# CONVEX C Optimization Guide

*Second Edition*


CONVEX COMPUTER CORPORATION

# CONVEX C Optimization Guide

Order No. DSW-089

Second Edition
April 1991

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 |
|---|---|---|
| Second | 720-001130-202 | Released with CONVEX C software V4.1, April, 1991. The book was reorganized and rewritten. |
| First | 720-001130-200 | Released with CONVEX C software V4.0, May, 1990. |

# Contents

## CONVEX C Optimization Guide
## Second Edition

## Appendixes

# About 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 contains a glossary of terms used throughout the document.
- ❑ Appendix F contains a bibliography on topics related to optimization.

## Related reading

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

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

## Ordering documentation

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

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

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

## Technical assistance

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

- Within the continental U.S., use 1(800)952-0379.
- From locations in Canada, use 1(800)345-2384.
- From all other locations, contact your local CONVEX office.

# Acknowledgments

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

# 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. These transformations are cumulative: each higher-level option retains the transformations of the previous option. Figure 1-1 summarizes the optimization options.

**Figure 1-1**
Optimization options

| Option | Description |
|---|---|
| `-no` | Machine-dependent scalar optimization. This option is the default. |
| `-O0` | Basic-block machine-independent scalar optimization |
| `-O1` | Basic-block and 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, "Optimization 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 deceases 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 functionally equivalent assembly instructions for the C source statement shown in Figure 2-1.

**Figure 2-1**
Instruction scheduling at `-no`

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

| Original Code | Optimized Code |
|---|---|
| `ld.w  d, s0` | `ld.w  d, s0` |
| `ld.w  c, s1` | `ld.w  c, s1` |
| `mul.s s1,s0` | `ld.w  b, s2` |
| `ld.w  b, s1` | `mul.s s0,s1` |
| `add.s s1,s0` | `ld.w  e, s0` |
| `ld.w  e, s1` | `ld.w  f, s3` |
| `div.s s1,s0` | `add.s s1,s2` |
| `ld.w  f, s1` | `div.s s0,s2` |
| `sub.s s1,s0` | `sub.s s3,s2` |
| `st.w  s0, a` | `st.w  s2, a` |

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

## 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* and *CONVEX Assembly-Language 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:

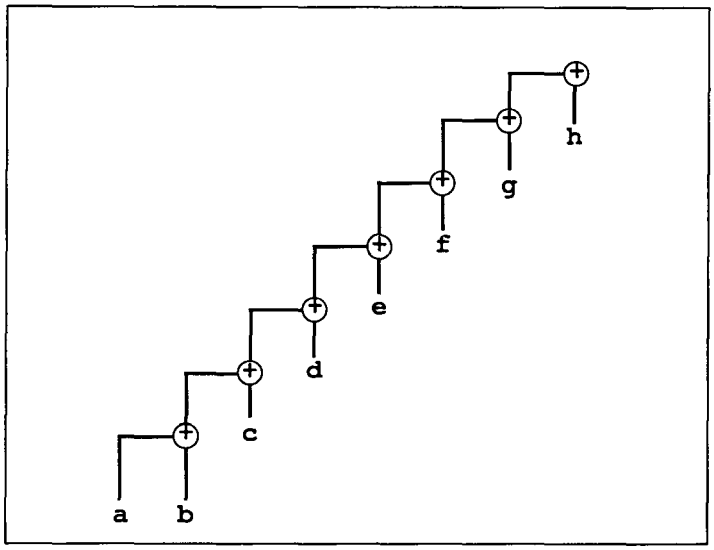
```
a + b + c + d + e + f + g + h
```

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 2-2 shows how the compiler represents this order internally:

**Figure 2-2**
Unbalanced tree representation

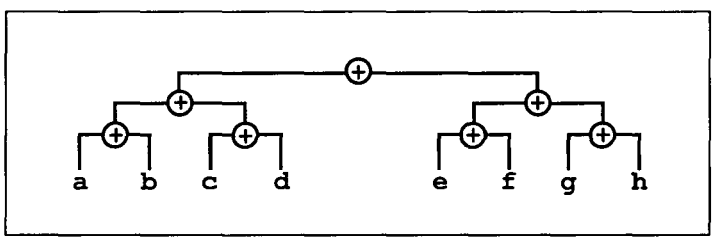



Another way to evaluate the integer expression is:

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

Because none of the four additions in the innermost parentheses requires the result of another addition, the additions can be done simultaneously on several functional units. `((a+b)+(c+d))` and `((e+f)+(g+h))` are then evaluated. The compiler represents this order internally as a balanced tree, as shown in Figure 2-3.

**Figure 2-3**
Balanced tree representation




In Figure 2-2, the depth of the tree is seven; in Figure 2-3, the depth of the tree is three. The machine instruction sequence generated for the tree in Figure 2-3 executes faster than the instruction sequence generated for the tree in Figure 2-2.

The deeper the tree representing the expression, the more time is required to evaluate the expression. The compiler chooses an evaluation order that minimizes the depth of the expression and maximizes instruction pipelining. 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 | Optimized Code |
|---|---|
| `ld.w d, s0` | `ld.w d, s0` |
| `ld.w c, s1` | `ld.w c, s1` |
| `ld.w b, s2` | `ld.w b, s2` |
| `mul.s s0,s1` | `mul.s s0,s1` |
| `add.s s1,s2` | `ld.w e, s0` |
| `st.w s2, t` | `ld.w f, s3` |
| | `add.s s1,s2` |
| `ld.w d, s0` | `st.w s2, t` |
| `ld.w c, s1` | `div.s s0,s2` |
| `ld.w b, s2` | `sub.s s3,s2` |
| `mul.s s0,s1` | `st.w s2, a` |
| `ld.w e, s0` | |
| `ld.w f, s3` | |
| `add.s s1,s2` | |
| `div.s s0,s2` | |
| `sub.s s3,s2` | |
| `st.w s2, a` | |

**Figure 2-4** Instruction scheduling at -O0

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.

**Figure 2-5**
Eliminating redundant assignments at `-O0`

| Original Code | Optimized Code |
|---|---|
| `x = y + c;` | `/*(statement removed)*/` |
| `/* x is not used */` | `...` |
| `x = 3.1416;` | `x = 3.1416;` |
| `...` | `...` |
| `y = (x + 7) * 2.15;` | `y = (x + 7) * 2.15;` |

## Assignment substitution

Assignment substitution eliminates redundant loads. The compiler retains the value assigned to a variable and replaces subsequent references to that variable with the assigned value. Figure 2-6 shows an example.

**Figure 2-6**
Assignment substitution

| Original Code | Optimized Code |
|---|---|
| `x = y + c;` | `REG = y + c;` |
| `x = x * 4.4;` | `REG = REG * 4.4;` |
| `t = x * b + 12.4;` | `t = REG * b + 12.4;` |
| `x = 4.179;` | `x = 4.179;` |

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

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

**Figure 2-7**
Folding and propagating constants at `-O0`

| Original Code | Optimized Code |
|---|---|
| `i = 5;` | `i = 5;` |
| `j = 0;` | `/*(assignment 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 under-flow 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 in Figure 2-8.

**Figure 2-8**
Algebraic and trigonometric simplification

| Original Expression | Optimized Expression |
|---|---|
| `x + 0` | `x` |
| `x * 1` | `x` |
| `x * 0` | `0` |
| `k & -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 the operations shown in Figure 2-8 for the 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 in Figure 2-9.

**Figure 2-9**
Constant propagation and folding at `-O1`

**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. Figure 2-10 shows how the compiler eliminates redundant assignments to the variables a and x.

**Figure 2-10**
Eliminating redundant assignments at -O1

**Original Code**

```
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 in Figure 2-11.

**Figure 2-11**
Redundant assignment eliminated at -O1

**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. Figure 2-12 shows an example.

**Figure 2-12**
Eliminating function assignments at `-O1`

| Original Code | Optimized Code |
|---|---|
| `int foo()`<br>`{`<br>`  ...`<br>`  i = intfun(x);`<br>`  ...`<br>`}`<br>`/* i is not used */` | `int foo()`<br>`{`<br>`  ...`<br>`  intfun(x);`<br>`  ...`<br>`}` |

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 or 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, "Optimization 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 in Figure 2-13 shows a common subexpression.

**Figure 2-13**
Eliminating a common subexpression

**Original Code**

```
void foo()
{
   ...
   a = b + c / (-j * b + sqrt(c));
   if (k < 1)
      l = 5;
   f = e - c / (-j * b + sqrt(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 in Figure 2-14.

**Figure 2-14**
Common subexpression eliminated

**Optimized Code**

```
void foo()
{
   ...
   t1 = c / (-j * b + sqrt(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 Figure 2-15, all variables used in the assignment to `a` remain invariant within the loop. The compiler recognizes this and moves the calculations and assignments out of the loop, performing these costly calculations only once.

**Figure 2-15**
Candidate for code motion

**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) + sqrt(c));
      ar[i] = a + b * c;
   }
   ...
}
```

At higher optimization levels, the compiler can vectorize the loop, as shown in Figure 2-16.

**Figure 2-16**
Code motion performed

**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 the C100 and C200 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 replaced with integer shifts because the CONVEX architecture has a logical shift instruction, but not an arithmetic shift instruction. (Logical shifts do not sign-extend.)

```
a / 2 remains a / 2
```

Multiplication involving real constants 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 Figure 2-17, 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.

**Figure 2-17**
Strength reduction of a loop

**Original Code**

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

Figure 2-18 shows that 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.

**Figure 2-18**
Loop's strength reduced

**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

Appropriate use of vector instructions is the key to high performance on CONVEX C100 and C200 Series architectures. Vectorization converts loops performing scalar operations on array elements into equivalent vector operations. The `-O2` compiler option instructs the compiler to vectorize loops in a program. For loops that cannot be vectorized, the compiler carries out the global transformations performed at `-O1`.

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

## Transformations the compiler performs

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

### Strip mining

Each vector register holds up to 128 elements. When the iteration count of a vectorizable loop is unknown or exceeds 128, the compiler strip-mines the loop before vectorizing it. Strip mining replaces the original loop with two loops. The inner loop has an iteration count that never exceeds 128. The outer loop controls the number of times the inner loop is executed.

**Figure 3-1**
Strip mining—original loop

**Original Loop**

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

In the vectorized loop code shown in Figure 3-2, `iout` is a variable that the compiler uses to count the number of elements remaining to be processed, and the vector operations are shown using the section notation described above. `i` is the starting index for each vector operation.

**Figure 3-2**
Strip mining—vectorized loop

**Vectorized Loop**

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

If `n` equals 300, `iout` is tested four times. For each comparison of `iout` to zero, the table in Figure 3-3 shows values of `i` and `iout` and the elements of `a` that are calculated.

**Figure 3-3**
Iterations of a strip-mine loop

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

The fourth test of `iout` fails, so the loop is not executed and no elements of `a` are processed.

## Loop distribution

Vectorization is only done on simple loop nests. A simple loop nest is one in which all calculations are done in the innermost loop. Nested loops 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 in Figure 3-4.

**Figure 3-4**
Candidate for loop distribution

**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 in Figure 3-5.

**Figure 3-5**
Loop distributed

**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 in Figure 3-6.

**Figure 3-6**
Candidate for loop interchange

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

**Figure 3-7**
Interchanged loops

**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

A vector register can sometimes be used as an accumulator, making it possible for the compiler to move loads and stores of the register outside the vector loop. As noted in Chapter 2, hoisting is the movement of an operation (such as loading a register) out of a loop to a basic block preceding the loop.

Sinking is the complement of hoisting. The compiler moves an operation, such as a register store, out of a loop to a basic block following the loop. Figure 3-8 shows a loop nest that is a candidate for hoisting and sinking.

**Figure 3-8**
Paired hoist and sink

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

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

Figure 3-9 shows an example of vector hoisting and sinking. In this example, `V0` refers to a vector register.

**Figure 3-9**
Vector hoisting and sinking

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

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

- The array reference and array assignment have the same subscript expression.
- Each subscript in the array reference and assignment is either a *loop induction variable* of the vectorized loop or a loop constant with respect to the vectorized loop.

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

## Conditional induction variables

A loop induction variable is normally defined as a variable whose value is incremented by a constant amount on every iteration of the loop. A *conditional induction* variable is similar to an induction variable except that it does not change on every iteration of the loop. The compiler frequently recognizes both types of induction variables and generates vector code for expressions involving them.

In Figure 3-10, `k` is a conditional induction variable.

**Figure 3-10**
Conditional induction variables

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

For the code in Figure 3-10, the compiler generates machine instructions that do the following:

- Save values of `i` for which `e[i]` is true.
- Count the number of those values.
- Load the vector strip of `b`.
- Expand the vector strip of `b` to the appropriate indices according to the saved truth values.
- Store the expanded vector in `a[0:99]`.
- Load the vector `d[0:99]`.
- Compress the vector according to the saved truth values.
- Store the vector in `c`.

For more information regarding the expand and compress operations of vectorization, refer to "Vector-merge register operations" in Appendix D.

## 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 next section defines the problem with unsigned induction variables. Recurrence is then discussed in the remainder of the chapter. Aliasing is defined and discussed in Chapter 8, "Aliasing."

## Unsigned induction variables

The compiler cannot vectorize loops that have an unsigned induction variable. For example the code in Figure 3-11 cannot vectorize because it has an unsigned induction variable.

**Figure 3-11**
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 in Figure 3-12.

**Figure 3-12**
Signed induction variable

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

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

Some loops are written in such a way that the compiler cannot determine whether or not a recurrence exists. 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 loop in Figure 3-13 contains an LCD.

**Figure 3-13**
Loop-carried dependency

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

The dependency is carried by the loop 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 loop in Figure 3-14 contains a forward dependency.

**Figure 3-14**
Forward loop-carried 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 loop in Figure 3-15 contains an LCD that points backward from `b[i+1]` to `b[i]`.

**Figure 3- 15**
Backward loop-carried dependency

```
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 in Figure 3-16.

**Figure 3-16**
Interchanged statements

```
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 eliminates 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 appears in Figure 3-17, where there is an LCD on the variable `x`.

**Figure 3-17**
Scalar expansion

**Original Loop**

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

**Vectorized Loop**

```
for( i=1;i<10;i++ ){
     temp[i]=a[i];
       =...temp[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 might not stop vectorization completely. Using temporary vectors, the compiler can sometimes vectorize part of a loop that contains an LCD. Figure 3-18 shows an example.

**Figure 3-18**
Candidate for partial vectorization

**Original Loop**

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

As shown in Figure 3-18, the assignment to `a[i]` depends on the value of `a[i-1]`, which is computed on the previous iteration. Figure 3-19 shows that the compiler isolates the dependency by distributing the loop and vectorizes the first distributed part. The second distributed part is executed with scalar instructions. This transformation is called *partial vectorization* because it distributes a loop into vector and scalar parts.

**Figure 3-19**
Partial vectorization

**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. Figure 3-20 shows a loop with two LIDs.

**Figure 3-20**
Loop-independent dependency

```
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.**

A LID can stop vectorization by preventing the compiler from eliminating an LCD. In Figure 3-21, the loop cannot be vectorized.

**Figure 3-21**
LID inhibiting vectorization

```
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

An apparent recurrence exists when the compiler lacks sufficient information to prove that an actual recurrence does not exist. Apparent recurrences usually result from using a loop constant of unknown sign or an array reference in an array subscript. Figure 3-22 shows a loop that cannot be vectorized because the sign of k is unknown.

**Figure 3-22**
Apparent recurrence

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

If k is positive or zero, the final value of each element of a[m:n] is 0.0. The compiler cannot interchange the statements because the assignment to a[i] must follow the assignment to a[i+k]. If k is -1, the final value of a[m:n-1] is 2.0; only a[n] is 0.0. The compiler must interchange the statements so the assignment to a[i+k] follows the assignment to a[i]. Because these conditions are contradictory, neither operation can be performed.

The loop in Figure 3-23 cannot be vectorized because the compiler cannot determine whether a recurrence exists.

**Figure 3-23**
Apparent recurrence inhibiting vectorization

```
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.

## Reduction

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

```
X = X operator Y;
```

where X is a variable not assigned or used elsewhere in the loop, Y is a loop constant expression not involving X, and operator is +, -, *, &, |, ==, or !=.

The loop in Figure 3-24 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.

**Figure 3-24**
Vectorization of reductions

**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.

## Optimization report

When you compile a program with the `-O2` or `-O3` option, the compiler generates an optimization report for each function. The `-or` option determines the report's contents, as shown in Figure 3-25.

**Figure 3-25** Optimization report options

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

For example, consider the matrix multiplication algorithm in Figure 3-26. (Line numbers are provided as a reference.)

**Figure 3-26** Matrix multiplication

```
1  extern float a[200][201];
2  extern float b[200][201],c[200][201];
3  void example1()
4  {
5    int i,j,k;
6
7    for( i=0; i<200; i++ ){
8      for( j=0; j<200; j++ ){
9        c[i][j] = 0.0;
10       for( k=0; k<200; k++ )
11         c[i][j] += a[i][k] * b[k][j];
12     }
13   }
14 }
```

The optimization report generated by compiling `example1.c` at optimization level `-O2` is shown in Figure 3-27.

**Figure 3-27**
Optimization report

```
%cc -c -O2 -or all example1.c
          Optimization by Loop for Routine example1

Line     Iter.   Reordering          Optimizing / Special          Exec.
Num.     Var.    Transformation      Transformation                 Mode
-------------------------------------------------------------------------
   7     i       Dist
   7-1   i       Scalar
   7-2   i       Scalar
   8-1   j       FULL VECTOR
   8-2   j       FULL VECTOR Inter
  10-2   k       Scalar

Line     Iter.   Analysis
Num.     Var.
-------------------------------------------------------------------------
   8-2   j       Interchanged to innermost

          Array References for Routine example1

Line   Var.   Optimi-   Dependencies
Num.   Name   zation
-------------------------------------------------------------------------
  11   c       Sunk
  11   c       Hoist
```

Some of the entries in the `Line Num.` column are actually pairs of numbers. This indicates that at least one loop nest in the program was distributed. In the example, the compiler transformed the loop nest on lines 7 through 13 into two loop nests, called distributed parts. In each pair of numbers, the first number is the line number of the loop in the source file and the second number is the number of the distributed part to which the source line is associated.

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

- The `i` loop at line 7 was distributed. It was not vectorized because the `j` loop was vectorized.
- The `j` loop at line 8 was distributed and both distributed parts were fully vectorized and one part was interchanged.
- The `k` loop at line 10 was not vectorized because the `j` loop was vectorized.

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

Figure 3-28 shows an example of other transformations the compiler performs. (Line numbers are provided as a reference.)

**Figure 3-28** C source

```
1 float example2(float a[],int n)
2 {
3    int i;
4    float sum = 0.0;
5
6    for( i=0; i<n; i++ )
7      sum += a[i];
8    return( sum );
9 }
```

Figure 3-29 shows the optimization report generated by compiling the function in Figure 3-28 for vectorization.

**Figure 3-29**
Reduction optimization report

```
%cc -O2 -c example2.c

          Optimization by Loop for Routine example2

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

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

# Parallel optimization

# 4

At optimization level -O3, the CONVEX 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 in Figure 4-1.

**Figure 4-1**
Matrix multiplication

```
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 in Figure 4-2.

**Figure 4-2**
Matrix multiplication with distributed loops

```
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]
  }
}
```

Figure 4-3 shows that the compiler moves the `j` loop to the innermost position in each nest so that it can retrieve contiguous elements on successive iterations.

**Figure 4-3**
Matrix multiplication after loop interchange

```
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.

**Figure 4-4**
Matrix multiplication after strip mining and vectorization

```
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 in Figure 4-5.

**Figure 4-5**
Matrix multiplication after second interchange

```
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 Figure 4-6, PARALLEL FOR represents a loop that multiple CPUs can process.

**Figure 4-6**
Matrix multiplication with parallel outer loops

```
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.

**Figure 4-7**
Matrix multiplication after hoist and sink

```
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.

The call to `sub` in Figure 4-8 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.

**Figure 4-8**
Use of `force_parallel` and `-re`

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

**Figure 4-9**
Loop without dependencies

```
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 loop in Figure 4-10 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.

**Figure 4-10**
Loop with backward LCD

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

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

**Figure 4-11**
Loop with forward LCD

```
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 in Figure 4-12.

**Figure 4-12**
Scalar assignments and reductions in parallel loops

```
for( i=0; i<n; i++ ){
  if( a[i] <= 0 ){
    s += b[i] * c[i];
    x = b[i];
  }
}
  IF (A(I) .LE. 0.0) THEN
    S = S + B(I) * C(I)
    X = B(I)
  ENDIF
ENDDO
```

## Parallelizing code outside of loops

The compiler does not automatically parallelize code outside a loop. You can use tasking 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.

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

**Figure 4-13**
Use of tasking pragmas

```
#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 in Figure 4-14.

**Figure 4-14**
Code transformed with tasking pragmas

```
#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:
  }
      DO I = 1,3
        GOTO (10,20,30)I
  10      <statement 1>
        GOTO 40
  20      <statement 2>
        GOTO 40
  30      <statement 3>
        GOTO 40
      ENDDO
  40 CONTINUE
```

# Optimizing 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.

   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.

```
% 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 have an existing output file, compare `no.out` with it.

```
% 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 with CXpa.

```
% 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 the CONVEX Performance Analyzer (CXpa), you can still perform most of the steps in this chapter. You cannot analyze the performance of individual loops, however. Refer to the *CONVEX Consultant User's Guide* to determine the appropriate options and commands for using the Consultant profilers.

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

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

If the output is significantly changed, use a binary search to isolate the 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 that 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 the same profiler you used in **Step 1**. 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** to determine which functions use the most CPU time. Compile the most time-consuming functions for vectorization. To do this, place the `-O2` option on the `cc` command line. Compile the rest of the program for program-unit optimization and CXpa profiling.

2. Compare the output of your program with the output produced in **Step 2**. The results, allowing for floating-point roundoff, should be unchanged. If the output is significantly changed, use the binary search procedure described in **Step 2** to isolate the 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 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. Enhancing vector optimization

1. Run the vectorized 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 are not vectorized and use significant amounts of CPU time. Note which of these loops are inner loops, which are candidates for vectorization.

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

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

3. When you are satisfied that no more loops can be vectorized or the loops that can be vectorized do not use a significant amount of time, you may still be able to

improve the efficiency of your code. Try the following techniques:

- ❑ Simplify conditionals. Even if a loop is vectorized, an embedded conditional can slow it down.
- ❑ Simplify array subscripts. Array subscripts that require many operations to evaluate can slow down the execution of a loop.
- ❑ Look for loops with short vector lengths (small trip counts). If the trip count is small, the loop probably runs faster in scalar form than it does in vector. On the C100 and C200 Series, this slowdown occurs when the trip count is less than five. Use the `scalar` pragma to stop the compiler from vectorizing such a loop.
- ❑ Look for unnecessary strip mines and inefficient strip-mine lengths. Use CXpa to determine whether a vector loop is strip mined. Use the `max_trips` pragma to stop the compiler from creating unnecessary strip mines around a vector loop.
- ❑ Look for coding practices that cause a slower execution speed, such as data type conversions.

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

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

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

**Note** ✎

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

## Step 5. Adding parallelization

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

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

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.

There are other optimization options you may want to consider:

- `-alias` — aliasing of array parameters and the aliasing algorithm used by the compiler. This option is available at all optimization levels, although it only has an impact at level `-O2` and `-O3`. The `array_args` and `ptr_args` arguments of this option affect the aliasing of array function parameters. Refer to "Array parameters" in Chapter 8 for some examples that demonstrate this use of `-alias`. The worst, cautious, and standard arguments of this option affect the relationship of scalar data type pointers to each other. Refer to "Aliasing algorithms" in Chapter 8 for a discussion of these arguments for `-alias`.
- `-ds` — dynamic selection of loops. This option is available at level `-O2` and higher. With this option, the compiler creates scalar, vector, parallel, and parallel-vector versions of all loops and selects the most profitable version of the loop at runtime. This is an automatic application of the `select` pragma with default values. Refer to "Do not vectorize loops with small trip counts" in Chapter 7 for more information.
- `-rl` — unrolling and dynamic selection of loops. This option is available at level `-O2` and higher. This option is equivalent to `-ds` and `-ur`.

- -uo — unsafe optimizations and pattern matching. This option is available at level `-O1` and higher. Refer to Appendix A, "The `-uo` option," for more information.
- `-ur` — partial and complete unrolling of loops. This option is available at level `-O2` and higher. With this option, the compiler attempts to completely unroll innermost loops that have a loop count less than five, or partially unroll simple innermost loops. Refer to "Do not vectorize loops with small trip counts" in Chapter 7 for more information.

# Efficient programming constructs

6

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 loop in Figure 6-1 will not vectorize unless you compile with the `-float sp_const` command line option.

**Figure 6-1**
Loop reduction with `-float sp_const`

```
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. Figure 6-2 shows four typical counted loops.

**Figure 6-2**
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. Figure 6-3 shows a loop that has no induction variable.

**Figure 6-3**
Loop without 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 in Figure 6-3 can be unrolled by hand, as shown in Figure 6-4. Because the loop overhead is eliminated, the unrolled code runs faster than the original loop.

**Figure 6-4**
Unrolled loop

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

If the iteration variable of a loop is incremented by a non-integer constant, the loop has no induction variable and the compiler cannot vectorize it. The loop in Figure 6-5, for example, increments `i` by a `float` value, which prevents vectorization of the loop.

**Figure 6-5**
Loop with `float` iteration value

```
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 in Figure 6-6, often prevent the compiler from vectorizing a loop.

**Figure 6-6**
Complicated iteration test

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

The complexity of the `while` test in Figure 6-6 prevents the compiler from generating code to determine the loop's iteration count at runtime. As a result, the compiler cannot vectorize the loop.

A stop value can be a variable or a constant, but its value must be determined at runtime prior to the execution of the loop and cannot change within the loop. Figure 6-7 shows a loop whose stop value changes within the loop.

**Figure 6-7**
Loop without fixed stop value

```
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 loop in Figure 6-8, do not vectorize.

**Figure 6-8**
Loop with alternate exit

```
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.

Figure 6-9 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.

**Figure 6-9**
Embedded conditional

```
#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 in Figure 6-10. This change to the source code improves performance dramatically.

**Figure 6-10**
Embedded conditional removed

```
#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;
   }
}
```

Figure 6-11 shows an example of boundary tests that can be removed from a loop.

**Figure 6-11**
Boundary test within a loop

```
main()
{
   int i,j;
   float a[1000][1001], b[1000][1001];

   for( i=0; i<1000; i++ )
      for( j=0; j<1000; j++ ){
         a[i][j] = 1.0;
         b[i][j] = 2.0;
      }

   for( i=0; i<1000; i++ ){
      for( j=0; j<1000; j++ ){
         if( i==0 || i==999 ){
            if( j==0 || j==999 )
               a[i][j] = 0.0;
            else
               a[i][j] = b[i][j];
         } else
            a[i][j] = b[i][j];
      }
   }
}
```

When boundary values are set outside the loop, as shown in Figure 6-12, this code fragment runs several times faster.

**Figure 6-12**
Boundary test outside a loop

```
main()
{
   int i,j;
   float a[1000][1001], b[1000][1001];

   for( i=0; i<1000; i++ )
     for( j=0; j<1000; j++ ){
       a[i][j] = 1.0;
       b[i][j] = 2.0;
     }

   for( i=1; i<999; i++ )
     for( j=1; j<999; j++ )
       a[i][j] = b[i][j];

   a[ 0][ 0] = 0.0;
   a[ 0][999] = 0.0;
   a[999][ 0] = 0.0;
   a[999][999] = 0.0;

   for( i=1; i<999; i++ ){
     a[  0][  i] = b[  0][ i];
     a[999][  i] = b[999][ i];
     a[  i][  0] = b[  i][ 0];
     a[  i][999] = b[  i][999];
   }
}
```

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 right-most, dimension is faster than using innermost loops that vary the leading, or left-most, 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. Figure 6-13 shows three loops in order of decreasing efficiency.

**Figure 6-13**
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 Figure 6-14, the compiler interchanges the `j` and `i` loops.

**Figure 6-14**
Optimization of array accesses

| Original Code | Optimized Code |
|---|---|
| `for(i=0; i<n; i++ )`<br>`  for(j=0; j<n; j++ )`<br>`  a[j][i][1] = 4.0;` | `for(j=0; j<n; j++)`<br>`  for(i=0; i<n; i++)`<br>`  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 C200 Series supercomputer requires eight clock cycles to access data stored in main memory. To speed up memory accesses, the CPU posts requests for data before the data is needed.

Main memory comprises at least four banks of dynamic RAM. The memory system stores data so that contiguous words are in separate memory banks. This is called *memory interleaving*. One memory bank is accessed on each clock cycle. As a result, sequential requests to ascending banks are optimal. Figure 6-15 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.

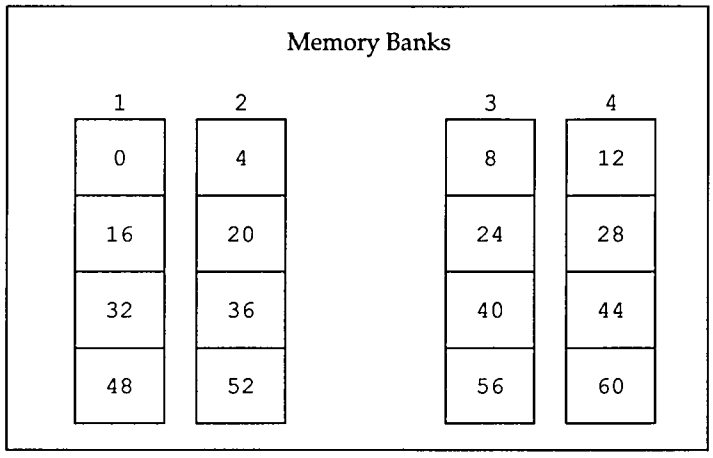



**Figure 6-15**
Four-way interleaved memory

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.

Eight memory banks are needed to return data at the rate of one word per clock cycle. A load instruction, for example, takes eight cycles to return data. If a program makes eight load requests, at a rate of one request per clock cycle, each to a separate bank, data returns at a rate of one per clock cycle, beginning eight clock cycles after the first request. Memory interleaving directly affects efficient array accesses. Figure 6-16 shows a one-dimensional array in eight-way interleaved memory.

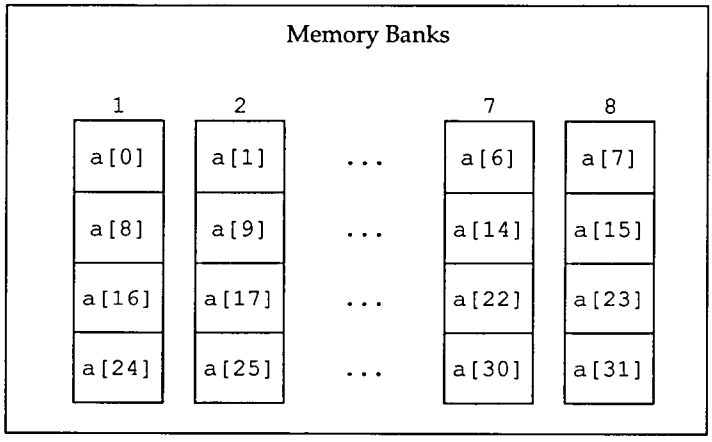



**Figure 6-16**
One-dimensional array in eight-way interleaved memory

The loop in Figure 6-17 processes an array sequentially. After an initial wait of eight clock cycles, the CPU receives one data word per clock cycle.

**Figure 6-17**
Sequentially processed array

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

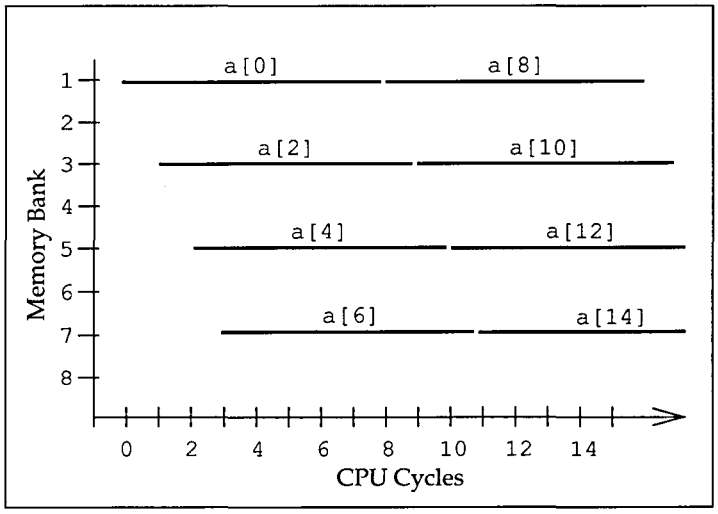
The loop in Figure 6-18, however, causes memory bank conflicts. The CPU must wait for memory requests to be filled.

**Figure 6-18**
Nonsequentially processed array

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

Figure 6-19 shows the timing relationships that cause these bank conflicts.

**Figure 6-19**
Bank conflict




Load requests occur each clock cycle. The first request, for `a[0]`, keeps bank 1 occupied for eight clock cycles. The CPU cannot access the data in `a[8]` until this first access is satisfied. This results in a delay of four clock cycles.

Memory bank conflicts occur when an array's stride does not efficiently use the memory of the computer. An array's stride is the difference in the index value between two successive iterations in addressable units, which are different on the C1 and C2 Series Computers. In the loop in Figure 6-20, `arr` has a stride of one.

**Figure 6-20**
Array with stride of one

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

The loop in Figure 6-21, which accesses every other element of the array, has a stride of two.

**Figure 6-21**
Array with stride of two

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

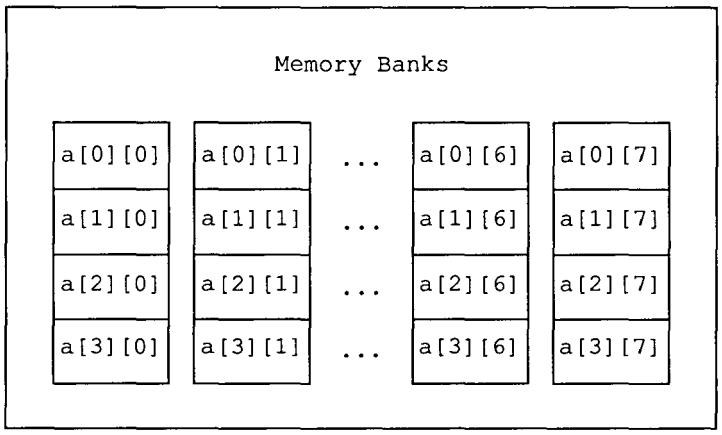
Arrays with a stride of two use only half the memory banks. Arrays with a stride of four use one bank in four. Whenever possible, avoid writing loops with a stride that is a multiple of a power of two. Odd strides give better performance than even strides do.

## Multidimensional arrays

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 6-22 shows how a two-dimensional, four-by-eight row-major array is stored in memory with eight-way interleave.

**Figure 6-22**
Two-dimensional array stored in eight-way interleaved memory




The loop in Figure 6-23 processes a two-dimensional array.

**Figure 6-23**
Loop with even trailing index to process 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 6-22, and eight clock cycles are required to load each element into the vector register.

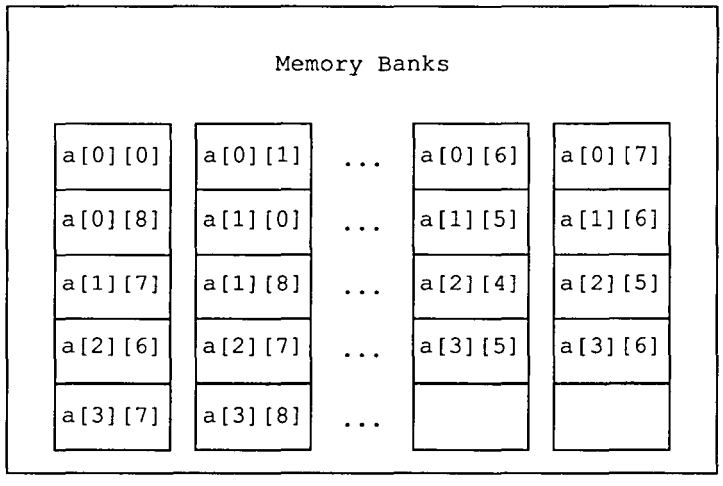
To avoid bank conflicts, declare the trailing index of `a` to be an odd number, as shown in the loop in Figure 6-24.

**Figure 6-24**
Loop with odd trailing index to process a two-dimensional array

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

Figure 6-25 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 6-25**
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 trip count of a loop is less than or equal to 128, the loop can be executed with a single set of vector operations. In this case, the compiler does not strip mine the loop. Loops are often written with variable trip counts. Unless the compiler can determine the value of the trip-count variable (through constant propagation, for example), the compiler must strip mine the loop to allow for a possible trip count greater than 128. Figure 7-1 shows a loop with a trip count that varies between 1 and 50.

**Figure 7-1**
Unnecessary strip mine

```
n = getval(n);/* returns a value from 1 to 50 */
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 in Figure 7-2.

**Figure 7-2**
Strip mine removed

```
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 trip count cannot exceed 50, use that value. This value permits the compiler to generate a more efficient loop.

## Do not vectorize loops with small trip counts

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

**Figure 7-3**
Trip count of four

```
#pragma _CNX scalar
for(i=0; i<4; i++)
  a[i] = b[i] * c[i];
```

The compiler usually vectorizes and strip mines loops with variable trip counts. The compiler strip mines the loop in Figure 7-4 because it cannot determine the trip count.

**Figure 7-4**
Variable trip count

```
n = getval(n); /* returns a value of 1, 2, 4,
                  8, 16, or 32 */
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 often this loop has a trip count so small that it should not be vectorized. You can rewrite this loop and use the `scalar` pragma to eliminate the overhead of a vectorized loop when the trip count is five. Figure 7-5 shows an example.

**Figure 7-5**
Using `max_trips` and `scalar`

```
n = getval(n);
if( n>=4 ) {
# 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. Figure 7-6 shows the use of the `select` pragma.

**Figure 7-6**
Using `select`

```
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 trip 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. In this case, however, you lose the benefit of the `max_trips` pragma. The `-ds` command line option is very similar to the `select` pragma. This option applies the `select` pragma to all loops in a compilation unit, using default parameters optimal for the computer the program is compiled for.

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

**Figure 7-7**
Using `unroll`

| **Original Loop** | **Loop Unrolled** |
|---|---|
| `#pragma _CNX unroll` | `a[0] += 1;` |
| `for( i=0; i<4; i++ ){` | `a[1] += 1;` |
| `  a[i] += 1;` | `a[2] += 1;` |
| `}` | `a[3] += 1;` |

The `unroll` pragma works only if the compiler can determine that the trip count is less than five. If a loop has a trip count greater than five, the compiler will attempt to partially unroll the loop. Figure 7-8 shows a loop and how it is partially unrolled.

**Figure 7-8**
Partially unrolled loop

**Original Loop**

```
# pragma _CNX unroll
  for( i=1;i<16;i++ )
    b[i] = b[i-1];
```

**Loop Partially Unrolled**

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

The unroll pragma must be used on the innermost loop. If it is not used on the innermost loop, it is ignored. 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 Figure 7-9, 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.

**Figure 7-9**
Dependencies prevent loop distribution

```
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 in Figure 7-10.

**Figure 7-10**
Array promoted to two dimensions

```
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 in Figure 7-11.

**Figure 7-11**
Distributed `i` loop

```
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 register load */
  for(i=0;i<1000;i++){/*interchanged vector
                        reduction */
    s0 += b[j][i];
  }
  q[j] = s0;    /* sunken register 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.

## Removing conditionals from loops

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

The compiler vectorizes the `i` loop in Figure 7-12, but the loop has a series of embedded `if` tests that slow it down.

**Figure 7-12**
Conditional within a vector loop

```
for( i=0; i<10000, i++ ){
  if( i<2000 ){
   c1[i]=a1[i] * 2000.0 + cos(a1[i]);
   b1[i]=b1[i]*c1[i]*c1[i]*c1[i]*c1[i]/a1[i];
  }
  if( i>=2000 && i<4000 ){
    c1[i] = a1[i] + cos(a1[i]);
    b1[i] += b1[i] + c1[i];
  }
  if( i>=4000 && i<6000 ){
    c1[i] = a1[i] + 2000.0;
    b1[i] = b1[i] * b1[i] * b1[i] * b1[i];
  }
  if( i>=6000 ){
    c1[i] = a1[i];
    b1[i] = 1.0;
  }
}
```

To improve execution speed, remove the conditional by distributing the loop. This produces four distributed parts, shown in Figure 7-13.

**Figure 7-13**
Conditional removed

```
for( i=0; i<2000, i++ ){
  c1[i]=a1[i] * 2000.0 + cos(a1[i]);
  b1[i]=b1[i]*c1[i]*c1[i]*c1[i]*c1[i]/a1[i];
}
for( i=2000; i<4000, i++ ){
    c1[i] = a1[i] + cos(a1[i]);
    b1[i] += b1[i] + c1[i];
}
for( i=4000; i<6000, i++ ){
  c1[i] = a1[i] + 2000.0;
  b1[i] = b1[i] * b1[i] * b1[i] * b1[i];
}
for( i=6000; i<10000, i++ ){
  c1[i] = a1[i];
  b1[i] = 1.0;
}
```

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

# Aliasing

# 8

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

## Aliasing and dependency

A potential alias can prevent the compiler from vectorizing some code. In some cases, the `-alias array_args` option, the `-alias ptr_args` option, and the `no_recurrence` pragma can be used to allow the vectorization of code that would not otherwise be vectorized. Care must be taken to ensure that this does not change the results of the program.

Aliasing compounds the problems of dependency and recurrence. Consider the example in Figure 8-1.

**Figure 8-1**
Array dependencies

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

This loop appears to be free of backward dependencies. Assuming `arra` and `arrb` are distinct, it can be safely vectorized. However, if `arra` is an alias for `arrb` (that is, if `arra` and `arrb` overlap), a backward dependency may exist. If `arra` and `arrb` are aliased together and the types of the elements are the same, the code is equivalent to that of Figure 8-2.

**Figure 8-2**
Aliased array dependencies

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

K is a value that determines how the arrays overlap. If K is positive, a backward dependency exists. When aliasing occurs, this value cannot be determined at compile time, so the compiler cannot know whether a backward dependency exists. To avoid problems due to possible undetected dependencies, the compiler does not try to vectorize the loop in Figure 8-1.

## Why aliasing occurs

The C language supports explicit aliasing in the form of unions. This type of aliasing rarely causes problems in vectorization. More often, optimization is prevented by implicit aliasing that results from the use of pointers. Pointer aliasing, in fact, is one of the most frequent causes of optimization difficulties in C.

Consider the example in Figure 8-3. Function `alex` has three parameters, `a`, `b`, and `n`. The variables `a` and `b` are pointers of type `float`, `n` is an `int`.

**Figure 8-3**
Pointer aliasing

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

Pointers `a` and `b` point to arrays that are passed at runtime. Depending on the addresses passed to the function, the array that `a` points to may overlap the array that `b` points to. The possibility of overlap creates a potential alias. Because of this potential alias, the compiler assumes a dependency may exist and does not vectorize the loop.

This loop is vectorized if you use a `no_recurrence` pragma. Unless you know that the arrays that `a` and `b` point to do not overlap (aliasing does not occur), this use of the `no_recurrence` pragma is unsafe.

## Aliasing algorithms

The algorithm the compiler uses to determine whether a potential alias exists depends on the compatibility mode. CONVEX C compilers prior to V4.0 use worst-case aliasing. This algorithm is still used when a program is compiled in backward-compatible (non-ANSI C) mode. When a program

is compiled in an ANSI C mode, the new, ANSI C aliasing algorithm is used.

The 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 compiler optimization reports) includes all global variables, local variables whose address is taken using the address ( & ) operator, and local static variables. This treatment of static variables is necessary to allow for recursion, since a static is, in a sense, "global" to a function on a recursive call. Here's the result:

- Every pointer's reference is a potential alias for every other pointer's reference.
- Every pointer's reference is a potential alias for every global variable (and vice versa).
- Every pointer's reference is a potential alias for every local variable whose address is taken using the & operator.
- Every pointer's reference is a potential alias for every local static variable.

The ANSI C algorithm is somewhat different. ANSI C provides stricter type-checking, which allows the compiler to use a stricter algorithm that finds fewer potential aliases. Instead of one giant array, the ANSI C algorithm assumes a separate array for each base type such as `int` , `float` , or `double`. Pointers and variables can only be aliased with pointers and variables of the same base type.

**Note** ✎

**Aliasing algorithms are applied on a function-wide basis, before optimization takes place. The aliasing algorithms are flow insensitive; this means that if an identifier is assigned the address of a loop induction variable after a loop, the loop cannot vectorize.**

The example in Figure 8-4 shows the difference between the two aliasing algorithms.

**Figure 8-4**
Code benefited by ANSI C aliasing

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

Here, `a` and `ib` are pointers to different base types. Under ANSI C rules, no potential alias exists, and the loop is vectorized. Under the old worst-case aliasing rules, a potential alias exists and the loop is not vectorized.

The ANSI C aliasing algorithm may not be safe in all cases, especially if your program is not compliant. The code fragment in Figure 8-5 shows how pointers of different base types can become aliased to one another.

**Figure 8-5**
ANSI C violation

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

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

The assignment to `fptr` makes the code non-ANSI C compliant. The code compiles with warning messages (shown as comments in this example). Because the code violates ANSI C rules, it is no longer safe to use the ANSI C aliasing algorithm. (You can convert the warning message generated by the code in Figure 8-5 to an error message by compiling it with the `-d arg_ptr_ref=e` command line option.)

You can specify the aliasing algorithm with three arguments of the `-alias` compiler option: `cautious`, `standard`, and `worst`. The `cautious` argument causes the compiler to use the ANSI C aliasing algorithm, but if it finds constructs that are not ANSI C code, it reverts to the worst-case aliasing algorithm. The `standard` argument causes the compiler to use the ANSI C aliasing algorithm. The `worst` argument causes the compiler to use the worst-case aliasing algorithm. By default, the compiler uses the `cautious` argument.

In the code in Figure 8-6, function `foo2` passes two `int` pointers to function `foo`, which expects a pointer of type `int` and a pointer of type `char`. This is flagged by the compiler as only a warning message. Because the formal parameters are of different base types, the ANSI C aliasing algorithm finds no potential aliases and the compiler vectorizes the loop, even though aliasing exists.

**Figure 8-6**
Aliasing the C compiler cannot detect

```
foo(int *ia, char *cb)
{
  int i;
  for (i=0; i<500; i++ )
    cb[i] = ia[i];
}

foo2()
{
  int arra[600];

  foo(&arra[0], &arra[100]);
}
```

## Array subscripts

It is possible for a variable that appears in an array subscript to become aliased with a pointer. In Figure 8-7, the variable `k` is a potential alias for `*iptr` because its address is passed to the function `getval`. The variable `k` may occupy the same memory address as `iptr[j]` on some iteration of the `j` loop. If this happens, the value of `k` is changed for all subsequent iterations. Because the value of `k` can be changed by an assignment to `iptr[j]`, a real dependency exists. The compiler does not vectorize the loop.

**Figure 8-7**
Operand `&` and aliasing

```
int k;
subex(iptr, n, j)
int *iptr, n, j;
{
  n= getval(&k,n);
  k=12;
  for (j=0; j<n; j++ )
    iptr[j] += k;
}
```

It is generally not safe to use the `no_recurrence` pragma on this loop.

## Induction and stop variables

Even when a dependency does not exist, aliasing can stop vectorization by preventing the compiler from recognizing a loop induction variable. In the example in Figure 8-8, the address of `j` is referenced using the address operator and assigned to `*ip`. As a result, the aliasing algorithm assumes that `j` is part of the global memory array and, therefore, a potential alias for `*iptr`. Because values are assigned to `iptr[j]` within the loop, this potential aliasing means that the value of `j` may be altered. As a result, `j` is not an induction variable and the `j` loop is not a counted loop that can be vectorized.

**Figure 8-8**
Operand `&` and induction variables

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

A `no_recurrence` pragma does not solve the problem, because the problem is not caused by dependencies. The solution is to remove the line of code that references the address of `j`.

The use of a pointer as a loop counter has the same effect. When the code in Figure 8-9 is compiled under the worst-case aliasing rules, the compiler recognizes that `*j` is not an induction variable—because it is a potential reference for `fptr[*j]`, it might be altered within the loop — and does not vectorize the loop.

**Figure 8-9**
Loop counters as pointers

```
ialex2(fptr, j, n)
float *fptr;
int *j, n;
{
  for (*j=0; *j<n; (*j)++ )
    fptr[*j] = 1.7;
}
```

Because `*j` is type `int` and `*fptr` is type `float`, this loop vectorizes under ANSI C aliasing rules.

The aliasing of a stop variable can also prevent vectorization of a loop. In the example in Figure 8-10, the stop variable in the `i` loop ( `n` ) becomes aliased when its address is passed to the function `foo`. Because `n` is a potential alias for `dptr[]`, its value could potentially be altered within the loop. The `i` loop, therefore, is not a counted loop and cannot be vectorized.

**Figure 8-10**
Aliasing of stop variables

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

Note that `n` and `dptr` are of different base types. Under ANSI C aliasing rules this loop can be vectorized as written. To vectorize this loop under the worst-case aliasing rules, create a temporary variable to hold the stop value for the loop, as shown in Figure 8-11.

**Figure 8-11**
Using a temporary variable as a stop value

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

Because `tmp` is not aliased to `dptr`, the stop value of the loop is now fixed, and the loop vectorizes.

## Global variables

The potential aliasing of a global variable and a pointer frequently causes optimization difficulties. The code in Figure 8-12, for example, uses a global variable as a stop value. This variable ( `n` ) is of the same type as the pointer `ik` and must therefore be considered a potential alias. Because

the value of `n` could be altered by an assignment to its alias, `ik[]`, the loop in the code is not a counted loop and cannot be vectorized.

**Figure 8-12**
Global variables as stop values

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

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

To vectorize this loop, use a local variable instead, as shown in Figure 8-13.

**Figure 8-13**
Use local variables as stop values

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

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

Changing the declaration of `ik` from a pointer to a global array also prevents aliasing and allows the loop to be vectorized, as shown in Figure 8-14.

**Figure 8-14**
Global arrays prevent aliasing

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

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

## Array parameters

When an array is passed as a parameter to a C function, it is passed as a pointer. This might appear to be a serious problem when you are trying to avoid aliasing. The CONVEX C compiler provides an option, however, that allows the compiler to treat array parameters as arrays, which are not aliased, instead of as pointers. This option is `-alias array_args`.

The loop in the code in Figure 8-15 does not vectorize because of potential aliasing. Under the ANSI C rules, `arra` is a potential alias for `arrb`. Under the old, worst-case rules, `arra` and `arrb` are also potential aliases for `n`.

**Figure 8-15**
Potential aliasing of array parameters

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

To vectorize this loop, compile the function with `-alias array_args`. You must also add a temporary variable to hold the loop's stop value, as shown in Figure 8-1, because of the aliasing of `n`.

**Figure 8-1**
Potential aliases and the `-alias array_args` option

```
parex(arra, arrb, n)
float arra[], arrb[];
int *n;
{
  int i, stop;
  stop = *n;
  for (i=0; i<stop; i++ )
    arra[i] = arrb[i] + i;
}
```

The loop now vectorizes.

Sometimes arrays are passed as pointers, declared in the function definition as pointers, and then accessed in the function definition as an array. This behavior is shown in Figure 8-17.

**Figure 8-17**
Potential aliasing of arrays passed as pointers

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

To vectorize this loop, compile the function with `-alias ptr_args`. You must also add a temporary variable to hold the loop's stop value, as shown in Figure 8-18.

**Figure 8-18**
Potential aliases and the `-alias ptr_args` option

```
parex( arra, arrb, n)
float *arra, *arrb;
int *n;
{
  int i, stop;
  stop = *n;
  for (i=0; i<stop; i++ )
    arra[i] = arrb[i] + i;
}
```

When the `-alias ptr_args` option is used, all array pointers declared as pointers in a function definition must be treated like arrays. If the pointer is modified, a syntax error will be generated when the code is compiled. For example, the code in Figure 8-19 generates a syntax error when it is compiled with the `-alias ptr_args` compiler option.

**Figure 8-19**
Incorrect syntax with `-alias ptr_args`

```
parex( arra, arrb )
float *arra, *arrb;
{
  int i;
  float both;
  for (i=0; i<100; i++ )
    *arra++ = *arrb++ = i;
}
```

## Caution

**The `-alias array_args` option and the `-alias ptr_args` option do not prevent aliasing of array parameters. When you use these options, the compiler assumes that all array parameters are distinct. If you pass overlapping arrays as parameters, these options hide the aliasing from the compiler. Undetected dependencies and errors can result.**

ANSI C declares that the ability of two array parameters to point to the same object in memory (that is, to become aliased to one another), is an obsolete feature. To ensure compatibility with future versions of ANSI C, do not write code that depends on the ability to pass overlapping or aliased arrays to a function.

## Preventing aliases

To prevent aliasing problems, avoid the use of pointers, structure pointers, and address operators whenever possible. Avoid code illustrated in Figure 8-20.

**Figure 8-20**
Poor code for vectorization

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

Instead, write code similar to that in Figure 8-21.

**Figure 8-21**
Vectorize arrays, not pointers

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

Or, use a `no_recurrence` pragma, as shown in Figure 8-22.

**Figure 8-22**
Use pragmas with pointers

```
divarrays()
{
  double *a, *b;
  int i;
  ...
# pragma _CNX no_recurrence
  for (i=0; i<1000; i++ ) /* vectorizes */
    a[i] = a[i] / b[i];
}
```

The code in Figure 8-23 does not vectorize because of pointers `a` and `b` and the variable `n`, which is used with the address operator.

**Figure 8-23**
Avoid using the & operator

```
alex4( a, b, n)
float *a, *b;
int n;
{
  int i;
  for (i=0; i<n; i++ )
    a[i] = b[i] + i;
  subl(&n);
}
```

The `no_recurrence` pragma tells the compiler to ignore the potential recurrence caused by the possible aliasing of `*a` and `*b`. The variable `n` still poses a problem, however. Because it is used with the address operator, `n` may be aliased with `a` and `b` . This potential alias prevents vectorization. To solve this problem, create a temporary variable `m` to replace the global `n` within the loop, as shown in Figure 8-24.

**Figure 8-24**
Replace global variables with temporary ones

```
int n;
...
alex4( a, b)
float *a, *b;
{
  int i;
  int m = n;
# pragma _CNX no_recurrence
  for (i=0; i<m; i++ )
    a[i] = b[i] + i;
  subl(&n);
}
```

## Conclusion

Aliasing, the assignment of alternate names to the same object, prevents the compiler from safely optimizing code. Aliasing can occur any time you use the pointer ( * ), address ( & ), or structure pointer ( -> ) operators. When a potential alias exists, the compiler does not automatically vectorize or parallelize the affected code. Affected code can be vectorized or parallelized using `no_recurrence` pragmas. (Take care to avoid actual recurrences.) To prevent aliasing problems, avoid the use of pointers wherever possible.

# Limits of optimization

# 9

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

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

A hidden alias is the most common cause of answers that change from one optimization level to another. In C, aliases most often occur through the use of pointers. Aliasing is possible any time you use a pointer ( * ), address ( & ), or structure pointer ( -> ) operator. Aliasing also occurs when you use array parameters, which, for efficiency, are always treated as pointers in C. Aliasing often prevents the compiler from 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 in Figure 9-1 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.

**Figure 9-1**
Hidden alias

```
#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 in Figure 9-2.

**Figure 9-2**
Hidden alias

```
% 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 in Figure 9-3.

**Figure 9-3**
Hidden alias

```
% cc -O2 -alias array_args -or none 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 nth 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 nth 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 nth 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.

The `main` function in Figure 9-4 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.

**Figure 9-4**
Apparent recurrence on `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 %d\n", j, arra[j]);
}
```

Because you know the value of `n` is 1, you can use the `no_recurrence` pragma, as shown in Figure 9-5. This pragma tells the compiler to ignore the apparent recurrence and vectorize the `i` loop.

**Figure 9-5**
Proper use of `no_recurrence`

```
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 in Figure 9-6, is inappropriate. The compiler warns you of the dependency but parallelizes the loop. Because of the forward dependency, the parallel code can produce incorrect results.

**Figure 9-6**
Improper use of `force_parallel`

```
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, which are discussed more fully in Chapter 3, are a special class of recurrence that the compiler knows how to vectorize. An apparent recurrence can prevent the compiler from vectorizing a loop containing a reduction. The loop shown in Figure 9-7 does not vectorize because of an apparent dependency between the reference to `arra[i]` and the assignment to `arra[arrx[j]]`.

**Figure 9-7**
Reduction and apparent recurrence

```
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];
  }
```

A `no_recurrence` pragma placed before the `j` loop tells the compiler that the indirect subscript does not cause a true recurrence. This pragma also tells the compiler to ignore the reduction on `arra[i]`. The compiler generates normal vector load, add, and store instructions for the first statement. The resulting code will run faster but produces incorrect answers.

To solve this problem, distribute the `j` loop, isolating the reduction from the other statements, as shown in Figure 9-8.

**Figure 9-8**
Reduction and apparent recurrence resolved

```
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];
```

The apparent recurrence is removed, and both loops vectorize. This problem occurs only if the reduction and the apparent recurrence involve the same variable. If the reduction and the apparent recurrence involve different variables, as in Figure 9-9, both reduction and recurrence are handled correctly without your intervention.

Figure 9-9
Reduction and recurrence on separate variables

```
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 trip, the loop can produce incorrect answers or cause the program to abort. This error can occur when a variable used as an iteration value is accidentally set to zero. If 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. Figure 9-10 shows two loops that iterate by zero.

**Figure 9-10**
Two loops iterating 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. Figure 9-11 shows an example.

**Figure 9-11**
Conditional vectorization

```
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 in Figure 9-12 contains an example of a loop in which the induction variable is replaced.

**Figure 9-12**
Test replacement

```
#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 *trip 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 trip count can cause the loop limit value (stride*trip+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*trip+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*trip+start must be not exceed $2^{32}$.) When a loop has a positive stride and the trip count overflows its memory location, the loop executes only once because the limit is now negative and the termination test fails.

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

Because we know that the largest allowable value for stride*trip+start is $2^{31}$-1 and the start value for the loop in Figure 9-12 loop is 38, the maximum trip count for the loop using the equation (stride*trip+start)/stride. To solve this for trip, 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 trip 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 trip 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. Figure A-1 shows such a loop.

**Figure A-1**
Pattern matching `if` tests

```
int a[100];

int max()
{
  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. Figure A-2 shows examples of patterns the compiler matches.

**Figure A-2**
Pattern matching examples

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

int max(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;

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

## Conversion elimination

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

**Figure A-3**
Eliminating type conversion

**Original Loop**

```
float a[100];

int max()
{
  int i;

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

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

**Figure A-4**
Type conversion eliminated

**Optimized Loop**

```
float a[100];

int max()
{
  int i;
  float real_i;

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

# CONVEX C intrinsics

# B

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

## What are intrinsics?

In the context of CONVEX C, there are two types of intrinsics: instructions and functions. Intrinsic instructions are commands in a CONVEX instruction set. For example, an intrinsic instruction in a CONVEX C100 Series architecture is add; an intrinsic instruction in a CONVEX C200 Series architecture is sqrt. While the add instruction is in the instruction set for a C200, the sqrt instruction is not implemented in the C100.

Intrinsic functions are useful for these reasons:

- ❑ They require less function overhead.
- ❑ They execute faster than non-intrinsic functions.
- ❑ They do not inhibit vectorization of loops.

There are some disadvantages of intrinsic functions:

- ❑ 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 are C-callable functions that can use intrinsic instructions. Intrinsic functions are used by default in the extended compatibility mode of the compiler.

To access these functions in the strict and standard compatibility modes, include `-U__NO_INLINE_MATH` on the `cc` command line. If you want to use intrinsic functions in the backward-compatible mode, include `-D__INLINE_MATH` on the `cc` command line.

Figure B-1 shows a C program that calls sqrt, an ANSI C function.

**Figure B-1**
Calling an intrinsic function

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

   y = sqrt(x);
   (void) printf("%d 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 C2 machine, the intrinsic sqrt 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.

Figure B-2 shows two macro definitions that call math intrinsic functions.

**Figure B-2**
math.h 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.

For example, when the program in Figure B-3 is compiled in the extended compatibility mode and executed on a C2 machine, an incorrect result is printed.

**Figure B-3**
Intrinsic function that signals no error on C2

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

  int x = -4;
  int y;

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

  return(0);
}
```

When this program is compiled for a C1 machine using the `-tm c1` command line option, and executed, the domain error is caught because the implementation of the sqrt intrinsic function on a C1 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 Figure B-4, the compiler removes the conditional statement because it does not realize that the acos function can modify errno.

**Figure B-4**
Code removed during optimization

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

At optimization level -O2, the code in Figure B-4 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:

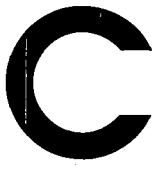
- __NO_INLINE_BINT
- __NO_INLINE_CTYPE
- __NO_INLINE_MATH
- __NO_INLINE_SIGNAL
- __NO_INLINE_STDIO
- __NO_INLINE_STDLIB
- __NO_INLINE_STRING
- __NO_INLINE_TIME

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 Program Status Word when your program is started. You must include a signal handler that determines what caused the signal and then take appropriate actions. Further detail of this approach is beyond the scope of this appendix. Refer to *CONVEX Architecture Reference*, Chapter 3, "Register Sets," for more information on the Program Status Word and the bits associated with intrinsic instruction errors.

# Compiler pragmas



This appendix describes CONVEX C compiler pragmas that affect optimization. Some pragmas listed here provide the compiler with information that it cannot deduce on its own, while others instruct the compiler to override default conditions that control optimization, vectorization, or parallelization. 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_recurrence`
- `no_side_effects`
- `no_vector`
- `prefer_parallel`
- `prefer_parallel_ext`
- `prefer_vector`
- `pstrip`
- `scalar`
- `select`
- `synch_parallel`
- `unroll`
- `vstrip`

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

**Figure C-1**
Restrictions on pragma use

| | force_parallel | force_parallel_ext | force_vector | no_parallel | no_vector | prefer_parallel | prefer_parallel_ext | prefer_vector | pstrip | scalar | select | synch_parallel | unroll | vstrip |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| force_parallel | | X | X | X | | X | X | X | | X | X | X | X | X |
| force_parallel_ext | X | | | X | | X | X | X | X | X | X | X | X | |
| force_vector | X | | | | X | X | X | X | X | X | X | X | X | |
| no_parallel | X | X | | | | X | X | | X | X | X | X | | X |
| no_vector | | | X | | | | | X | | X | X | | | X |
| prefer_parallel | X | X | X | X | | | X | X | | X | X | X | X | X |
| prefer_parallel_ext | X | X | X | X | | X | | | X | X | X | X | X | |
| prefer_vector | X | X | X | | X | X | | | X | X | X | X | X | |
| pstrip | | X | X | X | | | X | X | | X | | X | X | X |
| scalar | X | X | X | X | X | X | X | X | X | | X | X | | X |
| select | X | X | X | X | X | X | X | X | | X | | | X | |
| synch_parallel | X | X | X | X | | X | X | X | X | X | | | X | X |
| unroll | X | X | X | | | X | X | X | X | | X | X | | X |
| vstrip | X | | | X | X | X | | | X | X | | X | X | |

A 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 code in Figure C-2 shows the use of the tasking pragmas.

**Figure C-2**
Use of `begin_tasks`, `next_task`, and `end_tasks`

```
#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 in Figure C-3.

**Figure C-3**
Use of `force_parallel`

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

## 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_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 Figure C-4, if `j` is positive, no recurrence exists.

**Figure C-4**
Use of `no_recurrence`

```
#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.

**Figure C-5**
Use of
`no_side_effects`

```
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_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_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.

---

### 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.

## 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.

## scalar

The `scalar` pragma prevents the loop following the pragma from being vectorized, parallelized, distributed, or interchanged. This pragma does not prevent other loops from being vectorized or parallelized.

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

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

In Figure C-6, the compiler normally interchanges the `i` and `j` loop so that elements of `a`, `b`, and `c` are accessed contiguously. The `scalar` pragma ensures that the loop with the greater iteration count is retained as the innermost loop.

**Figure C-6**
Use of `scalar`

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

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

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

In Figure C-7, neither iteration count is sufficient to warrant vectorizing the loops.

**Figure C-7**
Use of `scalar` with small iteration count

```
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 loop in Figure C-8 might run faster parallelized and synchronized than if it is partially vectorized and the recurrence placed in a scalar, nonparallel loop.

**Figure C-8**
Use of `synch_parallel`

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

int calc(int n, int m)
{
  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 is effective only if you specify the `-O2` or `-O3` compiler option. `unroll` reduces loop overhead by replicating the body of the loop following the pragma. Complete unrolling is performed if the loop trip count is less than five and if the loop is very simple. Partial unrolling is performed on loops that cannot be vectorized, because the loop trip count is greater than five.

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

### 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

```
#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.*

## 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 vector operation in Figure D-1.

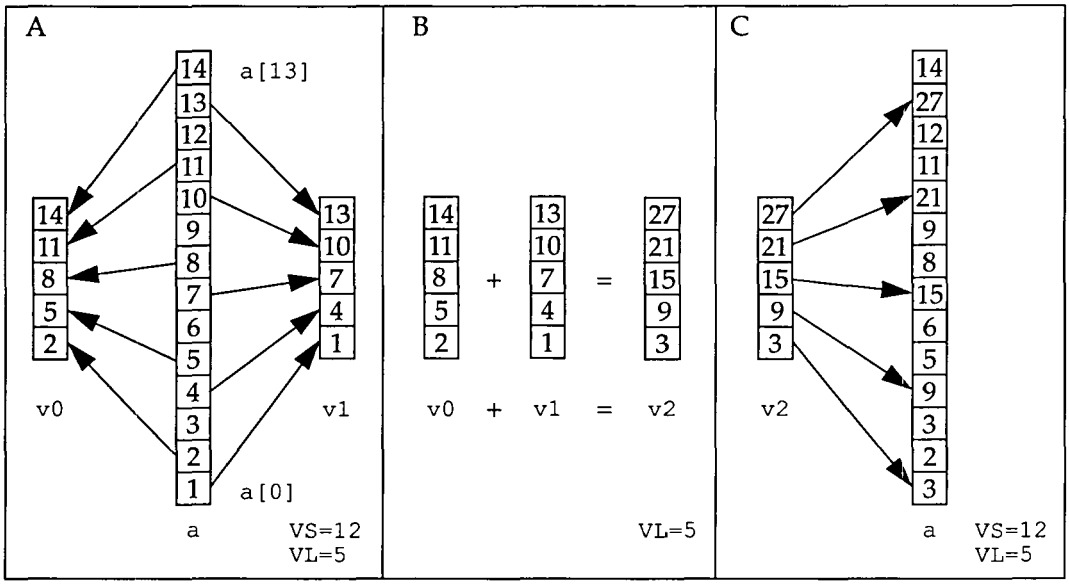
**Figure D-1**
Vector operation

```
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 D-2 shows the vector operations on array a.

**Figure D-2**
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*.

## Vector load

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

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

**Figure D-3**
Vector load example 1

| 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`.

Figure D-4 shows another example of vector load.

**Figure D-4**
Vector load example 2

| 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 in Figure D-5.

**Figure D-5**
Scalar extend

| 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.

Figure D-6 shows an example of vector store.

**Figure D-6**
Vector store example 1

| 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.

Figure D-7 shows another example of vector store.

**Figure D-7**
Vector store example 2

| 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.

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

**Figure D-8**
Vector addition

| 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 in Figure D-9 computes the product of a vector and a scalar.

**Figure D-9**
Vector multiplication

| 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 example in Figure D-10 generates a sum reduction.

**Figure D-10**
Vector 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 Figure D-11, a dot-product operation requires the sum of a series of products.

**Figure D-11**
Dot-product operation

```
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 in Figure D-12 .

**Figure D-12**
Dot-product vector code

```
ld.w    #0,s0
ld.w    #100,VL
ld.w    #4,VS
ld.w    a,v1
ld.w    n,v2
mul.w   v2,v1,v0
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 in Figure D-13.

**Figure D-13**
Vector maximum

| C | Assembly Language |
|---|---|
| `int a[100],b[100];` | `ld.w #100,VL` |
| | `ld.w #4,VS` |
| `for( i=0;i<100;i++ )` | `ld.w b,v0` |
| `  if(a[i] <= b[i])` | `ld.w a,v1` |
| `    ...` | `le.w v1,v0` |
| | `...` |

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

## Vector operations under mask—C200

Only C200 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 C200 Series computer.

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

In this mode, the bit of the VM register corresponding to each vector element is examined to either enable or disable 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 C code in Figure D-14 is an example of using operations under mask.

**Figure D-14**
Vector 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 C200 Series computers. This instruction uses the VM register to extract elements from one vector register and selectively place the elements in another vector register. Either zeros or ones of VM can be used by specifying the instruction's `.f` or `.t` (false or true) version, respectively. Only elements with the corresponding VM bit set (clear for `.f`) are loaded into the destination vector. Other elements in the destination vector corresponding to clear VM bits (set for `.f`) are skipped over. The `expand` instruction creates a destination vector with VL elements, including a number of elements of the source vector equal to the number of bits set (or clear) in the VM register.

### Examples

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 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 loop in Figure D-15 cannot be determined until the program is executed.

**Figure D-15**
Conditional test and masking

```
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 code in Figure D-16 uses indirect addressing.

**Figure D-16**
Indirect array addressing: C code

```
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 in Figure D-17.

**Figure D-17**
Indirect array addressing: assembly code

```
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.)

These are the steps used to compute `b[ib[i]] + 1`:

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`.

These are the steps used to store the result of `b[ib[i]]+1` in `a[ia[i]]`:

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.

# Glossary

# E

**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.

**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 );
}
```

**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

**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.*)

**execution stream**

A series of instructions that a CPU executes

**fine-grained**

See *granularity.*

**functional unit**

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

**gather**

A vector operation that loads values from an array into a vector register. The operands of this operation come from various locations in an array.

**granularity**

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

**hoist**

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

**interleaved memory**

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

**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 vector stride in the inner loop.

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

**mask, vector** — A 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

**oversubscript** — An array reference that falls outside declared bounds

**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

**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.

**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.

**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.

**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

**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*.

# Bibliography

F


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.

# Index

## E



## F



## G



## H



## I

## J



## L



## M

## N



## O



## P

## R

## S



## T

## U

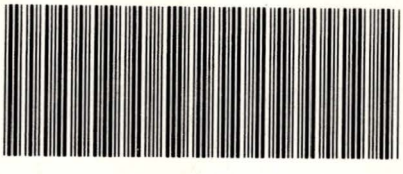


## V

## W



## Z



Order Number
DSW-089

Document Number
720-001130-202