利用 m4 宏语言进行 Fortran 77 循环展开*1)

张 林 波

（中国科学院计算数学与科学工程计算研究所，科学与工程计算国家重点实验室）

FORTRAN DO LOOP UNROLLING USING m4 MACRO LANGUGE

Zhang Lin-bo

(*State Key laboratory of Scientific and Engineering Computing, Institute of Computational Mathematics and Scientific/Engineering Computing, Chinese Academy of Sciences*)

Abstract

Do-loop unrolling is an effective technique for improving performance of application programs. This paper presents a method for unrolling nested Fortran DO-loops using the m4 macro language. m4 is a macro processor widely available on UNIX platforms. By using carefully designed m4 macros, DO-loop unrolling becomes much simpler. More over, with this method, code can be written for general unrolling parameters, allowing the program to be easily tuned on different computers to reach optimal performance.

§1. 引　　言

循环展开 (do-loop unrolling) 做为 Fortran 程序的一种简单、有效的优化手段早已为人们所知，被广泛应用于 Fortran 编译系统中. 对于向量机而言，通过对简单的循环适当进行展开可增加每次循环中的计算量，有助于充分发挥向量机上多流水线的并发处理能力，往往可以成倍地提高计算程序的浮点性能. 而在当前占主导地位的基于 RISC 微处理芯片的计算机系统 (包括微机、工作站、共享内存并行机以及 MPP 型计算机等)，处理器内部普遍采用超标量多流水线结构，可并发执行多条浮点指令，能够达到每秒数亿次的浮点运算峰值性能. 对内存的访问成为影响应用程序的实际性能的主要瓶颈之一，因而经常采用多级高速缓存、指令和数据预取等技术来缓解这一

* 1996 年 9 月 6 日收到.

1) 本工作在中国科学院软件所 "并行软件研究开发中心" (RDCPS) 完成.

矛盾，提高寄存器和高速缓存中数据的重复使用率(后者通常称为 cache 命中率)、挖掘处理器内部的并发处理能力是提高这类机器上应用程序实际性能的主要手段，适当的循环展开对计算性能的提高大有帮助. 虽然大部分 Fortran 编译系统都有自动进行循环合并、展开的能力，但由于其中牵涉到对循环内部数组变量的相关性分析，对多重循环效果往往不太理想，因此依然经常采用人工的方式进行循环展开.

我们称直接在 Fortran 程序中将 DO 循环写成展开的形式为手工循环展开，这是当前普遍使用的方法. 采用手工循环展开主要有两方面的缺陷：①源程序的长度随着循环展开的步数以及循环嵌套的重数急剧增加，大大削弱了源程序的可读性，也不便于程序的维护，同时在编写程序时容易出错；②对于编写好的程序，要改变循环展开的步数比较困难，不便于程序的移植，因为在不同类型的计算机上通常需要使用不同的循环展开步数来获得理想的计算性能. 基于这些原因，有必要研究、开发用于 Fortran 程序中循环展开的特殊工具.

m4[2] 是 UNIX 系统中的一个通用宏处理程序，它具有很强的宏定义、宏展开、算术运算及逻辑判断能力，并可通过宏的递归调用来实现循环处理. 一种自然的想法就是在 Fortran 源程序中插入 m4 的宏命令，通过 m4 的宏语言来编写循环语句的展开. 编译这样的源程序时，首先用 m4 进行预处理，产生一个标准 Fortran 程序，然后再调用 Fortran 编译系统进行编译. 在用 m4 进行预处理的过程中可任意指定循环展开的步数，这样便做到了源程序与循环展开步数无关，可以通过调整循环展开步数来优化程序的性能.

基于上述思想我们对一些线性代数计算中典型循环的循环展开进行了研究和实验，结果发现，使用 m4 宏语言不仅能编写出可任意指定循环展开步数的程序，而且可以大大简化循环展开的过程. 通过引进适当的宏定义，包含循环展开的 m4 程序可以做到几乎与不包含循环展开的 Fortran 程序一样简洁. 此外，由于 m4 在任何 UNIX 系统上均可使用，该方法亦具有很好的通用性.

我们将在 §2 节中简要介绍 Fortran 中的循环展开，在 §3 节中介绍用 m4 进行 Fortran 循环展开的方法，在 §4 节中介绍一种自动确定最优循环展开步数的方法并给出在不同类型的计算机上进行循环展开的性能比较.

§2. Fortran 77 中的循环展开

所谓循环展开，是指将一个循环体中的数步合并为一步，减少总的循环次数，增加每次循环中的计算工作量和涉及的变量. 下面是一个计算向量内积的简单例子：

```
T=0.0                          T=0.0
DO I=1,N                     * Clean up loop
   T=T+A(I)*B(I)                DO I=1, MOD(N,3)
ENDDO                              T=T+A(I)*B(I)
                                ENDDO
                             * Main loop
```

```
                        DO I=MOD(N,3)+1, N, 3
                           T=T+A(I)*B(I)+A(I+1)*B(I+1)+A(I+2)*B(I+2)
                        ENDDO
```

其中左边为原始循环，右边是对它进行 3 步循环展开后的 Fortran 程序. 展开后的程序由两个循环构成，第一个循环先对 N 模 3 的余数进行处理，第二个循环才是主循环体.

再看一个两重循环的例子 (矩阵转置乘向量):

```
DO J=1, N                  * --- Clean up loop on J
   T=0.0                         DO J=1, MOD(N,2)
   DO I=1, N                        TO=0.0
      T=T+A(I,J)*X(I)      * ------ Clean up loop on I
   ENDDO                            DO I=1, MOD(N,2)
   Y(J)=T                              TO=TO+A(I,J)*X(I)
ENDDO                               ENDDO
                           * ------ Main loop on I
                                    DO I=MOD(N,2)+1, N, 2
                                       TO=TO+A(I,J)*X(I)+A(I+1,J)*X(I+1)
                                    ENDDO
                                    Y(J)=TO
                                 ENDDO
                           * --- Main loop on J
                                 DO J=MOD(N,2)+1, N, 2
                                    TO=0.0
                                    T1=0.0
                           * ------ Clean up loop on I
                                    DO I=1, MOD(N,2)
                                       TO=TO+A(I,J)*X(I)
                                       T1=T1+A(I,J+1)*X(I)
                                    ENDDO
                           * ------ Main loop on I
                                    DO I=MOD(N,2)+1, N, 2
                                       TO=TO+A(I,J)*X(I)+A(I+1,J)*X(I+1)
                                       T1=T1+A(I,J+1)*X(I)+A(I+1,J+1)*X(I+1)
                                    ENDDO
                                    Y(J)=TO
                                    Y(J+1)=T1
                                 ENDDO
```

上例中对 I 和 J 循环各展开两步，其中在对 J 处理 N 的余数的循环中也对 I 的循环进行了展开. 在目前大部分计算机上采用上例中右边的代码通常可以比采用左边的代码获得更好的浮点性能.

从这些例子中可以看出，循环展开后程序的长度随循环嵌套的重数急剧增加，而且在编写代码时很容易在下标变量中发生错误. 另外，这些例子中只能写出给定循环展开步数的代码，程序编写完成后，要改变循环展开的步数比较麻烦.

我们将在下节中介绍如何利用 UNIX 系统中的 m4 宏语言来进行 Fortran 中的循环展开.

§3. 利用 m4 宏语言进行 Fortran 循环展开

m4 是 UNIX 系统中的通用宏处理程序. 它在处理时将输入文件的内容拷贝至输出文件, 并同时对输入文件中的宏命令进行处理, 进行宏展开. 输入文件中的宏命令可以是 m4 的内部命令, 也可以是用户定义的. 限于篇幅, 我们不在这里对 m4 的功能及用法进行详细介绍. 有关 m4 的来历及详细说明可以在许多关于 UNIX 操作系统的书中找到, 有兴趣者也可参阅 [2], 在 UNIX 系统下亦可用 man m4 命令来获得有关说明. 在后续介绍中我们做如下两点假设:

1) 我们采用由著名免费软件发行组织 GNU 所发行的 GNU m4[2] 来对包含 m4 命令的 Fortran 程序进行预处理, 因为该软件提供了一个命令行可选项 "-P" 使得所有的 m4 内部宏命令名的前面都冠以前缀 "m4_". 在我们自己定义的宏命令中也以 "m4_" 做为宏命令名的前缀. 这样可以避免与 Fortran 中的变量名发生冲突.

2) 为改善程序的可读性, 我们使用 "{" 和 "}" 做为 m4 程序中的 "引号" (quotes).

通过在 Fortran 程序中插入适当的 m4 宏命令, 很容易将循环展开写成可任意指定展开步数的形式. 以 §2 节中的第一个例子为例, 可以将它写成:

```
m4_define( {m4_forloop},
  {m4_pushdef({$1}, {$2})_m4_forloop({$1}, {$2}, {$3},
  {$4})m4_popdef({$1})} )
m4_define( {_m4_forloop},
  {m4_ifelse( m4_eval($1>$3),1,{},
  {$4{}m4_define({$1},m4_incr($1))_m4_forloop( {$1}, {$2}, {$3},
  {$4})})} )

      T=0.0
* Clean up loop
      DO I=1, MOD(N,step)
         T=T+A(I)*B(I)
      ENDDO
* Main loop
      DO I=MOD(N,step)+1, N, step
         T=T m4_forloop(i, 0, m4_decr(step), {+A(I+i)*B(I+i)})
      ENDDO
```

其中 "step" 用于控制循环开展的步数. 我们在源文件开头用下面形式的命令

```
m4_ifdef({step}, {}, {m4_define({step}, 3)})
```

来为 "step" 设定一个省缺值, 这样可在 m4 的命令行上通过 "-D" 可选项来改变 "step" 的值, 如:

```
m4 -P -Dstep=5 ...
```

将 "step" 设为 5. 当命令行上没有指定 "step" 的值时, 它将被赋予缺省值 3.

上面的例子只是为了说明利用 m4 进行循环展开的基本思想. 直接用这种形式编写多重循环展开仍是相当麻烦的. 为此我们定义了一组宏命令, 其中最基本的有以下三条命令:

```
m4_do, m4_expand, m4_local
```

这些宏命令的定义包含在一个文件 defs.m4 中. 限于篇幅, 我们不打算在此对它们做详细的介绍, 从下面给出的例子不难看出它们的用法, 有兴趣的读者可参考 [1].

m4 处理时, 输入文件中包含的许多空格、空行会被带到输出中. 由于 Fortran 77 的源程序书写格式有严格的要求, 因此会给源文件中 m4 宏命令的书写格式带来很大的限制, 影响源文件的可读性. 为了解决这个问题, 我们编写了一个后处理程序 m4post.c 用来对 m4 的输出进行整理, 以放宽对源文件格式上的限止. 例如, m4post 允许 m4 输出的 Fortran 程序的每行从任何一列开始, 当第一个非空字符为 "&" 时表示是连续行, 等等. 它按照 Fortran 77 的格式要求自动删除多余的空格、空行, 并尽可能合并多个连续行, 保证输出的程序符合 Fortran 77 的格式.

为了避免在语句标号上发生冲突, 我们所定义的宏命令中均采用 "DO ... ENDDO" 形式的 Fortran 循环语句, 它属于 Fortran 77 的扩展形式, 但已被目前绝大部分 Fortran 77 编译器所接受. 我们也编写了一个名为 do-enddo.c 的后处理程序, 必要时可使用它自动将 "DO ... ENDDO" 形式的语句转换成带标号的 "DO" 语句, 使得最终输出的 Fortran 程序严格符合 Fortran 77 的语法要求.

m4post 和 do-enddo 程序均以 UNIX 的标准输入文件为输入, 标准输出文件为输出. 假设包含 m4 宏命令的 Fortran 源程序为 prog.m4, 则将它转换成标准 Fortran 程序 prog.f 的处理命令行可利用 UNIX 的管道功能写成如下形式:

```
m4 -P defs.m4 prog.m4 | m4post | do-enddo > prog.f
```

当然, 在上述命令行中还可插入一些 "-D" 可选项来设定一些控制变量的值.

上述处理过程中没有对产生的 Fortran 程序进行结构化的整理 (如循环体向右错格排列等). 必要时可使用一些通用工具如 ToolPack[6] 来进行进一步的整理.

下面我们用几个典型的例子来说明如何利用上述宏命令来编写 Fortran 77 的循环展开程序. §2 节中第一个例子的循环展开可简单地写成:

```
t=0.0
m4_do(i, 1, n, 1, step, {
   t=t m4_expand(i, {&+a(i)*b(i)})
})
```

其中, m4_do 的第一个参数为 Fortran DO 循环的循环变量, 第二、三和第四个参数分别为循环的初值、终值和增量, 最后一个参数为循环展开的步数. m4_expand 的第一个参数为循环变量, 第二个参数为要展开的内容. 对于多重循环, m4_expand 的参数个数可以大于 2, 前面的参数分别给出对应的 Fortran 循环变量, 而最后一个参数则给出展开的内容. 注意 "+" 号前面的 "&" 字符, 它使得输出的 Fortran 程序具有如下的形式:

```
&+a(i)*b(i)
&+a(i+1)*b(i+1)
...
```

即由循环展开而生成的每一项均以连续行的形式独占一行，这样可以避免一行的长度超过 72 列. m4post 程序会尽可能地对这些短的连续行进行合并处理. 因而，当 step = 3 时，最终输出的 Fortran 程序与 §2 节中用手工编写的循环展开完全一样.

下面是一个计算矩阵乘法的三重循环的例子:

```
      DO K=1, L
         DO J=1, M
            DO I=1, N
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
            ENDDO
         ENDDO
      ENDDO
```

它的循环展开可写成:

```
m4_do(k, 1, l, 1, step_k, {
   m4_do(j, 1, m, 1, step_j, {
      m4_do(i, 1, n, 1, step_i, {
         m4_expand(i, j, { c(i,j) = c(i,j) m4_expand(k, {&+ a(i,k)*b(k,j)}) })
      })
   })
})
```

当 step_i = step_j = step_k = 2 时，上述代码经 m4 及 m4post 处理后将生成如下的 Fortran 程序段 (它没有经过任何人工处理，只是为了便于阅读用 ToolPack 对它进行了适当整理).

```
*     m4: start expansion
      ITMPK = MIN(L,MOD(L,2))
*     m4 - Clean up loop on k
      DO 130 K = 1,ITMPK
         ITMPJ = MIN(M,MOD(M,2))
*        m4 - Clean up loop on j
         DO 90 J = 1,ITMPJ
            ITMPI = MIN(N,MOD(N,2))
*           m4 - Clean up loop on i
            DO 70 I = 1,ITMPI
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
   70       CONTINUE
*           m4 - Unrolled loop on i
            DO 80 I = MAX(1,ITMPI+ (1)),N,2
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
               C(I+1,J) = C(I+1,J) + A(I+1,K)*B(K,J)
   80       CONTINUE
   90    CONTINUE
*        m4 - Unrolled loop on j
         DO 120 J = MAX(1,ITMPJ+ (1)),M,2
            ITMPI = MIN(N,MOD(N,2))
*           m4 - Clean up loop on i
            DO 100 I = 1,ITMPI
```

```
                C(I,J) = C(I,J) + A(I,K)*B(K,J)
                C(I,J+1) = C(I,J+1) + A(I,K)*B(K,J+1)
  100       CONTINUE
*           m4 - Unrolled loop on i
            DO 110 I = MAX(1,ITMPI+ (1)),N,2
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
               C(I,J+1) = C(I,J+1) + A(I,K)*B(K,J+1)
               C(I+1,J) = C(I+1,J) + A(I+1,K)*B(K,J)
               C(I+1,J+1) = C(I+1,J+1) + A(I+1,K)*B(K,J+1)
  110       CONTINUE
  120    CONTINUE
  130 CONTINUE
*     m4 - Unrolled loop on k
      DO 200 K = MAX(1,ITMPK+ (1)),L,2
         ITMPJ = MIN(M,MOD(M,2))
*        m4 - Clean up loop on j
         DO 160 J = 1,ITMPJ
            ITMPI = MIN(N,MOD(N,2))
*           m4 - Clean up loop on i
            DO 140 I = 1,ITMPI
               C(I,J) = C(I,J) + A(I,K)*B(K,J) + A(I,K+1)*B(K+1,J)
  140       CONTINUE
*           m4 - Unrolled loop on i
            DO 150 I = MAX(1,ITMPI+ (1)),N,2
               C(I,J) = C(I,J) + A(I,K)*B(K,J) + A(I,K+1)*B(K+1,J)
               C(I+1,J) = C(I+1,J) + A(I+1,K)*B(K,J) +
     +                    A(I+1,K+1)*B(K+1,J)
  150       CONTINUE
  160    CONTINUE
*        m4 - Unrolled loop on j
         DO 190 J = MAX(1,ITMPJ+ (1)),M,2
            ITMPI = MIN(N,MOD(N,2))
*           m4 - Clean up loop on i
            DO 170 I = 1,ITMPI
               C(I,J) = C(I,J) + A(I,K)*B(K,J) + A(I,K+1)*B(K+1,J)
               C(I,J+1) = C(I,J+1) + A(I,K)*B(K,J+1) +
     +                    A(I,K+1)*B(K+1,J+1)
  170       CONTINUE
*           m4 - Unrolled loop on i
            DO 180 I = MAX(1,ITMPI+ (1)),N,2
               C(I,J) = C(I,J) + A(I,K)*B(K,J) + A(I,K+1)*B(K+1,J)
               C(I,J+1) = C(I,J+1) + A(I,K)*B(K,J+1) +
     +                    A(I,K+1)*B(K+1,J+1)
               C(I+1,J) = C(I+1,J) + A(I+1,K)*B(K,J) +
     +                    A(I+1,K+1)*B(K+1,J)
               C(I+1,J+1) = C(I+1,J+1) + A(I+1,K)*B(K,J+1) +
     +                      A(I+1,K+1)*B(K+1,J+1)
  180       CONTINUE
  190    CONTINUE
  200 CONTINUE
*     m4: end expansion
```

由上面两个例子可以看出，对于简单循环，不管循环嵌套的重数是多少，用 m4 编写的循环展开几乎与不展开的循环一样简单.

最后，我们以摘自 BLAS 库 DTRSV 程序 [4] 中的一个循环块为例，该循环的特点是内层循环的长度依赖于外层循环变量. 源程序如下：

```
      DO 20, J = N, 1, -1
         IF( NOUNIT ) X( J ) = X( J )/A( J, J )
         TEMP = X( J )
         DO 10, I = J - 1, 1, -1
            X( I ) = X( I ) - TEMP*A( I, J )
   10    CONTINUE
   20 CONTINUE
```

我们将它用 m4 语言写成下述形式：

```
m4_local(t, j)
m4_do( j, n, 1, -1, step_i, {
   m4_texpand(j, {j}, 0, m4_decr(m4_unroll_j_end), 0, {
      if( nounit ) x( j ) = x( j )/a( j, j )
      m4_texpand(i, {j}, m4_incr(_j_), m4_unroll_j_end, 0, {
         x( i ) = x( i ) - x(j)*a( i, j )
      })
   })
   m4_pexpand(j, m4_unroll_j_end, m4_unroll_j_end, -1, {
      if( nounit ) x( j ) = x( j )/a( j, j )
   })
   m4_expand(j,{t = x( j )})
   m4_do(i, j - 1 - m4_unroll_j_end, 1, -1, step_j, {
      m4_expand(i, {x( i ) = x( i )
               m4_expand(j, {&- t*a( i, j )})
      })
   })
})
m4_undefine({t})
```

其中 m4_texpand, m4_pexpand 亦在 defs.m4 文件中定义. 当 step_i = 2, step_j = 3 时它的展开如下：

```
*     m4: start expansion
      ITMPJ = MAX(1,N- (MOD((N- (1))+1,2)-1))
*     m4 - Clean up loop on j
      DO 30 J = N,ITMPJ,-1
         IF (NOUNIT) X(J) = X(J)/A(J,J)
         T0 = X(J)
         ITMPI = MAX(1,J-1-0- (MOD((J-1-0- (1))+1,3)-1))
*        m4 - Clean up loop on i
         DO 10 I = J - 1 - 0,ITMPI,-1
            X(I) = X(I) - T0*A(I,J)
   10    CONTINUE
*        m4 - Unrolled loop on i
```

```
            DO 20 I = MIN(J-1-0,ITMPI+ (-1)),1,-3
               X(I) = X(I) - T0*A(I,J)
               X(I-1) = X(I-1) - T0*A(I-1,J)
               X(I-2) = X(I-2) - T0*A(I-2,J)
   20       CONTINUE
   30    CONTINUE
*        m4 - Unrolled loop on j
         DO 60 J = MIN(N,ITMPJ+ (-1)),1,-2
            IF (NOUNIT) X(J) = X(J)/A(J,J)
            X(J-1) = X(J-1) - X(J)*A(J-1,J)
            IF (NOUNIT) X(J-1) = X(J-1)/A(J-1,J-1)
            T0 = X(J)
            T1 = X(J-1)
            ITMPI = MAX(1,J-1-1- (MOD((J-1-1- (1))+1,3)-1))
*           m4 - Clean up loop on i
            DO 40 I = J - 1 - 1,ITMPI,-1
               X(I) = X(I) - T0*A(I,J) - T1*A(I,J-1)
   40       CONTINUE
*           m4 - Unrolled loop on i
            DO 50 I = MIN(J-1-1,ITMPI+ (-1)),1,-3
               X(I) = X(I) - T0*A(I,J) - T1*A(I,J-1)
               X(I-1) = X(I-1) - T0*A(I-1,J) - T1*A(I-1,J-1)
               X(I-2) = X(I-2) - T0*A(I-2,J) - T1*A(I-2,J-1)
   50       CONTINUE
   60    CONTINUE
*        m4: end expansion
```

上面给出的例子仅供有兴趣的读者参考，我们不打算对它做详细解释. 实际问题中所遇到的循环形式当然比这些例子要复杂得多，但根据笔者的经验，只要引进适当定义的辅助宏命令，总是可以大大简化循环展开程序的编写，达到事半功倍的效果. 读者不妨尝试自行定义一些宏命令来对一些循环进行循环展开，或许会比本文中介绍的命令更为简单、有效.

§4. 自动确定最佳循环展开步数

前面我们曾经提到，在不同类型的计算机上如何确定最佳循环展开步数以达到最好的浮点性能并非一件简单的事情. 直接从理论上对最佳循环展开步数进行估计需要对计算机的结构及所使用的编译系统有较深刻的了解，而对于普通用户来说这是难以做到的. 采用上节中介绍的方法编写的循环展开程序由于可方便地指定循环展开的步数，从而有可能设计一些简单的搜索算法，通过少量的试算来自动确定最佳的循环展开步数. 例如，当程序的浮点性能相对于循环展开的步数是一个单峰函数时，可以使用“瞎子爬山”的方法来寻找最佳循环展开步数. 在一些类型的计算机上，如一些基于向量或伪向量结构的计算机，由于受寄存器数目、浮点运算部件数、数据通道数、编译系统的自动循环展开等的影响，使得程序的浮点性能相对于循环展开的步数不是单峰函数，即使这样，使用“瞎子爬山”的方法仍然可以找到较为满

意的循环展开步数. 当然, 当程序的浮点性能特别重要时, 也可采用在一定范围内进行穷举的方法, 即对一定的范围内所有可能的循环展开步数的组合逐个进行试算, 从中找出具有最好浮点性能的一组做为最佳循环展开步数, 这样做的缺陷是需要进行大量的试算. 在本节中, 我们将介绍一些采用“瞎子爬山”方法寻找最佳循环展开步数的以及在一些类型的计算机上通过循环展开得到的性能改善方面的数值结果.

首先我们简单介绍一下“瞎子爬山”方法. 为了便于描述我们以 3 重循环为例, 此时需要确定三个循环展开步数 (S_1, S_2, S_3), 并假设初始搜索步长为 H (它们均为整数). 则我们在 $(S_1 \pm H, S_2, S_3)$, $(S_1, S_2 \pm H, S_3)$ 和 $(S_1, S_2, S_3 \pm H)$ 这 6 组循环展开步数中任选一组比 (S_1, S_2, S_3) 具有更好的浮点性能的循环展开步数做为下次搜索的出发点, 如果这 6 组循环展开步数的浮点性能都不比 (S_1, S_2, S_3) 好, 此时如果 $H = 1$ 则搜索结束, 即 (S_1, S_2, S_3) 即为最佳循环展开步数, 否则令 $H = H/2$ 继续进行搜索. 当目标函数为单峰函数时 (这里的目标函数可取为实际浮点性能或 CPU 时间的倒数). 从理论上采用该方法应该得到最优的循环展开步数, 但在实际搜索时, 由于每次运行时估算出的浮点性能或 CPU 时间会有所波动, 所以最终找到的循环展开步数往往只是近似最优的 (即它们的浮点性能接近最优的浮点性能).

我们将以下面的程序段为例给出在一些计算机上采用“瞎子爬山”方法进行搜索的结果, 它选自 BLAS 的 DGEMM 程序中计算双精度矩阵转置乘以矩阵转置, 即计算:

$$C := \beta C + \alpha A^T B^T$$

的一段程序. 原始程序如下:

```
      DO 200, J = 1, N
         DO 190, I = 1, M
            TEMP = 0.0
            DO 180, L = 1, K
               TEMP = TEMP + A( L, I )*B( J, L )
180         CONTINUE
            IF( BETA.EQ.ZERO )THEN
               C( I, J ) = ALPHA*TEMP
            ELSE
               C( I, J ) = ALPHA*TEMP + BETA*C( I, J )
            END IF
190      CONTINUE
200   CONTINUE
```

在各种类型的矩阵乘法中, 对它进行循环展开得到的性能改善最为明显. 相应的 m4 程序如下:

```
m4_local(t, i, j)
m4_do(j, 1, n, 1, step_j, {
   m4_do(i, 1, m, 1, step_i, {
      m4_expand(j, i, {t = zero})
      m4_do(l, 1, k, 1, step_l, {
```

```
            m4_expand(j, i, {
               t = t m4_expand(l, {&+a( l, i )*b( j, l )})
            })
         })
         if( beta.eq.zero )then
            m4_expand(j, i, {c( i, j ) = alpha*t})
         else
            m4_expand(j, i, {c( i, j ) = alpha*t + beta*c( i, j )})
         end if
      })
   })
   m4_undefine({t})
```

其中需要确定三个循环展开步数： step_i, step_j, step_l. 当 step_i = step_j = step_l = 1 时，循环展开的结果与原始 BLAS 的程序完全一样.

```
Initial searching step = 2
        1  1  1:    mflops=1.333333
        2  2  2:    mflops=4.538853
 2  2  2:   4.538853
        0  2  2:    out of searching range.
        4  2  2:    mflops=8.294625
 4  2  2:   8.294625
        2  2  2:    mflops=4.538853 (buffered)
        6  2  2:    mflops=11.286681
 6  2  2:   11.286681
 ... ... ... ... ... ... ... ... ... ... ...
 8 16  4:   26.343519
        8 16  3:    mflops=26.041666
        8 16  5:    mflops=25.588537
        7 16  4:    mflops=25.746653
        9 16  4:    mflops=25.000000
        8 15  4:    mflops=24.900396
        8 17  4:    mflops=26.511135
 8 17  4:   26.511135
        8 16  4:    mflops=26.343519 (buffered)
        8 18  4:    mflops=25.000000 (buffered)
        8 17  3:    mflops=25.667351
        8 17  5:    mflops=24.850895
        7 17  4:    mflops=24.727993
        9 17  4:    mflops=25.252525
Performance for all unrolling steps equal to 1: 1.333333
Best choice found:  8 17  4   Mflops=26.511135   Speedup=19.883353
```

图 1　RS6000/25T 上“瞎子爬山”算法搜索的结果， N = M = K = 500, LDA = 501

图 1 是在 IBM RS6000/25T (60 MHz PowerPC) 上进行自动搜索的结果. 图中三个数一组的表示循环展开步数, 如 "8 4 2" 表示 step_i = 8, step_j = 4, step_l = 2, 后面跟随的浮点数代表对应于该组循环展开步数的浮点性能, 以 Mflops (每秒百万次浮点运算) 为单位. 向右错位输出的代表搜索的过程, 否则则代表当前搜索的出发点, 亦即当前找到的最佳循环开展步数. "Initial searching step" 表示搜索开始时使用的搜索步长. 对于这样一个三重循环, 完成一次搜索通常需要进行 25–35 次编译及试算. 我们还在 Hitachi SR2201 (150 MHz HARP) 、 SUN SPARC 20 和运行 Linux 的微机 (100 MHz Pentium, f2c+GCC2.7.0) 上采用 "瞎子爬山" 法进行了搜索实验. 四种机型上通过搜索最终得到的性能分别为 26.5, 85.3, 19.6 和 20.4 (Mflops), 相对于原始 BLAS 程序的加速比分别为 19.9, 6.2, 3.8 和 6.5. 在 RS6000, SPARC 20 和 Pentium 上搜索的结果基本上接近最优, 而在 RS6000 上获得的加速比最大. 根据我们的实验, 在 SR2201 上对于该循环通过循环展开所能得到的最好性能约为 110 Mflops, 如果进一步结合局部转置技术, 当矩阵阶数足够大时可获得 250 Mflops 的实际浮点性能 (SR2201 处理节点的峰值性能为 300 Mflops).

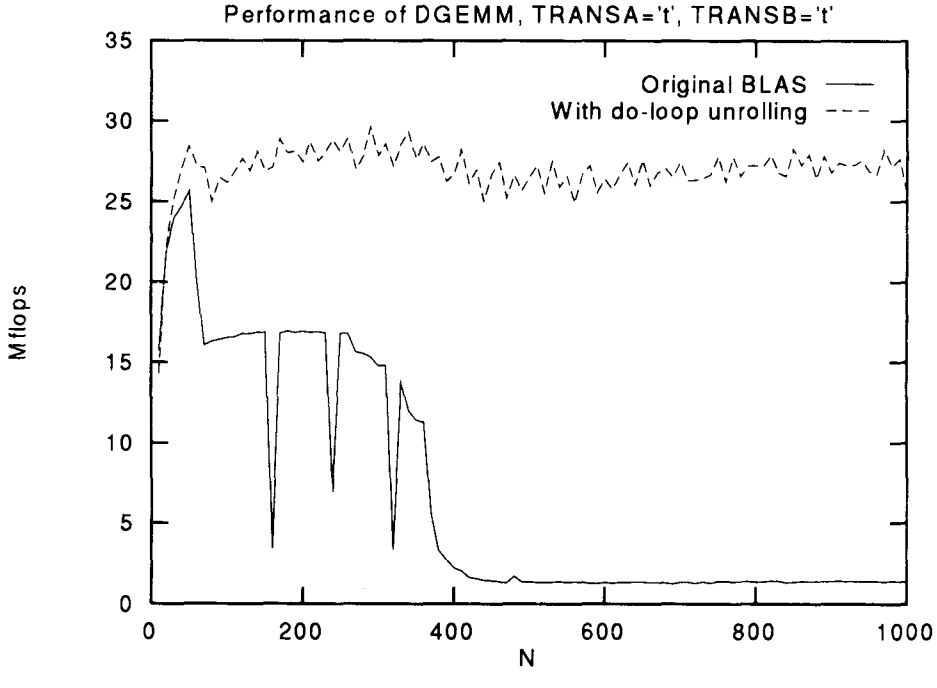

图 2 IBM RS6000/25T 上循环展开前后的性能比较, M = K = LDA =N

为了说明循环展开对于性能改善的效果, 我们在图 2-5 中给出了原始程序与进行循环展开后的程序对于不同阶数的矩阵的性能比较. 所使用的循环展开步数由表 1 给出, 其中对 SR2201 使用的是通过实验得出的一组较好的循环展开步数, 而对于

其它机型则直接使用“瞎子爬山”法搜索的结果，相应的加速比列在表 2 中.

表 1　“最佳”循环开展步数

机型	RS6000	SR2201	SPARC 20	Pentium 100
step_i, step_j, step_l	8, 17, 4	4, 4, 1	4, 5, 1	7, 10, 5

表 2　进行循环展开后所获得的性能加速比

N (= M = K = LDA)	100	200	300	400	500	600	700	800	900	1000
RS6000	1.58	1.62	1.88	12.02	19.58	20.03	20.65	20.29	19.25	18.97
SR2201	4.27	7.38	4.45	10.48	9.98	9.47	9.49	9.86	9.70	9.71
SPARC 20	0.98	1.88	2.19	3.70	3.71	4.14	4.07	4.15	4.19	4.21
Pentium	3.61	5.19	5.60	6.19	6.42	6.59	6.60	7.23	7.00	6.71

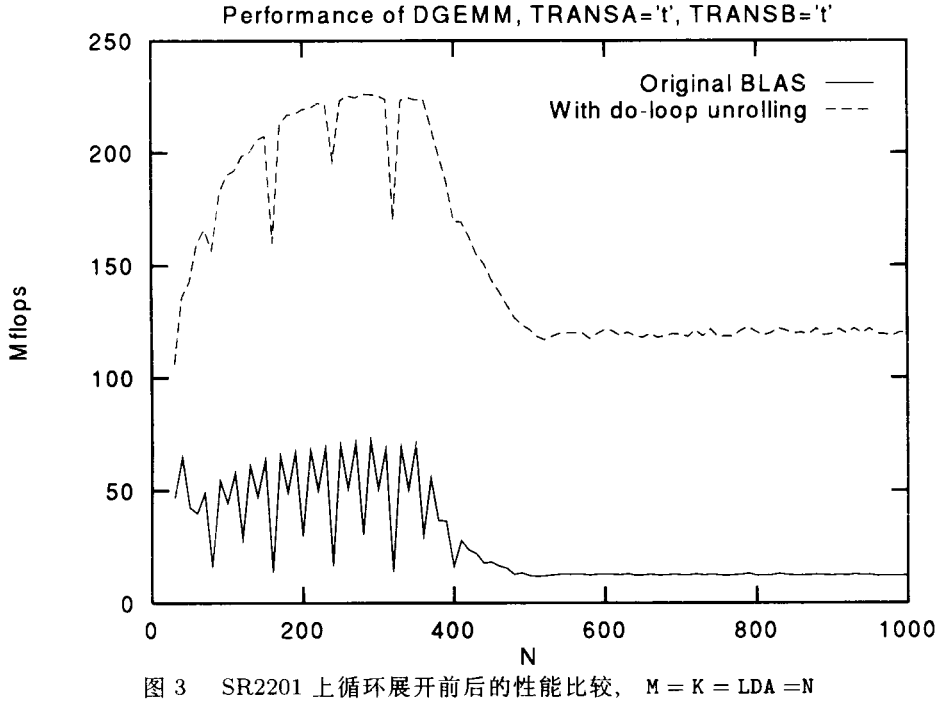

图 3　SR2201 上循环展开前后的性能比较，M = K = LDA =N

§5. 结　　论

从上节中介绍的数值结果可以看出，循环展开确实是 Fortran 77 中程序优化的一个简单有效的手段，它在向量机以及当前占主流的基于 RISC 微处理芯片的工作站及并行机上均可得到相当令人满意的性能加速比. 利用 m4 宏语言进行 Fortran 77 循环展开不仅可以大简化程序的编写，同时还便于根据不同的计算机类型改变循环展开

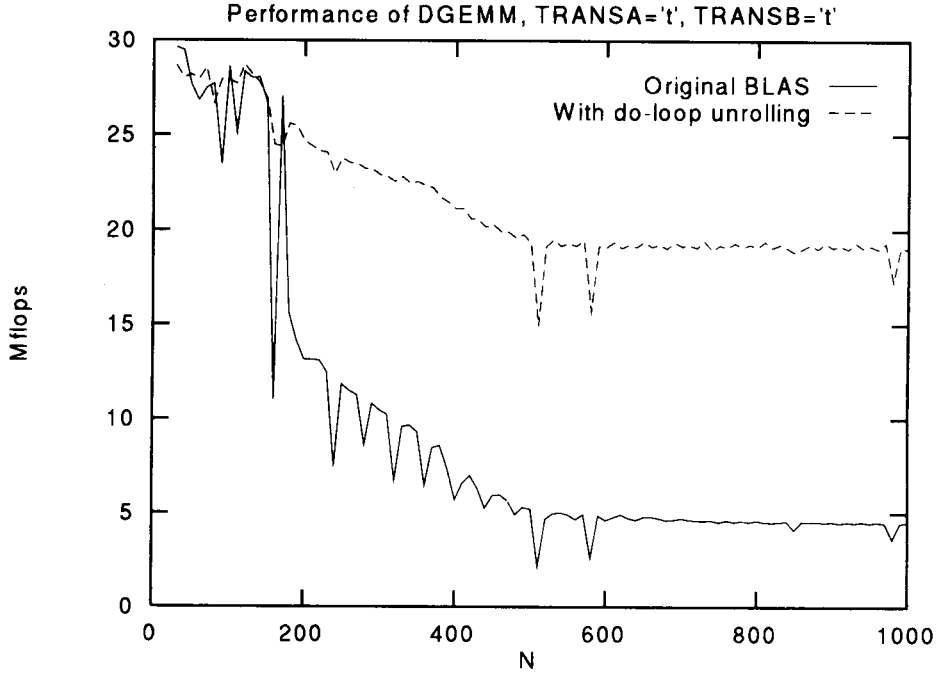

图 4 SPARC 20 上循环展开前后的性能比较， M = K = LDA =N

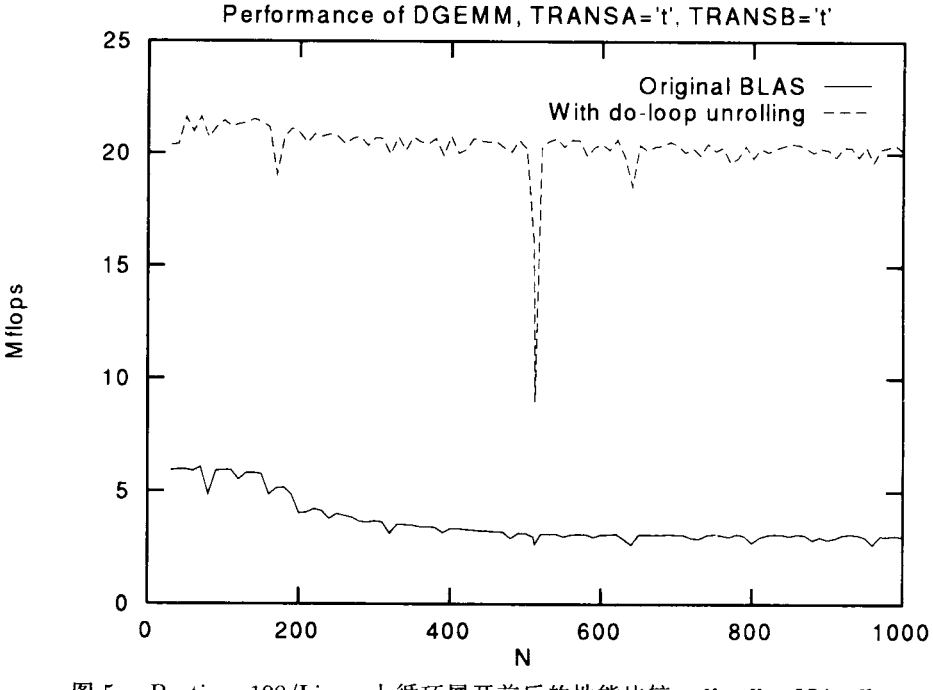

图 5 Pentium-100/Linux 上循环展开前后的性能比较， M = K = LDA =N

的步数，以达到最好的浮点计算性能．本文中只是针对几种典型的简单循环描述了如何借助于适当定义的 m4 宏命令进行循环展开，在进一步对 Fortran 77 程序中的各种常用循环类型进行系统研究的基础上，有可能定义出一整套用于 Fortran 77 循环展开的 m4 宏命令，形成一种专用的语言，用于编写具有高性能要求的、大量重复使用的标准程序库及应用程序．

为供对本文所介绍的内容有兴趣的读者参考，我们在以下匿名 FTP 地址提供了一些 Fortran 循环展开的实例，包括 m4 源程序及性能测试程序：

`ftp://ftp.cc.ac.cn/pub/home/zlb/mfor/`

利用本文所介绍的方法，我们目前正在中科院软件所并行软件研究开发中心研制一套可任意调节循环展开步数的 BLAS 程序库．从目前已经完成的部分来看取得了很好的效果．

参 考 文 献

[1] `ftp://ftp.cc.ac.cn/pub/home/zlb/mfor/`

[2] René Seindal, GNU m4, version 1.4, 参看 `ftp://ftp.pku.edu.cn/pub/gnu/m4-1.4.tar.gz` 中的 `m4-1.4/doc/m4.texinfo` 文件．

[3] C. Lawson, R. Hanson, D. Kincaid, F. Krogh, Basic Linear Algebra Subprograms for Fortran Usage, *ACM Trans. Math. Software,* **5** (1979), 308–323.

[4] J.J. Dongarra, J. DuCroz, S. Hammarling, R. Hanson, An Extended Set of Fortran Basic Linear Algebra Subprograms, *ACM Trans. Math. Software,* **14** (1988), 1–17.

[5] J.J. Dongarra, J. DuCroz, I.S. Duff, S. Hammarling, A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs, Argonne National Laboratory Report, ANL-MCS-P88-2, 1988.

[6] A.A. Pollicini (editor), Using Toolpack Software Tools, Proceedings of the ISPRA-Course held at the Joint Research Centre, Ispra, Italy, 17–21 November, 1986. Kluwer Academic Publishers, Dordrecht/Boston/London.