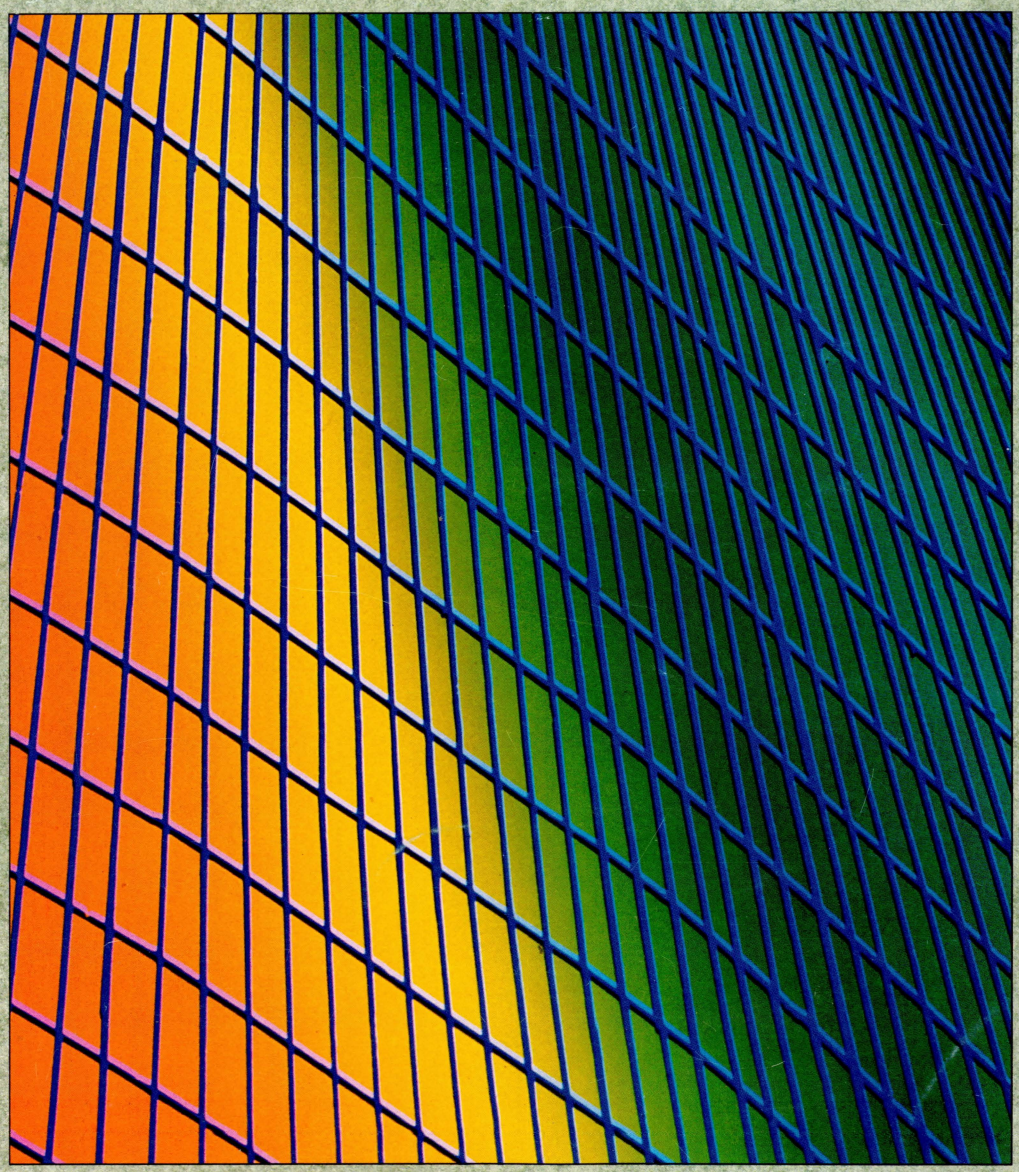
CONVEX

FORTRAN

Optimization Guide

Second Edition



CONVEX

FORTRAN

Optimization Guide

Second Edition

Develpment Software Documentation
CONVEX Computer Corporation

CONVEX FORTRAN Optimization Guide
Order No. DSW-034
Second Edition

Printed in the United States of America

Acknowledgments

The authors wish to thank all the people at CONVEX and the users who contributed their ideas and time in the development of this book.

Cover Design: Josie Davis

Cover Photograph: Courtesy of Dornier

About the Cover: This simulation from Dornier, a leading manufacturer of aircraft and spacecraft in the Federal Republic of Germany, was calculated on a CONVEX system. The simulation depicts surface pressures on an aircraft. Simulations are used to test aircraft thermally and aerodynamically.

Table of Contents

CONVEX FORTRAN Optimization Guide
First Edition

About This Guide

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

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

Audience

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

Organization

This document consists of these chapters:

- Chapter 1 introduces CONVEX's approach to program optimization. Chapter 1 defines the terms and concepts you need to understand how the CONVEX FORTRAN compiler works with CONVEX C100 and C200 Series™ architectures.
- In Chapter 2, you learn the basics of scalar optimization and how the compiler transforms programs compiled for scalar optimization (command line options `-no`, `-O0`, and `-O1`).
- In Chapter 3, you learn the basics of vector optimization and how the compiler transforms programs compiled for vector optimization (command line option `-O2`).
- In Chapter 4, you learn the basics of parallel optimization and how the compiler transforms programs compiled for parallel optimization (command line option `-O3`).
- Chapter 5 presents a strategy for developing your programs to enhance optimization and provides you with examples of using compiler options and directives and their effects on optimization.
- Chapter 6 discusses programming constructs that can aid or hinder optimization.
- Chapter 7 presents some tricks and tips for optimizing your programs to run on CONVEX C Series supercomputers.
- Chapter 8 tells how and when to use inline substitution to enhance optimization.
- Chapter 9 discusses common optimization problems you can encounter and presents some possible solutions.
- Appendix A covers some details about using the `-uo` compiler option.
- Appendix B explains how to use CONVEX FORTRAN compiler directives.
- Appendix C describes vector operations on the assembly-language level and presents examples of some assembly-language instructions.
- Appendix D contains a glossary of terms used throughout the document.
- Appendix E contains a bibliography on topics related to optimization.

Related Reading

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*, *CONVEX FORTRAN Language Reference Manual*, and *Release Notice, CONVEX FORTRAN Compiler V6.0*.
- ❑ For more information about CXpa™, see the *CONVEX Performance Analyzer (CXpa) User's Guide* and the *CONVEX Performance Analyzer (CXpa) Reference Manual*.
- ❑ For more information on `csd`, the source-level debugger, see the *CONVEX Consultant User's Guide*.
- ❑ For more information on parallel processing on CONVEX supercomputers, see the Fall 1988 issue of *Vector* (Volume II, Number 3).
- ❑ For more information on parallel programming in assembly language, see the *CONVEX Assembly-Language User's Guide* and the *CONVEX Architecture Reference*.

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. Figure 1-1 summarizes the optimization options.

Figure 1-1 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 `-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 `-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 perform the following:

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

Vector Optimization

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

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

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

Parallel Optimization

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

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

Every program has at least one *thread* or sequence of instructions that can execute on a single CPU. Parallel programs have more than one thread. On the C200 Series, 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 Chapter 4 and Appendix B.

Optimization Tools

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

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

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 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 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 deceases 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 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 in Figure 2-1.

Figure 2-1 Instruction Scheduling at -no

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

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

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

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

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

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* and *CONVEX Assembly-Language 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 reduced 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 the integer 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 2-2 shows how the compiler represents this order internally.

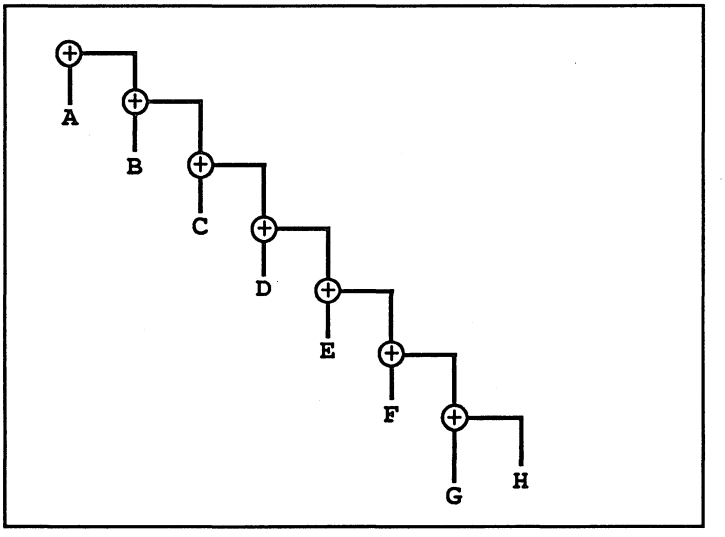


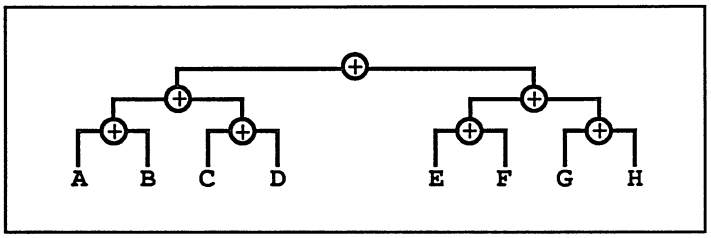
Figure 2-2
Unbalanced Tree Representation

Another way to evaluate the integer 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 2-3.

Figure 2-3
Balanced Tree Representation



In Figure 2-2, the depth of the tree is seven; in Figure 2-3, the depth of the tree is three. The machine instructions generated for the tree in Figure 2-2 execute slower than the instructions generated for the tree in Figure 2-3.

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.

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 two FORTRAN statements:

Figure 2-4 Instruction Scheduling at -O0

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

Original Code

```
ld.w  D, s0
ld.w  C, s1
ld.w  B, s2
mul.s s0,s1
add.s s1,s2
st.w  s2, T

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

Optimized Code

```
ld.w  D, s0
ld.w  C, s1
ld.w  B, s2
mul.s s0,s1
ld.w  E, s0
ld.w  F, s3
add.s s1,s2
st.w  s2, T
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 following code, for example, contains a redundant assignment, X=Y+C, which the compiler removes.

Figure 2-5 Eliminating Redundant Assignments at -O0

Original Code	Optimized Code
`X = Y + C`	`!(statement eliminated)`
`...`	`...`
`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. Figure 2-6 shows an example.

Figure 2-6 Assignment Substitution

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.

Figure 2-7 shows an example of constant propagation and folding.

Figure 2-7 Folding and Propagating Constants at `-O0`

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 underflow 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 in Figure 2-8.

Figure 2-8 Algebraic and Trigonometric Simplification

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 the same operations at the basic-block level. The scope of the optimization is now a function, subroutine, or program main section.

An example of constant propagation and folding appears in Figure 2-9.

Figure 2-9 Constant Propagation and Folding at -O1

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

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

Redundant-Assignment Elimination

At optimization level -O1, the compiler eliminates assignments to variables that do not have subsequent references within the program unit. Figure 2-10 shows how the compiler eliminates redundant assignments to the variables A and X.

Figure 2-10 Eliminating Redundant Assignments at -O1

Original Code

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

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

As shown in Figure 2-11, the compiler does not eliminate ASSIGN statements and assignments to dummy arguments, function names, and common variables.

Figure 2-11 Redundant Assignments Eliminated at -O1

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 call, the compiler eliminates the assignment and retains the function call. Figure 2-12 shows an example.

Figure 2-12 Eliminating Function Assignments at -O1

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

If the function has no side effects, the compiler eliminates the function call as well as the assignment, saving much more time. Functions that do not modify the value of an argument or common variable, read or write, or call another function have no side effects. Existing compilers cannot automatically determine whether a side effect exists and eliminates function 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 names separated by commas. The directive must precede the function call that does not contain side effects.

Caution ✎ **Do not use the NO_SIDE_EFFECTS directive on a call to a function that:**

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

For more information about the NO_SIDE_EFFECTS directive, see Appendix B.

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., then 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 a loop to a basic block preceding the loop. *Sinking* moves a store 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 in the following cases:

- 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 Figure 2-13 shows a common subexpression.

Figure 2-13 Eliminating a Common Subexpression

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 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 in Figure 2-14.

Figure 2-14 Common Subexpression Eliminated

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 Figure 2-15, the compiler determines that the subexpression must be calculated whether the condition associated with the `IF` statement evaluates to `.TRUE.` or `.FALSE.`

Figure 2-15 Eliminating a Common Subexpression in an `IF` Statement

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 shown in Figure 2-16.

Figure 2-16 Common Subexpression in IF Statement Eliminated

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 Figure 2-17, 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.

Figure 2-17 Candidate for Code Motion

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, as shown in Figure 2-18.

Figure 2-18
Code Motion
Performed

Optimized Code

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

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

Strength Reduction

In some cases, the compiler can replace an arithmetic operation with an equivalent operation that executes more quickly. Such replacements are called strength reductions. On the C100 and C200 Series machines, for example, the compiler transforms integer multiplication by 2, 4, 8, and 16 into integer shifts:

`J * 2` becomes `IISHFT(J, 1)`
`J * 4` becomes `IISHFT(J, 2)`

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

`A / 2` remains `A / 2`

Multiplication involving real constants is reduced to addition:

`X * 2` becomes `X + X`

Division by a constant is reduced to multiplication:

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

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

Strength Reduction of Induction Variables and Constants

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

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

In Figure 2-19, the compiler recognizes that I is incremented by 2 on each iteration and that X is incremented by 2*C, a loop constant.

Figure 2-19 Strength Reduction for a Loop

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
```

Figure 2-20 shows that 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.

Figure 2-20 Loop's Strength Reduced

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 C100™ and C200™ Series architectures. Vectorization converts scalar operations in loops on array elements into equivalent vector operations. The `-O2` compiler option instructs the compiler to vectorize loops in a program. For loops that cannot be vectorized, the compiler carries out the global transformations performed at `-O1`.

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

Basic Operation

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

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

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

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

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

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

Transformations the Compiler Performs

The compiler reorders the statements and instructions of a program to make it 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.

Figure 3-1 Strip Mining—Original Loop

Original Loop

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

In the vectorized loop code shown in Figure 3-2, `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.

Figure 3-2 Strip Mining—Vectorized Loop

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 in Figure 3-3 shows values of `I` and `IOUT` and the elements of `A` that are calculated.

Figure 3-3
Iterations of a Strip-Mine Loop

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.

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 outer-most loop and vectorizing each of the resulting loops or loop nests. Consider the loop in Figure 3-4.

Figure 3-4
Candidate Loop for Distribution

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 Figure 3-5.

Figure 3-5
Loop Distributed

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 most profitable vectorizable loop the innermost loop
- ❑ To make the most profitable parallelizable loop 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 in Figure 3-6.

Figure 3-6 Candidate for Loop Interchange

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 Figure 3-7, substantially improves performance over the row-by-row approach of the source code.

Figure 3-7 Interchanged Loops

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. Figure 3-8 shows a loop nest that is a candidate for hoisting and sinking.

Figure 3-8 Paired Hoist and Sink

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

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

Figure 3-9 shows an example of vector hoisting and sinking.

Figure 3-9 Vector Hoisting and Sinking

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

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

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

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

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 Figure 3-10, K is a conditional induction variable, not some upper limit on the vector.

Figure 3-10 Conditional Induction Variables

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

The compiler generates machine instructions that do the following:

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

Inhibitors of Vectorization

Any of these conditions inhibit or prevent vectorization:

- ❑ Computed or assigned `GOTO` statements
- ❑ Multiple loop entries or exits
- ❑ Function or subroutine calls
- ❑ I/O statements
- ❑ `EQUIVALENCE`d scalar or array variables
- ❑ Recurrences

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

Recurrence

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

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

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

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

Loop-Carried Dependency

A loop-carried dependency exists when one iteration of a loop computes a value that is referenced on another iteration. The loop in Figure 3-11 contains an LCD.

Figure 3-11 Loop-Carried Dependency

```
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. Figure 3-11 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 in Figure 3-12 contains a forward dependency.

Figure 3-12 Forward Loop-Carried Dependency

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

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

The compiler can vectorize some loops containing backward LCDs. The loop in Figure 3-13 contains an LCD that points backward from `B(I+1)` to `B(I)`.

Figure 3-13 Backward Loop-Carried Dependency

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

In this loop, the assignment to A(2) on the second iteration depends on the value assigned to B(2) on the first iteration. The compiler interchanges the statements within the loop so that the assignment to B occurs before the assignment to A, as shown in Figure 3-14.

Figure 3-14
Interchanged Statements

```
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. An example appears in Figure 3-15, where there is an LCD on the variable X.

Figure 3-15
Scalar Spreading

Original Loop

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

Vectorized Loop

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

In this example, the temporary vector TA replaces all references to scalar 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. Figure 3-16 shows an example.

Figure 3-16
Candidate for Partial Vectorization

Original Loop

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

As shown in Figure 3-16, the assignment to `A(I)` depends on the value of `A(I-1)`, which is computed on the previous iteration. Figure 3-17 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.

Figure 3-17 Partial Vectorization

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. Figure 3-18 shows a loop with two LIDs.

Figure 3-18 Loop-Independent Dependency

```
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 to `A`, prevents statements 2 and 3, which assign new values to `B` and `D`, from being evaluated first. Statement 1 is anti-dependent on statements 2 and 3. A forward LID exists between statements 1 and 2; another exists between statements 1 and 3.

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

An LID can stop vectorization by preventing the compiler from eliminating an LCD. In Figure 3-19, the loop cannot be vectorized.

Figure 3-19
LID Inhibiting Vectorization

```
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. Figure 3-20 shows a loop that cannot be vectorized because the sign of `K` is unknown.

Figure 3-20
Apparent Recurrence

```
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` is negative, the final value of `A(M:N-1)` is 2.0; only `A(N)` is 0.0. The compiler must interchange the statements so the assignment to `A(I+K)` follows the assignment to `A(I)`. Because these conditions are contradictory, neither operation can be performed.

The loop in Figure 3-21 cannot be vectorized because the compiler cannot determine whether a recurrence exists.

Figure 3-21
Apparent Recurrence Inhibiting Vectorization

```
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 one of the intrinsic MAX and MIN functions.

The loop in Figure 3-22 computes the sum of the elements of A(1:N) and notes the value of the greatest element. The compiler vectorizes both reductions.

Figure 3-22 Vectorization of Reductions

Original Loop	Vectorized Loop

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

When invoked with the -uo option (see Appendix A), the compiler recognizes loops that use an IF test to determine a maximum (or minimum) array element. Figure 3-23 shows such a loop.

Figure 3-23 Candidate for the -uo Option

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

Optimization Report

When you compile a program with the -O2 option, the compiler generates an optimization report for each program unit. The -or option determines the report's contents, as shown in Figure 3-24.

Figure 3-24 Optimization Report Options

-or Option	Report Contents
all	loop table and array table
loop	loop table only (default)
array	array table only
none	no report

For example, consider the matrix multiplication algorithm in Figure 3-25. (Line numbers are provided as a reference.)

Figure 3-25 FORTRAN Source

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

At line 8, individual elements `C(I,J)` are summed directly rather than stored in a temporary scalar variable. Introducing a temporary scalar later assigned to `C(I,J)` would inhibit vectorization. Figure 3-26 shows the optimization report generated by compiling the program `EXAMPLE1` at optimization level -O2.

Figure 3-26
Optimization Report

```
%fc -O2 -or all example1.f
              Optimization by Loop for Routine EXAMPLE1

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

Line     Iter.  Analysis
Num.     Var.
-------------------------------------------------------------------------
   4-1   I      Interchanged to innermost
   4-2   I      Interchanged to innermost

          Array References for Routine EXAMPLE1

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

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

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

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

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

Figure 3-27 shows an example of other transformations the compiler performs. (Line numbers are provided as a reference.)

Figure 3-27 FORTRAN Source

```
 1 FUNCTION EXAMPLE2(A,N)
 2 REAL A(N), SUM
 3
 4 SUM = 0.0
 5 DO I = 1, N
 6   SUM = SUM + A(I)
 7 ENDDO
 8 EXAMPLE2 = SUM
 9
10 END
```

Figure 3-28 shows the optimization report generated by compiling the function in Figure 3-27 for vectorization. The `-c` option suppresses loading because no main program is included.

Figure 3-28 Optimization Report

```
%fc -O2 -c example2.f
               Optimization by Loop for Routine EXAMPLE2

Line     Iter    Reordering          Optimizing / Special            Exec.
Num.     Var.    Transformation      Transformation                   Mode
-------------------------------------------------------------------------
   5     I       FULL VECTOR         Reduction
```

In the `Optimizing/Special Transformation` column, the word `Reduction` indicates that the compiler recognizes the reduction in the `I` loop and vectorizes it. The compiler strip mines the loop, creating an inner loop of up to 128 iterations. An outer loop controls repetitions of the inner loop, processing strips until the first `N` elements of array `A` have been reduced.

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 in Figure 4-1.

Figure 4-1 Matrix Multiplication

```
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.

Figure 4-2 Matrix Multiplication with Distributed Loops

```
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 so that it can retrieve contiguous elements on successive iterations.

Figure 4-3 Matrix Multiplication after Loop Interchange

```
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.

Figure 4-4 Matrix Multiplication after Strip Mining and Vectorization

```
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.

Figure 4-5 Matrix Multiplication after Second Interchange

```
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 Figure 4-6, PARALLEL DO represents a loop that can be processed by multiple CPUs.

Figure 4-6
Matrix Multiplication with Parallel Outer Loops

```
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
```

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.

Figure 4-7
Matrix Multiplication after Hoist and Sink

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

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

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

Inhibitors of Parallelization

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

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

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 Figure 4-8 prevents the compiler from automatically parallelizing the loop. The FORCE_PARALLEL directive overrides the compiler's decision, and the compiler generates parallel code for the loop.

Figure 4-8
Use of
`FORCE_PARALLEL`
and `-re`

```
  ...
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. Parallelizable code within the subroutine cannot execute in parallel. Additional code generated to parallelize the called routine is useless overhead. There is one exception: if a routine called from a parallel loop uses directives such as `BEGIN_SECTION` or `BEGIN_ORDER` to synchronize operations, it must be compiled at `-O3` to produce correct results.

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

For more information about compiler directives, see Appendix B.

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 Figure 4-9 has no dependencies. The compiler can strip mine and vectorize the inner loop and parallelize the strip-mine loop.

Figure 4-9
Loop Without Dependencies

```
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 in Figure 4-10 has a backward loop-carried dependency (LCD) caused by the assignment to `A(I+1)`. The loop cannot be vectorized or parallelized by the compiler. The loop remains in scalar form.

Figure 4-10
Loop with Backward LCD

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

The loop in Figure 4-11 has a forward LCD. Because forward LCDs do not interfere with vectorization, the compiler strip mines and vectorizes the loop. It is not safe to parallelize a loop that has an LCD, however. The result is a strip-mine vector instead of a parallel vector loop.

Figure 4-11
Loop with Forward LCD

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

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

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

The compiler can handle most scalar assignments and reductions within parallel loops. For example, the compiler can generate parallel code for the loop in Figure 4-12.

Figure 4-12 Scalar Assignments and Reductions in Parallel Loops

```
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.

Figure 4-13 shows how to insert tasking directives into a section of code containing three tasks that can be run in parallel.

Figure 4-13 Use of Tasking Directives

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

The compiler transforms this code into a parallel loop and creates machine code equivalent to that shown in Figure 4-14.

Figure 4-14 Code Transformed with Tasking Directives

```
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
```

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 `csd` to pinpoint and fix the logic error that is causing the problem. See Chapter 9 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 (described in Appendix E of the *CONVEX FORTRAN User's Guide*) 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 offending routine.

   When you have isolated the offending 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. 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 3a 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 in **Step 3a;** 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 in **Step 3a.**

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.

Step 3a. Add Selective Vectorization

1) Look at the CXpa output from **Step 2** to determine which routines consume the most CPU time. Compile the most time-consuming routines for vectorization. To do this, place the `OPTIONS -O2` statement above the subprogram in the source code or use the `-O2` option on the `fc` command line. Compile the rest of the program for program-unit optimization and CXpa profiling.

2) Compare the output of your program with the output produced in **Step 2**. The results, allowing for floating-point roundoff, should be unchanged.

   If the output is significantly changed, use the binary search procedure described in **Step 2** to isolate the offending routine. Check the source code and fix any errors that you find. If you do not find any logic errors, recompile the affected subprogram at `-O1` and continue optimizing the rest of the program.

3) Run the program under CXpa. Take note of the program's total execution time and the most time-consuming routines. Compare this CXpa output with the CXpa output from **Step 2** and determine the effect of vectorization on your program's performance.

4) Repeat **Step 3a**, vectorizing routines that consume a significant amount of CPU time in the new CXpa output and have not been vectorized. Continue until you have vectorized all time-consuming routines that can be properly vectorized; proceed to **Step 4**.

Step 4. Enhance Vector Optimization

1) Run the vectorized program under CXpa to produce a loop-level profile of the most time-consuming routines.

2) Study the CXpa profile and the optimization report. Look for loops that are not vectorized and consume significant amounts of CPU time. Note which of these loops are inner loops, which are candidates for vectorization.

   The goal is to increase the number of vectorized loops. Look for apparent recurrences that prevent the compiler from vectorizing time-consuming loops. If you find loops

with apparent recurrences that do not contain actual recurrences, use the NO_RECURRENCE directive to tell the compiler it is okay to vectorize the loop.

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

3) When you are satisfied that no more loops can be vectorized or the loops that can be vectorized do not consume a significant amount of time, you may still be able to improve the efficiency of your code. Try the following techniques:

   - ❑ Simplify conditionals. Even if a loop is vectorized, an embedded conditional can slow it down.
   - ❑ Simplify array subscripts. Array subscripts that require many operations to evaluate can slow down the execution of a loop.
   - ❑ Look for loops with short vector lengths (small trip counts). If the trip count is small, the loop probably runs faster in scalar form than it does in vector. On the C100 and C200 Series, this slowdown occurs when the trip count is less than five. Use the SCALAR directive to stop the compiler from vectorizing such a loop.
   - ❑ Look for unnecessary strip mines and inefficient strip-mine lengths. Use CXpa to determine whether a vector loop is strip mined. Use the MAX_TRIPS directive to stop the compiler from creating unnecessary strip mines around a vector loop.
   - ❑ Consider inlining any short routines that are called frequently or consume a large amount of CPU time. See Chapter 8 for information about inlining.

   For more examples of how to tune your code for better vector performance, refer to Chapter 7, "Manual Optimization Techniques."

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

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

Note ✎ **Automatic vectorization typically reduces CPU time by up to 90 percent. 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. Adding 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.

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 5a**. 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 5a. Adding 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 *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. Enhancing Parallel Optimization

1) Run the parallelized program under CXpa to produce a loop-level profile of the most time-consuming routines.

2) Study the CXpa profile and the optimization report. Look for loops that failed to parallelize. A scalar loop that consumes significant CPU time is a candidate for parallelization. Inner loops are less likely candidates.

3) Look for apparent dependencies that stop the compiler from parallelizing a scalar or vector loop. Remove an apparent dependency by inlining the subprogram call or by applying the `NO_RECURRENCE` or `FORCE_PARALLEL` directive. If you find a real dependency, consider replacing the routine with a call to a VECLIB™ subprogram that performs the same function in parallel. For more information about VECLIB, see the *CONVEX VECLIB User's Guide.*

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

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

Step 7. Wrapping 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. Figure 6-1 shows four typical counted loops.

Figure 6-1 Typical Counted Loops

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

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

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

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

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

If a loop uses an iteration variable that is not incremented or decremented by a constant nonzero integer value, the loop has no induction variable and the compiler cannot vectorize it. Figure 6-2 shows a loop that has no induction variable.

Figure 6-2
Loop Without Induction Variable

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

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

The loop in Figure 6-2 can be unrolled by hand, as shown in Figure 6-3. Because the loop overhead is eliminated, the unrolled code runs faster than the original loop.

Figure 6-3
Unrolled Loop

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

If the iteration variable of a loop is incremented by a non-integer constant, the loop has no induction variable and the compiler cannot vectorize it. The loop in Figure 6-4, for example, increments `I` by a `REAL` value, which prevents vectorization of the loop.

Figure 6-4
Loop with `REAL` Iteration Value

```
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.`, `.NE.`

More complicated iteration tests, such as the one shown in Figure 6-5, often prevent the compiler from vectorizing a loop.

Figure 6-5 Complicated Iteration Test

```
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 in Figure 6-5 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. Figure 6-6 shows a loop whose stop value changes within the loop.

Figure 6-6 Loop without Fixed Stop Value

```
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 in Figure 6-7, do not vectorize.

Figure 6-7 Loop with Alternate Exit

```
    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 instead of within the loop.

Figure 6-8 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.

Figure 6-8 Embedded Conditional

```
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 in Figure 6-9. This change to the source code improves performance dramatically.

Figure 6-9
Eambedded
Conditional
Removed

```
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
```

Figure 6-10 shows an example of boundary tests that can be removed from a loop.

Figure 6-10
Boundary Test
Within a Loop

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

When boundary values are set outside the loop, as shown in Figure 6-11, this code fragment runs several times faster.

Figure 6-11
Boundary Test
Outside a Loop

```
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 left-most, dimension is faster than using innermost loops that vary the trailing, or right-most, 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. Figure 6-12 shows three loops in order of increasing efficiency.

Figure 6-12
Loops in Order
of Increasing
Efficiency

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

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

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

In Figure 6-13, the compiler interchanges the `J` and `I` loops.

Figure 6-13 Optimization of Array Accesses

Original Code	Optimized Code

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

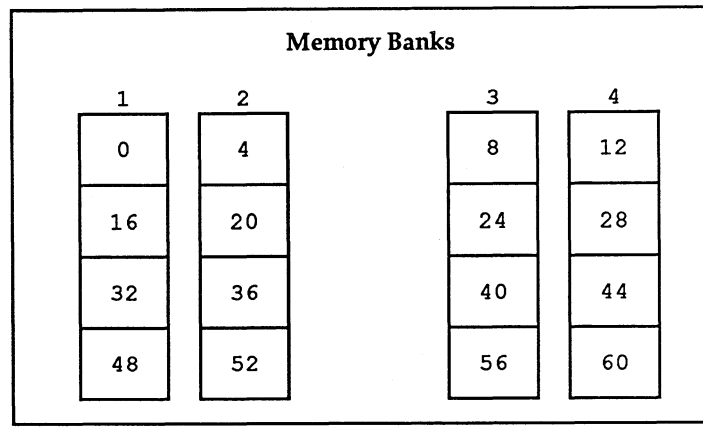
If the trip count of an outer loop is much smaller than that of the inner loop, the compiler may not interchange the loops even though it could achieve more efficient memory accesses by doing so. The compiler realizes that a few slow memory accesses can be faster than many fast accesses. If the compiler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

Memory Interleaving

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

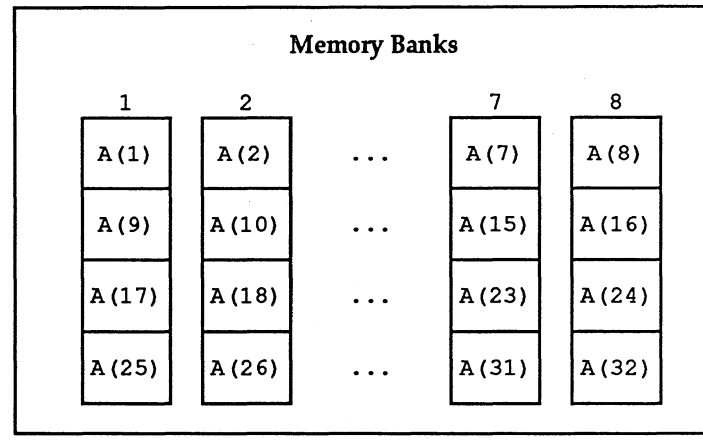
Main memory comprises at least four banks of dynamic RAM. The memory system stores data so that contiguous words are in separate memory banks. This is called *memory interleaving*. One memory bank is accessed on each clock cycle. As a result, sequential requests to ascending banks proceed at full speed. Figure 6-14 shows the configuration on a four-bank machine. Byte addresses are expressed in decimal notation.

Figure 6-14
Four-Way
Interleaved
Memory



Eight memory banks are needed to return data at the rate of one word per clock cycle. A load instruction, for example, takes eight cycles to return data. If a program makes eight load requests, at a rate of one request per clock cycle, each to a separate bank, data returns at a rate of one per clock cycle, beginning eight clock cycles after the first request. Memory interleaving directly affects efficient array accesses. Figure 6-15 shows a one-dimensional array in eight-way interleaved memory.

Figure 6-15
One-Dimensional
Array in Eight-
Way Interleaved
Memory



The loop in Figure 6-16 processes an array sequentially. After an initial wait of eight clock cycles, the CPU receives one data word per clock cycle.

Figure 6-16
Sequentially
Processed Array

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

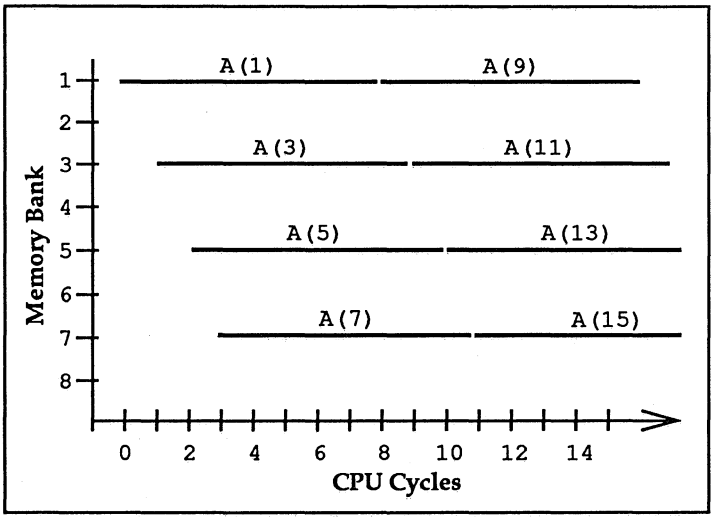
The loop in Figure 6-17, however, causes memory bank conflicts. The CPU must wait for memory requests to be filled.

Figure 6-17
Sequentially Processed Array

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

Figure 6-18 shows the timing relationships that cause these bank conflicts.

Figure 6-18
Bank Conflict



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

Memory bank conflicts occur when an array's stride does not efficiently use the memory of the computer. An array's stride is the difference in the index value between two successive iterations. In the loop in Figure 6-19, `A` has a stride of one.

Figure 6-19
Array With Stride of One

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

The loop in Figure 6-20, which accesses every other element of the array, has a stride of two.

Figure 6-20 Array With Stride of Two

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

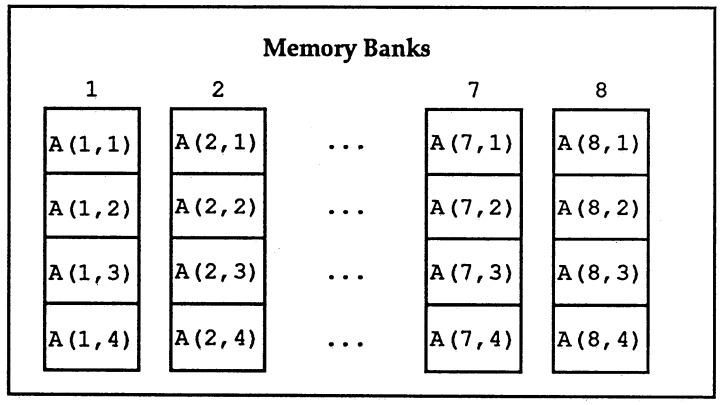
Arrays with a stride of two use only half the memory banks. Arrays with a stride of four use one bank in four. Whenever possible, avoid writing loops with a stride that is a multiple of a power of two. Odd strides give better performance than even strides do.

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 6-21 shows how a two-dimensional, eight-by-four column-major array is stored in memory with eight-way interleave.

Figure 6-21 Two-Dimensional Array Stored in Eight-Way Interleaved memory



The loop in Figure 6-22 processes a two-dimensional array.

Figure 6-22
Loop to Process a Two-Dimensional Array

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

When the inner loop is vectorized, the vector register load and vector store have a stride of eight. Only one memory bank is used in the inner loop, as shown in Figure 6-21, and eight clock cycles are required to load each element into the vector register.

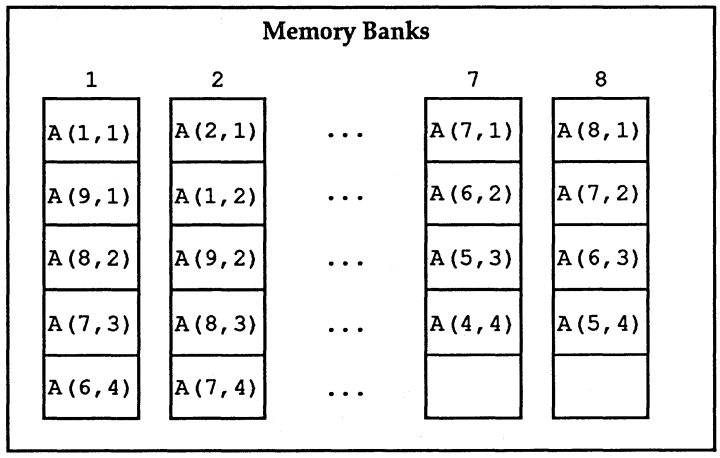
To avoid bank conflicts, declare the leading index of `A` to be an odd number, as shown in the loop in Figure 6-23.

Figure 6-23
Loop to Process a Two-Dimensional Array

```
      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 6-24 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 6-24
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. Figure 7-1 shows a loop with a trip count that varies between 1 and 50.

Figure 7-1 Unnecessary Strip Mine

```
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 in Figure 7-2.

Figure 7-2
Strip Mine Removed

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

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

Do Not Vectorize Loops with Small Trip Counts

Look for loops with small trip counts. On the C100 and C200 machines, a loop with a trip count less than five is usually not worth vectorizing. The compiler vectorizes loops with trip counts greater than two. For loops with variable trip counts or trip counts between three and five, you can use the SCALAR directive to prevent vectorization or the UNROLL directive to unroll the loop. Figure 7-3 shows such a loop.

Figure 7-3
Trip Count of Four

```
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 Figure 7-4 because it cannot determine the trip count.

Figure 7-4
Variable 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. Figure 7-5 shows an example.

Figure 7-5
Optimized for Variable Trip Counts

```
      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. Figure 7-6 shows the use of the SELECT directive.

Figure 7-6
SELECT Directive

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

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

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

Scalar loops with small constant trip counts can be more efficient if the loops are unrolled. Unrolling replaces a loop with a linear sequence of statements. Figure 7-7 shows such a loop and how it is unrolled.

Figure 7-7
Loop Unrolling

Original Loop	**Loop Unrolled**
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 works only if the compiler can determine that the trip count is less than five. If a loop has a variable trip count, you can partially unroll it by hand.

Promote an Array

Sometimes it is necessary to promote an array to a higher dimension to vectorize a loop. In Figure 7-8, only the J loop vectorizes. The compiler is unable to vectorize the I loop because of a recurrence. Values assigned to Q within the J loop depend on values assigned to array B by the four preceding statements. Those values of array B exist only until the next iteration of the I loop. There is a cycle of antidependencies between assignments to B(N). This cycle prevents the compiler from distributing the I loop.

Figure 7-8 Dependencies Prevent Loop Distribution

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

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

To eliminate the recurrence, promote B to a two-dimensional array, as shown in Figure 7-9.

Figure 7-9
Array Promoted to Two Dimensions

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

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

In the modified code, the calculation of `B(N,I)` is independent of the calculation of `B(N,I+1)`. These independent calculations permit the compiler to distribute the `I` loop, as shown in Figure 7-10.

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 Figure 7-11. The loop has a series of embedded `IF` tests that slow it down.

Figure 7-10 Distributed I Loop

```
 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
```

Figure 7-11 Conditional Within a Vector Loop

```
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 in Figure 7-12.

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

Figure 7-12
Conditional
Removed

```
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
```

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.

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

How to Use Inlining

Inlining is done in two steps:

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

Creating `.fil` Files

To create `.fil` files, follow these steps:

- ❑ Place the subprograms to be inlined in a source file separate from the program `MAIN` section and other subprograms.
- ❑ Use the `-il` option when you compile the files containing subprograms to be inlined.

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

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

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

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

Using the -is Option

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

```
-is <dir> [ -is <dir> ... ]
```

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

The compiler attempts to inline every .fil file found in the specified directories. If you do not want to inline specifc .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 which:

- 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 reentrancy, 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 `SAVE`d 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.

Limits of Optimization 9

Optimization changes the order in which instructions execute. In some cases, however, improper optimization can 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:

- ❑ Illegal (nonstandard) code
- ❑ Floating-point imprecision (roundoff error)
- ❑ Misused directives and options
- ❑ Compiler limitations
- ❑ Nondeterminism of parallel execution

Invalid 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 *ANSI FORTRAN 77 Standard* 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 read the *ANSI FORTRAN 77 Standard* quote properly, it is helpful to understand what the *Standard* means by the phrase, "may become defined."

> Variables, array elements, and substrings become defined as follows:
>
> (1) Execution of an arithmetic, logical, or character assignment statement causes the entity that precedes the equals to become defined.
> (2) As execution of an input statement proceeds, each entity that is assigned a value of its corresponding type from the input medium becomes defined at the time of such assignment.
> (3) Execution of a DO statement causes the DO variable to become defined.
> (4) Beginning of execution of action specified by an implied-DO list in an input/output statement causes the implied-DO-variable to become defined.
> (5) A DATA statement causes entities to become initially defined at the beginning of execution of an executable program.
> (6) Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.
> (7) When an entity of a given type becomes defined, all totally associated entities of the same type become defined except that entities totally associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed.
> (8) A reference to a subprogram causes a dummy argument to become defined if the corresponding actual argument is defined with a value that is not a statement label value. Note that there must be agreement between the actual argument and the dummy argument.
> (9) Execution of an input-output statement containing an input/output status specifier causes the specified integer variable or array element to become defined.

(10) Execution of an INQUIRE statement causes any entity that is assigned a value during the execution of the statement to become defined if no error condition exists.
(11) When a complex entity becomes defined, all partially associated real entities become defined.
(12) When both parts of a complex entity become defined as a result of partially associated real or complex entities becoming defined, the complex entity becomes defined.
(13) When all characters of a character entity become defined, the character entity becomes defined.

The program in Figure 9-1 contains hidden aliases that are harder to find than those in the ANSI Standard example.

Figure 9-1 Hidden Alias Example 1

```
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 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.

Figure 9-2 shows this program compiled and run at optimization levels -O0 and -O1. Note that the answers are changed at optimization level -O1.

Figure 9-2
Program ALIAS—
Compile and Run

```
% 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
```

Figure 9-3 shows another example of the hidden alias problem.

Figure 9-3
Hidden Alias
Example 2

```
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 = 1, N - 1
  Z(I) = Y(I) + 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, it produces incorrect answers.

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 *ANSI FORTRAN 77 Standard* 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

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 *ANSI FORTRAN 77 Standard* 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 can cause a problem. 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, investigate using a better 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 "Recurrence" in Chapter 3.

The `MAIN` section of the program in Figure 9-4 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.

Figure 9-4 Apparent Recurrence on I Loop

```
PROGRAM MAIN
REAL A(1025), B(1025)
COMMON /DATA/ A, B
DO J = 1, 1025
  A(J) = I
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 in Figure 9-5. This directive tells the compiler to ignore the apparent recurrence and vectorize the I loop.

Figure 9-5 Proper Use of NO_RECURRENCE

```
      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
```

Correct results obtained with vectorization does not imply that correct results will be obtained with parallelization. Using the FORCE_PARALLEL directive on this loop, as shown in Figure 9-6, 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.

Figure 9-6
Improper Use of
FORCE_PARALLEL

```
      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 reentrant routine must be set to their initial values during each execution of that routine.

Compiler Limitations

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 Figure 9-7 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.

Figure 9-7
Reduction
and Apparent
Recurrence

```
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 Figure 9-8.

Figure 9-8 Reduction and Apparent Recurrence Resolved

```
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 = I, 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 Figure 9-9, both reduction and recurrence are handled correctly without your intervention.

Figure 9-9 Reduction and Recurrence on Separate Variables

```
      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. Figure 9-10 shows three loops that iterate by zero.

Figure 9-10 Three Loops Iterating by Zero

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

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

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

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

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

Nondeterminism of Parallel Execution

In a parallel program, threads do not execute in a predictable or determined order. If you force the compiler to parallelize a loop when a dependency exists, the results are unpredictable and can vary from one execution to the next. Because the results depend on the order in which statements execute, the errors are nondeterministic. Unless you are sure that no loop carried dependency exists, it is safer to let the compiler choose which loops to parallelize.

Conditional Vectorization

A vectorized loop may fail if the indexes for a conditionally referenced array fall outside the array's bounds. Figure 9-11 shows an example.

Figure 9-11
Conditional
Vectorization

```
      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
```

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 C100 and C200 Series machines) usually runs faster if the loop is not vectorized. The `SCALAR` directive can prevent the compiler from vectorizing such loops. The `SELECT` directive

tells the compiler to generate multiple versions of a loop and code to allow dynamic (runtime) selection of the best version. Using the SELECT directive, you can specify optimum cutoff points for scalar, vector, and parallel processing.

Complicated Conditionals

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

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

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

The -uo Option

A

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

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

Simple Strength Reduction

Chapter 2 describes how the compiler replaces slow operations with faster ones on the assumption that arithmetically equivalent expressions always yield the same results.

Reducing an expression such as X/C to (1/C)*X can be unsafe because it can increase roundoff error.

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

Code Motion

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

Pattern Matching

Pattern matching allows the compiler to vectorize certain loops that it cannot otherwise vectorize. The compiler recognizes loops that use an `IF` test to determine a maximum (or minimum) value stored in an array. Figure A-1 shows such a loop.

Figure A-1 Pattern Matching `IF` Tests

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

The compiler recognizes loops containing recurrences that can be implemented with a special sequence of vector instructions. Figure A-2 shows examples of patterns the compiler matches.

Figure A-2 Pattern Matching Recurrences

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

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

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

Conversion Elimination

When you use the `-uo` option, the compiler eliminates costly type conversions by creating `REAL` induction variables. Consider the loop in Figure A-3.

Figure A-3 Eliminating Type Conversion

Original Loop

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

Figure A-4 shows the optimized loop. At optimization level `-O2`, the compiler vectorizes the optimized loop.

Figure A-4 Type Conversion Eliminated

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
```

Compiler Directives

B

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

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

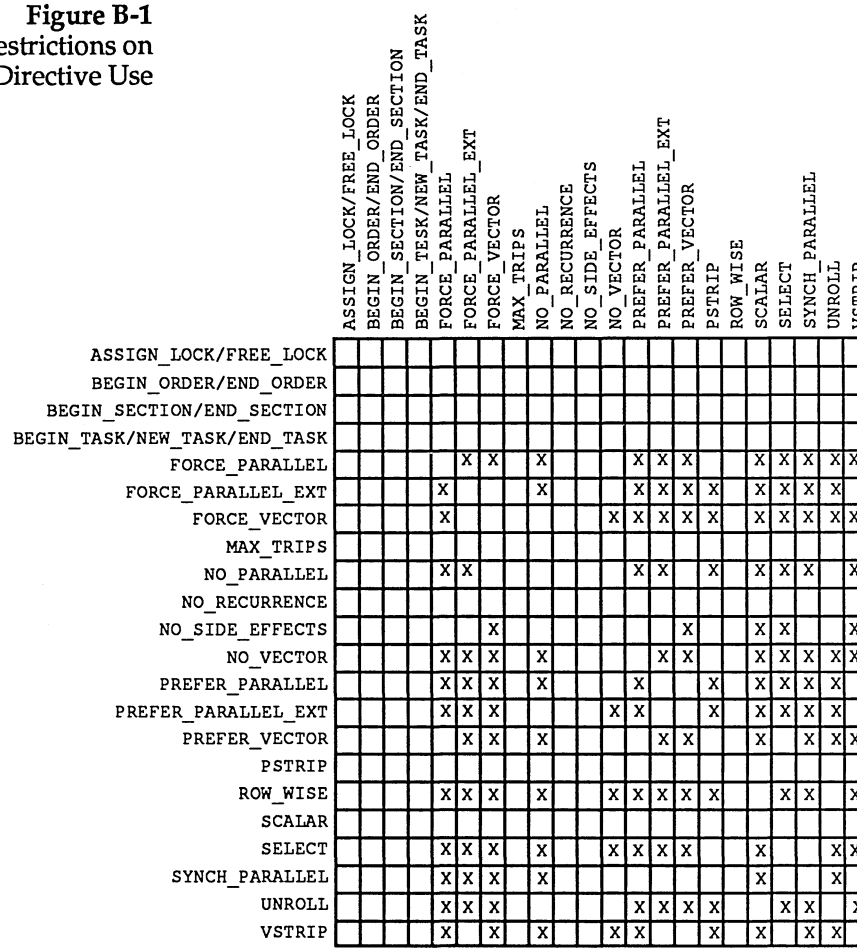
If you specify more than one directive, separate them with commas. A directive must fit on one line; it cannot be continued. A directive can be surrounded by any number of comment lines.

CONVEX FORTRAN supports these directives:

- ASSIGN_LOCK, FREE_LOCK
- BEGIN_ORDER, END_ORDER
- BEGIN_SECTION, END_SECTION
- BEGIN_TASKS, NEXT_TASK, END_TASKS
- FORCE_PARALLEL
- FORCE_PARALLEL_EXT
- FORCE_VECTOR
- MAX_TRIPS
- NO_PARALLEL
- NO_RECURRENCE
- NO_SIDE_EFFECTS
- NO_VECTOR
- PREFER_PARALLEL
- PREFER_PARALLEL_EXT
- PREFER_VECTOR
- PSTRIP
- ROW_WISE
- SCALAR
- SELECT
- SYNCH_PARALLEL
- UNROLL
- VSTRIP

Certain combinations of directives are invalid when used within the same program unit or loop and cause the compiler to issue a warning. The X's in Figure B-1 denote invalid combinations of directives.

Figure B-1
Restrictions on
Directive Use



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

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

ASSIGN_LOCK
FREE_LOCK

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

The formats of these directives are:

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

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

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

BEGIN_ORDER
END_ORDER

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

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

The formats of these directives are:

```
BEGIN_ORDER (<lockname>)
END_ORDER
```

where <lockname> is the name of a locking variable that has been assigned with ASSIGN_LOCK. Figure B-2 shows an example of how to use these directives.

Figure B-2 Use of BEGIN_ORDER and END_ORDER

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

BEGIN_SECTION END_SECTION

The BEGIN_SECTION directive identifies an unordered critical region. An unordered critical region begins with a BEGIN_SECTION directive and ends with an END_SECTION directive.

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

```
BEGIN_SECTION (<lockname>)
END_SECTION
```

where <lockname> is the name of a locking variable that has previously been assigned by the ASSIGN_LOCK directive.

An example of how to use this pair of directives appears in Figure B-3.

Figure B-3
Use of
BEGIN_SECTION
and
END_SECTION

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

BEGIN_TASKS
NEXT_TASK
END_TASKS

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

The code in Figure B-4 shows the use of the tasking directives.

Figure B-4
Use of
BEGIN_TASKS,
NEXT_TASK, and
END_TASKS

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

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

FORCE_PARALLEL

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

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

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

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

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

An example of how to use FORCE_PARALLEL appears in Figure B-5.

Figure B-5 Use of FORCE_PARALLEL

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

FORCE_PARALLEL_EXT

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

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

FORCE_PARALLEL_EXT allows the compiler to interchange outer loops for vectorization.

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

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

FORCE_VECTOR

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

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

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

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

MAX_TRIPS

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

```
MAX_TRIPS (<n>)
```

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

NO_RECURRENCE

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

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

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

When you use NO_RECURRENCE and the compiler finds a recurrence, the compiler breaks the recurrence by removing one or more dependencies of the cycle. In Figure B-6, if J is positive, no recurrence exists.

Figure B-6 Use of NO_RECURRENCE

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

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

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

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

NO_SIDE_EFFECTS

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

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

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

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

Place the directive before the call to the named function. Use the directive if the compiler gives the advisory message "More optimization is possible if this function call has no side effects," and the function is really one with no side effects. If no arguments exist, the directive applies to all functions following the directive. Figure B-7 shows an example of how to use this directive.

Figure B-7 Use of NO_SIDE_EFFECTS

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

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

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

NO_PARALLEL

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

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

NO_VECTOR

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

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

PREFER_PARALLEL

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

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

PREFER_PARALLEL_EXT

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

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

PREFER_VECTOR

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

PSTRIP

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

```
PSTRIP(<integer_constant>)
```

where <integer_constant> specifies the strip-mine length.

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

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

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

You cannot use PSTRIP with vector loops.

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

ROW_WISE

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. ROW_WISE tells the compiler to reverse the dimensions of the designated arrays. The format of this directive is

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

The following rules apply to using the ROW_WISE directive:

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

Figure B-8 shows a code segment containing a candidate for the ROW_WISE directive.

Figure B-8 Candidate Loop for ROW_WISE Directive

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

Although the example in Figure B-8 vectorizes, it runs slowly because the array is being accessed through non-contiguous memory. The array is now being accessed from contiguous memory, increasing the execution speed. Using ROW_WISE makes the accesses contiguous.

SCALAR

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

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

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

In Figure B-9, the compiler normally interchanges the I and J loop so that elements of A, B, and C are accessed contiguously. The SCALAR directive ensures that the loop of greater iteration count is retained as the innermost loop.

Figure B-9
Use of SCALAR

```
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 Figure B-10, neither iteration count is sufficient to warrant vectorizing the loops.

Figure B-10
Use of SCALAR
With Small
Iteration Count

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

SELECT

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

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

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

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

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

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

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

The statement

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

is equivalent to

```
C$DIR SCALAR
```

SYNCH_PARALLEL

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

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

On a machine with four processors, the loop in Figure B-11 might run faster parallelized and synchronized than if it were partially vectorized and the recurrence placed in a scalar, nonparallel loop.

Figure B-11 Use of SYNCH_PARALLEL

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

UNROLL

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

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

VSTRIP

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

The format of this directive is:

```
VSTRIP(<integer_constant>)
```

where <integer_constant> specifies the strip mine length, which must be less than or equal to 128.

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

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

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

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

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

Vector Operations

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 Architecture Reference.*

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. C100 and C200 Series machines have eight vector registers.

Vector-Length Register

C100 and C200 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 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 Figure C-1.

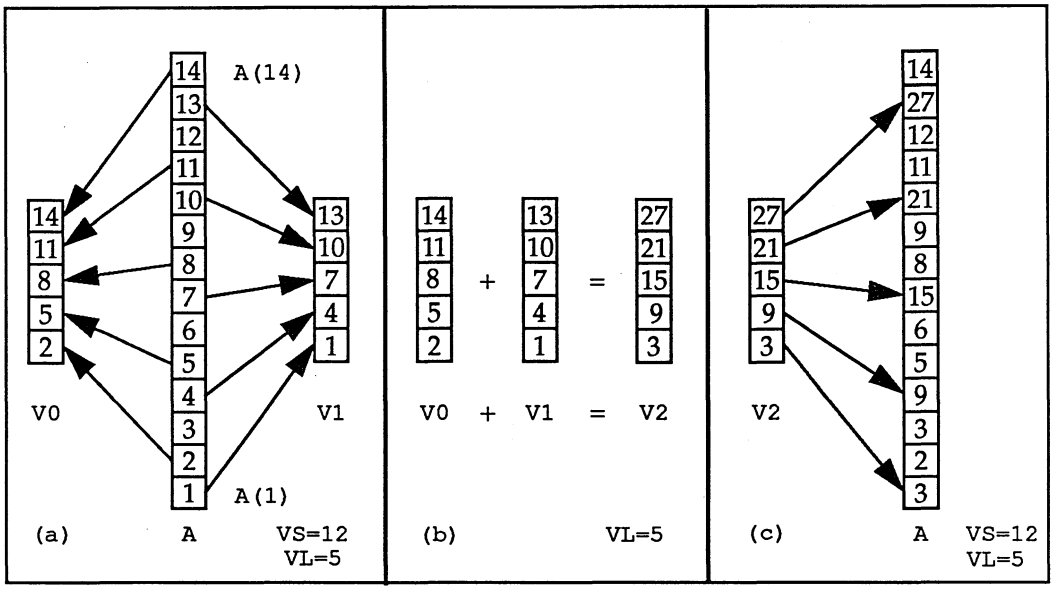
Figure C-1
Vector Operation

```
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 C-2 shows the vector operations on array A.

Figure C-2
Vector Operations for `A(I) = A(I + 1) + A(I)`



In (a), the CPU sets the vector-stride register to 12 (the number of bytes between elements of the array). In (b), the CPU has set the vector-length register controlling the operation to five. 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.*

Vector Load

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

For example, suppose the FORTRAN code on the left in Figure C-3 is vectorized.

Figure C-3
Vector Load
Example 1

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, 25, into VL. The second statement loads four into VS because each element in the array requires four bytes for storage and the loop's stride is one. The last statement loads the contents of array A into vector register V0.

Figure C-4 shows another example of vector load.

Figure C-4
Vector Load
Example 2

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

Some operations that appear to require a load statement use other instructions instead, as shown in Figure C-5.

Figure C-5
Scalar Extend

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 as 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.

Figure C-6 shows an example of vector store.

Figure C-6
Vector Store
Example 1

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 is 100 because 100 elements are loaded and stored. The VS register is four 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.

Figure C-7 shows another example of vector store.

Figure C-7
Vector Store
Example 2

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 eight because only every other element of array `C` is modified. The loop's stride is two, 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.

Figure C-8 shows the use of the vector add operator.

Figure C-8
Vector Addition

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 the fifth and sixth statements can be chained together because they map to different functional units.

The FORTRAN code in Figure C-9 computes the product of a vector and a scalar.

Figure C-9
Vector
Multiplication

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 in Figure C-10 generates a sum reduction.

Figure C-10
Vector 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 Figure C-11 computes a vector's maximum.

Figure C-11
Vector 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 in Figure C-12 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 Figure C-12, a dot-product operation requires the sum of a series of products.

Figure C-12
Dot Product
Operation

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

The assembly language for the dot-product operation is shown in Figure C-13.

Figure C-13
Dot Product
Vector Code

```
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 in the section, "VM Operations Under Mask— C200."

Consider the vector comparison in Figure C-14.

Figure C-14
Vector Maximum

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—C200

Only the C200 Series computers can perform vector operations under mask. The C100 Series computers can perform vector operations and mask operations, but multiple vector instructions must replace an individual vector operation under mask on the C200 Series computer.

Most vector operations can operate under mask. A vector-merge register bit is associated with each vector register element. When an operation is performed under mask, each element is either included or excluded from the operation based on the state of its corresponding VM bit.

In this mode, the bit of the VM register corresponding to each vector element is examined to either enable or disable that vector element from the operation.

The two forms of vector operations under mask are:

- True—Elements with VM bit equal to one are included. Instructions of this type have a `.t` suffix, such as `add.w.t`.
- False—Elements with VM bit equal to zero are included. Instructions of this type have a `.f` suffix, such as `div.b.f`.

The statement

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

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

The statement

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

adds only elements whose corresponding VM bits are one. Elements of `v2` whose corresponding VM bits are zero remain unmodified.

The complement of VM bits is used for `.f`, as in the statement

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

This version operates only on vector elements whose corresponding VM bits are 0.

For the remaining examples of operations under mask, assume these values before instructions are executed:

```
v0 = 0 1 2 3 4 5     VL = 6
v1 = 6 7 8 9 2 3     VM = 0 1 1 0 0 1
v2 = 5 5 5 5 5 5
```

The statement

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

produces

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

The statement

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

produces

```
v2 = 6 5 5 12 6 5
```

The FORTRAN code in Figure C-15 is an example of using operations under mask.

Figure C-15 Vector 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 C200 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 C200 Series computers may produce an assembly listing that differs from the C100 Series computers.

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 one.

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 Figure C-16 cannot be determined until the program is executed.

Figure C-16 Conditional Test and Masking

```
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 Figure C-17 uses indirect addressing.

Figure C-17 Indirect Array Addressing

```
SUBROUTINE GATHER (A, IA, B, IB)
INTEGER I, A(100), IA(100), B(100), IB(100)

DO I = 1, 100
  A(IA(I)) = B(IB(I)) + 1
ENDDO

PRINT *, (A(I), I = 1, 100)
RETURN
END
```

The assembly language that performs this function is shown in Figure C-18.

Figure C-18
Indirect Array Addressing: Assembly Code

```
ld.w    #1,s3
ld.w    #-4,a3
L3:
st.w    s3,LU+404
st.w    s3,-536(fp)
st.w    a3,-524(fp)
mov     ap,a5
ldea    LC+4,ap
calls   @12(a5)
ld.w    12(fp),ap
ld.w    -524(fp),a3
mov     a3,a5
add.w   fp,a5
add.w   #4,a3
st.w    s0,-508(a5)
ld.w    -536(fp),s0
add.w   #1,s0
le.w    #396,a3
mov.w   s0,s3
jbra.f  L3

ld.w    #100,VL
ld.w    #4,s0
ld.w    8(ap),a5
ld.w    4(ap),a1
add.w   #-4,a5
ld.w    0(ap),a2
ld.w    #1,s1
add.w   #-4,a2
ld.w    #4,VS
ld.w    -512(fp),v0
mul.w   v0,s0,v1
ldvi.w  v1,v2
ld.w    0(a1),v0
mul.w   v0,s0,v1
add.w   v2,s1,v3
mov     a2,a5
stvi.w  v3,VL
```

The three steps to this operation are:

1. Load the indirectly addressed elements of array `B` into a vector.
2. Increment each element by one.
3. Store the values into the indirectly addressed elements of array `A`.

The compiler accomplishes step 1 by loading the contents of array `IB` into a vector register (for example, `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)`.

Glossary

D

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.

bank conflict An attempt to load two elements concurrently from the same memory bank. On CONVEX C200 Series machines, each memory board is divided into four 64-bit memory banks. Arrays are stored in main memory across all available banks.

Loading each array element takes eight clock cycles, during which time no other element can be retrieved from the same bank. Storing contiguous array elements across four memory banks allows each of the three intervening clock cycles to be used for loading another element.

basic block A linear sequence of statements that ends with a conditional or unconditional branch. A basic block is the optimization unit considered at optimization level `-O0`. A subprogram contains at least one basic block and typically contains many. The following subroutine is divided into basic blocks:

```
        SUBROUTINE SUB(A, B, N)
        REAL A, B(N), TMP
Comment:  Begin basic block 1
        TMP = A
        IF (A .GE. 0) GOTO 10
Comment:  Begin basic block 2
        A = -A
   10   RETURN
        END
```

balancing See *tree-height reduction.*

chaining See *vector chaining.*

chime A chained vector time. The time required to perform the simultaneous instructions of one vector chain. On the CONVEX C100 and C200 Series machines, this is equal to the vector length plus 10 clock cycles.

coarse-grained See *granularity.*

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

conditional induction variable Loop induction variables that are not incremented on every iteration

constant folding Replacement of an operation on a constants 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

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

execution stream A series of instructions executed by a CPU

functional unit A part of the CPU that performs a set of operations on quantities stored in registers

fine-grained See *granularity.*

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.

granularity The amount of work executed in a single thread, between the time it is created and the time it terminates. Granularity ranges from half the entire program (coarse), to the single iterations of a loop (fine), to individual source statements (very fine). The overhead, or system time required to create and manage multiple threads, determines the granularity of parallelization that is profitable.

hoist An optimization process that moves a load from within a loop to the basic block preceding the loop

interleaved memory Memory that is divided into multiple banks to permit concurrent memory accesses

oversubscript An array reference that falls outside declared bounds

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 or loop invariant 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 computation An operation that yields the same result on every iteration of a loop

mask, vector 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.

memory bank conflict See *bank conflict*.

merge, vector 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.

mutual exclusion A protocol that prevents access to a given resource by more than one thread at a time

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

recurrence A cycle of dependencies among the operations within a loop. (See *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.

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.

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.

vector accumulator register (V) A vector register that can contain from 0 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-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.

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.*

Bibliography

E

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.

Index

A

B

C

D

E

F

G

H

I

J

P

R

S

T

U

V

W

Z

About CONVEX

In 1984, CONVEX Computer Corporation introduced the world's first supercomputer under $1 million. Today, the company leads the industry with its extensive product offering, innovative architecture, advanced system software, availability of third-party applications programs, international distribution, and installed base of supercomputers.

CONVEX markets its products primarily for scientific, engineering, and technical applications in areas such as geophysical research, aerospace simulations, computational chemistry, computer-aided engineering, image processing, and molecular biology. Through CONVEX's supercomputers, users can access leading-edge technology. These powerful systems run computationally intensive software much faster than other systems in the same price range.

CONVEX supercomputers take full advantage of integrated vector and scalar architecture. Multiprocessor versions use a unique technology, Automatic Self Allocating Processor (ASAP™). ASAP technology allows multiple jobs to run quickly and efficiently while boosting the performance of a single job. ASAP technology allows other jobs or parts of jobs to run concurrently with a parallel job when processor time is free.

CONVEX supercomputers run ConvexOS, a POSIX compliant implementation of the Berkeley UNIX® operating system, with extensions for supercomputer environments. Over 450 third-party software packages take advantage of CONVEX's advanced architecture. Additionally, CONVEX has developed the CONVEX-to-VAX User Environment (COVUE™), which enables VAX/VMS® users to be productive immediately in the UNIX environment.

CONVEX supercomputers can be integrated into existing computing environments with Ethernet™, Sun's Network File System™, UltraNet®, DECnet®, Apollo's Network Computing System™, TCP/IP, and IBM® communication protocols. CONVEX systems can be connected to Cray Research, Inc. supercomputers using HYPERchannel™.

Corporate Headquarters
CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
Phone: 214/497-4000

For more information or for the location of your local CONVEX sales office, contact:

U.S.A.

Eastern Area
7501 Greenway Center Drive
Suite 800
Greenbelt, MD 20770
Phone: 301/345-2400, 2401

Western Area
1731 Technology Drive
Suite 800
San Jose, CA 95110
Phone: 408/453-5700

Canada

60 Columbia Way
Suite 300, #10
Markham, Ontario
L3R 0C9
Phone: 416/940-3644

Europe

United Kingdom
Randalls Research Park
Randalls Way
Leatherhead
Surrey KT22 7TS
Phone: 44-372-386696

France
Parc d'Activites Pas du Lac
Immeuble le Daguerre
9, Avenue Ampere
78180 Montigny-le-Bretonneux
Phone: 33-1-30589300

Italy
Viale Colleoni 17
Palazzo Orione Ingr. 3
20041 Agrate Brianza
Phone: 39-39-638611

Germany
Lyoner Strasse 14
D-6000 Frankfurt/Main 71
Phone: 49-69-6668081

The Netherlands
Europalaan 514
3526 KS Utrecht
Phone: 31-30-888368

Denmark
Baltorpvej 156
DK-2750 Ballerup
Phone: 45-44-683000

Far East

Headquarters
1 Scotts Road
#25-06 Shaw Centre
Singapore 0922
Phone: 65-733-4355

Japan
Ichiban-cho Central Bldg.
22-1, Ichiban-cho
Chiyoda-Ku, Tokyo 102
Phone: 81-3-221-5105

Australia

80 Albert Street
8th Floor
Brisbane, Queensland 4000
Phone: 61-7-221-2404