# C AND C++ PROGRAMMING FOR THE VECTOR PROCESSOR

*Geir Johansen*, Cray Inc., Mendota Heights, Minnesota, USA

**Abstract:** *The Cray Standard C and C++ compilers have the ability to perform many types of optimizations to improve the performance of the code. This paper outlines strategies for C and C++ programming that will give the Cray Standard C and C++ compilers a greater opportunity to generate optimized code.*

## 1.0 Scope of the Paper

The purpose of this is to give guidance to C and C++ developers in writing code, so that the Cray compiler will produce better optimized code. An algorithm can be coded in several different ways; however, the Cray compiler can more easily optimize certain styles of coding. The more optimization that is initially performed by the compiler, the less time the user needs to spend on analyzing and manually optimizing the code. The paper is not intended for a specific Cray architecture, so in general, machine specific optimization issues such as memory contention are not discussed. Also, the paper does not discuss optimization techniques, such as loop fusing, that can be performed by the compiler. The paper focuses on writing code for the Cray Standard C and C++ compilers so that it has a better chance to optimize the code.

The paper generally limits its discussion to vectorization optimization, however, many of the ideas discussed will also help the compiler in producing code that has better multistreaming, tasking, and/or scalar optimization.

## 2.0 The Cray Standard C and C++ Compiler

The Cray Standard C and Cray C++ compiler are the same executable file. The Cray C/C++ compiler has four main components:

1) Edison Design Group (EDG) Front-end
2) Kuck & Associates (KAI) Inliner
3) PDGCS Optimizer
4) Code Generator

PDGCS (Program Dependence Graph Compiling System) is the area of the compiler that performs most of the optimization of the code. PDGCS has been a part of the compiler since the Cray YMP. The Cray C/C++ compiler greatly benefits from the fact that PDGCS is also the backend for the Cray Fortran compiler. The Cray C/C++ compiler leverages the extensive testing and development to PDGCS for the Cray Fortran compiler. Optimizations that PDGCS perform include:

∑ Reduction loops
∑ Loop fusing
∑ Loop unrolling
∑ Loop unwinding
∑ Loop interchange
∑ Loop splitting
∑ Pattern matching
∑ Code elimination

## 2.1 Determining Optimization of the Code

The Cray C/C++ compiler reports on the optimization of the code via the '-h report=' compiler option. The option takes the following arguments:

| | | |
|---|---|---|
| ∑ | i | inlining optimization |
| ∑ | m | multistreaming optimization |
| ∑ | s | scalar optimization |
| ∑ | t | tasking optimization |
| ∑ | v | vector optimization |
| ∑ | f | outputs report to <filename>.V |

Unfortunately, the Cray C++ does not have a listing feature similar to Cray Fortran to mark the optimization of the code within a listing of the program.

## 3.0 Defining the Datasets

The compiler performs two main functions in creating functionally correct code and optimizing that code for the machine. The function of creating functionally correct code for all possible datasets is usually the factor that limits the amount of the optimization that can be performed. A corollary to this fact is that the more information that the compiler knows about the dataset being operated on, the more potential there is for optimization of the code. The Fortran compiler's knowledge of the data it is operating on is typically why a Fortran coded code will outperform a similar C coded program. This section will illustrate ways that code can be written so that it is more "optimizable" by the compiler.

## 3.1 Ambiguous Aliasing of the Data

*Pointers have been lumped with the* goto *statement as a marvelous way to create impossible-to-understand programs.*

-Kernighan & Richie

While pointers may make program difficult to decipher for novice C programmers, they also make it more difficult for the compiler to optimize the code. The fundamental problem concerning pointers is that the compiler cannot be sure where the pointer is pointing, this situation is known as the *ambiguous aliasing* of data. If the compiler is uncertain of the location of the data, then it cannot determine if the code has any data dependencies. For example, compare the two routines:

```
$ cat pntr0.c
extern a[100];
extern b[100];

void pntr0()
{
    int i;
    for (i=0; i<64; i++)
        b[i] = i * a[i];
}


$ cat pntr1.c
void pntr1(int *a, int *b])
{
    int i;
    for (i=0; i<64; i++)
        b[i] = i * a[i];
}
```

The code generated for `pntr0` runs faster than the code for routine `pntr1` because the compiler knows that there is no data dependencies between `a` and `b` in `pntr0`. An example of data dependency would be if the program had an integer array `int c[100]` and `pntr1` was called with the parameters `pntr1(&c[0], &c[1])`.

The Cray compiler will not generate fully vectorized code for the loop in routine `pntr1`, however, the loop be vectorized with a computed maximum safe vector length. What this implies is that code will be inserted in the routine to check the data at run time for the maximum vector length that can be used. In the case of `pntr1(&c[0], &c[1])`, the vector length is equal to one, which means that the code will run the same as scalar code. When the overhead of computing the vector length is added, the resulting code will run slower than a scalar version of the routine.

If the user knows that pointers in the code will never overlap into another pointer's data area, then the pointer can be designated as a restricted pointer. A pointer can be designated as a restricted pointer by using the `restrict` keyword, or by using the Cray C/C++ '-h restrict' command line option. Rewriting the routine as follows will result in a loop that will fully vectorize:

```
void pntr1(int * restrict a, int
* restrict b])
{
    int i;
    for (i=0; i<64; i++)
        b[i] = i * a[i];
}
```

## 3.2 Scope of Pointers

While a restricted pointer designates a covenant between the user and the compiler that pointers will not overlap into another pointer's data, it does not insure that the pointer will not change its value within the routine. If during the scope of the pointer in a routine, only the data being pointed to by the pointer is operated on, and not pointer itself, then the user may increase the potential for optimization by designating as a `const` pointer. Using the example from the previous section, the call to the routine can be rewritten as follows:

```
void pntr1(int * const restrict
a, int * const restrict b])
{    .........     }
```

The `const` keyword informs the compiler that pointers `a` and `b` will continue to point to the same data segment in the routine. The values of the data segment can be changed, but not the pointer itself.

If the pointer being passed is only used to point to read-only data, the pointer can be further defined to show it is pointing to constant data (i. e. `const int * const restrict a`). This further refinement of the data has not been seen to increase optimization by the Cray C/C++ compiler.

## 3.3 Multiple Indirection of Data

As mentioned the compiler is conservative in assuming aliasing of data, which results in less optimization of the code. As more levels of indirection are added to variables, the more difficult it is for the compiler to determine data dependencies. If the compiler cannot rule out that there are no data dependencies in a loop, then this limits the amount of optimization that can be done by the compiler. The use of restricted pointers can be used to inform the compiler that there are no data dependencies, however, the process of determining all the pointers that need to be restricted to optimize the code may be time consuming. The following example shows what pointers need to be made restricted in order to optimize the code:

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

The pointers could also be made restricted by using the '–h restrict=' option. In the following example the Cray C++ compiler option '-h restrict=f' is used to make all function arguments restricted pointers, which allows the code to optimize:

```
//compiled with '-h restrict=a'
sub(float ***a,
    float ***b,
    float ***c,
    int n)
{
    int i,j,k;
  for(k=0;k<n;k++)
    for(j=0;j<n;j++)
      for(i=0;i<n;i++)
        c[k][j][i] += a[k][j][i]
* b[k][j][i];
}
```

Another option is to use compiler directives, such as `ivdep` or `concurrent`, to tell the compiler to assume the code has no data dependencies. The following example shows the `ivdep` directive being used to optimize the code:

```
sub(float ***a,
    float ***b,
    float ***c,
    int n)
{
    int i,j,k;
  for(k=0;k<n;k++)
    for(j=0;j<n;j++)
#pragma _CRI ivdep
      for(i=0;i<n;i++)
        c[k][j][i] += a[k][j][i]
* b[k][j][i];
}
```

### 3.3.1 Eliminating Indirection

A strategy for dealing with multiple indirection is to remove the indirection by either copying data to temporary variables or moving the algorithm to a subroutine. Here is an example of moving code to a subroutine:

```
for(i=0;i<z->b->size;i++)
   z->b->data[i] = x->b->data[i]
* y->b->data[i];
```

The above loop can be replaced by the following subroutine call and subroutine:

```
sub(x->b->data, y->b->data,z->b-
>data, z->b->size);
         .
         .
sub(float * restrict a,
     float * restrict b,
     float * restrict c,
     int n)
{
      int i;
      for(i=0;i<n;i++)
        c[i] = a[i] * b[i];
}
```

### 3.3.2 Memory Access of Multiple Indirection

Each level of indirection of a variable adds one more memory operation. For example, accessing the variable `A->B->*data` could result in as many as four memory accesses. The compiler does its best to try to keep variables in registers, but because of code complexity or perceived data dependencies, the compiler may unnecessarily need to reload the variable. For example, in the following code the user may want to consider loading the value of `A->B->*data` to temporary variable before calling and using this value in the routine:

```
  x = A->B->*data * a;
  printout(A); // A not changed
  y = A->B->*data * b;
```

## 3.4 Dimension of the Dataset.

The more information that can be given to the compiler about the dimension of the dataset being operated, the more opportunity the compiler has to perform optimization. An example of how knowing the dimension of the dataset at compilation increases optimization is shortloop vector optimization. If the compiler knows the loop count is less than the machine's vector size, then the compiler can replace the loop with vector operations. If the loop count is greater than machine's vector size, then the compiler can choose the optimal vector length for the loop. Knowing the dimension of the data being operated on also helps in the compiler performing other optimizations such as loop fusing and loop unrolling.

### 3.4.1 Variable Length Arrays (VLAs)

Variable Length Arrays (VLAs) can be used to communicate to the compiler more information about the dataset. Even if the size of the VLA is not known at the time of compilation, the compiler will benefit in knowing that the pointer is pointing to arrays of uniform length. In the following example the routine `vla0` has been rewritten to `vla1`, so that arrays `a` and `b` have been declared in the routine as VLAs. The compiler is able to take advantage of the this information by collapsing the loops:

```
void vla0(a,b,m,n)
int m;
int n;
float * restrict * restrict a;
float * restrict * restrict b;
{
  int i,j;
 for (j=0;j<m;j++)
  for (i=0;i<n;i++)
    b[j][i] = a[j][i];
}
```

Rewritten code:

```
void vla1(a,b,m,n)
int m;
int n;
float (* restrict a)[n];
float (* restrict b)[n];
{
  int i,j;
 for (j=0;j<m;j++)
  for (i=0;i<n;i++)
    b[j][i] = a[j][i];
}
```

```
CC-6003 cc: SCALAR File = vla1.c,
Line = 10
  A loop was collapsed into the
loop starting at line 11.
```

## 3.5 Scalar Temporaries

A scalar variable causing a data dependency, or *recurrence*, in a loop may have the dependency removed by converting the scalar to a vector variable. In the following example the scalar variable `s` is converted to a vector variable for the outer loop, then the loops are switched:

```
// original code
scal0(float * restrict a,
    float * restrict b,
    int m, int n)
{
    int i,j,k;
    float s;
    for(j=0;j<n;j++) {
      s= 0.0;
      for(i=0;i<m;i++) {
        s = s + a[i];
        b[i] = b[i] + s;
      }
    }
}

//modified code
sub(float * restrict a,
    float * restrict b,
    int m, int n)
{
    int i,j,k;
    float s[n];
    for(j=0;j<n;j++)
      s[j] = 0.0;
    for(i=0;i<m;i++) {
      for(j=0;j<n;j++) {
        s[j] = s[j] + a[i];
        b[i] = b[i] + s[j];
      }
    }
}
```

## 4 Eliminating Optimization Inhibitors

For C/C++ programs the factors that prevent a loop from vectorizing and performing other optimizations include:

∑ Subroutine calls
∑ Data type
∑ Unrestricted branches
∑ Data Dependencies

This section will provide some coding strategies to eliminate these optimization inhibitors.

### 4.1 Subroutine Calls

Unless a subroutine is a compiler intrinsic or a user defined `vfunction`, the compiler will not vectorize loops with subroutine calls. The following are ways to eliminate subroutines calls within a loop.

#### 4.1.1 Inlining

A subroutine can be inlined within a loop to increase optimization. After the subroutine has been inlined into the loop, there is potential the loop will be vectorized. At the very least, the overhead to call the subroutine will be removed from the loop. A subroutine can be marked to be inlined by using the `inline` directive or by using the C++ `inline` keyword. The Cray C/C++ option '-h inline' is used to control the level of inlining optimization. At this time, the Cray C/C++ compiler will only inline routines that are located in the specific file being or any of the header files it includes.

#### 4.1.2 Pushing loop to Subroutine

An alternative to inlining is to push the loop calling the subroutine in to the subroutine. After the loop is pushed into the subroutine the loop may be able to be optimized. Also, the overhead of calling the routine will be removed from the loop. The following example shows the loop in `sum0` being pushed into a subroutine:

```
$ cat push0.c
int sum0 (int * restrict a,
         int * restrict b,
         int n)
{
  int i;
  int s=0;
  for (i=0;i<n;i++)
    s += prod(a[i],b[i]);
  return s;
}

int prod(int a, int b) {
  return a * b;
}


$ cat push1.c
int sum1 (int *a, int *b,
         int n)
{
  // push loop into sum_prod()
  return sum_prod(a,b,n);
}

int sum_prod(int * restrict a,
         int * restrict b,
         int n)
{
  int i;
  int s=0;
  for (i=0;i<n;i++)
    s += a[i] * b[i];
  return s;
}
```

### 4.2 Vector Data Types

One reason that a loop may not vectorize is that the data being operated on is not word-size data. In practice, the compiler is able to vectorize a good portion of loops not operating on word-size data such as characters. A code may benefit from copying non-word-size data, such as bitfields, to word-size data, then operating on this new data structure.

#### 4.2.1 Partial-word data

An alternative to copying data is to redefine data structures so that they are word-size. In the following example, `struct A` is redefined so element `char c` is now declared in a `union` along with an `int`. The change will now allow the loop to vectorize under default optimization:

```
$ cat struct1.c
struct A {
    char c;
    int x;
}

sub(struct A * restrict *
restrict a1,
    struct A * restrict *
restrict a2,
    int n)
{
    int i;
    // loop does not vectoize
    for (i=0; i<n; i++) {
       a2[i]->c = a1[i]->c;
       a2[i]->x = a1[i]->x;
    }
}


$ cat struct2.c
struct A {
    union {char c; int i;} u;
    int x;
}

sub(struct A * restrict *
restrict a1,
    struct A * restrict *
restrict a2,
    int n)
{
    int i;
    // loop vectoizes
    for (i=0; i<n; i++)
       a2[i]->u.i = a1[i]->u.i;
       a2[i]->x = a1[i]->x;
}
```

#### 4.2.2 C++ Complex Data Type

The C++ Standard Library defines a complex class to operate on complex numbers. If a user has C++ code operating on complex numbers, they may consider using the complex data type defined in Cray C. For example, if the C++ program is performing calculations on arrays of complex number, then the program most likely will benefit from:

- Copying the C++ complex arrays to Cray C complex arrays.
- Perform the calculations in the C routine.
- Copy the resulting Cray C complex arrays back to the C++ complex arrays.

### 4.3 Program Branches

Program branches into, out of, or within a loop may cause a loop not to be optimized. Here are a couple of suggestions to help increase the optimization potential.

#### 4.3.1 Placement of Conditional Loop Exits

If the loop contains a conditional statement that results in the program exiting the loop, then it usually benefits optimization if these statements appear at the very beginning of the loop. For example, the following loop in `sub1` will vectorize, but the loop in `sub2` will not be vectorized by the compiler:

```
sub1(int * restrict a,
    int * restrict b,
    int * restrict c,
    int n)
{
    int i;
    for (i=0; i<n; i++) {
       if (a[i])
            break;
       if (b[i])
            break;
       c[i] = 0;
    }
}

sub2(int * restrict a,
    int * restrict b,
    int * restrict c,
    int n)
{
    int i;
    for (i=0; i<n; i++) {
       c[i] = 0;
       if (a[i])
            break;
       if (b[i])
            break;
    }
}
CC-6277 cc: VECTOR File = exit.c,
Line = 23
  A loop was not vectorized
because the loop exit test is too
complicated.
```

### 4.4 Data Dependencies

The elimination of data dependencies was covered in section 3 of the paper. Here are some other ways data dependencies can be eliminated.

#### 4.4.1 Compiler Directives

The `ivdep` directive informs the compiler to ignore vector dependencies for the immediately following loop. The directive is useful for optimizing loops that contain multiple indirection and cannot be optimized using restricted pointers. The use of the restricted pointers provides a stronger assertion to the compiler than the `ivdep` directive; so using restricted pointers to optimize code usually results in better performance.

The `concurrent` directive is similar to the `ivdep` directive. This directive informs the compiler to assume the following loop has no data dependencies and streaming optimization should be attempted on the loop.

#### 4.4.2 More Aggressive Optimization

The user may want to try increasing the level of optimization of the code through the use of the optimization command line options. The tradeoff of using higher level of optimizations is that compile time will increase. The Cray X1 C/C++ compiler has implemented the command line option '-h display_opt' that displays the optimization settings of the compilation. This option is useful in determining what optimization options can be increased by the user on the compilation command line. It is important to note that the exact optimizations used for the default optimization setting and '-O[0,1,2,3]' are subject to change from one major release to the Cray C/C++ compile to the next.

Examples of optimizations that are available through increased optimization settings are:

∑ Forward substitution
∑ Better dependence analysis
∑ Improved alias analysis
∑ Loop splitting

## 5.0 C++ Optimization

The best advice concerning the optimization of C++ on a Cray system is to avoid using C++ whenever possible. The main issue with C++ concerning optimization is the multiple level of indirection involved with C++. A class member function accessing a class member variable requires at least one level of indirection ("this" pointer) to access the variable. As discussed earlier multiple layers of indirection make it more difficult for the compiler to optimize the code. If C++ must be used, the following coding strategies will help it perform better.

### 5.1 Reduce Calls to Constructors and Destructors

Each time a class variable is declared, a call to a constructor routine and a destructor routine will be performed. Eliminating calls to constructors and destructors can improve the code's scalar performance.

#### 5.1.1 Initialization of Member Variables

A class constructor performs initialization of member variables. The initialization of these variables can be performed in the body of the constructor routine, or they can be done through member initialization. Member initialization is preferred, as it will result in only one call to the constructor. The following example shows how the constructor for `class A` is called only once when member initialization is used:

```
$ cat con.cc
```

```
#include <iostream>
using namespace std;

class A {
  private:
    int x;
  public:
    A() { x=0;
     cout << " A.1
constructor\n";
    }
    A(int i) { x=i;
     cout << " A.2
constructor\n";
    }
};

class B1 {
  private:
      class A y;
  public:
      B1(class A a) : y(a) {};
};

class B2 {
  private:
      class A z;
  public:
      B2(class A a) { z = a; };
};

int main() {
 cout << "Declaring class B1\n";
 class A a1(1);
 class B1 b1(a1);
 cout << "Declaring class B2\n";
 class A a2(2);
 class B2 b2(a2);
}

$ CC con.cc; ./a.out
Declaring class B1
 A.2 constructor
Declaring class B2
 A.2 constructor
 A.1 constructor
$
```

### 5.1.2 Placement of Class Variable Declarations

The declaration of a class variable invokes a call to constructor routine and a subsequent call to a destructor routine, so the user should be aware on the placement of the class variable declaration. For example, the following example results in N calls to `class A`'s constructor and destructor routine:

```
    for (i=0; i<N;i++) {
        class A tmp;
        tmp = a;
        a = b;
        b =tmp;
    }
```

The following code will result in only one call to class A's constructor and destructor routine:

```
    class A tmp;
    for (i=0; i<N;i++) {
        tmp = a;
        a = b;
        b =tmp;
    }
```

## 5.2 Call by Reference

The C language performs call by value when passing arguments to a routine, which results in a copy of the argument value being passed to the called routine. In C++, a routine can call by reference in passing arguments. This method is faster in that the arguments are not copied when calling the subroutine. The following example shows that a routine that passes the C++ Standard Library vector class as an argument is eight times faster when called by reference as oppose to called by value:

```
$ cat ref.cc
#include <iostream>
#include <vector>
#include <intrinsics.h>
using namespace std;

call_by_value(const vector<int>
a) { }
call_by_ref(const vector<int> &a)
{ }
#pragma _CRI noinline
call_by_value, call_by_ref

int main() {
 vector<int> a;
 int i,t0,t1,t2;
 t0 = _rtc();
 for(i=0;i<1000;i++)
    call_by_value(a);
 t1 = _rtc();
 for(i=0;i<1000;i++)
    call_by_ref(a);
 t2 = _rtc();
 cout << "Call by value / Call by
ref = "
      << (double)((double)(t1-
t0)/(double)(t2-t1))
      << endl;
}

$ CC ref.cc; ./a.out
Call by value / Call by ref =
8.04128
$
```

## 5.3 Inlining of Member Functions

One feature that greatly improves the performance of C++ code is that member functions that are define within a class are automatically inlined. As a result, the user should attempt to define member functions within a class, or mark them with the `inline` keyword. Also, in order to increase inlining throughout the program, the member functions should be defined in a header file rather than a corresponding source file for the header file. In the following example, the routine sub will inline `one_x`, since it has been defined within `class util`'s definition. The routine `two_x` will not be inlined using default optimization, however, the routine `three_x` will be inlined since it has been marked by the inline keyword. The routine four_x is not inlined, since it is not seen when compiling the routine sub.cc:

```
$ cat sub.c
#include “util.h”
sub(class Util u)
{
  u.one_x(); //inlined
  u.two_x(); //not inlined
  u.three_x(); //inlined
  u.four_x(); //not inlined
}

$ cat util.h
class Util
{
  private:
    int x;
  public:
    Util() : x(0) { };
    Util(int i) : x(i) { };
  public:
    void one_x() { x += 1; };
//inlined
    void two_x();
    void three_x();
    void four_x();
};

void Util::two_x() { x += 2; }
//not inlined

inline
void Util::three_x() { x += 3; }
//inlined

$ cat util.cc
#include "util.h"

inline
util::four_x() { x += 4; } //not
inlined
```

## 5.4 Use C Functions

The multiple indirection and layers of inlining may make the optimization of the code difficult for the compiler. The user may want to consider using C syntax when coding computative intensive sections of the program.

## 5.5 C++ Standard Library

Cray C++ contains the C++ Standard Library. At this time, the code of the C++ Standard Library has not been fully optimized for Cray systems.

## 6.0 Alternative Algorithm

This section presents vector friendly programming alternatives to the ubiquitous C linked list example. A linked list loop will not be optimized since each loop iteration requires the pointer value from the previous iteration. As a result, the loop will not be able to be multistreamed nor vectorized:

```
 // linked list example
list_t *p
for (p=top; p != 0; p=p->next)
    p->a = p->a * x
```

One alternative is to build an index array after the linked list has been initially created. The list node can now be accessed via the list array:

```
list_t *list =
malloc(n*sizeof(list_t *));
int n = 0;
```

```
for (p=top;p != 0; p=p->next)
    list[n++] = p;

// access the nodes via a
// list array
for(i=0;i<n;i++) {
    list[I].a = list[I].a * x;
```

Another alternative is to initially build the list as an expanding array of structs. The following code shows the routine `new_node()`, which is called to add a new node to the list. The resulting list is an array of structs:

```
//add new node to list
struct list *new_node(void) {
    if (count >= max) {
       max+= 100000;
       list = realloc (list,
max*sizeof(struct list));
    }
    return &list[count++];
}
```

## 7.0 C/C++ Alternatives

### 7.1 UPC

UPC (Unified Parallel C) is an extension to Cray C/C++. The UPC (Unified Parallel C) programming model will explicitly specify parallelism in the code.

### 7.2 Fortran

As discussed earlier, Fortran does have some advantages in its ability to be optimized. The user may want to consider writing portions of the program in Fortran. Cray Fortran implements new Fortran 2000 C interoperability features.

### 7.3 Libraries

If possible, the user will want to take advantage of library routines from the Cray Scientific Library (libsci), as these routines have been optimized for the Cray hardware. System libraries, such as `memcpy`, are sometimes overlooked, but they also have been optimized for the machine and should be used.

## Summary

The paper has outline several strategies to increase the optimization of C and C++ code. The focus of the techniques has been to increase information about the code to the Cray C/C++ compiler, so that the compiler has more opportunity to perform optimization. Other optimization techniques that cannot be performed by the compiler were presented.

## References

1. Kernighan, Brian W.; Richie, Dennis M., *The C Programming Language*, 1978.
2. Stroustrup, Bjarne, *The C++ Programming Language, Third Edition*, 1997.
3. Cray Inc., *Optimizing Code on the Cray PVP*, 1997.
4. Thomas, Kevin, *Cray X1 C/C++ Optimization* (presentation slides), 2003.