



C and C++ Programming For the Vector Processor

Geir Johansen
May 14, 2003



Discussion Topic

What the developer can do so that the C/C++ compiler has the best chance to optimize the code

- Focus on compiler, not machine
- Specify optimizations that only programmer, and not compiler, can perform



“Premature optimization is the root of all evil”
- Donald Knuth

- Setting the stage for optimization done by the compiler
- Reduce time of performance analysis



Not Discussed

- Specific Machine Optimizations
 - Memory contention
 - Cache usage
- Optimizations compiler can perform
- Performance analysis tools



Cray/C++ System Software Usage

- Operating system
- System Libraries
 - » Exercises optimization features
- Open Source Software

Cray C/C++ Application Usage

- Proprietary Software
- New Development
- Not many ISV codes



Cray C/C++ Compiler

- Standard C and C++ are same executable
- Components
 - Edison Design Group (EDG) Frontend
 - Kuck & Associates (KAI) Inliner
 - PDGCS Backend
 - Code Generator



“In the Cray compiler, it has always been
Fortran, Fortran, Fortran”

-- Mountain View based manager of Cray C/C++,
January 1999



PDGCS

- Program Dependence Graph Compiling System
- Performs optimization
- In development since Cray YMP
- Same code is used in Fortran compiler
 - Leverage extensive development and testing done to PDGCS for Fortran



Sample of PDGCS Optimizations

- » Reduction loops
- » Loop fusing
- » Loop unrolling
- » Loop unwinding
- » Loop interchange
- » Loop splitting
- » Pattern Matching
- » Code Elimination



Optimization Report Messages

- -h report=args argument
 - i inlining optimizations
 - m multistreaming optimizations
 - s scalar optimizations
 - t tasking optimizations
 - v vector optimizations
 - f Outputs messages to <filename>.V
- No listing feature for Cray C/C++



Definition of Dataset

- Compiler creates code for **all** possible data
- Increasing information to compiler about data improves optimization potential
- Fortran has advantage over C/C++



Data Definitions

- location (aliasing)
- scope
- dimension



Ambiguous Aliasing Issue

- Pointers heavily used in C/C++
- Pointers can create dependencies
- Compiler must be conservative in assuming aliasing



Loop not fully vectorized

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

- Possible data dependency between what a points to and what b points to
- Safe vector length optimization



Loop fully vectorized

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

- restrict keyword implies a covenant between the compiler and programmer that there is no data dependencies for the pointer



Scope of Pointers

- In previous slide, call to pntr could be further refined to:

```
void pntr(const int * const restrict a, int * const restrict b)
```

- Indicates that value of pointers a and b would not be altered
- The array a contains read-only data
- Doesn't work for the loop:

```
    for (i=0;i<64;i++)  
        *b++ = i * *a++
```




Multiple Indirection

- More difficult for compiler to determine data dependencies
- C++ notorious for multiple indirection
- Using restrict keyword will work, but ...
 - Time consuming
 - Less readable code



Multiple Indirection Solutions

- -h restrict=arg option
 - f function parameters
 - t C++ this pointer
 - a All pointers
- Experiences show that optimal code is not always generated with -h restrict option
- Used optimization directives to force optimization (i.e. ivdep, concurrent)



Memory Access of Multiple Indirection

- Multiple indirection is also costly in that more memory accesses are needed to access data (one possible memory access per level of indirection)

Example: A->B->*data

- Compiler will attempt to store variables in registers
- Reduce memory access of multiple indirection by:
 - Pushing code to subroutine
 - Using temporary variables



Dimension of Dataset

- Knowing dimension of data increases optimization potential
- If dimension $<$ machine vector size, then can perform shortloop vectorization
- Dimension assists with such optimizations such as loop fusing, loop rolling, loop interchange



VLAs

- VLAs (Variable Length Arrays) can be used to define the dimension of a passed array argument
- VLAs inform compiler that arrays are of uniform length
- VLAs only supported in Cray Standard C, not Cray C++



VLA example, part 1

```
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];
}
```



VLA example, part 2

- Compiler is now able to collapse the loops:

```
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];
}
```



Eliminate Optimization Inhibitors

- Factors that prevent vectorization and other optimization include:
 - Subroutine calls
 - Non-word data types
 - Branches in and out of loops
 - Data dependencies



Subroutine Calls

- Push loops into subroutines
- Inlining
 - -h inline command line option
 - Inline #pragma directive
 - C++ inline keyword
- Can only inline routines found in compilation file and files that are included



Non-word size data

- Non word size examples
 - Characters (for the most part, compiler will vectorize loops with chars)
 - Bit-fields
 - Structures not ending on a word boundary
- Possible solutions
 - Copying data to word friendly data structure
 - Add to data structure to align on word boundary

```
struct A { char c; int x;}
```

```
struct A { union u{char c; int i;}, int x;}
```



C++ Complex Data Type

- C++ Standard Library defines a complex template class to operate on complex numbers
- Performance can be increased by using the Cray C complex intrinsic
 - Copy C++ complex arrays to C complex array
 - Perform calculations in C routine
 - Copy resulting data Cray C complex arrays back to C++ complex arrays



Program Branches

- Place conditional exits at the very beginning of the loop
- Past experiences has found that using C “a?b:c” syntax assisted compiler in optimization. Example:

```
    if (cond)a[i]= x;
```

```
else a[i] = y;
```

Change to:

```
a[i] = (cond)?x:y;
```



Data Dependencies

- Use restricted pointers, const, and VLAA
- Use compiler directives
 - ivdep ignore vector dependencies
 - concurrent use for multistreaming
- Use more aggressive compiler optimization options
 - Tradeoff of higher optimization is increased compilation time
 - Possible incorrect results



Compiler Optimization Options

- Cray X1 has the `-h display_opt` option that outputs the optimization options the compiler is using.
- Exact optimization settings for default optimization setting and `-O[0,1,2,3]` are subject to change
- Examples of optimizations that are performed at higher optimization settings:
 - Forward substitution
 - Better dependence analysis
 - Improved alias analysis
 - Loop splitting



C++ Optimization Tips

- Don't use C++



C++ Optimization Tips

- No really, don't use C++
 - Many levels of indirection hurt optimization potential
 - Member variables are accessed with a this pointer



C++ Optimization Tips

- **Reduce Calls to Constructors/Destructors**
 - Use member initialization. For example instead of:

```
B(class A a) { x = a ; };
```

Use:

```
B(class A a) : x(a){};
```

- Declare temporary class variables outside of loop. In following example constructor and destructor called every loop iteration

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



C++ Optimization Tips (cont.)

- Use Call by Reference
 - Passed arguments do not need to be copied as they are in call by value
- Use C routines for computative intensive portions of the program
- Cray C++ version of C++ Standard Library not fully optimized



C++ Optimization Tips

- Take Advantage of inlining
 - Member functions declared within a Class definition are automatically inlined
 - Use inline keyword to inline functions other member functions
 - Place class member functions inside header (.h) files instead files of corresponding source (.C, .cc) files



Linked List Example

Pointer chasing caused by linked lists

```
list_t *p;  
for (p=top;p != 0;p=p->next) {  
    p->a = p->a * x;  
}
```

Each iteration of the loop requires the pointer value from the prior iteration, neutralizing optimization such as unrolling, software pipelining, vectorization, and streaming



Linked list alternative 1

After building the list, create an index array

```
lit_t *list = malloc(n*sizeof(list_t *));  
int n = 0;  
    for (p=top;p!=0;p=p->next) {  
        list[n++] = p  
    }
```

List nodes can now be accessed via list array

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



Linked list alternative 2

Use an expanding array of structs

```
struct list *new_node(void) {  
    if (count >= max) {  
        max += 10000;  
        list = realloc(list,max*sizeof(struct list));  
    }  
    return &list[count++];  
}
```



Compiler Optimization Opportunities

- Better optimization when restrict pointers are used
- Inlining from another C/C++ source (and binary) file
- Better listing information
- Optimize C++ Standard Library



Nuggets

- Use restricted pointers to reduce ambiguous aliasing
- Use of VLAs in routines can improve performance
- Avoid C++
- If using C++, be sure to inline