors are installed on a machine by running the `sysinfo` command on that machine.

The compiler parallelizes a loop whenever doing so appears to decrease the turnaround time, assuming the given number of processors. Use this option with caution because it may lead to inefficient use of processors.

`-mi` *n*

Specifies the expected memory interleave on the target machine. *n* is an integer representing the expected memory interleave, which you can obtain for your machine with the `getsysinfo` command. When this option does not appear, the interleave of the machine the compiler is running on is used.

`-tm` *target*

Specifies the target machine architecture for which compilation is to be performed. *target* can take the value `c1`, `c2`, `c32`, `c34` or `c38`. Use `c1` when compiling for a C1 Series machine; use `c2` when compiling for a C2 Series machine that is not equiped with an Enhanced Scalar Processor; use `c32` when compiling for a C3200 Series machine or a C2 Series machine that is equiped with an Enhanced Scalar Processor; use `c34` when compiling for a C3400 Series machine; use `c38` when compiling for a C3800 Series machine. The `sysinfo` command will tell you the type of machine you are running on and whether or not is has an Enhanced Scalar Processor (denoted as `scalar_acc` in the `sysinfo` output). If you specify a target machine, its instruction set is used regardless of the machine on which the compiler is running. If you do not specify a target machine, the compiler generates instructions for the class of machine on which it is running.

Note

The `file` utility may indicate that an executable generated with `-tm` specifying some C2 or C3 Series machine is a C1 executable. This is because `file` only checks for instruction set differences between the two machines. An executable generated with the `-tm` option specifying a C2 or C3 Series machine will contain instruction scheduling differences from a file generated for a C1 that `file` will not detect.

Loop replication options

The options listed in this section control loop replication optimizations.

`-ds`

Causes the compiler to automatically select loops to replicate and to compile several versions of such loops. The compiler then dynamically selects the version of each loop to be executed. This option is available only at optimization level `-O2` or `-O3`. Refer to the *CONVEX FORTRAN User's Guide,* Chapter 1, "Compiling programs."

`-rl`

Causes the compiler to automatically select loops and replicate them by unrolling or dynamic selection. `-rl` is equivalent to specifying `-ds` and `-ur`. This option is available only at optimization level `-O2` or `-O3`. Refer to the *CONVEX FORTRAN User's Guide,* Chapter 1, "Compiling programs."

`-ur`

Causes the compiler to automatically find unrollable loops and unroll them. Loops with trip counts less than 5 are unrolled completely. Those with trip counts of 5 or more are partially unrolled. Only innermost loops can be unrolled. This option is available only at optimization levels `-O2` or `-O3`. Refer to the *CONVEX FORTRAN User's Guide,* Chapter 1, "Compiling programs."

IF-DO optimization options

The options listed in this section control the degree of loop peeling and test promotion allowed.

`-nopeel`

Disallows loop boundary value peeling, which is enabled by default at optimization levels `-O2` and `-O3`. Refer to the `-peel` and `-peelall` options described in this section. Refer also to Chapter 3, "Vector optimization."

`-noptst`

Disallows test promotion, which is enabled by default at optimization levels `-O2` and `-O3`. Refer to `-ptst` and `-ptstall` below. Refer also to Chapter 3, "Vector optimization."

`-peel`

Removes the first and/or last iterations of a loop when doing so removes 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 and/or last iteration. By default, the compiler peels boundary values and expands code up to a predetermined conservative limit. With the `-peel` option, this limit is increased and code expansion may become significant. `-peel` must be used with the `-O2` or `-O3` optimization options. Refer to Chapter 3, "Vector optimization."

`-peelall`

Same as `-peel`, but allows code expansion without bound. For code containing large numbers of boundary value operations, this can greatly lengthen compiler time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables. `-peelall` must be used with the `-O2` or `-O3` optimization options. Refer to Chapter 3, "Vector optimization."

`-ptst`

Causes a test to be promoted out of the loop that encloses it by replicating the containing loop for each branch of the test. By default, the compiler replicates code up to a predetermined conservative limit. The `-ptst` option increases this limit and can cause a noticeable increase in compile time. `-ptst` must be used with the `-O2` or `-O3` optimization options. Refer to Chapter 3, "Vector optimization."

`-ptstall`

Same as `-ptst`, but allows code replication without bound. For loops containing large numbers of tests, 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. `-ptstall` must be used with the `-O2` or `-O3` optimization options. Refer to Chapter 3, "Vector optimization."

Other optimization options

This section lists optimization options that cannot be otherwise catagorized.

`-il`

Instructs the compiler to prepare an intermediate language (`.fil`) file for a subprogram that is to be used for inline substitution. The `-il` option cannot be used with the `-c`, `-cs`, or `-S` options. Optimization levels are ignored. Refer to the *CONVEX FORTRAN User's Guide*, Chapter 1, "Compiling programs."

`-is` *directory*

Instructs the compiler to attempt inline substitution of each subprogram for which there exists a `.fil` (intermediate-language file) file in the specified *directory*. This option must be repeated for each directory containing `.fil` files to be used for inline substitution. Refer to the *CONVEX FORTRAN User's Guide*, Chapter 1, "Compiling programs."

`-or`

Controls generation of the optimization report.

`-uo`

Performs potentially unsafe optimizations, for example, moving the evaluation of common subexpressions or invariant code from within conditionally executed code. This moved code may be executed unconditionally. Refer to Chapter 9, "Potentially unsafe optimizations."

Compiler directives B

This appendix briefly describes the directives that are available in CONVEX FORTRAN.

Some directives provide information to the compiler that it cannot determine on its own. Other directives instruct the compiler to override certain default conditions that control optimization, vectorization, or parallelization. A directive line has the following format:

```
C$DIR directive [, directive ]
```

A directive line begins in column one with the characters `C$DIR` followed by one or more of the directives described in this appendix. If two or more directives are specified, they are separated by commas. A directive must fit on one line; it cannot be continued. A directive can be surrounded by any number of comment lines.

The following directives are supported:

- BEGIN_TASKS, NEXT_TASK, END_TASKS
- DO_PRIVATE
- FORCE_PARALLEL_EXT
- FORCE_PARALLEL
- FORCE_VECTOR
- MAX_TRIPS
- NO_PARALLEL
- NO_PEEL
- NO_PROMOTE_TEST
- NO_RECURRENCE

- NO_SIDE_EFFECTS
- NO_VECTOR
- PEEL
- PEEL_ALL
- PREFER_PARALLEL_EXT
- PREFER_PARALLEL
- PREFER_VECTOR
- PSTRIP
- PROMOTE_TEST
- PROMOTE_TEST_ALL
- ROW_WISE
- SCALAR
- SELECT
- SYNCH_PARALLEL
- TASK_PRIVATE
- UNROLL
- VSTRIP

The following directives, which were supported in previous versions of CONVEX FORTRAN, are no longer supported:

- ASSIGN_LOCK, FREE_LOCK
- BEGIN_ORDER, END_ORDER
- BEGIN_SECTION, END_SECTION

The compiler issues an advisory when it encounters any of these directives.

Certain combinations of directives are invalid when used within the same program unit or loop and cause the program unit or loop to be rejected by the compiler. Table 2 lists invalid combinations.

Table 2
Restrictions on directive use

	force_parallel	force_parallel_ext	force_vector	no_parallel	no_vector	prefer_parallel	prefer_parallel_ext	prefer_vector	pstrip	scalar	select	synch_parallel	unroll	vstrip
force_parallel		x	x	x		x	x	x		x	x	x	x	x
force_parallel_ext	x			x		x	x	x	x	x	x	x	x	
force_vector	x				x	x	x	x	x	x	x	x	x	
no_parallel	x	x				x	x		x	x	x	x		x
no_vector			x					x		x	x			x
prefer_parallel	x	x	x	x			x	x		x	x	x	x	x
prefer_parallel_ext	x	x	x	x		x			x	x	x	x	x	
prefer_vector	x	x	x		x	x			x	x	x	x	x	
pstrip		x	x	x			x	x		x		x	x	x
scalar	x	x	x	x	x	x	x	x	x		x	x		x
select	x	x	x	x	x	x	x	x		x			x	
synch_parallel	x	x	x	x		x	x	x	x	x			x	x
unroll	x	x	x			x	x	x	x		x	x		x
vstrip	x			x	x	x			x	x		x	x	

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

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

When using directives on loops, remember that loops can be executed in the following ways:

- Serial
- Vector but not parallel
- Parallel but not vector
- Parallel outer strip and vector inner strip

BEGIN_TASKS, NEXT_TASK, END_TASKS

The BEGIN_TASKS directive identifies a sequence of tasks for independent, parallel execution. A sequence of tasks begins with a BEGIN_TASKS directive and ends with an END_TASKS directive. A NEXT_TASK directive precedes each individual task. A task is defined as a sequence of nonloop code that can be executed in parallel.

The following code illustrates the use of the tasking directives:

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

The preceding example is equivalent to the following loop:

```
C$DIR FORCE_PARALLEL
      DO 100 I = 1,3
        GOTO (10,20,30),I
 10       statement-1
        GOTO 100
 20       statement-2
        GOTO 100
 30       statement-3
100   CONTINUE
```

Up to 255 tasks can be specified between a BEGIN_TASKS and an END_TASKS directive.

Note

If the task contains a subroutine call and variable passed to the subroutine is referenced within the task, the compiler will issue a warning and fail to parallelize the task. If possible, moving the variable reference to before the BEGIN_TASKS directive will allow parallelization.

DO_PRIVATE

The DO_PRIVATE directive declares a list of variables and/or arrays private to the immediately following DO loop. The compiler assumes that variables declared DO_PRIVATE have no loop-carried dependencies. No starting or ending values can be assumed for these variables.

The DO_PRIVATE directive has the form

```
C$DIR DO_PRIVATE(varlist)
```

where

varlist
  is a list of scalar variables or arrays, separated by commas, that are to be private to the immediately following loop.

  Only scalar variables and statically-sized arrays can be declared private. Dynamic, allocatable, and automatic arrays are not allowed. Structures are not allowed. Including induction variables (i.e. DO loop indices) in *varlist* will yield wrong answers.

If a variable that appears in *varlist* is referenced in an iteration of the loop, it must have been assigned a value previously on that iteration of the loop. Values assigned outside the loop or in previous iterations will not be available.

If the variable is referenced after the loop, it must have been assigned a value after the loop. Values assigned inside the loop or before the loop are not available.

The following example demonstrates use of the DO_PRIVATE directive.

Example

```
C$DIR   PREFER_PARALLEL
C$DIR   DO_PRIVATE(S)
        DO I=1,N
          IF (I .GT. ILIM) THEN
            S = 3.0
          ELSE IF (I .LE. ILIM) THEN
            S = 2.0
          ENDIF
          A(I) = S
        ENDDO
```

In this example, S must have a value for the assignment to A(I) at the end of the loop. Without the DO_PRIVATE directive, the compiler cannot tell that S is always assigned. It therefore assumes that the value of S from a previous iteration might be needed, and fails to parallelize the loop. The presence of DO_PRIVATE(S) tells the compiler to ignore the possible dependency on S, so that the PREFER_PARALLEL directive can be honored and the loop can be parallelized.

FORCE_PARALLEL

The FORCE_PARALLEL directive tells the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. Certain actual dependencies, such as from one scalar to another, can cause the compiler to ignore this directive. You can use this directive on a loop whether or not the loop contains calls, but it may not be safe to do so.

This directive is effective only if the -O3 compiler option is specified.

FORCE_PARALLEL does not allow interchange or distribution of 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 the SCALAR or NO_RECURRENCE directive, a warning is issued. In addition, an error occurs when you use FORCE_PARALLEL and another parallelizing directive in the same loop nest.

Example

```
C$DIR FORCE_PARALLEL
      DO I = 1, N
      ENDDO
```

FORCE_PARALLEL_EXT

The FORCE_PARALLEL_EXT directive forces the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. Loops can be parallelized with FORCE_PARALLEL_EXT whether or not they contain calls.

This directive is effective only if the -O3 compiler option is specified. If FORCE_PARALLEL_EXT and the FORCE_VECTOR directive are specified for the same loop, the compiler first vectorizes the loop and then parallelizes the resulting strip-mine loop.

FORCE_PARALLEL_EXT allows interchange of 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 attempt to use this directive with the SCALAR or NO_RECURRENCE directive, an error occurs. In addition, an error occurs when you try to 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 a FORCE_VECTOR directive with a loop that would be fully vectorized without the directive and get incorrect answers because the directive causes the compiler to ignore dependencies.

This directive should be used with fully vectorizable loops. If FORCE_VECTOR and FORCE_PARALLEL_EXT are specified 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.

A warning is issued if this directive is used with the SCALAR directive or with the NO_RECURRENCE directive. In addition, an error occurs when you attempt to use FORCE_VECTOR and another vectorizing directive in the same loop nest.

MAX_TRIPS

The MAX_TRIPS directive instructs the compiler that the following loop is never executed 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. This directive can be used to prevent strip mining, when it might otherwise be performed. The elimination of strip mining results in more efficient code generation.

NO_PARALLEL

The NO_PARALLEL directive tells the compiler not to parallelize the loop that immediately follows; vectorization is not prevented.

If the NO_PARALLEL and NO_VECTOR directives both precede a loop, the result is the same as if the SCALAR directive 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. Refer to 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. Refer to Chapter 3, "Vector optimization," for more information.

NO_RECURRENCE

The NO_RECURRENCE directive instructs the compiler to disregard an apparent recurrence in a loop. If there is no other impediment to vectorization, the loop is vectorized.

You must place this directive immediately before a DO statement or a labeled statement that begins a loop. Comment lines can appear between the directive and the start of the loop.

The NO_RECURRENCE directive does not affect recurrences caused by a nested DO loop. The directive can, however, be used on each loop in a nest to give the vectorizer maximum opportunity for improving the performance of the nest.

When the NO_RECURRENCE directive is used, the compiler breaks the recurrence by arbitrarily removing one or more dependencies of the cycle.

Example:

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

In this example, if J is positive, there is no recurrence.

The compiler always processes a NO_RECURRENCE directive when the apparent recurrence involves an array element. The compiler always ignores a NO_RECURRENCE directive when the apparent recurrence involves 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 on the NO_RECURRENCE directive, refer to Chapter 10, "Limits of optimization."

NO_SIDE_EFFECTS

The `NO_SIDE_EFFECTS` directive instructs the compiler that the specified functions do not modify the value of a parameter or common variable, perform a 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 scalar optimization to remove a function call if it occurs in an expression assigned to a scalar variable that is never used. The function call can be removed because the function has no side effects—it does not matter whether or not the call is made. Such optimization opportunities usually arise after other optimizations are performed and rarely occur in the original source text.

Although the directive can appear anywhere in a program unit, to be effective it must be used before the named function is called. Use the directive if the compiler gives the advisory message `More optimization is possible if this function call has no side effects`. If there are no arguments, the directive applies to all functions referenced (textually) after the directive.

Example

```
C$DIR NO_SIDE_EFFECTS (F1,F2)
          .
          .
          .
          X=Y* F1(5,Z)-W !IF THE X= DOES NOT REACH
          .              !A USE OF X, THE ASSIGNMENT
          .              !STMT CAN BE REMOVED
          .
```

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 that immediately follows; parallelization is not prevented.

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

PEEL

The PEEL directive allows the compiler to peel the loop immediately following the directive, expanding the code beyond the default conservative limit, but not without bound. Refer to 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. Refer to Chapter 3, "Vector optimization," for more information.

PREFER_PARALLEL

The PREFER_PARALLEL directive tells the compiler to parallelize the loop immediately following the directive if it is safe to do so. The compiler checks first for actual loop-carried dependencies; if none is found, the loop is parallelized.

This directive prevents interchange and distribution of outer loops for vectorization. An error occurs when you try to use PREFER_PARALLEL and another parallelizing directive in the same loop nest.

PREFER_PARALLEL_EXT

The PREFER_PARALLEL_EXT directive tells the compiler to parallelize the loop immediately following the directive only if it appears safe to do so. The compiler checks first for actual loop-carried dependencies; if none are found, the loop is parallelized.

This directive does not prevent interchange of outer loops for vectorization. If you also choose to vectorize this loop, use the PREFER_VECTOR directive. An error occurs when you try to use PREFER_PARALLEL_EXT and another parallelizing directive in the same loop nest.

PREFER_VECTOR

The PREFER_VECTOR directive tells the compiler to vectorize the loop immediately following the directive if it is safe to do so. The compiler checks first for actual recurrences. If no recurrences are found, the compiler tries to interchange the loop to be the innermost loop and vectorize it.

An error occurs when you try to use PREFER_VECTOR and another vectorizing directive in the same loop nest.

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. Refer to 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. Refer to Chapter 3, "Vector optimization," for more information.

PSTRIP

The PSTRIP directive tells the compiler that the parallel loop immediately following the directive is to be strip mined using the specified length. The format of this directive is

PSTRIP (*integer_constant*)

where *integer_constant* is an integer constant that specifies the strip-mine length.

Parallel strip mining groups the loop iterations into blocks of *n/(2ep)*, where *n* is the actual loop trip count and *ep* is the expected number of processors, which is obtained from the -ep option or, in absence of -ep, the number of processors in the machine on which the program is running. Each block is executed entirely by a single thread. Parallel strip mining occurs only at -O3.

If you do not specify PSTRIP directives, the compiler selects a default value appropriate for the architecture of the machine for which you are compiling. The default number of loop iterations to group, or when -ep is 1, is 1. At -O3 when -ep is 2 or more, the compiler will use longer strips to reduce the inter-processor overhead.

The PSTRIP directive overrides the compiler default and specifies the number of iterations per block to perform. PSTRIP cannot be used with vector loops.

Table 3 shows the maximum strip-mine lengths used with the -O3 and -ep options.

Table 3
Maximum parallel strip-mine lengths at -O3

Processors (-ep)	Default compiler length	PSTRIP(*k*) length
1	1	1
More than 1	max(*n*/(2*ep*),1)	*k*

Compiler directives

ROW_WISE

FORTRAN 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. The ROW_WISE directive tells the compiler that the designated arrays have their dimensions reversed. Thus, array elements are stored in a manner consistent with programming languages such as C and Ada. The format of this directive is:

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

The following cautions apply to the use of 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 reversed when viewed in the debugger.
- If the ROW_WISE directive is applied to a dummy argument, the actual argument must also appear in a ROW_WISE directive within the caller. The compiler cannot detect this situation.

The following example illustrates a situation in which use of the ROW_WISE directive can improve performance of a program.

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

Although the preceding example vectorizes, performance is slowed because the array is being accessed with noncontiguous memory (FORTRAN stores arrays in column-major order). If, however, the code segment in the preceding example is preceded by the directive `C$DIR ROW_WISE (A)`, it is interpreted by the compiler as follows:

```
C$DIR ROW_WISE (A)
DIMENSION A(1000,4)
DO I = 1,4
  DO J = 1,1000
    A(J,I) = 0
  ENDDO
ENDDO
```

The array is now being accessed from contiguous memory, thus increasing the execution speed.

SCALAR

The SCALAR directive prevents the DO loop that follows from being vectorized or parallelized. The body of the loop can still be vectorized or parallelized if an outer loop is interchanged with the `scalar` loop.

The SCALAR directive is useful when the iteration count of the loop is too low for the overhead involved in setting up vectorization, or when the numerical results must be the same as for a scalar loop. This directive can also be used to prevent loop interchange, which may not choose the best loop to interchange when the compiler cannot determine the iteration counts of the loops involved.

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.

Example 1

```
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 this example, the compiler normally interchanges the I loop with the 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.

Example 2

```
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)
```

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

SELECT

The SELECT directive causes the compiler to generate multiple versions of a loop and to select, at runtime, which version to execute based on specified trip counts. The compiler generates 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 (iteration) count at which the compiler is to select vector, parallel, or parallel-vector execution, respectively, for the loop. Parallel-vector execution implies that the loop is vectorized and the strip-mine loop is parallelized.

If you omit a trip count by using two adjacent commas, the compiler selects a default value. If you use an asterisk (*) in place of a trip count, the compiler does not generate code for the corresponding mode. If a specified mode is not available for the loop, the compiler selects a default mode.

If the actual trip count is less that the smallest trip count specified in the directive, the loop runs scalar. If the actual trip count is greater than the largest trip count specified in the directive, the loop runs in the mode of the largest trip count.

Examples

```
C$DIR SELECT(10,4,20)
C    Run scalar if actual trip count = 1-4.
C    Run parallel if trip count = 5-10.
C    Run vector if trip count = 11-20.
C    Run parallel-vector if trip count > 20.

C$DIR SELECT (0,*,*)
C    Run scalar if loop has no vectorizable code.

C$DIR SELECT (*,*,*)
C    Equivalent to C$DIR SCALAR.
```

SYNCH_PARALLEL

The SYNCH_PARALLEL directive tells the compiler that the following loop is to be executed in parallel; however, instead of ignoring dependencies, the compiler inserts synchronization code that causes the dependencies to be honored at runtime. This directive is effective only if the -O3 compiler option is specified.

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, it may be preferable to parallelize the loop with the SYNCH_PARALLEL directive, particularly if all the dependencies are in seldom-executed branches.

Example

```
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
```

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

TASK_PRIVATE

The TASK_PRIVATE directive declares a list of variables and/or arrays private to the immediately following task. A task is a sequence of linear code that can be executed in parallel with other tasks. In CONVEX FORTRAN, tasks are defined using the BEGIN_TASKS, NEXT_TASK, and END_TASKS directives. The TASK_PRIVATE directive must immediately precede or appear on the same line as the BEGIN_TASKS directive. The compiler assumes that variables declared TASK_PRIVATE have no dependencies between the following tasks; therefore no starting or ending value can be assumed for the task-private variable within a task.

The TASK_PRIVATE directive has the form

```
C$DIR TASK_PRIVATE(varlist)
```

where

varlist
: is a list of variables or arrays, separated by commas, that are to be private to each following task. The following tasks are defined by a BEGIN_TASKS directive and one or more NEXT_TASK directives. The scope of the task-private variables is terminated along with the task list when an END_TASKS directive is encountered.

  Only variables and statically-sized arrays can be declared private. Dynamic, allocatable, and automatic arrays are not allowed. Structures are not allowed. Including induction variables (i.e. DO loop indices) in *varlist* will yield wrong answers.

If a variable that appears in *varlist* is referenced within a task, it must have been assigned a value previously within that task. Values assigned outside the task list or in other tasks will not be available.

If the variable is referenced after the task list, it must have been assigned a value after the task list. Values assigned inside or before the task list are not available.

UNROLL

The UNROLL directive reduces loop overhead by replicating the body of the loop that follows. Unrolling is performed only on scalar loops. This directive is effective only if the -O2 compiler option is specified.

To be eligible for unrolling, a loop must contain no internal branching and must have an iteration count that the compiler determines. The compiler unrolls a loop completely only if its iteration count is less than five; otherwise, partial unrolling is performed. Complete unrolling occurs before vectorization, and partial unrolling after vectorization.

VSTRIP

The VSTRIP directive tells the compiler that the vector loop immediately following the directive is to be strip mined using the specified length. It is especially useful for automatically parallelized vector loops, for example, loops that are vectorized and run with the outer strip parallel. The directive has the following format:

```
VSTRIP (integer_constant)
```

where *integer_constant* is an integer constant that specifies the strip-mine length, which must be less than or equal to 128.

Vector strip mining executes the 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 directives and the compiler doesn't know the number of iterations (or knows that it is larger than 128*ep, where ep is the estimated number of processors), the compiler selects a default value of 128 for the strip-mine length. Also, loops are executed in 128-element strips at optimization level -O2 or if the value of the -ep flag is 1. At optimization level -O3 when -ep is 2 or more, the compiler uses more and shorter strips if doing so reduces the length of the longest strip.

The VSTRIP directive overrides the compiler default and specifies a shorter strip-mine length. The shorter strip creates more iterations of the strip-mine loop so that it can be effectively parallelized.

Table 4 shows the maximum strip-mine lengths used with the -O3 and -ep options.

Table 4
Maximum vector strip-mine lengths at -O3

Processors (-ep)	Default compiler length	VSTRIP(k) length
1	1	128
More than 1	max(min((*n*+*ep*-1)/*ep*,128),8)	*k*

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

Examples:

Table 5 shows the maximum and actual vector strip lengths when the system includes four processors (-ep=4).

Table 5
Four-processor system strip lengths

Trip count	Maximum strip length	Actual strip length(s)
2	8	2
514	128	128, 128, 128, 128, 2 (for the 5 iterations)

Vector operations

C

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

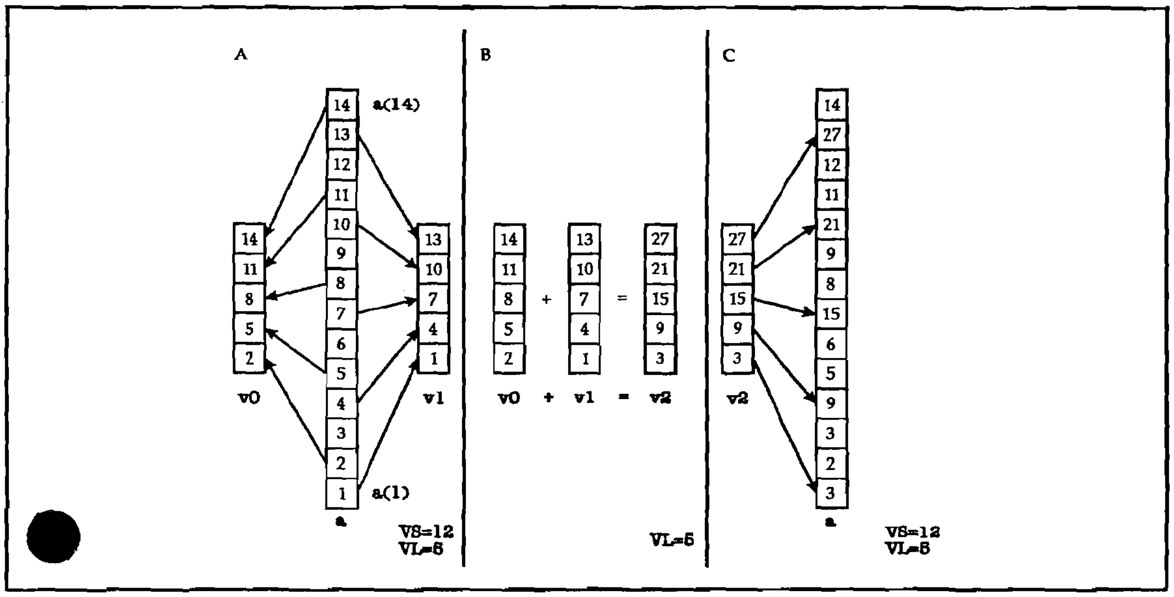
To see how the vector hardware works, consider the vector operation in the following example.

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 8 shows the vector operations on array A.

Figure 8
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 Architecture Reference Manual (C Series).*

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 regist and the VL register contains the number of array elements to l 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, intc VL. The second statement loads 4 into VS because each element in the array requires four bytes for storage and the loop's strid 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 modifie 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.

FORTRAN	**Assembly language**
`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.

FORTRAN	**Assembly language**
`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.

```
FORTRAN                      Assembly language

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.

```
FORTRAN                      Assembly language

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.

Example

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

FORTRAN **Assembly language**

```
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.

There are two forms of vector operations under mask:

- 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 following statement:

```
add.w    v0,v1,v2
```

adds all elements (restricted by vector length) of v0 and v1, placing the results in v2.

The following 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 following statement:

```
add.w.t v0,v1,v2
```

produces:

```
v2 = 5 8 10 5 5 8
```

The following 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.

Example

```
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.

Example

```
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
```

There are three steps to this operation:

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, Vib), incrementing each address by B's base address, and storing the address in a vector register. For example, if IB(5) is 10, Vib(5) is the value of B(10). The compiler then increases each element of Vib by one.

The compiler then loads the contents of array IA into vector register Via, increments each element by the base address of array A, and scatters the contents of Vib using the addresses in Via. For example, if IA(5) is 17 and IB(5) is 10, B(10)+1 is stored in A(17).

Optimization report

D

When you compile a program with the -O2 or -O3 option, the compiler generates an optimization report for each program unit. The -or option determines the report's contents, as shown in Table 6.

Table 6
Optimization report contents

-or option	Report contents
`all`	Loop table and array table
`loop`	Loop table only (default)
`array`	Array table only
none	No report

Loop table

The loop table lists the optimizations that were performed on each loop and, if appropriate, the reasons why a possible optimization was not performed. Loop nests are reported in the order in which they are encountered, and separated by a blank line. A description of each column of the loop table follows.

`Line Num.`

Specifies the source line of the beginning of the loop. If the line number has two parts separated by a hyphen, the second part is the distributend number (due to loop distribution).

`Id Num.`

Specifies a unique ID number for every loop. This ID number can then be referenced by other parts of the report. Both loops appearing in the original program source and loops created by the compiler are given loop ID numbers; loops created by the compiler are also enumerated in the `New Loops` column as described further on. No distinction between compiler-generated loops and loops that existed in the original source is made in the `Id Num` column; loops are assigned unique, sequential numbers as they are encountered.

`Iter. Var.`

Specifies the name of the iteration variable controlling the loop. If the variable is compiler-generated, its name is listed as `*VAR*`; if there is no iteration variable, it is listed as `*NONE*`. If the iteration variable has two parts separated by a colon, the second part is the inline substitution instance of that variable. If it consists of a truncated variable name followed by a colon and a number, the number is a reference to the variable name footnote table which appears after the loop table, analysis table, and test table in the report.

`Reordering Transformation`

Indicates which reordering transformations were performed. Reordering transformations are performed on loops and loop nests, and typically involve reordering and/or duplicating sections of code to facilitate more efficient execution. This column has one of the values shown in Table 7.

Table 7
Reordering transformations reported in optimization report

Value	Explanation
`Scalar`	No reordering transformation was performed.
n`% VECTOR`	The loop was partially vectorized, with the percentage (*n*) specified being executed in vector mode.
`FULL VECTOR`	The loop was fully vectorized, with all operations being executed in vector mode.
`PARALLEL`	The loop runs in parallel mode.
`PARA/VECTOR`	The loop was vectorized, and the strip mine loop runs in parallel mode.
`Inter`	Loop interchange was performed. Typically appears with vectorization indicator.
`Dist`	Loop distribution was performed.
`DynSel`	Dynamic selection was performed.
`Peel`	Loop peeling was performed.
`Promote`	Test promotion was performed.
`*`	Appears at left of loop-producing transformation optimizations (distribution, dynamic selection, peeling, promotion).

`New Loops`

Specifies the loop ID number(s) for loops created by the compiler. These ID numbers are listed in the `Id Num.` column and can be referenced in other parts of the report; however, the loops they represent were not present in the original source code.

`Optimizing/Special Transformation`

Indicates which, if any, optimizing transformations were performed. An optimizing transformation reduces the number of operations executed, or replaces operations with simpler operations. A special transformation allows the compiler to vectorize or parallelize code under special circumstances. When appropriate, this column has one of the values shown in Table 8.

Table 8
Optimizing/special transformations reported in optimization report

Value	Explanation
`Unroll`	The loop was completely or partially unrolled.
`Reduction`	The compiler recognized a reduction and vectorized the loop.
`Pattern`	The compiler recognized a special pattern and vectorized the loop.
`Synch`	The compiler inserted synchronization code to ensure correct execution of a parallel loop.
`Removed`	The compiler removed the loop.
`No Strip`	The loop did not need to be strip mined.

Analysis table

If necessary, an analysis table is included in the optimization report to further elaborate on optimizations reported in the loop table. A description of each column of the analysis table follows.

`Line Num.`

Specifies the source line of the beginning of the loop.

`Id Num.`

References the ID number assigned to the loop in the loop table.

`Iter. Var.`

Specifies the name of the iteration variable controlling the loop, `*VAR*`, or `*NONE*`, as described in the "Loop table" section of this appendix.

`Analysis`

Indicates why a transformation or optimization was not performed, or additional information on what was done.

Test table

If any test promotion or removal optimizations were performed, a test table is included in the optimization report. A description of each column in the test table follows.

`Line Num.`

Specifies the source line number of the beginning of the `IF` test.

`Col. Num.`

Specifies the source column number of the beginning of the `IF` test.

`Test Transformation`

Specifies the transformation performed: either `TEST PROMOTED` or `TEST REMOVED`.

`Analysis`

Presents a further explanation of the transformation performed, including the source line number of the original loop from which the test was transformed, and (if applicable), in parentheses, the loop ID number of the compiler-generated loop from which the test was transformed.

Variable name footnote table

Variable names that are too long to fit in the `Iter. Var.` columns of the loop and array table sections are truncated and followed by a colon and a footnote number. These footnotes are explained in the variable name footnote table. The headings in the variable name footnote table are explained below.

`Footnoted Iter. Var.`

Specifies the truncated variable name and its footnote number.

`User Variable Name`

Specifies the actual name of the variable as given by the user in the source code.

Array table

The array table lists array references that prevented optimization or array references on which special optimizations were performed. The array table contains the following information.

`Line Num.`

Specifies the source line on which the reference occurs.

`Var. Name`

Specifies the name of the array being referenced.

`Optimization`

If an optimization was performed on the array in question, this column contains one of the values shown in Table 9.

Table 9
Optimizations reported in array table

Value	Explanation
`Hoist`	The vector load was found to be loop invariant and was moved outside the loop.
`Sink`	The vector store was found to be loop invariant and was moved outside the loop.

`Dependencies`

If an array or memory recurrence prevented optimization, this column shows the names of variables in the recurrence, in the form *name*`@`*linenumber*. If the reference could be to any memory location, it is in the form `*MEM*@`*linenumber*. If the reference is to a subprogram call, it is in the form `*CALL*@`*linenumber*.

Examples

The following examples enumerate the contents of the optimization report. In discussing the examples, loops are referred to by their ID numbers.

Example 1

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 LOOPINDEX = 1, 200
 5         DO J = 1, 200
 6           C(LOOPINDEX,J) = 0
 7           DO K = 1, 200
 8             C(LOOPINDEX,J) = C(LOOPINDEX,J) +
      ^        A(LOOPINDEX, K) * B(K, J)
 9           ENDDO
10         ENDDO
11       ENDDO
```

```
12
13        END
```

At line 8, individual elements C(LOOPINDEX,J) are summed directly rather than stored in a temporary scalar variable. Introducing a temporary scalar later assigned to C(LOOPINDEX,J) would inhibit vectorization. Figure 9 shows the optimization report generated by compiling the program EXAMPLE1 at optimization level -O2.

Figure 9
Optimization report for Example 1

```
%fc -O2 -or all example1.f
            Optimization for Procedure EXAMPLE1

Line      Id    Iter.     Reordering        New      Optimizing / Special
Num.      Num.  Var.      Transformation    Loops    Transformation
--------------------------------------------------------------------------
    4       1   ILOOPI:1 *Dist              (2-3)    No Strip
    4-1     2   ILOOPI:1  FULL VECTOR Inter
    5-1     4   J         Scalar

    4-2     3   ILOOPI:1  FULL VECTOR Inter
    5-2     5   J         Scalar
    7-2     6   K         Scalar

Line      Id    Iter.   Analysis
Num.      Num.  Var.
--------------------------------------------------------------------------
    4-1     2   ILOOPI:1Interchanged to innermost
    4-2     3   ILOOPI:1Interchanged to innermost

Footnoted   User Variable
Iter. Var.  Name
--------------------------------------------------------------------------
ILOOPI:1    ILOOPINDEX

            Array References for Procedure EXAMPLE1

Line   Var.   Optimi-    Dependencies
Num.   Name.  zation
--------------------------------------------------------------------------
    8  C      Sunk
    8  C      Hoist
```

Loop number 1 is the loop that appears on line 4 of the source. It iterates over the variable LOOPINDEX. It was distributed and two new loops, numbers 2 and 3, were created. The optimizaiton report indicates that loop 1 was not strip mined.

The line number for loop number 2 tells us that it came from the loop at line 4 of the source (loop number 1), and it is in the first distributed part of that distribution. Loop 2 is interchanged with the innermost loop in its nest (see analysis table), loop number 4, which immediately follows it in the report. Number 4 iterates over J and runs scalar.

Loop number 3, which was created from the loop at source line 4 (loop number 1), is in the second distributed part of that loop's distribution. It is interchanged with the innermost loop in the nest, which is number 6. Note that a blank line preceeds loop 3, indicating the beginning of a new nest.

Loop number 5 appeared at line 5 in the source and is in the second distributed part of loop 1. It iterates over J and runs scalar.

Loop number 6 appeared at line 7 in the source code and is in the second distributed part of loop number 1. It also runs scalar.

The analysis table elaborates on the interchanges performed on loops 2 and 3, both of which were interchanged with the innermost loops in their respective nests to facilitate vectorization.

The array table shows that the load of array C, which appears at line 8, was hoisted from the loop, and that the store of C was sunk.

Example 2

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 10 shows the optimization report generated by compiling the subroutine `EXAMPLE2` above for vectorization. The `-c` option suppresses loading because no main program is included. `-ds` is included to demonstrate how dynamic selection is denoted in the optimization report.

Figure 10
Optimization report for Example 2

```
%fc -O2 -c -ds example2.f
           Optimization for Procedure EXAMPLE2

Line      Id    Iter.     Reordering        New       Optimizing / Special
Num.      Num.  Var.      Transformation    Loops     Transformation
------------------------------------------------------------------------
    6        1  I        *Promote          (2-3)
    6        2  I        *Promote          (4-5)
    6        4  I        *Peel             (6)
    6        6  I        *DynSel           (7-8)
    6        7  I         FULL VECTOR                 Reduction

    6        8  I         Scalar

    6        5  I        *Peel             (9)
    6        9  I        *DynSel           (10-11)
    6       10  I         FULL VECTOR                 Reduction

    6       11  I         Scalar

    6        3  I        *Peel             (12)
    6       12  I        *DynSel           (13-14)
    6       13  I         FULL VECTOR                 Reduction

    6       14  I         Scalar


Line     Col.    Test               Analysis
Num.     Num.    Transformation
------------------------------------------------------------------------
    8     14     TEST PROMOTED     Test promoted out of loop on 6 (7)
   10     19     TEST PROMOTED     Test promoted out of loop on 6 (7)
   13     16     TEST REMOVED      Peeled first iteration of loop on 6 (13)
    .
    .
    .
```

There is only one loop in this example, and it appears at line 6. Loops 2 and 3, which are noted under `New Loops` on loop 1's line, are the result of promoting tests from the original loop; loops 4 and 5 are the result of promoting tests from loop 2. Loop 6 is created when loop 4 is peeled. Two loops, 7 and 8, are then created to facilitate dynamic selection of loop 6. Loop 7 is the vector version of loop 6, and loop 8 is the scalar; parallel and parallel/vector versions were omitted because this example was compiled at `-O2`.

Loops 3 and 5 were both peeled and the resulting loops were replicated into scalar and vector versions by dynamic selection.

After all the transformations are completed, six loops (ID numbers 7, 8, 10, 11, 13, and 14) remain in the program. These remaining loops can be easily spotted under the `Reordering Transformation` column, as they are the loops that are not marked with the "*" transformation indicator. Loops marked with this symbol no longer exist because they are replaced by the new loops indicated in the `New Loops` column.

The test table of this report gives details of the test promotions and peelings that are mentioned in the loop table. For brevity, several of the peeling explanations that appeared at the end of the report were omitted from Figure 10.

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 C200, C3200, and C3400 Series machines, the basic unit of memory banking is a pair of boards known as an MCM pair and consists of eight 64-bit (or, equivalently, sixteen 32-bit) memory banks. On CONVEX C3800 Series machines, the basic unit is a single board known as an NMB (Neptune Memory Board) and consists of sixteen 64-bit (or, equivalently, thirty-two 32-bit) memory banks. Arrays are stored in main memory across all available banks.

Loading an array element from a bank renders the bank unaccessable for a period known as the *refresh time*. On C3200 and C3400 Series machines this refresh time is 8 clock cycles; on C3800 Series machines it is less than or equal to 12 clock cycles.

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. For basic operations such as add and multiply on the CONVEX C Series machines, this is equal to the vector length plus a small amount of overhead.

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.

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.

trip count

The number of times a loop executes.

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

Order Number
DSW-034

Document Number
720-000930-205