Simple Loop Performance on the Cray X1

Lee Higbie, Tom Baring, Ed Kornkven

University of Alaska Fairbanks, Arctic Region Supercomputing Center

ABSTRACT: We analyzed the performance of the Cray X1 on two sets of nested loops that were iterating over triply subscripted arrays and found performance variations from 1.8 to 818 M(64-bit)FLOPS (a 457:1 ratio). The interaction between the multi-level cache, vector operations, memory bank conflicts, and “multi-streaming” are the apparent cause for most of the large speed variations. We used only two simple computations but varied many parameters such as the order of the loops, the subscripting style, compilation options, precision, and language. With the inclusion of 32-bit and 128-bit floating point arithmetic, the speed range increased to about 40K to 1, from .04 to almost 1500 MFLOPS. We found that compiler feedback on loop vectorization and multistreaming was a poor performance predictor. We conclude the paper with some observations on estimating and improving Cray X1 application performance.

KEYWORDS: Cray X1, performance, vectorization, multistreaming, memory hierarchy

1. Introduction

One of ARSC's large users experienced poor performance on our X1, Klondike. The individual was migrating a large code from the Cray T3E to the X1[1] and sought help increasing the performance on the loop nest below. This paper describes the analysis his request prompted and some insights we gained in optimizing codes for the X1.

The results are not useful for performance determination, only for "optimization" and estimating relative performance. We think these results are interesting because the simplicity of the code provides insights for efficiently using the X1 and its compilers.

Many of these insights were "obvious" 20 years ago[2] and have been forgotten with the move from vector to single program, multiple data processing. The complications of vector/SPMD processing of the X1 with its multi-level memory hierarchy and improved compiler make some of the old rules-of-thumb inappropriate, as we show.

For example, on the X1, with its high ratio of vector to scalar processing speed, one might expect vectorization to be of paramount importance. With these simple test loops, all but the extremely slowest were vectorized, which indicates that vectorization, though necessary for high performance, is hardly sufficient.

The submitted code fragment is a normalization and the original code looked particularly inefficient for the X1. We rewrote the original code in four alternate forms, one of which executed about two orders of magnitude faster than the slowest. This prompted the extensive analysis of this code snippet.

We also analyzed a second three-loop nest with a slightly more complex calculation, part of which vaguely resembles a PDE integration step. The performance tests were done on a single MSP on ARSC's Cray X1, Klondike.

The original code test, like all the tests in the first set, performed the calculation:

```
do i=1, n1
  do j=1, n2
    do k=1, n3
      u3(i,j,k) = x3(i,j,k)/d3(i,j,k)
      v3(i,j,k) = y3(i,j,k)/d3(i,j,k)
      w3(i,j,k) = z3(i,j,k)/d3(i,j,k)
    enddo
  enddo
enddo
```

The only unusual characteristic of the data is the middle dimension of the arrays was set at n2+2, apparently halo cells for faster calculation on a Cray T3E. The tests in the first set involved changing the order of the loops, the style of indexing, and the compilation options, but all performed the calculation above.

For the first set of tests, the dimensions of the arrays were set at (128, 130, 128) and the data precision at 64-bits so the calculation was performed on a 128 x 128 x 128 cube. Performance on the 72 variations of the calculation varied from 2.1 to 818 MFLOPS (394:1 ratio). Changing the dimensioning to operate on a cube of 129 elements on a side, raised the lowest speed to almost 8 MFLOPS so the speed variation was about 105:1.

The second set of tests performed this calculation

```
do i=2, n1-1
  do j=2, n2-1
    do k=2, n3-1
      u3(i,j,k) = (v3(i,j,k)*((x3(i+1,j,k) -                &
                  - x3(i-1,j,k)) + (x3(i,j+1,k)-x3(i,j-1,k)) &
```

```
            + (x3(i,j,k+1)-x3(i,j,k-1)))                    &
             +  w3(i,j,k)**y3(i,j,k) ) * sin(z3(i,j,k))
      z3(i,j,k) = w3(i,j,k)**y3(i,j,k)
      if (i==j .or. j==k) then
        u3(i,j,k) = 1.0
      endif
    enddo
  enddo
enddo
```

The dozens of coding variations of this second calculation on the interior of an n1 x n2 x n3 cube produced speeds on the X1 from 6 to 537 MFLOPS. This second snippet in a dozen variations was also run on an IBM P655+/P690+ system, where the speed varied from 11 to 57 MFLOPS, only 6% of the variation of the Cray X1.

In the next section we will describe the code and data set variations and some of our motivations for trying them. Section 3 will describe the analysis methodology and present important performance results and a pointer to the complete results and to our thousands of separate raw timing data. We will also describe the compilers' feedback on their optimization of the various forms. Section 4 summarizes the results, describing the user-level information available and how well it predicts performance and makes a few style suggestions for application programmers.

2. Test and Code Variations

The "cube" being operated on was parameterized as n1 x n2 x n3 so the dimensions of the arrays in the first tests were (n1, n2+2, n3). In the second set of tests, because differencing uses adjacent values, the arrays were dimensioned (n1, n2, n3) and the operations were conceptually performed only on the interior points, a (n1-2) x (n2-2) x (n3-2) cube. The MFLOPS rates presented here are based on user operation counts, not hardware reported values.

2.1 Indexing

The submitted code used 1-dimensional indexing and precomputed the values in the array index.

```
index(i,j,k) = i+(j-1)*n1+(k-1)*n1*(n2+2)
```

(The innermost loop was

```
do k=1, n3
    ind = index(i, j, k)
    u3(ind) = x3(ind)/d3(ind)
    v3(ind) = y3(ind)/d3(ind)
    w3(ind) = z3(ind)/d3(ind)
enddo
```

for the 1-dimensional code variation.)

Though single subscripting may improve performance on old machines by replacing index calculations by index look-ups, it tends to reduce the cache's effectiveness on modern systems. By substituting the calculation for the table lookup, we thought the number of memory references would be reduced and the compiler would be more likely to vectorize or multi-stream the entire loop nest.

Using triply dimensioned arrays, the entire index calculation should be obvious and should allow the compiler complete optimization freedom.

Because Fortran stores arrays with the first index varying most rapidly, sometimes called column-major order, the stride of the vectors through memory is large as written above. In many situations this will allow use of only one datum from each cache line before it is removed from cache. For C coding, ordering the loops as shown is desirable. We ran tests with the structure shown in Section 1 and with the loop order changed. We refer to the order shown in the sample above as ijk order and that with the first and third statements interchanged as kji order.

2.2 Inter-loop Dependencies and Compiler Directives

With any loop structure, the compiler may not be able to determine if there is a hazardous recurrence that might cause vectorization or multi-streaming to produce wrong answers. We ran all of our test loops both without and with compiler directives (Fortran) or pragmas (C) to ignore vector dependencies.

2.3 Full Arrays

As noted, the middle dimension of the arrays in the first set of tests is 2 elements longer than the computation. We wondered if computing on the entire array, doing the extra percent of so of calculations, by setting the middle loop statement to

```
do j=1,n2+2
```

would improve vectorization by operating on contiguous blocks of memory. We tested this hypothesis with negative results. This variation was only tried on the Fortran tests on the first loop structure.

2.4 Inlineable Code

Good coding practice dictates using small blocks of code. In the context of these tests, this means coding the computation as a separate function or subroutine. For these loops, we chose values of n1, n2 and n3 from 128 to 517 so the number of floating point operations ranged from about 6M to 415M. With such large iteration counts, it might seem that coding the loop inline, instead of making an external function or subroutine call would provide little benefit. Prefetching of vectors and possible additional knowledge the compiler may use from the calling context encouraged us to try the functions both in the same file, so they are readily inlineable, and in a separate file.

2.5 Language & Compiler

There is much folklore or urban-legend level information saying that C is superior to Fortran or vice versa. We decided to explore the performance differences between Fortran and C. We coded the loops in C using subscripts, not pointers. For the C tests the entire program was coded in C—no interlanguage procedural calls were made.

2.6 Compile Statement "Optimization" Level

We ran the tests after recompiling with the compiler optimization flags set to ask for highly optimized and "aggressive" speedup: –O3,aggress (Fortran) or –h aggress, -O3 (C). We chose these two options because we observe that most users do not experiment extensively with compiler flags, but use either what come for free, the defaults, or try the simple flags

that ask for extensive and aggressive optimization.

2.7 Arithmetic and Data Precision

The original tests were run in 64-bit mode because that is most commonly used on Cray X1 applications. We decided to run tests using 32-bit and 128-bit data by declaring kind = 4 or kind = 16. Because the X1 has 32-bit floating point hardware, unlike many earlier Crays, we expected it to run about twice as fast as the kind = 8 runs. The Cray does not have 128-bit floating point hardware, so extensive software is used to support the greater precision and we expected it to run much more slowly.

2.8 Memory Bank Conflicts

As our last test parameter, we varied the array dimensions. Vectors with strides that are powers of 2 often load and store slowly because of memory bank conflicts and low cache utilization. The severity of code slowdown due to memory bank conflicts depends on many factors and compilers can often mitigate it by interchanging loop order as described in section 2.2. The way we experimented with bank conflicts for our test code was to change the first and last array dimensions from 128 to non-power-of-2 values.

Changing the dimensioning of the arrays, reduced the number of performance bands, indicating its importance.

2.9 Summary of Test Variations

The set of options compared is:

1. Singly or triply subscripted arrays.
   a. For singly subscripted arrays: index computed or looked up in a table
2. Add compiler directives (Fortran) or pragmas (C) telling the compiler there are no inter-iteration dependencies.
3. For the first set of tests: computing on the full n1 x (n2+2) x n3 memory block or only on the significant n1 x n2 x n3 portion. MFLOPS rates for these tests assumed 3*(n1 * n2 * n3) computations.
4. Placing the function code in the same file as the calling routine (so it's inlineable) or in a separate file.
5. Writing the entire code in Fortran or in C,
6. Compiling with default or aggressive optimizations.
7. Varying the data precision: kind = 4, 8 or 16.
8. Adjusting the sizes of the arrays. Eight values were tested: 128, 129, 131, 144, 256, 319, 473, 517, though only the smallest sizes, 128 and 129 were used extensively.

In nearly all cases several tests were run to help verify that the timings were reasonable. The duplicate runs also give us a measure of execution time variability. Loop execution time "standard deviations" were generally on the order of a few percent and never exceeded 15% for those cases where many timing runs were made. ("Standard deviations" were provided by Excel's stdev function.)

3. Selected Details and Observations

Details of the runs, including much of the raw data, are available on the web at [3]. We noted a number of characteristics of the program that affected performance. In all cases we were studying the code, compilation and results as an application programmer. We did not consult assembly language listings. We did use the listing option for the compiler to indicate how it has treated each loop.

We timed our test loops with:

```
        Start timer
        Call subroutine              (Fortran)
or      testNo = function (args)     (C)
        Stop timer
        Call checkRoutine
```

The checkRoutine was inserted to guarantee that the compiler could not fuse loops or optimize across tests. It verified that the computations were correct and reset the values in the arrays. The speeds were computed assuming 3 floating operations per iteration for the simple loop test and 20 for the more complicated code. The code of each subroutine was the necessary declarations followed by the code shown in Section 1 or that code plus compiler directives/pragmas. The values of n1, n2 and n3 were parameters in Fortran and defined values in C.

Tables 1 and 2 show the aggressively compiled loop speeds on the 72 simple test loops for 64-bit data. The performance falls into eight bands, with large gaps between them. With the parameters set at 128, there were:

1. 22 Fortran and 4 C cases with speeds between 742 and 819 MFLOPS
2. 3 Fortran and 1 C cases between 555 and 596 MFLOPS
3. 8 Fortran and 4 C cases between 149 and 169 MFLOPS
4. 3 Fortran and 3 C cases between 55 and 82 MFLOPS (the Fortran ones were between 62 and 65 MFLOPS)
5. 4 Fortran and 1 C cases between 23.7 and 24 MFLOPS
6. 8 Fortran and 5 C cases between 6.6 and 6.9 MFLOPS
7. No Fortran and 3 C cases at 4.9 MFLOPS
8. No Fortran and 3 C cases at 2.1 MFLOPS

Only half as many C cases were run as Fortran. All the Fortran loops vectorized and all but the six in the two lowest bands vectorized for C. (Complete test results are available at [3].)

3.1 Indexing

Nine of the 22 Fortran and half of the C cases in Band 1 (highest performance) used a single subscript, the rest used triply subscripted arrays. On the flip side, in Fortran 13 of 16 triply-subscripted cases were in the top band and the last three in band 4. In C, there were triply subscripted array cases in every band except number 5. The compiler's feedback showed more vectorization and multistreaming with computed subscripts instead of a table lookup. All the triply subscripted array loops vectorized and multistreamed; some of the others did not. Though vectorization was necessary for top speed, it was not sufficient, and was not a good predictor of top performance.

3.2 Compiler Directive Use

The IVDEP compiler directive seemed to have little effect on the simple loops with Fortran cases scattered in every band except the second, nearly always running at about the same speed as the equivalent case without the directive. For the more complex second set of tests, the directives significantly improved both the vectorization/multistreaming and the per-

formance.

In C, pragmas had a positive affect. No case in the lowest two bands used IVDEP pragmas. The cases with the pragmas were relatively evenly scattered in the top six bands.

3.3 Computing on Full Array

No significant speed or vectorization differences were found when we changed the middle index of the simple code snippet to run over the entire array.

3.4 Inlineable Functions

While the Fortran compiler's vectorization notes were changed by making the functions inlinable, the performance differences were small, typically only a few percent.

In C, all the fastest cases, those in bands 1 and 2 and half of those in 3, in-lined the function code. For the slowest tests, all those in bands 7 and 8, the function code was in a separate file and was not in-lined by the compiler.

3.5 Language Choice: Fortran or C

For these simple loops, Fortran is clearly superior overall. Use of pragmas and putting function code in the same file as the calling routine is required for performance parity. Further, as noted below, aggressive optimization was important for good C performance, but made little difference in Fortran. On 6 loops, including two of the triply subscripted ones, the C compiler did not vectorize the code. These loops had the longest execution times and were the tests in performance bands 7 and 8.

3.6 Compiler Options

For the Fortran loops, inserting the compiler optimization flags –O3,aggress had no affect on vectorization or multi-streaming as shown by the compiler's indication of loop handling. The speedup with optimization averaged 7% and ranged from a 10% slowdown to a 75% speedup. The one loop that showed significant slowdown also had large variability in speed. The "standard deviation" was 5% on the five –O3,aggress runs. The only other loop with a slowdown of more than 2%, ran 6% slower and had a standard deviation of 6.4%. Both of these loops were vectorized and multi-streamed and ran at about 585 MFLOPS.

For the C loops, inserting the optimization flags –h aggress, -O3 had considerable impact on vectorization and multi-streaming. While half the loops showed no speedup, the others changed drastically so the average speedup was 30 times. One loop increased in speed by a factor of 286. The Cray documentation indicates that –O3 increases vectorization when pointers are used. This code used no pointers—the C functions were written with one or three subscripts.

The major factor in the speed difference seems to be the default compilation showed no multi-streaming. All the loops that ran significantly faster with aggressive optimization were ones the compiler multi-streamed.

3.7 Data Precision

Contrary to expectation, reducing the floating point precision to 32-bits did not double the speed. Many of the test cases showed no speed increase. The average speed increase was about 50%. The 32-bit speeds ranged from 62% to 235% of those with 64-bit data.

Using 128-bit floating point requires software arithmetic that is much slower on the X1 than its standard 64-bit operations. On the first set of loops, where the computations are very simple, the 128-bit arithmetic ran at about 20% of the 64-bit speed for arrays dimensioned 128, and about 9% for array dimensions of 129. For the more computationally complicated second set of loops, the 128-bit speed was typically about 1% of that for 64-bit.

3.8 Memory Bank Conflicts

Efficient use of the memory on the X1 [1] requires using many of its banks by avoiding dimensions that are a multiple of 16. Increasing the values of the parameters to 129 from 128 meaning the arrays were dimensioned (129, 131, 129), smoothed performance considerably. The performance bands blurred so only six were obvious. The bottom one disappeared and the top two merged. The remaining lowest performance band had only a single test loop.

With parameters at 129 and using aggressive optimization, the slowest Fortran loop ran at 113 MFLOPS. Oddly, the fastest loops slowed by a few percent. The speed variability (ratio of best to worst) was reduced from 122 to 7.

In C, with aggressive optimization, the ratio of the best to worst speed was reduced from 390 to 105.

3.9 Comparison to an "Ordinary" Supercomputer

We ran the first set of tests on Iceberg, our IBM P655+/P690+ system. IBM's speed variation was 3.2% (array dimensions = 129) or 4.3% (array dimensions = 128) of Cray's. The total speed variation for the Fortran code on the IBM system was 11 to 57 MFLOPS with the speeds falling into two or three bands.

4. Summary of Results

Neither the code structure nor the compiler's optimization notes allow one to easily predict the code segment's performance. The lack of clear correlation is especially noticeable for operation on a cube of 128 elements per side. For the average programmer we believe that timing code segments is required to understand performance on the X1.

For programmers working on numeric-intensive applications we suggest the following coding guidelines (some have been known for decades[2]):

1. Focus on vectorizing code hot-spots
2. Structure arrays or order loops so that the innermost loop index corresponds to the first subscript in multi-dimensioned Fortran arrays or the last subscript in C arrays.
3. Avoid multiple-of-16 subscript ranges
4. Use Fortran
5. In C use aggressive optimization and, when correct, use pragmas to encourage vectorization and multistreaming.

6. Check the performance directly; don't rely on static analyses. (Trust only profiling--don't believe in vectorization.)

5. References

[1] www.cray.com/products/systems/x1/ and http://www.cray.com/craydoc/manuals/S-2315-52/html-S-2315-52/x4666.html

[2] Higbie, L., Vectorization and Conversion of Fortran Programs for the CRAY-1 (CFT) Compiler, Publication No. 2240207, Cray Research, Inc., Mendota Heights, Minnesota, June 1979. Several subsequent publications have elaborated and refined this work.

[3] www.arsc.edu/~higbie/<exact location to be determined>

[4] Experience with the Full CCSM (PDF), J. B. Drake, P. H. Worley, I. Carpenter, M. Cordery, in Proceedings of the 46th Cray User Group Conference, Knoxville, TN, May 17-21, 2004, available at http://www.csm.ornl.gov/evaluation/PHOENIX/index.html. There are several other reports referenced at this web site. Most deal with scalability and with complete applications.

6. Contact and Biographical Information

Lee Higbie is a Vector Specialist at the Arctic Region Supercomputing Center of the University of Alaska, Fairbanks. He can be reached at higbie@arsc.edu or 907-450-8688. He has been working in supercomputing for more than three decades.

Tom Baring is a Vector Specialist at the Arctic Region Supercomputing Center of the University of Alaska, Fairbanks. He can be reached at baring@arsc.edu or 907-450-8619. He has worked on Crays for almost a decade, with a focus on usability and high performance.

Ed Kornkven is a Vector Specialist at the Arctic Region Supercomputing Center of the University of Alaska, Fairbanks. He can be reached at kornkven@arsc.edu or 907-450-8669. His professional interests include programming models, languages and compilers for high performance computing.

Array Dimensions	IVDEP Compiler Directive?	1-D Index Generation	Loop Order	Inlinable Functions?	n2+2 Iterations?	Compiler	Execution Time, ms	User MFLOPS	Compiler Optimization Info					
									Inline			In Fnctn		
3	no	-	kji	yes	yes	ftn	7.7	818.1		Vp	M	M	C	V
1	no	Com	kji	no	no	ftn	7.7	816.7				M	C	V
1	yes	Com	kji	no	yes	ftn	7.7	815.4				M	C	V
1	no	Com	kji	yes	yes	ftn	7.7	813.9		Vp	M	M	C	V
3	yes	-	kji	no	no	ftn	7.8	810.1				M	C	V
1	no	Com	kji	yes	no	ftn	7.8	809.5		Vp	M	M	C	V
3	no	-	kji	yes	no	ftn	7.8	807.3		Vp	M	M	C	V
3	yes	-	kji	yes	yes	ftn	7.8	806.9		Vp	M	M	C	V
1	yes	Com	kji	yes	yes	ftn	7.8	804.9		Vp	M	M	C	V
1	yes	Com	kji	no	no	ftn	7.8	804.0				M	C	V
1	yes	Com	kji	yes	no	ftn	7.8	803.6		Vp	M	M	C	V
3	no	-	kji	no	no	ftn	7.9	800.3				M	C	V
3	no	-	ijk	no	yes	ftn	7.9	799.1				MV	C	C
3	yes	-	kji	no	yes	ftn	7.9	797.9				C	C	MV
3	yes	-	kji	yes	no	ftn	7.9	796.6		Vp	M	M	C	V
3	yes	-	ijk	no	yes	ftn	7.9	792.4				MV	C	C
3	no	-	kji	no	yes	ftn	8.0	790.7				C	C	MV
1	no	Com	kji	no	yes	ftn	8.0	789.2				C	C	MV
3	no	-	ijk	yes	no	ftn	8.0	783.1	r	V	M	Vm	C	Mr
3	no	-	ijk	no	no	ftn	8.1	780.6				Vm	C	Mr
3	no	-	ijk	yes	yes	ftn	8.1	778.9	r	V	M	Vm	C	Mr
1	no	Com	ijk	no	yes	ftn	8.5	742.6				Vm	M	r
1	no	Com	ijk	yes	yes	ftn	10.7	585.5	r	V	M	Vm	M	r
1	no	Com	ijk	yes	no	ftn	10.8	584.7	r	V	M	Vm	M	r
1	no	Com	ijk	no	no	ftn	11.3	555.6				Vm	M	r
1	no	TLU	kji	yes	yes	ftn	37.4	168.0		Vw	Vp			Vw
1	yes	TLU	kji	yes	yes	ftn	37.5	167.6		Vw	Vp			Vw
1	yes	TLU	kji	no	no	ftn	37.7	167.0						Vw
1	no	TLU	kji	no	yes	ftn	37.7	166.7						Vw
1	yes	TLU	kji	yes	no	ftn	37.9	166.0		Vw	Vp			Vw
1	no	TLU	kji	no	no	ftn	38.0	165.7						Vw
1	yes	TLU	kji	no	yes	ftn	38.4	163.7						Vw
1	no	TLU	kji	yes	no	ftn	39.5	159.3		Vw	Vp			Vw
3	yes	-	ijk	yes	no	ftn	97.4	64.6	Vw	Vp	M	M	C	Vw
3	yes	-	ijk	yes	yes	ftn	100.8	62.4	Vw	Vp	M	M	C	Vw
3	yes	-	ijk	no	no	ftn	101.1	62.2				M	C	Vw
1	yes	Com	ijk	yes	yes	ftn	261.8	24.0	Vw	Vp	M	M		Vw
1	yes	Com	ijk	no	no	ftn	263.0	23.9				M		Vw
1	yes	Com	ijk	no	yes	ftn	263.2	23.9				M		Vw
1	yes	Com	ijk	yes	no	ftn	264.0	23.8	Vw	Vp	M	M		Vw
1	yes	TLU	ijk	yes	yes	ftn	927.7	6.8		Vw	Vp			Vw
1	no	TLU	ijk	yes	yes	ftn	927.9	6.8		Vw	Vp			Vw
1	no	TLU	ijk	no	yes	ftn	928.3	6.8						Vw
1	no	TLU	ijk	no	no	ftn	928.8	6.8						Vw

Array Dimensions	IVDEP Compiler Directive?	1-D Index Generation	Loop Order	Inlinable Functions?	n2+2 Iterations?	Compiler	Execution Time, ms	User MFLOPS	Compiler Optimization Info		
1	yes	TLU	ijk	yes	no	ftn	928.9	6.8	Vw	Vp	Vw
1	yes	TLU	ijk	no	no	ftn	929.1	6.8			Vw
1	no	TLU	ijk	yes	no	ftn	929.1	6.8	Vw	Vp	Vw
1	yes	TLU	ijk	no	yes	ftn	943.2	6.7			Vw

Table 1. Performance on Fortran Loops. This table shows the loop structure, compiler's analysis output, and the performance. "Useful MFLOPS" is the computed million-floating-operations/sec rate based on the "useful" work, the number of operations in the original code segment. Each row of the table represents the average of several runs. The time variation from run to run was small. These runs are all for n1 = n2 = n3 = 128 and with –O3,aggress compilation option.

Optm Info Key: C - Collapsed m - streamed but not partitioned
M - Multistreamed r - unrolled
V - Vectorized w - unwound

When a loop is collapsed, it is combined with its containing loop in one higher iteration-count loop. An unrolled loop is one that is rewritten to execute several iterations of the loop code for each loop iteration. The loop index is incremented by N and the body of the loop is expanded to operate on N iterations: I, I+1, …I+N-1, say.

Array Dimensions	IVDEP pragma?	1-D Index Generation	Loop Order	Inlinable Functions?	n2+2 Iterations?	Compiler	Execution Time, ms	User MFLOPS	C to Fortran Speed Ratio	Compiler Optimization Info (in function)		
3	no	-	ijk	yes	no	c	7.8	810.7	1.00	M	C	Vr
3	yes	-	ijk	yes	no	c	7.8	807.0	1.01	M	C	Vw
1	no	Com	ijk	yes	no	c	7.8	804.1	0.99	M	C	Vr
1	yes	Com	ijk	yes	no	c	7.9	797.7	0.99	M	C	Vw
3	no	-	kji	yes	no	c	10.6	595.1	0.76	Vm	C	Mr
1	no	TLU	ijk	yes	no	c	37.4	168.2	1.06			Vwr
1	yes	TLU	ijk	yes	no	c	37.5	168.0	1.01			Vw
3	yes	-	ijk	no	no	c	41.6	151.4	0.19			Vw
1	yes	Com	ijk	no	no	c	42.0	149.9	0.19			Vw
1	yes	TLU	ijk	no	no	c	77.0	81.7	0.49			Vw
3	yes	-	kji	yes	no	c	100.0	62.9	0.97	M	C	Vw
1	no	Com	kji	yes	no	c	113.8	55.3	0.09	V	M	r
1	yes	Com	kji	yes	no	c	265.3	23.7	1.00	M	C	Vw
1	yes	Com	kji	no	no	c	917.8	6.9	0.29			Vw
3	yes	-	kji	no	no	c	918.6	6.8	0.11			Vw
1	yes	TLU	kji	yes	no	c	931.9	6.8	1.00			Vw
1	no	TLU	kji	yes	no	c	932.5	6.7	1.00			Vwr
1	yes	TLU	kji	no	no	c	933.3	6.7	1.00			Vw
1	no	TLU	ijk	no	no	c	1281.0	4.9	0.03			r
1	no	Com	ijk	no	no	c	1281.6	4.9	0.01		C	r
3	no	-	ijk	no	no	c	1281.6	4.9	0.01		C	r
3	no	-	kji	no	no	c	3016.9	2.1	0.00			r
1	no	Com	kji	no	no	c	3017.6	2.1	0.00			r
1	no	TLU	kji	no	no	c	3030.4	2.1	0.31			r

Table 2. Performance of C loops. In addition to the computation speed, the ratio of the C code performance to the Fortran performance is shown. Here the C kji loops are compared to the Fortran ijk loops. This means that we are comparing code blocks that access memory in the same order. These are all for n1 = n2 = n3 = 128 and with –O3 –h aggress compilation option.

(See explanation of some codes in Table 1.)