RAL-TR-96-014

A Blocked Implementation of Level 3 BLAS for RISC Processors[1]

Michel J. Daydé[2] and Iain S. Duff[3]

ABSTRACT

We describe a version of the Level 3 BLAS which is designed to be efficient on RISC processors. This is an extension of previous studies by the same authors (see Amestoy, Daydé, Duff & Morère (1995), Daydé, Duff & Petitet (1994), and Daydé & Duff (1995)) where they describe a similar approach for efficient serial and parallel implementations of Level 3 BLAS on shared and virtual shared memory multiprocessors.

All our codes are written in Fortran and use loop-unrolling, blocking, and copying to improve the performance. A blocking technique is used to express the BLAS in terms of operations involving triangular blocks and calls to the matrix-matrix multiplication kernel (GEMM). No manufacturer-supplied or assembler code is used. This blocked implementation uses the same blocking ideas as in Daydé et al. (1994) except that the ordering of loops is designed for efficient reuse of data held in cache and not necessarily for parallelization. A parameter which controls the blocking allows efficient exploitation of the memory hierarchy on the various target computers.

We present results on a range of RISC-based workstations and multiprocessors, viz. DEC 3000/400 AXP, DEC 8400 5/300, HP 715/64, IBM RS/6000-750, MEIKO CS2-HA, SGI Power Challenge L, and SUN SPARC 20/50.

Keywords: Level 3 BLAS, matrix-matrix kernels, RISC processors, loop-unrolling, blocking.
AMS(MOS) subject classifications: 65F05, 65F50.

---

[1]Part of this study was funded by Conseil Régional Midi-Pyrénées under project DAE1/RECH/9308020.

[2]Email: dayde@enseeiht.fr. ENSEEIHT-IRIT, 2 rue Camichel, 31071 Toulouse CEDEX, France.

[3]Email: isd@rl.ac.uk. Also at CERFACS, 42 av. G. Coriolis, 31057 Toulouse Cedex, France.

Current reports available by anonymous ftp from matisa.cc.rl.ac.uk in the directory "pub/reports". This report is in file daduRAL96014.ps.gz.

Computing and Information Systems Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

February 26, 1996.

Contents

1 Introduction

This report describes a version of single and double precision Level 3 BLAS computational kernels (Dongarra, Du Croz, Duff & Hammarling (1990*b*),Dongarra, Du Croz, Duff & Hammarling (1990*a*)) called the *blocked BLAS*, designed to be efficient on RISC processors. It is based on the use of the matrix-matrix multiplication kernel GEMM. We show that this implementation is portable and efficient on a range of RISC-based computers.

This version of the Level 3 BLAS is an evolution of the one described by Daydé et al. (1994) by the same authors for MIMD vector multiprocessors. They report on experiments on a range of computers (ALLIANT, CONVEX, IBM and CRAY) and demonstrate the efficiency of their approach whenever a tuned version of the matrix-matrix multiplication is available. They conclude by saying that similar ideas could be used to design a tuned uniprocessor Level 3 BLAS for computers where the processor accesses data through a cache since blocking would also be beneficial.

The availability of powerful RISC processors is of major importance in today's market since they are used both in workstations and in the most recent parallel computers. Because of the success of RISC-based architectures, we have decided to study the design of a version of the Level 3 BLAS that is efficient on RISC processors. This tuned version of the Level 3 BLAS uses the same blocking ideas as in Daydé et al. (1994), except that the ordering of loops is designed for efficient reuse of data held in cache.

Our basic idea for designing the Level 3 BLAS is to partition the computations across submatrices so that the calculations can be expressed in terms of calls to GEMM and operations involving triangular matrices. All the codes we are using are written in Fortran and tuned using blocking, copying and loop-unrolling. We believe that these codes provide an efficient implementation of the Level 3 BLAS on computers where a highly tuned version is not available. In this paper, the timings for the non-GEMM blocked kernels are for versions using our own blocked GEMM code. We note that, in cases when the vendor supplies a more efficient version of GEMM, it is trivial for us to use this in these other kernels. By doing so, we can often do far better than the vendor-supplied versions of these other kernels. At this time, we are very concerned with portability and so have made no attempt to include specific tuning techniques that are crucial on some computers. Additionally, our experiments often use non-ideal matrix orders (for example, orders of powers of two). While this is good for portability and robustness, on some machines the times would be better for other matrix orders. Additionally, most machines have more than one-level of cache. In our present work, we only allow one level of blocking and choose a block size based on the largest on-chip cache. Further optimization for particular machines might be obtained by multi-level blocking. We would be happy to discuss with users and vendors the possibility of designing more highly-tuned, but less portable, kernels for specific machines. We also hope to receive input and comments from users to improve this software.

The implementation of the kernels using both blocking and loop unrolling is described in Sections 3 to 8 (examples of codes are included), more details on this implementation are reported by Qrichi Aniba (1994). We only consider the implementation of the real and the

double precision Level 3 BLAS kernels. We report results from uniprocessor executions on a range of RISC-based computers (in practice we have proceeded to experiments on a larger set of machines) :

1. DEC 3000/400 AXP
2. DEC 8400 5/300
3. HP 715/64
4. IBM RS/6000-750
5. MEIKO CS2-HA (using a HyperSparc processor)
6. SGI Power Challenge L
7. SUN SPARC 20/50

2 Blocked implementation of Level 3 BLAS for RISC processors

2.1 RISC processors

Vector processors are commonly used in supercomputers. Recently very fast RISC processors, which can also process vectors efficiently, have come on to the market. They are usually more efficient than vector processors on scalar applications. The main reason for their success in the marketplace is their very good cost to performance ratio. They are used as a CPU both in workstations and in most of the current MPPs (DEC Alpha on CRAY T3D, SPARC on CM5 and PCI CS2, HP PA on CONVEX EXEMPLAR, and RS/6000 on IBM SP1 and SP2). Table 2.1.1 gives the uniprocessor performance of some current RISC processors on the double precision 100-by-100 and 1000-by-1000 LINPACK benchmarks (Dongarra 1992), together with the clock speed and the nominal peak performance. We include all of our target machines in this table and note that, for some of them, there are no LINPACK figures.

Computer	LINPACK 100*100	LINPACK 1000*1000	Clock (MHz)	Peak performance
DEC 8400 5/300	140	411	300	600
DEC 3000/400 AXP	26	90	133	133
IBM POWER2-990	140	254	71.5	286
IBM RS/6000-750	–	–	62.5	125
HP 9000/755	41	120	99	198
HP 715/64	–	–	64	128
MEIKO CS2-HA	–	–	100	100
SGI POWER Challenge	104	261	75	300
SUN SPARC 20/50	–	–	50	50

Table 2.1.1: Performance in Mflop/s of RISC computers on the double precision LINPACK benchmarks

2.2 Efficient exploitation of the memory hierarchy

The ability of the memory to supply data to the processors at a sufficient rate is crucial on most modern computers. This necessitates complex memory organizations, where the memory is usually arranged in a hierarchical manner. The minimization of data transfers between the levels of the memory hierarchy is a key issue for performance (Gallivan, Jalby & Meier (1987), Gallivan, Jalby, Meier & A. (1988)).

Most of the RISC-based architectures have a memory hierarchy involving a cache. The cache memory is used to mask the memory latency (typically the cache latency is around 1-2 clocks while it is often 10 times higher for the memory). The code performance is high so long as the cache hit ratio is close to 100%. This may happen if the data involved in the calculations can fit in cache or if the calculations can be organized so that data can be kept in cache and efficiently reused. One of the most commonly used techniques for that purpose is called blocking and examples of this are reported in the following sections. Blocking enhances spatial and temporal locality in computations. Unfortunately blocking is not always sufficient since the cache miss ratio can be dramatically increased in quite an unpredictable way by memory accesses using a stride greater than 1 (see Bodin & Seznec (1994)).

Some strides are often called *critical* because they generate a very high cache miss ratio (i.e. when referencing cache lines that are mapped into the same physical location of the cache). These critical strides obviously depend on the cache management strategy. For example, in the execution of the following loop :

```
do i=1,n,4
     temp = temp + a(i)
enddo
```

each read of a(i) causes a cache miss, assuming that a(i) is one word and that the cache line length is equal to four words (assuming that the cache is initially empty).

Copying blocks of data (submatrices for example) that are heavily reused may help to improve memory and cache accesses (by avoiding critical strides for example). Since it may induce a large overhead, it is, however, not always a viable technique. We illustrate this in our blocked implementation of the BLAS.

Note that blocking and copying are also very useful in limiting the effect of TLB (Translation Lookaside Buffer) misses or memory paging.

Our basic idea for efficient implementation of the BLAS on RISC processors is to express all the Level 3 BLAS kernels in terms of subkernels that either deal with NB $\times$ NB submatrices that involve GEMM operations or operations involving triangular submatrices. Additionally, all the calculations on blocks are performed using tuned Fortran codes with loop-unrolling. Copying is occasionally used. Of course, the relative efficiency of this approach depends on the availability of a highly tuned GEMM kernel. This approach

is relatively independent of the computer : only the NB parameter, corresponding to the block size, and the loop-unrolling depth in some cases should be tuned according to the characteristics of the target machine. NB is determined by the size of the cache (see Section 3.1) and the loop-unrolling depth from the number of scalar registers. Note that Kågström, Ling & Loan (1993)) use similar ideas. Their GEMM-based BLAS only requires the availability of a highly tuned matrix-matrix multiplication and Level 1 and Level 2-based operations.

In the following sections, we describe the blocked implementation of the real and double precision Level 3 BLAS : GEMM, SYMM, TRSM, TRMM, SYRK, SYR2K (all these names are prefixed by S or D depending on whether the routine is single or double precision).

For each kernel there are a number of options, for example whether the matrix is transposed or not. For the sake of clarity, we comment only on one of these variants of the kernels and we illustrate our blocking strategy on matrices that are only partitioned into four blocks. In practice, the matrices are partitioned into NB × NB blocks where NB is chosen according to the machine characteristics.

3 Blocked implementation of GEMM

3.1 Description of the blocked GEMM

GEMM performs one of the matrix-matrix operations :

$$\mathbf{C}{=}\alpha\ \mathrm{op}(\mathbf{A})\ \mathrm{op}(\mathbf{B}){+}\beta\mathbf{C}$$

where α and β are scalars, $\mathbf{A}$ and $\mathbf{B}$ are rectangular matrices of dimensions m×k and k×n, respectively, $\mathbf{C}$ is a m × n matrix, and op($\mathbf{A}$) is $\mathbf{A}$ or $\mathbf{A}^t$.

We consider the following case (corresponding to op equal to "No transpose" in both cases):

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,2} & C_{2,2} \end{pmatrix} = \alpha \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} + \beta \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

DGEMM can obviously be organized in terms of a succession of matrix-matrix multiplication on submatrices as follows :

1. $C_{1,1} \leftarrow \beta C_{1,1} + \alpha A_{1,1} B_{1,1}$ (DGEMM)

2. $C_{1,1} \leftarrow C_{1,1} + \alpha A_{1,2} B_{2,1}$ (DGEMM)

3. $C_{1,2} \leftarrow \beta C_{1,2} + \alpha A_{1,1} B_{1,2}$ (DGEMM)

4. $C_{1,2} \leftarrow C_{1,2} + \alpha A_{1,2} B_{2,2}$ (DGEMM)

5. $C_{2,1} \leftarrow \beta C_{2,1} + \alpha A_{2,1} B_{1,1}$ (DGEMM)

6. $C_{2,1} \leftarrow C_{2,1} + \alpha A_{2,2} B_{2,1}$ (DGEMM)

7. $C_{2,2} \leftarrow \beta C_{2,2} + \alpha A_{2,1} B_{1,2}$ (DGEMM)

8. $C_{2,2} \leftarrow C_{2,2} + \alpha A_{2,2} B_{2,2}$ (DGEMM)

The ordering of these eight computational steps is determined by considerations on efficient reutilization of data held in cache. We have decided to reuse the submatrices of **A** as much as possible and we perform all operations involving a submatrix before moving to another one (see Figure 3.1.1). For our simple example, it means that we perform the calculations as follows : Step 1, Step 3, Step 5, Step 7, Step 2, Step 4, Step 6, and Step 8. This approach is similar to that used by Dongarra, Mayes & Radicati di Brozolo (1991). In practice, NB is chosen so that all the submatrices of **A**, **B**, and **C** required for each submultiplication fit in the largest on-chip cache, except for the MEIKO CS2-HA because the HyperSparc only possesses an external cache on that computer. On some machines, access to off-chip caches has so low latency that we can improve performance by using a larger block size. This is true, for example, on the SGI Power Challenge. The most efficient use of multi-level cache machines is outwith the scope of this paper since we wish to keep our codes portable. Since all the computational kernels call GEMM, the block size NB is always determined as the most appropriate block size for GEMM, i.e. the largest even integer (to enhance loop-unrolling) such that :

$$3(NB)^2 prec < CS$$

where $prec$ is the number of bytes corresponding the precision used (4 bytes for single precision and 8 bytes for double precision in IEEE format) and CS is the cache size in bytes. For example with a 64Kbytes cache, NB is set to 52 using 64-bit arithmetic.

We report in Table 3.1.1 the block sizes used in our experiments. We also include the cache organization (direct mapped or set associative). Note that the DEC processor (DEC 21164) used on the DEC 8400 5/300 possesses 2 levels of internal cache of size equal to 8KB and 96KB respectively, and an external cache of from 1 MB up to 64 MB. We have tuned our codes with respect to the second level of cache since our experiments show this is the most efficient.

The blocked code is reported in Figure 3.1.1. Its main features are the following :

- The multiplication of **C** by β is performed before all other calculations.

Computer	Cache characteristics		Block size		Organization
	Size	Organization	Single	Double	of operations
DEC 3000/400-AXP	8 KB	Direct	24	16	TRIADIC
DEC 8400 5/300	96 KB	3-way	88	60	TRIADIC
HP 715/64	64 KB	Direct	72	52	TRIADIC
IBM RS/6000-750	32 KB	4-way	42	36	TRIADIC
MEIKO CS2-HA	256 KB	4-way	140	100	NOTRIADIC
SGI Power Challenge	16 KB	Direct	36	24	TRIADIC
SUN SPARC 20/50	16 KB	4-way	36	24	NOTRIADIC

Table 3.1.1: Block size used in the blocked BLAS on the target computers.

- The submatrix of **A** is multiplied by α and transposed into AA to avoid non-unit strides because of access by rows in the innermost loops of the calculations. These are organized in such a way that AA is kept in cache as long as required.

We use two tuned Fortran codes to perform calculations on submatrices (see Figure 3.1.2) :

- DGEMML2X2 is a tuned code for performing matrix-matrix multiplication on square matrices of even order.
- DGEMML is a tuned code that includes additional tests over DGEMML2X2 to handle matrices with odd order. It is occasionally slightly less efficient than DGEMML2X2.

We have used two versions for all the tuned codes :

- the TRIADIC option for computers where triadic operations are either supported in the hardware (for example the floating-point multiply-and-add on IBM RS/6000) or efficiently compiled
- The NOTRIADIC option for other computers

The use of triadic operations should not normally degrade the performance severely on processors that do not support these operations since efficient code generation can transform them into dyadic operations. However, in early versions of SPARC compilers, we saw that there was sometimes such a degradation. Thus we prefer to offer both options.

The tuned code DGEMML2X2 using the TRIADIC options is reported in Figure 3.1.2. The code corresponding to the NOTRIADIC option follows in Figure 3.1.3. The selection between the options is effected using the C preprocessor. All the tuned codes described in the rest of this paper offer both options.

Slight modifications (Dongarra et al. 1991) would allow further improvement in performance on the IBM RS/6000.

```
*
*     Form C := beta*C
*
      IF( BETA.EQ.ZERO )THEN
          DO 20  J = 1, N
             DO 10  I = 1, M
                C( I, J ) = ZERO
   10        CONTINUE
   20     CONTINUE
      ELSE
           DO 40  J = 1, N
              DO 30  I = 1, M
                 C( I, J ) = BETA*C( I, J )
   30         CONTINUE
   40      CONTINUE
      END IF
*
*     Form  C := alpha*A*B + beta*C.
*
      DO 70  L = 1, K, NB
         LB = MIN( K - L + 1, NB )
         DO 60  I = 1, M, NB
            IB = MIN( M - I + 1, NB )
            DO II = I, I + IB - 1
               DO LL = L, L + LB - 1
                  AA(LL-L+1,II-I+1)=ALPHA*A(II,LL)
               ENDDO
            ENDDO
            DO 50  J = 1, N, NB
               JB = MIN( N - J + 1, NB )
*
*  Perform multiplication on submatrices
*
               IF ((MOD(IB,2).EQ.0).AND.(MOD(JB,2).EQ.0)) THEN
                  CALL DGEMML2X2(IB,JB,LB,AA,NB,B(L,J),LDB,C(I,J),LDC)
               ELSE
                  CALL DGEMML(IB,JB,LB,AA,NB,B(L,J),LDB,C(I,J),LDC)
               END IF
 50         CONTINUE
 60      CONTINUE
 70   CONTINUE
```

Figure 3.1.1: Blocked code for GEMM

```
*
*           C := alpha*A*B + C.
*
            DO 70  J = 1, N, 2
               DO 60  I = 1, M, 2
                  T11 = C(I,J)
                  T21 = C(I+1,J)
                  T12 = C(I,J+1)
                  T22 = C(I+1,J+1)
                  DO 50  L = 1, K
                     B1 = B(L,J)
                     B2 = B(L,J+1)
                     A1 = A(L,I)
                     A2 = A(L,I+1)
                     T11 = T11 + B1*A1
                     T21 = T21 + B1*A2
                     T12 = T12 + B2*A1
                     T22 = T22 + B2*A2
   50             CONTINUE
                  C(I,J) = T11
                  C(I+1,J) = T21
                  C(I,J+1) = T12
                  C(I+1,J+1) = T22
   60          CONTINUE
   70       CONTINUE
```

Figure 3.1.2: Tuned code for GEMM (TRIADIC option)

```
*
*           C := alpha*A*B + C.
*
            DO 70  J = 1, N, 2
               DO 60  I = 1, M, 2
                  T11 = C(I,J)
                  T21 = C(I+1,J)
                  T12 = C(I,J+1)
                  T22 = C(I+1,J+1)
                  DO 50  L = 1, K
                     B1 = B(L,J)
                     B2 = B(L,J+1)
                     A1 = A(L,I  )
                     A2 = A(L,I+1)
                     T1 = B1*A1
                     T2 = B1*A2
                     U1 = B2*A1
                     U2 = B2*A2
                     T11 = T11 + T1
                     T21 = T21 + T2
                     T12 = T12 + U1
                     T22 = T22 + U2
   50             CONTINUE
                  C(I,J) = T11
                  C(I+1,J) = T21
                  C(I,J+1) = T12
                  C(I+1,J+1) = T22
   60          CONTINUE
   70       CONTINUE
```

Figure 3.1.3: Tuned code for GEMM (NOTRIADIC option)

3.2 Single precision implementation on the IBM RS/6000

The IBM RS/6000 FPU performs its arithmetic using 64-bit operands. As a consequence, single precision operations are performed in the following way :

1. Convert operands from single to double precision.
2. Perform double precision computation.
3. Convert double precision result into single precision.

These conversions can be very costly and explain why the IBM RS/6000 is slower in single precision than in double precision. Therefore, we have slightly modified the tuned code SGEMML2X2 to convert operands within the innermost loop only once. The matrix **A** is copied into a double precision working array in the blocked code. The code is shown in Figure 3.2.1, where the array A refers to this double precision copy.

```
*
*           C := A*B + C.
*
      DO 70  J = 1, N, 2
         DO 60  I = 1, M, 2
            T11 = DBLE(C(I  ,J))
            T21 = DBLE(C(I+1,J))
            T12 = DBLE(C(I  ,J+1))
            T22 = DBLE(C(I+1,J+1))
            DO 50  L = 1, K
               B1 = DBLE(B(L,J  ))
               B2 = DBLE(B(L,J+1))
               A1 = A(L,I  )
               A2 = A(L,I+1)
               T11 = T11 + B1*A1
               T21 = T21 + B1*A2
               T12 = T12 + B2*A1
               T22 = T22 + B2*A2
   50       CONTINUE
            C(I  ,J) = REAL(T11)
            C(I+1,J) = REAL(T21)
            C(I  ,J+1) = REAL(T12)
            C(I+1,J+1) = REAL(T22)
   60    CONTINUE
   70 CONTINUE
```

Figure 3.2.1: Tuned code for SGEMM on IBM

As we can see in Table 3.2.1, the performance of the blocked implementation of SGEMM without conversion is much worse than the double precision one (as soon as matrices are bigger than the block size). This performance decrease is due to the large number of unnecessary single precision to double precision conversions. Our modification allows a

significant reduction in the number of operand conversions. Thus, the single precision performance is much improved and slightly better than the double precision one as on the other computers. We have not used this data conversion in the other single precision kernels on the IBM, but they should be designed in the same way. Since IBM provides a tuned BLAS implementation in its scientific library, we have decided not to expend too much effort on tuning our code for the IBM, and certainly not to use machine dependent tricks for optimizing on that machine.

	SGEMM			DGEMM
m=n=k	Standard	Blocked		Blocked
		no conv.	conv	
32	17	68	68	34
64	22	53	89	67
96	23	54	94	81
128	23	54	92	85

Table 3.2.1: Performance in Mflop/s of the blocked implementation of SGEMM on the IBM RS/6000-750 with and without explicit conversion for single to double precision.

3.3 Numerical experiments

We show in Tables 3.3.1 and 3.3.2 the performance achieved on DEC 3000/400 AXP, DEC 8400 5/300, IBM RS/6000-750, HP 715, MEIKO CS2-HA, SGI POWER Challenge, and SUN SPARC 20/50 workstations. We also include the performance of the manufacturer-supplied library version when available (we use -lblas on the IBM, the SGI, and the HP, and -ldxml on the DEC 8400). Although a tuned BLAS is also available on the IBM using the ESSL Library, we do not use it because it is sometimes slower than using -lblas and is not available without extra payment. "Standard" in column 2 of the tables refers to the standard Fortran version. The performance reported is the average performance achieved on a set of 4 matrix-matrix multiplications where all matrices are square of order 32, 64, 96, and 128.

The blocked implementation of GEMM usually provides a gain of more than 2 over the standard Fortran code when the matrices exceed the cache size. Note that better performance can be achieved if the matrices are already located (preloaded) in the cache, which is not the case in our experiments. On the MEIKO CS2-HA, the KAP preprocessor that we use performs extremely efficient optimizations (using loop-unrolling) and, since the matrices are relatively small and fit in cache (the size of the external cache is 256KB), the standard version of DGEMM when arrays are not transposed is the same as our tuned version (optimization performed by KAP and by hand are equivalent since blocking has no effect). On the DEC 8400, the vendor-supplied library routines perform significantly better than our blocked code, in both single and double precision, probably because better use is made of the multi-level cache. For possibly the same reason, the vendor-supplied library GEMM routines on the SGI usually perform better than our blocked code, particularly in double precision. However, if we increase the block size, we can improve the performance

Processor	DGEMM	op(**A**), op(**B**)			
		'N','N'	'N','T'	'T','N'	'T','T'
DEC3000/400-AXP	standard	26	26	21	15
	blocked	47	51	47	45
DEC 8400 5/300	standard	95	98	76	61
	blocked	216	208	215	211
	library	335	327	345	318
IBM RS/6000-750	standard	29	29	38	26
	blocked	79	65	82	82
	library	89	81	87	85
HP 715/64	standard	15	16	20	22
	blocked	29	30	30	34
	library	52	47	46	51
MEIKO CS2-HA	standard	39	36	30	27
	blocked	38	45	39	43
SGI Power Challenge	standard	74	73	107	74
	blocked	131	121	132	131
	library	212	204	204	196
SUN SPARC 20/50	standard	11	11	14	8
	blocked	26	24	26	27

Table 3.3.1: Average performance in Mflop/s of the blocked implementation of DGEMM on RISC workstations (using square matrices of order 32, 64, 96, and 128).

Processor	SGEMM	op(**A**), op(**B**)			
		'N','N'	'N','T'	'T','N'	'T','T'
DEC3000/400-AXP	standard	32	33	24	18
	blocked	67	70	68	69
DEC 8400 5/300	standard	105	108	77	65
	blocked	246	267	239	258
	library	412	388	416	395
IBM RS/6000-750	standard	27	27	22	20
	blocked	87	85	83	91
	library	96	96	104	95
HP 715/64	standard	18	18	22	24
	blocked	55	59	55	56
	library	81	63	71	81
MEIKO CS2-HA	standard	28	33	33	36
	blocked	60	58	69	78
SGI Power Challenge	standard	74	73	82	70
	blocked	203	164	207	204
	library	218	210	209	194
SUN SPARC 20/50	standard	23	22	23	18
	blocked	42	39	42	41

Table 3.3.2: Average performance in Mflop/s of the blocked implementation of SGEMM on RISC processors (using square matrices of order 32, 64, 96, and 128).

of the blocked codes by up to 15% even though the submatrices do not then fit in the on-chip caches.

We also report in Table 3.3.3 the average performance of DGEMM when the inner dimension of the matrix-matrix product is small (k equals 8 and 16) since it is of special interest for sparse matrix calculations (Amestoy et al. (1995), Amestoy & Duff (1989), and Puglisi (1993)). We only consider the case where **A** and **B** are not transposed.

Processor	DGEMM	k	
		8	16
DEC3000/400-AXP	standard	32	29
	blocked	32	38
IBM RS/6000-750	standard	38	38
	blocked	57	64
	library	62	72
HP 715/64	standard	17	16
	blocked	23	25
	library	38	42
MEIKO CS2-HA	standard	33	37
	blocked	32	35
SUN SPARC 20/50	standard	16	15
	blocked	19	21

Table 3.3.3: Average performance in Mflop/s of the blocked implementation of DGEMM on RISC workstations (where **C** is a square matrix of order 32, 64, 96, and 128 and inner dimension of the product, k, equal to 8 and 16).

We show in Table 3.3.4 the performance of the best available version of DGEMM and SGEMM to us, that is we use either our implementation or a tuned manufacturer-supplied version, when both **A** and **B** are not transposed on square matrices of order 500 and 1000 in order to study whether we can get close to the theoretical peak performance.

The performance achieved by the tuned versions of GEMM is relatively far from the peak performance for all the RISC processors except the IBM RS/6000-750 and the SGI Power Challenge. On the IBM RS/6000-750 and the IBM Power2, it is possible to reach peak performance by changing the leading dimensions of the matrices (Agarwal, Gustavson & Zubair 1994).

4 Blocked implementation of TRSM

TRSM solves one of the matrix equations :

$$\mathbf{AX}=\alpha\mathbf{B},\ \mathbf{A^tX}=\alpha\mathbf{B},\ \mathbf{XA}=\alpha\mathbf{B},\ \text{or}\ \mathbf{XA^t}=\alpha\mathbf{B}$$

Processor	Version	Kernel	Size 500	Size 1000	Peak
DEC3000/400-AXP	blocked	DGEMM	47	45	133
		SGEMM	71	70	
DEC 8200 5/300	library	DGEMM	334	313	600
		SGEMM	431	418	
IBM RS/6000-750	library	DGEMM	98	94	125
		SGEMM	111	110	
HP 715/64	library	DGEMM	35	35	128
		SGEMM	71	68	
MEIKO CS2-HA	blocked	DGEMM	49	49	100
		SGEMM	88	88	
SGI Power Challenge	library	DGEMM	240	233	300
		SGEMM	267	265	
SUN SPARC 20/50	blocked	DGEMM	28	28	50
		SGEMM	43	43	

Table 3.3.4: Performance in Mflop/s of the best available implementation of DGEMM and SGEMM on RISC workstations (**A** and **B** are not transposed).

where α is a scalar, **X** and **B** are m $\times$ n matrices and **A** is a unit, or non-unit, upper or lower triangular matrix. **B** is overwritten by **X**.

We consider the following case (corresponding to the parameters "Left", "No transpose", and "Upper", i.e. we solve for $\mathbf{AX} = \alpha\mathbf{B}$ where **A** is not transposed, and upper triangular):

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ 0 & A_{2,2} \end{pmatrix} \begin{pmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{pmatrix} = \alpha \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

1. Solution of $A_{2,2}X_{2,1}=\alpha B_{2,1}$ and $B_{2,1}$ is overwritten by $X_{2,1}$ (TRSM)
2. Solution of $A_{2,2}X_{2,2}=\alpha B_{2,2}$ and $B_{2,2}$ is overwritten by $X_{2,2}$ (TRSM)
3. $B_{1,1} \leftarrow \alpha B_{1,1}$-$A_{1,2}B_{2,1}$ (GEMM)
4. $B_{1,2} \leftarrow \alpha B_{1,2}$-$A_{1,2}B_{2,2}$ (GEMM)
5. Solution of $A_{1,1}X_{1,1}=B_{1,1}$ and $B_{1,1}$ is overwritten by $X_{1,1}$ (TRSM)
6. Solution of $A_{1,1}X_{1,2}=B_{1,2}$ and $B_{1,2}$ is overwritten by $X_{1,2}$ (TRSM)

Therefore, TRSM can be computed as a sequence of triangular solutions (TRSM) and matrix-matrix multiplications (GEMM). The ordering of computational steps is chosen so that each submatrix $A_{i,i}$ on the diagonal of **A**, involved in each solution step, is kept in

the cache for as long as it can be used. As for GEMM, we use two distinct versions of the tuned Fortran code for the solution step : TRSML2X2 when the order of **B** is even and TRSML otherwise. The blocked code is shown in Figure 4.1. The DGEMM subroutine is the one we described in Section 3. The tuned Fortran code (TRIADIC option) for TRSML2X2 is shown in Figure A.1 in the Appendix.

```
*
*           B := alpha*inv( A )*B.
*
            IF( UPPER )THEN
                IF( ALPHA.NE.ONE )THEN
                  DO 35, J = 1, N
                     DO 30, I = 1, M
                        B( I, J ) = ALPHA*B( I, J )
30                   CONTINUE
35                CONTINUE
                END IF

                DO 50 I= M,1,-NB
                   IB=MIN(NB,I)
                   DO 45 J=1,N,NB
                      JB= MIN(NB,N-J+1)

        IF ((MOD(IB,2).EQ.0).AND.(MOD(JB,2).EQ.0)) THEN
                   CALL DTRSML2X2_LUN( DIAG,
     $                        IB,JB,ONE, A(I-IB+1,I-IB+1),
     $                        LDA, B(I-IB+1,J), LDB )
        ELSE
                   CALL DTRSML_LUN( DIAG,
     $                        IB,JB,ONE, A(I-IB+1,I-IB+1),
     $                        LDA, B(I-IB+1,J), LDB )
        END IF

                   CALL DGEMM ( 'No transpose', 'No transpose',
     $                        I-IB,JB,IB,-ONE, A(1,I-IB+1),
     $                        LDA,B(I-IB+1,J),LDB,ONE,B(1,J),LDB)
45               CONTINUE
50              CONTINUE
```

Figure 4.1: Blocked code for TRSM

We report in Tables 4.1 and 4.2 the performance achieved on our range of RISC workstations for all variants when **A** is unit (the performance is similar when **A** is non-unit). We are not using explicit conversions from single to double precision for the tuned code on the IBM workstation.

The performance gain provided by the blocked implementation of DTRSM compared with the standard Fortran version is close to a factor of 3 and is more impressive than that obtained for DGEMM. The double precision results for our blocked version on the SGI can be much improved (by up to 30%) by using a larger block size although for some options our single precision blocked code already outperforms the library version. In both single and double precision, our blocked code outperforms the vendor code on the DEC 8400,

Processor	DTRSM	Variant							
		'Left'				'Right'			
		'U','N'	'L','N'	'U','T'	'L','T'	'U','N'	'L','N'	'U','T'	'L','T'
DEC3000/400-AXP	standard	13	13	21	21	15	15	15	15
	blocked	39	39	38	38	40	40	39	39
DEC 8400 5/300	standard	40	73	70	72	97	97	93	96
	blocked	204	187	199	231	184	175	186	191
	library	184	182	176	183	179	203	181	177
MEIKO CS2-HA	standard	13	12	31	30	12	12	11	12
	blocked	50	47	43	42	48	49	47	44
HP 715/64	standard	12	12	20	19	13	15	13	13
	blocked	30	28	31	31	31	31	30	28
	library	33	34	34	31	36	43	42	37
IBM RS/6000-750	standard	23	24	41	40	27	27	27	26
	blocked	64	67	58	67	71	77	58	66
	library	79	84	77	77	83	72	76	71
SGI Power Challenge	standard	21	47	76	73	73	73	73	72
	blocked	106	105	108	107	106	110	98	94
	library	139	132	179	180	162	165	168	170
SUN SPARC 20/50	standard	7	7	15	14	9	9	9	9
	blocked	22	21	25	22	22	21	19	20

Table 4.1: Average performance in Mflop/s of the blocked implementation of DTRSM on RISC processors (using square matrices of order 32, 64, 96, and 128).

Processor	STRSM	Variant							
		'Left'				'Right'			
		'U','N'	'L','N'	'U','T'	'L','T'	'U','N'	'L','N'	'U','T'	'L','T'
DEC3000/400-AXP	standard	14	14	23	24	16	16	16	17
	blocked	56	57	58	74	60	61	57	58
DEC 8400 5/300	standard	41	80	71	78	113	105	102	106
	blocked	259	250	252	212	218	230	213	228
	library	212	229	194	199	204	201	201	195
HP 715/64	standard	13	12	25	23	17	18	15	15
	blocked	48	46	46	48	53	51	47	46
	library	51	48	49	48	62	58	80	69
MEIKO CS2-HA	standard	18	18	46	42	43	42	43	41
	blocked	74	81	76	69	61	58	60	56
IBM RS/6000-750	standard	27	24	25	24	28	35	29	29
	blocked	56	67	56	59	64	61	53	56
	library	79	78	85	82	80	80	73	72
SGI Power Challenge	standard	21	46	64	63	71	72	71	71
	blocked	168	167	151	149	169	166	143	145
	library	141	141	169	168	159	160	164	164
SUN SPARC 20/50	standard	11	12	24	23	13	14	13	13
	blocked	40	45	36	34	38	42	33	38

Table 4.2: Average performance in Mflop/s of the blocked implementation of STRSM on RISC processors (using square matrices of order 32, 64, 96, and 128).

and would be even faster if we used calls to the vendor-supplied GEMM routines from within our blocked code.

5 Blocked implementation of SYMM

SYMM performs one of the matrix-matrix operations :

$$\mathbf{C}{=}\alpha\mathbf{AB}{+}\beta\mathbf{C}, \text{ or } \mathbf{C}{=}\alpha\mathbf{BA}{+}\beta\mathbf{C}$$

where α and β are scalars, $\mathbf{A}$ is an $m \times m$ symmetric matrix (only the upper or lower triangular parts are used) and $\mathbf{B}$ and $\mathbf{C}$ are m $\times$ n matrices.

We consider the following case (corresponding to the parameters "Left", "Upper", i.e. $\mathbf{C}{=}\alpha\mathbf{AB}{+}\beta\mathbf{C}$ where only the upper part of $\mathbf{A}$ is referenced):

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} = \alpha \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{1,2}^t & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} + \beta \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

1. $C_{1,1} \leftarrow \beta C_{1,1} + \alpha A_{1,1} B_{1,1}$ (SYMM)
2. $C_{1,2} \leftarrow \beta C_{1,2} + \alpha A_{1,1} B_{1,2}$ (SYMM)
3. $C_{1,1} \leftarrow C_{1,1} + \alpha A_{1,2} B_{2,1}$ (GEMM)
4. $C_{1,2} \leftarrow C_{1,2} + \alpha A_{1,2} B_{2,2}$ (GEMM)
5. $C_{2,1} \leftarrow \beta C_{2,1} + \alpha A_{2,2} B_{2,1}$ (SYMM)
6. $C_{2,2} \leftarrow \beta C_{2,2} + \alpha A_{2,2} B_{2,2}$ (SYMM)
7. $C_{2,1} \leftarrow C_{2,1} + \alpha A_{1,2}^t B_{1,1}$ (GEMM)
8. $C_{2,2} \leftarrow C_{2,2} + \alpha A_{1,2}^t B_{1,2}$ (GEMM)

Therefore, SYMM can be expressed as a sequence of SYMM and GEMM operations. The SYMM operations are used for the matrix-matrix multiplication involving the blocks $A_{i,i}$ (only the upper triangular part is stored since the submatrices are symmetric). A straightforward way of avoiding the multiplication step by a triangular matrix consists of copying the submatrices $A_{i,i}$ into a working array AA where both the upper and the lower triangular part are stored. Therefore, instead of using a SYMM operation for multiplications using the submatrices $A_{i,i}$, we can use a GEMM operation involving

AA. The additional operations that we make are compensated by the performance gain due to the use of GEMM.

Therefore, SYMM is expressed as a sequence of GEMM operations :

1. Copy $A_{1,1}$ into AA
2. $C_{1,1} \leftarrow \beta C_{1,1} + \alpha AA.B_{1,1}$ (GEMM)
3. $C_{1,2} \leftarrow \beta C_{1,2} + \alpha AA.B_{1,2}$ (GEMM)
4. $C_{1,1} \leftarrow C_{1,1} + \alpha A_{1,2}B_{2,1}$ (GEMM)
5. $C_{1,2} \leftarrow C_{1,2} + \alpha A_{1,2}B_{2,2}$ (GEMM)
6. Copy $A_{2,2}$ into AA
7. $C_{2,1} \leftarrow \beta C_{2,1} + \alpha AA.B_{2,1}$ (GEMM)
8. $C_{2,2} \leftarrow \beta C_{2,2} + \alpha AA.B_{2,2}$ (GEMM)
9. $C_{2,1} \leftarrow C_{2,1} + \alpha A_{1,2}^t B_{1,1}$ (GEMM)
10. $C_{2,2} \leftarrow C_{2,2} + \alpha A_{1,2}^t B_{1,2}$ (GEMM)

The GEMM operations on block rows of **C** can be combined. This allows us to perform GEMM operations on longer vectors and decreases the overhead due to subroutine calls. The corresponding blocked code is reported in Figure 5.1.

We present in Tables 5.1 and 5.2 the performance of the blocked version of SYMM and we compare it to the performance of the standard Fortran version. We see a big improvement over standard Fortran BLAS when using our blocked version, usually a factor of two but occasionally as much as a factor of nearly eight. On the SGI, we can improve the performance of our blocked code by up to about 20% by using larger blocks and so would outperform the vendor code both in single and double precision. It seems clear that DEC have used a similar trick to ours on the DEC 8400 since the performance of their library code for SYMM is close to their GEMM performance.

```
              DO 70 I = 1,M ,NB
                 NB_COL_A = MIN(M-I+1,NB)
c
c     .. Copy diagonal block of A into full array AA.
c        Diagonal block of A is NB_COL_A-by-NB_COL_A and
c        upper triangular.
c
                 DO 42 JJ = 1, NB_COL_A
                    DO 41 II = 1, JJ
                       AA(II,JJ) = A(II+I-1,JJ+I-1)
41                  CONTINUE
42               CONTINUE
                 DO 44 JJ = 1, NB_COL_A
                    DO 43 II = 1, JJ
                       AA(JJ,II) = A(II+I-1,JJ+I-1)
43                  CONTINUE
44               CONTINUE
c
c      .. Multiply diagonal blocks of A.
c
                 DO 45 K=1,N,NB
                    NB_COL_B=MIN(N-K+1,NB)
                    CALL DGEMM('N','N',NB_COL_A,NB_COL_B,
     $                          NB_COL_A, ALPHA, AA, NNB,
     $                          B(I,K), LDB ,BETA,C(I,K),LDC)
45               CONTINUE
c
c      .. Update block row of C.
c
                 DO 60 J=1,I-NB,NB
                    NB_LIG_A=MIN(M-J+1,NB)
                    NB_COL_B=N
                    CALL DGEMM('T','N',NB_COL_A,NB_COL_B,
     $                          NB_LIG_A, ALPHA, A(J,I),LDA,
     $                          B(J,1),LDB,ONE,C(I,1),LDC)
                    CALL DGEMM('N','N',NB_LIG_A,NB_COL_B,
     $                          NB_COL_A, ALPHA, A(J,I),LDA,
     $                          B(I,1),LDB,ONE,C(J,1),LDC)
60               CONTINUE
70            CONTINUE
```

Figure 5.1: Blocked code for SYMM

Processor	DSYMM	Variant			
		'L','U'	'L','L'	'R','U'	'R','L'
DEC3000/400-AXP	standard	24	19	19	18
	blocked	42	42	43	44
DEC 8400 5/300	standard	87	96	91	90
	blocked	188	183	180	183
	library	315	310	300	300
HP 715/64	standard	4	4	16	16
	blocked	28	28	30	31
	library	44	45	45	46
MEIKO CS2-HA	standard	15	15	40	38
	blocked	37	36	40	42
IBM RS/6000-750	standard	21	18	31	30
	blocked	69	71	71	70
	library	75	81	85	76
SGI Power Challenge	standard	82	76	80	80
	blocked	116	116	113	114
	library	81	79	134	134
SUN SPARC 20/50	standard	16	17	11	11
	blocked	23	24	23	23

Table 5.1: Average performance in Mflop/s of the blocked implementation of DSYMM for RISC processors (using square matrices of order 32, 64, 96, and 128).

Processor	SSYMM	Variant			
		'L','U'	'L','L'	'R','U'	'R','L'
DEC3000/400-AXP	standard	24.1	21.1	30.4	29.4
	blocked	65.1	65.3	66.7	66.0
DEC 8400 5/300	standard	93	96	99	97
	blocked	233	228	231	229
	library	381	377	360	366
HP 715/64	standard	4	4	18	17
	blocked	48	48	54	53
	library	89	84	88	83
MEIKO CS2-HA	standard	6	6	40	36
	blocked	60	62	75	72
IBM RS/6000-750	standard	13	12	28	28
	blocked	88	84	84	85
	library	90	93	93	90
SGI Power Challenge	standard	89	87	77	77
	blocked	189	189	158	157
	library	87	83	133	131
SUN SPARC 20/50	standard	26	27	22	22
	blocked	42	38	39	37

Table 5.2: Average performance in Mflop/s of the blocked implementation of SSYMM for RISC processors (using square matrices of order 32, 64, 96, and 128).

6 Blocked implementation of TRMM

TRMM performs one of the matrix-matrix operations :

$$\mathbf{B}{=}\alpha\mathbf{AB},\ \mathbf{B}{=}\alpha\mathbf{A^tB},\ \text{or}\ \mathbf{B}{=}\alpha\mathbf{BA},\ \mathbf{B}{=}\alpha\mathbf{BA^t}$$

where α is a scalar, **B** is an m $\times$ n matrix, **A** is a unit, or non-unit, upper or lower triangular matrix.

We consider the following case (corresponding to the parameters "Left", "No transpose", and "Upper", i.e. $\mathbf{B}{=}\alpha\mathbf{AB}$ where **A** is upper triangular):

$$\begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \alpha \begin{pmatrix} A_{1,1} & A_{1,2} \\ 0 & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

1. $B_{1,1} \leftarrow \alpha A_{1,1} B_{1,1}$ (TRMM)
2. $B_{1,1} \leftarrow B_{1,1} + \alpha A_{1,2} B_{2,1}$ (GEMM)
3. $B_{1,2} \leftarrow \alpha A_{1,1} B_{1,2}$ (TRMM)
4. $B_{1,2} \leftarrow B_{1,2} + \alpha A_{1,2} B_{2,2}$ (GEMM)
5. $B_{2,1} \leftarrow \alpha A_{2,2} B_{2,1}$ (TRMM)
6. $B_{2,2} \leftarrow \alpha A_{2,2} B_{2,2}$ (TRMM)

TRMM is expressed as a sequence of GEMM and TRMM operations. The computations of the submatrices of **B** within the same block row are independent. The GEMM operations can be combined. We use a tuned Fortran code called DTRMML2X2 for performing the multiplication of diagonal blocks of **A**.

The blocked code is reported in Figure 6.1. The code for DTRMML2X2 (TRIADIC option) is reported in Figure A.2 in Appendix.

We report in Tables 6.1 and 6.2 the performance of the blocked version of TRMM in the case where **A** is unit. Our blocked code can be seen to be usually more than twice as efficient as standard BLAS. Although larger blocking on the SGI does not help much on this kernel, we notice that, for most options in single precision, the blocked code outperforms the vendor code. On the DEC 8400, our blocked code performs consistently better than the vendor code, particularly in single precision. The blocked code would be even faster if we used the vendor-supplied GEMM routines.

```
*
*     Form  B := alpha*A*B.
*

      DO 50 K=1,N,NB
         NB_COL_B=MIN(N-K+1,NB)
         DO 40 I = 1,M,NB
            NB_LIG_A = MIN(M-I+1,NB)

            IF ((MOD(NB_LIG_A,2).EQ.0)
     $         .AND.(MOD(NB_COL_B,2).EQ.0)) THEN

               CALL DTRMML2X2_LUN(DIAG,
     $                     NB_LIG_A,NB_COL_B,ALPHA, A(I,I),
     $                     LDA,B(I,K), LDB )
            ELSE

               CALL DTRMML_LUN(DIAG,
     $                     NB_LIG_A,NB_COL_B,ALPHA, A(I,I),
     $                     LDA,B(I,K), LDB )
            END IF

            CALL DGEMM('No transpose','No transpose',
     $                  NB_LIG_A,NB_COL_B,M-I-NB_LIG_A+1,
     $                  ALPHA,A(I,I+NB_LIG_A),LDA,
     $                  B(I+NB_LIG_A,K), LDB,ONE, B(I,K), LDB)

40       CONTINUE
50    CONTINUE
```

Figure 6.1: Blocked code for TRMM

Processor	DTRMM	Variant							
		'Left'				'Right'			
		'U','N'	'L','N'	'U','T'	'L','T'	'U','N'	'L','N'	'U','T'	'L','T'
DEC3000/400-AXP	standard	23	22	20	20	15	15	15	15
	blocked	46	45	45	45	45	44	44	45
DEC 8400 5/300	standard	84	81	74	70	104	101	93	94
	blocked	194	180	183	184	190	182	175	184
	library	175	174	178	175	180	174	172	171
HP 715/64	standard	16	15	21	20	14	15	14	15
	blocked	30	29	33	31	34	33	32	30
	library	41	41	41	39	40	39	39	38
MEIKO CS2-HA	standard	13	12	31	30	11	11	11	12
	blocked	50	47	43	42	49	49	47	44
IBM RS/6000-750	standard	30	28	40	41	32	33	33	32
	blocked	49	69	75	56	72	50	48	63
	library	72	80	80	77	79	84	76	76
SGI Power Challenge	standard	54	53	77	76	73	73	72	72
	blocked	123	122	124	124	116	114	108	109
	library	171	168	183	141	177	168	174	175
SUN SPARC 20/50	standard	11	11	14	14	9	9	9	9
	blocked	22	22	23	24	22	22	19	20

Table 6.1: Average performance in Mflop/s of the blocked implementation of DTRMM for RISC processors (using square matrices of order 32, 64, 96, and 128).

Processor	STRMM	Variant							
		'Left'				'Right'			
		'U','N'	'L','N'	'U','T'	'L','T'	'U','N'	'L','N'	'U','T'	'L','T'
DEC3000/400-AXP	standard	28	28	21	20	16	12	12	12
	blocked	71	73	70	70	72	70	69	71
DEC 8400 5/300	standard	91	88	70	70	107	105	97	108
	blocked	261	233	225	242	236	225	224	232
	library	206	213	206	211	209	206	205	206
HP 715/64	standard	20	17	26	24	15	13	13	14
	blocked	49	49	53	51	56	66	56	53
	library	72	71	69	63	75	72	77	71
MEIKO CS2-HA	standard	18	18	46	42	43	42	43	41
	blocked	74	81	76	69	61	58	60	56
IBM RS/6000-750	standard	28	27	24	23	32	20	19	21
	blocked	60	59	58	61	57	56	58	52
	library	87	80	85	80	84	79	76	84
SGI Power Challenge	standard	50	52	56	54	71	64	63	63
	blocked	193	195	160	160	179	180	166	165
	library	151	151	180	131	176	161	175	174
SUN SPARC 20/50	standard	22	21	19	20	13	9	9	9
	blocked	37	42	39	39	38	39	36	36

Table 6.2: Average performance in Mflop/s of the blocked implementation of STRMM for RISC processors (using square matrices of order 32, 64, 96, and 128).

7 Blocked implementation of SYRK

SYRK performs one of the symmetric rank-k operations :

$$\mathbf{C}{=}\alpha\mathbf{A}\mathbf{A}^{\mathbf{t}}{+}\beta\mathbf{C}, \text{ or } \mathbf{C}{=}\alpha\mathbf{A}^{\mathbf{t}}\mathbf{A}{+}\beta\mathbf{C}$$

where α and β are scalars, $\mathbf{C}$ is an n $\times$ n symmetric matrix (only the upper or lower triangular parts are updated), and $\mathbf{A}$ is a n $\times$ k matrix in the first case and a k $\times$ n matrix in the second case.

We consider the following case (corresponding to "Upper", and "No transpose", i.e. we perform $\mathbf{C}{=}\alpha\mathbf{A}\mathbf{A}^{\mathbf{t}}{+}\beta\mathbf{C}$ where only upper triangular part of $\mathbf{C}$ is updated):

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ 0 & C_{2,2} \end{pmatrix} = \alpha \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} A_{1,1}^t & A_{2,1}^t \\ A_{1,2}^t & A_{2,2}^t \end{pmatrix} + \beta \begin{pmatrix} C_{1,1} & C_{1,2} \\ 0 & C_{2,2} \end{pmatrix}$$

1. $C_{1,1} \leftarrow \beta C_{1,1} + \alpha A_{1,1} A_{1,1}^t$ (SYRK)
2. $C_{1,1} \leftarrow C_{1,1} + \alpha A_{1,2} A_{1,2}^t$ (SYRK)
3. $C_{1,2} \leftarrow \beta C_{1,2} + \alpha A_{1,1} A_{2,1}^t$ (GEMM)
4. $C_{1,2} \leftarrow C_{1,2} + \alpha A_{1,2} A_{2,2}^t$ (GEMM)
5. $C_{2,2} \leftarrow \beta C_{2,2} + \alpha A_{2,1} A_{2,1}^t$ (SYRK)
6. $C_{2,2} \leftarrow C_{2,2} + \alpha A_{2,2} A_{2,2}^t$ (SYRK)

The symmetric rank-k update is expressed as a sequence of SYRK for updating the submatrices $C_{i,i}$ and GEMM for the other blocks. The updates of the submatrices of $\mathbf{C}$ can be performed independently. The GEMM updates of off-diagonal blocks can be combined. Note that, at the price of extra operations, we could perform the update of the diagonal blocks of $\mathbf{C}$ using GEMM instead of SYRK.

The corresponding blocked code is shown in Figure 7.1. Note that it is more efficient to perform the multiplication of matrix C by β before calling GEMM rather than performing this multiplication within GEMM. The code for SYRKL2X2 (TRIADIC option) is reported in Figure A.3 in Appendix. In Tables 7.1, and 7.2 the performance of the standard Fortran code and of the blocked implementation are compared for all the variants. For this kernel, our gains over using standard BLAS are significant, usually by a factor of close to two. Although using a larger block size on the SGI improves performance by usually less than 10%, we consistently outperform the vendor code by a significant amount in single precision, even at the lower block size. Our blocked code is substantially better than the vendor kernel on the DEC 8400 and would be even faster if we used the vendor-supplied GEMM.

```
              DO 130, I = 1, N,NB
                 NB_LIG_C=MIN(NB,N-I+1)
*
*       Multiplication of diagonal block of C
*

                 IF (BETA.EQ.ZERO) THEN
                    DO J = 1, NB_LIG_C
                       DO II = 1, J
                          C(II+I-1,J+I-1) = ZERO
                       ENDDO
                    ENDDO
                 ELSE
                    DO J = 1, NB_LIG_C
                       DO II = 1, J
                          C(II+I-1,J+I-1) = BETA*C(II+I-1,J+I-1)
                       ENDDO
                    ENDDO
                 END IF

                 DO 90, L=1,K,NB

                 NB_COL_A=MIN(NB,K-L+1)

               IF ((MOD(NB_LIG_C,2).EQ.0).AND.(MOD(NB_COL_A,2).EQ.0))THEN
                         CALL DSYRKL2X2(NB_LIG_C,NB_COL_A,ALPHA,
     $                                  A(I,L),LDA,ONE,C(I,I),LDC)
                 ELSE
                         CALL DSYRKL(NB_LIG_C,NB_COL_A,ALPHA,
     $                                  A(I,L),LDA,ONE,C(I,I),LDC)
                 END IF

90               CONTINUE

                 NB_COL_C=N-NB_LIG_C-I+1
                 NB_COL_A=K

                  CALL DGEMM('N','T',NB_LIG_C,NB_COL_C,NB_COL_A,
     $                       ALPHA,A(I,1),LDA,A(I+NB_LIG_C,1),LDA,
     $                       BETA, C(I,I+NB_LIG_C),LDC)

130       CONTINUE
```

Figure 7.1: Blocked code for SYRK

Processor	DSYRK	Variant			
		'U','N'	'L','N'	'U','T'	'L','T'
DEC3000/400-AXP	standard	21	21	22	22
	blocked	42	42	48	49
DEC 8400 5/300	standard	81	82	74	78
	blocked	172	162	182	189
	library	82	86	100	85
HP 715/64	standard	17	15	24	24
	blocked	26	23	32	31
	library	16	16	28	27
MEIKO CS2-HA	standard	20	20	37	36
	blocked	44	44	40	40
IBM RS/6000-750	standard	29	28	46	47
	blocked	55	54	70	69
	library	88	83	83	85
SGI Power Challenge	standard	56	55	105	105
	blocked	115	114	135	137
	library	111	111	114	114
SUN SPARC 20/50	standard	11	11	15	15
	blocked	18	18	23	24

Table 7.1: Average performance in Mflop/s of the blocked implementation of DSYRK for RISC processors (using square matrices of order 32, 64, 96, and 128).

Processor	SSYRK	Variant			
		'U','N'	'L','N'	'U','T'	'L','T'
DEC3000/400-AXP	standard	27	27	25	24
	blocked	62	60	70	65
DEC 8400 5/300	standard	87	89	75	79
	blocked	224	215	241	247
	library	87	94	106	90
HP 715-64	standard	21	19	28	26
	blocked	56	47	68	62
	library	19	19	29	30
MEIKO CS2-HA	standard	32	29	48	47
	blocked	66	57	70	65
IBM RS/6000-750	standard	28	26	23	24
	blocked	50	45	50	49
	library	90	95	95	103
SGI Power Challenge	standard	51	54	79	80
	blocked	165	162	166	168
	library	99	98	100	99
SUN SPARC 20/50	standard	21	21	25	23
	blocked	33	32	39	38

Table 7.2: Average performance in Mflop/s of the blocked implementation of SSYRK for RISC processors (using square matrices of order 32, 64, 96, and 128).

8 Blocked implementation of SYR2K

SYR2K performs one of the symmetric rank-2k operations:

$$\mathbf{C}{=}\alpha\mathbf{A}\mathbf{B}^{\mathbf{t}}{+}\alpha\mathbf{B}\mathbf{A}^{\mathbf{t}}{+}\beta\mathbf{C}, \text{ or } \mathbf{C}{=}\alpha\mathbf{A}^{\mathbf{t}}\mathbf{B}{+}\alpha\mathbf{B}^{\mathbf{t}}\mathbf{A}{+}\beta\mathbf{C}$$

where α and β are scalars, $\mathbf{C}$ is an n $\times$ n symmetric matrix (only the upper or lower triangular parts are updated) and $\mathbf{A}$ and $\mathbf{B}$ are n $\times$ k matrices in the first case and k $\times$ n matrices in the second case.

We consider the following case (corresponding to "Upper", and "No transpose", i.e $\mathbf{C}{=}\alpha\mathbf{A}\mathbf{B}^{\mathbf{t}}{+}\alpha\mathbf{B}\mathbf{A}^{\mathbf{t}}{+}\beta\mathbf{C}$ where only the upper triangular part of $\mathbf{C}$ is updated):

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ 0 & C_{2,2} \end{pmatrix} = \alpha \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1}^t & B_{2,1}^t \\ B_{1,2}^t & B_{2,2}^t \end{pmatrix}$$

$$+\alpha \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \begin{pmatrix} A_{1,1}^t & A_{2,1}^t \\ A_{1,2}^t & A_{2,2}^t \end{pmatrix} + \beta \begin{pmatrix} C_{1,1} & C_{1,2} \\ 0 & C_{2,2} \end{pmatrix}$$

1. $C_{1,1} \leftarrow \beta C_{1,1} + \alpha A_{1,1} B_{1,1}^t + \alpha B_{1,1} A_{1,1}^t$ (SYR2K)
2. $C_{1,1} \leftarrow C_{1,1} + \alpha A_{1,2} B_{1,2}^t + \alpha B_{1,2} A_{1,2}^t$ (SYR2K)
3. $C_{1,2} \leftarrow \beta C_{1,2} + \alpha A_{1,1} B_{2,1}^t$ (GEMM)
4. $C_{1,2} \leftarrow C_{1,2} + \alpha B_{1,1} A_{2,1}^t$ (GEMM)
5. $C_{1,2} \leftarrow C_{1,2} + \alpha A_{1,2} B_{2,2}^t$ (GEMM)
6. $C_{1,2} \leftarrow C_{1,2} + \alpha B_{1,2} A_{2,2}^t$ (GEMM)
7. $C_{2,2} \leftarrow \beta C_{2,2} + \alpha A_{2,1} B_{2,1}^t + \alpha B_{2,1} A_{2,1}^t$ (SYR2K)
8. $C_{2,2} \leftarrow C_{2,2} + \alpha A_{2,2} B_{2,2}^t + \alpha B_{2,2} A_{2,2}^t$ (SYR2K)

SYR2K is expressed as a sequence of SYR2K for updating the triangular submatrices $C_{i,i}$, and GEMM on the other blocks. The update of the submatrices of $\mathbf{C}$ can be effected simultaneously. There is no need to compute both $\alpha\mathbf{A}\mathbf{B}^t$ and $\alpha\mathbf{B}\mathbf{A}^t$ (since it is the same matrix but transposed). Thus, only one of the two operations is performed and the result is stored into a working array called CC. This can be done using GEMM.

Using these remarks SYR2K is computed in the following way :

1. $CC \leftarrow \alpha A_{1,1}.B_{1,1}^t$ (GEMM)
2. $CC \leftarrow CC + \alpha A_{1,2}.B_{1,2}^t$ (GEMM)
3. $C_{1,1} \leftarrow \beta C_{1,1} + CC + CC^t$
4. $C_{1,2} \leftarrow \beta C_{1,2} + \alpha A_{1,1} B_{2,1}^t$ (GEMM)
5. $C_{1,2} \leftarrow C_{1,2} + \alpha B_{1,1} A_{2,1}^t$ (GEMM)
6. $C_{1,2} \leftarrow C_{1,2} + \alpha A_{1,2} B_{2,2}^t$ (GEMM)
7. $C_{1,2} \leftarrow C_{1,2} + \alpha B_{1,2} A_{2,2}^t$ (GEMM)
8. $CC \leftarrow \alpha A_{2,1}.B_{2,1}^t$ (GEMM)
9. $CC \leftarrow CC + \alpha A_{2,2}.B_{2,2}^t$ (GEMM)
10. $C_{2,2} \leftarrow \beta C_{2,2} + CC + CC^t$

As for SYRK, with a larger number of blocks, the GEMM updates of the off-diagonal blocks can be combined. The corresponding blocked code is given in Figure 8.1. We report in Tables 8.1 and 8.2 the results obtained using the blocked implementation of SYR2K. We note that SYR2K is the one kernel where the ESSL Library on the IBM has significantly better performance than the standard vendor-supplied BLAS. Although using a larger block size on the SGI improves performance by between 10% and 20%, even at the lower block size we consistently outperform the vendor code, by a significant amount in single precision. Our blocked code is substantially better than the vendor kernel on the DEC 8400 and would be even faster if we used the vendor-supplied GEMM.

```
      DO 130, I = 1, N,NB
         NB_LIG_C=MIN(NB,N-I+1)
         CALL DGEMM('N','T',NB_LIG_C,NB_LIG_C,K,
     $              ALPHA,A(I,1),LDA,B(I,1),LDB,ZERO,CC,NB)
         DO JJ=1,NB_LIG_C
            DO II=1,JJ
               C(I+II-1,I+JJ-1) = BETA*C(I+II-1,I+JJ-1)
               C(I+II-1,I+JJ-1) = C(I+II-1,I+JJ-1)+CC(II,JJ)
               C(I+II-1,I+JJ-1) = C(I+II-1,I+JJ-1)+CC(JJ,II)
            ENDDO
         ENDDO
         J=I+NB_LIG_C
         NB_COL_C=MAX(N-J+1,0)
         CALL DGEMM('N','T',NB_LIG_C,NB_COL_C,K,
     $              ALPHA,A(I,1),LDA,B(J,1),LDB,BETA,C(I,J),LDC)
         CALL DGEMM('N','T',NB_LIG_C,NB_COL_C,K,
     $              ALPHA,B(I,1),LDB,A(J,1),LDA,ONE,C(I,J),LDC)
130      CONTINUE
```

Figure 8.1: Blocked code for SYR2K

Processor	DSYR2K	Variant			
		'U','N'	'L','N'	'U','T'	'L','T'
DEC3000/400-AXP	standard	23	20	14	14
	blocked	50	49	47	48
DEC 8400 5/300	standard	83	89	82	82
	blocked	187	181	191	198
	library	106	103	130	127
HP 715/64	standard	5	5	2	2
	blocked	33	31	34	35
	library	5	5	3	3
MEIKO CS2-HA	standard	15	17	10	10
	blocked	44	44	46	46
IBM RS/6000-750	standard	18	15	39	35
	blocked	70	66	81	82
	library	19	15	36	36
SGI Power Challenge	standard	87	87	89	88
	blocked	115	113	126	128
	library	112	113	116	115
SUN SPARC 20/50	standard	13	13	15	15
	blocked	23	22	24	25

Table 8.1: Average performance in Mflop/s of the blocked implementation of DSYR2K for RISC processors (using square matrices of order 32, 64, 96, and 128).

Processor	SSYR2K	Variant			
		'U','N'	'L','N'	'U','T'	'L','T'
DEC3000/400-AXP	standard	25	23	14	14
	blocked	69	70	67	67
DEC 8400 5/300	standard	92	97	87	88
	blocked	254	251	241	240
	library	127	126	139	136
HP 715/64	standard	5	5	2	2
	blocked	60	59	61	59
	library	5	5	2	2
MEIKO CS2-HA	standard	7	7	3	3
	blocked	64	64	75	79
IBM RS/6000-750	standard	14	11	38	38
	blocked	81	79	90	87
	library	14	11	35	35
SGI Power Challenge	standard	76	76	78	77
	blocked	159	158	199	200
	library	102	100	101	100
SUN SPARC 20/50	standard	23	23	30	28
	blocked	39	38	40	40

Table 8.2: Average performance in Mflop/s of the blocked implementation of SSYR2K for RISC processors (using square matrices of order 32, 64, 96, and 128).

9 Conclusion

The Level 3 BLAS are a set of computational kernels targeted at matrix-matrix operations with the aim of providing efficient and portable implementations of algorithms on high-performance computers. The linear algebra package LAPACK (Anderson, Bai, Bischof, Demmel, Dongarra, DuCroz, Greenbaum, Hammarling, McKenney, Ostrouchov & Sorensen 1995), for example, makes extensive use of the Level 3 BLAS. We have described an efficient and portable implementation of the Level 3 BLAS for RISC processors.

The Level 3 BLAS are expressed as a sequence of matrix-matrix multiplications (GEMM) and operations involving triangular blocks. The combination of blocking, copying, and loop unrolling allows efficient exploitation of the memory hierarchy and only the blocking parameter and the loop unrolling depth are machine dependent. Both the performance of GEMM and performance of the kernel dealing with triangular matrices are crucial.

We have shown here that significant Megaflop rates can be achieved, only using tuned Fortran kernels. Although our primary aim is not to outperform the vendor-supplied libraries, our portable implementation compares reasonably well with the manufacturer-supplied libraries on the IBM RS/6000, the HP 715/64, the DEC 8400 5/300, and the SGI Power Challenge. It is interesting that, although the vendor-supplied GEMM routines are better than our blocked version of GEMM on the DEC 8400 5/300 and the SGI Power Challenge, many of our blocked versions of the other kernels are better than the vendor-supplied equivalents, sometimes by a large margin. Note also that the availability of a highly tuned version of the matrix-matrix multiplication kernel GEMM would improve the performance figures of our blocked code substantially. For example, when using the manufacturer-supplied version of DGEMM within our blocked version of DTRSM, we achieve a close or marginally better performance than that of the DTRSM kernel available in the vendor-supplied library on the HP 715/64 and the IBM RS/6000-750. We would suggest that some vendors could easily increase the performance of their non-GEMM Level 3 BLAS kernels by using the techniques described in this paper. Finally, for some machines, perfomance could be enhanced by judiciously selecting appropriate leading dimensions of the matrices although we do not consider this because it is dependent on the machine architecture and cache management strategy.

We demonstrated in Daydé et al. (1994) how this blocked version could be used to parallelize the Level 3 BLAS. A preliminary version was successfully used for developing both serial and parallel tuned versions of the Level 3 BLAS for a 30-node BBN-TC2000 (Amestoy et al. 1995, Daydé & Duff 1995). We are currently experimenting on other shared and virtual shared memory machines in order to develop tuned serial and parallel implementations for them.

10 Availability of codes

The codes described in the present paper are available using ftp anonymous at ftp.enseeiht.fr. The software is located in pub/numerique/BLAS/RISC. A compressed

tarfile called blas_risc.tar.Z contains the following codes :

- A set of test routines that check the correct execution and compute the Megaflop rates of the blocked implementation compared with the standard version of the Level 3 BLAS.
- The blocked implementation of the Level 3 BLAS.

We advise the user to check the availability of tuned serial codes (manufacturer-supplied library) before using our blocked implementation.

11 Acknowledgments

We are grateful to Nick Hill of the Rutherford Appleton Laboratory for his advice on the DEC 8400, to Andrew Cliffe of AEA Technology, Harwell for performing the runs on the SGI Power Challenge, and to Gérard Le Blanc for help in accessing the IBM RS/6000-750 at IMFT in Toulouse.

References

Agarwal, R. C., Gustavson, F. G. & Zubair, M. (1994), 'Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms', *IBM Journal of Research and Development* **38**, 563–576.

Amestoy, P. R. & Duff, I. S. (1989), 'Vectorization of a multiprocessor multifrontal code', *Int. J. of Supercomputer Applics.* **3**, 41–59.

Amestoy, P. R., Daydé, M. J., Duff, I. S. & Morère, P. (1995), 'Linear algebra calculations on a virtual shared memory computer', *Int Journal of High Speed Computing* **7**(1), 21–43.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S. & Sorensen, D. (1995), *LAPACK Users' Guide, second edition*, SIAM Press.

Bodin, F. & Seznec, A. (1994), Cache organization influence on loop blocking, Technical Report 803, IRISA, Rennes, France.

Daydé, M. J. & Duff, I. S. (1995), 'Porting industrial codes and developing sparse linear solvers on parallel computers', *Computing Systems in Engineering* **6**(4/5), 295–305.

Daydé, M. J., Duff, I. S. & Petitet, A. (1994), 'A parallel block implementation of Level 3 BLAS kernels for MIMD vector processors', *ACM Transactions on Mathematical Software* **20**, 178–193.

Dongarra, J. J. (1992), Performance of various computers using standard linear algebra software, Technical Report CS-89-85, University of Tennessee, Knoxville, Tennessee.

Dongarra, J. J., Du Croz, J., Duff, I. S. & Hammarling, S. (1990*a*), 'Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms: Model implementation and test programs', *ACM Transactions on Mathematical Software* **16**(1), 18–28.

Dongarra, J. J., Du Croz, J., Duff, I. S. & Hammarling, S. (1990*b*), 'A set of Level 3 Basic Linear Algebra Subprograms.', *ACM Transactions on Mathematical Software* **16**, 1–17.

Dongarra, J. J., Mayes, P. & Radicati di Brozolo, G. (1991), Lapack working note 28 : The IBM RISC System/6000 and linear algebra operations, Technical Report CS-91-130, University of Tennessee.

Gallivan, K., Jalby, W. & Meier, U. (1987), 'The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory', *SIAM Journal on Scientific and Statistical Computing* **8**, 1079–1084.

Gallivan, K., Jalby, W., Meier, U. & A., S. (1988), 'Impact of hierarchical memory systems on linear algebra algorithm design', *Int Journal of Supercomputer Applications* **2**(1), 12–48.

Kågström, B., Ling, P. & Loan, C. V. (1993), Portable high performance GEMM-based Level-3 BLAS, *in* R. F. Sincovec et al., ed., 'To appear Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing', SIAM.

Puglisi, C. (1993), **QR** Factorization of Large Sparse Overdetermined and Square Matrices using the Multifrontal Method in a Multiprocessor Environment, PhD thesis, CERFACS. Technical Report TH/PA/93/33.

Qrichi Aniba, A. (1994), Implémentation performante di BLAS de niveau 3 pour les processeurs RISC, Technical Report Rapport 3ème Année, Département Informatique et Mathématiques Appliquées, ENSEEIHT.

A Appendix

We show here some of the tuned Fortran codes referenced in the paper.

```
      IF( ALPHA.NE.ONE )THEN
        DO 80  J = 1, N
          DO 70  I = 1, M
            B( I, J ) = ALPHA*B( I, J )
 70       CONTINUE
 80    CONTINUE
      END IF
*
      DO 50 I = M , 1, -2
         DO K = M, I+1, -1
            AA2(K) = A(I-1,K)
            AA1(K) = A(I,K)
         ENDDO
         DO 40 J = 1, N, 2
            B11 = B(I  ,J  )
            B21 = B(I-1,J  )
            B12 = B(I  ,J+1)
            B22 = B(I-1,J+1)
*
*     Update B
*
            DO 30 K = M, I + 1, -1
               B1 = B(K,J  )
               B2 = B(K,J+1)
               A1 = AA1(K)
               A2 = AA2(K)
               B11 = B11 - A1*B1
               B21 = B21 - A2*B1
               B12 = B12 - A1*B2
               B22 = B22 - A2*B2
 30         CONTINUE
*
*     Compute solution
*
            IF ( NOUNIT) THEN
                 T1  = ONE / A( I, I )
                 B11 = B11 * T1
                 B12 = B12 * T1
                 U1  = ONE / A( I-1, I-1 )
            END IF
            A1 = A( I - 1, I )
            B21 = B21 - B11 * A1
            B22 = B22 - B12 * A1
            IF ( NOUNIT) THEN
                 B21 = B21 * U1
                 B22 = B22 * U1
            END IF
            B( I    , J     ) = B11
            B( I - 1, J     ) = B21
            B( I    , J + 1 ) = B12
            B( I - 1, J + 1 ) = B22
 40      CONTINUE
 50   CONTINUE
```

Figure A.1: Tuned code for TRSM

```
        DO 20 I=1,M,2
         DO K=I+2,M
            AA1(K) = A(I,K)
            AA2(K) = A(I+1,K)
         ENDDO
         DO 10 J=1,N,2
            T11 = B(I,J)
            T21 = B(I+1,J)
            T12 = B(I,J+1)
            T22 = B(I+1,J+1)
*
*          .. Update triangular block
*
            A1 = A(I,I+1)
            T11 = T11 + A1*T21
            T12 = T12 + A1*T22
            DO K=I+2,M
                 A1 = AA1(K)
                 A2 = AA2(K)
                 B1 = B(K,J)
                 B2 = B(K,J+1)
                 T11 = T11 + A1*B1
                 T21 = T21 + A2*B1
                 T12 = T12 + A1*B2
                 T22 = T22 + A2*B2
            ENDDO
         B(I,J) = T11
         B(I+1,J) = T21
         B(I,J+1) = T12
         B(I+1,J+1) = T22
   10   CONTINUE
   20  CONTINUE
```

Figure A.2: Tuned code for TRMM

```
      JSTART=1
      DO 10, I=1,N,2
         T11 = BETA*C(I,I)
         T12 = BETA*C(I,I+1)
         T22 = BETA*C(I+1,I+1)
         DO L = 1,K
            AA1(L) = A(I,L)
            AA2(L) = A(I+1,L)
         ENDDO
         DO 15, L = 1,K
            A1 = AA1(L)
            A2 = AA2(L)
            B1 = ALPHA*A1
            B2 = ALPHA*A2
            T11 = T11 + B1*A1
            T12 = T12 + B1*A2
            T22 = T22 + B2*A2
   15    CONTINUE
         C(I,I) = T11
         C(I,I+1) = T12
         C(I+1,I+1) = T22
         JSTART=JSTART+2
         DO 20 J=JSTART,N,2
            T11 = BETA*C(I,J)
            T21 = BETA*C(I+1,J)
            T12 = BETA*C(I,J+1)
            T22 = BETA*C(I+1,J+1)
            DO L = 1, K
               B1 = ALPHA*A(J,L)
               B2 = ALPHA*A(J+1,L)
               A1 = AA1(L)
               A2 = AA2(L)
               T11 = T11 + B1*A1
               T21 = T21 + B1*A2
               T12 = T12 + B2*A1
               T22 = T22 + B2*A2
            ENDDO
            C(I,J) = T11
            C(I+1,J) = T21
            C(I,J+1) = T12
            C(I+1,J+1) = T22
   20    CONTINUE
   10 CONTINUE
```

Figure A.3: Tuned code for SYRK