SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 219–226 (1979)

Unrolling Loops in FORTRAN*

J. J. DONGARRA AND A. R. HINDS
Argonne National Laboratory, Argonne, Illinois 60439, U.S.A.

SUMMARY

The technique of 'unrolling' to improve the performance of short program loops without resorting to assembly language coding is discussed. A comparison of the benefits of loop 'unrolling' on a variety of computers using an assortment of FORTRAN compilers is presented.

KEY WORDS Unrolled loops FORTRAN Loop efficiency Loop doubling

INTRODUCTION

It is frequently observed that the bulk of the central processor time for a program is localized in 3 per cent of the source code.[6] Often the critical code from the timing perspective consists of one (or a few) short inner loops typified, for instance, by the scalar product of two vectors. A simple technique for the optimization of such loops, with consequent improvement in overall execution time, should then be most welcome. 'Loop unrolling' (a generalization of 'loop doubling'),[4] applied selectively to time-consuming loops, is just such a technique.

TECHNIQUE

When a loop is unrolled, its contents are replicated one or more times, with appropriate adjustments to array indices and the loop increment. For instance, the DAXPY[9] sequence, which adds a multiple of one vector to a second vector:

```
      DO 10 I = 1,N
        Y(I) = Y(I)+A*X(I)
   10 CONTINUE
```

would, unrolled to a depth of four, assume the form

```
      M = N-MOD(N,4)
      DO 10 I = 1,M,4
        Y(I)   = Y(I)   +A*X(I)
        Y(I+1) = Y(I+1)+A*X(I+1)
        Y(I+2) = Y(I+2)+A*X(I+2)
        Y(I+3) = Y(I+3)+A*X(I+3)
   10 CONTINUE
```

* Work performed under the auspices of the U.S. Department of Energy.

0038–0644/79/0309–0219 $01.00

Received 24 January 1978
Revised 9 October 1978

In this recoding, four terms are computed per loop, with the loop increment modified to count by fours. Additional code has to be added to process the MOD(N,4) elements remaining upon completion of the unrolled loop should the vector length not be a multiple of the loop increment. The choice of four was for illustration, with the generalization to other orders obvious from the example. Actual choice of unrolling depth in a given instance would be guided by the contribution of the loop to total program execution time and consideration of architectural constraints to be discussed below.

It is important to note that the order in which results are calculated is unchanged by unrolling, introducing no new hazards should a result vector overlap a source vector. Note also that the technique is not restricted to any particular source language, thankfully avoiding resort to assembly level coding.

Perhaps not immediately obvious, the success of unrolling in enhancing loop performance can be attributed to three main factors.

First, there is the direct reduction in loop overhead—the increment, test and branch functions—which, for short loops, may actually dominate execution time per iterate. Unrolling simply divides the overhead by a factor equal to the unrolling depth, although additional code required to handle 'leftovers' will reduce this gain somewhat. Clearly, savings should increase with increasing unrolling depth, but the marginal savings fall off rapidly after a few terms. The reduction in overhead is the primary source of improvement on 'simple' computers.

Second, for advanced architectures employing segmented functional units, the greater density of non-overhead operations permits higher levels of concurrency *within* a particular segmented unit. Thus, in the DAXPY example, unrolling would allow more than one multiplication to be concurrently active on a segmented machine such as the CDC 7600.[10] Optimal unrolling depth on such machines might well be related to the degree of functional unit segmentation.

Third, and related to the above, unrolling often increases concurrency *between* independent functional units on computers so equipped. Thus, in our DAXPY example, a CDC 7600, with independent multiplier and adder units, could obtain concurrency between addition for one element and multiplication for the following element, in addition to the segmentation concurrency obtainable within each unit.

Another feature of 'advanced' machines which contributes to the success of unrolling is the increasing use of pipelining in the decoding and preparation of instructions for execution. Testing and branching are inherently disruptive of pipeline flow. The reduction in these disruptions afforded by unrolling ensures a smoother instruction flow on this class of computer.

An important constraint on unrolling depth for certain advanced computers is the presence of an 'instruction stack', affording high performance for loops which can be completely stack contained due to the elimination of instruction fetch time and conflicts with memory operand fetching. Generally, performance is degraded if unrolling increases the size of a loop such that it can no longer be contained in the stack. This infrequently eliminates unrolling as a strategy, more often simply limiting unrolling depth. As an example of this limitation, DAXPY unrolled to a depth of four on the IBM 370/195 performs 27 per cent faster than rolled code when compiled with IBM's FORTRAN H compiler at optimization level 2. But when the code is unrolled to a depth of five, the increase over the rolled code has decreased in comparison to the ratio when the code is unrolled to a depth of four.

EXPERIMENTAL RESULTS

The effects of unrolling a loop to various depths can be seen in Table I. Here we have looked at subroutines DAXPY and DDOT, and taken codings involving one term through

Table I. Unrolling to various depths (vector length 200; IBM 370/195)

DAXPY	Compiler						
Number of terms in loop		1	2	3	4	5	6
Ratio of execution time for n terms vs optimum number of terms in loop	H Opt = 2	1·27	1·15	1·04	1	1·18	1·20
	H Ext Opt = 2 (Release 2.2)	1·24	1·13	1·04	1	1·20	1·16
DDOT	Compiler						
Number of terms in loop		1	2	3	4	5	6
Ratio of execution time for n terms vs optimum number of terms in loop	H Opt = 2	2·37	1·34	1·17	1·08	1	1·11
	H Ext Opt = 2 (Release 2.2)	2·40	1·40	1·19	1·08	1	1·13

six terms in the loop. We ran the various coding on the IBM 370/195 using the H compiler optimization level 2 and H extended compiler optimization level 2. The numbers in Table I represent the time to execute a loop of a given unrolled level normalized by the execution time for the optimum unrolled level coding. Optimum performance level for DAXPY is obtained with four terms in the loop and for DDOT five terms. As more terms are inserted in the loop the performance increases, until the instruction stack is filled. Once past that point, the instructions needed to execute the loop can no longer be maintained in the stack thus causing instruction fetches to be made from memory, thereby degrading performance.

Table II reports the ratios of 'rolled' to 'unrolled' execution times for two FORTRAN loops over a variety of machine–compiler combinations. The operations chosen for comparison are the dot product (DDOT) and multiple of a vector plus a vector (DAXPY) (see Appendix for listings) with vector lengths of 200. The use of ratios rather than actual times isolates the performance improvements due to unrolling, rather than reflecting relative computer speeds or compiler efficiencies.

In the multiprogram environment of modern computers, it is often very difficult to measure reliably the execution time of a program. Significant variations can occur depending on the load of the machine, the amount of I/O interference and resolution of the timing program. The timing data were gathered by a number of people in quite different environments. Therefore, the reported ratios should not be interpreted as absolute but should be useful in giving a feeling for the execution improvements for this technique.

Significantly, unrolling has improved performance in nearly every case, on both simple and sophisticated computers, and with both optimizing and non-optimizing compilers.

Optimum unrolling depth is a function of the machine, the compiler, the release of the compiler and the level of optimization the compiler performs. As a result, true optimum unrolling performance can only be guaranteed with the use of assembly language coding.

Tables III to VI report the variation in unrolling performance as a function of vector length for four machine–compiler combinations. Absolute times and performance ratios are tabulated for DDOT and DAXPY for vector lengths ranging from 10 to 200 on the IBM 370/195 (H Opt = 2), CDC 7600 (FTN Opt = 1), UNIVAC 1110 (FORTV) and PDP–10 (F10 OPT).

Table II. Ratios for versions of DDOT and DAXPY (vector length 200)

Machine	Compiler	Dot product DDOT† $a \leftarrow \sum_{i=1} x_i \cdot y_i$	Multiple of a vector plus a vector DAXPY‡ $\mathbf{y} \leftarrow a \cdot \mathbf{x} + \mathbf{y}$
IBM 360/65	G	1·57	1·37
IBM 360/65	H Opt = 2	1·50	1·00
IBM 360/75	G	1·81	1·58
IBM 360/91	G	1·41	1·40
IBM 360/91	H Opt = 0	1·71	1·50
IBM 370/158	G	1·41	1·26
IBM 370/158	H Opt = 2	1·29	1·19
IBM 370/168–1	H Ext Opt = 2	1·29	1·16
IBM 370/195	H Ext Opt = 0	1·64	1·40
IBM 370/195	H Ext Opt = Release 2.0	2·40	0·97
IBM 370/195	H Ext Opt = 2 Release 2.2	2·40	1·24
CDC 6400/6600	FUN	1·17	1·11
CDC 6400/6600	MNF	1·08	1·04
CDC 6600	Local	1·47	1·18
CDC 6600	FTN, Opt = 1	1·36	1·33
CDC 6600	MNF	0·96	1·03
CDC 6600	FTN Extended Opt	1·78	1·54
CDC 6600	FTN Extended No Opt	0·84	1·36
CDC 7600	Local	1·08	1·21
CDC 7600	CHAT No Opt	1·85	1·45
CDC 7600	CHAT Opt	2·45	1·00
CDC 7600	FTN Opt = 1	2·07	2·25
CDC 7600	FTN No Opt	1·17	1·90
UNIVAC 1110	FORTV	1·43	1·06
UNIVAC 1110	FTN, V	1·80	1·54
Honeywell	FORTRAN-Y Optimized	1·20	1·01
Burroughs 6700	FORTRAN IV	1·23	1·06
PDP–10	F10/opt	1·63	1·46
PDP–10	F40	1·58	1·11

† Loop unrolled to depth of five terms.
‡ Loop unrolled to depth of four terms. (See Appendix for a listing of the codes.)

Table III. Ratios for DDOT and DAXPY (IBM 370/195 H, Opt = 2)

Length	DDOT ratio (rolled/unrolled)	DAXPY ratio (rolled/unrolled)
10	0·9	1·2
20	1·2	1·2
40	1·4	1·1
60	1·6	1·2
80	1·7	1·2
100	1·9	1·3
120	2·0	1·3
140	2·0	1·4
160	2·1	1·3
180	2·2	1·4
200	2·2	1·4

Table IV. Ratios for DDOT and DAXPY
(CDC 7600 FTN No Opt)

Length	DDOT ratio (rolled/ unrolled)	DAXPY ratio (rolled/ unrolled)
10	0·8	1·1
20	1·0	1·6
40	1·0	1·3
60	1·0	1·8
80	1·1	1·8
100	1·1	1·9
120	1·1	1·8
140	1·1	1·9
160	1·3	1·9
180	1·1	1·9
200	1·2	1·9

Table V. Ratios for DDOT and DAXPY
(UNIVAC 1110 FORTV)

Length	DDOT ratio (rolled/ unrolled)	DAXPY ratio (rolled/ unrolled)
10	0·9	0·9
20	1·0	0·9
40	1·2	1·0
60	1·3	1·0
80	1·3	1·0
100	1·4	1·0
120	1·4	1·0
140	1·4	1·0
160	1·4	1·1
180	1·4	1·1
200	1·4	1·1

Table VI. Ratios for DDOT and DAXPY
(PDP–10 F10 OPT)

Length	DDOT ratio (rolled/ unrolled)	DAXPY ratio (rolled/ unrolled)
10	1·4	1·2
20	1·2	1·4
40	1·6	1·3
60	1·7	1·4
80	1·6	1·4
100	1·5	1·5
120	1·6	1·5
140	1·6	1·4
160	1·6	1·4
180	1·6	1·4
200	1·6	1·5

As expected, the benefits of unrolling increase as vector length increases. At small vector lengths, the somewhat longer initialization sequence for the unrolled loop combined with the necessity of cleanup code can lead to small advantage for the rolled version.

All timings and ratios in this paper include the linkage overhead times for the subroutines DDOT and DAXPY. This results in the performance ratios underestimating, particularly for short vector lengths, the benefits of unrolling. The long vector performance is more indicative of the behavior of unrolling in open code.

CONCLUSIONS AND DISCUSSION

An elementary technique, applicable at the source level, has been presented for the performance improvement of short program loops. The method is simple to apply, requires no rethinking of existing algorithms, is applicable in most common programming languages, and only marginally degrades program readability. Data presented demonstrate the utility of 'unrolling' for a wide variety of computer and FORTRAN compiler combinations.

The performance improvements of unrolling result from a reduction in loop overhead operations. As an added benefit, loop unrolling provides greater utilization of special architectural features of advanced machines normally accessible only through assembly or special languages.

We do not suggest that this method be applied to programs blindly, for there are many non-critical parts of a program where the effects will be negligible. But in sections of a code known by the programmer to dominate execution time, loop unrolling can yield dramatic savings. Indeed, the simplicity and success of unrolling suggest that it be made available automatically in future compilers. With the compiler having knowledge of the architecture of a machine, the program speedup can be optimized.

ACKNOWLEDGEMENTS

The routines and timings for this study were obtained during a LINPACK[8] investigation into an efficient FORTRAN coding of the BLAS. We would like to thank the LINPACK test sites for their cooperation in gathering the timings and V. Barr for her help in preparing some of the FORTRAN codings used.

APPENDIX

```
      DOUBLE PRECISION FUNCTION DDOT(N,DX,INCX,DY,INCY)
C
C     FORMS THE DOT PRODUCT OF TWO VECTORS.
C     USES UNROLLED LOOPS FOR INCREMENTS EQUAL TO ONE.
C     JACK DONGARRA, LINPACK, 6/17/77.
C
      DOUBLE PRECISION DX(1),DY(1),DTEMP
      INTEGER I,INCX,INCY,IX,IY,M,MP1,N
C
      DDOT = 0.0D0
      DTEMP = 0.0D0
      IF(N.LE.0)RETURN
      IF(INCX.EQ.1.AND.INCY.EQ.1)GOTO 20
C
C        CODE FOR UNEQUAL INCREMENTS OR EQUAL INCREMENTS
C          NOT EQUAL TO 1
C
```

```
      IX = 1
      IY = 1
      IF(INCX.LT.0)IX = (-N+1)*INCX + 1
      IF(INCY.LT.0)IY = (-N+1)*INCY + 1
      DO 10 I = 1,N
        DTEMP = DTEMP + DX(IX)*DY(IY)
        IX = IX + INCX
        IY = IY + INCY
   10 CONTINUE
      DDOT = DTEMP
      RETURN
C
C        CODE FOR BOTH INCREMENTS EQUAL TO 1
C
C
C        CLEAN-UP LOOP
C
   20 M = MOD(N,5)
      IF( M .EQ. 0 ) GO TO 40
      DO 30 I = 1,M
        DTEMP = DTEMP + DX(I)*DY(I)
   30 CONTINUE
      IF( N .LT. 5 ) GO TO 60
   40 MP1 = M + 1
      DO 50 I = MP1,N,5
        DTEMP = DTEMP + DX(I)*DY(I) + DX(I + 1)*DY(I + 1) +
     *   DX(I + 2)*DY(I + 2) + DX(I + 3)*DY(I + 3) +
     *   DX(I + 4)*DY(I + 4)
   50 CONTINUE
   60 DDOT = DTEMP
      RETURN
      END
```

(Double precision was used on IBM; for all other machines, single precision was used.)

```
      SUBROUTINE DAXPY(N,DA,DX,INCX,DY,INCY)
C
C     CONSTANT TIMES A VECTOR PLUS A VECTOR.
C     USES UNROLLED LOOPS FOR INCREMENTS EQUAL TO ONE.
C     JACK DONGARRA, LINPACK, 6/17/77.
C
      DOUBLE PRECISION DX(1),DY(1),DA
      INTEGER I,INCX,INCY,IXIY,M,MP1,N
C
      IF(N.LE.0)RETURN
      IF (DA .EQ. 0.0D0) RETURN
      IF(INCX.EQ.1.AND.INCY.EQ.1)GOTO 20
C
C        CODE FOR UNEQUAL INCREMENTS OR EQUAL INCREMENTS
C          NOT EQUAL TO 1
C
      IX = 1
      IY = 1
      IF(INCX.LT.0)IX = (-N+1)*INCX + 1
      IF(INCY.LT.0)IY = (-N+1)*INCY + 1
      DO 10 I = 1,N
        DY(IY) = DY(IY) + DA*DX(IX)
        IX = IX + INCX
        IY = IY + INCY
   10 CONTINUE
      RETURN
C
C        CODE FOR BOTH INCREMENTS EQUAL TO 1
```

```
C
C
C        CLEAN-UP LOOP
C
   20 M = MOD(N,4)
      IF( M .EQ. 0 ) GO TO 40
      DO 30 I = 1,M
        DY(I) = DY(I) + DA*DX(I)
   30 CONTINUE
      IF( N .LT. 4 ) RETURN
   40 MP1 = M + 1
      DO 50 I = MP1,N,4
        DY(I) = DY(I) + DA*DX(I)
        DY(I + 1) = DY(I + 1) + DA*DX(I + 1)
        DY(I + 2) = DY(I + 2) + DA*DX(I + 2)
        DY(I + 3) = DY(I + 3) + DA*DX(I + 3)
   50 CONTINUE
      RETURN
      END
```

(Double precision was used on IBM; for all other machines, single precision was used.)

REFERENCES

1. C. B. Kreitzberg and B. Shmeiderman, *The Elements of FORTRAN Style, Techniques for Effective Programming*, Harcourt Brace Jovanovich, New York, 1972.
2. D. Van Tassel, *Program Style, Design, Efficiency, Debugging, and Testing*, Prentice-Hall, Englewood Cliffs, N.J., 1974.
3. D. Pager, 'Some notes on speeding up certain loops by software, firmware, and hardware means', *IEEE Trans. on Comp.*, **C21-1** 97–100 (January 1972).
4. D. E. Knuth, 'Structured programming with Go To statements', *Comp. Surveys*, **6**, No. 4 261–302 (December 1974).
5. F. H. McMahon, L. J. Sloan and G. A. Long, *STACKLIB, LASL, LTSS*–510 (January 1977).
6. D. E. Knuth, 'An empirical study of FORTRAN programs', *Software—Practice and Experience*, **1**, 105–133 (1971).
7. J. J. Dongarra, J. R. Bunch, C. M. Moler and G. W. Stewart, *LINPACK Working Note* #9, *Preliminary LINPACK User's Guide*, ANL TM–313 (August 1977).
8. J. J. Dongarra, *LINPACK Working Note* #3, *FORTRAN BLAS Timing*, Argonne National Laboratory (November 1976).
9. C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh, 'Basic linear algebra subprograms for use with FORTRAN', submitted to *Trans. Math. Software* (July 1977).
10. *CDC Compass Version* 3 *Reference Manual*, 60360900, Control Data Corporation, 1976.