The RISC BLAS: A Blocked Implementation of Level 3 BLAS for RISC Processors

MICHEL J. DAYDÉ
ENSEEIHT-IRIT
and
IAIN S. DUFF
CERFACS and Rutherford Appleton Laboratory

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 authors and colleagues on a similar approach for efficient serial and parallel implementations on virtual-memory and 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 our implementation for vector machines except that the ordering of loops is designed for efficient reuse of data held in cache and not necessarily for parallelization. All the codes are specifically tuned for RISC processors. The software also includes a tuned version of GEMM. 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: CRAY T3D, DEC 8400 5/300, HP 715/64, IBM SP2, MEIKO CS2-HA, SGI Power Challenge 10000, and SUN UltraSPARC-1 model 140. This implementation of the Level 3 BLAS is available on anonymous FTP, and we welcome input from users to improve and extend our BLAS implementation.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: Mathematical Software; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*Computations on matrices*; G.1.0 [**Numerical Analysis**]: General—*Numerical algorithms*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear systems* (direct and iterative methods)

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Blocking, Level 3 BLAS, loop-unrolling, matrix-matrix kernels, RISC processors

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

Authors' addresses: M. J. Daydé, ENSEEIHT-IRIT, 2 rue Camichel, Toulouse Cedex, 31071, France; I. S. Duff, Rutherford Appleton Laboratory, Oxon, OX11 0QX, England.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2000 ACM 0098-3500/99/0900–0316 $5.00

1. INTRODUCTION

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 et al. 1995], for example, makes extensive use of the Level 3 BLAS.

This article describes a version of single- and double-precision Level 3 BLAS computational kernels [Dongarra et al. 1990a; 1990b] called the *RISC 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] for MIMD vector processors. 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. Thus, all the codes are specifically tuned for RISC processors, and the software includes a tuned version of GEMM.

Our basic idea in the design of 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 are tuned using blocking, copying, and loop-unrolling. We believe these codes provide an efficient implementation of the Level 3 BLAS on computers where a highly tuned version is not available. In this article, 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 (see Section 9). 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 only included a few specific tuning techniques that are crucial on some computers. Additionally, our experiments often use nonideal—critical—leading dimensions for the matrices involved in the calculations (e.g., powers of two). On some machines the times would be better for other values. We would be happy to discuss with users and vendors the possibil-

ity 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.

This implementation of the Level 3 BLAS is available on anonymous FTP, and we welcome input from users to improve and extend our BLAS implementation. More details and more experiments can be found in Daydé and Duff [1996] and Daydé and Duff [1997].

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 T3E, SPARC on CM5 and PCI CS2, HP PA on CONVEX EXEMPLAR, and RS/6000 on IBM SP1 and SP2).

We report results from uniprocessor executions on a range of RISC-based computers (in practice, we have performed experiments on a larger set of machines):

(1) CRAY T3D (1 node) located at IDRIS

(2) DEC 8400 5/300 located at RAL

(3) HP 715/64 located at ENSEEIHT

(4) IBM SP2 (1 thin node) located at CNUSC

(5) MEIKO CS2-HA (using a HyperSparc processor) located at CERFACS

(6) SGI Power Challenge 10000 using a MIPS R10000 processor located at CERFACS

(7) SUN UltraSPARC-1 model 140 located at ENSEEIHT

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 et al. 1987; 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 [Bodin and 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, assuming `a(i)` is one word and assuming the cache line length is equal to four words (assuming that the cache is initially empty), when executing the loop

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

then each read of `a(i)` causes a cache miss.

Copying blocks of data (e.g., submatrices) that are heavily reused may help to improve memory and cache accesses (e.g., by avoiding critical strides). Since it may induce a large overhead, it is, however, not always a viable technique (e.g., when the number of memory references required by the copy is the same order as the number of flops involved in the calculation to be performed). We illustrate copying 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.

2.3 Motivations and Design of the RISC BLAS

We have previously implemented full and sparse linear solvers on computers where a tuned version of the BLAS was not available (or was not available without cost), or where the tuning of some of the BLAS kernels was not done efficiently. Because the performance of the Level 3 BLAS is crucial to most of our work, we decided to invest some time in the tuning of the Level 3 BLAS kernels.

Our main goal when designing the RISC BLAS was to provide reasonable performance with very simple software, on a range of computers. Our interest in RISC processors arose from the fact that they are used in workstations and parallel computers where a well-tuned version of the BLAS was often not available.

We considered the blocking of the triangular solver from the Level 3 BLAS—TRSM—in Daydé and Duff [1989]. Then, we developed a blocked version of the Level 3 BLAS for MIMD vector multiprocessors [Amestoy and Daydé 1993; Daydé et al. 1994]. At the same time, we also studied the development of a parallel version of the Level 3 BLAS for Transputers [Berger et al. 1991]. Some of the ideas in the Transputer parallel version and in the blocked version for MIMD vector processors were used to design the serial and parallel versions of the Level 3 BLAS for the BBN TC2000 [Amestoy et al. 1995]. Note that the BBN TC2000 used a RISC processor: the Motorola 88100. The serial version developed for the BBN was extended and modified to be portable and efficient on a wide range of RISC processors [Daydé and Duff 1996; Qrichi Aniba 1994]. The corresponding software—Version 0—was made available in 1995 and was installed in various places. The version we refer to in the present article is Version 1.0.

The RISC BLAS differs from the blocked version of the Level 3 BLAS for MIMD vector multiprocessors in the following ways:

—The loop ordering is dictated by consideration of efficient cache reuse rather than parallel implementation.

—The codes are now tuned for RISC processors rather than for vector processors, and, additionally, we provide a tuned GEMM code.

—SYMM and SYR2K are blocked in a different way and only make use of GEMM (as described in Ling [1993] and suggested implicitly in Sheikh and Liu [1989]).

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 perform GEMM operations on square submatrices of order NB or perform operations involving triangular submatrices. Additionally, all the calculations in these subkernels 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. Our approach is relatively independent of the computer: only the NB parameter, corresponding to the block size, and in some cases the loop-unrolling depth need to be set 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 by the number of scalar registers. The value of NB is set within the installation makefile by selecting an architecture name.

Note that Kågström et al. [1998a; 1998b] use similar ideas. Using their terminology, the RISC BLAS is a GEMM-based BLAS. In fact, most of the manufacturers develop BLAS in that way [Sheikh and Liu 1989].

The main differences between the RISC BLAS and the work by Kågström et al. are the following:

—The RISC BLAS only makes use of Level 3 BLAS operations. These operations are effected using the tuned Fortran codes included in our

software. A lot of effort has been made to provide tuned building blocks (using blocking, copying, and loop-unrolling) for all the kernels including GEMM.

—The GEMM-based BLAS does not provide a tuned implementation of GEMM and relies on the one available on the computer. It is based on the use of GEMM and Level 1 and Level 2-based operations.

—The RISC BLAS is still a on-going effort. Version 2.0 will be delivered soon: some improvements in the GEMM design (e.g., multilevel blocking) have been implemented, and complex versions of GEMM will be included.

—We incorporate some specific optimizations in our software that appear to be crucial on some processors (e.g., we have a version tuned for the SGI Power Challenge 10000), but we have not included all possible optimizations, in order to keep the code simple (our intention is not to provide the best possible implementation on a particular processor but a good one over a range of processors). We will certainly distribute separately tuned versions of the RISC BLAS for specific processors in the future.

The installation makefile provided in our software offers default options for a wide range of computers (including those used in our experiments). Basically, the user has only to select an architecture name (e.g., RS6K64 for an IBM RS6000 Power with a 64KB cache or SPARC10 for SUN workstations using a SPARC 10 processor), the corresponding organization of operations (TRIADIC or NOTRIADIC for the IBM RS6000 and the SUN, respectively) according to the recommendations in the makefile, and the compiler and linker options (we provide default options). Examples when using GEMM are included in Section 3.1. We use the C preprocessor as the main mechanism to generate the version of the RISC BLAS for a particular processor. Our software has been tested on a wide range of RISC processors running the UNIX operating system. We have not yet studied extensions for non-UNIX systems, but generating a Fortran version (without C preprocessor directives) before porting the code is straightforward.

Modifying the software to add a new processor is extremely simple: only the block size (which depends on the cache size) has to be set using a very simple calculation rule as explained in Section 3.1. The TRIADIC organization of operations is almost always the best choice.

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, e.g., 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 square blocks of order NB 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

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

where α and β are scalars; A and B are rectangular matrices of dimensions $m \times k$ and $k \times n$, respectively; C is an $m \times n$ matrix; and op(A) is A or A^T.

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

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \leftarrow \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} + \beta \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

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

(1) $C_{11} \leftarrow \beta C_{11} + \alpha A_{11} B_{11}$ (GEMM)

(2) $C_{11} \leftarrow C_{11} + \alpha A_{12} B_{21}$ (GEMM)

(3) $C_{12} \leftarrow \beta C_{12} + \alpha A_{11} B_{12}$ (GEMM)

(4) $C_{12} \leftarrow C_{12} + \alpha A_{12} B_{22}$ (GEMM)

(5) $C_{21} \leftarrow \beta C_{21} + \alpha A_{21} B_{11}$ (GEMM)

(6) $C_{21} \leftarrow C_{21} + \alpha A_{22} B_{21}$ (GEMM)

(7) $C_{22} \leftarrow \beta C_{22} + \alpha A_{21} B_{12}$ (GEMM)

(8) $C_{22} \leftarrow C_{22} + \alpha A_{22} B_{22}$ (GEMM)

The ordering of these eight computational steps is determined by considerations on efficient reutilization of data held in cache. The submatrix of A, because of the access by rows, leads to nonunit strides in the innermost loops of the calculations. It is then multiplied by α and transposed into a working array AA. AA is kept in cache as long as required. This is why we have decided to organize the calculations in order to reuse the submatrices of A as much as possible (thus, trying to amortize the cost of this copy), and we perform all operations involving a submatrix before moving to another one (see Figure 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 et al. [1991].

The blocked code is reported in Figure 1. We use two tuned Fortran codes to perform calculations on submatrices (see Figure 2):

```
*
*     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 62 II = I, I + IB - 1
               DO 61 LL = L, L + LB - 1
                  AA(LL-L+1,II-I+1)=ALPHA*A(II,LL)
61             CONTINUE
62          CONTINUE
            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_NN(IB,JB,LB,AA,NB,B(L,J),LDB,C(I,J),LDC)
               ELSE
                  CALL DGEMML_NN(IB,JB,LB,AA,NB,B(L,J),LDB,C(I,J),LDC)
               END IF
 50         CONTINUE
 60      CONTINUE
 70   CONTINUE
```

Fig. 1. Blocked code for GEMM.

—DGEMML2X2_NN is a tuned code for performing matrix-matrix multiplication on square matrices of even order.

—DGEMML_NN is a tuned code that includes additional tests over DGEMML2X2_NN to handle matrices of odd order. It is occasionally slightly less efficient than DGEMML2X2_NN.

```
*
*           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
```

```
*
*           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
```

Fig. 2. Tuned code for GEMM (left: TRIADIC option, right: NOTRIADIC option).

We have used two versions for all the tuned codes:

—the TRIADIC option for computers where triadic operations are either supported in the hardware (e.g., the floating-point multiply-and-add on IBM SP2) or are 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_NN using the TRIADIC options is shown on the left side of Figure 2, while the code corresponding to the NOTRIADIC option is shown on the right side. The selection between the options is effected using the C preprocessor that generates one of the two codes. All

the tuned codes described in the rest of this article offer both options. In Version 1.0 of the RISC BLAS, we use a 2-by-2 and a 4-by-2 unrolling in double and single precision, respectively. In Version 2.0, the same loop-unrolling depth will be used in both precisions on 64-bit processors such the DEC, the IBM, and the MIPS processors.

There is some freedom for selecting an appropriate block size. Since the elements of the working array AA that are used to store the transpose of the submatrix of A, and the submatrix of B, are referenced several times in the innermost computational loops (see Figure 2), NB should be chosen to guarantee that these subarrays fit in cache. Additionally, the elements of C that are updated should also fit in cache [Bodin and Seznec 1994; Gallivan et al. 1988; Hennessy and Patterson 1996; Ling 1993].

If the leading dimension of an array is critical—e.g., a multiple of a power of two—the effective space in the cache to hold a submatrix of that array may be drastically reduced (typically, several elements of the submatrix are to be stored at the same cache location, increasing the number of cache misses). It is then useful to detect these critical leading dimensions [Ling 1993; Kågström et al. 1998a; 1998b]. One possibility is to decrease NB in order to decrease the number of cache misses in an attempt to fit the submatrix into the cache. Copying these blocks into a working array with a favorable leading dimension is also very useful (as used in the RISC BLAS). Note that avoiding powers of 2 as dimensions of a 2D array is not always sufficient to prevent a catastrophic behavior of caches [Bodin and Seznec 1994].

Similarly to Bell [1991] and Dongarra et al. [1991], NB is selected for the RISC BLAS in a simple way: it 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 multilevel cache machines is outwith the scope of this article, but some multilevel blocking is implemented in Version 2.0 of our software. Additionally in Version 2.0, the multiplication of C by β is effected within the sectioning loops so that the submatrix of C can be reused more efficiently. The way we manage critical leading dimensions will be improved in future releases of the RISC BLAS.

Since all the computational kernels call GEMM, the block size NB is always determined as the most appropriate block size for GEMM. That is, we choose the largest even integer (even—to enhance loop-unrolling by using the tuned codes such as DGEMML2X2 NN without needing additional tests to handle matrices with odd order) such that

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

Table I. Block Size Used in the Blocked BLAS on the Target Computers

Computer	Cache Characteristics		Block Size		Organization of Operations	Clock (MHz)	Peak Perf.
	Size	Org.	Single	Double			
CRAY T3D	8KB	Direct	24	16	TRIADIC	150	150
DEC 8400 5/300	96KB	3-way	88	60	TRIADIC	300	600
HP 715/64	64KB	Direct	72	52	TRIADIC	64	128
IBM SP2	64KB	4-way	72	52	TRIADIC	66	264
MEIKO CS2-HA	256KB	4-way	140	100	NOTRIADIC	100	100
SGI Power 10000	32KB	Direct	42	36	TRIADIC	195	390
SUN Ultra-1 140	16KB	Direct	36	24	NOTRIADIC	143	286

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

We show the block sizes used in our experiments in Table I. 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 has two levels of internal cache of size equal to 8KB and 96KB, respectively, and an external cache of between 1MB and 64MB. We have tuned our codes with respect to the second level of internal cache, since our experiments show this is the most efficient. The SGI Power Challenge the using R10000 processor and the UltraSPARC-1 have an external cache of size equal to 1MB and 512KB respectively.

Single-Precision Implementation on the IBM RS/6000 and IBM SP2. Slight modifications [Dongarra et al. 1991] allow further improvement in performance on the IBM Power and Power2 processors. The IBM FPU performs its arithmetic using 64-bit operands. As a consequence, these processors perform single-precision operations in the following way:

(1) Convert operands from single to double precision.

(2) Perform double-precision computation.

(3) Convert the double-precision result into single precision.

These conversions can be very costly and explain why the IBM is slower in single precision than in double precision. Therefore, we have slightly modified the tuned code SGEMML4X2 to convert operands within the innermost loop once only. The matrix A is copied into a double-precision working array in the blocked code, and the elements of arrays C and B are stored in double-precision temporary scalars.

We have used this data conversion only in SGEMM on the IBM, but it should be used everywhere else. Since IBM provides a tuned BLAS implementation in its scientific library, we have decided not to spend too much effort on tuning our code for the IBM.

Table II. Average Performance in Mflop/s of the Blocked Implementation of DGEMM and SGEMM on RISC Workstations (using square matrices of order 32, 64, 96, and 128)

		op(A), op(B)			
Processor	DGEMM/SGEMM	'N', 'N'	'N', 'T'	'T', 'N'	'T', 'T'
CRAY T3D	standard	/12	/12	/15	/8
	blocked	/49	/40	/49	/49
	library	/90	/66	/87	/71
DEC 8400 5/30	standard	95/105	98/108	76/77	61/65
	blocked	216/246	208/267	215/239	211/258
	library	335/412	327/388	345/416	318/395
HP 715/64	standard	15/18	16/18	20/22	22/24
	blocked	29/55	30/59	30/55	34/56
	library	52/81	47/63	46/71	51/81
IBM SP2 (thin node)	standard	31/32	32/32	51/27	51/27
	blocked	132/153	111/167	146/171	135/191
	library	189/206	182/197	197/220	161/202
MEIKO CS2-HA	standard	39/28	36/33	30/33	27/36
	blocked	38/60	45/58	39/69	43/78
SGI Power 10000	standard	79/91	77/95	110/12	90/108
	blocked	152/152	197/174	206/247	225/249
	blocked (64KB)	206/271	194/261	200/264	186/266
	library	136/152	168/173	198/247	209/249
SUN Ultra-1 140	standard	28/42	26/42	33/54	26/45
	blocked	67/112	64/116	67/112	63/118

3.2 Numerical Experiments

We show in Table II the performance achieved on CRAY T3D, DEC 8400 5/300, IBM SP2, HP 715, MEIKO CS2-HA, SGI Power Challenge, and SUN UltraSPARC-1. We also include the performance of the manufacturer-supplied library version when available (we use -lblas on the SGI and the HP, ESSL on the IBM SP2, SCILIB on the CRAY T3D, and -ldxml on the DEC 8400). We only include the single-precision experiments for the CRAY T3D, since single precision corresponds to 64-bit arithmetic on that machine (we proceed similarly for the other kernels). "Standard" in column 2 of Tables II, III, V, and VI refers to the standard Fortran version. The performance reported is the average performance achieved on a set of 4 matrix-matrix multiplications where matrices are square of order 32, 64, 96, and 128. We also report, on the SGI Power Challenge, the performance achieved using an increased block size corresponding to a cache size equal to 64KB instead of 32KB (line: blocked (64KB)).

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,

Table III. Average Performance in Mflop/s of the Blocked Implementation of DGEMM (SGEMM on CRAY T3D) 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)

		k	
Processor	GEMM (64-bit)	8	16
CRAY T3D	standard	15	15
	blocked	37	24
	library	73	44
HP 715/64	standard	17	16
	blocked	23	25
	library	38	42
IBM SP2 (thin node)	standard	33	17
	blocked	87	62
	library	85	78
MEIKO CS2-HA	standard	33	37
	blocked	32	35
SGI Power 10000	standard	91	49
	blocked	18	106
	library	18	108
SUN Ultra-1 140	standard	32	17
	blocked	51	31

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 multilevel cache. The vendor-supplied library GEMM routines on the SGI perform similarly in single precision and are slightly worse in double precision than our blocked code. However, if we increase the block size, we can improve the performance of the blocked codes by more than 60% in some cases even though the submatrices do not then fit in the on-chip caches (see SGI Power results).

We also report in Table III the average performance of DGEMM (SGEMM on the CRAY T3D) 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 and Duff 1989; Amestoy et al. 1995; Puglisi 1993]. We only consider the case where A and B are not transposed.

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

Table IV. Performance in Mflop/s of the Best Available Implementation of DGEMM and SGEMM on RISC Workstations (A and B are not transposed)

Processor	Version	Kernel	Size 500	Size 1000	Peak
CRAY T3D	library	SGEMM	102	103	150
DEC 8200 5/300	library	DGEMM	334	313	600
		SGEMM	431	418	
HP 715/64	library	DGEMM	35	35	128
		SGEMM	71	68	
IBM SP2 (thin node)	library	DGEMM	211	212	266
		SGEMM	232	234	
MEIKO CS2-HA	blocked	DGEMM	49	49	100
		SGEMM	88	88	
SGI Power 10000	blocked	DGEMM	233	231	388
		SGEMM	305	296	
SUN Ultra-1 140	blocked	DGEMM	66	64	286
		SGEMM	124	122	

The performance achieved by the tuned versions of GEMM is relatively far from the peak performance for all the RISC processors except the IBM SP2. On the IBM SP2, it is possible to reach peak performance by changing the leading dimensions of the matrices [Agarwal et al. 1994].

4. BLOCKED IMPLEMENTATION OF TRSM

TRSM solves one of the matrix equations

$$AX = \alpha B, \quad A^T X = \alpha B, \quad XA = \alpha B, \quad \text{or} \quad XA^T = \alpha B$$

where α is a scalar; X and B are $m \times n$ matrices; and A is a unit, or nonunit, 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 $AX = \alpha B$ where A is not transposed, and upper triangular):

$$\begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} = \alpha \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

(1) Solution of $A_{22}X_{21} = \alpha B_{21}$ and B_{21} is overwritten by X_{21} (TRSM)

(2) Solution of $A_{22}X_{22} = \alpha B_{22}$ and B_{22} is overwritten by X_{22} (TRSM)

(3) $B_{11} \leftarrow \alpha B_{11} - A_{12}B_{21}$ (GEMM)

(4) $B_{12} \leftarrow \alpha B_{12} - A_{12}B_{22}$ (GEMM)

(5) Solution of $A_{11}X_{11} = B_{11}$ and B_{11} is overwritten by X_{11} (TRSM)

(6) Solution of $A_{11}X_{12} = B_{12}$ and B_{12} is overwritten by X_{12} (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.

As soon as the matrices are large enough compared with the block size, GEMM represents a high percentage of the total number of floating-point operations required by the blocked version of TRSM. For example, when the triangular system is "Left," assuming that NB divides m and n exactly (so that the number of blocks is $m/NB \times n/NB$), the percentage of operations spent in GEMM is equal to $1 - NB/m$. Note that the blocked versions of the other kernels behave similarly.

We report, in Table V, the performance achieved on our range of RISC workstations for all variants when A is unit and the system is left (the performance is similar when A is nonunit and when the system is right). 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. In both single and double precision, our blocked code outperforms the vendor code on the DEC 8400, and would be even faster if we used calls to the vendor-supplied GEMM routines from within our blocked code.

5. BLOCKED IMPLEMENTATION OF TRMM

TRMM performs one of the matrix-matrix operations

$$B = \alpha AB, \quad B = \alpha A^T B$$

or

$$B = \alpha BA, \quad B = \alpha BA^T$$

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

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

$$\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \leftarrow \alpha \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

(1) $B_{11} \leftarrow \alpha A_{11} B_{11}$ (TRMM)

(2) $B_{11} \leftarrow B_{11} + \alpha A_{12} B_{21}$ (GEMM)

(3) $B_{12} \leftarrow \alpha A_{11} B_{12}$ (TRMM)

(4) $B_{12} \leftarrow B_{12} + \alpha A_{12} B_{22}$ (GEMM)

(5) $B_{21} \leftarrow \alpha A_{22} B_{21}$ (TRMM)

(6) $B_{22} \leftarrow \alpha A_{22} B_{22}$ (TRMM)

Table V. Average Performance in Mflop/s of the Blocked Implementations of DTRSM and DTRMM (STRSM and STRMM on the CRAY T3D) on RISC Processors (using square matrices of order 32, 64, 96, and 128)

		Kernel							
		TRSM: "Left"				TRMM: "Left"			
Processor	Version	'U' 'N'	'L' 'N'	'U 'T	'L' 'T'	'U' 'N'	'L' 'N'	'U 'T	'L' 'T'
CRAY T3D	standard	12	12	16	16	12	12	13	13
	blocked	38	38	35	34	47	48	36	36
	library	87	86	82	81	12	12	16	16
DEC 8400 5/300	standard	40	73	70	72	84	81	74	70
	blocked	204	187	199	231	194	180	183	184
	library	184	182	176	183	175	174	178	175
HP 715/64	standard	12	12	20	19	16	15	21	20
	blocked	30	28	31	31	30	29	33	31
	library	33	34	34	31	41	41	41	39
IBM SP2 (thin node)	standard	41	45	53	53	31	29	54	55
	blocked	113	140	109	123	106	121	116	109
	library	155	157	146	147	154	174	155	166
MEIKO CS2-HA	standard	13	12	31	30	13	12	31	30
	blocked	50	47	43	42	50	47	43	42
SGI Power 10000	standard	67	82	118	116	77	74	117	116
	blocked	212	212	261	254	228	224	242	178
	library	211	211	259	252	228	224	242	178
SUN Ultra-1 140	standard	22	22	34	33	24	25	34	34
	blocked	57	57	57	57	64	64	64	57

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 TRMML2X2 for performing the multiplication of diagonal blocks of A.

We report in Table V the performance of the blocked version of TRMM in the case where A is unit and when the system is left. Performance is similar when the system is right. Our blocked code can be seen to be usually more than twice as efficient as standard BLAS. Larger blocking on the SGI does not help much on this kernel except in single precision. 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. TRMM is obviously not tuned at all in the SCILIB library that we have access to on the CRAY T3D.

6. BLOCKED IMPLEMENTATION OF SYMM

SYMM performs one of the following matrix-matrix operations

$$C = \alpha AB + \beta C$$

or

$$C = \alpha BA + \beta C$$

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

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

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \leftarrow \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} + \beta \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

(1) $C_{11} \leftarrow \beta C_{11} + \alpha A_{11} B_{11}$ (SYMM)

(2) $C_{12} \leftarrow \beta C_{12} + \alpha A_{11} B_{12}$ (SYMM)

(3) $C_{11} \leftarrow C_{11} + \alpha A_{12} B_{21}$ (GEMM)

(4) $C_{12} \leftarrow C_{12} + \alpha A_{12} B_{22}$ (GEMM)

(5) $C_{21} \leftarrow \beta C_{21} + \alpha A_{22} B_{21}$ (SYMM)

(6) $C_{22} \leftarrow \beta C_{22} + \alpha A_{22} B_{22}$ (SYMM)

(7) $C_{21} \leftarrow C_{21} + \alpha A_{12}^T B_{11}$ (GEMM)

(8) $C_{22} \leftarrow C_{22} + \alpha A_{12}^T B_{12}$ (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_{ii} (only the upper triangular part is stored, since the submatrices are symmetric). A straightforward way of avoiding the multiplication step involving these symmetric submatrices consists in copying the submatrices A_{ii} into a working array AA where both the upper and the lower triangular part are stored as described in Ling [1993]. Therefore, instead of using a SYMM operation for multiplications using the submatrices A_{ii}, 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.

Using this strategy, SYMM is expressed as a sequence of GEMM operations:

(1) Copy A_{11} into AA

(2) $C_{11} \leftarrow \beta C_{11} + \alpha AA.B_{11}$ (GEMM)

(3) $C_{12} \leftarrow \beta C_{12} + \alpha AA.B_{12}$ (GEMM)

(4) $C_{11} \leftarrow C_{11} + \alpha A_{12}B_{21}$ (GEMM)

(5) $C_{12} \leftarrow C_{12} + \alpha A_{12}B_{22}$ (GEMM)

(6) Copy A_{22} into AA

(7) $C_{21} \leftarrow \beta C_{21} + \alpha AA.B_{21}$ (GEMM)

(8) $C_{22} \leftarrow \beta C_{22} + \alpha AA.B_{22}$ (GEMM)

(9) $C_{21} \leftarrow C_{21} + \alpha A_{12}^T B_{11}$ (GEMM)

(10) $C_{22} \leftarrow C_{22} + \alpha A_{12}^T B_{12}$ (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.

We present in Table VI the performance of the blocked version of SYMM, and we compare it with 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 2 but occasionally nearly a factor of 8. On the SGI, our blocked code and the manufacturer-supplied version perform similarly. It seems clear that DEC has used a similar device to ours on the DEC 8400, since the performance of their library code for SYMM is close to their GEMM performance. SYMM is obviously not tuned at all in the SCILIB library that we have access to on the CRAY T3D.

7. BLOCKED IMPLEMENTATION OF SYRK

SYRK performs one of the following symmetric rank-k operations

$$C = \alpha AA^T + \beta C$$

or

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

where α and β are scalars; C is an $n \times n$ symmetric matrix (only the upper or lower triangular parts are updated); and 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 $C = \alpha AA^T + \beta C$ where only the upper triangular part of C is updated):

$$\begin{pmatrix} C_{11} & C_{12} \\ 0 & C_{22} \end{pmatrix} \leftarrow \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{pmatrix} + \beta \begin{pmatrix} C_{11} & C_{12} \\ 0 & C_{22} \end{pmatrix}$$

(1) $C_{11} \leftarrow \beta C_{11} + \alpha A_{11}A_{11}^T$ (SYRK)

(2) $C_{11} \leftarrow C_{11} + \alpha A_{12}A_{12}^T$ (SYRK)

Table VI. Average Performance in Mflop/s of the Blocked Implementations of DSYMM, DSYRK, and DSYR2K (SSYMM, SSYRK, and SSYR2K on the CRAY T3D) for RISC Processors (using square matrices of order 32, 64, 96, and 128)

		SYMM				SYRK				SYR2K			
Processor	Version	'L' 'U'	'L' 'L'	'R 'U'	'R' 'L'	'U' 'N'	'L' 'N'	'U' 'T'	'L' 'T'	'U' 'N'	'L' 'N'	'U 'T	'L' 'T'
CRAY T3D	standard	4	7	11	11	11	11	16	16	7	7	5	5
	blocked	45	45	48	48	39	38	44	42	42	42	47	48
	library	4	7	11	11	11	11	16	16	7	7	5	5
DEC 840	standard	87	96	91	90	81	82	74	78	83	89	82	82
5/300	blocked	188	183	180	183	172	162	182	189	187	181	191	198
	library	315	310	300	300	82	86	100	85	106	103	130	127
HP 715/64	standard	4	4	16	16	17	15	24	24	5	5	2	2
	blocked	28	28	30	31	26	23	32	31	33	31	34	35
	library	44	45	45	46	16	16	28	27	5	5	3	3
IBM SP2	standard	52	51	34	34	31	30	60	60	42	41	81	89
(thin node)	blocked	116	114	113	121	106	99	115	109	113	103	123	128
	library	156	165	141	168	169	179	177	174	146	161	169	177
MEIKO	standard	15	15	40	38	20	20	37	36	15	17	10	10
CS2-HA	blocked	37	36	40	42	44	44	40	40	44	44	46	46
SGI Power	standard	44	37	98	97	73	70	139	137	40	37	40	39
10000	blocked	241	237	226	220	192	182	192	184	208	204	219	215
	library	241	237	220	214	198	198	199	197	210	216	254	251
SUN Ultra-1	standard	26	25	26	26	24	24	36	36	29	29	26	26
140	blocked	61	61	62	62	61	63	69	70	62	61	64	65

(3) $C_{12} \leftarrow \beta C_{12} + \alpha A_{11} A_{21}^T$ (GEMM)

(4) $C_{12} \leftarrow C_{12} + \alpha A_{12} A_{22}^T$ (GEMM)

(5) $C_{22} \leftarrow \beta C_{22} + \alpha A_{21} A_{21}^T$ (SYRK)

(6) $C_{22} \leftarrow C_{22} + \alpha A_{22} A_{22}^T$ (SYRK)

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

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. In Table VI 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 2. On the SGI, the vendor code and the blocked version perform similarly. Using a larger block size on the SGI improves perfor-

mance by up to 40% in double 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. The CRAY SCILIB code is obviously not tuned.

8. BLOCKED IMPLEMENTATION OF SYR2K

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

$$C = \alpha AB^T + \alpha BA^T + \beta C$$

or

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

where α and β are scalars; C is an $n \times n$ symmetric matrix (only the upper or lower triangular parts are updated); and A and 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., $C = \alpha AB^T + \alpha BA^T + \beta C$ where only the upper triangular part of C is updated):

$$\begin{pmatrix} C_{11} & C_{12} \\ 0 & C_{22} \end{pmatrix} \leftarrow \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11}^T & B_{21}^T \\ B_{12}^T & B_{22}^T \end{pmatrix}$$

$$+ \alpha \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \begin{pmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{pmatrix} + \beta \begin{pmatrix} C_{11} & C_{12} \\ 0 & C_{22} \end{pmatrix}$$

(1) $C_{11} \leftarrow \beta C_{11} + \alpha A_{11} B_{11}^T + \alpha B_{11} A_{11}^T$ (SYR2K)

(2) $C_{11} \leftarrow C_{11} + \alpha A_{12} B_{12}^T + \alpha B_{12} A_{12}^T$ (SYR2K)

(3) $C_{12} \leftarrow \beta C_{12} + \alpha A_{11} B_{21}^T$ (GEMM)

(4) $C_{12} \leftarrow C_{12} + \alpha B_{11} A_{21}^T$ (GEMM)

(5) $C_{12} \leftarrow C_{12} + \alpha A_{12} B_{22}^T$ (GEMM)

(6) $C_{12} \leftarrow C_{12} + \alpha B_{12} A_{22}^T$ (GEMM)

(7) $C_{22} \leftarrow \beta C_{22} + \alpha A_{21} B_{21}^T + \alpha B_{21} A_{21}^T$ (SYR2K)

(8) $C_{22} \leftarrow C_{22} + \alpha A_{22} B_{22}^T + \alpha B_{22} A_{22}^T$ (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 C can be effected simultaneously. There is no need to compute both αAB^T and αBA^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 as described in Ling [1993].

Using these remarks SYR2K is computed in the following way:

(1) $CC \leftarrow \alpha A_{11}.B_{11}^T$ (GEMM)

(2) $CC \leftarrow CC + \alpha A_{12}.B_{12}^T$ (GEMM)

(3) $C_{11} \leftarrow \beta C_{11} + CC + CC^T$

(4) $C_{12} \leftarrow \beta C_{12} + \alpha A_{11} B_{21}^T$ (GEMM)

(5) $C_{12} \leftarrow C_{12} + \alpha B_{11} A_{21}^T$ (GEMM)

(6) $C_{12} \leftarrow C_{12} + \alpha A_{12} B_{22}^T$ (GEMM)

(7) $C_{12} \leftarrow C_{12} + \alpha B_{12} A_{22}^T$ (GEMM)

(8) $CC \leftarrow \alpha A_{21}.B_{21}^T$ (GEMM)

(9) $CC \leftarrow CC + \alpha A_{22}.B_{22}^T$ (GEMM)

(10) $C_{22} \leftarrow \beta C_{22} + CC + CC^T$

As for SYRK, with a larger number of blocks, the GEMM updates of the off-diagonal blocks can be combined. We report, in Table VI, the results obtained using our blocked implementation of SYR2K. 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. SYR2K is not tuned on the CRAY T3D.

9. USE OF THE MANUFACTURER-SUPPLIED GEMM

In Table VII, we show the effect of using a tuned version of GEMM—the manufacturer-supplied version—within our RISC BLAS. We only show the performance of one of the variants for each Level 3 BLAS kernel. It is compared with the manufacturer-supplied library kernel.

The performance of the RISC BLAS is substantially increased when using the tuned vendor code for GEMM within our blocked version. We achieve better performance than the manufacturer-supplied library on the HP except for SYMM and TRMM. On the IBM SP2, we are still far from the ESSL performance on TRSM, TRMM, and SYRK, while we outperform the vendor code for SYMM and the single-precision SYR2K. On the CRAY T3D, except for TRSM which is tuned in the manufacturer-supplied library, we outperform the vendor codes.

10. CONCLUSION

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-

Table VII. Comparison of the Average Performance in Mflop/s of the RISC BLAS Using the Manufacturer-Supplied GEMM with the Manufacturer-Supplied Library (using square matrices of order 32, 64, 96, and 128)

Computer	Kernel	Variant	Version	64 Bits	32 Bits
CRAY T3D	TRSM	"Left," "Upper," "No transpose," "Unit"	blocked	57	—
			SCILIB	87	—
	SYMM	"Left," "Upper"	blocked	87	—
			SCILIB	4	—
	TRMM	"Left," "Upper," "No transpose," "Unit"	blocked	54	—
			SCILIB	12	—
	SYRK	"Upper," "No transpose"	blocked	42	—
			SCILIB	11	—
	SYR2K	"Upper," "No transpose"	blocked	58	—
			SCILIB	7	—
HP 715/64	TRSM	"Left," "Upper," "No transpose," "Unit"	blocked	41	54
			-lblas	35	53
	SYMM	"Left," "Upper"	blocked	40	69
			-lblas	46	79
	TRMM	"Left," "Upper," "No transpose," "Unit"	blocked	35	54
			-lblas	43	64
	SYRK	"Upper," "No transpose"	blocked	30	58
			-lblas	16	20
	SYR2K	"Upper," "No transpose"	blocked	41	68
			-lblas	5	5
IBM SP2	TRSM	"Left," "Upper," "No tranpose," "Unit"	blocked	114	115
			ESSL	155	223
	SYMM	"Left," "Upper"	blocked	175	209
			ESSL	156	193
	TRMM	"Left," "Upper," "No transpose," "Unit"	blocked	129	127
			ESSL	154	178
	SYRK	"Upper," "No transpose"	blocked	117	90
			ESSL	169	185
	SYR2K	"Upper," "No transpose"	blocked	163	175
			ESSL	146	192

unrolling depth are machine dependent. Both the performance of GEMM and the 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 well with the manufacturer-supplied libraries on the IBM SP2, 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 CRAY T3D, many of our blocked versions of the other kernels are better than the vendor-supplied equivalents, sometimes by a large margin. We have also demonstrated that the availability of a highly tuned version of the matrix-matrix multiplication kernel GEMM improves 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. It is the same for DSYMM on the IBM SP2. We suggest that some vendors could easily increase the performance of their non-GEMM Level 3 BLAS kernels by using the techniques described in this article. Finally, for some machines, performance could be enhanced by judiciously selecting appropriate leading dimensions of the matrices (e.g., avoiding powers of 2), 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 and 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.

11. AVAILABILITY OF CODES

The codes described in the present article are available using anonymous FTP 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.

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.

We acknowledge the support of CNUSC and IDRIS for providing accesses to the IBM SP2 and the CRAY T3D respectively.

REFERENCES

AGARWAL, R. C., GUSTAVSON, F. G., AND ZUBAIR, M. 1994. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM J. Res. Dev. 38*, 5 (Sept. 1994), 563–576.

AMESTOY, P. R. AND DAYDÉ, M. J. 1993. Tuned block implementation of Level 3 BLAS for the CONVEX C220 and RISC processors. Distributed implementation of LU factorization using PVM. Tech. Rep. RT/APO/93/3 Toulouse, France.

AMESTOY, P. R. AND DUFF, I. S. 1989. Vectorization of a multiprocessor multifrontal code. *Int. J. Supercomput. Appl. High Perform. Eng. 3*, 3, 41–59.

AMESTOY, P. R., DAYDÉ, M. J., DUFF, I. S., AND MORÈRE, P. 1995. Linear algebra calculations on a virtual shared memory computer. *Int. J. High Speed Comput. 7*, 1, 21–43.

ANDERSON, E., BAI, Z., BISCHOF, C., DEMMEL, J., DONGARRA, J., DUCROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., OSTROUCHOV, S., AND SORENSEN, D. 1995. *LAPACK Users' Guide*. 2nd ed. SIAM, Philadelphia, PA.

BELL, R. 1991. IBM RISC System/6000 NIC tuning guide for Fortran and C. Tech. Rep. GG24-3611-01, IBM International Technical Support Centers.

BERGER, P. AND DAYDÉ, M. J. 1991. Implementation and use of Level 3 BLAS kernels on a Transputer T800 ring network. Tech. Rep. TR/PA/91/54, CERFACS.

BODIN, F. AND SEZNEC, A. 1994. Cache organization influence on loop blocking. Tech. Rep. 803, IRISA, Rennes, France.

DAYDÉ, M. J. AND DUFF, I. S. 1989. Use of Level 3 BLAS in LU factorization on the CRAY-2, the ETA 10-P, and the IBM 3090 VF. *Int. J. Supercomput. Appl. High Perform. Eng. 3*, 2, 40–70.

DAYDÉ, M. J. AND DUFF, I. S. 1995. Porting industrial codes and developing sparse linear solvers on parallel computers. *Comput. Syst. Eng. 4*, 5, 295–305.

DAYDÉ, M. J. AND DUFF, I. S. 1996. A blocked implementation of Level 3 BLAS for RISC processors. Tech. Rep. RAL-TR-96-014. Rutherford Appleton Lab., Didcot, Oxon, United Kingdom. Also ENSEEIHT-IRIT Tech. Rep. RT/APO/96/1 and CERFACS Rep. TR/PA/96/06.

DAYDÉ, M. J. AND DUFF, I. S. 1997. A block implementation of Level 3 BLAS for RISC processors, revised version. Tech. Rep. RT/APO/97/2, ENSEEIHT-IRIT.

DAYDÉ, M. J., DUFF, I. S., AND PETITET, A. 1994. A parallel block implementation of Level-3 BLAS for MIMD vector processors. *ACM Trans. Math. Softw. 20*, 2 (June 1994), 178–193.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND DUFF, I. 1990a. Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Softw. 16*, 1 (Mar. 1990), 18–28.

DONGARRA, J. J., DU CROZ, J. J., HAMMARLING, S., AND DUFF, I. S. 1990b. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1 (Mar. 1990), 1–17.

DONGARRA, J. J., MAYES, P., AND RADICATI DI BROZOLO, G. 1991. LAPACK Working Note 28: The IBM RISC System/6000 and linear algebra operations. Tech. Rep. CS-91-130. Department of Computer Science, University of Tennessee, Knoxville, TN.

GALLIVAN, K., JALBY, W., AND MEIER, U. 1987. The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. *SIAM J. Sci. Stat. Comput. 8*, 6 (Nov. 1, 1987), 1079–1084.

GALLIVAN, K., JALBY, W., MEIER, U., AND SAMEH, A. H. 1988. Impact of hierarchical memory systems on linear algebra algorithm design. *Int. J. Supercomput. Appl. High Perform. Eng. 2*, 1, 12–48.

HENNESSY, J. L. AND PATTERSON, D. A. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

KÅGSTRÖM, B. AND VAN LOAN, C. 1998. Algorithm 784: GEMM-based level 3 BLAS: portability and optimization issues. *ACM Trans. Math. Softw. 24*, 3, 303–316.

KÅGSTRÖM, B., LING, P., AND VAN LOAN, C. 1998. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw. 24*, 3, 268–302.

LING, P. 1993. A set of high-performance Level 3 BLAS structured and tuned for the IBM 3090 VF and implemented in Fortran 77. *J. Supercomput. 7*, 3 (Sept. 1993), 323–355.

PUGLISI, C. 1993. *QR* Factorization of large sparse overdetermined and square matrices using the multifrontal method in a multiprocessor environment. Ph.D. Dissertation.

QRICHI ANIBA, A. 1994. Implémentation performante du BLAS de niveau 3 pour les processeurs RISC. Tech. Rep. Rapport 3ème Année. Département Informatique et Mathématiques Appliquées, ENSEEIHT, Toulouse, France.

SHEIKH, Q. AND LIU, J. 1989. Basic linear algebra subprogram optimization on the CRAY-2 system. CRAY Channels Spring.

Received: December 1996; revised: March 1998, February 1999, and April 1999; accepted: April 1999