Cray Fortran 90 Optimization

Sylvia Crain, Cray Research, A Silicon Graphics Company,
655F Lone Oak Drive, Eagan, Minnesota 55121

ABSTRACT: *This paper provides a discussion of the optimization technology used in the Cray CF90 compiler. It also outlines some specific optimizations important to the compiler and provides numbers on Cray CF90 performance relative to the Cray CF77 compiler. Lastly, it suggests strategies for transitioning from Fortran 77 to Fortran 90.*

1 The Optimizer Technology

The optimizer used by Cray's Fortran 90 compiler is called PDGCS which stands for Program Dependence Graph Compiling System. This optimizer provides state of the art technology to support the Fortran 90 language. The premise of the optimizer is hierarchical memory optimization which structures optimizations to reduce memory traffic, take advantage of cache, and reduce the number of loads and stores required within a program. The program dependence graph (PDG) is at the heart of the optimizer and is integral to the design. Modifications to the PDG build on top of one another. As each optimization is applied, the PDG is recomputed and opportunities for further optimization are then reevaluated. Selection of each optimization is dictated by a series of heuristics which choose the best optimizations in each case for decreasing memory traffic and increase the size of the blocks for subsequent optimizations. The approach of integrating optimizations and creating an interplay between them is the direction of the compiler industry as a whole and has the potential to provide very high levels of performance well beyond that achievable with Cray's CF77 compiler.

2 Important CF90 Optimizations

Several of the optimizations within the Cray CF90 compiler provide good performance boosts to Fortran 90 performance. These include optimizations such as inlining of array intrinsics, and various optimizations that restructure loops in many different ways.

Array syntax allows for representing arrays in a concise manner with Fortran 90 syntax. Array operations are expanded into a series of loops and calls within the compiler. Inlining of these intrinsics allows for other optimizations to take place on the loop bodies and also eliminates the overhead of the subroutine calls. Even with no other optimization, removal of the subroutine call overhead can provide a good win.

Loop fusion is an important optimization to aid array syntax. After an array intrinsics is expand into a series of loops, the loops can then be fused or combined into a single larger loop where other potential optimizations can then be performed.

Loop Interchange is swap inner and outer loops. This allows for moving dependencies out of the loop thus providing greater opportunities for optimization within loops as well as improving data locality and cache optimization.

Outer Loop Vectorization promotes the concept of vector invariant code as opposed to scalar invariant code. With vector invariant code, a vectors worth of elements are pulled to the outer loop and the inner loop is run as scalar. Outer loop vectorization is used in instances where may people would think loop interchange would be applied, but to preserve the data locality outer loop vectorization is often preferred. As CF77 does not perform outer loop vectorization, his type of optimization can increase the performance of a loop by 6 to 7 times over the performance obtained with CF77.

Loop unrolling is duplicating the body of a loop for some number of iterations of the loop on sequential iterations. When a loop is unrolled this allows for further optimization by providing a larger code block for optimization and improved code scheduling. Jamming is combining the contents of two separate loops into a single loop when the iteration counts are the same. Combining these two optimizations, unroll & jam is unrolling an outer loop and then fusing the unrolled loop into a single larger code block to be further optimized.

Loop splitting or partial vectorization breaks a loop into a vector loop which contains as much of the work as possible and a separate scalar loop which contains the dependencies. The philosophy of this optimization is that being able to vectorize any part of a loop will provide better performance than not vectorizing any of it. This is a very aggressive optimization and must be applied carefully so as not to miss dependencies. The payoffs with loop splitting can show a 2 times speedup over non-split code which in this instance runs scalar on CF77. In the loop example below, ZA(k-1,j) and ZV(k,j) are dependent upon earlier iterations of the loop and thus are split into a scalar loop. Additionally, QA referenced on the right hand side of the equation is pulled into a vector temporary for computation.

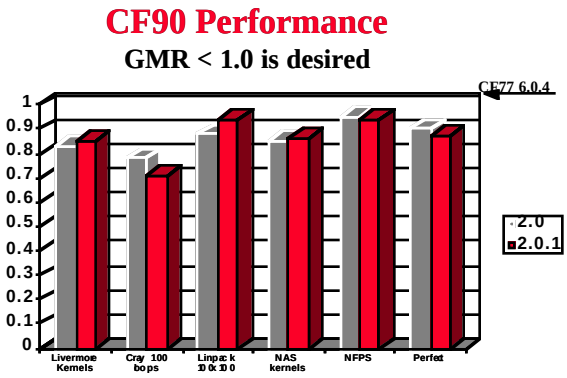
```
      fw = 0.17500d0
      do 23 j = 2,6
         do 23 k = 2,n
            QA=ZA(k,j+1)*ZR(k,j)
     &      + ZA(k,j-1)*ZB(k,j)
     &      + ZA(k+1,j)*ZU(k,j)
     &      + ZA(k-1,j)*ZV(k,j) +ZZ(k,j)
23    ZA(k,j) = ZA(k,j) +fw*(QA -ZA(k,j))
```

Loop collapse is a technique where a doubly nested loop is linearized into a single loop. The following example shows a loop that is optimized by collapsing the i and j loops into a new loop k which runs from nsize2 * nsize1. Linearization improves data locality and can provide significant improvements in performance. This example shows nearly a 9 times speedup over CF77 for the same loop.

```
      do i = 1, nsize2
         do j = 1, nsize1
            a(j,i) = a(j,i) - a(j-1,i) * .99999
         enddo
      enddo
```

3 Current Performance of CF90

Examples of individual loops do not necessarily predict the overall performance of a code. The following graph shows a comparison of the performance of CF77 6.0.4 relative to the current release of CF90. The test suites used are standard benchmarks (Livermore Kernels, Linpack, NAS and Perfect) and codes from customers (Cray 100 and NFPS.) All codes are written in Fortran 77. Numbers less than 1 in all cases represent superior execution performance.



As you can see, the geometric mean ratio (GMR) shows CF90 out performing CF77 on all 6 suites in the study. It is possible that individual codes within a given suite may be slower than CF77, but to provide a GMR better than CF77, other codes must run significantly faster. These numbers show that for a broad base of codes, Cray's Fortran 90 compiler out performs the Cray Fortran 77 compiler.

Lets assume for a moment that Cray decided to move a back to our CF77 compiler. On average, customers would loose performance they have gained by moving to the new CF90 compiler. Without the new restructuring optimizer technology provided with PDGCS, we would have no way to make up the performance loss in moving back to CF77.

4 The Transition Strategy

To complete the transition from CF77 to CF90, several steps must be taken. First, Cray must ensure that the basic performance of CF90 equals or surpasses that of the CF77 compiler. As was shown in the previous graph, this has been accomplished.

Cray is aware that other areas of improvement will be discovered as the product matures. Wherever possible, Cray will identify and remedy these situations. Based on differences in the technology between the two Fortran compilers, however, there will be instances where in order to provide the best general performance on a class of loops, performance of an individual loop may not be as fast as was seen in CF77.

Additionally, the new restructuring technology offers opportunities for further tuning and refinements within users codes. Just as users have tuned their applications to execute very effectively with CF77, there is now an opportunity to do further tuning of codes to gain greater performance than was previously possible with CF77.

5 Summary

Aggressive restructuring of user's code by the CF90 compiler can provide greater performance than was possible with Cray's CF77 compiler. This increase in performance is accomplished by exploiting memory hierarchy to improve data locality. The CF90 optimizer is still maturing and thus additional areas for performance improvement will be discovered and implemented. To achieve maximum performance with CF90 however, a partnership between the compiler, Software development and the code developer is required to identify areas for improvement, and to determine the most effective avenue for realizing this performance. The solution may require compiler changes, or modifications to user applications to take best advantage of existing optimizations.