The IBM RISC System/6000 and Linear Algebra Operations

Jack J. Dongarra

and

Peter Mayes

and

Giuseppe Radicati di Brozolo

January 17, 1991

Electronic mail addresses: dongarra@cs.utk.edu, na.mayes@na-net.ornl.gov, and RADICATI@IECSEC.EARN

This work was supported in part by IBM and the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

The IBM RISC System/6000 and Linear Algebra Operations *

Jack J. Dongarra
Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301; and Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831

Peter Mayes

NAG Ltd., Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, UNITED KINGDOM
and

Giuseppe Radicati di Brozolo

IBM European Center for Scientific and Engineering Computing, 00147 Roma, via Giorgione 159, ITALY

January 17, 1991

Abstract

This paper discusses the IBM RISC System/6000 workstation and a set of experiments with blocked algorithms commonly used in solving problems in numerical linear algebra. We describe the performance of these algorithms and discuss the techniques used in achieving high performance on such an architecture.

1 IBM RISC System/6000: System Overview

The IBM RISC System/6000 computer is a superscalar second-generation RISC architecture [2]. It is the result of advances in compiler and architecture technology that have evolved since the late 1970s and early 1980s.

Like other RISC processors, the RISC System/6000 implements a register-oriented instruction set, the CPU is hardwired rather than microcoded, and it features a pipelined implementation. The floating-point unit is integrated in the CPU, minimizing the overhead associated with separate floating-point coprocessors.

Unlike other RISC processors, however, the RISC System/6000 has the ability to dispatch

*This work was supported in part by IBM and the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

multiple instructions and to overlap the execution of the fixed-point, the floating-point, and the branch functional units. The 184 instructions are divided among the functional units and are designed to minimize interaction among the functional units.

The IBM RISC System/6000 is intended to satisfy the requirements of both commercial and scientific applications. Our focus here is on the performance of the RISC System/6000 for scientific applications, which require very high floating-point performance as well as specialized peripherals, such as high-quality graphics adapters. These, in turn, require very high memory bandwidths to the central processing unit.

In what follows, we give a brief overview of the design of the CPU and memory and of some aspects of the I/O system. In particular, we discuss those architectural features most important for designing and implementing high-performance mathematical software. Note that specific details refer to the Model 530. The specification of other members of the RISC System/6000 family may be different in some aspects. For a more complete discussion of the hardware, we refer the interested reader to the January 1990 issue of the *IBM Journal of Research and Development.*

1.1 Central Processing Unit

The CPU architecture is based on a design that exploits modern compiler technology, and an implementation that exploits VLSI and CMOS technology, to allow as much parallel instruction execution as possible. The RISC System/6000 processor consists of three separate but integrated functional units:

1. The Instruction Cache and Branch Processing Unit feeds a stream of instructions to the fixed-point and floating-point units. The branch processor provides all the branching, interrupt, and condition code functions within the system. An important feature of the branch processing unit is the "zero-cycle branch." A zero-cycle branch is achieved by

executing branches simultaneously with fixed- or floating-point operations so that the stream of data to these units is not interrupted. In practice, this configuration means that loop boundaries do not interrupt pipelining.

2. The Fixed-Point Unit (FXU) is designed to execute the fixed-point arithmetic, the logic instructions, and the data address computations and to schedule the movement of data between the floating-point unit and the data cache. Read or write transfers between the floating-point unit and the data cache require one cycle to complete.

3. The Floating-Point Unit (FPU) supports the execution of the floating-point instructions. The FPU has a set of thirty-two 64-bit floating-point registers that access the data cache directly. It conforms to the ANSI/IEEE 745-1985 standard for binary floating-point arithmetic. The FPU is organized for double-precision computations. Thus, data held in the floating-point registers are always represented in double-precision format. Therefore, when single-precision data are loaded, they are expanded to double-precision format.

In addition, there are a number of features in the architecture which enhance performance.

- Register renaming is an important feature of the machine. This allows data for the next instruction to be loaded into a floating-point register that is currently being used by an earlier instruction.

- In addition to the usual arithmetic operations, there are compound instructions that multiply two operands and add (or subtract) the product to a third operand. These floating-point multiply-and-add (FMA) instructions take two cycles to complete. However, one FMA instruction may be issued in each clock cycle, provided that the operands are independent. Thus it is possible to complete two floating-point computations in each cycle.

- The FMA instructions actually produce only one rounding error rather than two and are therefore more accurate than required by the IEEE standard. This additional accuracy has been used, for example, in some of the math intrinsic functions. However, if strict adherence to the IEEE standard is required, a compile-time option can be used to disable the generation of compound instructions.

- The floating-point divide is implemented by a Newton-Raphson approximation algorithm. A division requires 16 to 19 cycles to complete and provides correctly rounded results, but is obviously expensive if computed unnecessarily inside a loop. If division by a constant is taken out of the loop and replaced by a multiplication with the reciprocal, the code is more efficient, but the results are not necessarily identical. If it is important to have exactly equivalent code, the added precision and speed of the multiply-add instruction can be used to implement a reciprocal multiplication plus correction algorithm at the cost of a multiply and two multiply-adds (5 cycles). This algorithm is cheaper than a division and still provides correctly rounded results.

This design has allowed the implementation of a CPU that executes up to four instruction per cycle: one branch instruction, one condition register instruction, one fixed-point instruction, and one floating-point multiply-add instruction. A second pipelined instruction can begin on the next cycle on an independent set of operands. This means that two independent floating-point operations per cycle can be executed.

Of particular interest is the fact that loads and independent floating-point operations can occur in parallel. The compiler takes advantage of this capability in many cases: with a well-designed algorithm, it is possible to execute two floating-point operations on separate data items and "hide" one memory reference all in the same cycle. At a clock cycle of 25 MHz, this translates into a peak performance of 50 million floating-point operations per second (Mflops).

1.2 Memory and Caches

The RISC System/6000 memory banks implement a four-way interleaved design that provides two words (two 64-bit words) of data every machine cycle. A system can have from 16 to

256 Mbytes of total memory.

Separate instruction and data caches provide conflict-free access to data and instructions. The instruction cache is organized as an 8-Kbyte, two-way set-associative cache, which has a 64-byte (16-instruction) line size. The data cache is a four-way set-associative 64-Kbyte cache, which is divided into four identical chips of 16 Kbytes each. The cache is implemented as a store-back cache to minimize the memory bus traffic: data are written back to memory only when an updated line in cache is replaced. The cache-line size is 128 bytes. A synchronous 128-bit memory bus allows 400 Mbytes per second to be transferred to or from memory: it takes eight cycles to load a cache line (16 double-precision words) from memory to cache. A 64-bit data bus connects the floating-point unit and the data cache: it takes one cycle to transfer a double-precision word between the data cache and the floating-point registers.

1.3 Serial Optical Link

The I/O unit contains an I/O channel controller and two serial link adapters, which provide an interface to optics cards that drive fiber-optics links. It is intended for attachment of disks, graphics adapters, and other high-speed peripherals. (Support for this high-speed optical link is planned for future release.) The serial optical link has a bandwidth of 220 Mbits per second, and it allows the attachment of remote devices up to 2000 meters away. The link is also suitable for interprocessor message and data transfers in a multiprocessor configuration, and work is under way to investigate its suitability for closely coupled multiprocessing.

2 Fortran Techniques for Performance on Matrix Operations

As mentioned in the preceding section, the RISC System/6000 can complete a floating-point multiply-and-add (FMA) instruction every cycle, so that a Model 530 running at 25 MHz has a theoretical peak speed of 50 Mflops. Many factors limit the amount of concurrency that can be effectively used, thus limiting the performance that an algorithm can achieve. Most notably, unnecessary memory references can have a severe impact on the performance attainable. Indeed, the movement of data between memory and registers can be more costly than arithmetic operations on the data. This cost provides considerable motivation to restructure existing algorithms and to devise new algorithms that minimize data movement.

In this section we describe a model to predict the performance of simple Fortran loops and to serve as a guide to writing efficient Fortran code for the RISC System/6000. (We observe that the Fortran compiler usually takes advantage of all the parallelism of which the CPU is capable.) Our model is based on the following rules:

1. Each FMA instruction requires two cycles to complete. Two FMAs that operate on independent data will be scheduled on consecutive cycles, and therefore two floating-point operations will be executed simultaneously.

2. Loads from cache to floating-point registers require one cycle to complete. They will be overlapped with FMAs that were scheduled earlier, even if they operate on registers that the earlier FMA is still using (register renaming).

3. Stores do not overlap with FMAs.

4. Loop boundaries do not interrupt pipelining (zero-cycle branch).

5. When a cache miss occurs, the floating-point unit must wait 11 cycles before the whole cache line is available. The latency from memory to cache accounts for 8 cycles. In our model we add to this an additional latency of 3 cycles, which fits closely the experimental data we collected. The details of the data transfer may be more complicated in reality, but this is the average effect that a Fortran programmer might expect to see.

In the following three subsections we use this model to explain the different levels of performance that can be achieved by using different levels of Basic Linear Algebra Subprograms (BLAS) kernels [8, 5, 4], and we describe some Fortran techniques to implement the BLAS efficiently. High performance was achieved by constructing the codes in such a way that the compiler can easily generate code that matches the architecture of the machine. The techniques used were blocking (or strip-mining), loop unrolling, and loop jamming—all fairly standard techniques used by compiler writers. We hope that some of these techniques will be incorporated into subsequent versions of the compiler, so that even less work will be required to exploit the machine.

2.1 Level 1 BLAS

The two Level 1 BLAS operations that occur most frequently in linear algebra are the DOT:

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

and the AXPY:

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

We begin by examining the performance of these operations when using data stored in cache.

For the DOT operation, each FMA instruction requires two loads, one for `X(I)` and for `Y(I)`. Loading the data requires two cycles, and performing the FMA also requires two cycles. There is no possibility of re-using data, so the best we can expect is that the loading of the next two operands is overlapped with an FMA. This corresponds to a theoretical speed of 25 Mflops; in practice, we measured 24.5 Mflops (see Table 1).

For the AXPY operation, each FMA instruction requires two loads and one store. Again, there is no possibility of reusing data, so the best we can hope for in this case is one FMA instruction

Table 1: Speed in Mflops of Level 1 BLAS

	DOT		AXPY	
Type of memory access	predicted	measured	predicted	measured
all data in cache	25	24.5	16.67	16.4
all data from memory:				
x and y with unit stride	14.81	14.6	11.43	11.3
x with stride 16	3.65	3.2	3.40	3.2
x and y with stride 16	2.08	1.8	2	1.4

every three cycles. This corresponds to 16.67 Mflops; in practice, we measured 16.4 Mflops (see Table 1).

For data that must be accessed from memory, we must take account of the time taken for data to arrive in the registers. Each time a cache miss occurs (every 16 elements for stride-one access), the processing is interrupted, and the CPU must wait for the cache line to become available. In our model, the CPU must wait for 11 machine cycles. Thus, the cost of moving contiguous data from memory to registers is, on the average, 1.69 cycles per element (i.e., 11 cycles to move a cache line from memory to cache plus 1 cycle to transfer each of the 16 elements from cache to register ((11+16)/16) cycles per element). In a DOT operation, on the average 2 floating-point operations (1 FMA) are scheduled every 3.38 machine cycles, giving 14.81 Mflops in theory and 14.6 Mflops in practice. If the vectors are accessed with stride 16 (the length of a cache line), each element will be available after a delay of 12 cycles (= 11 + 1), giving 25/12 = 2.08 Mflops in theory and 1.8 Mflops in practice. Table 1 shows the measured performance and the prediction using the model for some other memory access patterns.

2.2 Level 2 BLAS

Here we consider the Level 2 BLAS DGEMV operations

$$y \leftarrow y + Ax \qquad \text{and} \qquad y \leftarrow y + A^T x.$$

The basic operation in Fortran is given in Figure 1, where `A(I,J)` must be replaced by `A(J,I)`

```
DO
   DO
      Y(I) = Y(I) + A(I,J)*X(J)
   CONTINUE
CONTINUE
```

Figure 1: Generic matrix-vector multiply code

for the operation with A^T. Depending on the ordering of the DO-loops, the inner loop is either a DOT or an AXPY. We have seen from the discussion of the two Level 1 BLAS operations that, because the RISC System/6000 system can perform an FMA instruction with all its operands in registers, it is better suited to DOT operations than to AXPY operations. (Note that this contrasts with the situation on vector machines such as the CRAY Y-MP, where the two vector loads and one vector store required match the architecture well. Also, by unrolling, it is possible to keep the vector `Y(I)` in a vector register for longer, thus increasing the ratio of floating-point operations to memory references.) We have also seen from Table 1 that when accessing data from memory, it is very important to access the data with stride one, so that all the elements in a cache line are used when that line is loaded. For these two reasons, we consider the operation

$$y \leftarrow y + A^T x,$$

which can be expressed as a DOT operation with A accessed with unit stride.

For this operation, the peak speed is again 25 Mflops—exactly the same as for the DOT. However, in this case we can unroll the dot product to re-use each `X(J)` a number of times. As the depth of unrolling increases, the ratio of operations to loads increases from one and tends towards two. For example, for unrolling to depths 2, 3, and 4, the ratio of operations to loads is 4/3, 6/4, and 8/5, with a theoretical peak speed of 33.3, 37.5, and 40 Mflops, respectively. The code for this operation unrolled to depth 4 is shown in Figure 2. In practice, there is little benefit in unrolling to very large depths, as there are only a finite number of floating-point registers, and the performance reaches a plateau. The code in Figure 2 performs at 36.3 Mflops, and a speed of 40.3 Mflops has been measured for unrolling to depth 8.

```
      DO 20 I = 1, M, 4
         TEMP1 = ZERO
         TEMP2 = ZERO
         TEMP3 = ZERO
         TEMP4 = ZERO
         DO 10 J = 1, N
            TEMP1 = TEMP1 + A(J,I  )*X(J)
            TEMP2 = TEMP2 + A(J,I+1)*X(J)
            TEMP3 = TEMP3 + A(J,I+2)*X(J)
            TEMP4 = TEMP4 + A(J,I+3)*X(J)
 10      CONTINUE
         Y(I  ) = Y(I  ) + TEMP1
         Y(I+1) = Y(I+1) + TEMP2
         Y(I+2) = Y(I+2) + TEMP3
         Y(I+3) = Y(I+3) + TEMP4
 20   CONTINUE
```

Figure 2: Model Code for $y \leftarrow y + A^T x$

Table 2: Speed in Mflops of Level 2 BLAS

	Data in Cache				Data in Memory			
	$y \leftarrow y + Ax$		$y \leftarrow y + A^T x$		$y \leftarrow y + Ax$		$y \leftarrow y + A^T x$	
depth	DOT	AXPY	DOT	AXPY	DOT	AXPY	DOT	AXPY
1	22.7	15.6	22.6	15.5	8.7	9.0	11.0	7.7
2	30.4	23.5	30.4	23.4	10.4	10.0	11.2	9.5
3	34.1	24.0	34.2	23.8	10.6	12.3	11.4	9.7
4	36.3	24.0	36.4	23.6	11.3	9.8	11.3	10.3

Table 2 lists the speed of the various DGEMV operations and also includes speeds for data accessed from memory. This table shows that for data accessed from cache, the speed of the operation $y \leftarrow y + Ax$ based on DOT is the same as that for the operation with A^T—there is no penalty for accessing with stride from cache.

First, we notice that for data accessed from memory, for the $y \leftarrow y + Ax$ operation it is slightly better to use the AXPY operation, which accesses the matrix with unit stride, rather than the DOT version, which accesses the matrix with stride equal to its leading dimension. Second, we see that although the speed of the $y \leftarrow y + Ax$ operation based on AXPY (9.0 Mflops) and the

Table 3: Speed of $C \leftarrow C + AB$ on the RISC System 6000-530

Conditions before operation	*Speed in Mflops*
All arrays initially in cache	47.5
A or B initially in cache	45.4
C initially in cache	42.5
No arrays initially in cache	41.5

$y \leftarrow y + A^T x$ operation based on DOT (11.0 Mflops) are slower than the corresponding Level 1 BLAS speeds based on unit stride (11.3 Mflops and 14.6 Mflops respectively—see Table 1), the speeds for accessing the matrix across a row are much faster for the Level 2 BLAS than for the corresponding Level 1 BLAS. This is because when elements of a row of a matrix are accessed, all the elements in the corresponding cache line are loaded into cache, and some will be immediately available when the next row is accessed.

2.3 Level 3 BLAS

In performing the matrix-matrix multiply operation

$$C \leftarrow C + AB,$$

where we assume that all three arrays are in cache, it is possible to increase the ratio of operations to loads to 2:1 by unrolling the DO-loops in two directions and thereby re-using each loaded element twice. Note that this ratio is optimal, in the sense that it is precisely what the hardware supports. The code fragment in Figure 3 illustrates this technique. In theory, this approach would result in a speed close to the theoretical maximum of 50 Mflops on a 25 MHz machine. In practice, we have measured 47.5 Mflops—see Table 3. Note that a production version would be complicated by the need to include code for the cases when M and N are not a multiple of two.

In general, the arrays A, B and C will be too large to fit into cache together; in any case, they need to be loaded from memory initally. It is still possible to arrange for the operations

```
      DO 30 J = 1, N, 2
         DO 20 I = 1, M, 2
            T11 = ZERO
            T21 = ZERO
            T12 = ZERO
            T22 = ZERO
            DO 10 K = 1, L
               T11 = T11 + A(I,  K)*B(K,J  )
               T21 = T21 + A(I+1,K)*B(K,J  )
               T12 = T12 + A(I,  K)*B(K,J+1)
               T22 = T22 + A(I+1,K)*B(K,J+1)
10          CONTINUE
            C(I,  J  ) = C(I,  J  ) + T11
            C(I+1,J  ) = C(I+1,J  ) + T21
            C(I,  J+1) = C(I,  J+1) + T12
            C(I+1,J+1) = C(I+1,J+1) + T22
20       CONTINUE
30    CONTINUE
```

Figure 3: Code fragment for near-optimal performance of $C \leftarrow C + AB$

to be performed with data largely in cache by dividing the matrix into blocks, as shown in Figure 4. We may then fix the block $\tilde{A}$ of the matrix A and perform every operation involving this block before moving on to another block of A. In other words, we compute the products $C_1 \leftarrow C_1 + \tilde{A}B_1$, $C_2 \leftarrow C_2 + \tilde{A}B_2$,...,$C_6 \leftarrow C_6 + \tilde{A}B_6$. In this way the block $\tilde{A}$ can be kept in cache and the data reused many times.

In addition, if we assume that the leading dimension of B is such that the block B_i can all be contained in cache, the overhead of loading B_i from memory is not too great. Moreover, each column of B_i is accessed a number of times. Thus we may perform the matrix-matrix product of these blocks at close to the peak speed of the machine.

To illustrate the overhead of cache loading, we show in Table 3 the speed of the operation

$$C \leftarrow C + AB$$

where C is 24 by 24, and A is 24 by 128, and where different arrays are forced to be accessed either from cache or from memory. These dimensions were chosen so that all three arrays can

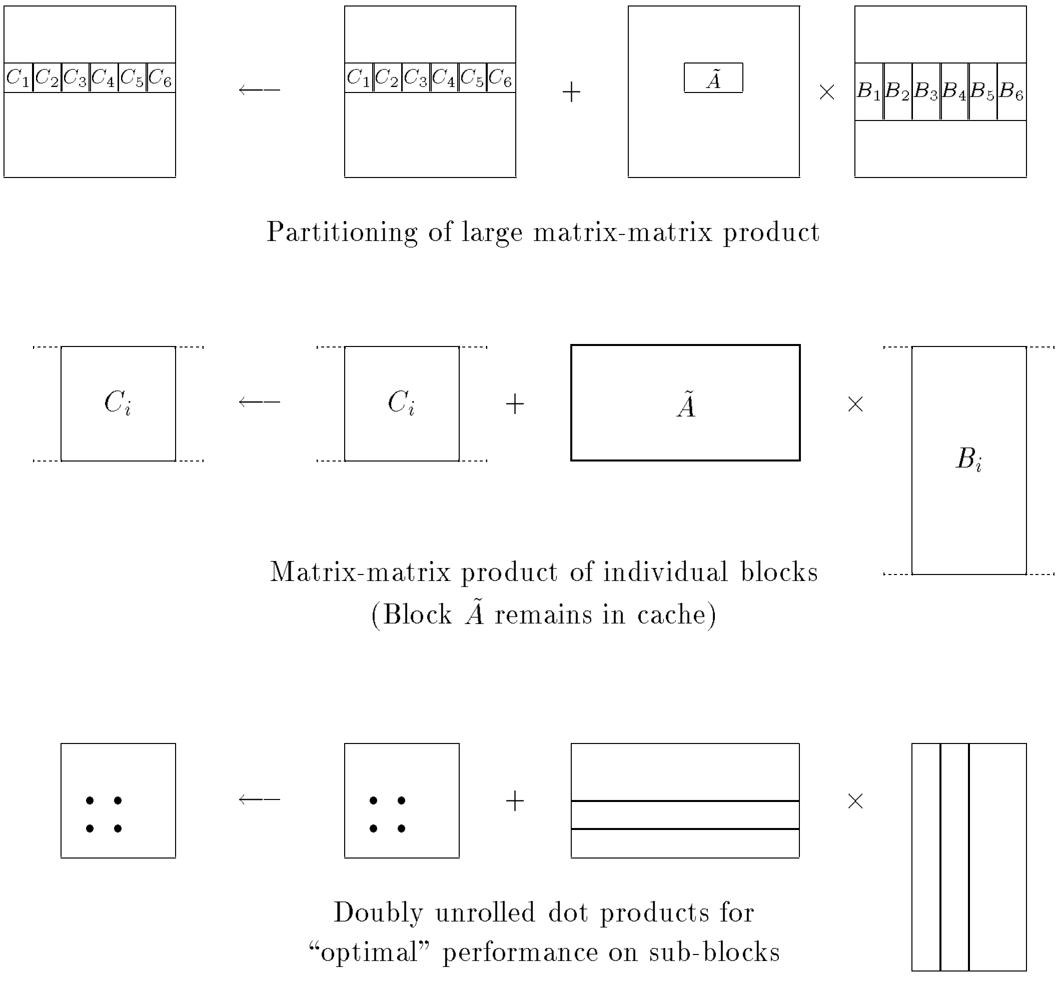

Figure 4: Blocked matrix-matrix multiply (DGEMM)

comfortably fit into cache together, and the length of the dot products is sufficiently long so that they reach their asymptotic speed.

One other important detail of the blocking strategy merits discussion. Suppose that the matrix A is declared with a very unfavorable leading dimension. Then it is possible that only a few columns of the matrix $\tilde{A}$ will fit into cache before new columns begin to flush the old columns. For example, if the leading dimension of A is 512, and $\tilde{A}$ is 32 by 32, it turns out that only 16 columns of $\tilde{A}$ will fit in cache. To overcome this problem, we copy the block $\tilde{A}$ into a work array and then perform all the operations with the work array, rather than addressing a part of the array A. This approach requires us to access A with a bad leading dimension only once, rather than 16 times, for the matrix dimensions mentioned above.

A Fortran version of the Level 3 BLAS routine DGEMM using these techniques is available from *netlib* (send mail to *netlib@ornl.gov*; in the mail message type: `send dmr from misc`).

2.4 Summary of BLAS Performance

Figure 5 shows a graph of the speed of the three BLAS routines DDOT, DGEMV, and DGEMM for increasing matrix dimensions. The operations performed by DGEMV and DGEMM are chosen so that dot products are performed on contiguous elements, i.e., $y \leftarrow y + A^T x$ for DGEMV and $C \leftarrow C + A^T B$ for DGEMM.

This graph clearly shows the benefit of increasing the ratio of floating-point operations to memory references achieved by using the Level 3 BLAS. For matrix-matrix multiply we are doing $O(n^3)$ operations on $O(n^2)$ data, representing a favorable *surface-to-volume* effect. Hence matrix-matrix multiply offers much greater opportunity for exploiting the memory hierarchy than the lower-level BLAS routines. All the experiments described here were performed on a IBM RISC System/6000 Model 530 running at 25 MHz, using the AIX XL compiler version 01.01.0000.0000 with the -O option. The BLAS shown in Figure 5 were implemented in standard Fortran 77.

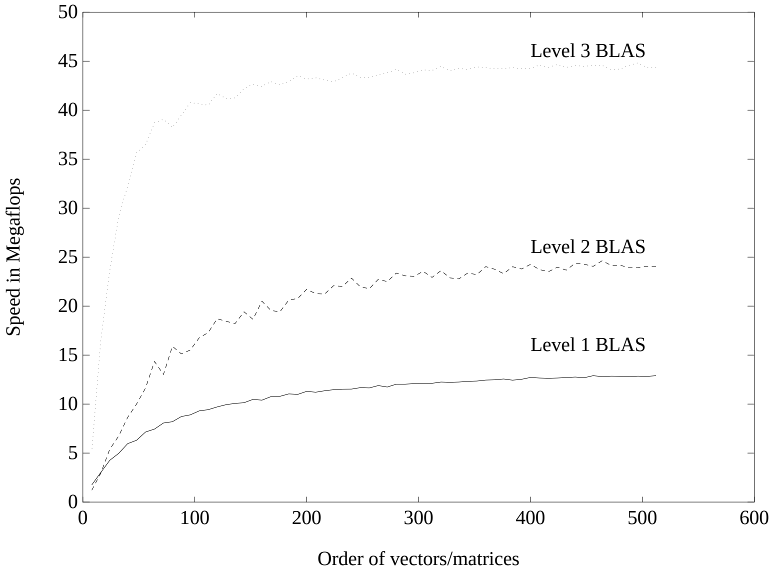

Figure 5: Speed of Level 1, 2, and 3 BLAS on the RISC System 6000-530

3 Block Algorithms and LAPACK

Experience with machines having a memory hierarchy [6, 7] indicates that it is often preferable to partition the matrix or matrices into blocks and to perform the computation by matrix-matrix operations on the blocks. By organizing the computation in this fashion, one can provide for full reuse of data while a given block is held in the cache or local memory. This approach avoids excessive movement of data to and from memory, and its benefits on the RISC System/6000 in particular are clear from the previous section.

Many algorithms can be blocked. For example, researchers have used blocking to rewrite codes for the solution of partial differential equations. Such codes make efficient use of supercomputers with small main memory and large solid-state disks [9]. All experience with these techniques has shown them to be enormously effective at squeezing the best possible performance out of advanced architectures.

Recent work by numerical analysts has shown that the most important computations for dense matrices are also blockable. A major software development project dealing with blocked algorithms for linear algebra, called LAPACK (shorthand for Linear Algebra Package), is based on this idea [1].

The LAPACK library will provide routines for solving systems of simultaneous linear equations, least-squares solutions of overdetermined systems of equations, and eigenvalue problems. The library is intended to be efficient and transportable across a wide range of computing environments, with special emphasis on modern high-performance computers. To achieve high efficiency, LAPACK developers are restructuring most of the algorithms from LINPACK and EISPACK in terms of calls to a small number of extended BLAS, each of which implements a block matrix operation such as matrix multiplication, rank-k matrix updates, and the solution of triangular systems. These block operations can be optimized for each architecture, but the numerical algorithms that call them will be portable.

3.1 Performance of Blocked Algorithms on the RISC System/6000

We used three blocked variants from LAPACK to compare the performance of LU factorization for a general matrix. These blocked variants are shown in Figure 6. The lightly shaded parts indicate the matrix elements accessed in forming a block row or column, and the darker shading indicates the block row or column being computed. The *left-looking* variant computes a block column at a time using previously computed columns. The *right-looking* variant (the familiar recursive algorithm) computes a block row and column at each step and uses them to update the trailing submatrix. The *Crout* variant is a hybrid algorithm in which a block row and column are computed at each step using previously computed rows and previously computed columns.

All of the computational work for the LU variants is contained in three routines: the matrix-matrix multiply DGEMM, the triangular solve with multiple right-hand sides DTRSM, and the unblocked LU factorization for operations within a block column. Figures 7–9 show the distribution of work among these three routines.

Each variant calls its own unblocked variant, and the row interchanges use about 2% of the total time. The average speed of DGEMM is over 40 Mflops for all three variants, but the average

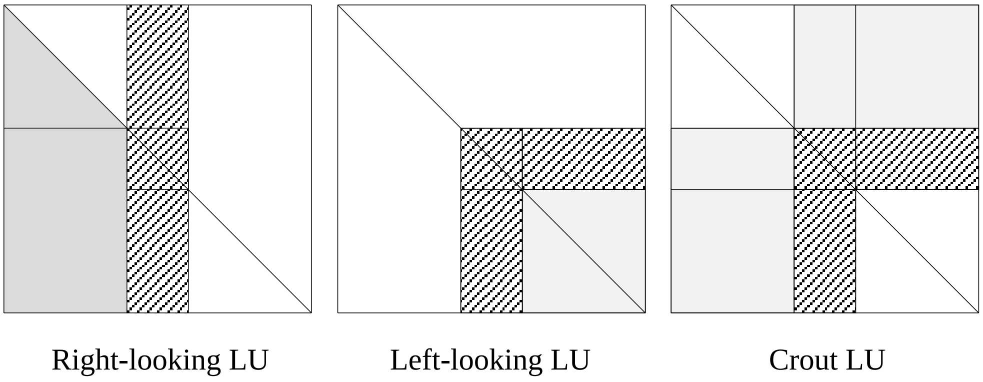

Figure 6: Variants of LU factorization on the RISC System 6000-530

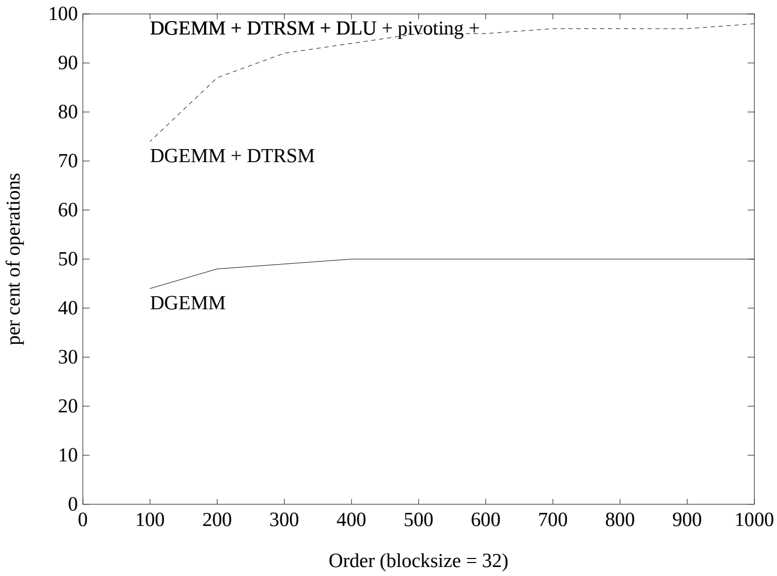

Figure 7: Breakdown of work in left-looking LU

speed of DTRSM depends on the size of the triangular matrices. For the left-looking variant, the triangular matrices at each step range in size from b to $n-b$, where b is the blocksize and n the order of the original matrix, and the average performance is 38 Mflops. For the right-looking and Crout variants, on the other hand, the triangular matrices are always of order b, and the average speed is only 29 Mflops. Clearly the average performance of the Level 3 BLAS routines in a blocked routine is as important as the percentage of Level 3 BLAS work.

Despite the differences in the performance rates of their components, the block variants of the LU factorization tend to show similar overall performance, with a slight advantage to the

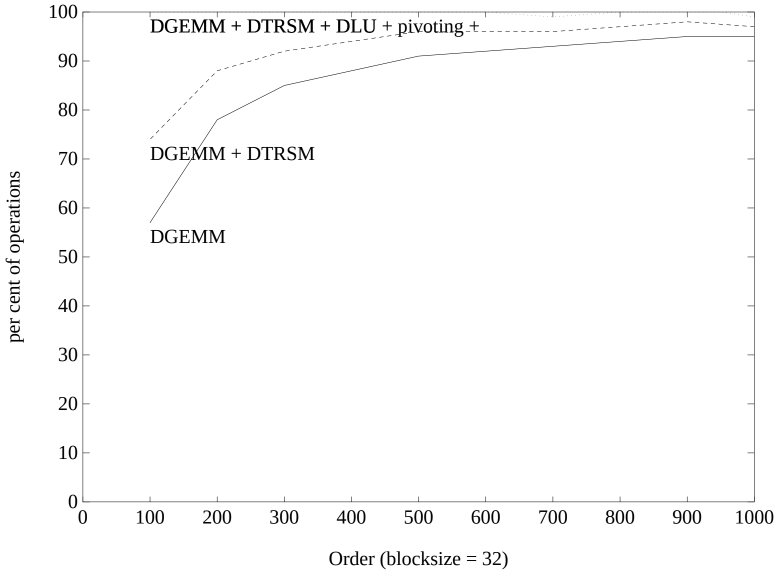

Figure 8: Breakdown of work in right-looking LU

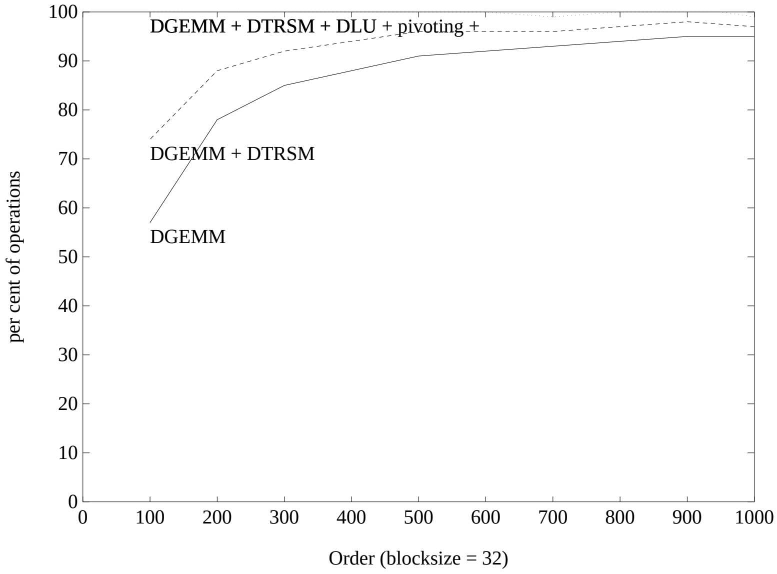

Figure 9: Breakdown of work in Crout LU

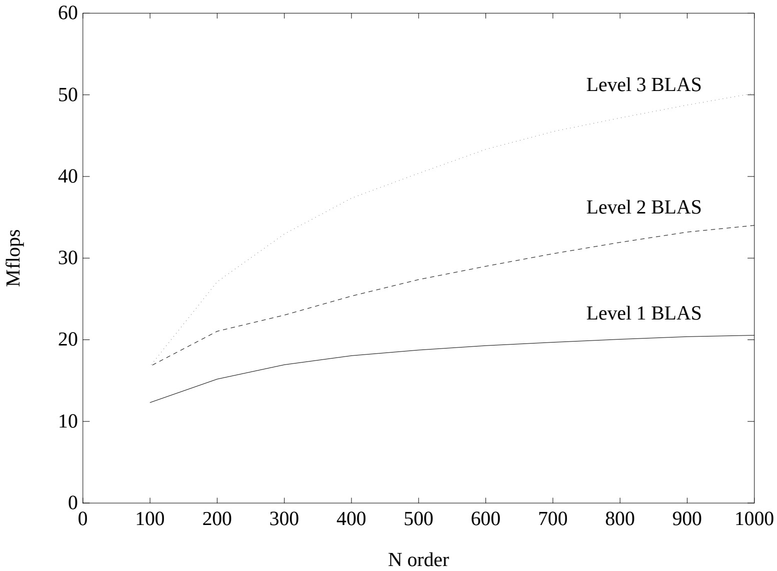

Figure 10: Speed of LU Variants on the RISC System 6000-530

right-looking and Crout variants because more of the operations are in DGEMM. Figure 10 shows the performance rates in Mflops of these three variants for different matrix sizes on a IBM RS/600-530, along with the performance of the LINPACK routine DGEFA. The optimal blocksize on the RISC System/6000 computers is 32 for most matrix sizes, but the performance varies less than 10% over a wide range of blocksizes.

4 Summary and Conclusions

The aim of this work has been to examine the performance of block algorithms on the IBM RISC workstation. Based on our experiments, we draw the following conclusions.

1. Neither the memory bandwidth nor the cycle time for the IBM RISC System/6000 is at the level of current-generation vector supercomputers. There is, however, no technical reason why this situation could not be improved.

2. The IBM RISC processor is close to matching the performance level of vector processors with matched cycle times [10]. Because of the regularity of vector loops and the ability of the RISC architecture to issue floating-point instruction every cycle and complete two

floating-point operations per cycle we expect that the RISC superscalar machines will perform at the same rates as the vector machines for vector operations with similar cycle times. Moreover, the RISC machines will exceed the performance of those vector processors on non-vector problems.

3. The LAPACK software based on blocked operations performs at near-optimal performance with minimal effort. One should note, however, that the workstation does not match the I/O performance and the number of users accommodated on largeer computers.

4. Essential to high performance is the use of optimized versions of the Level 1, 2, and 3 BLAS. The techniques and ideas used here to gain performance on the IBM RISC System/6000 should work on all RISC-based machines. To a large extent, the success will depend on the Fortran compiler's ability to generate efficient code. (We believe that this high performance is due, at least in part, to the fact that compiler writers were involved in the early design stages, rather than after the hardware designers had completed much of their work.)

5 Acknowledgements

We are grateful to Ron Bell of IBM (UK) Ltd for giving us access to a draft of his guide to Fortran and C programming on the RISC System/6000 [3].

We would also like to thank Ramesh Agarwal and Fred Gustavson of the IBM Thomas J. Watson Research Center, Yorktown Heights, and Stan Schmidt and Joan McComb of the IBM Kingston Laboratory. The optimized BLAS on which our work is based, were developed jointly by Yorktown and Kingston, and early acccess to these routines was crucial.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. Computer Science Dept. Technical Report CS-90-105, University of Tennessee, Knoxville, TN, September 1990. (LAPACK Working Note #20).

[2] H.B. Bakoglu, G.F. Grohoski, and R.K. Montoye. The IBM RISC System/6000 processor: Hardware overview. *IBM Journal of Research and Development*, 34:12–23, 1990.

[3] R. Bell. IBM RISC System/6000 performance tuning for numerically intensive Fortran and C programs. ITSC Technical Bulletin GG24-3611-00, IBM Corporation, 1990.

[4] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[5] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of fortran basic linear algebra subroutines. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[6] J. J. Dongarra and D. C. Sorensen. Linear algebra on high-performance computers. In U. Schendel, editor, *Proceedings of Parallel Computing '85*, pages 3–32, New York, 1986. North Holland.

[7] Kyle Gallivan, Robert Plemmons, and Ahmed Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, 1990.

[8] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[9] H. Lomax and T. H. Pulliam. A three-dimensional implicit code for the ILLIAC IV. In Garry Rodrigue, editor, *Computational Physics on Parallel Computers*, New York, NY, 1982. Academic Press.

[10] M.L. Simmons and H.J. Wasserman. Los Alamos experiences with the IBM RISC System/6000 workstation. Report LA-11831-MS, Los Alamos National Laboratory, 1990.

Appendix: The Model 550

Since this report was first prepared, IBM has announced a new model in the RISC System/6000 family—the Model 550. This model has exactly the same architecture as the Model 530 used in the experiments reported earlier, but has a faster CPU, running at 41.6 MHz (compared with 25 MHz for the Model 530), and a faster memory. In this appendix we reproduce versions of Tables 1–3, with data gathered from the Model 550. We also reproduce Figure 5, which demonstrates the performance attainable with the three levels of BLAS.

Table 4 (similar to Table 1) shows the speed of various Level 1 BLAS operations. In this case the predictions are based on the clock speed of 41.6 MHz, and a time of 9 cycles to load a cache line from memory to cache. This value fits the observed data better than the 11 cycles used for the Model 530. The other tables correspond exactly to those in the text.

Table 4: Speed in Mflops of Level 1 BLAS on the RISC System 6000-550

	DOT		AXPY	
Type of memory access	predicted	measured	predicted	measured
all data in cache	41.6	41.15	27.72	27.4
all data from memory:				
x and y with unit stride	26.62	26.04	20.17	19.53
x with stride 16	7.20	5.90	6.62	5.29
x and y with stride 16	4.16	3.40	3.96	2.60

Table 5: Speed in Mflops of Level 2 BLAS on the RISC System 6000-550

	Data in Cache				Data in Memory			
	$y \leftarrow y + Ax$		$y \leftarrow y + A^T x$		$y \leftarrow y + Ax$		$y \leftarrow y + A^T x$	
depth	DOT	AXPY	DOT	AXPY	DOT	AXPY	DOT	AXPY
1	38.0	26.2	38.0	26.3	16.1	15.5	18.8	13.4
2	51.0	39.2	51.0	39.2	17.9	17.4	19.2	16.4
3	57.5	40.0	57.8	40.0	19.0	21.2	19.7	17.1
4	61.0	40.0	61.7	40.2	19.6	17.2	19.8	17.2

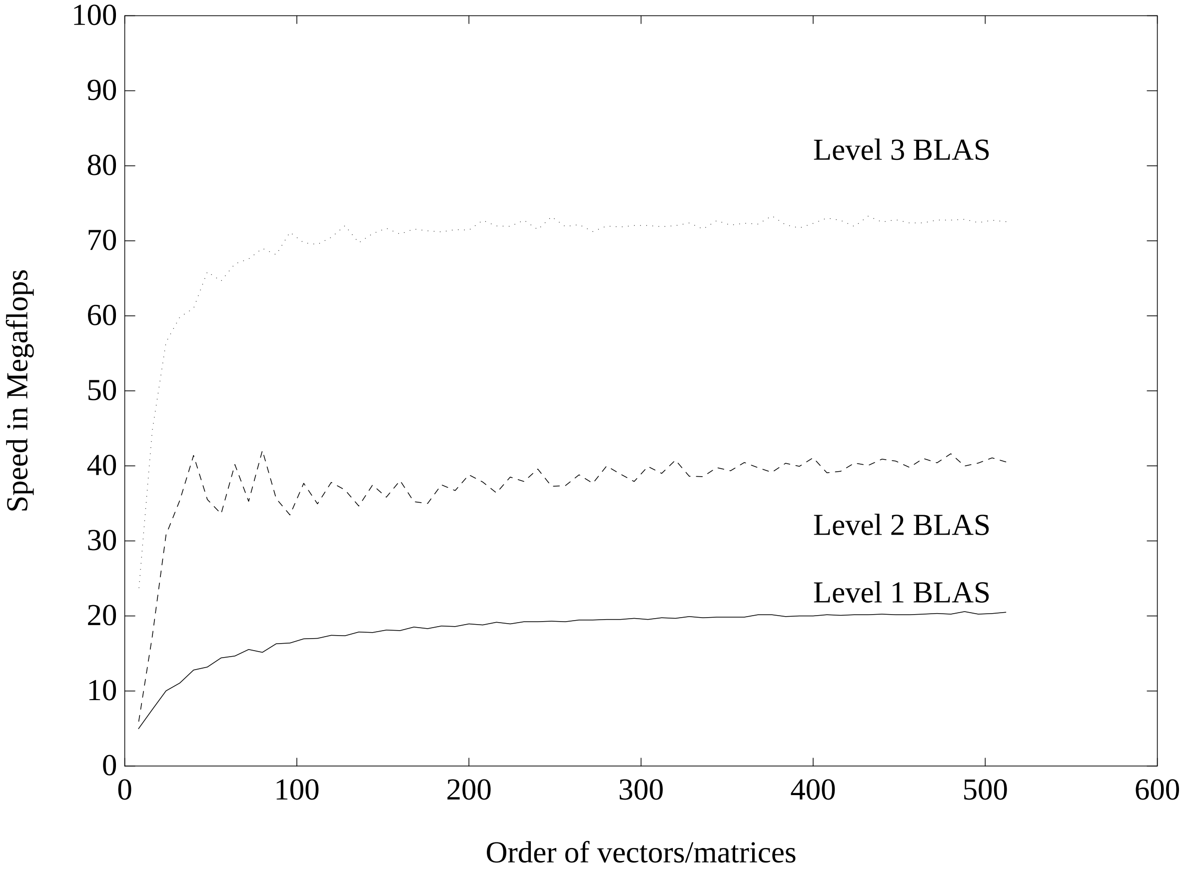

Figure 11: Speed of Level 1, 2, and 3 BLAS on the RISC System 6000-550

Table 6: Speed of $C \leftarrow C + AB$ on the RISC System 6000-550

Conditions before operation	*Speed in Mflops*
All arrays initially in cache	79.3
A or B initially in cache	75.6
C initially in cache	70.9
No arrays initially in cache	70.2