Algorithm and Architecture Aspects of Producing ESSL BLAS on POWER2

Contents

Ramesh C. Agarwal and Fred G. Gustavson

ESSL Overview

The Engineering Scientific Subroutine Library (ESSL) is a high-performanc e library of mathematical subroutines for RS/6000 workstations (ESSL/6000), as well as ES/9000 and ES/3090 vector and scalar mainframes (ESSL/370) [1]. Currently, ESSL V2.1 consists of 425 subroutines that cover the following computational areas: Linear Algebra Subprograms, Matrix Operations, Linear Algebraic Equations, Eigensystems Analysis, Fourier Transforms, Convolutions/Co rrelations and other related Signal Processing routines, Sorting and Searching, Interpolation, Numerical Quadrature, and Random Number Generation. ESSL also provides a small set of parallel routines for mainframes. Several versions of

most ESSL subroutines are provided. With few exceptions, these include short-precision and long-precision versions for both real and complex data types.

ESSL can be used for both developing and enabling many different types of scientific and engineering applications, as well as numerically intensive applications in other areas such as finance. Existing applications can be enabled by replacing comparable subroutines (or inline code) with calls to ESSL subroutines. The availability of a variety of ESSL subroutines significantly reduces the effort required in developing a new application. This also makes migration of an application to a new platform easier because ESSL is tuned for all platforms where it is available. Some of the industries that use these applications are: Aerospace, Automotive, Education, Electronics, Finance, Petroleum, Research, and Utilities. Examples of applications in these industries that can take advantage of ESSL subroutines are: Computational Chemistry, Computational Fluid Dynamics, Dynamic Systems Simulations, Electronic Circuit Design, Mathematical Analysis, Nuclear Engineering, Quantitative Analysis, Reservoir Modeling, Seismic Analysis, and Time Series Analysis.

ESSL/6000 provides a set of subroutines tuned for performance on the RS/6000 (RS/6000) workstations family, including both the older POWER workstations and the newer POWER2 workstations. These are predominantly compatible with the ESSL/370 product, allowing easy cross-platform migration between mainframes and workstations. All ESSL/6000 computational subroutines are written in Fortran and they are callable from application programs written in Fortran and C. All the subroutines are fully described in the Guide and Reference book [1].

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. These routines are functionally similar to the public domain software LAPACK [2 , 3], though not necessarily the same calling sequences are used. The user can modify the application to call these ESSL subroutines instead of LAPACK subroutines. This direct approach will provide the highest level of performance.

ESSL includes a complete set (140 routines) of tuned Basic Linear Algebra Subprograms (BLAS). BLAS are an industry-wide standard [4 , 6 , 8], providing a uniform functionality and call-sequence interface; an application using calls to BLAS is highly portable across high-performance platforms from different vendors. In LAPACK subroutines, most of the computing is done in these BLAS subroutines. LAPACK expects platform vendors to provide a

set of tuned BLAS to achieve high performance. As an alternative to calling ESSL routines directly, the user application can call LAPACK subroutines and link to ESSL for the tuned BLAS. This approach provides portability and performance. However, the performance for large problems achieved in this indirect approach is slightly lower than calling ESSL subroutines directly, but it is still very good.

In this paper, we describe our approach to producing highly-tuned BLAS on the POWER and POWER2 workstations. We describe the salient features of these two systems. We then describe the implementation of several tuned BLAS routines: DAXPY, DDOT, DGEMV, and DGEMM. For these examples, we use the LAPACK supplied BLAS as vanilla routines. We conclude with performance details for DGEMV and DGEMM. We obtained the performance data given in this paper from initial prototype routines; the final product version of ESSL may have slightly different performance characteristics.

Algorithm and Architecture Approach to Producing ESSL-Type Software

In this paper, we describe our approach to producing highly tuned BLAS on the POWER and POWER2 family of workstations. We start with a vanilla routine and restructure its computation, often significantly increasing the total code length, to produce a routine which is highly tuned for the underlying platform. In designing the tuned routine, we try to exploit the novel features of the POWER and POWER2 systems. However, while doing so, we must maintain identical functionality.

We will describe our approach as being *algorithmic* to indicate that changes are made to the source code of the original routine. In making changes to the vanilla source, we make heavy use of the law of linear superposition, which allows regrouping of the computations in many different ways. This enables techniques such as cache and TLB blocking as well as algori thmic *prefetching* [9]. In algorithmic prefetching, we modify the data access pattern so that an array that is likely to miss in cache is brought into cache before it is actually needed. This technique overlaps the memory access latency with useful work.

In contrast to our *algorithmic* approach, the literature frequently documents cache blocking or prefetching, or both, being performed

directly on the vanilla code during the compilation process. At times, this may require a modification to the architecture to support a cache prefetching instruction [11 , 13]. It is highly desirable to have automatic tools to enhance performance in the compilation of the original unmodified (vanilla) code. The Fortran compiler available for RS/6000 workstations has a preprocessing feature which performs automatic cache blocking. This is satisfactory in most simple situations and when the leading dimension of an array is favorable. In a multidimensional array, if some of the leading dimensions are powers of two (or related to them), then cache congruence class conflicts often reduce the effective cache size. Correctly determining when corrective actions, such as copying the data to a contiguous memory area, will be beneficial is beyond the capabilities of most automatic tools.

For the ESSL library routines which are expected to be used by many users over the product life cycle of a machine, we chose to invest an additional effort to hand tune the code and attempt to achieve the best possible level of performance in all situations. As an example, we were able to implement prefetching without requiring any special hardware assistance and by only using Fortran source code.

POWER2 CPU Considerations

POWER2 differs from POWER in many respects. POWER has one fixed-point unit (FXU) and one floating-point unit (FPU) and therefore can perform one floating-po int multiply-add instruction and one fixed-point instruction every cycle, if no dependencies exist. Additionally, branch instructions can overlap with other computations, effectively resulting in zero-cycle branches. The FXU handles all load/store instructions, including floating-point load/stores. Additionally, floating-point stores also require the FPU; therefore, these instructions and floating-point arithmetic instructions do not execute concurrently. A full description of a POWER machine can be found in [14].

POWER2 has two FXUs and two FPUs and therefore can perform two fixed-poin t instructions and two floating-point instructions every cycle, if no dependencies exist. Unlike the POWER design, floating-point store instructions can execute concurrently with floating-point arithmetic instructions on POWER2. To fully utilize the additional functional units of POWER2, an algorithm needs a higher degree of parallelism, often achieved by a higher degree of loop unrolling. Often the compiler can do this additional unrolling automatically, if the user selects an appropriate set of compiler options.

In most linear algebra applications, the basic computational kernel is either a DAXPY or an inner product (also called a dot product). A dot product formulation is more desirable because it requires fewer memory references. The dot product result is accumulated in a floating-point register (FPR), using a floating-point multiply-add/subtract instruction. Since floating-point instructions have a two-cycle pipeline delay, at least four independent floating-point instructions must be concurrently executing to keep both FPUs fully utilized. Thus, at least four independent dot products should be accumulated in four different FPRs. By comparison, POWER required only two independent dot products.

POWER2 enhancements also include floating-point quad-word load/store instructions. A quad-word load instruction can load two consecutive double-words of memory into two consecutive FPRs, and the quad-word store instruction can store two consecutive FPRs into two consecutive double-words of memory. Using both FXUs, this results in an effective bandwidth of 4 double-words per cycle between the cache and the FPRs. This factor-of-four increase in bandwidth over POWER implementations is particularly important for those computational kernels where the bandwidth between registers and cache limit performance. BLAS-1 (for example, DDOT and DAXPY) are examples of such kernels. For these kernels, on a per-cycle basis, POWER2 performs up to four times faster than POWER. However, generating quad-word load/store instructions is not always easy for the compiler; loops often require additional unrolling. Also quad-word load/stores require two consecutive FPRs; this restriction imposes additional constraints on the register assignment logic of the compiler.

Quad-word accesses that cross a cache line boundary require at least an additional cycle. When cache bandwidth limits performance, quad-word loads that span cache line boundaries will degrade performance. Special coding techniques may avoid such a situation. For example, assuming that all double-precision arrays are aligned on double-word boundaries, two possibilities exist: each array is either aligned on a quad-word boundary or an odd double-word boundary. If the array is aligned on a quad-word boundary, a quad-word load will never cross the cache line boundary. (For this discussion, we assume that an array is accessed in the loop with unit stride using quad-word load instructions.) This follows from the fact that all cache lines are quad-word aligned. If the array is aligned on an odd double-word boundary, one load is purposely performed outside the loop, thus making all quad-word loads inside the loop quad-word aligned. For double-precision, two-dimensional arrays, an even-valued leading dimension is suggested so that, if the first column is quad-word aligned, all other columns are also quad-word aligned. For multidimensional arrays, it is sufficient to have the first

leading dimension as even.

POWER2 Cache Considerations

POWER2 workstations have two different data cache line sizes: 128 and 256 bytes, depending on the amount of main memory [15]. We use a special subroutine, IRLINE, to determine the line size. This determination of cache size parameters is done only once, on the first call to the routine. The cache line size of the machine can be used to determine the cache size; cache size parameters are set for the particular POWER or POWER2 machine.

In this paper, we will refer to the data cache simply as cache. All POWER2 models have 1024 cache lines arranged in four-way associative sets. The cache capacity on the largest POWER2 model is four times as big as the largest POWER model. POWER2 models also have a significantly higher memory bandwidth; all POWER and POWER2 machines fetch a complete cache line in eight cycles after the first word of the line arrives.

All POWER2 workstations have a 512-entry, two-way set-associa tive translation look-aside buffer (TLB). This is significantly larger than the 128-entry TLB on POWER machines. Furthermore, TLB miss processing requires considerably fewer cycles on POWER2 systems than on POWER systems. This is significant for blocking considerations (of large problems) on POWER2; for most problems, the TLB size of POWER2 is not a limitation.

In computational kernels where array elements are used many times, appropriate cache and TLB blocking provide the best possible level of performance. For these kernels, such as BLAS-3 routines, the delay in accessing a cache line is not important because the cache miss delay is amortized over many uses. However, when arrays are used only once, as is the case for BLAS-1 and BLAS-2 routines, cache miss latency becomes an important consideration.

On POWER2 machines, a cache miss latency of 15-20 cycles occurs between when the originating load instruction reaches the FXU execute stage and when the desired data word first arrives in the cache. During the subsequent seven cycles, the remainder of the cache line arrives, starting with the data which follows the requested word and finishing with any remaining data at the beginning of the cache line.

Each of the two FXUs can process only one cache miss at a time. In

a typical stride-one access code, when a load results in a cache miss, a 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 may avoid this problem and improve memory system performance. This algorithmic *prefetching* significantly improves performance of some codes on POWER2 machines.

Algorithmic prefetching has been implemented for BLAS-2 for POWER systems [9]. Prefetching on POWER is difficult because POWER has only one FXU and a single cache miss will stall its pipeline. However, in those loops where arithmetic operations exceed load/store operations, specialized techniques can provide prefetch capability on POWER [9]. The main concept is to load FPRs with data at the beginning of the loop, prior to an "expected miss" load, so that when the cache miss stalls the FXU, the FPU is kept busy with useful work, using data already loaded in FPRs.

Implementing algorithmic *prefetching* is fairly straightforward for the stride-one situation on POWER2. Prefetching is easier in POWER2 because of multiple fixed-point units. Typically, the programmer can schedule a *dummy load* of an element from the next cache line several cycles before the data is needed. A dummy load is a load for which the data is not actually used in a computation. If the operand is already in cache, then the load completes in one cycle and the FXU is again available. On the other hand, if this load results in a cache miss, it will tie up one of the FXUs until the cache miss processing finishes. However, the other FXU is still available to perform the required (nondummy) loads. Many computing kernels require only one FXU to feed both FPUs when quad-word loads are exploited.

ESSL uses algorithmic prefetching on POWER and POWER2 in many computing kernels. Prefetching requires extensive loop unrolling and dummy loads, but it can be done in Fortran. Several different variations in prefetching exist; some are quite intricate and complex. If the cache bandwidth is a consideration, then the loop is unrolled in such a way that the data loaded during a dummy load is actually used in a later computation. Depending on the kernel and the number of arrays to be prefetched, this can become complicated, especially when using quad-word loads and considering relative quad-word alignments of different arrays. On the other hand, if the cache bandwidth is not a consideration (one FXU can handle all of the required loads), then a simple prefetching scheme using a dummy load may suffice. In this case, if the data is actually in cache, prefetching will not degrade performance. If the data is not in cache, prefetching will significantly improve performance.

BLAS-1 Implementation

Because there is very little reuse of data, the available bandwidth between the cache and the FPRs tends to limit BLAS-1 performance. Use of quad-word load/store instructions achieves maximum bandwidth. An optimal implementation tries to avoid crossing a cache line boundary on quad-word accesses. Here, we will describe implementation and performance of two key BLAS-1 kernels: DAXPY and DDOT. For this discussion, we are assuming that all arrays are accessed with unit stride; otherwise, use of quad-word load/store instructions is impossible. Opportunity for algorithmic prefetching is limited in BLAS-1.

DAXPY

DAXPY updates a vector *y* with the product of a scalar and another vector *x* . The scalar multiplier is loaded outside the loop and remains in a register throughout the computation. DAXPY requires two loads and one store for each multiply-add (FMA); therefore, available load-store bandwidth limits DAXPY performance.

Depending on the alignments of the *x and y* arrays, three possibilities exist. When both arrays are quad-word aligned, coding for optimal quad-word storage references is simple. If both arrays exhibit odd double-word alignment, then ESSL computes one element outside the loop; the remaining array references are now quad-word aligned. The difficult case arises when the alignment is even for one array and odd for the other.

In this case, we unroll the loop by four and access the first element of the odd-aligned array outside the loop, restructuring the computation so that all accesses inside the loop are quad-word aligned. The resulting loop executes in three cycles, achieving the peak bandwidth of two quad-word transfers, a total of 32 bytes, between cache and the FPRs every cycle.

On a 50-MHz prototype POWER2 system, the theoretical best DAXPY performance is 133 MFLOPS, 8 FLOPS every three cycles. When the data is in cache, we achieve close to 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, 2 FLOPS every three cycles. Dual units and quad-word storage references provide a factor-of-four improvement.

DDOT

The DDOT function computes the dot product of two vectors. Each FMA requires two loads; therefore, DDOT seems well-suited to the quad-word storage reference capabilities of POWER2. DDOT should run at the peak rate of 4 FLOPs per cycle, assuming that data is in cache. However, due to the register renaming implementation, the FXUs can not perform two Load Quads every cycle on a continuous basis. Eight quad-word loads in five cycles (80% of peak) is the limit. Thus, the DDOT limit on the 50-MHz prototype POWER2 is 160 MFLOPS (80% of peak), which we nearly achieve. Recall that in the DAXPY case, we achieve almost 100% of the peak POWER2 load-store bandwidth because there is a mix of both quad-word loads and stores.

We developed two versions of the DDOT function. One assumes the data is cache resident while the other assumes that the data is unlikely to be in cache. For the version which assumes data is in cache, we unrolled the loop by eight and accumulated partial sums in four temporary variables. We add the four sub-results together outside the loop. As in the DAXPY case, ESSL coding accounts for even-odd double-word alignments of both arrays. However, the cost of examining the alignments of both arrays and setting up the unrolling adds extra overhead to the subroutine, which is significant for small arrays. On a 50-MHz prototype machine, with a peak DDOT rate of 160 MFLOPS, we measured 148 MFLOPS for the four possible double-word alignments of the two arrays, for a size 2000 dot product (which fits in the cache).

We also developed a version of DDOT function which assumes that data is not in cache and includes algorithmic prefetching. For these routines, we unrolled the loop by 16 and included dummy loads that are 16 elements apart for both arrays.

We compare the two versions of DDOT using measurements on the prototype POWER2 machine (50-MHz and 256KB cache size) where the data is not in the cache. The prefetch version (which assumes data is not in cache) performs at about 103 MFLOPS, while the other subroutine (which assumes that the data is in cache) performs at about 74 MFLOPS. This represents a 40% performance improvement, due to algorithmic prefetching. However, the prefetch version of this subroutine requires extra (dummy) loads; when the data is actually in cache, its performance is about 10% less than the version optimized for cache resident data.

If the data loaded in registers (for prefetching) can actually be used in the computation, then we can optimize performance for cases where data is in cache as well as when data is not in cache. The coding becomes complicated when considering the relative quad-word alignments of the two arrays, as well as their relative

cache-line alignments. This extra logic adds overhead to the routine that is justifiable only for long arrays, say of the order of 2000 elements. We implemented one such version on the previously mentioned 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 either one or two vectors; the matrix elements are generally used only once. When the matrix is not in cache, optimal coding fully uses the matrix data brought into cache and simultaneously prefetches the next cache line.

Achieving significant reuse of register data and exploiting the POWER2 quad-word storage references are crucial steps toward minimizing the number of FXU load operations, and thus provide opportunities for scheduling (prefetch) cache misses during otherwise idle FXU cycles. We use register, cache, and TLB blocking for the matrix and the vectors to maximize reuse of the data. To illustrate the computational techniques, we will describe the matrix-vector multiplication subroutine DGEMV, where the matrix *A* is stored in the normal form (that is, column major order).

DGEMV - Normal Case

In DGEMV, a vector *x* is multiplied by a matrix A with the result being added to another vector y (y z y + A * *x)*. The most important consideration for DGEMV is cache prefetching; an optimal implementation is cache-line length specific. Here, we describe the implementation on machines with a 256-byte line size.

Without pre-fetching, the blocked code for DGEMV would resemble:

```
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 load
        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
```

The outermost blocking is across the columns of the matrix, which minimizes the finite cache and TLB effects. Within a vertical block, we unroll the computation by a large factor; we implement horizontal 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 double-words). However, because there are only 32 FPRs, we restrict the unrolling to 24. The innermost loop operates on the number of columns in a vertical block. Outside this loop, we load 24 elements (T0, T1, ..., T23) of the *y* vector into 24 FPRs. Within the loop, we process a subcolumn of the *A* matrix of size 24. Because of the cache blocking, we can assume that this block of the *A* matrix remains in cache.

Here *XJ* corresponds to a floating-point register, for which the reuse factor is 24. Note that, in the case of variables *F0 and F1* , one quad-word load fills two FPRs and hence feeds two FMA instructions. Thus, except for the initial load of *X(J),* the innermost loop consists of 24 FMAs and accesses for 24 matrix elements; the accesses can be performed as 12 quad-word loads. Thus, for 12 cycles, one FXU can feed data at a rate sufficient to sustain the FPU peak of 2 FMAs per cycle. The other FXU is free to prefetch data by executing a dummy load of an element from a column of *A* that is not likely to be in cache. ESSL prefetches data by inserting the following instruction in the inner loop:

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

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. By prefetching two columns ahead, our measurements show that the required data is brought into cache prior to its use.

If the leading dimension of the *A* matrix is even and the initial alignment of the matrix is on an odd double-word boundary, we process one row outside the main blocking loop, so that each sub-block in the main loop is aligned on a quad-word boundary. This ensures that none of the quad-word loads inside the inner loop cross a cache line boundary. If the leading dimension of the matrix

is odd, then for every other column, quad-word loads will cross the cache line boundary, slightly degrading the performance. Therefore, we recommend that multidimensional arrays have even-valued leading dimensions.

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

Figure 1 compares the performance of the ESSL version of DGEMV to that of the LAPACK vanilla routine. The figure plots performance data for square matrices of sizes 10 to 100 in steps of 5 and sizes 150 to 1000 in steps of 50. For values of N above 200, we see the effect of the matrices which do not reside in cache.

A BLAS-3 Routine - DGEMM

For BLAS-3 routines, appropriate cache and TLB blocking generally achieve the best possible level of performance. Blocking for different cache sizes is done at run time. We use the IRLINE subroutine to determine the line size.

For these kernels, the delay in accessing a cache line is not important since the data is used multiple times. DGEMM, which computes the product of two matrices, is typical of BLAS-3 routines. For DGEMM, fairly good performance can be obtained from the vanilla code, if appropriate cache-blocking preprocessing options are used at the compile time. Problems arise when the matrix dimensions are powers of two (or multiples thereof). In those cases, the effective cache size reduces because of cache congruence class conflicts. In this case, preprocessor cache blocking is not very effective.

ESSL BLAS-3 routines include cache and TLB blocking customized for the platform on which they are run. They provide robust performance in almost all situations. If necessary, ESSL copies sub-arrays into temporary buffers, eliminating problems due to poor leading dimensions. Because the algorithm uses the copied data many times, the cost of copying is insignificant. Because the arrays are blocked for cache, we can assume that data remains in cache; the only consideration is the innermost loop performance. For BLAS-3

kernels, the bandwidth between cache and FPRs is not a consideration, because ESSL unrolls the nested loops in a manner that achieves significant reuse of register data. The unrolling of loops also provides enough independent operations to fully utilize multiple functional units and to avoid FPU pipeline delays.

For the previous POWER release of ESSL, we implemented 2 by 2 unrolling (that is, in the innermost loop, a 2 by 2 block of the result matrix was computed). This is equivalent to computing four dot products in the innermost loop. This was sufficient to achieve the peak performance on POWER.

On POWER2, to assure robust performance with multiple functional units, we implement a 4 by 2 unrolling. This results in the peak performance at the innermost loop level for all combinations of matrix storage formats (by rows or columns). Because the 4 by 2 blocking is also optimal for POWER systems, this helps in producing a single source code for POWER and POWER2 machines. Distinct compilations for POWER and POWER2 machines can provide some performance opportunity for exploiting POWER2 quad-word load/store instructions that are not available on POWER.

Figure 2 shows DGEMM performance on two different POWER2 models and one POWER model. The figure plots performance data for square matrices of sizes 10 to 100 in steps of 5, sizes 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, due to our choice of 4 by 2 blocking. Note that even for matrix sizes as small as 20, performance reaches 200 MFLOPS on the 66.5-MHz POWER2 machine. The performance remains uniform, in the range of 90-95%

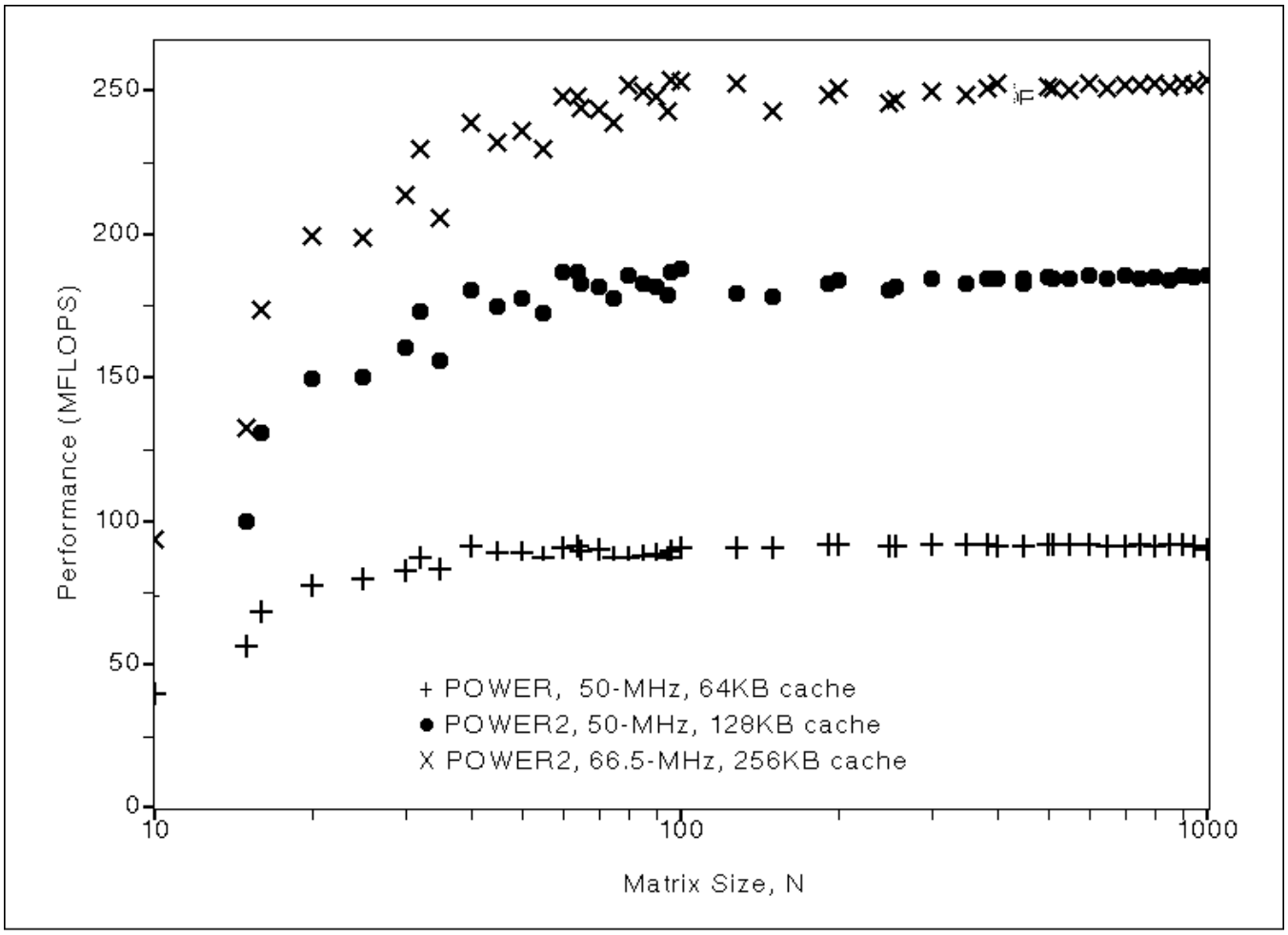

Figure 2 Performance of ESSL DGEMM on an RS/6000

Last modified: Fri Sep 20 9:03:03 US/Eastern 1996