Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms

by R. C. Agarwal
F. G. Gustavson
M. Zubair

We describe the algorithms and architecture approach to produce high-performance codes for numerically intensive computations. In this approach, for a given computation, we design algorithms so that they perform optimally when run on a target machine—in this case, the new POWER2™ machines from the RS/6000 family of RISC processors. The algorithmic features that we emphasize are functional parallelism, cache/register blocking, algorithmic prefetching, loop unrolling, and algorithmic restructuring. The architectural features of the POWER2 machine that we describe and that lead to high performance are multiple functional units, high bandwidth between registers, cache, and memory, a large number of fixed- and floating-point registers, and a large cache and TLB (translation lookaside buffer). The paper gives two examples that illustrate how the algorithms and architectural features interplay to produce high-performance codes. They are BLAS (basic linear algebra subroutines) and narrow-band matrix routines. These routines are included in ESSL (Engineering and Scientific Subroutine Library); an overview of ESSL is also given in the paper.

Introduction

The new POWER2™ workstations [1–4] of the RISC System/6000® (RS/6000) family of processors provide multiple fixed-point units (FXUs) and floating-point units (FPUs) which can work in parallel if there are no dependencies. We call this functional parallelism. To achieve functional parallelism requires that the underlying numerical algorithm be parallelized at a very low level

(instruction level). The functional parallelism can be achieved if, at the innermost loop level, several computations can be done in parallel. Various algorithmic techniques can be utilized to facilitate functional parallelism. Loop unrolling is an example of such a technique. Multiple FXUs help in prefetching of data into cache. One of the two FXUs can be utilized for cache prefetching, while the second FXU continues to remain available to load/store data (already in cache) into registers. This also requires algorithmic modifications. In many situations, the computation may have to be restructured to get around the serial bottlenecks and arithmetic pipeline delays. POWER2 workstations also provide quad load/store instructions which double the effective bandwidth between floating-point registers (FPRs) and cache. To exploit the quad load/store instructions, the algorithm may have to be modified to access data with stride one. These are all examples of algorithmic techniques which can help to exploit the functional parallelism capability of POWER2 and to realize its full performance potential of four floating-point operations (flops) per cycle. The techniques described in this paper have been heavily utilized in the development of some of the ESSL (Engineering and Scientific Subroutine Library) subroutines for the RS/6000 family of workstations. (Although the PowerPC™ family [5, 6] of processors is not discussed in this paper, the techniques described here work equally well, with appropriate modification, for PowerPC implementations.)

In this paper, we first give an overview of ESSL. We then describe salient CPU and cache/memory features of POWER2 which can affect optimal algorithm design. Next we describe various algorithmic techniques used to facilitate utilization of multiple function units. In each case, we provide appropriate examples and discuss the performance achieved.

ESSL overview

ESSL is a high-performance library of mathematical subroutines for IBM RISC System/6000 workstations (ESSL/6000), and ES/9000™ and ES/3090™ vector and/or scalar mainframes (ESSL/370) [7]. Currently, ESSL V2.2 consists of 441 subroutines that cover the following computational areas: linear algebra subprograms, matrix operations, linear algebraic equations, eigensystems analysis, Fourier transforms, convolutions/correlations, and other related signal-processing routines, sorting and searching, interpolation, numerical quadrature, and random number generation.

ESSL can be used for both developing and enabling many different types of scientific and engineering applications, as well as for numerically intensive applications in other areas such as finance. Existing applications can be enabled by replacing comparable subroutines and in-line code with calls to ESSL subroutines. Because of the availability of a large number of subroutines in ESSL, the effort required in developing a new application is significantly reduced. This also makes it easier to move an application to a new platform, because ESSL is tuned for all platforms where it is available.

In the area of linear algebra, ESSL routines can be used in two different ways. ESSL provides a set of routines to solve linear equations of various kinds. Most of these routines are functionally the same as the public domain software LAPACK [8, 9], though not necessarily in the same call sequence. The user application can be modified to call these ESSL subroutines instead of LAPACK subroutines. This approach provides the highest level of performance. Alternatively, ESSL also provides a complete set of tuned basic linear algebra subprograms (BLAS). There are 140 BLAS subroutines in ESSL. BLAS are an industry-wide standard [10–14], providing a uniform functionality and call sequence interface, which makes an application using calls to BLAS highly portable across high-performance platforms from different vendors. In LAPACK subroutines, most of the computing is done in BLAS subroutines. LAPACK expects platform vendors to provide a set of tuned BLAS to achieve high performance. This is the alternative approach, in which the user application calls LAPACK subroutines and links to ESSL for tuned BLAS. This approach provides portability and performance at the same time. The performance achieved in this approach is slightly lower than that obtained by calling ESSL subroutines directly, but it is still very good.

ESSL/6000 provides a set of subroutines tuned for performance on the RISC System/6000 family of workstations which include the older POWER™ workstations and the newer POWER2™ workstations. These are predominantly compatible with the ESSL/370 product, resulting in easy cross-platform migration between mainframes and workstations. All ESSL/6000 computational subroutines are written in FORTRAN; they are callable from application programs written in FORTRAN and in C. All of the subroutines are fully described in the *ESSL Guide and Reference* [7].

POWER2 CPU considerations

POWER2 workstations differ from the original POWER workstations in many respects. POWER workstations have one fixed-point unit (FXU) and one floating-point unit (FPU) and therefore can perform one floating-point multiply–add instruction and a fixed-point instruction every cycle, if there are no dependencies. Additionally, branch instructions can also be overlapped and therefore in effect result in zero-cycle branches. All load/store instructions, including the floating-point load/stores, are handled by the FXU. Additionally, floating-point stores also require the FPU, and therefore these instructions cannot be

overlapped with the floating-point arithmetic instructions. See [15] for a full description of a POWER machine.

POWER2 workstations have two FXUs and two FPUs and therefore can perform two fixed-point instructions and two floating-point instructions every cycle, if there are no dependencies. In contrast to POWER, on POWER2 floating-point store instructions can also be overlapped with floating-point arithmetic instructions. Floating-point instructions have a two-to-three-cycle pipeline delay; therefore, to keep both FPUs fully utilized, at least four independent floating-point instructions must be executing at the same time. (See [1–3] for details.)

Two additional innovative features of POWER2 are the floating-point quad load/store instructions. The quad load instruction can load two consecutive doublewords of memory into two consecutive FPRs, and the quad store instruction can store two consecutive FPRs into two consecutive doublewords of memory. With both FXUs in use, this results in an effective bandwidth of four doublewords per cycle between the cache and the FPRs. This increased bandwidth is particularly important for those computational kernels where performance would otherwise be limited by the bandwidth between registers and cache. BLAS-1 (level-1 BLAS), for example DDOT and DAXPY, are examples of such kernels. For these kernels, on the same cycle time basis, POWER2 can perform four times faster than POWER. However, it is not always easy for the compiler to generate quad load/store instructions; loops often require additional unrolling. Also, quad load/stores require two consecutive FPRs; this restriction imposes additional constraints on the register assignment logic of the compiler. Please note that quad load/store instructions can be used only if two consecutive data elements from memory are needed in the loop. This may also require a restructuring of the loops.

Individual quad accesses which cross a cache line boundary require additional cycle(s). If the loop performance is limited by the cache bandwidth, quad loads crossing the cache line boundary reduce the available bandwidth. Special coding techniques can be used to handle such a situation. For example, assuming that all double-precision arrays are aligned on doubleword boundaries, there are two possibilities: The array is either aligned on a quadword boundary or on an odd doubleword boundary. Here, we are also assuming that the array is accessed in the loop with stride one using quad load instructions. Then, if the array is aligned on a quadword boundary, a quad load will never cross the cache line boundary. This follows from the fact that the cache line size is a multiple of quadwords. If the array is aligned on an odd doubleword boundary, one load is purposely handled outside the loop, thus making all quad loads inside the loop quadword-aligned. For double-precision two-dimensional arrays, if possible, the user should make the leading dimensions even. This ensures that if the first column is quadword-aligned, all other columns are also quadword-aligned.

POWER2 cache considerations

The data cache size on POWER2 workstations is 256 KB; however, only 128 KB may be accessible on machines having fewer than four memory cards. In this paper, we refer to the data cache as simply the cache. All POWER2 models have 1024 cache lines arranged in four-way associative sets. The cache size of the largest POWER2 models is four times greater than that of the largest POWER models. POWER2 models also have a significantly higher bandwidth to the memory system. The memory system bandwidths on all machines are designed to fetch a complete cache line in eight cycles. Thus, machines with larger caches also have higher bandwidth to the memory system. All POWER2 workstations have a 512-entry, two-way set-associative TLB (table lookaside buffer). This is significantly larger than the 128-entry TLB on POWER machines. Furthermore, the number of cycles required for TLB miss processing on POWER2 is considerably smaller than the number of cycles required for TLB miss processing on POWER. This has a very significant impact on considerations of blocking (of large problems) for POWER2. For most problems, the TLB size of POWER2 is not a consideration in blocking. In those computational kernels where arrays are used several times, appropriate cache and TLB blocking is sufficient to give the best possible level of performance. For these kernels, the delay in accessing a cache line is not important because the data are used several times. BLAS-3 routines are examples of such kernels. However, in BLAS-1 and BLAS-2 routines, the arrays are used only once, and cache miss latency becomes the important consideration.

Loop unrolling

Loop unrolling is a common technique used to exploit multiple functional units and quad load/store capabilities. (See [16] for a discussion of how the compilers for POWER and POWER2 handle this.) In its simplest form, unrolling a loop amounts to a mini-vectorization of the loop. For example, unrolling a loop by eight is equivalent to using a vector length of eight. Generally speaking, vectorization results in independent computations, unless the vector operation is a reduction operation. Inner product (dot product) computation and finding the maximum or minimum of a vector are examples of reduction operations. For these examples also, parallelization is easily achieved if the vectors are long. For the inner product example, outside the loop, we can set up four variables (or registers) for partial sums. The

loop is on the length of the inner product and is unrolled by four. Inside the loop, the four partial sum registers are updated with multiply–add operations. This results in four independent computations which can take place in parallel. At the completion of the loop, the four partial sum terms are added together to form the inner product. The final sum computation is only partially parallelized.

Loop unrolling also helps in reusing the data loaded in registers, and thereby reduces the bandwidth (load/store) requirements between registers and cache and between cache and main memory. The loop unrolling need not be limited to the innermost loop. A series of nested loops can be unrolled to further facilitate data reuse. Some loop variables can be reused only if outer loops are unrolled. The matrix–matrix multiplication is a good example of nested loop unrolling. By unrolling all three nested loops, all variables in the loop can be reused. The degree of loop unrolling is generally limited by the number of registers available. Heavy loop unrolling may require more registers (floating and/or fixed) and therefore may result in spills. Spills are caused when the number of "logical" registers needed by the compiler exceeds the available number of architected registers. In that case, inside the innermost loop, "logical registers" are saved and restored from temporary memory location, resulting in a large degradation of performance. It is advisable for the user to look at the listings generated by the compiler. If the listings indicate spills in the innermost loop, the degree of unrolling should be reduced.

In many situations, an array is not accessed with stride one in the innermost loop, so quad load/store instructions cannot be used. However, this array may have a stride-one access in one of the outer loops. In that case, to facilitate the use of quad/load store instructions, that outer loop should be unrolled. If multiple arrays are accessed in the innermost loop, they may have a stride-one access pattern in different nesting levels of the nested loop. To facilitate quad access of all these arrays, all corresponding nested loops at nesting levels in which an array has stride-one access must be unrolled.

To summarize, loop unrolling serves two goals. The main goal is parallelization of the computation; the secondary goal is reduction in register/cache/memory bandwidth requirements. At some degree of unrolling, we will have achieved a sufficient level of functional parallelism and memory bandwidth reduction to result in peak performance, if the data remain in cache. The question is "Can we get a higher level of performance by further unrolling the loop(s)?" The answer depends on the ratio of the computation cost (number of floating-point operations, or "flops") to the data movement cost (number of data items involved in the computation). For the BLAS-3 (matrix–matrix multiplication is an example) kind of computation, where the ratio of flops to data movement cost is large, we do some kind of cache blocking so that data remain in cache. For these computations, occasional cache misses do not seriously affect overall performance. However, for those computations in which flops and data movement cost are about equal (for example, BLAS-1 and BLAS-2), data in cache are not reused, and the cache miss latency becomes very important. In such applications, by reducing the bandwidth requirements (via loop unrolling) between registers and cache, we can free up one of the FXUs to do cache prefetching. This is discussed in detail in the next section.

Algorithmic prefetching

On POWER2 machines, the cache miss latency is roughly of the order of 14–20 cycles. The desired data come after 14–20 cycles, followed by the rest of the line in a wraparound fashion in the next seven cycles. There are two FXUs, and each of them can process only one cache miss at a time. In a typical stride-one access code, if a load results in a cache miss, the subsequent load also results in a cache miss to the same line. This ties up both FXUs in fetching the same cache line. Specialized coding gets around this problem and can thus greatly improve memory system performance. We call this algorithmic prefetching [17]; it can significantly improve performance on POWER2 machines. It is fairly easy to implement algorithmic prefetching for the stride-one situation on POWER2. Typically, several cycles before the data from the next cache line are actually needed, a dummy load of an element from the next cache line is done. (A dummy load is a load where the data loaded are not actually used in the computation.) If the next line is already in cache, the load is completed in one cycle, and the FXU becomes available for the next set of instructions. On the other hand, if this load results in a cache miss, it ties up one of the FXUs until the cache miss is processed. However, the other FXU is still available to do the required loads into FPRs to feed both FPUs. When quad loads are extensively used, many computing kernels require only one FXU to feed both FPUs.

Prefetching on POWER is more difficult, because POWER has only one FXU, and a cache miss will stall it. However, in those loops where there are more arithmetic operations than load/store operations, specialized techniques can be used to do prefetching on POWER [17]. The primary concept is to load FPRs with data at the beginning of the loop, so that when the cache miss stalls the FXU, the FPU is kept busy with useful work using data already loaded in the FPRs.

We have used algorithmic prefetching on POWER and POWER2 in many computing kernels. However, prefetching requires extensive loop unrolling. This loop unrolling with dummy loads can actually be done in FORTRAN. In prefetching, there are several different

variations, some being quite intricate and complex. If the cache bandwidth is a consideration, the loop is unrolled in such a way that the data loaded during a "dummy load" are actually used in the computation. Depending on the kernel and the number of arrays to be prefetched, this can become complicated, especially if quad loads are to be used and relative quad alignments of different arrays are to be taken into account. On the other hand, if the cache bandwidth is not a consideration (one FXU can handle all of the required loads), a simple prefetching scheme will do a dummy load and not use the data. In this case, if the data are actually in cache, prefetching does not degrade performance. If the data are not in cache, prefetching improves performance significantly.

BLAS implementation using loop unrolling and cache prefetching

We now illustrate the use of the above techniques in implementing some key BLAS routines.

- *BLAS-1 implementation*

Because there is very little reuse of data loaded in registers, BLAS-1 performance tends to be limited by the available bandwidth between the cache and the FPRs. Maximum bandwidth is achieved by using quad load/store instructions. An optimal implementation also tries to avoid crossing a cache line boundary on quadword accesses. There is also some limited opportunity for algorithmic prefetching. Here, we describe the implementation and performance of two key BLAS-1 kernels—the DAXPY and DDOT routines. We assume that all arrays have stride one; otherwise, it is not possible to use quad load/store instructions.

DAXPY

In DAXPY, we update a vector **y** with a scalar multiple of another vector **x**. The scalar multiplier is loaded outside the loop and remains in a register throughout the computation. For each multiply–add (FMA), DAXPY requires two loads and one store; therefore, its performance is limited by the available bandwidth to the FPRs. Depending on the alignments of the **x** and **y** arrays, there are four possibilities. When both arrays are even-aligned, the coding to obtain optimal quad load/stores is easy. If both arrays are odd-aligned, one element is computed outside the loop. This makes both the remaining arrays quad-aligned. The difficult case arises when one array is even-aligned and the other is odd-aligned. In this case also, we can restructure the computation so that all quad accesses inside the loop are quad-aligned. This requires accessing one element of the odd-aligned array outside the loop. The loop is unrolled by four and executes in three cycles, achieving the peak bandwidth of two quadword accesses between the cache and the FPRs every cycle. On a 50-MHz POWER2, the best performance that can theoretically be expected is 133 MFLOPS. For data in cache, we actually achieve nearly this performance for all four possible alignments of the arrays. By comparison, on a 50-MHz POWER machine, the best possible performance is 33 MFLOPS.

DDOT

The DDOT function computes the dot product of two vectors. Since FMA requires two loads, this seems to match the capabilities of POWER2 ideally, assuming that data are in cache. Thus, on POWER2, DDOT should run at its peak rate of four flops per cycle. However, as a result of the register-renaming implementation, the FXUs cannot perform two load quads every cycle on a continuous basis. The best that can be achieved is eight load quads in five cycles. Thus, the best performance that can be expected on a 50-MHz POWER2 is 160 MFLOPS, and we nearly achieved that level of performance. Recall that in the DAXPY case we were doing both quad loads and quad stores; therefore, in that case we did achieve the peak bandwidth of two quad load/stores every cycle.

We developed two versions of the DDOT function, one for data in cache and another for long sequences where data are unlikely to be in cache. For the function which assumes that data are in cache, we unrolled the loop by eight and used four temporary variables to accumulate the partial results. The four subresults were then added together outside the loop. As in the DAXPY case, we had to take into account even–odd quad alignments of both arrays in order to achieve near-peak performance. However, the cost of examining the alignments of both arrays and setting up the unrolling by eight adds extra overhead to the subroutine, which is significant for small n. As an example, on a 50-MHz machine, for a dot product of size $n = 2000$, we measured 148 MFLOPS for all four possible quad alignments of the two arrays.

We also developed a version of the DDOT function which does algorithmic prefetching for long sequences. Here we assume that data are not in cache. The subroutine must know the cache line size of the machine in order to do optimal prefetching. When data are not in cache, the performance is limited by the available bandwidth between cache and the memory system, and the prefetching is implemented to maximize it. Recall that during prefetching, one FXU is tied down when a cache miss occurs, but the other FXU remains available to load/store data already in cache. For these routines, we unrolled the loop by 16 and did dummy loads 16 elements apart for both arrays. On a POWER2 machine (50 MHz and 256KB cache size), for data not in cache, this subroutine performed at about 103 MFLOPS, while the subroutine which assumed that the data were in cache performed at

```
T0  = Y(I)                    ! load 24 y elements
T1  = Y(I+1)                  ! in 24 FPRs.

    ...

T23 = Y(I+23)

DO J = J1, J1+JBLK-1
    XJ = X(J)                 ! load an element of x

    F0  = A(I,   J)           ! one load quad loads
    F1  = A(I+1,J)            ! both F0 and F1
    T0  = T0 + XJ*F0
    T1  = T1 + XJ*F1

      ...

    F0  = A(I+22,J)
    F1  = A(I+23,J)
    T22 = T22 + XJ*F0
    T23 = T23 + XJ*F1
ENDDO

Y(I)    = T0                  ! store y elements
Y(I+1)  = T1                  ! after the loop

  ...

Y(I+23) = T23
```

Figure 1

DGEMV matrix–vector multiplication (without prefetching).

about 74 MFLOPS. This represents a 40% performance improvement, due to algorithmic prefetching. However, the prefetched version of this subroutine requires extra loads, and if the data are actually in cache, its performance drops by about 10% compared to the subroutine optimized for data in cache.

If the data loaded in registers (for prefetching) can actually be used in the computation, we can achieve the best performance for data in cache as well as for data not in cache. The coding becomes complicated because one must take into account not only relative quadword alignments of the two arrays, but also their relative cache line alignments. This extra logic adds overhead to the routine which is justifiable only for long sequences, say of the order of 2000. We implemented one such version on the above machine. For data in cache, this routine performed in the 150–156-MFLOPS range. For data not in cache, it performed in the 97–103-MFLOPS range. The performance of the subroutine varied in a narrow range depending on the relative alignments of the two arrays.

- *BLAS-2 implementation*

BLAS-2 computations typically involve a matrix and one or two vectors. In these computations, the matrix elements are generally used only once; for a large matrix, most of it cannot be in cache. When the matrix is not in cache, the best that can be expected is to fully utilize the matrix data brought into cache and simultaneously prefetch the next cache line. During the prefetch, computing is being done on the data just brought into cache. We must also use register, cache, and TLB blocking for the matrix and the vectors, in order to fully use the data before they are swapped out of the cache and the TLB. Reference [17] describes algorithmic prefetching as it was implemented for BLAS-2 for POWER. We now consider implementation on POWER2. As pointed out earlier, because of multiple function units on POWER2, prefetching is easier. It is particularly important to get sufficient reuse of data loaded in registers so as to minimize the load/store requirements. This requires the inner loop to be unrolled by a large factor, and the use of the quad load/store capability of POWER2. To illustrate the computational techniques, we describe the example of the matrix–vector multiplication subroutine DGEMV, where the matrix **A** is stored in the normal form, i.e., column major order.

The location in memory of matrix **A** is significant. Typically, we declare a two-dimensional array in FORTRAN as $A(LDA, *)$ where $LDA \geq M$. Here, LDA refers to the leading dimension of the array **A**. In what follows, a matrix stored this way is called a dense matrix. If the (1, 1) doubleword of matrix **A** is stored in memory location α, the FORTRAN convention of storing **A** places the (i, j)th doubleword of **A** in memory location $\alpha + i - 1 + (j - 1) \times LDA$. (We assume that this address and α are in units of doublewords.) It is important to realize that the value of LDA can influence how well blocks of **A** fit into cache and TLB. For cache considerations, a good LDA is an odd multiple of the line size. For $LDA \geq 512$, each column of **A** begins on a different page. In order to avoid a TLB miss, n must be chosen so that translation information for n pages fits comfortably into the TLB. We have determined experimentally that $n = 200$ is a good choice for the two-way set-associative TLB with 512 entries.

DGEMV—Normal case

In DGEMV, a vector **x** is multiplied by a matrix **A** and then added to another vector **y** ($\mathbf{y} \leftarrow \mathbf{y} + \mathbf{A}*\mathbf{x}$). Cache prefetching is the most important consideration in DGEMV. An optimal implementation of cache prefetching

requires knowing the cache line size. Here, we describe the implementation on machines with a 256-byte line size. The outermost blocking was on the number of columns, to minimize the finite cache and TLB effects. Within a vertical block, we unrolled the computation by a large factor (i.e., we implemented a horizontal register blocking). The ideal block size (the unrolling factor) corresponds to the cache line size, so that in each subcolumn there is exactly one cache line (32 doublewords). However, because of the floating-point register limitation (there are only 32 FPRs), we restricted the unrolling to 24. The innermost loop operates on the number of columns in a vertical block. Outside this loop, we loaded 24 elements of the **y** vector into 24 FPRs, T0, T1, $\cdots$, T23, corresponding to the horizontal block size or the loop unrolling factor. Within this loop, we processed a subcolumn of the **A** matrix of size 24. This is like a mini-vectorization with a vector length of 24. We can assume, because of the cache blocking, that a block of the **A** matrix remains in cache. The code shown in **Figure 1** is indicative of the code without prefetching. Here XJ corresponds to a floating-point register. Note that one quad load loads two FPRs, feeding both FPUs. Thus, one FXU can feed both FPUs, except for the initial load of x(j) into FPR XJ. There are 24 FMAs in the innermost loop (the J loop), requiring 24 loads for the matrix elements which can be performed as 12 quad loads. This gives a 24× reuse factor for XJ. Thus, for 12 cycles, we can keep one FXU feeding both FPUs at the peak rate of two FMAs per cycle. The other FXU is free to handle cache miss processing by doing a dummy load of an element from the second next column of A, which is not likely to be in cache. The prefetching is accomplished by inserting the following instruction in the inner loop:

```
D = A(I+23, J+2)      ! dummy load for prefetch
```

The dummy variable D is not used in the loop. Its sole purpose is to bring the desired section of column (J + 2) into cache if it is not already in cache. The second FXU on POWER2 accomplishes this goal. By prefetching two columns ahead, our measurements show that all of the required data are in cache. If the LDA of the **A** matrix is even and the initial alignment of the matrix is on an odd doubleword boundary, we process one row outside the main blocking loop, so that each subblock in the main loop is aligned on a quadword boundary. This is to make sure that none of the quad loads inside the inner loop cross a cache line boundary. If LDA of the matrix is odd, for every other column quad loads will cross the cache line boundary, slightly degrading the performance. This is the reason why we recommended earlier that the leading dimensions of multidimensional arrays should be even.

This implementation of DGEMV with algorithmic prefetching is optimal even when the matrix is actually in cache. In that case, the prefetch load does not result in a cache miss and becomes an ordinary load. Since the innermost loop is not limited by the load/store bandwidth, this extra load has no impact on the execution of the loop. For matrices which fit in cache, we achieved 96% of the peak performance. For very large matrices which do not fit in cache, we achieved 81% of the peak (on a 50-MHz machine).

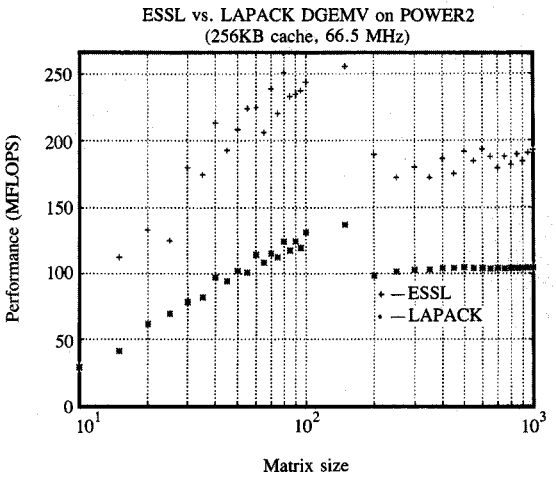

Figure 2

ESSL performance compared with LAPACK for BLAS-2 matrix–vector multiplication.

Figure 2 compares the performance of ESSL DGEMV against that of the LAPACK "vanilla" routine. The performance is plotted for square matrices of sizes 10 to 100 in steps of 5, and 150 to 1000 in steps of 50. For values of n above 200, we see the effect of the matrices no longer residing in cache.

- *BLAS-3 routine—DGEMM*

For BLAS-3 routines, appropriate cache and TLB blocking is generally sufficient to give the best possible level of performance. For these kernels, the delay in accessing a cache line is not important because the data are used multiple times. DGEMM is typical of BLAS-3 routines. It basically computes the product of two matrices. For DGEMM, fairly good performance can be obtained from the vanilla code, if appropriate preprocessing options are used at the compile time. In most cases, the preprocessor does a reasonable job of cache blocking. The problem arises when the matrix dimensions are powers of two (or related to them). In those cases, because of the cache congruence class conflicts, the effective cache size is

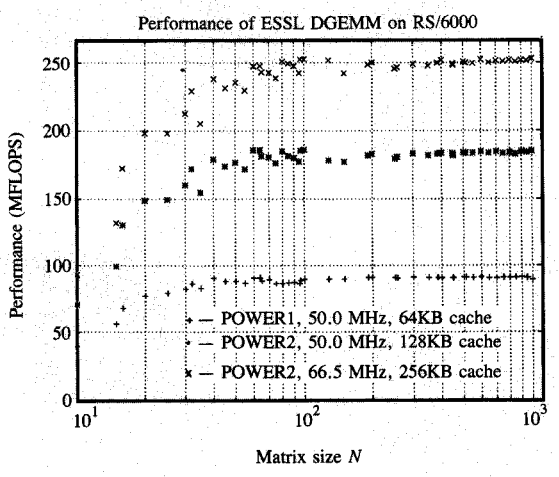

Figure 3

Performance of ESSL for BLAS-3 matrix multiplication.

reduced. In this case, preprocessor cache blocking is not very effective.

ESSL BLAS-3 routines do cache and TLB blocking customized for the platform on which they are run. They are designed to provide robust performance in almost all situations. If necessary, subarrays are copied into temporary buffers in order to eliminate any problem due to bad leading dimensions. Because the copied data are used many times, the cost of copying becomes insignificant. If the arrays are blocked for cache, we can assume that data remain in cache. In that case, the only consideration is to obtain peak performance at the innermost loop level. For BLAS-3 kernels, the bandwidth between cache and FPRs is not a consideration, because the nested loops can be unrolled in many different ways to get a significant reuse of data loaded into registers. The unrolling of loops also makes it possible to utilize multiple functional units fully, and avoids FPU pipeline delays.

For the previous release of ESSL, which was only for POWER, we implemented a two-by-two unrolling; i.e., a two-by-two block of the result matrix was computed in the innermost loop. This is equivalent to computing four dot products in the innermost loop. This was sufficient to give the peak performance on POWER. On POWER2, to ensure robust performance in utilizing multiple functional units, we implemented a four-by-two unrolling. This resulted in the peak performance at the innermost loop level for all combinations of the matrix form parameters. Form parameters specify whether the matrices are stored by rows or columns. The four-by-two blocking used for POWER2 is also optimal for POWER. This helps in producing a single source code for POWER and POWER2 machines. The cache and TLB blocking is customized for the platform. This requires different compilations for POWER and POWER2. By using a different compilation for POWER2 machines, we also obtained some additional performance by exploiting quad load/store instructions which are not available on POWER. Blocking for different cache sizes is done at run time. The line size of the machine can be used to determine the cache size at run time. We use the special subroutine IRLINE to determine line size. Once the line size is known, we can set cache size parameters for the particular POWER or POWER2 machine. This determination of cache size parameters is done only once.

Figure 3 shows DGEMM performance on two different models of POWER2 and one model of POWER. The performance is plotted for square matrices of sizes 10 to 100 in steps of 5, 150 to 1000 in steps of 50, and all powers of two from 16 to 512. For small values of N, the performance is somewhat uneven, because of our choice of four-by-two blocking. Note that even for matrix sizes as small as 20, performance reaches 200 MFLOPS on a 66.6-MHz POWER2 machine. For large-size matrices, including powers of two, the performance is essentially uniform in the range of 90–95% of the peak for the machine.

To summarize, a very efficient DGEMM has been produced on POWER and POWER2 machines by using cache blocking and dot-product-based kernels. DGEMM is the basic computing kernel and building block for almost all of the computing in the area of linear algebra for dense matrices. Therefore, it must demonstrate uniformly good performance for all reasonable choices of parameters and matrix storage formats. ESSL DGEMM has this property.

Algorithmic restructuring of narrow-band matrix computations

On POWER2 machines, the divide and square-root instructions take many cycles to complete. They utilize one of the two FPUs. If the next floating-point instruction depends on the result of the previous divide/sqrt instruction, the other FPU is stalled. In some numerical kernels, cycles spent in the divide/sqrt instructions account for a large fraction of the total computation time. If during this period one of the FPUs remains under-utilized, the full performance potential of POWER2 is not achieved. This constitutes a serial bottleneck in the computation. In many such situations, it may be possible to restructure the computation so that both FPUs can be fully utilized. We now consider Cholesky factorization of a positive definite symmetric narrow-band matrix. This numerical kernel serves as an example that illustrates algorithmic restructuring of the computation. For this computation, factoring each column requires one square-root operation, one divide operation, and approximately $m^2/2$

multiply–add operations. When the bandwidth m of the matrix is very small (narrow-band case), the cycles spent in the square-root operation (about 27 cycles) and the divide operation (about 19 cycles) dominate. In the standard computational algorithm, the square-root and divide operations are done serially; thus, during these operations, the other FPU remains idle.

A general (for any bandwidth) implementation of the Cholesky band factorization consists of three nested loops. The outermost loop is on the length n of the band matrix, and the two inner loops are on the bandwidth m. For very small values of m, the loop-setup time for the inner loops becomes very significant; therefore, an optimized implementation completely unrolls these loops. We have done this by writing different routines for each value of m. We illustrate the techniques for the computation $m = 2$. This case corresponds to a positive definite symmetric penta-diagonal problem which is frequently encountered in some applications. For our implementation, we choose a left-looking algorithm (see form jki of [18]), and for simplicity we leave out the computational details for the factorization of the first m columns. This also assumes that only the lower part of the symmetric band matrix is stored as a compact band matrix, with the main diagonal as the top row (row zero) and the next two diagonals as the next two rows of a two-dimensional array. A straightforward implementation of the algorithm is represented by the code shown in **Figure 4**. In this code, only one FPU is utilized during the computation of the square-root and divide operations. Since these two operations account for most of the loop time, we are not getting any benefit from the second FPU. Another problem with the above code is that a1 and a2 values are stored in memory and then immediately loaded in the next iteration (j + 1), as a1j1 and a2j1 values. On many high-performance machines, a store/load sequence can be delayed by many cycles because of the memory system and pipeline latencies. This can easily be avoided by reusing the values from the previous iteration of the loop. This eliminates the store/load sequence penalty, and it also reduces the total number of loads by two. This is shown in the code presented in **Figure 5**.

Now let us address the problem of serial bottleneck. An equivalent expression for computing t is sqrt(a0) * (1./a0). By using this identity, both sqrt(a0) and 1./a0 can be computed in parallel using both FPUs, as shown in **Figure 6**.

The code shown in Figure 6, in comparison to the code in Figure 5, significantly improves the performance; however, the code in Figure 6 can still be improved. This is because the FPU computing the divide finishes its work in only 19 cycles, while the FPU computing the square root takes about 27 cycles. The next operation

```
do j = m,n                  ! the outer loop starts from
   a0 = a(0,j)              ! column j
   a2j2 = a(2,j−2)
   a0 = a0 − a2j2*a2j2
   a1 = a(1,j)              ! next four instructions
   a2 = a(2,j)              ! can be done
   a1j1 = a(1,j−1)          ! as two quad loads
   a2j1 = a(2,j−1)
   a1 = a1 − a1j1*a2j1
   a0 = a0 − a1j1*a1j1
   a0 = sqrt(a0)
   a(0,j) = a0
   t = 1./a0                ! this is the serial bottleneck
   a1 = a1*t                ! scale rest of the column
   a2 = a2*t                ! with the inverse of the
   a(1,j) = a1              ! diagonal element
   a(2,j) = a2
enddo
```

Figure 4

Example of matrix factorization without restructuring.

```
a1j1 = a(1,m−1)             ! pre-load a1j1 and a2j1
a2j1 = a(2,m−1)             !
do j = m,n                  ! the outer loop starts
   a0 = a(0,j)              ! from column j
   a2j2 = a(2,j−2)
   a0 = a0 − a2j2*a2j2
   a1 = a(1,j)
   a2 = a(2,j)
   a1 = a1 − a1j1*a2j1      ! note a1j1 and a2j1
   a0 = a0 − a1j1*a1j1      ! values are used
   a0 = sqrt(a0)            ! from the previous iteration
   a(0,j) = a0
   t = 1./a0                ! this is the serial bottleneck
   a1j1 = a1*t              ! scale rest of the column
   a2j1 = a2*t              ! with the inverse of the
   a(1,j) = a1j1            ! diagonal element
   a(2,j) = a2j1
enddo
```

Figure 5

Kernel of Figure 4 with reuse of values from prior iteration.

```
a1j1 = a(1,m-1)          ! pre-load a1j1 and a2j1
a2j1 = a(2,m-1)          !
do  j = m,n              ! the outer loop starts
    a0 = a(0,j)          ! from column j
    a2j2 = a(2,j-2)
    a0 = a0 - a2j2*a2j2
    a1 = a(1,j)
    a2 = a(2,j)
    a1 = a1 - a1j1*a2j1
    a0 = a0 - a1j1*a1j1
    ra0 = 1./a0          ! These two instructions
    a0 = sqrt(a0)        ! are computed in parallel
    a(0,j) = a0
    t = a0*ra0
    a1j1 = a1*t          ! scale rest of the column
    a2j1 = a2*t          ! with the inverse of the
    a(1,j) = a1j1        ! diagonal element
    a(2,j) = a2j1
enddo
```

Figure 6

Kernel of Figure 5 with restructured computation of t.

(t = a0*ra0) depends on both the results. Thus, one of the FPUs remains idle for about eight cycles. This inefficiency can also be avoided if we further restructure the computation. It can be shown that the next diagonal element of A can be computed using only the current divide operation and thus avoids using the current square-root operation. This can be accomplished by choosing a mathematically equivalent computation of the next diagonal element. To see this, consider the two-by-two symmetric matrix A and its Cholesky factor L:

$$A = \begin{bmatrix} a11 & a21 \\ a21 & a22 \end{bmatrix}, \quad L = \begin{bmatrix} l11 & \\ l21 & l22 \end{bmatrix}.$$

The required computations are given below:

```
l11 = sqrt(a11)
l21 = a21/l11
l22 = sqrt(a22 - l21*l21)
```

The above computation can also be rewritten as follows:

Table 1 Performance timing of the codes in Figures 4 through 7.

Code	*n*	*m*	*lda*	*Time* (s)	*Cycles per loop iteration*	*Computed cycles per loop iteration*
Figure 4	10000	2	3	0.009091	60.61	56.6
Figure 5	10000	2	3	0.007855	52.4	50.6
Figure 6	10000	2	3	0.005714	38.1	36.6
Figure 7	10000	2	3	0.004662	31.1	31.1

```
ra11 = 1./a11
l11  = sqrt(a11)
l21  = a21*ra11*l11
l22  = sqrt(a22 - a21*a21*ra11)
```

The above equations are mathematically equivalent. Note that the l22 computation can be initiated as soon as the ra11 computation is complete. Actually, it would be useful to have a reciprocal square-root (rsqrt) instruction. In that case, the following code could be implemented:

```
t   = rsqrt(a11)
l11 = t*a11
121 = a21*t
122 = sqrt(a22 - 121*121)
```

This code requires only one expensive (multicycle) intrinsic function. In terms of hardware complexity and/or the number of cycles required, a hardware implementation of the rsqrt instruction should be comparable to the sqrt instruction. However, it is currently not available on RS/6000 machines. We have just demonstrated another algorithmic computational restructuring technique for the two-by-two case. By incorporating these two ideas we can restructure the computation of Figure 5 as shown in **Figure 7**.

Since the ra0 computation is a divide, it takes about eight fewer cycles to complete than the sqrt operation. Thus, as soon as the ra0 computation is complete, we can initiate the computation of a0n (l22 in the equations above) of the next iteration, even before the final value of a0 of the current iteration has been computed! In this code, essentially the entire loop computation is overlapped with the sqrt computation. If we unroll the loop by two, we can achieve a slightly better scheduling of the instructions. Although we have illustrated the algorithmic restructuring technique for $m = 2$, it is applicable for even wider bands. For wider bands, where our code was for a general value of m, the above restructuring technique has provided a significant performance enhancement, for m up to 20 or so.

```
do j = m+2,n                    ! we are not giving the initialization
                                ! details (a0n, a1j1, a2j1 values set
                                ! outside the loop).
   ra0 = 1./a0n                 ! a0n represents the value of a(0,j)
                                ! just before the sqrt/divide operation.
                                ! It was computed in the previous iteration
                                ! ra0 computation is overlapped with
                                ! the sqrt computation below which was
                                ! initiated in the previous iteration.
   a1 = a(1,j)
   a2 = a(2,j)
   a1 = a1 - a1j1*a2j1
   a0n = a(0,j+1)               ! load a0n for the next iteration
   a0n = a0n - a2j1*a2j1        ! note that this a2j1 corresponds
                                ! to a2j2 in the code of Figure 5.
   t1 = a1*a1
   a0n = a0n - t1*ra0           ! a0n = a0n - a1*a1*ra0
                                ! now we need the result of the sqrt
                                ! computation initiated in the last iteration.
   t = a0*ra0
   a(0,j) = a0
   a0 = sqrt(a0n)
   a1j1 = a1*t                  ! scale rest of the column with
   a2j1 = a2*t                  ! the inverse of the diagonal element
   a(1,j) = a1j1
   a(2,j) = a2j1
enddo
```

Figure 7

Kernel of Figure 5 with restructuring of both square-root and divide computations.

We produced four codes based on Figures 4 through 7 and made timing runs for $n = 10000$. Although unrolling these loops by two helps the scheduling, we did not unroll the four codes we produced. In these timings we first flushed the cache. Thus, for each run, we had to bring 30000 doublewords into cache. Since each line contained 32 doublewords, 938 lines were brought in during each run. Assuming a latency of 17 cycles for each line to arrive accounts for 0.000239 seconds of time. This time amount corresponds to 1.6 cycles per loop iteration. The numbers in the column cycles per loop iteration are equal to time/$(n * CT)$, where CT is the cycle time of 15×10^{-9} seconds. The computed cycle time is obtained by hand-counting the number of cycles for both the fixed- and floating-pointing units for each of the codes in Figures 4 through 7. In each case, we added 1.6 cycles to the estimate to account for data coming from memory. In these hand calculations, we used the observed values of 19 cycles for a divide and 27 cycles for a square root. In the hand calculation for Figure 4, we did not add in any extra

cycles for the store load instruction sequence. As can be seen from **Table 1**, there is a surprisingly good agreement between the observed cycle counts and the hand-calculated cycle counts. The discrepancy of four cycles for Figure 4 is perhaps due to the penalty for the store/load sequence. Nonetheless, the main point is that algorithmic restructuring works extremely well in this case; the code in Figure 7 runs twice as fast as the code in Figure 4.

For narrow-band matrices, it is more efficient to use Gaussian $\mathbf{L}_1\mathbf{D}\mathbf{L}_1^T$ factorization in place of the $\mathbf{L}\mathbf{L}^T$ Cholesky factorization, where $\mathbf{L}_1$ is a unit lower triangular matrix and $\mathbf{D}$ is a diagonal matrix. The two factorizations are related by $\mathbf{L} = \mathbf{L}_1\mathbf{D}^{1/2}$. The Gaussian factorization does not require any square-root computation. For the narrow-band case, the techniques discussed above can be used to overlap the divide operation with the rest of the loop computation in the Gaussian factorization. For the very narrow-band case, the divide operation dominates the loop computation. Therefore, even though we can overlap the divide computation with the rest of the computation, the second FPU remains under-utilized. The only way we can increase performance in this case is to reduce the total number of divides. We can use the following idea* to reduce the number of divides. Let x = a*b and y = 1/x; then, 1/a = b*y and 1/b = a*y. The only potential problem with this approach is that there is a slight increase in the probability of underflow or overflow of exponents. To illustrate this technique in factoring symmetric positive definite matrices, consider a two-by-two diagonal block (A) of a symmetric band matrix and its Gaussian factor L1:

$$A = \begin{bmatrix} a11 & a21 \\ a21 & a22 \end{bmatrix}, \qquad L1 = \begin{bmatrix} d11 & \\ l21 & d22 \end{bmatrix}.$$

The required computations are given below:

```
d11 =a11
r1  =1/d11
l21 =a21*r1
d22 =a22 - l21*a21
r2  =1/d22
```

In the above computation, we wish to compute r1 and r2 using only one divide operation. This can be done by choosing x to be equal to the product of the two minors of A:

```
del = a22*a11 - a21*a21     ! second minor
x   = a11*del               ! product of the two minors
y   = 1/x                   ! only divide
r1  = y*del                 ! r1 = 1/d11
l21 = a21*r1
r2  = y*a11*a11             ! r2 = 1/d22
```

The above computations are mathematically equivalent. Using this technique, we have replaced two divides with one divide and a few multiply–adds. We have not explicitly computed the d22 term. This is because, on the diagonal, instead of storing the diagonal terms, we store their reciprocals. By storing the reciprocals, we avoid divides during the solve phase of the computation.

Summary

In this paper, we have described the novel architectural features of the POWER2 workstations. These features include multiple functional units, quad load/store instructions, and a very high-bandwidth memory system. If one uses the quad load capability of the POWER2 machines, a single FXU can feed both FPUs at the peak rate; therefore, the other FXU can be used to prefetch data into cache. Thus, the multiple functional units of POWER2 allow for the possibility of prefetching data into cache. In other words, POWER2 capabilities can be used to provide functional parallelism, if one develops high-performance numerical algorithms to do so. We have exploited functional parallelism by developing many highly tuned routines for ESSL. The three main techniques we used to exploit functional parallelism were loop unrolling, algorithmic prefetching, and algorithmic restructuring of the computation to serial bottlenecks. We have provided several examples of these techniques. For many BLAS-1 and BLAS-2 routines, on the same cycle time basis, we have demonstrated performance on POWER2 machines that is up to four times higher than that available using POWER. The quadword access facility, along with our use of algorithmic cache prefetching, was primarily responsible for this high level of performance. For BLAS-3 routines, the performance improvement over POWER is slightly more than a factor of two. The two FXUs and two FPUs are responsible for the factor of two. The significantly higher bandwidth of the memory system makes the improvement factor greater than two. Finally, using algorithmic restructuring of computation, we have also demonstrated a factor of two improvement for the Cholesky factorization of very narrow-band matrices.

Acknowledgments

We would like to thank several individuals who helped make this project a success. First we thank

* We heard about this idea from Jim Shearer of IBM Research. He said he heard about it from Don Coppersmith of IBM Research. Don thinks the idea is part of the folklore.

S. Bhattacharjee, Debasish Chakroborty, Prasad Palkar, and Anil Sharma of TISL and Jorge Lepre and Sylvia Rodriguez of IBM Argentina. These six talented people are responsible for implementing all of the level-three BLAS as well as most of the dense and banded linear algebra subroutines of ESSL. We thank Yogendra Singh of TISL and Luis Reyna and Mary Beth Miller for helping us locate and bring these people to Yorktown. We thank Richard Anderson, Benjamin Trimble, Bernie Rudin, and Randy Groves for recognizing and supporting an ESSL library tuned for POWER2 at an early stage. The ESSL team in Kingston, in particular Joan McComb, was very helpful to our development effort. Peg Cargiulo was helpful in getting source code to our workers, and in many other administrative matters. David Jensen was very helpful in providing support for our NIC POWER2 machine and backing up our code. We also thank Jacob Thomas of IBM Austin for providing us with access to and dedicated time on the Bheema POWER2 machine. We thank Robert Blainey of IBM Toronto for helping us understand the POWER2 optimization features of the new XLF compiler. We thank Larry Thatcher and Troy Hicks of IBM Austin for helping us understand the POWER2 hardware. Finally, we wish to thank our management, William Pulleyblank and Shmuel Winograd, for believing that we would succeed and for providing us with the funding to make this project a reality.

References

1. S. W. White and S. Dhawan, "POWER2: Next Generation of the RISC System/6000 Family," *IBM J. Res. Develop.* **38,** 493–502 (1994, this issue).
2. D. J. Shippy and T. W. Griffith, "POWER2 Fixed-Point, Data Cache, and Storage Control Units," *IBM J. Res. Develop.* **38,** 503–524 (1994, this issue).
3. T. N. Hicks, R. E. Fry, and P. E. Harvey, "POWER2 Floating-Point Unit: Architecture and Implementation," *IBM J. Res. Develop.* **38,** 525–536 (1994, this issue).
4. J. I. Barreh, R. T. Golla, L. B. Arimilli, and P. J. Jordan, "POWER2 Instruction Cache Unit," *IBM J. Res. Develop.* **38,** 537–544 (1994, this issue).
5. M. T. Vaden, L. J. Merkel, C. R. Moore, T. M. Potter, and R. J. Reese, "Design Considerations for the PowerPC 601 Microprocessor," *IBM J. Res. Develop.* **38,** 605–620 (1994, this issue).
6. T. B. Brodnax, R. V. Billings, S. C. Glenn, and P. T. Patel, "Implementation of the PowerPC 601 Microprocessor," *IBM J. Res. Develop.* **38,** 621–632 (1994, this issue).
7. *Engineering and Scientific Subroutine Library Version 2 Release 2, Guide and Reference*, Order No. SC23-0526-01, Volumes 1–3, 1994; available through IBM branch offices.
8. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User's Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.
9. E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK: A Portable Linear Algebra Library for High-Performance Computers" (LAPACK Working Note 20), *Computer Science Department Technical Report CS-90-105*, University of Tennessee, Knoxville, 1990.
10. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krough, "Basic Linear Algebra Subprograms for FORTRAN Usage," *ACM Trans. Math. Software* **5,** No. 3, 308–323 (1979).
11. J. J. Dongarra, J. DuCroz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Trans. Math. Software* **14,** No. 1, 1–17 (1988).
12. J. J. Dongarra, J. DuCroz, S. Hammarling, and R. J. Hanson, "Algorithm 656. An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs," *ACM Trans. Math. Software* **14,** No. 1, 18–32 (1988).
13. J. J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Software* **16,** No. 1, 1–17 (1990).
14. J. J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff, "Algorithm 679. A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs," *ACM Trans. Math. Software* **16,** No. 1, 18–28 (1990).
15. Special issue on the IBM RISC System/6000 Processor, *IBM J. Res. Develop.* **34**, No. 1 (January 1990).
16. R. J. Blainey, "Instruction Scheduling in the TOBEY Compiler," *IBM J. Res. Develop.* **38,** 577–593 (1994, this issue).
17. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "Improving Performance of Linear Algebra Algorithms for Dense Matrices, Using Algorithmic Prefetch," *IBM J. Res. Develop.* **38,** No. 3, 265–275 (May 1994).
18. J. J. Dongarra, F. G. Gustavson, and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Rev.* **26,** No. 1, 91–112 (1984).

Received September 28, 1993; accepted for publication March 23, 1994

Ramesh C. Agarwal *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (AGARWAL at YKTVMV, agarwal@watson.ibm.com).* Dr. Agarwal received a B.Tech. (Hons.) degree from the Indian Institute of Technology (IIT), Bombay. He was the recipient of The President of India Gold Medal while there. He received M.S. and Ph.D. degrees from Rice University and was awarded the Sigma Xi Award for best Ph.D. thesis in electrical engineering. He has been a member of the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center since 1983. Dr. Agarwal has done research in many areas of engineering, science, and mathematics and has published over 60 papers in various journals. Currently, his primary research interest is in the area of algorithms and architecture for high-performance computing on workstations and scalable parallel machines. In 1974, Dr. Agarwal received the Senior Award from the IEEE Acoustics, Speech, and Signal Processing (ASSP) group, for best papers. He has received several Outstanding Achievement Awards and a Corporate Award from IBM. Dr. Agarwal is a Fellow of the IEEE and a member of the IBM Academy of Technology.

Fred G. Gustavson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (GUSTAV at YKTVMV, gustav@watson.ibm.com).* Dr. Gustavson is manager of Algorithms and Architectures in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. in physics, and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute. He joined IBM Research in 1963. One of his primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. Dr. Gustavson has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for POWER2 for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Outstanding Invention Award, two IBM Outstanding Technical Achievement Awards, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award.

Mohammad Zubair *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (ZUBAIR at YKTVMV, zubair@watson.ibm.com).* Dr. Zubair received his Ph.D. degree in 1987 from the Indian Institute of Technology (IIT), New Delhi. From 1981 to 1987, he was at the Center for Applied Research in Electronics, IIT Delhi. In 1987, he became an Assistant Professor at Old Dominion University, Norfolk, VA, and in 1993 he became an Associate Professor. He joined IBM Research in 1994. Dr. Zubair's primary research interest is in the algorithm and architecture aspects of large-scale scientific computing. He has published over 30 papers in various journals and conference proceedings.