Vector Processing in Cray C++ Programs [C]

Because vector operations cannot be expressed directly in Cray C++, the compiler must be capable of vectorization, which means transforming scalar operations into equivalent vector operations. The candidates for vectorization are operations in loops and assignments of structures. This chapter describes the criteria used by the compiler to determine if candidates are vectorized. An extended example begins on page Section C.14, page 193.

Unless otherwise noted, vectorization is described at the default level, corresponding to the `-h vector2` option. The `-h vector1` option disables vectorization of search and reduction loops, and `-h vector0` disables all automatic vectorization. The `-h vector3` option enables more aggressive dependence analysis and restructuring at a cost in increased compile time. When the use of this option results in additional vectorization, it is noted in this chapter.

The `-O3` option implies `-h vector3`, `-h inline3`, and `-h task3`. Vectorization options and directives are further discussed beginning on page Section C.10, page 184.

C.1 Equivalent results

Many vectorization criteria are based on the rule that the vectorized program must produce the same results as the scalar program. To increase the opportunities for vectorization, the Cray C++ compiler relaxes this requirement in certain cases, as follows:

- Reductions are reordered. For example, a scalar loop summing the elements of an array is transformed into a vector reduction even though the elements are summed in a different order. You must identify the cases in which this leads to an unacceptable difference in the sum.
- Signals arising from floating-point exceptions can be handled at different points in the vectorized program.
- For loops in which the value zero for an unknown stride would require special treatment, the stride is assumed to be nonzero.

```
/* Behavior undefined if k == 0. */
for (i=0, j=0; i<; i++, j+=k) {
    a[j] = a[j] + b[i];
}
```

C.2 Innermost loops

With default vectorization, a `for`, `while`, `do...while`, or `goto` loop is a candidate for vectorization only if it is an innermost loop.

```
for (i=0; i<n; i++) {   /* Not vectorized */
      a[i] = 0;  /* Not in innermost loop */
      for (j=0; j<m; j++) { /* Vectorized */
            b[i][j] = 0;
      }
}
```

If you manually split the loop on `i`, and do the assignments in separate loops, both vectorize, as in the following example. (With `-h vector3`, the compiler performs this restructuring automatically.)

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

C.3 Vectorizable expressions

Most operators and data types are vectorizable, in the sense that their presence in a loop does not prevent its vectorization. The following sections describe exceptions to this general rule.

C.3.1 Function calls

Function calls do not vectorize except for certain intrinsic and library functions and call-by-register functions.

```
extern double a[], b[], c[];

double f(double x) {
    return 2.0 * x;
}

void g(int n) {
    int i;
    for (i=0; i<n; i++) {/* Not vectorized */
        a[i] = f(b[i]);
    }
    for (i=0; i<n; i++) {    /* Vectorized */
        #pragma _CRI inline f
        b[i] = f(c[i]);
    }
}
```

As in this example, the compiler can be directed to replace the function call with an inline expansion. For more information on inlining, see the *Cray C/C++ Reference Manual*, publication SR–2179. If the resulting loop meets the other criteria described in this section, it is vectorized.

```
#include <math.h>
void f2(int n) {
    int i;
    for (i=0; i<n; i++) {    /* Vectorized */
        a[i] = sqrt(b[i]);
    }
}
```

For calls to the `sqrt` function to be vectorized, the file must include the header `<math.h>` (or a compatible declaration of `sqrt`), and must not be compiled in strict conformance mode (specified by the `-h stdc` option) or with the `-h matherror=errno` option (which ensures full domain- and range-checking in library functions.)

```
#pragma _CRI vfunction vf
extern double vf(double);

void f3(int n) {
    int i;
    for (i=0; i<n; i++) {    /* Vectorized */
        b[i] = vf(c[i]);
    }
}
```

The `vfunction` directive identifies function `vf()` as a call-by-register function that must be written in assembly language or Fortran. For more information on the `vfunction` directive, see the *Cray C/C++ Reference Manual*, publication SR–2179.

C.3.2 Structure assignments

A loop cannot be vectorized if it contains both assignments to a structure and accesses to the structure's components. However, neither type of operation alone inhibits vectorization. To understand the vectorization messages for loops containing structure assignments, note the following details.

If an assignment to a structure is not in a loop, and if the structure spans more than two words, then the structure value is copied with vector operations.

A loop that assigns the values of one array of structures to another is vectorized in one of the two following ways:

- For structures larger than sixteen words, each structure value is copied with a separate vector operation. In this case the vector length is the word size of the structure, and the stride is one.
- For smaller structures, the structures are divided into words, and there is a separate vector copy for all the words at each fixed word offset. In this case the vector length is the number of iterations specified for the loop, and the stride is the word size of the structure.

C.4 Recurrences

A *recurrence* is an expression within a loop that depends on a value calculated in a previous iteration of the loop. Most recurrences cannot be vectorized, because vectorization causes the calculations for successive iterations of a loop to overlap, and a value calculated in one iteration is not available to succeeding iterations. Thus, the next example cannot be vectorized because each iteration (after the first) requires a value calculated in the immediately preceding iteration.

```
b[0] = a[0];
for (i=1; i<; i++) { /* Not vectorized */
     b[i] = b[i-1] + a[i];
}
```

With a few exceptions, the Cray C++ compiler must determine that a loop does not contain a recurrence, before it can vectorize the loop. Three types of recurrences that can be vectorized are reductions, recurrences with an identifiable threshold, and some gather/scatter recurrences. These are discussed below.

The compiler must be conservative and assume the existence of recurrences in code that are beyond its ability to analyze. In such cases, if you know that there is no recurrence, you can assert this by using an `ivdep` directive or `-h ivdep` option. If the potential recurrence is the result of possible aliasing through pointers, you can also make use of restricted pointers. Restricted pointers are described on page Section C.9, page 180.

C.4.1 Reductions

Some reductions, such as summing all the elements of an array, can be vectorized. In particular, a reduction can be vectorized if the operator is one of the arithmetic operators `+`, `-`, `*`, or `/`, or one of the bitwise operators `&`, `|`, or `^`.

```
sum = 0;
for (i=0; i<n; i++) { /* Vectorized */
     sum += a[i];
}
```

C.4.2 Output dependence

The term recurrence is also used (loosely) to describe an *output dependence*. If there are multiple assignments to the same array element within a loop, and the assignments were reordered by vectorization, then wrong values might be left in the array.

```
k[0] = 1; k[1] = 0;  /* ... */
for (i=0; i<n; i++) { /* Not vectorized */
     a[i] = 3;
     a[k[i]] = 5;
}
```

After this loop, the values of `a[0]` and `a[1]` are `5` and `3`, respectively. If the file containing this code is compiled with the option `-h ivdep`, the loop vectorizes, and both values are (incorrectly) `5`. This shows the care that must be taken with this option.

C.4.3 Gather/scatter

The previous example illustrated how a scatter can create a potential output dependence. A gather or a scatter can also create a potential true recurrence.

```
for (i=0; i<n; i++) { /* Not vectorized */
     a[i] = a[k[i]];
}
```

The next example, on the other hand, vectorizes despite potential recurrences from a gather and a scatter into the same array. The key point is that the same array of indices is used for both the gather and scatter.

```
extern int a[], b[], k[];

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

If the index array `k` has no repeated values, there are no recurrences in the loop, and asserting this with an `ivdep` directive provides the best performance. Without the directive, the loop still vectorizes, with least loss of performance if `k` has few repeated values. At the default optimization level, the compiler does not attempt to vectorize a loop with more than two instances of such gather/scatter recurrences. This limit can be disabled by the `-h vector3` or the `-h aggress` option).

Indirect addressing can also be expressed using an array of pointers.

```
extern int *p[];

for (i=0; i<n; i++) {     /* Vectorized */
     *p[i] += 1;
}
```

C.4.4 Recurrence threshold

The *threshold* of a recurrence is the number of iterations that occur before a newly calculated value is required. A recurrence with a threshold greater than 64 (or 128 on CRAY C90 systems) vectorizes without any loss of performance on a Cray Research system. Recurrences with smaller thresholds can still profitably be vectorized, with a vector length equal to the threshold. In the following example, the vector length is 30:

```
for (i=30; i<n; i++) { /* Vectorized */
     a[i] += a[i-30];
}
```

The preceding code generates the following message:

```
=> cc -hreport=v -c t1.c
cc-8072 cc: VECTOR File = t1.c, Line = 3
  Loop starting at line 3 was vectorized with
  a vector length of 30.
```

The value of the threshold need not be known at compile time. In the following example, the vector length is the value of `k`, if 0 < `k` < 64, and 64 (128 on CRAY C90 systems) otherwise. If you compile this code by using the `-h report=v` option, the compiler states that the loop is "vectorized with a computed maximum safe vector length" (see Section C.5, page 169).

```
for (i=k; i<n; i++) {        /* Vectorized */
     a[i] += a[i-k];
} /* ... with computed safe vector length */
```

C.5 Computed safe vector length

To offer better performance on some loops that involve pointers, or other potential recurrences, the compiler can generate code to compute, at run time, a safe vector length for the vector operations. For example, in the following loop, the initial values of `p` and `q` are used to compute a safe vector length for the assignment.

```
void f (int n, int *p, int *q) {
    int i;
    for (i=0; i<n; i++) {  /* Vectorized */
        *p++ = *q++;
    }
/* ... with computed safe vector length */
}
```

This type of vectorization can be viewed as a form of *conditional vectorization*, because the computed safe vector length might turn out to be 1, in which case the loop is, in effect, executed as a scalar loop. The length computation involves nontrivial run-time overhead, and if the safe vector length does turn out to be 1, the vector code executes more slowly than scalar code would. For this reason, each loop that is conditionally vectorized should be examined to see if it can be asserted to be either a scalar loop with the `novector` directive or to be an unconditional vector loop with `ivdep`.

C.6 `-h vector3` capabilities

The `-h vector3` option enables a more extensive analysis and restructuring of loops, subject to the assumptions and limitations of Autotasking, discussed in Chapter 8, page 127.

C.6.1 Reordering expressions

At the default optimization level, vectorization attempts to substitute vector operations for scalar operations in the original order. This approach does not allow the following example to be vectorized, but with `-h vector3` the order of the statements is reversed, and the loop can then be vectorized.

```
for (i=n-1; i>0; i--) {     /* Vectorized */
      a[i] = b[i];          /* if vector3 */
      c[i] = a[i-1] + 1;
}
```

C.6.2 Partial vectorization

A mixture of vectorizable and unvectorizable operations in a single loop inhibits any vectorization of that loop at the default optimization level. With `-h vector3`, in some cases, the loop is split into two or more loops to permit vectorization of as many operations as possible. Because the first line in the body of the following loop involves a recurrence, it is split off into a separate loop, leaving the second and third lines as the body of a vectorizable loop.

```
#include <math.h>

/* Loop partially vectorized if vector3 */
for (i=0; i<n; i++) {
     a[i] += a[i-1];
     b[i] = a[i] * 2.0;
     c[i] = fabs( b[i] );
}
```

You can tell that a loop was split by requesting vectorization messages with `-h report=v`. Each split loop is identified by the line number on which it begins and the message:

```
A loop was split into 2 loops to allow
 for partial vectorization.
```

All the split loops continue to be identified in messages by the line number of the original loop. Thus two or more (often contradictory) vectorization messages can appear to refer to the same loop, but each message actually refers to one of the split loops. The preceding example gets the following messages:

```
Loop starting at line 6 was not vectorized.
 It contains a recurrence on "a" at line 7.
Loop starting at line 6 was vectorized.
```

If scalar optimization messages are also requested, there are two additional messages:

```
Loop starting at line 6 was unrolled 16 times.
Loop starting at line 6 was bottom loaded.
```

For both the vector and scalar messages, the first message applies to the split loop containing the recurrence, and the second to the split loop containing the rest of the original loop.

When the expressions in a loop are reordered and then partially vectorized, it might be difficult to interpret messages.

C.6.3 Dependence analysis

The more extensive dependence analysis enabled by `-h vector3` includes normalization of index expressions and forward substitution of constant values.

These techniques succeed in cases for which the dependence analysis at the default level fails. Examples include index expressions that are, or involve, a loop invariant, index expressions that can be algebraically simplified, and references to both scalar and array members of the same structure.

```
for (i=1; i<n; i+=2) {       /* Vectorized */
     a[i] = a[2];           /* if vector3 */
}

k = 1;
for (i=0; i<n; i++) {       /* Vectorized */
     a[i] = a[i+k];         /* if vector3 */
}

for (i=0; i<n; i++) {       /* Vectorized */
     a[i] = a[2*i+1];       /* if vector3 */
}

extern struct {int a[100], b;} t;

for (i=0; i<n; i++) {       /* Vectorized */
     t.a[i] = t.b;          /* if vector3 */
}
```

In each of these examples, the loop is vectorized with computed safe vector length at the default vectorization level.

C.6.4 Alias analysis

With `-h vector3`, the objects into which pointers point are tracked, where possible, and better advantage is taken of the facts that each object allocated with `malloc` is disjoint from all other objects, and that two objects with different types are disjoint.

```
#include <stdlib.h>

extern double a[];
double *p = malloc(n*sizeof(int));

for (i=0; i<n; i++) {       /* Vectorized */
      p[i] = a[i];          /* if vector3 */
}
extern struct s1 { double x[100]; } *p1;
extern struct s2 { double x[100]; } *p2;

for (i=0; i<n; i++) {       /* Vectorized */
      p1->x[i] = p2->x[i];  /* if vector3 */
}
```

In each of these examples, the loop is vectorized with computed safe vector length at the default vectorization level.

With `-h vector3`, restrict qualifiers on pointers are analyzed in more contexts. Implementation limits are discussed on page Section C.9.7, page 183.

C.6.5 Pattern matching

With `-h vector3`, if a loop matches a pattern from a predefined database, it will be replaced by a call to a library function. There are patterns for selected Scientific Library functions, including `CGEMM`, `SGEMM`, `CGEMV`, `SGEMV`, `CGERC`, `CGERU`, and `SGER` (see the *Scientific Libraries Reference Manual*, publication SR–2081).

```
/* With vector3, replaced by SGEMV: */
for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
        a[i] += c[i][j] * b[j];
    }
}
```

C.6.6 Loop inversion

When an inner loop cannot vectorize because of cross-iteration dependencies, the compiler attempts to vectorize the outer containing loop. This inversion allows vectorization of outer loops or tasking of inner loops. For example:

```
for ( i = 0; i < n; i++ ) {
   a[i] = b[i] * 2;
   for ( j = 0; j < m; j++ ) {
      c[i][j+1] = c[i][j] + a[i];
   }

}
```

In the preceding code, the `i` loop is vectorized as follows:

```
for ( i = 0; i < n; i++ ) {
   a[i] = b[i] * 2;
}
for ( j = 0; j < m; j++ ) {
   for ( i = 0; i < n; i++ ) {
      c[i][j+1] = c[i][j] + a[i];
   }
}
```

Inversion for tasking is done in similar fashion.

C.6.7 Loop collapse

Loop collapse enhances the performance of the generated code by improving load balancing (of a task and vector loop pair), reducing the inner loops' overhead, and increasing the vector length of a vector loop. For example:

```
int     a[5][5][5];
for ( i=0; i < 5; i++ ) {
   for ( j=0; j < 5; j++ ) {
      for ( k=0; k < 5; k++ ) {
         a[i][j][k] = 0;
      }
   }
}
```

The preceding loops collapse to the following:

```
for ( k=0; k < 125; k++ ) {
   a[0][0][k] = 0;
}
```

Loop collapse for tasking is done in similar fashion.

C.6.8 Loop fusion

Loop fusion enhances the performance of the generated code by combining loop overhead and by expanding loop body size to expose more block optimization opportunities.

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

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

/* loop #2 */
for ( i=1; i < n; i++ ) {
d[i] = a[i] * 2;
}
```

The preceding loops collapse to the following:

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

The following conditions prevent loop fusion:

- Loops with reductions will not be fused.
- Vector loops will not be fused with scalar loops.
- Introduction of loop-carried dependencies disables loop fusion.

C.7 Search loops

A *search loop* is a loop with one or more conditional `break` statements or other early exit statements. For example:

```
for (i=0; i<n; i++) { /* Vectorized */
     if (a[i] <b[i])
         break;
}
```

An explicit upper bound on the number of iterations is not a requirement for vectorization. The following variations are equivalent to the previous example, except that they lack the upper bound.

```
for (i=0; ; i++) {        /* Vectorized */
     if (a[i] < b[i])
         break;
}
```

```
i = 0;
while ( a[i] >= b[i] ) {  /* Vectorized */
        i += 1;
}

i = -1;
do {                      /* Vectorized */
     i += 1;
} while ( a[i] >= b[i] );
```

C.7.1 Indirect addressing

Vectorization is inhibited if an exit condition involves indirect addressing.

```
for (i=0; i<; i++)  {  /* Not vectorized */
     if (a[b[i]])
         break;
}
```

An operand range error could occur during execution of the vectorized version of the loop if values in `b` beyond the `break` index are not appropriate indices for `a`. You can direct the compiler to ignore this possibility, and vectorize the loop, through use of the `ivdep` directive.

C.7.2 Floating-point condition

Vectorization is not inhibited if an exit condition involves floating-point arithmetic.

```
extern double d[];
for (i=0; i<; i++)  {  /* Vectorized */
     if (d[i] <0.0)
         break;
}
```

A floating-point exception could occur during execution of the vectorized version of the loop if values in `d` beyond the `break` index are not legal floating-point values. In practice, this is rarely a problem, but in the exceptional cases, you can inhibit vectorization with either a `novector` or `novsearch` directive immediately preceding the loop in question or with the `-h novsearch` option. The latter inhibits vectorization of all search loops and is useful in diagnosing this type of problem.

C.7.3 Placement of exits

With few exceptions, vectorization is inhibited if all exits from a loop do not appear at the top of the loop body, and in particular before any conditional or unconditional assignments.

```
for (i=0; i<n; i++) { /* Vectorized */
     if (a[i]) {
         if (b[i])
             break;
         if (c[i])
             break;
     }
     d[i] = 0;
}

for (i=0; i<n; i++) { /* Vectorized */
     if (a[i] && (b[i] || c[i]))
         break;
     d[i] = 0;
}

for (i=0; i<n; i++)  { /* Not vectorized */
     if (a[i])
         d[i] = 0;
     if (b[i])
         break;
}

for (i=0; i<n; i++)  { /* Not vectorized */
     d[i] = 0;
     if (a[i])
         break;
     if (b[i])
         break;
}
```

C.7.4 Last exit is special

In the two previous examples, a condition that limits how many values of `d[i]` are modified is computed *after* the assignment. Vectorizing such loops requires more rearrangement of the operations in the loop bodies than the compiler currently undertakes. However, the last conditional exit from a loop is treated as a special case, which can, in effect, be moved above preceding unconditional assignments; this allows the loop to vectorize, as in the following two examples:

```
for (i=0; i<n; i++)  { /* Vectorized */
     c[i] = 1;
     if (b[i])
         break;
}

for (j=0; j<n; j++)  { /* Vectorized */
     if (a[j])
         break;
     c[j] = 1;
     if (b[j])
         break;
}
```

C.7.5 Exit dependence

The condition for the exit must not depend upon values computed (in a previous iteration) in the portion of the loop following the exit. For example:

```
for (i=0; i<n; i++) { /* Not vectorized */
     if (a[i] > 100)
         break;
     a[i+1] += b[i];
}
```

Although more complicated loops might present a challenge, it is relatively easy to restructure the previous example so that it vectorizes.

```
if (a[0] > 100) {
    i = 0;
} else {
    for (i=1; i<n; i++) { /* Vectorized */
         a[i] += b[i-1];
         if (a[i] > 100)
          break;
    }
    if (i == n)
        a[n] += b[n-1];
}
```

C.8 Loops with branches

Search loops involve branches out of a loop. The following sections discuss branches into and within a loop.

C.8.1 Branch into loop

With default vectorization, if any statement branches into the loop from outside it, the loop is not vectorized. Example:

```
i = 0;
if (cond)
    goto inside;
for (; i<n; i++) { /* Vectorized if vector3 */
    a[i] = 0;
inside:
    b[i] = 0;
}
```

Such a loop is eliminated as a candidate for vectorization at an early stage in compilation; therefore, it is not mentioned in the vectorization messages. Such a branch can usually be eliminated at the cost of duplicating some code. With `-h vector3`, this example is restructured and vectorized.

C.8.2 Backward branch

If a loop contains a backward branch, it cannot be vectorized. A backward branch actually defines a loop, and the compiler attempts to vectorize only that innermost loop.

```
for (i=0; i<n; i++) { /* Not vectorized */
       a[i] = 0;
again: b[i] -= 1;         /* Vectorized */
       if (b[i] > 10) goto again;
}
```

C.8.3 Forward branch

Except for `switch` statements with four or more cases, forward branches within a loop permit vectorization.

```
for(i=0; i<n; i++) { /* Vectorized */
    switch ( e[i] ) {
        case 0:   a[i] = b[i];
                  break;
        case 1:   c[i] = d[i];
                  break;
        default: a[i] = c[i] = 0;
                  break;
    }
}
```

The previous example also vectorizes if the branches are expressed with `goto` and `continue` statements, as shown in the following example:

```
for(i=0; i<n; i++) { /* Vectorized */
    if ( e[i] != 0 ) goto L1;
    a[i] = b[i];
    continue;
L1: if ( e[i] != 1 ) goto L2;
    c[i] = d[i];
    continue;
L2: a[i] = c[i] = 0;
}
```

C.9 Restricted pointers

Cray C++ offers the `restrict` type qualifier language extension for asserting the absence of aliasing through pointers. A pointer with restrict-qualified type is termed a restricted pointer. In brief, the compiler might assume that, like an array name, a restricted pointer provides, at the point of its declaration, the only means of designating an array object that is disjoint from any other object. Examples of the use of `restrict` to enable the compiler to vectorize loops with pointers are given in this section. You can also obtain restricted pointers by using the `-hrestrict=a` and `-hrestrict=f` options of the `CC` command.

Note that the compiler ignores uses of `restrict` that it currently cannot vectorize. Some cases that are ignored are discussed in Section C.9.7, page 183.

C.9.1 Function parameters

If a function has pointer parameters that, in each call, point into disjoint (array or non-array) objects, the usage of these parameters can be asserted to the compiler by declaring the pointer parameters to be restrict-qualified. This will eliminate potential aliasing through those pointers as an obstacle to vectorization of the loops in the function, and might allow improved scheduling of scalar code, as well.

Example:

```
void f1 (int n, int * restrict p,
                int * restrict q) {
   int i;

   /* Compiler may analyze dependences  */
   /* as if the following two lines     */
   /* were present:                     */
   /*   p = malloc(n * sizeof(int));    */
   /*   q = malloc(n * sizeof(int));    */

   for (i=0; i<n; i++) { /* Vectorized  */
        p[i] = q[i];
    }
}
```

This operation will also vectorize if the restricted pointers are incremented.

```
void f2 (int n, int * restrict p,
                int * restrict q) {
   int i;
   for (i=0; i<n; i++) { /* Vectorized */
        *p++ = *q++;
    }
}
```

C.9.2 Alternative syntax

A function parameter declared as:

```
Type p[restrict]
```

is interpreted as though it were:

```
Type * restrict p
```

This alternative syntax is an extension of the interpretation, required by the C Standard, of a parameter declaration of the form `Type p[]` as though it were `Type *p`. This syntax is especially convenient for parameters that are pointers to variable length arrays, because it allows a more array-like syntax.

Example:

```
void f(int n, double a[restrict][n]);
```

can be used instead of:

```
void f(int n, double (* restrict a)[n]);
```

C.9.3 File scope

Restricted pointers are also useful when declared at file scope. In the next example, `restrict` is used to assert that the global pointer `p` points into a unique array object, which in this case is obtained from a `malloc()` call.

```
#include <stdlib.h>

int * restrict p;

void alloc_p (int n) {
   p = malloc(n * sizeof p[0]);
}
```

C.9.4 Block scope

A restricted pointer declared in the outermost block of a function can be used to point to an array allocated for use within the function. Continuing the previous example:

```
extern int * restrict p;

void modify_p (int n) {
   int * restrict q = malloc(n * sizeof q[0]);
   int i;
   for (i=0; i<n; i++) {    /* Vectorized */
      q[i] = p[i];
   }
   for (i=0; i<n-1; i+=2) { /* Vectorized */
      p[i]   = q[i+1] + q[i];
      p[i+1] = q[i+1] - q[i];
   }
   free(q);
}
```

C.9.5 Unrestricted pointers

Like array names, restricted pointers are not immune from aliasing through unrestricted pointers. At the point of its declaration, a restricted pointer might be assumed to provide the only means of designating an array object that is disjoint from any other object, but a restricted pointer's value can be assigned to one or more unrestricted pointers. Such an assignment might occur in an assignment expression or in the assignment of an argument value to a parameter in a function call. Although permitted, the use of restricted and unrestricted pointers together should be avoided, since such use often inhibits vectorization.

C.9.6 Comparison with `ivdep`

An assertion made by the use of the `restrict` keyword in the preceding examples is stronger than the assertion that would be made by `ivdep`. The type qualifier asserts that two restricted pointers, such as `p` and `q` in the previous example, point to entirely disjoint arrays, whereas the directive asserts that if there is a recurrence, the threshold is at least 64. The possibility of such a recurrence might require the code generated for the loop to include a complete memory reference (CMR) instruction at the top of the loop. A CMR can increase the execution time of a loop by as much as 30%.

In the next example, there is a recurrence with threshold 32, so it is not correct to use `ivdep`, and the loop body is just complicated enough to inhibit vectorization with a computed safe vector length. Thus, the use of restricted pointer parameters is the only way to vectorize the loop without rewriting it. The qualifier can be used explicitly in the parameter declarations, as shown in the example, or implicitly, through the `-h restrict=f` option.

```
void g (int n, int * restrict p,
               int * restrict q) {
   int i;
   for ( i=32; i<n; i++ ) { /* Vectorized */
        p[i] = p[i-32] + q[i*3];
   }
}
```

C.9.7 Implementation limits

At default vectorization level, the current implementation only takes advantage of the qualifier for simple identifiers declared with file scope, as function parameters, or in the outermost block of a function. In other contexts, uses of the qualifier are checked only for syntactic correctness. The option `-h` `vector3` enables a more detailed analysis that can take advantage of restricted pointers that are members of structures or elements of arrays.

```
struct t { int n; int * restrict v; };

void h1 (struct t s1, struct t s2) {
   int i, n;

   if( 3*(s1.n-1) >= s2.n )
      s1.n = s2.n/3;

   for( i=0; i<s1.n; i++ ) { /* Vectorized */
      s1.v[i] = s2.v[3*i];   /* if vector3 */
   }
}
```

C.10 Vectorization directives

As illustrated in some of the examples in this section, vectorization directives provide a way for you to communicate information about a specific loop to the compiler.

The following vectorization directives are available:

- `ivdep`
- `novector`
- `noreduction`
- `novsearch`
- `shortloop`

These directives must be placed immediately preceding an iteration (`for`, `while`, or `do...while`) statement or a label. If one of these directives appears preceding a label that the compiler does not recognize as the top of a loop, it has no effect. If one of these directives appears in any other context, an error message is issued.

There is a `-h` option corresponding to each of these directives, except `shortloop`. The `-h nopragma=allvector` option disables these vectorization directives. The `-h vector0` option disables vectorization for all loops.

C.10.1 `ivdep` directive

The `ivdep` directive tells the compiler to ignore vector dependences for the immediately following loop; the loop then vectorizes if all other conditions are satisfactory. The `ivdep` directive has the following form:

```
#pragma _CRI ivdep
```

This directive is useful for some loops that contain pointers and indirect addressing. Recurrences are discussed on page Section C.4, page 166.

C.10.2 `novector` directive

This directive disables vectorization of the immediately following loop, regardless of other vectorization-related directives or options. The `novector` directive has the following form:

```
#pragma _CRI novector
```

C.10.3 `noreduction` directive

The `noreduction` directive tells the compiler not to vectorize the immediately following loop as a vector reduction, regardless of other vectorization-related directives or options. If the loop is not a reduction loop, the directive is ignored. You can choose to use this directive when the loop iteration count is small or when the order of evaluation is numerically significant. The directive has the following form:

```
#pragma _CRI noreduction
```

C.10.4 `novsearch` directive

The `novsearch` directive tells the compiler not to vectorize the immediately following loop as a vector search, regardless of other vectorization-related directives or options. If the loop is not a search loop, the directive is ignored. This is useful for unusual cases in which a search loop generates a floating-point exception when vectorized.

C.10.5 `shortloop` directive

The `shortloop` directive improves performance of a vectorized loop by allowing the compiler to omit the run-time test to determine whether it has been completed. The directive has the following form:

```
#pragma _CRI shortloop
```

The `shortloop` directive asserts that the immediately following loop executes in at most 64 iterations.

C.11 Memory bank conflicts

Memory contention on Cray Research systems is caused by successive accesses to the same memory bank. Use the `target` command to see how many memory banks are configured on your system.

Data elements are stored in consecutive memory banks.

Example:

```
float a[100]
```

If a program declares an array as shown, and `a[0]` happens to be in bank `n`, then `a[1]` would be in bank `n+1`, and so on.

Each access to a data bank causes the bank to be unavailable for some number of clock periods (depending on the type of memory, see the `target`(1) command). Any successive accesses to a bank that is currently unavailable will simply hold until the bank is free.

C.11.1 Algorithm considerations

The best algorithm for accessing memory in vector operations is to ensure that successively accessed elements are in distinct memory banks. Recall that C arrays are stored in row-major ordering. The following example illustrates access to memory where each successively accessed element is in the same memory bank:

```
float a[100][256], b[100][256];
for (i=0;i<256;i++)  /* Avoid this order */
   for (j=0;j<100;j++)
       a[j][i] = b[j][i] * 2;
```

Order of access is: `a[0][0]`, `a[1][0]`, `a[2][0], ...`. These elements are all in the same memory bank. The reason is that the row-length of the array a is 256, thus adjacent elements in the columns of array a are 256 words (banks) apart.

If your Cray Research system is configured with 256 banks or fewer, adjacent elements in any column of array a in the example, are going to be in the same memory bank. Because the preceding loop accesses down the columns, each successive access will hold; the performance of the vector load and store will degrade depending on the bank-busy time for your system's type of memory. Memory contention can slow a vector load or store by up to a factor of 8.

C.11.2 Memory optimization

Since the number of memory banks is a power of two, efficient memory access can be ensured by arranging for vectors to be loaded and stored with odd strides.

Two ways to modify the preceding example so that its vector load and store use odd stride are as follows:

Modified code, version 1:

```
float a[100][256], b[100][256];
for (j=0;j<100;j++)
 for (i=0;i<256;i++)
   a[j][i] = b[j][i] * 2;
```

The access in the modified code is now along the rows (stride 1) whereas the access in the original code is down the columns (stride 256).

Modified code, version 2:

```
float a[100][257], b[100][257];
for (i=0;i<256;i++)
  for (j=0;j<100;j++)
    a[j][i] = b[j][i] * 2;
```

The access in the modified code is down the columns, but now with a stride of one because each row length is now 257.

C.12 Summary

The previous sections described the theoretical possibilities for vectorization as well as restrictions imposed by the Cray C++ compiler. Most of the restrictions arise from hardware considerations or are compromises in favor of (relative) simplicity of implementation. Others are simply deficiencies in the current implementation and will be addressed in future releases.

To vectorize as much of an application as possible, you should be prepared to experiment with the capabilities of the compiler. You can use a command line like the following to cause vectorization information for each source file to be written to `stdout`.

```
cc -h report=v -c file1.c ...
```

For each loop, there is a message that gives the file name and line number at which the loop occurs and either states that the loop vectorized or gives a reason why it did not. The format of these messages is the same as that used for other optimization messages and for error messages.

In particular, inlining, restricted pointers, and the `ivdep` directive can be useful for vectorization, though they do not solve every problem. For some examples of code modifications and their effects on performance, see the examples in the following section.

C.13 Modifying loops

This section presents examples of situations in which rewriting a small amount of C code can make considerable performance improvements.

Each example gives a discussion of the issue, along with a small example of unmodified and modified code. Performance ratios are given to indicate the amount of improvement in the modified code over the unmodified code. The timings used to generate the performance figures were obtained by inserting calls to the `SECOND`(3) function in the C code.

C.13.1 Multiple dimension recurrence

Some dependency conflicts can be eliminated by switching the loop control variables. For example, switching the variables resolves the conflict created by element `daa[i][j-1]` in the following code:

```
/* Original  Unrolled 16 times, not vectorized */
for (i=0; i<iterat; i++)
```

```
   for (j=1; j<iterat+1; j++)
      daa[i][j] = daa[i][j-1]*dbb[i][j];

/* Modified  Inner loop vectorized */
for (j=1; j<iterat+1; j++)
   for (i=0; i<iterat; i++)
      daa[i][j] = daa[i][j-1]*dbb[i][j];
```

This modification to the loop produces the following performance improvement (ratio of execution time for unmodified versus modified code) for different values of `iterat`:

Iterations	Improvement
5	1.6
50	5.3
100	5.8
500	5.9
1000	5.9

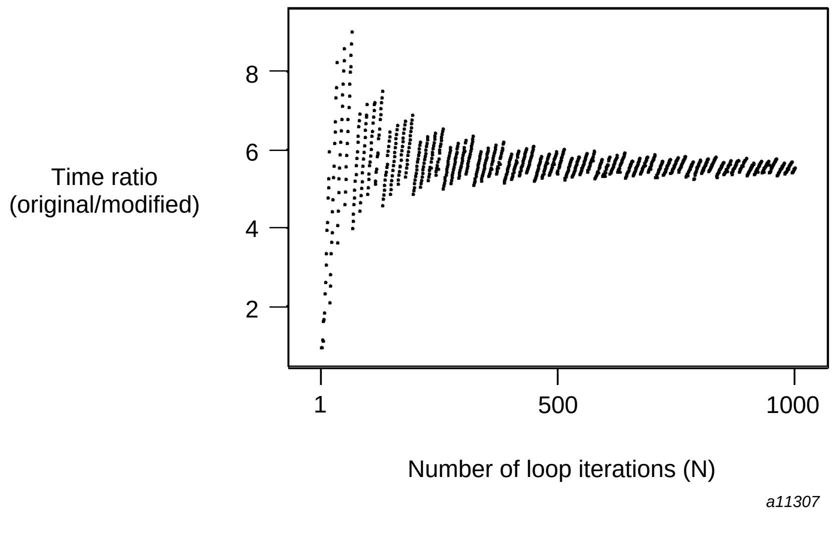

Figure 59. Performance improvement for multiple dimension recurrence

C.13.2 Loop interchange

In the following example, although the inner loop is vectorized, the outer loop has the larger number of iterations. Performance improves when the loops are inverted. (If both loops have nearly the same number of iterations, the improvement is small.)

```
/* Original Inner loop vectorized */
for (i=0; i<iterat; i++)
   for (j=0; j<10; j++)
      daa[i][j] = dbb[i][j]*j;

/* Modified */
/* Inner loop vectorized and unrolled 2 times */
for (j=0; j<10; j++)
   for (i=0; i<iterat; i++)
      daa[i][j] = dbb[i][j]*j;
```

This modification to the `for` loop produces the following performance improvement (ratio of execution time for unmodified versus modified code) for different values of `iterat`:

Iterations	Improvement
5	1.0
50	0.9
100	1.0
500	1.3
1000	1.4

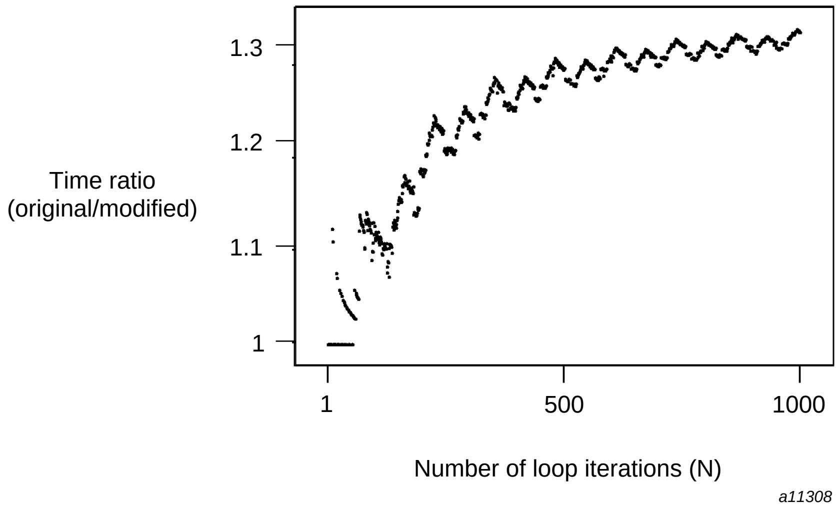

Figure 60. Performance improvement for loop interchange

C.13.3 Remove conditional `if`

Conditional tests create overhead in vector loops. In this example, the vector loop is rewritten to remove the conditional test on the loop control variable.

```
/* Original  Vectorized */
m = iterat/2 + 1;
for (i=0; i<iterat; i++)
   if (i>m) aa[i] = bb[i] * cc[i];

/* Modified  Vectorized and unrolled 2 times */
for (i = m + 1; i < iterat; i++)
   aa[i] = bb[i] * cc[i];
```

This modification to the `for` loop produces the following performance improvement (ratio of execution time for unmodified versus modified code) for different values of `iterat`:

Iterations	Improvement
5	1.3
50	1.3
100	1.5
500	3.2
1000	4.1

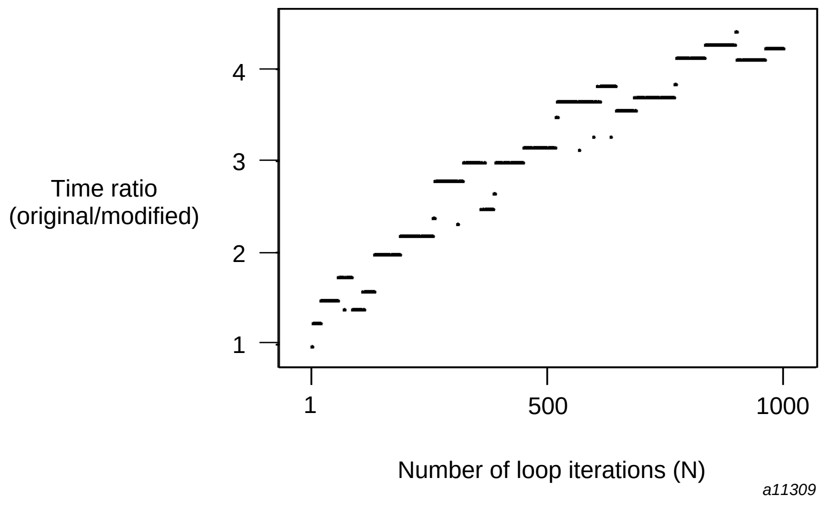

Figure 61. Performance improvement for conditional tests

C.13.4 `strlen` function

In the following example, the loop with array of type `char` is replaced with a call to `strlen`, a vectorized library function.

```
char ss[1000];
int length
while (ss[i] != '')         /* Original */
   ++i;

length = strlen(ss);           /* Modified */
```

This modification to the loop produces the following performance improvement (ratio of execution time for unmodified versus modified code) for different values of `iterat`:

Iterations	Improvement
5	1.2
50	1.6
100	2.4
500	3.3
1000	3.3

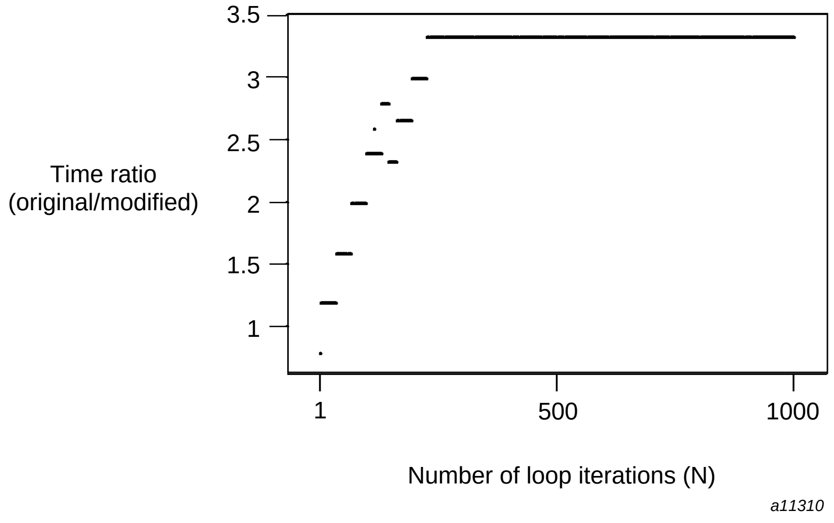

Figure 62. Performance improvement with call to `strlen`

C.14 Compilable example

This section is a complete, compilable program. Assume that it is placed in a file named `ex.c` and compiled with the following command:

```
cc -c -h report=v ex.c
```

The compiler writes vectorization information to `stdout`.

```
#include <stdlib.h>

extern int a[], b[], c[], d[], n;
extern int a2[][100];
extern char s, t[], u[];

extern struct stack {
     int a[100];
     int n;
} sa, sb;

extern struct point {
     double x, y;
} p1[], p2[], * pp;

extern struct vector {
     double x, y;
} * pv;

extern struct packed {
     int a:32;
     int b:32;
} sc[], * ps;

int *p, *q;
int * restrict rp, * restrict rq;

#pragma _CRI vfunction vf
extern int vf(int i, int j);

int f(int i, int j) {
     return(i + j);
}

     void examples() {
     int i, j, k;

     /* Vectorization of innermost loops:              */
     /* The innermost loop (do-while loop) vectorizes,   */
     /* the outer two loops (while loop and for loop)   */
     /* are not candidates for vectorization.           */

     i = 0;
     while (i<10) {
         for (j=0; j<n; j++) {
             k = 0;
```

```
        do {                        /* Vectorized */
            a[k] = c[j];
        }
        while (++k<n);
    }
    i++;
}

/* Trip count calculation:                     */
/* The trip count for the first loop can be      */
/* calculated at compile time, giving a "short"   */
/* vector loop, with minimal overhead.           */
/* The trip count for the second loop must be    */
/* calculated at run time: trip count = n-j+1.   */
/* A trip count loop cannot be calculated prior   */
/* to entering the third loop, and so it is      */
/* vectorized as a "search", with more overhead   */
/* than the first two loops.                    */

for (i=0; i<64; i++) {              /* Vectorized */
    a[i] = 0;
}

for (i=j; i<=n; i++) {              /* Vectorized */
    a[i] = 0;
}

for (i=0; b[i]!=0; i++) {           /* Vectorized */
    a[i] = b[i];
}

/* Function calls within loops:                */
/* The first loop does not vectorize, because of  */
/* call to the function "f".                   */
/* The second loop does vectorize, since the call */
/* is inlined.                                */
/* The third loop also vectorizes, since the call */
/* is to a "vfunction" written in CAL.          */

for (i=0; i<n; i++) {           /* Not vectorized */
    a[i] = f(b[i], c[i]);
}
```

```
  for (i=0; i<n; i++) {               /* Vectorized */
      #pragma _CRI inline f
      a[i] = f(b[i], c[i]);
  }

  for (i=0; i<n; i++) {               /* Vectorized */
      a[i] = vf(b[i], c[i]);
  }

  /* Recurrences:                                 */
  /* The first two loops do not vectorize because    */
  /* of recurrences.                              */
  /* The third loop vectorizes because the           */
  /* recurrence is effectively eliminated when its   */
  /* threshold (or safe vector length) is 64.        */
  /* The fourth loop can vectorize with a vector     */
  /* length of 32.                                */

  for (i=0; i<a[10]; i++) {       /* Not vectorized */
      a[i] = b[i];
  }

  for (i=1; i<200; i++) {         /* Not vectorized */
      a[i] = a[i-1];
  }

  for (i=64; i<200; i++) {            /* Vectorized */
      a[i] = a[i-64];
  }

  for (i=32; i<200; i++) {            /* Vectorized */
      a[i] = a[i-32];
  }
  /* Branches:                                       */
  /* The first loop cannot be vectorized because     */
  /* there is a branch into the loop.                */
  /* The second loop cannot be vectorized because    */
  /* there is a backward branch within the loop.     */

i = 0;
  if (n) goto lab;
  while (i<100) {                 /* Not vectorized */
      b[i] = a[i];
```

```
lab:
         a[i++] = 0;
     }

     for (i=0; i<n; i++) {           /* Not vectorized */
lab1:
         a[i] += b[i];
         if (a[i]<0) goto lab1;
     }

     /* Search loops:                                  */
     /* The first loop vectorizes.                     */
     /* The second loop does not vectorize, because    */
     /* the exit from the loop is preceded by a        */
     /* conditional assignment.                        */

     for (i=0; i<n; i++) {               /* Vectorized */
         if (a[i] > 0)
             break;
     }

     for (i=0; i<n; i++) {           /* Not vectorized */
         if (a[i] < 0) {
             a[i] = 0;
         }
         if (b[i] > 0) {
             break;
         }
     }
     /* Gather/Scatter:                                */
     /* The first loop vectorizes, with a gather from  */
     /* a and a scatter to b.                          */
     /* The second loop vectorizes, with scalar code   */
     /* to accommodate the potential recurrences.      */
     /* The third loop does not vectorize because      */
     /* of the potential output dependence on d.       */

    for (i=0; i<n; i++) {               /* Vectorized */
         d[i] = a[c[i]];
         b[c[i]] = d[i];
     }

     for (i=0; i<n; i++) {               /* Vectorized */
         d[a[i]] += c[i];
     }

     for (i=0; i<n; i++) {           /* Not vectorized */
         d[i] = b[i] + c[i];
         d[a[i]] = 1;
     }

     /* Structures:                                    */
     /* The first two loops vectorize because they     */
     /* reference objects with a basic type.           */
     /* The third loop vectorizes like the second loop */
     /* because the structure is smaller than 16 words.*/

     for (i=0; i<n; i++) {               /* Vectorized */
         sa.a[i] = sb.a[i];
     }

     for (i=0; i<n; i++) {               /* Vectorized */
         sc[i].a = sc[i].b;
     }

     for (i=0; i<n; i++) {               /* Vectorized */
         p1[i].x = p2[i].y;
         p1[i].y = p2[i].x;
     }

     for (i=0; i<n; i++) {               /* Vectorized */
         p1[i] = p2[i];
     }

     /* Pointers:                                      */
     /* The first loop does not vectorize because the  */
     /* pointer is incremented by a value which is not */
     /* not a loop invariant.                          */
     /* The second loop vectorizes with a run-time     */
     /* computation of a safe vector length. If that  */
     /* length turns out to be 1, the loop runs slower */
     /* than if it were not vectorized.                */
     /* The third loop vectorizes unconditionally      */
     /* because the pragma asserts that the safe       */
     /* vector length is at least 64.                  */
     /* The fourth loop vectorizes unconditionally     */
     /* because the use of restricted pointers asserts */
     /* that there is no recurrence. (It is somewhat  */
     /* faster than the third loop, which has a CMR.)  */

     p = a;
     for (i=0; i<n; i++) {           /* Not vectorized */
         *p = 0;
         p += c[i];
     }

     p = a; q = b;
     for (i=0; i<n; i++) {              /* Vectorized */
         *p++ = *q++;
     }        /* ... with computed safe vector length */

     p = a; q = b;
     #pragma _CRI ivdep
     for (i=0; i<n; i++) {               /* Vectorized */
         *p++ = *q++;
     }

     rp = a; rq = b;
     for (i=0; i<n; i++, rp++, rq++) {   /* Vectorized */
         *rp = *rq;
     }

     /* Arrays of characters:                         */
     /* The first loop vectorizes.                    */
     /* The second loop does not vectorize because the */
     /* stride is not known to be +1 at compile time. */
     /* The third loop does not vectorize because it   */
     /* references an array not of character type.     */
     /* The fourth loop does not vectorize because it  */
     /* uses an operator which is not supported.       */
     /* The fifth loop does not vectorize because it   */
     /* is a reduction.                               */
     /* The sixth loop vectorizes because it is a      */
     /* straightforward search.                       */

     for (i=0; i<n; i++) {               /* Vectorized */
         t[i] += u[i];
     }
```

```
for (i=0; i<n; i+=k) {          /* Not vectorized */
    t[i] = 0;
}

for (i=0; i<n; i++) {           /* Not vectorized */
    t[i] = a[i];
}

for (i=0; i<n; i++) {           /* Not vectorized */
    t[i] = u[i] % 16;
}

s = 0;
for (i=0; i<n; i++) {           /* Not vectorized */
    s += u[i];
}

for (i=0; i<n; i++) {               /* Vectorized */
    if (t[i] == 0) break;
}
/* Capabilities enabled by -h vector3 include:    */
/* Reordering statements to eliminate dependences */
/* Splitting loops to allow partial vectorization */
/* Improved dependence analysis, based on better: */
/*   Analysis of index expressions                */
/*   Forward substitution of constant values      */
/*   Tracking of objects to which pointers point  */
/*   Recognition that objects allocated with      */
/*     malloc are disjoint from all other objects */
/*   Recognition that different members of the    */
/*      same structure are disjoint               */
/*   Recognition that objects with different      */
/*     types are disjoint                         */

for (i=0; i<n; i++) {               /* Vectorized */
    a[i] = b[i];                    /* if vector3 */
    c[i] = a[i+1] + 1;
}                 /* ... with statements reversed */

for (i=0; i<n; i++) {     /* Partially vectorized */
    a[i] = a[i-1];                  /* if vector3 */
    b[i] = b[i] + 1;
}          /* ... by splitting the recurrence on */
```

```
                /*   'a' into a separate, scalar loop */

     for (i=0; i<n; i++) {              /* Vectorized */
          t[i] = u[i];                  /* if vector3 */
          b[i] = b[i] * 2.0;
     }   /* ... by splitting the two assignments into */
         /* separate loops, which are both vectorized */

     for (i=0; i<n; i+=2) {              /* Vectorized */
         a[i] = a[3] * b[i];             /* if vector3 */
     }          /* ... by improved dependence analysis */

     for (i=0; i<n; i++) {               /* Vectorized */
         a2[i][i] = a2[i][n-1] * b[i];   /* if vector3 */
     }          /* ... by improved dependence analysis */

     for (i=0; i<n; i++) {               /* Vectorized */
         sa.a[i] = sa.n;                 /* if vector3 */
     }          /* ... by improved dependence analysis */
     k = n/2;
     for (i=0; i<n/2; i++) {             /* Vectorized */
         a[i] = a[i+k];                  /* if vector3 */
     }    /* ... by forward substitution of value of k */

     p = a;
     q = b;
     for (i=0; i<n; i++) {               /* Vectorized */
         *p++ = *q++;                    /* if vector3 */
     }             /* ... by improved pointer tracking */

     p = malloc( n * sizeof(int) );
     for (i=0; i<n; i++) {               /* Vectorized */
         p[i] = a[i];                    /* if vector3 */
     }                   /* ... by knowledge of malloc */

     for (i=0; i<n; i++) {               /* Vectorized */
         pv[i].x = pp[i].x;              /* if vector3 */
         pv[i].y = pp[i].y;
     }               /* ... by improved alias analysis */

}
```