# CONVEX C Optimization Guide

*Third Edition*


CONVEX COMPUTER CORPORATION

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

# CONVEX C Optimization Guide

Order No. DSW-089

Third Edition
January 1993

CONVEX Press
Richardson, Texas, USA

# CONVEX C Optimization Guide

Order No. DSW-089



Printed in the United States of America

# Revision Information for

# CONVEX C Optimization Guide

| Edition | Document No. | Description |
|---|---|---|
| Third | 720-001130-203 | Released with CONVEX C software V5.0, January, 1993. New information was added on pointer tracking and restricted pointers. A new optimization-report appendix was added. Other sections were reorganized and rewritten. |
| Second | 720-001130-202 | Released with CONVEX C software V4.1, April, 1991. |
| First | 720-001130-200 | Released with CONVEX C software V4.0, May, 1990. |

# Contents

# How to use this guide

This guide describes methods for optimizing C 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 approaches for controlling and enhancing 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 C Optimization Guide* is for experienced C programmers. Readers need not be familiar with the CONVEX implementation of scalar, vector, and parallel optimization. Although intended primarily for users of CONVEX C, some of the methods described in this book may apply to other C 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 C compiler works.
- 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 C 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 approaches for optimizing your programs to run on CONVEX C Series supercomputers.
- Chapter 8 presents the special optimization problems caused by aliasing, which usually arise from the use of pointers.
- 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 discusses CONVEX C intrinsic functions, their effects on optimization, and their impact on the errno variable.
- Appendix C explains how to use CONVEX C optimization pragmas.
- Appendix D describes vector operations on the assembly-language level and presents examples of some assembly-language instructions.
- Appendix E describes the optimization report.
- The Bibliography contains a bibliography on topics related to optimization.

- The Glossary defines a terms used throughout the document.

## Notational Conventions

This section discusses notational conventions used in this book.

### Command syntax

Consider this example:

```
COMMAND input_file [...] {a | b} [output_file]
  ①       ②       ③     ④       ⑤
```

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

### General conventions

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

- Italics:
  - Designate user-supplied variables in a command-line example
  - Introduce new and important terms
  - Identify variables in mathematical equations
  - Indicate document titles

- `Constant-width font` designates:
  - System output in screens and examples
  - Command names and options
  - System calls
  - Data structures and types
  - Directives, program statements, display examples, printout examples, file names, and error messages returned

- **`Bold, constant-width font`** designates user

input in screens and examples, and must be typed exactly as it appears.

- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipsis shows that lines of code have been left out of an example.
- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, RETURN refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, CTRL-X indicates that you must press and hold down the CTRL key and then press the X key.
- The word "enter" in a phrase such as "enter `ls`" means that you type the command and then press RETURN.
- References to the *ConvexOS Man Pages* appear in the form adb(1), where the name of the man page is followed by its section number enclosed in parentheses.

**Note**

**A note highlights supplemental information.**

**Caution**

**A caution highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.**

## 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 1(800)952-0379.
- From locations in Canada, use 1(800)345-2384.
- From all other locations, contact your local CONVEX office.

# The basics

1

Optimization improves the performance of programs. To optimize programs, the CONVEX C 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 C compiler offers five optimization options, which are specified on the `cc` command line. The compiler transforms code according to the optimization option you specify. At each higher level, the compiler performs all the optimizations performed at lower levels, plus some new ones.

| Option | Description |
|---|---|
| `-no` | Machine-dependent scalar optimization. This option is the default. |
| `-O0` | Basic-block machine-independent scalar optimization |
| `-O1` | Basic-block and function-level machine-independent scalar optimization |
| `-O2` | Vector optimization (not available with Scalar C compiler) |
| `-O3` | Parallel optimization (not available with Scalar C compiler) |

# Scalar optimization

A scalar value is a single value or entity. A scalar instruction operates on one or more 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 (function) 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 optimization level `-O0`, the compiler does machine-independent optimizations within the scope of a basic block.

At `-O1`, the compiler performs machine-independent optimizations across multiple basic blocks in a function; the optimization is local to the function but global with respect to basic blocks.

To improve performance, machine-independent optimizations:

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

## Vector optimization

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

The compiler also transforms many loops that it cannot vectorize into loops that it can vectorize. This increases the number of loops that the compiler can optimize, which can lessen 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 performance improvement you can achieve with parallel optimization depends 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.

## Loop parallelism

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 the thread of code associated with that task. At this point, two or more CPUs are working on different threads of the same job.

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

## Task parallelism

To parallelize C programs containing groups of independent tasks, you can use the C tasking pragmas. For more information about tasking pragmas, refer to Chapter 4, "Parallel optimization," and Appendix C, "Compiler pragmas."

## Programming tools

The CONVEX visual debugger, CXdb, is a symbolic debugger with a window interface that makes interacting with the debugger easier than interacting with command-line-only debuggers. In addition to traditional debugger features, CXdb has special functions, such as source-unit stepping, that provide you with extra control. For more information on using CXdb, refer to the *CONVEX CXdb User's Guide.*

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

The CONVEX Performance Analyzer, CXpa, is a tool for examining your program's performance at routine, loop, and basic-block level. You can use CXpa or one of the profilers in the CONVEX Consultant to track the effects of optimizations. For more information on how to use CXpa, 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. Scalar instructions operate on one or a pair of scalar values, as in the C statement:

```
scalar1 += scalar2;
```

The CONVEX C 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. These optimizations cannot be disabled. At optimization level `-O0`, the compiler performs machine-dependent and machine-independent optimizations. The compiler optimizes one basic block (a linear sequence of statements with a single entry and a single exit) at a time at this level. At level `-O1`, the compiler optimizes across all the basic blocks within one function.

## 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. 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 CXpa) typically decreases as the optimization level increases.**

# 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. Operations such as add, multiply, and store/load execute simultaneously on separate functional units.

At optimization level `-no`, the compiler rearranges instructions derived from a single C source statement to maximize use of the functional units. Compare the two assembly-language codes for the C source statement shown here:

**C source:** `a = (b + c * d) / e - f;`

**Original code**

```
ld.w  d, s0
ld.w  c, s1
mul.s s1,s0
ld.w  b, s1
add.s s1,s0
ld.w  e, s1
div.s s1,s0
ld.w  f, s1
sub.s s1,s0
st.w  s0, 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
div.s s0,s2
sub.s s3,s2
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 can also execute simultaneously.

Concurrent execution of machine instructions on multiple functional units, which occurs on a single CPU, is distinct from parallel processing, which occurs on multiple CPUs. For more information on functional units, refer to the *CONVEX Architecture Reference Manual (C Series).*

## Span-dependent instructions

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

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

## Register allocation

CONVEX C uses a technique for allocating registers that fully exploits the CONVEX register set. This technique 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:

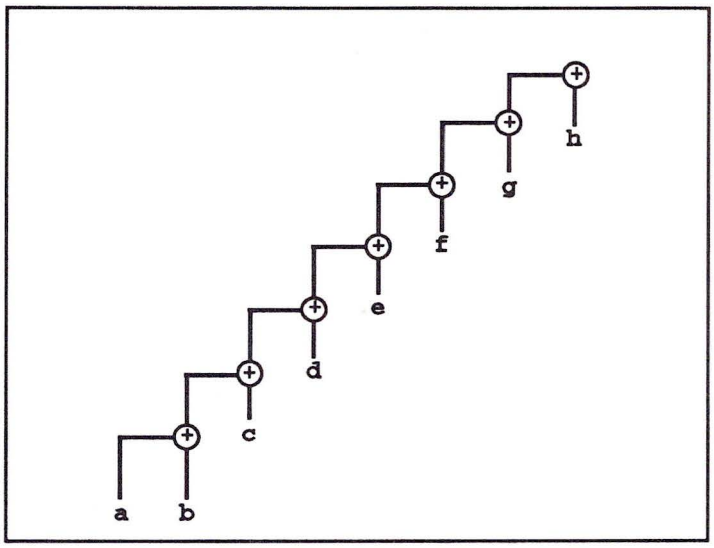
```
a + b + c + d + e + f + g + h
```

The expression can be evaluated, as specified in the ANSI C standard, as follows:

```
((((((a + b) + c) + d) + e) + f) + g) + h)
```

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

**Figure 1**
Unbalanced tree representation

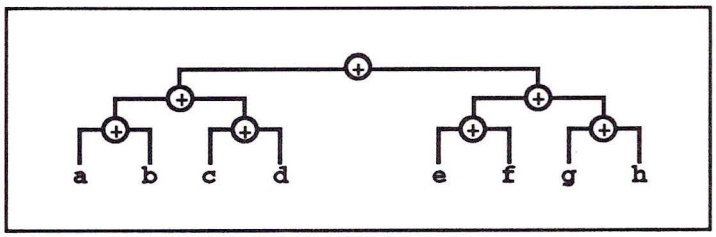



Another way to evaluate the integer expression is:

```
(((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.

**Figure 2**
Balanced tree representation




In Figure 1, the depth of the tree is seven; in Figure 2, the depth of the tree is three. The machine instruction sequence generated for the tree in Figure 2 executes faster than the instruction sequence generated for the tree in Figure 1.

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. Because the compiler chooses evaluation order to ensure the most efficient execution, you can write expressions in any 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 C statements:

**C 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. Other optimizations are also performed.

## 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:

**Original code**

```
x = y + c;
/* x is not used */
x = 3.1416;
...
y = (x + 7) * 2.15;
```

**Optimized code**

```
/*(statement removed)*/
...
x = 3.1416;
...
y = (x + 7) * 2.15;
```

## Assignment substitution

Assignment substitution eliminates redundant loads. The compiler retains the value assigned to a variable and replaces subsequent references to that variable with the assigned value. Here is an example:

**Original code**

```
x = y + c;
x = x * 4.4;
t = x * b + 12.4;
x = 4.179;
```

**Optimized code**

```
REG = y + c;
REG = REG * 4.4;
t   = REG * b + 12.4;
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.

## 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 `x`. This is known as *constant propagation*, which is a form of assignment substitution.

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. Here is an example of constant propagation and folding:

| Original code | Optimized code |
|---|---|
| `i = 5;` | `i = 5;` |
| `j = 0;` | `/*(assignment removed)*/` |
| `...` | `...` |
| `j = j + 2;` | `j = 2;` |
| `...` | `...` |
| `k = k + i * j;` | `k = k + 10;` |

The compiler folds many ANSI C library functions when they are applied to constant arguments. For example, `sin(0.0)` becomes `0.0`. The compiler uses intrinsic functions to compute these values at compile time. For more information on intrinsic functions, refer to Appendix B, "CONVEX C intrinsics."

The compiler recognizes type-converted constants before propagating and folding them. If a program contains the expression `x=1`, where `x` is `double`, the compiler converts `1` to `1.0` before propagating it.

The C compiler reports integer overflow at compile time if the `-d integer_overflow=e` option is specified on the `cc` command line. 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 operation causing the error or bring the value of the constant within acceptable bounds as described in `limits.h` and `float.h`.

## 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 to eliminate redundant register loads.

## Algebraic and trigonometric simplification

The compiler simplifies algebraic and trigonometric expressions, as shown here:

| Original expression | Optimized expression |
|---|---|
| `x + 0` | `x` |
| `x * 1` | `x` |
| `x * 0` | `0` |
| `k & -1` | `k` |
| `k & 0` | `0` |
| `k \| -1` | `-1` |
| `k \| 0` | `k` |
| `-1 * x` | `-x` |
| `x - x` | `0` |
| `x / -1` | `-x` |
| `x / x` | `1` |
| `0 - x` | `-x` |
| `0 / x` | `0` |

The compiler performs obvious variations of these operations for commutative operators. For example, it converts `x+(0+y)` to `x+y`.

## Optimizations performed at -O1

Global optimization is done across all basic blocks but within a single function. 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.

An example of constant propagation and folding appears below:

**Original code**

```
main()
{
   int a,b,c,i;
   a = 5;
   b = 15;
   scanf("%d", &i);
   if (i<=0) {
      a = 6;
      c = a;
      b = a + c;
   } else {
      c = a + b;
      b = a + b + c;
   }
   printf("%d,%d,%d",
        a, b, c );
}
```

**Optimized code**

```
main()
{
  int a,b,c,i;
  a = 5;
  b = 15;
  scanf("%d", &i);
  if (i<=0) {
     a = 6;
     c = 6;
     b = 12;
  } else {
     c = 20;
     b = 40;
  }
  printf("%d,%d,%d",
     a, b, c );
}
```

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 local variables that do not have subsequent references within the function. The following example shows how the compiler eliminates redundant assignments to the variables `a` and `x`:

**Original code**

```
float x;
void foo( float y, float z)
{
   float a;
   ...
   x = y * z;
   if (a > 0) {
   ...
      a = x * y + 3.1416;
   } else {
      ...
      x = (x + 7) * z + 3.1416;
   }
   ...
   /* a is not used later in this routine */
}
```

The local variable `a` is assigned but never used, so the compiler can eliminate the assignment as shown here:

**Optimized code**

```
float x;
void foo( float y, float z)
{
   ...
   x = y * z;
   if (a > 0) {
      ...
   } else {
      ...
      x = (x + 7) * z + 3.1416;
   }
   ...
   /* a is not used later in this routine */
}
```

If the right side of a redundant assignment statement contains a function call, the compiler eliminates the assignment and retains the function call. Here is an example:

**Original code**

```
int foo()
{
   ...
   i = intfun(x);
   ...
}
/* i is not used */
```

**Optimized code**

```
int foo()
{
   ...
   intfun(x);
   ...
}
```

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 a global variable, read or write, or call another function have no side effects. The compiler cannot automatically determine whether a side effect exists. It can eliminate function calls only if you explicitly request it with the `no_side_effects` pragma.

The form of this pragma is

```
#pragma _CNX no_side_effects (func_list)
```

where *func_list* is a list of function names separated by commas. The pragma must precede the function call that does not contain side effects.

**Caution**

**Do not use the `no_side_effects` pragma on a call to a function that:**

- **Changes the value of an object pointed to by one of its pointer arguments**
- **Changes the value of a global variable**
- **Calls another function that performs one of these operations (including I/O functions)**
- **Changes the value of a "static" local variable**

Because all scalar and structure arguments are passed by value, changing the value of an argument is not a side effect. However, changing the value of an object, such as an array, pointed to by a pointer argument is a side effect.

For more information about the `no_side_effects` pragma, refer to Appendix C, "Compiler pragmas."

## 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 a constant, then the compiler eliminates the unreachable alternative of the `if` statement.

## Hoisting and sinking scalar and array references

The compiler can *hoist* some scalar and array references from 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 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` and above, 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` and above, to a section of an array that can be kept in a vector register during the loop's execution

Sinking occurs without hoisting in the following case:

- At optimization level `-O1`, to a scalar variable that is only assigned values during the loop's execution.

For more information, refer to "Paired hoist and sink" in Chapter 3.

## 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, or vice versa.

In the following example, if the compiler determines that x and y are unchanged between the first and second statements, it replaces x with y in the third statement.

```
x = y;                  /* statement 1 */
...
w = z - x;              /* statement 2 */
```

becomes

```
...
w = z - y;              /* statement 3 */
```

## 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, it 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 here shows an example of a common subexpression:

**Original code**

```
void foo()
{
   ...
   a = b + c / (-j * b + (c * c * c));
   if (k < 1)
      l = 5;
   f = e - c / (-j * b + (c * c * c));
   ...
}
```

The compiler recognizes that a common subexpression is used before and after the `if` statement. It 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 here:

**Optimized code**

```
void foo()
{
   ...
   t1 = c / (-j * b + (c * c * c));
   a = b + t1;
   if (k < 1)
      l = 5;
   f = e - t1;
   ...
}
```

## Code motion

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

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

**Original code**

```
#include <math.h>
extern double a,b,c,e;
void gcm()
{
   double ar[10];
   int i;
   ...
   for( i=0; i<10; i++ ){
      a = c / (-(e * b) + (c * c * c));
      ar[i] = a + b * c;
   }
   ...
}
```

At higher optimization levels, the compiler can vectorize the loop, as shown here:

**Optimized code**

```
#include <math.h>
extern double a,b,c,e;
void gcm()
{
   double ar[10];
   int i;
   ...
   a = c / (-(e * b) + sqrt(c));
   t1 = a + b * c;
   for( i=0; i<10; i++)
      ar[i] = t1;
   ...
}
```

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

## Strength reduction

In some cases, the compiler can reduce the strength of arithmetic operations and operations involving induction variables and constants. Such reductions make the operations execute faster.

### Arithmetic operations

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

```
j * 2 becomes j << 1
j * 4 becomes j << 2
```

Integer divisions are not directly replaced with integer shifts because the CONVEX architecture has a logical-shift instruction, but no arithmetic-shift instruction. (Logical shifts do not sign-extend.)

```
a / 2 becomes

  if (a<=0)
     a++;
  discard(convert(a) >> 1)
```

The `convert` operation handles the sign extension by converting `a` to a larger data type; `discard` gets rid of the extra bits.

Multiplication involving real constants and small integers is reduced to addition:

```
x * 2 becomes x + x
```

Under the `-uo` option only, division by a constant is reduced to multiplication:

```
x / c becomes x * d where d = 1 / c
```

Because `c` is a constant, `d` also is a constant, which can be computed at compile time.

Refer to Appendix A for more information about the `-uo` option.

### Induction variables and constants

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

In the following example, the compiler recognizes that `i` is incremented by 2 on each iteration and that the value assigned to `x` is larger by `2*c` on successive iterations.

**Original code**

```
void gsr()
{
   int i = 1;          /* induction var */
   ...
   do {
      x = i * c;     /*  loop induction value */
      ...
      i += 2;
   } while( i<=100 );
}
```

The compiler produces code that calculates `c+c` only once and increments `x` by the value saved in `t2` instead of calculating `i*c` on every iteration. This only occurs when `c` is a loop constant:

**Optimized code**

```
void gsr()
{
    int i = 1;
    ...
    t1 = c;
    t2 = c + c;
    do {
        x = t1;
        ...
        t1 += t2;
        i += 2;
    } while( i <= 100 );
}
```

# Vector optimization

# 3

Vector instructions are the key to high performance on the CONVEX C Series supercomputers. Vectorization converts scalar loops operating on array elements into equivalent vector instructions. These instructions use vector registers capable of containing up to 128 array elements. The compiler partitions arrays longer than 128 elements into sections of no more than 128 elements. This partitioning 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.

```
for( i=0; i<100; i++ )
  a[i] = b[i] + c[i];
```

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 an array section involving only four instructions,

```
a[0:99] = b[0:99] + c[0:99]
```

where `a[i:k]` means `a[i]` through `a[k]`.

## Automatic transformations

The compiler transforms the statements of a program to improve vectorization. The following subsections explain the most important transformations.

### Strip mining

A vector register can hold 128 elements. When the compiler cannot determine the iteration count of a loop, or the iteration count exceeds 128, the compiler strip-mines the loop before vectorizing it. Strip mining creates an inner loop, with an iteration count that never exceeds 128, and an outer loop. Consider the following example:

**Original Loop**

```
for( i=0; i<n; i++ )
    a[i] = b[i] + c[i];
```

The compiler vectorizes this loop as shown in Figure 3-2, using `iout` as a variable to count the number of elements remaining to be processed and `i` as the starting index for each vector operation.

**Stripmined Loop**

```
i = 0;
for( iout=n-1; iout>=0; iout-=128 ){
    k = i + min(127, iout);
    a[i:k] = b[i:k] + c[i:k]; /* vector code */
    i += 128;
}
```

For `n=300`, this code tests `iout` four times. For each test, `i` and `iout` have the values shown below. If the test succeeds, the vector code processes the elements of `a` shown in the third column.

| i | iout | Elements Processed |
|---|---|---|
| 0 | 299 | 0...127 |
| 128 | 171 | 128...255 |
| 256 | 43 | 256...299 |
| 384 | -85 | |

The fourth test of `iout` fails, so the vector loop does not execute again and no additional 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 in which all calculations are not performed within the innermost loop, however, can be vectorized by distributing the outer loop and vectorizing each of the resulting loops or loop nests. Consider the loop nest shown here:

**Original Loop**

```
for( i=0; i<n; i++ ){
    b[i][0] = 0;
    for( j=0; j<m; j++ )
        a[i] += b[i][j] * c[i][j];
    d[i] = e[i] + a[i];
}
```

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 vectorizable loops, as shown here:

**Vectorized Loop**

```
for( i=0; i<n; i++ )
    b[i][0] = 0;

for( i=0; i<n; i++ )
    for( j=0; j<m; j++ )
        a[i] += b[i][j] * c[i][j];

for( i=0; i<n; i++ )
    d[i] = e[i] + a[i];
```

## Loop interchange

The compiler interchanges nested loops for the following reasons:

- To make the most profitable parallelizable loop the outermost loop
- To make memory accesses to consecutive memory locations
- 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 here:

**Original Loop**

```
for( i=0; i<n; i++ )
    for( j=0; j<m; j++ )
        a[j][i] = b[j][i] + c[j][i];
```

To vectorize the original loop, the compiler interchanges the `i` and `j` loops so that contiguous elements of `b` and `c` are loaded into vector registers. This optimization, shown in the example below, substantially improves performance over the column-by-column approach of the source code.

**Vectorized Loop**

```
for( j=0; j<m; j++ )
    for( i=0; i<n; i++ )
        a[j][i] = b[j][i] + c[j][i];
```

## Paired hoist and sink

Often, the compiler can use a vector register as an accumulator, making it possible to remove register loads and stores from the vector loop. As Chapter 2 states, moving an operation (such as a register load) from of a loop to a basic block preceding the loop is called hoisting.

Sinking is the complement of hoisting: the compiler moves an operation (such as a register store) from a loop to a basic block following the loop. Consider this loop nest:

```
for( i=0; i<n; i++ )
    for( j=0; j<n; j++ )
        a[i] += b[i][j];
```

When you compile this code at `-O2` or `-O3`, the compiler interchanges and vectorizes the innermost loop. It also hoists with respect to the vectorized loop the vector load of a above the loop and sinks the vector store of a below the loop. This eliminates the need for repeated loads and stores.

The following example shows how the compiler hoists and sinks the load and store of register `V0`:

```
i = 0;
for( iout=n-1; iout>=0; iout-=128 )
{
    k = i + min(127, iout);
    V0 = a[i:k]; /* vector load */
    for( j=0; j<n; j++ )
        V0 += b[i:k][j];
    a[i:k] = V0; /* vector store */
    i += 128;
}
```

The compiler can hoist vector loads and sink vector stores only if the arrays on both sides of the equals sign have the same subscript, and the subscript is a *loop induction variable* of the vectorized loop or a loop constant.

The compiler can interchange loops to make a subscript a loop constant so that sinking and hoisting are possible.

## Optimizations of `for-if` loops

To vectorize a loop, the compiler needs a loop induction variable, which is incremented by a constant value on each iteration, or a *conditional* induction variable, which is incremented depending on some condition.

The following loop uses a conditional induction variable, `k`:

```
for( i=0, k=0; i<100; i++ )
    if( e[i] )
        a[i] = b[k++];
```

For this loop, the compiler generates code that does the following:

1. Determines the values of `i` for which `e[i]` is true and stores the results as a vector. (Call it `truth`.)
2. Counts the number of those values. (Call it `count`.)
3. Loads `count` elements of `b` into a vector register.
4. Expands the vector strip of `b` to the indices indicated by the `truth` vector.
5. Stores the expanded vector into `a[0:99]`.

This results in an efficient loop despite the embedded `if` statement and conditional use of induction variable `k`.

Frequently, however, the compiler can transform `for-if` constructs to eliminate much of the overhead caused by such conditional tests. It does this by promoting tests, eliminating tests, and peeling loop-boundary values.

### Test promotion

Often, the compiler can "promote" a test out of a loop. The compiler promotes a test by hoisting it out of the loop in much the same way that it hoists a vector load.

Consider, for example, the following loop:

```
for (i=0; i<n; i++)
    if ( foo=bar )
        a[i] += b[i];
```

The test in this loop involves two variables, `foo` and `bar`, that do not change value within the loop. Because `foo` and `bar` are loop constants, the outcome of the test does not change from one iteration of the loop to the next. The compiler recognizes that this test only needs to be performed once and moves it out of the loop:

```
if (foo=bar)
    for (i=0; i<N; i++)
        a[i] += b[i];
```

Although the compiler can vectorize a test within a loop, moving the test out of the vector loop makes the loop run faster. Doing the test first also eliminates unnecessary executions of the loop. If the test fails, execution skips to the next instruction following the loop.

This optimization permits the test to be performed once instead of `N` times. Although the compiler can vectorize a test within a loop, the resulting vector loop runs slower than than the same vector loop without the test.

The following example shows a slightly more complicated test construct, which incorporates an `else` statement:

```
for(i=0; i<n; i++)
   if (foo==bar)
     a[i]=b[i]
   else
     a[i]=0;
```

The compiler promotes this test, creating two loops:

```
if (foo==bar)
   for(i=0; i<n; i++)
       a[i]=b[i]
 else
   for (i=0; i<n; i++)
     a[i]=0;
```

Because it replicates loops, test promotion can increases the size of your code.To prevent code size from growing exponentially, the compiler has a built-in limit on the number of test promotions performed. To increase this limit, use the `-ptst` option on the `cc` command line. To increase the limit to its maximum, use `-ptstall`. (Increasing the amount of test promotion can also increase compile time.)

You can increase the limit for a given loop by using the `promote_test` or `promote_test_all` pragma on that loop.

To turn test promotion off altogether, use the `-noptst` option. To turn test promotion off for a single loop, use the `no_promote_test` pragma on that loop.

### Test elimination

The CONVEX C compiler can eliminate many unnecessary `IF` tests from vectorizable loops, allowing it to produce more efficient vector code.

Constant propagation allows the compiler to perform many tests, eliminating the need for it to generate code that performs the test at run time. For example, the test in the following code always succeeds:

```
foo=0;
bar=0;
for(i=0; i<n; i++)
   if (foo=bar)
     a[i] += b[i];
```

The compiler recognizes this, through constant propagation, and eliminates the test:

```
for (i=0; i<N; i++)
    a[i] += b[i];
```

This test always fails:

```
foo=0;
bar=1;
for(i=0; i<n; i++)
   if (foo=bar)
        a[i] += b[i];
```

Since the test fails, the compiler recognizes that the loop is dead code and eliminates it altogether. Not executing a loop is always faster than executing one, so this can result in a significant speedup.

The construction of a loop control can make some tests redundant. Consider, for example, the following loop:

```
for (i=0; i<n; i++)
   if (i>-1)
        a[i] += b[i];
```

The `if` test is redundant because of the loop control, which initializes `i` to 0 and increments it one each iteration.The compiler recognizes that the condition can never occur and removes the test:

```
for (i=0; i<n; i++)
    a[i] += b[i];
```

### Boundary-value peeling

Many loops handle boundary conditions as a special case using `if` tests. Consider this example:

```
for (i=0; i<=n; i++)
   if (i==0)
     a[i]=b[i]
   else
     if(i==n)
       a[i]=c[i]
     else
       a[i]= -a[i];
```

This code tests for the first and last element of the array and handles those s special cases. The compiler recognizes this construct and "peels" away the boundary-value tests:

```
a[1]= b[1];
for (i=1; i<n; i++)
  a[i]=-a[i];
a[n]=c[n];
```

The compiler can peel tests that occur on the first iteration of the loop, the last iteration of the loop, or both. It cannot peel tests that occur between the first and last iterations of the loop. For example, it cannot peel either of the two tests in the following loop:

```
for (i=0; i<=n; i++)
  if (i==10)
    a[i]=b[i]
  else
    if (i==20)
      a[i]=c[i]
    else
      a[i]= -a[i];
```

Boundary-value peeling, like test promotion, can increase the size of your object code.To prevent code size from growing exponentially, the compiler has a built-in limit on the number of tests peeled. To increase this limit, use the `-peel` option on the `cc` command line. To increase the limit to its maximum, use `-peelall`. (Increasing the amount of peeling can also increase compile time.)

You can increase the limit for a given loop by using the `peel` or `peel_all` pragma on that loop.

To turn peeling off altogether, use the `-nopeel` option. To turn peeling off for a single loop, use the `no_peel` pragma on that loop.

## Inhibitors of vectorization

Any of these conditions inhibit or prevent vectorization:

- `switch` statements
- Multiple loop entries or exits
- Function calls
- Unsigned induction variables
- Recurrences
- Aliased scalar or array variables or pointers

The following sections discuss the problems of unsigned induction variables and recurrence. For a discussion of aliasing problems, see Chapter 8.

### Unsigned induction variables

The compiler cannot vectorize loops that have an unsigned induction variable. For example, the following code cannot vectorize because it has an unsigned induction variable:

```
unsigned char i;
for( i=0; i<10; i++ )
    a[i] = 0.0;
```

When this code is compiled at level `-O2` and above, the optimization report states that the loop does not have an induction variable. This situation can be corrected by using signed induction variables only, as shown here:

```
char i;
for( i=0; i<10; i++ )
    a[i] = 0.0;
```

## 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 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 function-call recursion.)

Recurrence is closely related to *data dependency.* A data dendency 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. An *apparent recurrence* exists when the compiler lacks sufficient information to prove that an actual recurrence does not exist. 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 following loop contains an LCD:

```
for( i=0; i<n; i++ ){
    a[i+1] = a[i] + 3.14;
}
```

The loop carries the dependency from one iteration to the next.

Loop-carried dependencies can be backward or forward. A backward LCD exists when one iteration references a variable whose value was assigned on a previous iteration. Figure 3-13 shows a backward LCD. The first iteration of the loop assigns a value to `a[1]`, the second iteration references that value and assigns a new value to `a[2]`, 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 following loop contains a forward dependency:

```
for( i=0; i<n-1; i++ ){
    a[i] = a[i+1] + 3.14;
}
```

In this example, the first iteration assigns a value to `a[0]` and references `a[1]`; the second iteration assigns a value to `a[1]` and references `a[2]`. The reference to `a[i]` depends on the fact that the `i+1`th iteration, which assigns a new value to `a[i]`, has not yet executed. A forward dependency, therefore, does not prevent vectorization of a loop.

The compiler can vectorize some loops containing backward LCDs. The following loop in contains an LCD that points backward from `b[i+1]` to `b[i]`:

```
for( i=0; i<n-1; i++ ){
    a[i] = b[i] + c[i];
    b[i+1] = d[i] * 3.14;
}
```

In this loop, the assignment to `a[1]` on the second iteration depends on the value assigned to `b[1]` 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 here:

```
for( i=0; i<n-1; i++ ){
    b[i+1] = d[i] * 3.14;
    a[i] = b[i] + c[i];
}
```

When a scalar variable causes an LCD, the compiler may eliminate the recurrence with a transformation called *scalar expansion*. 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, with an LCD on the variable `x`, appears here:

**Original Loop**

```
for(i=0;i<10;i++){
    x = a[i];
    a[i] = b[i]
    b[i] = x;
   }
```

**Vectorized Loop**

```
 for(i=1;i<10;i++){
    temp[i]=a[i];
    a[i] = b[i];
    b[i] = tmp[i];
   }
x = temp[9];
```

In this example, the temporary vector `temp` replaces all references to scalar `x` in the loop. When the loop ends, the value of `temp[9]` is assigned to `x`.

A backward LCD that cannot be eliminated does not always stop vectorization completely. Using temporary vectors, the compiler can sometimes vectorize part of a loop that contains an LCD. Here is an example:

**Original Loop**

```
for( i=1; i<n; i++ ){
   a[i] = a[i-1] + b[i] * c[i];
}
```

In the original loop, the assignment to `a[i]` depends on the value of `a[i-1]`, which is computed on the previous iteration. The vectorized loop, shown below, isolates the dependency by distributing the loop and vectorizes the first distributed part. The second distributed part is executed with scalar instructions. This transformation is called *partial vectorization* because it distributes a loop into vector and scalar parts.

**Vectorized Loop**

```
for( i=1; i<n; i++ )
   t[i] = b[i] * c[i];
for( i=1; i<n; i++ )          /* scalar */
   a[i] = a[i-1] + t[i];
```

### Loop-independent dependency

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

```
for( i=0; i<n; i++ ){
   a[i] = b[i] + d[i]; /* statement 1 */
   b[i] = 0.0;         /* statement 2 */
   d[i] = d[i] + 1.0;  /* statement 3 */
}
```

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 2 and 3 are dependent on Statement 1. A forward LID exists between statements 1 and 2; another exists between statements 1 and 3.

**Note**

**LIDs do not normally prevent vectorization. LCDs, which cause recurrences, can prevent vectorization. Vectorization is prevented when a backward LCD exists which cannot be converted into a forward LCD by statement reordering.**

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

```
for( i=0; i<n-1; i++ ){
   a[i] = b[i] - c[i];    /* statement 1 */
   b[i+1] = a[i] + d[i]; /* statement 2 */
}
```

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 prevent this interchange.

### Apparent recurrences

When it appears that a recurrence may exist, but the compiler lacks sufficient information to be sure, an apparent recurrence exists. Most apparent recurrences result from loop constants whose signs are unknown to the compiler or array subscripts that contain array references.

The following example shows a loop that cannot be vectorized because the sign of k is unknown:

```
for( i=m; i<=n; i++ ){
   a[i+k] = 2.0;
   a[i] = 0.0;
}
```

If k is less than zero, a backward dependency exists. To vectorize the loop, the compiler must convert this to a forward dependency by interchanging the two assignment statements. But if k is greater than or equal to zero, a forward dependency exists. Interchanging the two statements would convert this into a backward dependency. The compiler knows the conditions are contradictory and neither operation can be performed.

The following loop cannot be vectorized because the compiler cannot determine whether a recurrence exists:

```
for( i=0; i<n; i++ ){
   a[j[i]] = a[k[i]] + 1;
}
```

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 array a form a recurrence.

## Reductions

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 `+`, `-`, `*`, `&`, `|`, `==`, or `!=`.

The loop shown below in computes the sum and the product of the elements of `a[0:99]`. The compiler vectorizes both reductions. If there is a data type conversion in the reduction, it will inhibit vectorization.

**Original Loop**

```
sum  = 0.0;
prod = 1.0;
for( i=0;i<100;i++ ){
   sum  += a[i];
   prod *= a[i];
}
```

**Vectorized Loop**

```
sum = vsum(a[0:99]);
prod = vprod(a[0:99]);
```

In the optimized code, `vsum()` and `vprod()` are single vector machine instructions that return the sum and the product, respectively, of up to 128 elements.

# Parallel optimization

# 4

At optimization level -O3, the CONVEX C 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 executes on a single CPU.

The CONVEX C compiler achieves 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 by adding synchronization code.

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

```
for( i=0; i<n; i++ ){
  for( j=0; j<n; j++ ){
    c[i][j] = 0.0;
    for( k=0; k<n; k++ )
      c[i][j] += a[i][k] * b[k][j];
  }
}
```

The compiler processes this loop nest by distributing the loop nest containing the `i` and `j` loops, as shown here:

```
for( i=0; i<n; i++ ){
  for( j=0; j<n; j++ )
    c[i][j] = 0.0;
}

for( i=0; i<n; i++ ){
  for( j=0; j<n; j++ ){
    for( k=0; k<n; k++ )
      c[i][j] += a[i][k] * b[k][j]
  }
}
```

The following code shows how the compiler moves the `j` loop to the innermost position in each nest so that it can retrieve contiguous elements on successive iterations.

```
for( i=0; i<n; i++ ){
  for( j=0; j<n; j++ )
    c[i][j] = 0.0;
}

for( i=0; i<n; i++ ){
  for( k=0; k<n; k++ ){
    for( j=0; j<n; j++ )
      c[i][j] += a[i][k] * b[k][j];
  }
}
```

The compiler strip mines both `j` loops to the optimal vector length, a function of the loop upper bound (`n`). In the following examples, `mvsl` represents that function, and `v0` and `v1` represent vector registers that can contain up to 128 64-bit elements.

```
m = mvsl(n);
for( i=0; i<n; i++ )
  for( jouter=0; jouter<n; jouter+=m )
    c[i][jouter:min(n-1,jouter+m-1)] = 0.0;

for( i=0; i<n; i++ )
  for( k=0; k<n; i++ )
    for(jouter=0; jouter<n; jouter+=m ){
      min_jouter = min(n-1, jouter+m-1);
      v0 = c[i][jouter:min_jouter];
      v1 = b[k][jouter:min_jouter];
      v0 += v1 * a[i][k];
      c[i][jouter:min_jouter] = v0;
    }
```

In the second nest, the compiler interchanges the `jouter` strip-mine loop outside of the k loop, as shown here:

```
m = mvsl(n);
for( i=0; i<n; i++ )
  for( jouter=0; jouter<n; jouter+=m )
    c[i][jouter:min(n-1, jouter+m-1)] = 0.0;

for( i=0; i<n; i++ )
  for( jouter=0; jouter<n; jouter+=m )
    for( k=0; k<n; k++ ){
      min_jouter = min(n-1, jouter+m-1);
      v0 = c[i][jouter:min_jouter];
      v1 = b[k][jouter:min_jouter];
      v0 += v1 * a[i][k];
      c[i][jouter:min_jouter] = v0;
    }
```

In the following code, PARALLEL FOR represents a loop that multiple CPUs can process.

```
m = mvsl(n);
PARALLEL FOR( i=0; i<n; i++ )
  for( jouter=0; jouter<n; jouter+=m )
    c[i][jouter:min(n-1, jouter+m-1 )] = 0.0;

PARALLEL FOR( i=0; i<n; i++ )
  for( jouter=0; jouter<n; jouter +=m )
    for( k=0; k<n; k++ ){
      min_jouter = min(n-1, jouter+m-1);
      v0 = c[i][jouter:min_jouter];
      v1 = b[k][jouter:min_jouter];
      v0 += v1 * a[i][k];
      c[i][jouter:min_jouter] = v0;
    }
```

The compiler hoists a vector load and sinks a vector store out of the k loop. The remaining reference to vector v1 chains with the vector addition and vector multiplication in the next statement, resulting in even faster execution.

```
m = mvsl(n);
PARALLEL FOR( i=0; i<n; i++ )
  for( jouter=0; jouter<n; jouter+=m )
    c[i][jouter:min(n-1, jouter+m-1)] = 0.0;

PARALLEL FOR( i=0; i<n; i++ )
  for( jouter=0; jouter<n; jouter+=m ){
    min_jouter = min(n-1, jouter+m-1 );
    v0 = c[i][jouter:min_jouter];
    for( k=0; k<n; k++ ){
      v1 = b[k][jouter:min_jouter];
      v0 += v1 * a[i][k];
    }
    c[i][jouter:min_jouter] = v0;
  }
```

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 in loops
- Function calls in loops
- Aliased scalar or array variables or pointers
- Nondeterminism of parallel execution
- Loop-carried dependencies

### Loops with function calls

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

If you use `force_parallel`, you must recompile the called function (or any routines called indirectly) for re-entrancy with the `-re` option. For more information about compiler pragmas, refer to Appendix C, "Compiler pragmas."

The call to `sub` in the following code prevents the compiler from automatically parallelizing the loop. The `force_parallel` pragma overrides the compiler's decision, and the compiler generates parallel code for the loop.

```
 ...
#pragma _CNX force_parallel
for( i=0; i<n; i++ )
  sub( a, b, i );

/* compile file containing sub() with -re */
void sub( float a[], float b[], int i ){
  a[i] = b[i] * 3.14;
}
```

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 function is called only from within a parallelized loop, compile the function at a level lower than `-O3`. Only one loop at a time can be run in parallel. Parallelizable code within the function cannot execute in parallel. Additional code generated to parallelize the called routine is useless overhead.

**Caution**

**If you use `force_parallel` to parallelize a loop containing an actual recurrence, the behavior of the loop can change from one execution to the next. Errors can 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 pragma.**

For more information about compiler pragmas, refer to Appendix C.

## 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 below has no dependencies. The compiler can strip mine and vectorize the inner loop and parallelize the strip-mine loop.

```
for( i=0; i<n; i++ ){
  a[i] += 3.14;
}
```

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 following loop has a backward loop-carried dependency (LCD) caused by the assignment to `a[i+1]`. The compiler cannot vectorize or parallelize the loop, so the loop remains in scalar terms.

```
for( i=0; i<n-1; i++ ){
  a[i+1] = a[i] + 3.14;
}
```

The loop below 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.

```
for( i=0; i<n-1; i++ ){
  a[i] = a[i+1] + 3.14;
}
```

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 needed, until an operation in another thread, on which the halted thread depends, has been performed.

The `synch_parallel` pragma instructs the compiler to generate synchronization code. More information about CONVEX C pragmas appears in Appendix C.

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 below:

```
for( i=0; i<n; i++ ){
  if( a[i] <= 0 ){
    s += b[i] * c[i];
    x = b[i];
  }
}
```

## Parallelizing code outside of loops

The compiler does not automatically parallelize code outside a loop. You can use tasking pragmas to instruct the compiler to parallelize such code. The `begin_tasks` pragma tells the compiler to begin parallelizing a series of tasks. The `next_task` pragma marks the end of a task and the start of the next task. The `end_tasks` pragma marks the end of a series of tasks to be parallelized. For more information about tasking pragmas, see Appendix C.

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

```
#pragma _CNX begin_tasks
        statement 1
#pragma _CNX next_task
        statement 2
        statement 3
#pragma _CNX next_task
        statement 4
#pragma _CNX end_tasks
```

The compiler transforms this code into a parallel loop and creates machine code equivalent to that shown here:

```
#pragma _CNX force_parallel
for( i=0; i<3; i++ )
  switch( i ){
    case 0:
      statement 1
      break;
    case 1:
      statement 2
      statement 3
      break;
    case 2:
      statement 4
      break:
  }
```

# Optimizing C applications

# 5

This chapter describes a strategy for optimizing C 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 functions that account for the major part of your program's execution time. Once 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. Compiling the program

1. Compile the program with minimal optimizations (`-no`). Use the `-pa` option to include instrumentation for profiling with CXpa.

```
% cc -pa prog.c -o no.exe
```

2. Run the resulting program using CXpa, making note of the performance information. Save the program's output in `no.out` and check the output for correctness.

```
% cpa no.exe
CONVEX Performance Analyzer.
Type 'help' for help.
reading executable no.exe...
(cpa) monitor all
(cpa) run > no.out
(cpa) analyze
```

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 expected results, allowing for roundoff error, use CXdb to pinpoint and fix the logic error that is causing the problem. Refer to Chapter 9 for possible causes of such errors. If you are certain there is no logic error, check for violations of ANSI C standards (refer to Chapter 9). If the code does not violate ANSI C standards, use the `contact` utility to report a possible compiler problem.

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.

If you have an existing output, compare it with `no.out`:

```
% diff no.out original.out
```

There should be no differences reported, other than those that result from rounding floating-point numbers. Check any output to ensure it is correct.

## Step 2. Adding scalar optimizations

1. Compile the program with scalar optimization (`-O1`). Use the `-pa` option to include instrumentation for profiling:

```
% cc -O1 -pa prog.c -o O1.exe
```

2. Run this version under CXpa and note the CPU time of the program. Name the output file `O1.out`.

```
% cpa O1.exe
CONVEX Performance Analyzer.
Type 'help' for help.
reading executable O1.exe...
(cpa) monitor all
(cpa) run > O1.out
(cpa) analyze
```

(If you use one of the profilers contained in the CONVEX Consultant instead of CXpa, you can still perform most steps in this chapter. You cannot analyze individual loops, however. Refer to the *CONVEX Consultant User's Guide* to determine the appropriate options and commands for using the Consultant profilers.)

3. Compare the new output with the output from **Step 1**. Scalar optimization rarely affects output, so the results, allowing for roundoff differences, should be the same.

   If the output is significantly different, use a binary search to isolate the function responsible for the change. Compile half the functions at `-no` and the other half at
   `-O1`. Run the program and check the output to determine which half contains the function responsible for the change. Then, split the suspect group of functions in half. Compile half of the suspect functions at `-no` and the other half at `-O1`. Continue this process until you isolate the function causing the problem.

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

4. Run the program under CXpa. Note the program's total execution time and which functions use the most time. Concentrate your optimization efforts on these functions.

## Step 3. Adding 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 pragmas, encroaches on the limits of floating-point precision, or violates certain ANSI C standard restrictions, the code may no longer produce the same output after it has been vectorized. It is also possible 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 or if you have never done it before, 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. Adding selective vectorization

1. Look at the CXpa output from **Step 2**. Determine which functions use the most CPU time. Compile the most time-consuming functions at `-O2`. Compile the remaining functions at `-O1`. Use the `-pa` option for 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 function causing the problem. Check the source code and fix any errors that you find. If you do not find any logic errors, recompile the affected function at `-O1` and continue optimizing the rest of the program.

3. Run the vectorized program under CXpa. Take note of the program's total execution time and the most time-consuming functions. 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 functions that use a significant amount of CPU time in the new CXpa output and have not been vectorized. Continue until you have vectorized all time-consuming functions that can be properly vectorized; proceed to **Step 4**.

## Step 4. Improving vector performance

1. Run the vectorized program under CXpa to produce a loop-level profile of the time-intensive functions.

2. Study the profile and optimization report. Look for inner loops that are not vectorized and use large amounts of CPU time. The goal is to increase the number of vectorized loops. Look for apparent recurrences that prevent vectorization. If you can prove that a recurrence is only apparent, use the `no_recurrence` pragma to vectorize the loop.

   Complicated tests can prevent the compiler from vectorizing a loop. If a loop containing a complicated test does not vectorize, try removing the test from the loop.

3. Try the following techniques:

   - Simplify `if` tests. Even if a loop is vectorized, an embedded test can slow it down.
   - Simplify array subscripts. Complicated subscripts can slow down the execution of a loop.
   - Look for loops with short vector lengths. If the iteration count is less than five, the loop probably runs faster in scalar form. Use the `scalar` pragma to stop the compiler from vectorizing a short loop.
   - Look for unnecessary or inefficient strip mines. Use the optimization report to determine whether a vector loop is strip mined. Use the `max_trips` pragma to stop the compiler from creating unnecessary strip mines.
   - Watch for bank conflicts and adjust array dimensions if necessary.
   - Look for unnecessary type conversions and other coding practices that slow down execution.

   For more examples of how to tune your code's performance, see Chapter 7.

4. Recompile your code and run it under CXpa. Check the output. If it has changed, locate the pragma that caused the change and remove it. Check the profile. Note the effect of any changes you made on each function's CPU time. Some changes may cause your code to slow

**Note**

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

## Step 5. 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 chance 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 start 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 turn-around 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 functions use the most CPU time and compile them for parallelization. To do this, place the `-O3` option on the `cc` 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 function causing the problem. Check the source code and fix any errors that you find. If you do not find logic errors, recompile the affected function at `-O2` and continue optimizing the rest of the program.

3. Run the program under CXpa. Note the process virtual times in each function. Refer to 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 function have an efficiency less than or equal to one, parallelization of the function is probably counter-productive and should be removed. Refer to the *CXpa User's Guide* for information on interpreting process virtual time.

4. Repeat **Step 5a**, parallelizing those functions that use a significant amount of process virtual time in the new CXpa output and have not been parallelized. Continue until you have parallelized all functions that can be safely and productively parallelized; then proceed to **Step 6**.

## Step 6. Improving parallel performance

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

2. Study the CXpa profile and the optimization report. Look for loops that failed to parallelize. A scalar loop that uses 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 applying the `no_recurrence` or `force_parallel` pragma. If you find a real dependency, consider replacing the function with a call to a VECLIB routine that performs the same function in parallel. For more information about VECLIB, refer to the *CONVEX VECLIB User's Guide.*

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

C provides a set of simple and effective programming constructs that are readily optimized by advanced compilers such as the CONVEX C 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 C, floating-point variables and constants can be declared to be `float` or `double`. 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.

In the backward-compatible mode, all calculations involving variables of type `float` are performed in type `double`: the `float` operands are converted to `double`, the result is calculated in `double`, and when assigned it is converted back to type `float`. The conversion overhead required by `float` operations makes them more expensive than `double` operations.

## `-float sp_ops` command line option

There are two ways to increase the performance of `float` operations in the backward-compatible mode: use only type `double` or compile the program using the `-float sp_ops` command line option. The latter approach is much easier to implement because no code must be changed. In the ANSI C compatibility modes, the `-float sp_ops` command line option is the default.

The `-float sp_ops` command line option causes the compiler to perform all `float` operations using 32-bit instructions instead of 64-bit instructions. This option increases performance by removing conversions from type `float` to type `double`.

However, there are two conditions in which this option is not effective: when floating-point constants are used in calculations and when `float` arguments are passed to functions.

Floating-point constants default to type `double` in all compatibility modes of the compiler. The presence of a constant of type `double` in an operation forces other floating-point operands to be converted to type `double`. You can fix this problem by casting the constants to type `float`, by using the `F` suffix in the ANSI C compatibility modes, or by using the `-float sp_const` command line option.

The second situation in which the `-float sp_ops` has no effect is in function calls. In the backward-compatible mode, the compiler promotes `float` arguments in a function call to type `double`. In the ANSI C compatibility modes, this promotion occurs only when there is no function prototype for the called function.

## `-float sp_const` command line option

The `-float sp_const` command line option causes the compiler to treat all floating-point constants as type `float`. The default is to treat all floating-point constants as type `double`. This command line option increases performance by removing conversions from `float` to `double` in calculations that contain `float` operands mixed with `double` constants.

This option also permits certain optimizations to take place that are inhibited by data type conversions, such as reductions. For example, the following loop will not vectorize unless you compile with the `-float sp_const` command line option.

```
float xsum = 0.0;
for( i=0; i<n; i++ )
   xsum += a[i] * 3.1415;
```

## Integer operations

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 compile time or 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 constant amount on every iteration. If the incrementing or decrementing takes place only if some condition is true, then the variable is a conditional induction variable.

Counted loops can be `for` loops, `do` loops, `while` loops, or loops written with `if` and `goto` statements. The following example shows four typical counted loops.

```
int i;
float a[1000],b[1000],c[1000];

for( i=0; i<1000; i++ )
  a[i] += b[i];

i = 999;
do {
  a[i] += b[i] / 4.16F;
  --i;
} while( i>=0 );

i = 0;
while( i<1000 ){
  a[i] *= b[i];
  i += 4;
}

i = 0;
St: a[i] = b[i] * c[i];
  ++i;
if( i < 1000 )
  goto St;
```

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

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

```
i = 1;
while( i<1000 ){
  a[i] = b[i] * c[i];
  i = i * 2;
}
```

When this loop executes, it effectively 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 above can be unrolled by hand, as shown below. Because the loop overhead is eliminated, the unrolled code runs faster than the original loop.

```
a[  1] = b[  1] * c[  1];
a[  2] = b[  2] * c[  2];
a[  4] = b[  4] * c[  4];
a[  8] = b[  8] * c[  8];
a[ 16] = b[ 16] * c[ 16];
a[ 32] = b[ 32] * c[ 32];
a[ 64] = b[ 64] * c[ 64];
a[128] = b[128] * c[128];
a[256] = b[256] * c[256];
a[512] = b[512] * c[512];
```

If the iteration variable of a loop is incremented by a non-integer constant, the loop has no induction variable and the compiler cannot vectorize it. The loop below, for example, increments `i` by a `float` value, which prevents vectorization of the loop.

```
int i = 0;
float z = 4.0;
float a[1000];

while( i<1000 ){
  a[i] *= z;
  i += z;
}
```

**Caution**

**If the start, stop, or iteration value of a loop falls outside the range of signed integer (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 signed integers.**

For a `while` loop to vectorize, the `while` test must compare the induction variable to a fixed stop value. The test can use any of these comparison operators: >, <, >=, <=.

More complicated iteration tests, such as the one shown here, often prevent the compiler from vectorizing a loop:

```
j = 0;
i = 1;
while( (i<1000) && (j<1000) ){
  a[i] = a[i+j];
  ++i;
  j += m;
}
```

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

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

```
i = 0;
n = 1000;
while( i<n ){
  a[i] = b[i];
  if( a[i+1] > 0 )
    n = a[i+1];
  ++i;
}
```

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 following example, do not vectorize.

```
for(i=0; i<1000; i++ ){
  a[i] = c[i] + b[i];
  if( a[i] < 0.0 )
    break;
}
```

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.

The following example shows a series of conditionals embedded within a `for` loop. The conditionals do not prevent vectorization of the loop, but they do cause the generated code to execute more slowly.

```
#include <math.h>
main()
{
  int i;
  float a[10000],b[10000],c[10000],d[10000];

  for( i=0; i<10000; i++ ){
    if( i<2000 ){
      c[i] = a[i] * 2000.0 + cos(a[i]);
      b[i] = b[i] * c[i] * c[i] / a[i];
    }
    if( i>=2000 && i<4000 ){
      c[i] = a[i] + cos(a[i]);
      b[i] = b[i] + c[i];
    }
    if( i>=4000 && i<6000 ){
      c[i] = a[i] + 2000.0;
      b[i] = b[i] * b[i] * b[i];
    }
    if( i>=6000 ){
      c[i] = a[i];
      b[i] = 1.0;
    }
  }
}
```

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

```
#include <math.h>
main()
{
  int i;
  float a[10000],b[10000],c[10000],d[10000];

  for( i=0; i<2000; i++ ){
    c[i] = a[i] * 2000.0 + cos(a[i]);
    b[i] = b[i] * c[i] * c[i] / a[i];
  }
  for( i=2000; i<4000; i++ ){
    c[i] = a[i] + cos(a[i]);
    b[i] = b[i] + c[i];
    }
  for( i=4000; i<6000; i++ ){
    c[i] = a[i] + 2000.0;
    b[i] = b[i] * b[i] * b[i];
  }
  for( i=6000; i<10000; i++ ){
    c[i] = a[i];
    b[i] = 1.0;
  }
}
```

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 C, arrays are stored in row-major order. As a result, using innermost loops that vary the trailing, or rightmost, dimension is faster than using innermost loops that vary the leading, or leftmost, dimension. Write loop nests so that the inner loop accesses the trailing dimension.

CONVEX C automatically interchanges many loops to optimize the efficiency of array accesses. Vector stride and memory interleaving also affect a loop's efficiency. These issues are discussed later in this chapter. The following example shows three loops in order of decreasing efficiency:

```
for( i=0; i<n; i++ ) /* most efficient */
  a[0][0][i] = 4.0;

for( i=0; i<n; i++ )
  a[0][i][0] = 4.0;

for( i=0; i<n; i++ ) /* least efficient */
  a[i][0][0] = 4.0;
```

In the following example, the compiler interchanges the `j` and `i` loops.

**Original Code**

```
for(i=0; i<n; i++ )
  for(j=0; j<n; j++ )
    a[j][i][1] = 4.0;
```

**Optimized Code**

```
for(j=0; j<n; j++)
  for(i=0; i<n; i++)
    a[j][i][1] = 4.0;
```

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. 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 trailing dimension of an array, you can minimize the number of compromises the compiler must make.

## Memory interleaving

The CONVEX C3 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 are optimal. Figure 3 shows the configuration of four-byte data (which may be `float` or `int`) stored in a four-bank machine. Byte addresses are expressed in decimal notation.

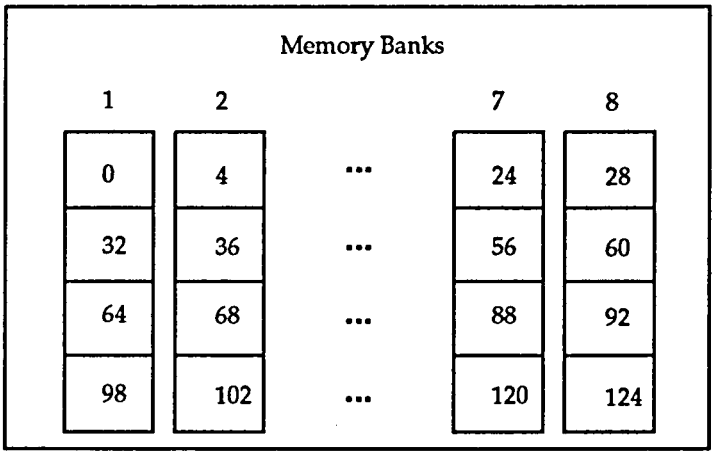


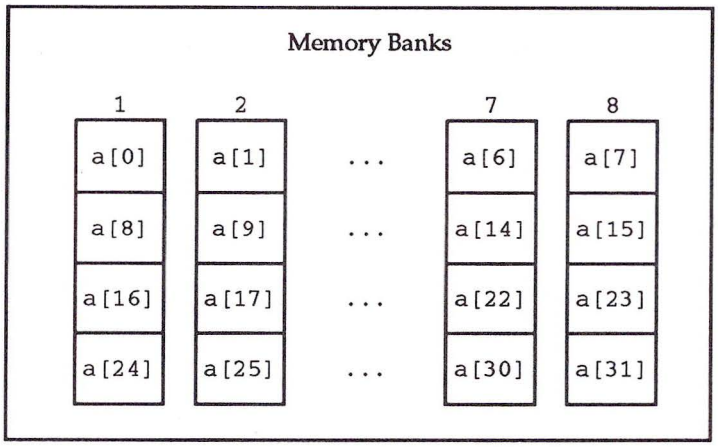

**Figure 3**
Eight-way interleaved memory

If your system is running ConvexOS 9.0 or higher, you can determine your computer's memory interleave using the `getsysinfo` command. Refer to the getsysinfo(1) man page for more information on its use.

Returning data at the rate of one word per clock cycle requires eight memory banks. A load instruction, for example, takes eight cycles to return data. If a program makes eight load requests, at a rate of one request per clock cycle, each to a separate bank, data returns at a rate of one per clock cycle, beginning eight clock cycles after the first request. Memory interleaving directly affects efficient array accesses.

Figure 4 shows a one-dimensional array in eight-way interleaved memory.

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




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

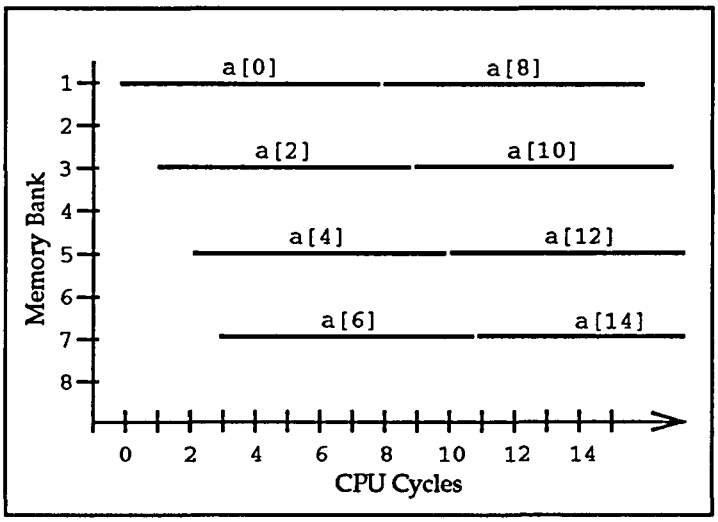
```
for( i=0; i<32; i++)
  a[i] += 1.0;
```

The loop below causes memory bank conflicts. The CPU must wait for memory requests to be filled.

```
for( i=0; i<32; i+=2 )
  a[i] += 1.0;
```

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

**Figure 5**
Bank conflict




Load requests occur each clock cycle. Accessing `a[0]` uses bank 1 for eight clock cycles. The CPU cannot access `a[8]` until this first access is complete. This takes four clock cycles.

Bank conflicts occur when an array's stride uses memory inefficiently. Stride is the difference in index value between two successive iterations. In the loop below, `arr` has a stride of one:

```
for( i=0; i<32; i++ )
  arr[i] += 1;
```

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

```
for( i=0; i<32; i+=2 )
  arr[i] += 1;
```

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

## Multidimensional arrays

C arrays are stored in row-major order: `a[0][0]`, `a[0][1]`, `a[0][2]`, and `a[0][3]` are stored in contiguous memory locations. In other languages, for example, FORTRAN, arrays are stored in column-major order: `A(1,1)`, `A(2,1)`, `A(3,1)`, and `A(4,1)` are stored contiguously.

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

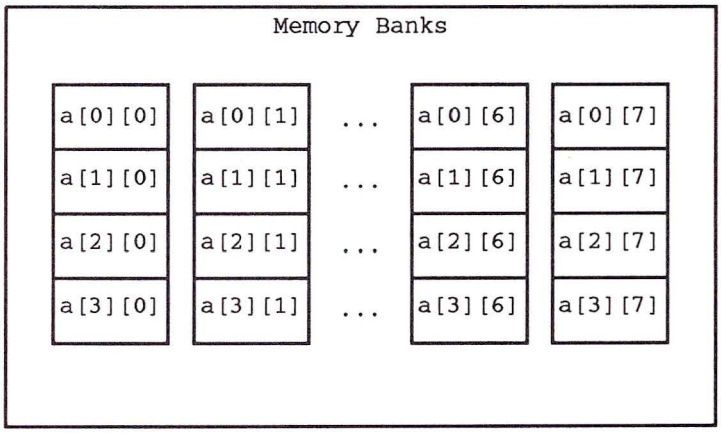



**Figure 6**
Two-dimensional array stored in eight-way interleaved memory

The following loop processes a two-dimensional array:

```
float a[4][8];
for( j=0; j<8; j++ )
  for( i=0; i<4; i++ )
    a[i][j] += 1;
```

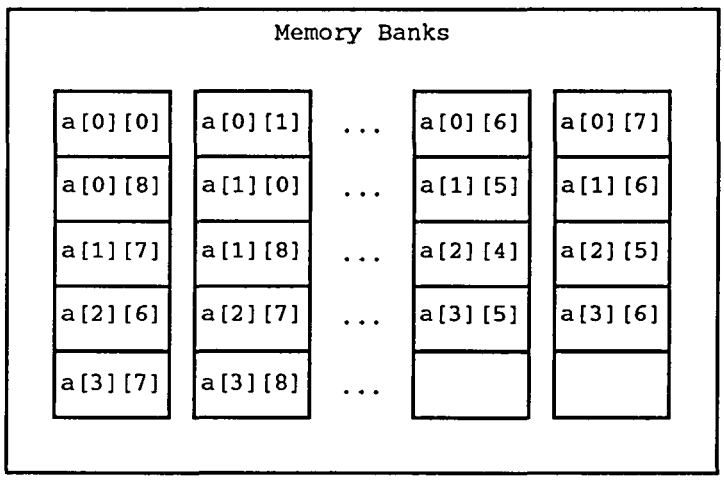
When the inner loop is vectorized, the vector register load and vector store have a stride of eight. Only one memory bank is used in the inner loop, as shown in Figure 6, and eight clock cycles are required to load each element into the vector register.

To avoid bank conflicts, declare the trailing index of a to be an odd number, as shown in the following loop:

```
float a[4][9];
for( j=0; j<8; j++ )
  for( i=0; i<4; i++ )
    a[i][j] += 1;
```

Figure 7 shows how this array is accessed and arranged in memory. The elements of the ninth column are never used, but they force each column to start in a different memory bank, which resolves bank conflicts.

**Figure 7**
Trailing 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 `short int` which occupies a half word, and `char`, 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. A `short int` array incurs bank conflicts on every other memory access. A `char` array incurs bank conflicts on three out of every four memory accesses. Avoid using `char` data types in a loop whenever possible.

# Manual optimization techniques

7

No matter how sophisticated the compiler is, optimization is more an art than a science. This chapter presents optimization techniques that C 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 C programs.

## Eliminate unnecessary strip mines

If the compiler determines that the iteration or 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 iteration counts. Unless the compiler can determine the value of the iteration variable (through constant propagation, for example), the compiler must strip mine the loop to allow for a possible iteration count greater than 128. The following code shows a loop with an iteration count that varies between 1 and 50:

```
n = getval(n);/* returns a value from 1 to 50 */
for( i=0; i<n; i++ )
  a[i] = b[i] * c[i];
```

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` pragma to prevent strip mining the loop, as shown below:

```
n = getval(n);
#pragma _CNX max_trips 50
for( i=0; i<n; i++ )
  a[i] = b[i] * c[i];
```

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

## Do not vectorize loops with small iteration counts

Look for loops with small iteration counts. For short loops, the benefit of vectorization usually does not outweigh the overhead. When it can determine the iteration count, the compiler does not vectorize a loop that is too short to vectorize profitably. Often, however, the iteration count depends on a variable whose value is unknown to the compiler. If you know that a loop has a short iteration count, you can prevent vectorization using the `scalar` pragma or unroll the loop using the `unroll` pragma.

Because it cannot determine the iteration count, the compiler vectorizes and strip mines the loop below:

```
n = getval(n); /* returns n = 1, 2, 4, or 8 */
for( i=0; i<n; i++ )
  a[i] = b[i] * c[i];
```

You can use the `max_trips` pragma to prevent the compiler from strip mining the loop, but that still produces an inefficient vector loop. When this loop is executed with `n` equal to 1, 2, 3, or 4, the iteration count is too short to justify vectorization.

You can rewrite this loop and use the `scalar` pragma to eliminate the overhead of a vectorized loop when the iteration count is less than 5. Here is an example:

```
n = getval(n);
if( n>=5 ) {
# pragma _CNX max_trips 32
  for( i=0; i<n; i++ )
    a[i] = b[i] * c[i];
} else {
# pragma _CNX scalar
  for( i=0; i<n; i++ )
    a[i] = b[i] * c[i];
}
```

Instead of rewriting the loop and using the `scalar` pragma, you can use the `select` pragma, which tells the compiler to create multiple versions of the loop.The following example shows the use of the `select` pragma.

```
n = getval(n);
#pragma _CNX select (4, *, *)
for( i=0; i<n; i++ ){
  a[i] = b[i] * c[i];
}
```

`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 iteration count is greater than or equal to four. The second and third arguments (*) tell the compiler not to create parallel and vector-parallel versions of the loop.

Because the `select` pragma does not require rewriting code, this approach is usually safer and easier. The `select` pragma overrides the `max_trips` pragma.

You can specify dynamic selection using the `-ds` option instead of the `select` pragma. This affects all loops with variable iteration counts within the compilation unit. The code created selects a version of the loop based on the same crossover points used by the compiler for loops with fixed iteration counts. Use this option with care. Indiscriminate use of `-ds` can cause some code to slow down.

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

**Original loop**

```
#pragma _CNX unroll
for( i=0; i<4; i++ ){
  a[i] += 1;
}
```

**Loop unrolled**

```
a[0] += 1;
a[1] += 1;
a[2] += 1;
a[3] += 1;
```

If a loop is too complex, or its iteration count is unknown, the compiler cannot completely unroll it. Partial unrolling may still be possible. The compiler unrolls the statement inside the loop into four statements and reduces the loop's iteration count. Here is an example:

**Original loop**

```
#pragma _CNX unroll
for ( i=0;i<=15;i++ )
  b[i] = b[i-1];
```

**Loop partially unrolled**

```
for ( i=0;i<=15;i+=4)
  {
   b[i] = b[i-1];
   b[i+1] = b[i];
   b[i+2] = b[i+1];
   b[i+3] = b[i+2];
}
```

If the iteration count is not evenly divisible by four, the compiler creates a second loop, following the divisible loop, to handle the remaining one, two, or three iterations.

**Original loop**

```
#pragma _CNX unroll
for ( i=0;i<=18;i++ )
  b[i] = b[i-1];
```

**Loop partially unrolled**

```
for ( i=0;i<=15;i+=4)
  {
   b[i] = b[i-1];
   b[i+1] = b[i];
   b[i+2] = b[i+1];
   b[i+3] = b[i+2];
}
for ( i=16; i<=18; i++)
  b[i] = b[i-1];
```

The compiler can sometimes unroll the second loop.

The `unroll` pragma must be used on the innermost loop. It is ignored on other loops. Alternatively, the `-ur` option causes the compiler to perform loop unrolling on loops selected by the compiler on the basis of profitability.

## Promoting arrays

Sometimes it is necessary to promote an array to a higher dimension to vectorize a loop. In the following code, 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 dependencies between assignments to `b[n]`. This cycle prevents the compiler from distributing the `i` loop.

```
float gls[1000];
int i, j;
float b[4], p[4], q[4];

for( i=0; i<1000; i++ ){ /* scalar */
  b[0] = gls[i +10] * p[0] + gls[i + 9] * p[1]
       + gls[i + 7] * p[2] + gls[i + 4] * p[3];
  b[1] = gls[i + 1] * p[0] + gls[i + 5] * p[1]
       + gls[i + 8] * p[2] + gls[i +10] * p[3];
  b[2] = gls[i + 5] * p[0] + gls[i + 2] * p[1]
       + gls[i + 6] * p[2] + gls[i + 9] * p[3];
  b[3] = gls[i + 8] * p[0] + gls[i + 6] * p[1]
       + gls[i + 3] * p[2] + gls[i + 7] * p[3];
  for( j=0; j<4; j++ ){ /* vector */
    q[j] += b[j];
  }
}
```

To eliminate the recurrence, promote b to a two-dimensional array, as shown here:

```
float gls[1000];
int i, j;
float b[4][1001], p[4], q[4];

for( i=0; i<1000; i++ ){
  b[0][i]= gls[i +10] * p[0] + gls[i + 9]
         * p[1] + gls[i + 7] * p[2]
         + gls[i + 4] * p[3];
  b[1][i]= gls[i + 1] * p[0] + gls[i + 5]
         * p[1] + gls[i + 8] * p[2]
         + gls[i +10] * p[3];
  b[2][i]= gls[i + 5] * p[0] + gls[i + 2]
         * p[1] + gls[i + 6] * p[2]
         + gls[i + 9] * p[3];
  b[3][i]= gls[i + 8] * p[0] + gls[i + 6]
         * p[1] + gls[i + 3] * p[2]
         + gls[i + 7] * p[3];
  for( j=0; j<4; j++ ){
    q[j] += b[j][i];
  }
}
```

In the modified code, the calculation of `b[i][n]` is independent of the calculation of `b[i+1][n]`. These independent calculations permit the compiler to distribute the `i` loop, as shown here:

```
float gls[1000];
int i, j;
float b[4][1001], p[4], q[4];

for( i=0; i<1000; i++ ){ /* vector */
  b[0][i] =gls[i + 10] * p[0] + gls[i + 9] *
           p[1] + gls[i + 7] * p[2] +
           gls[i + 4] * p[3];
  b[1][i] =gls[i + 1] * p[0] + gls[i + 5] *
           p[1] + gls[i + 8] * p[2] +
           gls[i + 10] * p[3];
  b[2][i] =gls[i + 5] * p[0] + gls[i + 2] *
           p[1] + gls[i + 6] * p[2] +
           gls[i + 9] * p[3];
  b[3][i] =gls[i + 8] * p[0] + gls[i + 6] *
           p[1] + gls[i + 3] * p[2] +
           gls[i + 7] * p[3];
}
for( j=0; j<4; j++ )    /*interchanged scalar*/
  {
   s0 = q[j];            /*hoisted load*/
   for(i=0;i<1000;i++)   /*interchanged*/
     s0 += b[j][i];      /*vector reduction*/
   q[j] = s0;            /*sunken store*/
  }
```

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 this code to execute.

# Aliasing

An *alias* is an alternate name for some object. Aliasing occurs in a program when two or more names are attached to the same memory location. A possible or *potential alias* occurs when the compiler cannot be sure whether two names refer to distinct memory locations.

A potential alias can prevent the compiler from vectorizing a piece of code. The judicious use of options, qualifiers, and pragmas allows you to vectorize some code that cannot otherwise be vectorized. Care must be taken to ensure that this does not change the results of the program.

## Aliasing and dependency

Aliasing compounds the problems of dependency and recurrence.

Consider the following example:

```
for ( i=0; i<n; i++ )
  arra[i] = arrb[i];
```

This loop appears free of backward dependencies. If `arra` and `arrb` are distinct, the compiler can safely vectorize it. If `arra` is an alias for `arrb` (that is, if `arra` and `arrb` overlap), however, a backward dependency can exist. For example, if `arra` and `arrb` overlap by `K` elements, the code shown above is equivalent to the code shown below:

```
for ( i=0; i<n; i++ )
  arrb[i+K] = arrb[i];
```

In this code, a dependency exists. If K is negative, the dependency is forward, and the compiler can safely vectorize the loop. If K is positive, the dependency is backward, and the compiler cannot safely vectorize the loop.

The compiler does not know the value of K, so it cannot determine whether aliasing causes a backward dependency. To avoid wrong answers, the compiler does not vectorize a loop when such aliasing exists.

## Why aliasing occurs

Both explicit and implicit aliasing occur in C. Explicit aliasing, in the form of unions, rarely causes problems in vectorization. Implicit aliasing, which results from pointers, often causes problems.

Consider the function `alex`:

```
void alex(a, b, n)
float *a, *b;
int n;
{
  int i;
  for ( i=0; i<n; i++ )
    a[i] = b[i] / 2.0;
}
```

This function has two parameters, `a` and `b`, that point to arrays of type `float`. The compiler does not know the addresses passed to these pointers. The array that `a` points to may overlap the array that `b` points to. The possible overlap creates a *potential alias*. Because of this potential alias, a dependency may exist, and the compiler does not vectorize the loop.

If you know that the arrays pointed to by `a` and `b` do not overlap, you can vectorize this loop by using a `no_recurrence` pragma. If `a` and `b` do point to the same array, however, an alias exists, and using the `no_recurrence` pragma is incorrect.

## Aliasing algorithms

The compiler has two different algorithms for detecting potential aliases. The algorithm chosen depends on the compatibility mode. CONVEX C compilers prior to V4.0 use worst-case aliasing. In backward-compatible (non-ANSI) mode, the current compiler chooses the "worst-case" algorithm used by CONVEX C compilers prior to V4.0. In ANSI or extended mode, however, the compiler chooses the newer, ANSI-C aliasing algorithm.

The older, worst-case aliasing algorithm views all pointer references as subscripts into a giant array that encompasses all of memory. This mythical array (known as `*MEM*` in the compiler's optimization report) includes all global variables and local variables whose address is taken using the address ( & ) operator. Here's the result:

- Every pointer reference is a potential alias for every other pointer reference.
- Every pointer reference is a potential alias for every global variable (and vice versa).
- Every pointer reference is a potential alias for every local variable that is used with &.

ANSI C provides stricter type-checking. This allows the current compiler to use a stricter algorithm that eliminates many potential aliases found by the worst-case algorithm. Instead of one giant `*MEM*` array, the ANSI-C algorithm uses a model with separate arrays for each base type (such as `int`, `float` , or `double`). Pointers and variables cannot alias with pointers or variables of a different base type.

## Note

**The compiler applies the chosen aliasing algorithm on a function-wide basis, before optimization takes place. Beginning with CONVEX C V5.0, aliasing algorithms are flow sensitive. They can detect whether a potential alias begins before a loop or following a loop, and aliases that begin following a loop do not interfere with vectorization.**

The ANSI aliasing algorithm permits the compiler to vectorize the code shown below; the worst-case algorithm does not.

```
void alex1(a, ib, n)
float *a;
int *ib, n;
{
  int i;
  for ( i=0; i<n; i++ )
    a[i] = ib[i]/2.0;
}
```

Pointers `a` and `ib` point to different base types. The worst-case algorithm considers them to be aliased through `*MEM*`, but under ANSI C rules, no potential alias exists, and the loop is vectorized.

The ANSI C aliasing algorithm may not be safe if your program is not ANSI compliant. The non-ANSI-compliant code shown below allows two pointers of different base types to become aliased with one another. (The assignment to `fptr` makes the code non-ANSI C compliant.)

```
int array[100];
int *dummy;
float *fptr;

*dummy = array;/* illegal pointer/integer
                  combination. */
fptr = dummy;  /* operands of = point to
                  incompatible types. */
```

Compiling this code generates the warning messages shown in the comments. (Compiling with `-d arg_ptr_ref=e` converts these warnings into error messages.)

Because of the ANSI standard violation, this code does not compile safely under the ANSI aliasing algorithm.

In the following example, function `caller` passes two `long int` pointers to `foo`, which expects one pointer to long int and one pointer to `short int`. Because `ia` and `ib` have different base types, the ANSI C aliasing algorithm assumes that no alias between `*ia` and `*ib` can exist. In ANSI mode, the compiler compiles this code (with a warning) and vectorizes the loop even though aliasing and a recurrence do exist.

```
void foo(long int *ia, short int *fb)
{
  int i;
  for (i=0; i<500; i++ ) {
    ia[i]=1;
    fb[i] = ia[i];
  }
}
caller()
{
  long int arra[600];
  foo(&arra[0], &arra[100]);
}
```

## Specifying an aliasing mode

To specify an aliasing mode, use one of the following command line options:

- `-alias cautious`
- `-alias standard`
- `-alias worst`

In `cautious` mode, the compiler uses the ANSI C aliasing algorithm. If the compiler finds constructs that could cause hidden aliasing, it switches to the worst-case aliasing algorithm. If you do not specify an aliasing mode, the compiler uses `cautious` mode by default.

In `standard` mode, the compiler always uses the ANSI-C aliasing algorithm.

In `worst` mode, the compiler uses the worst-case aliasing algorithm.

## Pointer tracking

Beginning with version 5.0, the CONVEX C compiler uses pointer tracking to eliminate many of the potential aliases found by the ANSI or `-pcc` alias-detection algorithm.

This powerful algorithm tracks all pointer assignments and references within its scope. Using information available at compile time, it determines what addresses pointers can be set to. Using this information, it then separates pointers into classes, where pointers in one class are aliased with other pointers in that class but cannot be aliased with pointers in another class. If the alias-detection algorithm finds a possible alias between two pointers that turn out to be in different classes, pointer tracking can eliminate that possible alias.

Like many optimizations, the CONVEX C implementation of pointer tracking has "global" scope. The compiler can track pointer assignments and uses within a function but not between functions. If a possible alias involves two local variables, pointer tracking can help determine whether an alias actually exists. If it involves a formal argument or a global or static variable, pointer tracking cannot determine whether an alias exists under all circumstances.

Consider the following example. This function takes two pointers, `ia` and `ib`, and copies the arrays they point to into local arrays, `lia` and `lib`. It then sets the pointers to the addresses of the local arrays and adds the contents of `lia` and `lib`, placing the results into `lib`.

```
void foo(int *ia, int *ib)
{
  int i;
  int lia[500]; int lib[500];

  for (i=0; i<500; i++ ) {
    lia[i] = ia[i];
    lib[i] = ib[i];
  }

  ia = &lia[0];
  ib = &lib[0];
  for (i=0; i<500; i++ )
    lib[i] = lib[i] + lia[i];
  ...
}
```

Does a possible alias, which may cause a recurrence, exist in this function? According to the ANSI aliasing algorithm, yes. The function takes the address of `lia` and `lib`, which are both integer-type arrays. This means that `lia` and `lib` could become aliased. A potential recurrence exists and the loop that adds the two together cannot be vectorized.

Even flow-sensitive analysis does not help. The code takes the addresses of `lia` and `lib`, causing them to become potential aliases for `*ia` and `*ib` before the loop occurs. Pointer tracking can help, though. The compiler knows the addresses of `lia` and `lib` and knows that these are separate memory locations. It sees that these addresses are assigned to `ia` and ib, respectively. Each pointer now has only one possible value, and these are separate values; `ia` and `ib` belong to separate pointer classes. Because the pointers belong to separate classes, no alias exists, and the compiler vectorizes the loop.

In the following example, aliasing can occur outside the scope of the function. Pointer tracking cannot help. Function `foo` receives two pointers that are used to access and add the contents of two arrays. If `ib` points to an array that overlaps `*ia`, an alias exists and the loop cannot be vectorized. Pointer tracking begins and ends within the function. The compiler cannot tell what addresses are passed to `ia` and `ib`. This prevents the compiler from vectorizing the loop.

```
void foo(int *ia, int *ib)
{
  int i;

  for (i=0; i<500; i++ )
    ib[i] = ib[i] + ia[i];
}
```

Pointer tracking can allow the compiler to vectorize the same loop if you give the compiler some assurance that the pointers are distinct at the beginning of the function. One way to do this is by compiling with the `-alias ptr_args` option, which tells the compiler to treat all pointer parameters as nonoverlapping arrays. (This requires that you treat the parameters as arrays, rather than pointers, in your code as well.)

Another way to assure the compiler that pointer parameters are distinct is to use the `restrict` qualifier. (Make sure you include the header file `restrict.h`.)

```
#include <restrict.h>
void foo(int * restrict ia, int * restrict ib)
{
  int i;

  for (i=0; i<500; i++ )
    ib[i] = ib[i] + ia[i];
}
```

The `restrict` qualifier tells the compiler that a pointer provides exclusive access to the data object at a given memory location. When you use the `restrict` qualifier, you must access the object pointed to by a `restrict` pointer only through that pointer. If you reference the object by some other means, the compiler may incorrectly optimize code as a result.

The `restrict` qualifier is an extension to ANSI C and is not available in backward-compatible mode.

**Note**

**If you need or want to disable pointer tracking for any reason, you can do this by using the `-nptr` option on the `cc` command line.**

Pointer tracking works on the level of complete arrays rather than array sections. If two pointers become aliased to two sections of the same array, as in the following example, pointer tracking considers both pointers to be potential aliases for the entire array (and thus for each other). Because of the potential alias between the pointers, the compiler does not vectorize the loop.

```
void foo(float c[], int n)
{
  float *a, float *b;
  int i;
  a=&c[0];
  b=&c[n];
  for (i=0; i<n; i++ )
    a[i]=b[i];
}
```

In this case, you can see that no aliasing actually occurs. The section of array `c` accessed by pointer `a` runs from `c[0]` to `c[n-1]`, while the section accessed by `b` runs from `c[n]` to `c[2n-1]`. These sections do not overlap, so you can safely vectorize this loop using the `no_recurrence` pragma.

Pointer tracking does not account for conditional control flow. Potential aliasing within a conditional branch can affect loops that are not reachable from that branch.

## Iteration and stop values

Aliasing a variable in an array subscript can make it unsafe for the compiler to vectorize a loop.

In the following example, the code passes `&j` to `getval`; `getval` can use that address in any number of ways including, possibly, assigning it to `iptr`. (Even though `iptr` is not passed to `getval`, `getval` might still access it as a global variable or through another alias.) This makes `j` a potential alias for `*iptr`.

```
void subex(iptr, n, j)
int *iptr, n, j;
{
  n = getval(&j,n);

  for (j--; j<n; j++)
    iptr[j] += 1;
}
```

This potential alias means that `j` and `iptr[j]` might occupy the same memory space for some value of `j`. The assignment to `iptr[j]` on that iteration would also change the value of `j` itself. The possible alteration of `j` prevents the compiler from safely vectorizing the loop. The Optimization Report says that no induction variable could be found for the loop, and the compiler does not vectorize the loop even if the `no_recurrence` pragma is used.

Avoid taking the address of any variable that will be used as the iteration variable for a loop. To vectorize the loop in `subex`, use a temporary variable, `i`:

```
void subex(iptr, n, j)
int *iptr, n, j;
{
  int i;
  n = getval(&j,n);
  i=j;
  for (i--; i<n; i++)
    iptr[i] += 1;
}
```

By preventing the compiler from finding an induction variable, aliasing can prevent vectorization of a loop even when no possibility of recurrence exists.

In the next example, `ialex` takes the address of `j` and assigns it to `*ip`. Thus, `j` becomes an alias for `*ip` and, potentially, for `*iptr`. Assigned values to `iptr[j]` within the loop could alter the value of `j`. As a result, the compiler cannot use `j` as an induction variable and, without an induction variable, it cannot count the iterations of the loop. When the compiler can't count the iterations of a loop, the compiler can't vectorize it.

```
int *ip
void ialex(iptr)
int *iptr;
{
 int j;
 *ip = &j;
 for (j=0; j<2048; j++ )
   iptr[j] = 107;
}
```

Once again, the `no_recurrence` pragma does not help. (The problem is not caused by a recurrence.) To vectorize the loop, remove the line of code that takes the address of `j` or introduce a temporary variable.

Using a pointer as a loop counter prevents vectorization also. Compiling the following function, the compiler finds that `*j` is not an induction variable (because an assignment to `iptr[*j]` could alter the value of `*j` within the loop). The compiler does not vectorize the loop.

```
void ialex2(iptr, j, n)
int *iptr;
int *j, n;
{
  for (*j=0; *j<n; (*j)++ )
    iptr[*j] = 107;
}
```

Aliasing of stop variables can prevent vectorization also. In the following code, the stop variable `n` becomes a possible alias for `*iptr` when `&n` is passed to `foo`. This means that `n` can be altered during the execution of the loop. As a result, the compiler cannot count the number of iterations and cannot vectorize the loop.

```
void salex(int *iptr, int n)
{
  int i;
  foo(&n);
  for ( i=0; i<n; i++ )
    iptr[i] += iptr[i];
  return;
}
```

To vectorize the affected loop, eliminate the call to `foo`, move the call below the loop (in which case, flow-sensitive analysis takes care of the aliasing), or create a temporary variable as shown below:

```
void salex(int *iptr, int n)
{
  int i, tmp;
  foo(&n);
  tmp = n;
  for ( i=0; i<tmp; i++ )
    iptr[i] += iptr[i];
  return;
}
```

Because `tmp` is not aliased to `iptr`, the loop has a fixed stop value and the compiler vectorizes it.

## Global variables

Potential aliases involving global variables cause optimization problems in many programs. The compiler cannot tell whether another function causes a global variable to become aliased.

The following code uses a global variable, n, as a stop value. Because n may have its address taken and assigned to ik outside the scope of the function, n must be considered a potential alias for *ik. The value of n, therefore, can be altered on any iteration of the loop. The compiler cannot determine the stop value and cannot vectorize the loop.

```
int n, *ik;
void foo(int *ik)
{
  int i;

  for ( i=0; i<n; i++ )
    ik[i]=i;
}
```

Using a temporary local variable solves the problem.

```
int n;
void foo(int *ik)
{
  int i,stop = n;

  for ( i=0; i<stop; ++i )
    ik[i]=i;
}
```

If ik is a global variable instead of a pointer, the problem does not occur. Global variables do not cause aliasing problems except when pointers are involved.

```
int n, ik[1000];
void foo()
{
  int i;

  for ( i=0; i<n; i++ )
    ik[i] = i;
}
```

## Array parameters

When C passes an array parameter to a function, it passes it as a pointer. This increases the number of potential aliases and could reduce vectorization. CONVEX C provides an option, however, that tells the compiler to treat array parameters, for aliasing purposes, as arrays. (This does not affect the way arrays are actually passed.) This option is `-alias array_args`.

Potential aliasing prevents vectorization of the loop in `parex` below. Because they are passed as pointers, arrays `arra` and `arrb` are potential aliases under both the worst-case and ANSI aliasing rules.

```
void parex(float arra[], float arrb[], int n)

{
  int i;
  for (i=0; i<n; i++ )
    arra[i] = arrb[i] + i;
}
```

If you compile with the `-alias array_args` option, the CONVEX C compiler considers `arra` and `arrb` to be array variables, rather than pointers, for aliasing purposes. An array variable does not cause aliasing problems unless you use it as a global or take its address and assign it to a pointer. Neither of those things are done here, so the loop vectorizes when compiled with `-alias array_args`.

An array can be declared as a pointer in the function definition and accessed as an array in the function body, as shown here:

```
void parex(float *arra, float *arrb, int n)

{
  int i;
  for (i=0; i<n; i++ )
    arra[i] = arrb[i] + i;
}
```

Because `arra` and `arrb` are declared as pointers, you cannot vectorize the loop using `-alias array_args`. You must use `-alias ptr_args` instead.

If you use `-alias ptr_args`, all pointer parameters declared as pointers must be treated as arrays. Accessing an array parameter as a pointer causes a syntax error when you compile with this option. The following code, for example, generates an error when compiled with the `-alias ptr_args` option:

```
void parex(float *a, float *b)

{
  int i;
  for (i=0; i<100; i++ )
    *a++ = *b++;
  ...
}
```

If you have a function that uses pointer parameters and does not treat them as arrays, you can use `-alias restrict_args` instead of `-alias ptr_args`. The `-alias restrict_args` option tells the compiler to treat the code as though a `restrict` qualifier (described in the next section) was applied to each pointer parameter.

**Caution**

**Neither `-alias array_args`, `-alias ptr_args`, nor `-alias restrict_args` prevents aliasing of array parameters. If you use these options, you take responsibility for ensuring that array parameters are distinct. If you pass overlapping parameters, these options hide that fact from the compiler. Undetected dependencies, incorrect optimization, and errors can result.**

ANSI C declares the ability of two array parameters to point to the same object to be an obsolescent feature. To ensure compatibility with future versions of ANSI C, and to minimize aliasing, do not write code that passes overlapping arrays to a function.

## Restricting pointers

The `-alias array_args` and `-alias ptr_args` options do not target specific pointers, but affect entire files. This can prevent you from using these options in some cases where code might benefit.

Consider the following example:

```
void foo(int *a,int *b,int *c,int*d)
{
  int i;
  for (i=0; i<1000; i++)
    a[i] += b[i];
  for (i=1; i<1000; i++)
    c[i] += d[i-1];
}

void caller()
{
  int *ptr1, *ptr2, *ptr3;
  ...
  foo(ptr1, ptr2, ptr3, ptr3);
}
```

The first loop in function `foo` has an apparent recurrence because of the potential alias between `*a` and `*b`. The second loop has an genuine recurrence because of the alias between `*c` and `*d`. Using `-alias ptr_args` tells the compiler to ignore the possibility of aliasing in both cases. It vectorizes both loops, even though vectorizing the second loop yields incorrect results.

Fortunately, CONVEX C provides a way of telling the compiler that pointers `a` and `b` do not overlap without using the `-alias ptr_args` option. The `restrict` qualifier, used in a pointer type declaration, tells the compiler that the pointer provides exclusive access to any data it points to. In other words, a `restrict` pointer cannot be aliased to any other pointer. (It is up to the programmer, not the compiler, to ensure this.)

In this case, you can use the `restrict` qualifier to declare `a` and `b`, but not `c` or `d`:

```
#include <restrict.h>

void foo(int * restrict a,int * restrict b,
  int *c,int *d)
{
  int i;
  for (i=0; i<1000; i++)
    a[i] += b[i];
  for (i=1; i<1000; i++)
    c[i] += d[i-1];
}c
```

The compiler knows that `restrict` pointers cannot be aliased, so it vectorizes the first loop. It still recognizes the potential alias due to unrestricted pointers `c` and `d` and does not vectorize the second loop.

You can also use the `restrict` qualifier with array parameters. The `restrict` qualifier appears inside the array brackets before the first (leftmost) dimension of the array. If you omit the first dimension of the array, the `restrict` qualifier appears in place of the first dimension.

```
#include <restrict.h>

void foo(int a[restrict],int b[restrict 1000],
    int c[1000],int d[1000])
{
  int i;
  for (i=0; i<1000; i++)
    a[i]+= b[i];
  for (i=1; i<1000; i++)
    c[i]+= d[i-1];
}
```

The following example shows how to use `restrict` with arrays of one and two dimensions.

```
#include <restrict.h>
void foo(float a[restrict 50],
    float b[restrict 50][50])
{
  int i, j;
  for (i=0; i<50; i++ )
    for (j=0; j<50; j++)
      b[i][j] = a[j];
}
```

If you omit the first dimension of a multidimensional array, the use of `restrict` looks like this:

```
#include <restrict.h>
void foo(float a[restrict 50],
    float b[restrict][50])
{
  int i, j;
  for (i=0; i<50; i++ )
    for (j=0; j<50; j++)
      b[i][j] = a[j];
}
```

Be careful not to alias `restrict` pointers. Aliasing a `restrict` pointer can destroy the validity of the `restrict` qualifier and can lead to incorrect vectorization. The following code misuses a `restrict`-qualified pointer:

```
#include <restrict.h>
int a[1000];
void solve (int * restrict p)
{
 int i;
 for (i=1; i<1000; i++)
   p[i]=a[i-1];
 }
 ...
solve(a);
```

The compiler does not detect misuses of the `restrict` qualifier. If you use it improperly, incorrect vectorization can result.

## Returning pointer values

Functions that return pointer values can lead to aliasing that prevents vectorization. In the following code, `myalloc` returns a pointer value that is assigned to `b`. This assignment results in a potential alias between `a` and `b`, preventing the compiler from vectorizing the loop.

```
void foo(float *a, int n)
{
  int i;
  float *b;
  b= myalloc(n);
  for (i=0; i<50; i++ )
    b[i] = a[i];
}
```

To vectorize this loop, you must tell the compiler that `myalloc` returns a unique pointer value on every call. The `restrict` qualifier has no effect when used in the declaration of a pointer return value:

```
float * restrict myalloc(int n)
{
  ...
}
```

Instead of `restrict`, use the `returns_unique_pointer` pragma after the function declaration:

```
extern float *myalloc();
#pragma _CNX returns_unique_pointer(myalloc)
...
float *foo(float *a, int n)
{
  int i;float *b;
  b=myalloc(n);
  for ( i=0; i<n; i++ )
    b[i] = a[i];
  return b;
}
```

You do not normally need to use the `restrict` qualifier on local pointers, such as `b` in this example, although you can do so. Pointer tracking handles unrestricted local pointers quite well.

You do not need to use `returns_unique_pointer` on a call to `malloc` or `calloc`. The CONVEX C compiler automatically treats calls to these functions as though you had used the `returns_unique_pointer` pragma on each of them. The `-nptr` option, which turns off pointer tracking, also turns off this special treatment of `malloc` and `calloc`.

## Restricting `extern` pointers

You can use the `restrict` qualifier to declare `extern` pointers. The `restrict` qualifier imposes a severe restriction on the use of an `extern` pointer. Once assigned, a `restrict`-qualified `extern` pointer can only point to one data object for the duration of a program. It must not be reassigned.

You can use a `restrict`-qualified `extern` pointer, for example, to hold a pointer to an object returned by a function such as `malloc`.

```
float * restrict b;

foo()
{ int n;
  ...
  b= (float *) malloc(n);
}
```

## Preventing aliases

The combination of the ANSI-C aliasing algorithm and pointer tracking allows the compiler to detect genuine aliases while minimizing the number of false alarms. If you follow the advice given in the previous sections, the compiler can automatically vectorize most loops. It cannot, however, vectorize loops that contain genuine aliases and are unsafe to vectorize. You can minimize the number of aliases, though, by writing your code according to the guidelines in this section.

Reduce or localize pointer uses. Reduce the use of structure pointers and address operators as well. Avoid code such as this:

```
divarrays()
{
  double *a, *b;
  int i;
  ...
  for (i=0; i<1000; i++ ) /* no vectorization */
    a[i] = a[i] / b[i];
}
```

Instead, write code like this:

```
divarrays()
{
  double arra[1000],arrb[1000];
  int i;
  ...
  for (i=0; i<1000; i++ ) /* vectorizes */
    arra[i] = arra[i] / arrb[i];
}
```

If you have a loop that does not vectorize because of an alias, but you are sure the alias cannot cause a recurrence, you can apply a `no_recurrence` pragma to the loop.

```
void foo(float *a, float *b, float c[], int n)
{
  int i;
  a=&c[0];
  b=&c[n];
# pragma _CNX no_recurrence
  for (i=0; i<n; i++ )
    a[i]=b[i];
}
```

The `no_recurrence` pragma tells the compiler to ignore any apparent recurrence in the loop, whether from aliasing or some other cause. Do not use this pragma unless you are sure that no recurrence exists. In this example, `a` and `b` become aliased when they assigned addresses based on `&c`. Because of the way the array sections are used, however, no recurrence exists. The `no_recurrence` pragma allows the compiler to vectorize the loop.

You can also use the `no_recurrence` pragma to vectorize a loop with an apparent recurrence. Again, you must take responsibility for ensuring that no recurrence exists. If you use the `no_recurrence` pragma on a loop where a recurrence exists, incorrect optimizations can result.

# Limits of optimization

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 levels

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 C source. Optimizations done on source that violates certain ANSI C standard rules can cause the compiler to generate incorrect code.**

## Incorrect results

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

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

## Erroneous code

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

### Hidden aliases

Hidden aliasing is the most common cause of code whose results change at higher optimization levels. In C, aliasing can occur through the use of a pointer ( * ), address ( & ), or structure pointer ( -> ) operator. Aliasing also occurs when you use array parameters, which C treats as pointers. Aliasing can prevent the compiler from safely vectorizing a loop. If you use the `-alias array_args` option, aliasing of array parameters is hidden. This can cause the compiler to vectorize or parallelize a loop when it is not safe to do so.

The program shown below contains an alias. The formal parameters `arr1` and `arr2` are assigned the same actual parameter, `arrk`, at runtime. This causes a recurrence in the `i` loop, which should prevent vectorization, but the compiler ignores it when `-alias array_args` is used.

```
#include <stdio.h>

void confused(int arr1[], int arr2[])
{
  int i,j;

  for ( i = 1; i < 128; i++ )
    arr1[i] = arr1[i] + arr2[i - 1];

  for ( j = 0;j < 100;j += 10 ) {
    (void)printf("arr1[%2d]= %2d ",
        j, arr1[j]);
    (void)printf("arr2[%2d]= %2d\n",
        j, arr2[j]);
  }
}
main()
{
  int i, arrk[128];

  for ( i = 0; i < 128; i++ )
    arrk[i] = 1;

  confused(arrk, arrk);
}
```

When this program is compiled at -O1 and executed, you see the results shown here:

```
% cc -O1 alias.c
% a.out
arr1[ 0]= 1  arr2[ 0]= 1
arr1[10]= 11 arr2[10]= 11
arr1[20]= 21 arr2[20]= 21
arr1[30]= 31 arr2[30]= 31
arr1[40]= 41 arr2[40]= 41
arr1[50]= 51 arr2[50]= 51
arr1[60]= 61 arr2[60]= 61
arr1[70]= 71 arr2[70]= 71
arr1[80]= 81 arr2[80]= 81
arr1[90]= 91 arr2[90]= 91
```

If you compile the program at -O2 without the -alias array_args option, no vectorization occurs and you get the same results. If you compile the program at -O2 with the -alias array_args option, however, you get the output shown below:

```
% cc -O2 -alias array_args alias.c
% a.out
arr1[ 0]= 1 arr2[ 0]= 1
arr1[10]= 2 arr2[10]= 2
arr1[20]= 2 arr2[20]= 2
arr1[30]= 2 arr2[30]= 2
arr1[40]= 2 arr2[40]= 2
arr1[50]= 2 arr2[50]= 2
arr1[60]= 2 arr2[60]= 2
arr1[70]= 2 arr2[70]= 2
arr1[80]= 2 arr2[80]= 2
arr1[90]= 2 arr2[90]= 2
```

The function `confused` expects to receive two arrays as parameters, but the main function has passed it the same array twice. This produces a dependency in the `i` loop: the calculation of the *n*th element depends on the previous calculation of the (*n*-1)th element. The `-alias array_args` option assures the compiler that the two arrays are distinct. The compiler ignores the possible dependency and vectorizes the loop. The vectorized loop adds the (*n*-1)th element of the loop to the *n*th element, just as the scalar version did, but does all of the adds simultaneously, using the original value of the (*n*-1)th element instead of the calculated value. As a result, the calculated values for each *n*th element are changed.

### Invalid subscripts

The value of a subscript expression must be greater than or equal to the lower bound of the array, which is zero in C, and less than or equal to the upper bound, which is the size of the array minus one.

Subscripts that go out of bounds are a frequent cause of answers that vary between optimization levels and programs that abort and dump core.

## Floating-point imprecision

The compiler applies normal arithmetic rules to real numbers. It assumes that two arithmetically equivalent expressions produce the same numerical result.

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, 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.

At optimization level `-O2`, round-off problems are caused by differences between the rounding algorithms used in the vector and scalar processing units. An expression evaluated to 32-bit precision in the scalar unit may yield a positive or negative number very close to zero, while the same expression evaluated to 32 bits in the vector unit yields zero. When evaluated to 64-bit precision, the answers may agree more closely, but 32-bit and 64-bit answers usually differ significantly.

Vector reductions can cause problems with floating-point precision. 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 differ greatly in magnitude. If this causes a problem, run the reduction loop as a `scalar` loop. Or try modifying your algorithm so that the smallest magnitude values are combined first.

## Misused pragmas and options

Misused pragmas 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, refer to "Recurrence" in Chapter 3.

In the following example, the `main` function in initializes `arra`, calls `calc`, and displays the new array values. In the `calc` function, the apparent recurrence on `a[i+n]` prevents the compiler from vectorizing the `i` loop.

```
#include <stdio.h>
#define SIZE 1024
float arra[SIZE], arrb[SIZE];

void calc(int n)
{
  int i;

  for ( i = 0; i < SIZE; i++ )
    arra[i] = arra[i + n] + arrb[i];
}
main()
{
  int j;

  for ( j = 0; j < SIZE; j++ )
    arra[j] = j;
  calc(1);
  for ( j = 0; j < SIZE; j++ )
    (void)printf("%d %f\n", j, arra[j]);
}
```

Because you know the value of n is 1, you can use the `no_recurrence` pragma, as shown below. This pragma tells the compiler to ignore the apparent recurrence and vectorize the `i` loop.

```
void calc(int n)
{
  int i;

# pragma _CNX no_recurrence
  for ( i = 0; i < 1024; i++ )
    arra[i] = arra[i + n] + arrb[i];
}
```

That correct results have been obtained with vectorization does not imply that correct results can be obtained with parallelization. Using the `force_parallel` pragma on this loop, as shown below, 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.

```
void calc(int n)
{
  int i;

# pragma _CNX force_parallel
  for ( i = 0; i < 1024; i++ )
    arra[i] = arra[i + n] + arrb[i];
}
```

## Compiler limitations

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

- Reductions
- Ambiguous evaluation order
- Iterations with a step size of zero
- Nondeterminism of parallel execution
- Conditional vectorization
- Replaceable loop test variables

### Reductions

Reductions, discussed in Chapter 3, are a special class of recurrence that the compiler can safely vectorize. An apparent recurrence can prevent the compiler from vectorizing a loop with a reduction. The loop shown below does not vectorize because of an apparent dependency between the reference to `arra[i]` and the assignment to `arra[arrx[j]]`.

```
for ( i=0; i<5; i++ )
  for ( j=0; j<5; j++ ) {
    arra[i] += arrb[j] * arrc[j];
    arra[arrx[j]] = arrb[j] + arrc[j];
  }
```

Placing a `no_recurrence` pragma before the `j` loop causes the compiler to ignore the reduction on `arra[i]` as well as the apparent recurrence. The compiler vectorizes the loop normally, which produces incorrect answers.

To solve this problem, distribute the `j` loop, isolating the reduction from the other statements, as shown here:

```
for ( i=0; i<5; i++ )
  for ( j=0; j<5; j++ )
    arra[i] += arrb[j] * arrc[j];

for ( j=0; j<5; j++ )
  arra[arrx[j]] = arrb[j] + arrc[j];
```

This code removes the apparent recurrence, and both loops vectorize.

This problem occurs only when a reduction and an apparent recurrence involve the same variable. When a reduction and an apparent recurrence involve different variables, as in the following example, the compiler can handle both:

```
for ( i=0; i<5; i++ )
# pragma _CNX no_recurrence
  for ( j=0; j<5; j++ ){
    arra[i] += arrb[j] * arrc[j];
    arrd[arry[j]] = arrd[i] + arrb[j];
  }
```

### 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.

**Note**

**The `-parens` command line option can affect the way parentheses are treated in all compatibility modes. If you specify `-parens explicit`, parentheses are honored regardless of the compilation mode. If you specify `-parens ignore`, parentheses are ignored and the compiler can reorder a floating-point expression. This is the default in the backward-compatible and extended modes of the compiler. If you specify `-parens implicit`, the compiler honors all parentheses, grammar, and associativity rules for floating-point expressions, and no reordering can be performed. This is the default for the standard and strict compatibility modes.**

**These options affect only floating-point expressions. Integer expressions can always be reordered in any compilation mode.**

### Iterating by zero

If the compiler vectorizes a loop that iterates a variable by zero on each iteration, 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 it detects that the variable has been set to zero, the compiler does not vectorize the loop. If the compiler cannot detect the assignment, however, the previously described symptoms occur. The following example shows two loops that iterate by zero:

```
float a[100],b[100],c[100];

void sub1(int zr)
{
  int i,j = 1;

  for( i=0; i<100; i++ ){
    b[i] = a[j];
    a[j] = c[i];
    j += zr;
  }
  i = 0;
  while( i<100 ){
    a[i] = b[i];
    i += zr;
  }
}

main()
{
  sub1(0);
}
```

Because `zr` is an argument passed to `sub1`, the compiler does not detect that `zr` has been set to zero. Both loops vectorize at `-O2`. The first loop runs, even when vectorized, but produces wrong answers. The other loop runs infinitely when compiled at `-O1`, and causes 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 may appear intermittently. Unless you are sure that no loop-carried dependency exists, it is safer to let the compiler choose which loops to parallelize.

### Conditional vectorization

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

```
int a[10000],b[10000],c[10];

main()
{
   int i;
   for(i=0; i<10; i++)
     a[i] = -5;
   for(i=10; i<10000; i++)
     a[i] = 0;
   for(i=0; i<10000; i++)
     if( a[i]<0 )
       b[i] = a[i] + c[i];
}
```

In the example, c is subscripted from 1 to 10000, which can cause a memory-reference error.

### Test replacement

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

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

```
#include <stdio.h>
void dump(int);

void foo(int n)
{
   int ires,ipack,icountit;
   icountit = 0;
   for(ires=38; ires <= n; ++ires) {
      ipack = ((ires*1024+38)*64)*64;
      dump(ipack);
      ++icountit;
   }
   (void)printf("%d\n",icountit);
}

main()
{
   foo(970);
}

void dump(int dummy)
{
}
```

When compiled at optimization level `-O1` or higher, the compiler replaces references to `ires`, the original induction variable, with suitably equivalent references to `ipack`, because `ipack` is referenced more often in the loop. The value by which `ipack` increases on each iteration (1024*64*64, or $2^{22}$) becomes the loop's *stride*. The number of times the loop executes is called the iteration count (`n` in the example), and the initial value of the induction variable is the start value.

Test replacement, a standard optimization performed at levels -O1 and above, normally does not cause problems. However, when the loop stride is large, as in the example in Figure 9-12, a large iteration count can cause the loop limit value (stride*iteration+start) to overflow its memory location. In the example, the induction variable is a default integer (four bytes), which occupies 32 bits in memory. That means if stride*iteration+start ($N*2^{22}+1$) is greater than $2^{31}-1$, the value overflows into the sign bit and the computer treats it as a negative number. (If the stride value is negative, the absolute value of stride*iteration+start must be not exceed $2^{32}$.) When a loop has a positive stride and the iteration count overflows its memory location, the loop executes only once because the limit is now negative and the termination test fails.

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

Because we know that the largest allowable value for stride*iteration+start is $2^{31}-1$ and the start value for the loop in Figure 9-12 loop is 38, the maximum iteration count for the loop using the equation (stride*iteration+start)/stride. To solve this for iteration, remove the start value by subtracting 38 and divide by stride. This gives $(2^{31}-1-38)/2^{22}=511.999$. Because these numbers are integers, passing the function a value larger than 511 will cause an error.

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

## Slower code

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

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

### Misused pragmas

The `synch_parallel` pragma 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 it can 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 machine the program was compiled on or the number of CPUs specified by the `-ep` option. The `vstrip` and `pstrip` pragmas 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` pragma can prevent the compiler from vectorizing such loops. `select` tells the compiler to generate multiple versions of a loop and code to allow dynamic (runtime) selection of the best version. Using `select`, 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 executed more frequently than the larger clause.

A short vector length (small iteration count) makes a loop containing complicated conditionals less efficient.

To increase efficiency, simplify conditionals, remove them from the loop, or use the `scalar` pragma to prevent loops from being vectorized.

# The -uo option

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

- Simple strength reductions (-O1 and higher)
- Code motion (-O1 and higher)
- Pattern matching (-O2 and higher)
- Elimination of type conversions (-O1 and higher)

## 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 `x*(1/c)` 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 all paths to 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 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.

The following example shows such a loop.

```
int a[100];

int min()
{
  int i, imin;

  for( imin=0, i=1; i<100; i++ )
    if(a[i] < a[imin])
      imin = i;

  return( imin );
}
```

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

```
float a[100];
float b[100];

int foo(float z)
{
  int i, zi, max=0;

  for( i=1; i<100; i++ )
    a[i] = a[i-1] + b[i];

  for( i=1; i<100; i++ )
    if( a[i] > a[max] )
      max = i;
}
```

## Conversion elimination

When you use the -uo option, the compiler eliminates costly type conversions by creating real induction variables. Consider the following loop:

**Original Loop**

```
float a[100];

int foo()
{
  int i;

  for( i=0; i<100; i++ )
    a[i] = i;
}
```

Here is the optimized loop:

**Optimized Loop**

```
float a[100];

int foo()
{
  int i;
  float real_i;

  for( i=0, real_i=0.0; i<100; i++,real_i++ )
    a[i] = real_i;
}
```

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

# CONVEX C intrinsics

# B

This chapter defines intrinsic functions and intrinsic instructions, problems that they cause, and how to work around the problems.

## What are intrinsics?

CONVEX C has two types of intrinsics: instructions and functions. Intrinsic instructions are part of the CONVEX instruction set. For example, `sqrt` is an intrinsic instruction in the CONVEX C3 Series architecture.

Intrinsic functions eliminate function-call overhead. They execute faster than ordinary functions and do not inhibit the vectorization of loops.

Some intrinsic functions have drawbacks. The math intrinsic functions do not modify `errno` when an error occurs. The compiler does not recognize a dependency between `errno` and the math intrinsic functions.

Intrinsic functions, called from C, can use intrinsic instructions. In extended compatibility mode, the compiler uses intrinsic functions by default. To access these functions in the strict and standard compatibility modes, use `-U__NO_INLINE_MATH` on the `cc` command line. To use intrinsic functions in the backward-compatible mode, use `-D__INLINE_MATH`.

The following C program calls `sqrt`, an ANSI C function:

```
#include <math.h>
#include <stdio.h>
int main()
{
  int x = 4;
  double y;

  y = sqrt(x);
  (void) printf("%f n", y );
  return(0);
}
```

If this program is compiled and executed on a C1 machine, an intrinsic `sqrt` function is used with the program. But if it is compiled and executed on a C3 machine, the intrinsic instruction is used. This is because the C1 instruction set does not have a `sqrt` instruction; the `sqrt` function must be implemented using a software square root algorithm.

You can see which functions are implemented as intrinsics by searching the include files for the `__NO_INLINE` macro. Intrinsic functions are implemented as function-like macros defined with CONVEX reserved function names. Some intrinsics, such as `abs()` and `labs()` are defined in `<stdlib.h>`. Most are defined in `<math.h>`.

The following example shows macro definitions that call math intrinsic functions:

```
# if   !defined(__NO_INLINE) && \
       !defined(__NO_INLINE_MATH)
 /* fast implementations of the routines
    defined by ANSI C */
#    define acos(x) _mth$d_acos((double)(x))
#    define asin(x) _mth$d_asin((double)(x))
     .
     .
     .
# endif
```

In this example, the `acos` function is a function-like macro that calls `_ mth$d_acos`. Do not call functions that define intrinsic functions directly in your programs. These function names are subject to change.

## Intrinsic function behavior

Intrinsic functions may not modify the `errno` variable when an error occurs. When an intrinsic instruction detects an error, it does not generate a signal because when a C program begins execution, the bit in the program-status-word register that controls the generation of intrinsic error signals is not set.

When compiled in the extended compatibility mode and executed on a C3 machine, the following program produces an incorrect result:

```
#include <math.h>
#include <stdio.h>
#include <errno.h>
int main()
{

  int x = -4;
  double y;

  errno = 0;
  y = sqrt(x);
  if( errno == EDOM )
    (void) printf("domain error n");
  else
    (void) printf("%f n", y );

  return(0);
}
```

When you compile this program using the `-tm c1` command line option, and executed it on a C1 computer, the domain error is caught. (The implementation of the `sqrt` intrinsic function on a C1 machine modifies the `errno` variable when an error occurs.)

## errno and optimization

Another problem associated with the use of intrinsic functions is that at levels of optimization higher than -no, the compiler removes redundant code or replaces slow code with faster pieces of code. This can cause a problem with some intrinsic functions.

In the code below, the compiler removes the conditional statement because it does not realize that the acos function can modify errno.

```
...
errno = 0;
a = acos(x);
if(errno == EDOM) {
   ...
}
...
```

At optimization level -O2, the following code is optimized to this statement:

```
a = acos(x);
```

## How to disable intrinsics

Only the math intrinsic functions are not accessible by default in the strict and standard compatibility modes. The reason for this is that ANSI C requires `errno` to be modified when certain function errors occur. Similarly, to maintain compatibility with previous compilers, the math intrinsic functions are not accessible by default in the backward-compatible compatibility mode.

To prevent your program from using all intrinsic functions, you can define the `__NO_INLINE` macro on the command line using the `-D` command line option. However, this might be too stringent. There are several different types of intrinsic functions, and you may need to disable only one type.

Here are the eight types of intrinsic functions:

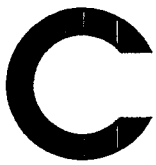
- `__NO_INLINE_BINT`
- `__NO_INLINE_CTYPE`
- `__NO_INLINE_MATH`
- `__NO_INLINE_SIGNAL`
- `__NO_INLINE_STDIO`
- `__NO_INLINE_STDLIB`
- `__NO_INLINE_STRING`
- `__NO_INLINE_TIME`

Each of these macros is named after the include file in which it can be found.

For example, the stdio.h include file, which contains some I/O functions, declares function-like macros defined with intrinsic functions. You can disable these intrinsic functions by including the `-D__NO_INLINE_STDIO` option on the command line. You might want to use the intrinsic functions after you have completely debugged your program.

Another way to avoid the `errno` problem is to set the Intrinsic Error Enable bit of the Processor Status Word when your program is started. You must include a signal handler that determines what caused the signal and then take appropriate actions. For more information on the PSW and the bits associated with intrinsic instruction, see the *CONVEX Architecture Reference Manual (C Series)*.

# Compiler pragmas

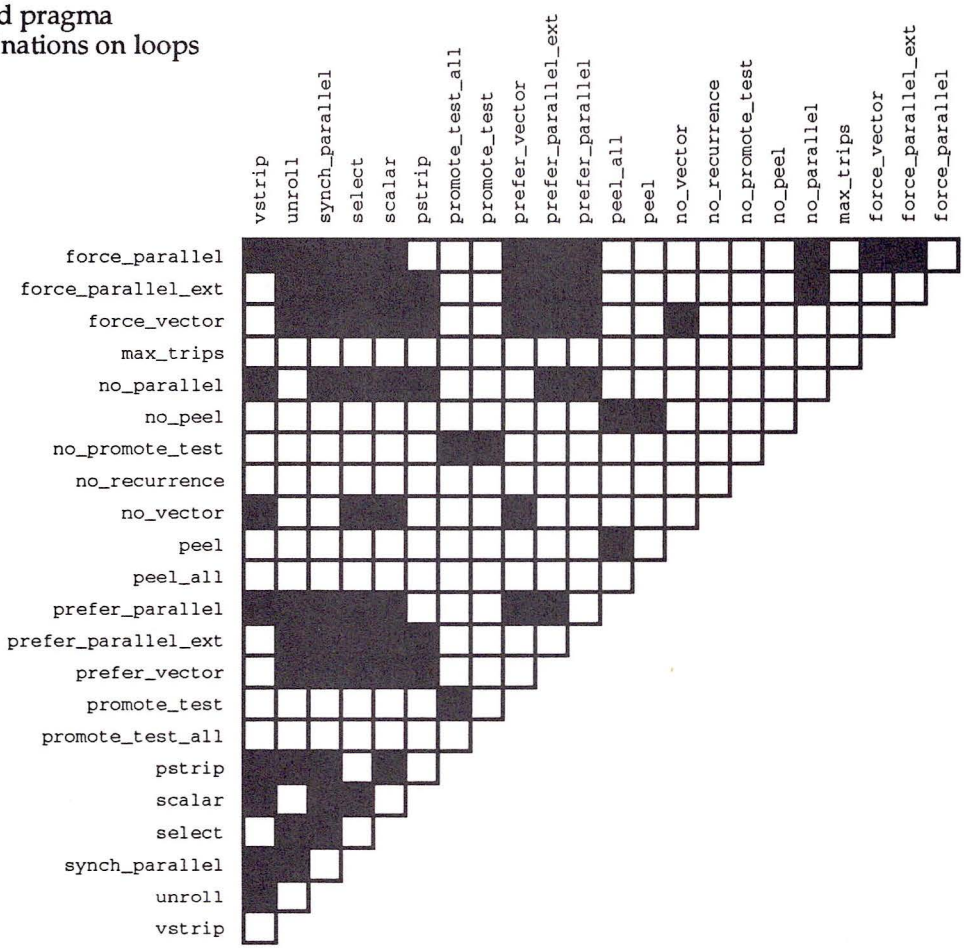


This appendix describes CONVEX C compiler pragmas that affect optimization. These are the CONVEX C optimization pragmas:

- `begin_tasks`
- `end_tasks`
- `force_parallel`
- `force_parallel_ext`
- `force_vector`
- `max_trips`
- `next_task`
- `no_parallel`
- `no_peel`
- `no_promote_test`
- `no_recurrence`
- `no_side_effects`
- `no_vector`
- `peel`
- `peel_all`
- `prefer_parallel`
- `prefer_parallel_ext`
- `prefer_vector`
- `promote_test`
- `promote_test_all`
- `pstrip`
- `returns_unique_pointer`
- `scalar`
- `select`
- `synch_parallel`
- `unroll`
- `vstrip`

Certain pragma combinations, when used on the same loop, are invalid and cause the compiler to issue a warning. The black squares in Figure 8 denote invalid pragma combinations.

**Figure 8**
Invalid pragma combinations on loops




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

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

## 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` pragma identifies a sequence of tasks for independent, parallel execution. The sequence of tasks ends with `end_tasks`. `next_task` precedes each task except the first.

The following code shows the use of the tasking pragmas.

```
#pragma _CNX begin_tasks
  <statement>
  ...
#pragma _CNX next_task
  <statement>
  ...
#pragma _CNX next_task
  <statement>
  ...
#pragma _CNX end_tasks
```

You can specify a maximum of 255 tasks between a `begin_tasks` and an `end_tasks` pragma.

### force_parallel

The `force_parallel` pragma is effective only if you specify the `-O3` compiler option. The pragma tells the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. You can use this pragma 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` pragma.

This pragma does not allow the compiler to interchange or distribute loops outside the `force_parallel` loop for vectorization. To enable those optimizations, use `force_parallel_ext`.

**Caution**

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

If you use this pragma with `scalar` or `no_parallel`, a warning is issued. Also, a warning is issued when you use `force_parallel` and another parallelizing pragma in the same loop nest.

An example of how to use `force_parallel` appears below:

```
#pragma _CNX force_parallel
for( i=0; i<n; i++ )
  sub(a, i);
```

Remember, you must compile `sub` with `-re` for this loop to execute correctly.

### force_parallel_ext

The force_parallel_ext pragma is effective only if you specify the -O3 compiler option. This pragma forces the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. You can use this pragma 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 pragma causes the compiler to ignore any apparent dependencies between iterations. When you use this pragma on a loop, you may not get correct results. Check answers generated by the parallelized code.**

If you use this pragma with scalar or no_recurrence a warning is issued. Also, an error occurs when you use force_parallel_ext and another parallelizing pragma in the same loop nest.

### force_vector

The force_vector pragma 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 pragma and get incorrect answers because the pragma causes the compiler to ignore dependencies.

Use this pragma 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 pragma causes the compiler to ignore any apparent dependencies between iterations. When you use this pragma 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. A warning is also issued when you try to use force_vector and another vectorizing pragma in the same loop nest.

### max_trips

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

```
#pragma _CNX max_trips(<n>)
```

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

### no_parallel

The no_parallel pragma tells the compiler not to parallelize the loop immediately following the pragma. The pragma does not prevent vectorization of the loop.

If no_parallel and no_vector both precede the same loop, that loop will run in scalar mode.

### no_peel

The no_peel pragma stops the compiler from performing loop peeling, an optimization that can increase code size and compile time.

### no_promote_test

The no_promote_test pragma stops the compiler from performing test promotion, an optimization that can increase code size and compile time.

## no_recurrence

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

no_recurrence does not affect recurrences caused by a nested for loop. You can, however, use the pragma 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 the following code, if j is positive, no recurrence exists.

```
#pragma _CNX no_recurrence
for( i=0; i<n; i++ )
  a[i] = a[i + j];
```

The compiler always accepts a no_recurrence pragma on an apparent recurrence involving an array element; the compiler always ignores a no_recurrence pragma on an actual 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, refer to Chapter 9, "Limits of optimization."

## no_side_effects

The `no_side_effects` pragma tells the compiler that the specified function does not modify the value of an argument or global variable, perform input or output, or call another subprogram.

The format of this pragma is

```
#pragma _CNX no_side_effects(func[,func])
```

The argument *func* specifies a user-defined function.

This pragma 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, and rarely occur in the original source text.

Place the pragma before the call to the named function but after its declaration. If the function has not been declared, its use in the pragma implies an `extern int func()` declaration.

```
int f1(int, int);
#pragma _CNX no_side_effects(f1)
x = y * f1(5, z) - w;
```

A function call with no side effects is invariant with respect to a loop in these cases:

- When its arguments do not vary within the loop and the function call can be moved out of the loop
- When it does not modify a nonlocal variable
- When it does not perform I/O

### no_vector

The no_vector pragma tells the compiler not to vectorize the loop immediately following the pragma. This pragma does not prevent parallelization.

If no_parallel and no_vector both precede the same loop, that loop will run in scalar mode.

### peel

The peel pragma increases the compiler's internal limit on boundary-value peeling. Peeling replicates code and can increase the size of your executable.

### peel_all

The peel_all pragma increases the compiler's internal limit on boundary-value peeling to the maximum possible value. Peeling replicates code and can increase the size of your executable.

### prefer_parallel

The prefer_parallel pragma tells the compiler to parallelize the loop immediately following the pragma 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 pragma prevents the compiler from interchanging and distributing loops outside the prefer_parallel loop for vectorization, whereas prefer_parallel_ext does not.

### prefer_parallel_ext

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

This pragma allows the compiler to interchange loops outside the prefer_parallel_ext loop 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` pragma tells the compiler to vectorize the loop immediately following the pragma 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 tries to vectorize the interchanged loop.

### promote_test

The `promote_test` pragma increases the compiler's internal limit on test promotion. Promoting tests can increase your compile time and the size of your executable.

### promote_test_all

The `promote_test_all` pragma increases the compiler's internal limit on test promotion to the maximum possible value. Promoting tests can increase your compile time and the size of your executable.

## pstrip

The pstrip pragma tells the compiler to strip mine the parallel loop immediately following the pragma. The compiler strip mines the loop according to the strip-mine length you specify. You cannot use pstrip with vector loops.

The format of this pragma is

```
#pragma _CNX pstrip(<integer_constant>)
```

where <integer_constant> specifies the strip-mine length.

The default action of parallel strip mining combines loop iterations into groups of n/(2*ep), where n is the actual loop trip count, and ep is the number of processors (specified with the -ep compiler option) when the number of processors is greater than one. If the number of expected processors is one, the number of loop iterations in a group is always one. To override the default, use pstrip to specify with <integer_constant> the number of iterations to group.

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.

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.

## returns_unique_pointer

The returns_unique_pointer tells the compiler that a function returns a unique pointer every time it is called. The compiler makes optimization decisions based on this information. If you use this pragma incorrectly, it can result in wrong optimizations and incorrect results.

## scalar

The `scalar` pragma prevents the compile from vectorizing, parallelizing, distributing, or interchanging a loop.

Use the `scalar` pragma when you require numerical results identical to those of the scalar loop. Vector operations, such as floating-point sum and product reduction, can give slightly different answers from the scalar equivalents.

You can also use this pragma to prevent the compiler from interchanging or distributing when you believe its iteration count does not justify these optimizations.This can happen when the compiler cannot determine count of a loop, such as the two loops in the following example. The compiler does not know the size of `n` or `m`, so it interchanges the `i` and `j` loops to optimize array accesses. The `scalar` pragma prevents the compiler from interchanging and vectorizing the `i` loop, whose iterations count does not warrant the overhead of vectorization.

```
float a[1001][2], b[1001][2], c[1001][2];

int sum(int n, int m)
{
  int i,j;

# pragma _CNX scalar
  for( i=0; i<n; i++ ) /* n = 2 */
    for( j=0; j<m; j++ ) /* m =1000 */
      a[j][i] = b[j][i] + c[j][i];
}
```

In the following code, neither iteration count warrants vectorizing the loop:

```
float a[2][2], b[2][2], c[2][2];

int sum(int n, int m)
{
  int i,j;

# pragma _CNX scalar
  for( i=0; i<n; i++ ) /* n = 2 */
#   pragma _CNX scalar
    for( j=0; j<m; j++ ) /* m = 2 */
      a[i][j] = b[i][j] + c[i][j];
}
```

## select

The `select` pragma tells the compiler to generate multiple versions of a loop that 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 pragma is

```
#pragma _CNX 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 pragma. 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 replace a trip count with an asterisk, 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 pragma, 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:

```
#pragma _CNX select(10,4,200)
```

The loop runs scalar if the actual trip count is 1 to 4, and parallel if the trip count is 5 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

```
#pragma _CNX select(*,*,*)
```

causes the loop to run in scalar mode.

## synch_parallel

The `synch_parallel` pragma is effective only if you specify the `-O3` compiler option. This pragma 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 pragmas, 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` pragma, particularly if all the dependencies are in seldom executed branches.

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

```
float a[100], b[100], d[100], e[100], f[100];

int calc()
{
  int i;

# pragma _CNX synch_parallel
  for( i=0; i<32; i++ )
    if( a[i] < 0 ){
      a[i] = a[i] + b[i];
      d[i] = e[i] * f[i];
    }
}
```

## unroll

The `unroll` pragma takes effect only at `-O2` and `-O3`. `unroll` reduces loop overhead by replicating the body of a loop. The compiler cannot completely unroll simple loops with iteration counts less than five. It can partially unroll loops with iteration counts greater than five.

The compiler cannot completely unroll loops with internal branches or unknown iteration counts. It cannot completely unroll a loop that contains another loop. It does not unroll a vectorized loop.

### vstrip

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

The format of this pragma is shown below:

```
#pragma _CNX vstrip(integer_constant)
```

where *integer_constant* specifies the strip-mine length.

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.

`vstrip` overrides the compiler default and specifies a different strip-mine length. A shorter strip optimizes the iterations of the strip-mine loop so that the loop can be effectively parallelized.

To determine the approximate maximum strip-mine length when the number of expected processors (specified with the `-ep` option) is more than one, the compiler uses the formula

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

where `n` is the actual loop trip count.

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

# Vector operations

D

This appendix describes the vector instruction set on CONVEX C Series computers. These descriptions can help you create efficient code. You do not need to know assembly language to read and understand this chapter. The assembly language examples are fragments; they cannot be assembled. For more detailed information about vector operations and hardware, refer to the *CONVEX Architecture Reference Manual (C Series).*

## 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-accumulator register (V), also known as a vector register, is used to store arrays of operands. C100 and C200 Series machines have eight vector registers. Each vector register can hold up to 128 64-bit elements, which can be integer or floating-point data. Data must be of uniform size and precision.

### Vector-length register

C100 and C200 Series machines have one vector-length (VL) register. The value in the VL register is the number of elements used in subsequent vector operations.

## Vector-stride register

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

## Vector-merge register

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

The VM register is often used for these operations:

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

## CONVEX vector architecture

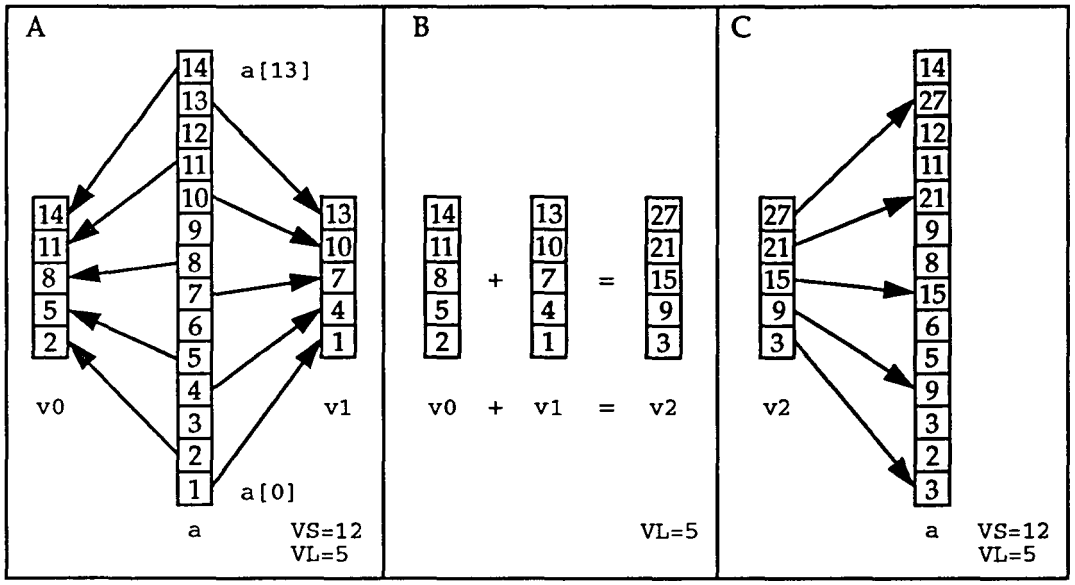
To see how the vector hardware works, consider the following vector operation:

```
int a[14] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14};

main()
{
   int i;

   for(i=0; i<14; i+=3 )
     a[i] = a[i+1] + a[i];
}
```

The code modifies every third element of the array and uses the VS, VL, and V registers.

Figure 9 shows the vector operations on array a.

**Figure 9**
Vector operations for `a[i] = a[i + 1] + a[i];`




In panel A, the CPU sets the vector-stride register to 12 (the number of bytes between elements of the array). The CPU has set the vector-length (VL) register controlling the operation to five. The VL register controls loading elements from array a into vector registers v0 and v1.

In panel B, the CPU adds the contents of vector registers v0 and v1 and stores the result in v2.

In panel 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 C statements are vectorized in assembly language. For a complete list of the vector instruction set, refer to the *CONVEX Architecture Reference Manual (C Series).*

### Vector load

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

Suppose the C code shown on the left in is vectorized:

| C | Assembly language |
|---|---|
| `int a[25];` | `ld.w #25,VL` |
| | `ld.w #4,VS` |
| `for(i=0; i<25; i++ )` | `ld.w a,v0` |
| `  a[i] += 4;` | |

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 4 into VS because each element in the array requires four bytes for storage and the loop's stride is one. The last statement loads the contents of array a into vector register v0.

Here is another example of vector load:

| C | Assembly language |
|---|---|
| `float b[100];` | `ld.w #5,VL` |
| | `ld.w #80,VS` |
| `for(i=0; i<100; i+=20 )` | `ld.s b,v0` |
| `  b[i] += 8.0;` | |

The assembly-language code required to load b into vector register v0 appears on the right.

VL contains 5 because only five elements of array b are modified. It is unnecessary to load all the elements of b into the vector. Similarly, VS contains 80 because each element requires four 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 here:

| C | Assembly language | |
|---|---|---|
| `int c[100];` | `ld.w` | `#5,s0` |
| | `ld.w` | `#100,VL` |
| `for(i=0; i<100; i++ )` | `ld.w` | `#4,VS` |
| `  c[i] = 5;` | `ste.w` | `s0,c` |

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 stored.

## 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.

Here is an example of vector store:

| C | Assembly language | |
|---|---|---|
| `int b[100], c[100];` | `ld.w` | `#100,VL` |
| | `ld.w` | `#4,VS` |
| `for(i=0; i<100; i++ )` | `ld.w` | `b,v0` |
| `  c[i] = b[i];` | `st.w` | `v0,c` |

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

This code shows another example of vector store:

| C | Assembly language | |
|---|---|---|
| `int b[100],c[100];` | `ld.w` | `#50,VL` |
| | `ld.w` | `#4,VS` |
| `for(i=0; i<50; i++ )` | `ld.w` | `b,v0` |
| `  c[i*2] = b[i];` | `ld.w` | `#8,VS` |
| | `st.w` | `v0,c` |

In this example, the VS register is increased to 8 because only every other element of array `c` is modified. Only every other element of array `c` is modified, and each element of array `c` requires 4 bytes for storage. So the stride of the loop is 8.

## Binary vector operators

Four binary operators used in vector arithmetic are addition, subtraction, multiplication, and division. Binary operators used for logical operations are `and`, `or`, and `xor`. The 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.

This code shows the use of the vector add operator:

| C | Assembly language |
|---|---|
| `int b[100],c[100];` | `ld.w #50,VL` |
| | `ld.w #4,VS` |
| `for(i=0; i<50; i++ )` | `ld.w b,v0` |
| `  c[i] += b[i];` | `ld.w c,v1` |
| | `add.w v1,v0,v2` |
| | `st.w v2,c` |

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

The C code shown here computes the product of a vector and a scalar:

| C | Assembly language |
|---|---|
| `int c[100];` | `ld.w #100,VL` |
| | `ld.w #8,s0` |
| `for( i=0;i<100;i++ )` | `ld.w #4,VS` |
| `  c[i] *= 8;` | `ld.w c,v0` |
| | `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`). Reduction operations are also provided to implement the bitwise operators such as `&`, `|`, and `^`.

The following example generates a sum reduction:

| C | Assembly language |
|---|---|
| `int c[100];` | `ld.w  #0,s0` |
| `int isum = 0;` | `ld.w  #100,VL` |
| | `ld.w  #4,VS` |
| `for( i=0;i<100;i++ )` | `ld.w  c,v0` |
| `  isum += c[i];` | `sum.w  v0` |
| | `st.w  s0,isum` |

During a vector reduction, a vector register (`vi`) is paired with a scalar register (`si`). In this example, `s0` is the scalar register that corresponds to `isum`, and `v0` is the vector that is reduced.

The statement

```
sum.w   v0
```

can be replaced with

```
sum.w   s0
```

Both statements produce the same result.

## Chaining

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

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

```
int i, sum=0;
int d[100], n[100], a[100];
for( i=0; i<100; i++ ){
  d[i] = n[i] * a[i];
  sum += d[i];
}
```

The assembly language for the dot-product operation is shown here:

```
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
sum.w   v0
st.w    v0,d
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. (This is only one possible assembly-language code for this operation. The way instructions chain actually depends on the specific CONVEX architecture in use.)

## 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 (VM) 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, "Vector operations under mask— C200."

Consider the vector comparison shown here:

**C**

```
int a[100],b[100];

for( i=0;i<100;i++ )
  if(a[i] <= b[i])
    ...
```

**Assembly language**

```
ld.w    #100,VL
ld.w    #4,VS
ld.w    b,v0
ld.w    a,v1
le.w    v1,v0
...
```

The arrays are loaded into vectors, and the vectors are compared.

## Vector operations under mask—C3

C2 and C3 Series computers can perform vector operations under mask. C100 Series computers can perform vector operations and mask operations, but multiple vector instructions must replace an individual vector operation under mask on the C3 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 the vector element from the operation.

There are two forms of vector operations under mask:

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

The 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 zero.

For the remaining examples of operations under mask, assume these values before each instruction is executed:

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

The statement

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

produces

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

The statement

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

produces

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

The following C code is an example of using operations under mask:

| C | Assembly language |
|---|---|
| `for( i=0;i<100;i++ )` | `ld.w  a,v0` |
| `  if(a[i] == b[i])` | `ld.w  b,v1` |
| `    c[i] = d[i];` | `eq.w  v0,v1` |
| | `ld.w  d,v0` |
| | `st.w.t  v0,c` |

Assuming the VL and VS registers are appropriately initialized, the code on the left can be vectorized with the assembly-language code on the right.

## Vector-merge register operations

The `merge`, `mask`, `compress`, and `expand` operations use the vector-merge (VM) 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 indices of the operands are used to create the result vector. For `merge`, the indices 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 C3 Series computers. This instruction uses the VM register to extract elements from one vector register and selectively place the elements in another vector register. Either zeros or ones of VM can be used by specifying the instruction's `.f` or `.t` (false or true) version, respectively. Only elements with the corresponding VM bit set (clear for `.f`) are loaded into the destination vector. Other elements in the destination vector corresponding to clear VM bits (set for `.f`) are skipped over. The `expand` instruction creates a destination vector with VL elements, including a number of elements of the source vector equal to the number of bits set (or clear) in the VM register.

### Examples

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.

The following examples below show how these instructions work:

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
```

## Vector-operation examples

This section shows examples of common vector operations. 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 operations are and, equal, not_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 following loop cannot be determined until the program is executed.

```
int a[100],b[100],c[100],d[100];

void calc()
{
  int i;

  for( i=0; i<100; i++ )
    if(a[i] == b[i])
      d[i] = c[i];
}
```

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[i] is stored in d[i-1].

## 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 `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 following code in uses indirect addressing.

```
int a[100],ia[100],b[100],ib[100];

void gather()
{
  int i;

# pragma _CNX force_vector
  for( i=0; i<100; i++ )
    a[ia[i]] = b[ib[i]] + 1;
}
```

The assembly language that performs this function is shown here:

```
ld.w    #100,VL       ;Preliminary
ld.w    #4,s0         ;address
ldea    b,a5          ;calculations
ldea    a,a1          ;"
ld.w    #1,s1         ;"
ld.w    #4,VS         ;"
ld.w    ia,v0         ;Calculating indirect
mul.w   v0,s0,v1      ; addresses
ld.w    ib,v2         ;"
mul.w   v2,s0,v0      ;"
ldvi.w  v0,v3         ;Uses a5 for addressing
add.w   v3,s1,v2      ;Computing b[ib[i]] + 1
mov     a1,a5         ;Store result in a[ia[i]]
stvi.w  v2,v1         ;Uses a5 for addressing
```

The instructions marked "Preliminary address calculations" perform the following:

1. Load 100 into VL (set the vector length).
2. Load 4 into `s0` (4 is the stride).
3. Load address of array `b` into `a5`.
4. Load address of array `a` into `a1`.
5. Load 1 into `s1` (used in the computation).
6. Load 4 into VS (sets vector stride to 4).

Compute `b[ib[i]]` + 1 requires these steps:

1. Load values of `ib` into `v2`.
2. Multiply `v2` by `s0` (which contains 4) and store the result in `v1`. Now `v1` contains addressing offsets corresponding to the subscripts `ib[i]`.
3. The `ldvi` instruction takes the contents of `a5`, and adds them to `v0`, yielding a set of addresses. Store the contents of those addresses in `v3`. Now the values of `b[ib[i]]` are in `v3`.
4. Add `v3` to `s1` (contains 1) and store the result in `v2`. Now `v2` contains `b[ib[i]]+1`.

Storing the result of `b[ib[i]]+1` in `a[ia[i]]` requires these steps:

1. Load values of array `ia` into `v0`.
2. Multiply `v0` by `s0` (which contains 4) and store the result in `v1`. Now `v1` contains addressing offsets corresponding to the subscripts `ia[i]`.
3. Move `a1` to `a5` (`a3` had address of `a`). This sets up the next instruction.
4. The `stvi` instruction takes the contents of `a5` and adds them to `v2`, yielding a set of addresses. Those are the addresses of `a[ia[i]]`. Store the contents of `v1` (that is, `b[ib[i]]+1`) at those addresses.

# Optimization report

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

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

The following sections describe the tables and explain how to interpret them.

## Loop table

The loop table lists the optimizations performed on each loop and the reasons why other possible optimization were not performed. Loop nests are reported in the order they are encountered and separated by blank lines.The loop table has six columns:

```
Line Num.
Id Num.
Iter. Var.
Reordering Transformation
New Loops
Optimizing/Special Transformation
```

### Line Num.

This column shows the source line where the loop starts. If the line number has two parts separated by a hyphen, the second part is the distributed part number (due to loop distribution).

### Id Num.

This columns contains a unique ID number for each loop. Other parts of the report use this ID number. Both loops in the original source and loops created by the compiler have loop ID numbers.

### Iter. Var.

This column shows the name of the loop iteration variable. `*VAR*` means a compiler-generated variable. `*NONE*` means the loop has no iteration variable. If the variable has two names separated by a colon, the second name is the inline-substitution instance of that variable. If it has a name followed by a colon and a number, the variable name is truncated. For the full name, see the footnote table.

## Reordering Transformation

This column indicates which reordering transformations were performed.These transformations rearrange or duplicate loops or loop nests to increase execution speed. This column contains one or more of these symbols:

| Symbol | Meaning |
|---|---|
| `scalar` | No reordering transformation |
| `FULL VECTOR` | Completely vectorized |
| *n*`% VECTOR` | Partially vectorized |
| `PARALLEL` | Parallelized |
| `PARA/VECTOR` | Vectorized with parallel stripmine |
| `Dist` | Distributed |
| `DynSel` | Dynamic selection |
| `Inter` | Interchanged |
| `Peel` | Boundary values peeled |
| `Promoted` | Test promoted |
| * | Replaced by new loop |

## New Loops

This column shows the loops created by the compiler. The numbers in this column correspond to the ID numbers in the `Id Num.` column.

## Optimizing/Special Transformation

This column shows any optimizing or special transformations performed. An optimizing transformation reduces the number or complexity of operations. A special transformation allows the compiler to vectorize or parallelize code under special circumstances. This column can contain any of these symbols:

| Symbol | Meaning |
|---|---|
| `UNROLL` | Loop unrolled |
| `Reduction` | Vectorized based on a reduction |
| `Pattern` | Vectorized based on pattern matching |
| `Synch` | Parallel loop synchronized |
| `Removed` | Loop eliminated |
| `No strip` | Not strip mined |

## Analysis table

An analysis table elaborates on the reordering, optimizing, and special transformations when needed. The analysis table contains four columns, described below.

### Line Num.

This column shows the source line where the loop begins.

### Id Num.

This column shows the ID number assigned in the loop table.

### Iter. Var.

This column shows the name of the iteration variable controlling the loop. (Some iteration variables may appear as `*VAR*` or `*NONE*`, as described in the loop-table section of this appendix.

### Analysis

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

## Test table

The test table elaborates on any test promotions or test removals. This table has four columns, described below.

### Line Num.

This column shows the line number where the `if` test begins.

### Col. Num.

This columns shows the source column number where the `if` test begins.

### Test Transformation

This column tells you whether the test was promoted or removed.

### Analysis

This column explains the transformation. This explanation includes the line number and ID number of the original loop from which the test was transformed.

## Variable-name footnote table

This table explains variable names that are truncated in the loop and array tables. This table has two columns, described below.

### Footnoted Iter. Var.

This column shows the truncated variable name and its footnote number.

### User Variable Name

This column shows the full variable name as it appears in the source code.

## Array table

The array table shows array references that prevent optimization or cause special optimizations. This table has four columns, described below.

### Line Num.

This column shows the line number where the array reference appears.

### Var. Name

This column shows the name of the array referenced.

### Optimization

This column tells you if any optimizations (hoists or sinks) were performed on the array.

### Dependencies

When a recurrence prevents optimization, this column shows you the name of variable involved, in the form *name@linenumber.*

`*CALL*@`*linenumber* means the recurrence is caused by a call.

`*MEM*@`*linenumber* means the recurrence involves a memory reference that the compiler cannot assign a variable name to.

## Examples

Consider the following matrix-multiplication code. (Line numbers are shown for reference.)

```
float a[200][201], b[200][201], c[200][201];

void example1()
{
 int i,j,k;

 for( i=0; i<200; i++ )                    /* 7*/
   for( j=0; j<200; j++ )                  /* 8*/
     {
      c[i][j] = 0.0;
      for( k=0; k<200; k++ )              /* 11*/
        c[i][j] += a[i][k] * b[k][j];
     }
}
```

At -02, the compiler optimizes the code as follows:

```
for( i=0; i<200; i++ )                    /* 7-1*/
  for( j=0; j<200; j++ )                  /* 8-1*/
    c[i][j] = 0.027;

for( i=0; i<200; i++ )                    /* 7-2*/
  for( k=0; k<200; k++ )                  /* 8-2*/
    for( j=0; j<200; j++ )               /* 11-2*/
      c[i][j] += a[i][k] * b[k][j];
```

This produces the optimization report shown on the next page.

```
          Optimization by Loop for Routine example1

Line     Id     Iter.   Reordering       New      Optimizing / Special
Num.     Num.   Var.    Transformation   Loops    Transformation
-------------------------------------------------------------------
   7       1    i       *Dist            (2-3)    No strip
   7-1     2    i        Scalar
   8-1     4    j        FULL VECTOR

   7-2     3    i        Scalar
   8-2     5    j        FULL VECTOR Inter
  11-2     6    k        Scalar

Line     Id     Iter.   Analysis
Num.     Num.   Var.
-------------------------------------------------------------------
   8-2     5    j       Interchanged to innermost

          Array References for Routine example1

Line  Var.  Optimi-   Dependencies
Num.  Name  zation
-------------------------------------------------------------------
  12  c       Sunk
  12  c       Hoist
```

The compiler assigns every loop an ID number, shown in the second column of the loop table. Some optimizations, such as loop distribution, dynamic selection, and loop peeling, create additional loops. The compiler assigns a unique ID number to each new loops.

For distributed loops, the compiler adds a suffix to the line number. This suffix identifies a distributed part of the loop. The loop at line 7, for example, is split into two distributed parts, 7-1 and 7-2. All loops nested within 7-*n* have the same suffix.

Look at the first line of the first table in the optimization report. The "Reordering Transformation" column says that the compiler distributed the loop. The "New Loops" column says that the two new loops have ID numbers 2 and 3. If you scan down the "Id. Num" column, you see that these ID numbers correspond to scalar loops 7-1 and 7-2.

The break in the table between line 8-1 and line 7-2 helps to distinguish between loop nests. The report shows the `j` loop at line 8 is present in both distributed nests, but the `k` loop at line 11 is only present in the second nest. The report also shows that the compiler interchanged the `j` and `k` loops in the second nest.

The array report shows that the compiler hoisted the load and sunk the store of `c` from line 12, but not from line 9.

Here is an example of a vector reduction the compiler performs:

```
float example2(float a[],int n)
{
 int i;
 float sum = 0.0;

 for( i=0; i<n; i++ )
    sum += a[i];
 return( sum );
}
```

Compiling `example2` at `-O2` generates the following optimization report:

```
          Optimization by Loop for Routine example2

Line     Id     Iter.   Reordering       New      Optimizing / Special
Num.     Num.   Var.    Transformation   Loops    Transformation
-------------------------------------------------------------------------
   6       1    i       FULL VECTOR               Reduction
```

The `Optimizing/Special Transformation` column indicates that the compiler recognized the vector reduction.

# Bibliography

## CONVEX documents

The folowing documents, available from CONVEX Press, provide information related to this book:


- *CONVEX C Guide* (DSW-086).
- *CONVEX Performance Analyzer (CXpa) User's Guide* (DSW-251).
- *CONVEX CXdb User's Guide* (DSW-473).
- *CONVEX Consultant User's Guide* (DSW-025).


## Other documents


- 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 for Information Systems — Programming Language* C. New York, New York: American National Standards Institute, 1990.
- 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.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language.* Englewood Cliffs, NJ: Prentice Hall, 1988.

- Levesque, John M. and Joel W. Williamson. *A Guidebook to FORTRAN on Supercomputers*. San Diego: Academic Press, Inc., 1989.

- Padua, David A, and Michael J. Wolfe, "Advanced Compiler Optimizations for Supercomputers." *Communications of the ACM* (December 1986).

- Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge, MA: Cambridge University Press, 1986.

- Schofield, C. F. *Optimising FORTRAN Programs*. England: Ellis Horwood Limited, 1989.

- Sedgewick, Robert. *Algorithms in C*. Reading, MA: Addison-Wesley, 1990.

- Stone, Harold S. *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley, 1987.

- Wolfe, Michael Joseph. *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: The MIT Press, 1989.

# Glossary

## A

**alias**
Multiple name for a single memory location. A typical alias arises in a function to which a nonlocal has been passed, if that memory location is also referenced within the function as a nonlocal. In the following example, z is an alias for y in this invocation of function sub:

```
int z[1000];

void sub(int y[])
{
   ...
}

main()
{
  sub(z);
   ...
}
```

Another kind of alias occurs across function calls. In the following example, b is an alias for c in this invocation of function sub:

```
void sub(int b[], int c[])
{
    ...
}

main()
{
  int a[100];

  sub(a,a);
   ...
}
```

Refer to Chapter 8, "Aliasing," for a more thorough explanation of aliases.

### ASAP

Automatic Self-Allocating Processors, a unique architectural feature designed by CONVEX. A cornerstone of ASAP is the communication register, which allows CPUs to seek out and execute the next piece of work as soon as possible.

## B

### balancing

See *tree-height reduction.*

### 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 function contains at least one basic block and typically contains many. The following function is divided into three basic blocks:

```
float abs(float a)
{
  float tmp;
        /* begin basic block 1 */
  tmp = a;
  if( a < 0 )
        /* begin basic block 2 */
    a = -a;
        /* begin basic block 3 */
  return( a );
}
```

## C

### chaining

See *vector chaining.*

### chime
A chained vector time. The time required to perform the simultaneous instructions of one vector chain. On 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. This is the default storage method for arrays in FORTRAN. For example in FORTRAN, for an array `A(3,4)`, element `A(3,1)` immediately precedes element `A(1,2)` in memory.

### 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 allows 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
A variable that changes linearly within the loop but is not incremented on every iteration

### constant folding
Replacement of an operation on a constant or 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 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 until x and/or y is reassigned.

**CPU**
Central processing unit

**CPU time**
The amount of time the CPU requires to execute a program. Because programs share access to a CPU, a program's wall-clock time may not be the same as its CPU time. If a program can use multiple processors, the CPU time may be greater than the wall-clock time. (See *wall-clock time.*)

**critical region**
A segment of code that must be executed by only one CPU at a time

## D

**data dependency**
A relationship between two statements, such that one statement must precede the other to produce the intended result. (See also *loop-carried dependency* and *loop-independent dependency.*)

## E

**execution stream**
A series of instructions that a CPU executes

## F

**fine-grained**
See *granularity.*

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

## G

**gather**
A vector operation that loads values from an array into a vector register. The operands of this operation come from various locations in an array.

### 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.

## H

### hoist

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

## I

### interleaved memory

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

## L

### loop-carried dependency (LCD)

A dependency between two operations executed on different iterations of a given loop and on the same iteration of all enclosing loops. A loop carries a dependency from an indexed assignment to an indexed use if, for some iteration of the loop, the assignment stores a value that is referred to on a later iteration of the loop.

For example, an LCD from `a[i+1]` to `a[i]` exists in the following loop:

```
for( i=0; i<100; i++ )
  a[i + 1] += a[i];
```

An LCD from `b[i+1]` to `b[i]` exists in the following loop:

```
for( i=0; i<100; i++ ){
  a[i] = b[i] + c[i];
  b[i + 1] = d[i] * 3.14;
}
```

### 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:

```
for( i=0; i<100; i++ ){
  a[i] = b[i] + c[i];
  b[i] = 0.0;
}
```

An LID from `b[99]` to `b[i]` exists in the following loop, though only on the hundredth iteration:

```
for( i=0; i<100; i++ ){
  a[i] = b[99] + c[i];
  b[i] = 0.0;
}
```

### loop induction variable

A variable whose value is incremented by a constant amount on each iteration of the loop. For example, in the following loop, `j` and `k` are induction variables, but `l` is not.

```
for( i=0; i<N; i++ ){
  j = j + 2;
  k = k + N;
  l = l + i;
}
```

### 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

### loop invariant computation

vector stride in the inner loop.

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

## M

**mask, vector**
A bit pattern that selects the operands that are computed in a vector operation. The operands 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 values selected 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

## O

**oversubscript**
An array reference that falls outside declared bounds

## P

**parallel vector loop**
A nested loop structure such that the innermost loop is vectorized and the outer strip-mine loop can run in parallel if a CPU is available

**parallelization**
The act of creating code that enables sections of code to run simultaneously on multiple CPUs. At optimization level `-O3`, the CONVEX C compiler automatically parallelizes your program and recognizes compiler pragmas with which you can specify parallelization.

**pipelining**
Grouping multiple instructions 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 (VM) 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 C function

## R

**recurrence**
A cycle of dependencies among the operations within a loop. (See also *data dependency.*)

**reentrancy**
The ability of a function 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]`, element `a[1][3]` immediately precedes element `a[2][0]` in memory. This is the representation used by Ada and C.

## S

**scalar expansion**
The substitution of a temporary vector for a scalar during the vectorization of a loop.

**scalar spreading**
The substitution of a temporary vector for a scalar during the parallelization of a loop.

**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**
The method used to prevent two threads from accessing the same critical region simultaneously. You can synchronize programs using compiler pragmas or assembly-language instructions. You do so, however, at the cost of additional overhead; synchronization may cause at least one CPU to wait for another.

## T

**thread**
An independent execution stream that a CPU fetches and executes. 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 pragmas to source code, or coded explicitly in assembly-language programs.

**thread-private or thread-specific**
Data that is accessible by a single thread only (not shared among the threads constituting a process). Thread-specific data allows the same virtual address to refer to different physical memory locations.

**tree-height reduction**
Expressions are represented internally as trees whose height corresponds to the depth of the expression. These trees are optimized by tree-height reduction or balancing. For example, the height of `a+b+c+d+e+f+g+h` could be seven: `(((((((a+b)+c)+d)+e)+f)+g)+h)`.
However, the compiler orders this expression so that more than one addition can occur at the same time: `(((a+b)+(c+d))+((e+f)+(g+h)))`. The height of this tree is three. Shorter heights mean faster execution. Tree height reduction occurs only for floating-point expressions.

## V

**vector-accumulator register (V)**
A vector register that can contain 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-length register (VL)**
A vector register that holds the number of elements used in subsequent vector operations

**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**
The distance in bytes between adjacent array elements. This figure is used to load arrays into vector accumulators or transfer them to memory from a vector accumulator.

**vector-stride register (VS)**
A vector register that holds the distance in bytes between adjacent array elements

## W

**wall-clock time**
The time an application requires to complete its processing. If an application starts running at 1:00 p.m. and finishes at 5:00 a.m., its wall-clock time is 16 hours. See *CPU time*.

# Index

## D

## J



## L



## M

## Q

## T



## U

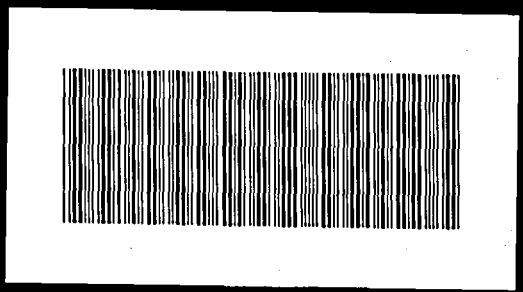


## V

## W



## Z



Order Number
DSW-089

Document Number
720-001130-203