DXML: A High-performance Scientific Subroutine Library

by

Chandrika Kamath, Roy Ho, and Dwight P. Manley

ABSTRACT

Mathematical subroutine libraries for science and engineering applications are an important tool in high-performance computing. By identifying and optimizing frequently used, numerically intensive operations, these libraries help in reducing the cost of computation, enhancing portability, and improving productivity. The Digital eXtended Math Library is a set of public domain and Digital proprietary software that has been optimized for high performance on Alpha systems. In this paper, DXML and the issues related to library software technology are described. Specific examples illustrate how algorithms can be optimized to take advantage of the architecture of Alpha systems. Modern algorithms that effectively exploit the memory hierarchy enable DXML routines to provide substantial improvements in performance.

INTRODUCTION

The Digital eXtended Math Library (DXML) is a set of mathematical subroutines, optimized for high performance on Alpha systems. These subroutines perform numerically intensive subtasks that occur frequently in scientific computing. They can therefore be used as building blocks for the optimization of various science and engineering applications in industries such as chemical, aerospace, petroleum, automotive, electronics, finance, and transportation.

In this paper, we discuss the role of mathematical software libraries, followed by an overview of the contents of the Digital eXtended Math Library. DXML includes optimized versions of both the standard BLAS and LAPACK libraries as well as libraries designed and developed by Digital for signal processing and the solution of sparse linear systems of equations. Next, we describe various aspects of library software technology, including the design and testing of DXML subroutines. Using key routines as examples, we illustrate the techniques used in the performance optimization of the library. Finally, we present data that demonstrates the performance improvement obtained through the use of DXML.

THE ROLE OF MATH LIBRARIES

Early mathematical libraries concentrated on supplementing the functionality provided by the Fortran compilers. In addition to routines such as sin and exp, which were included in the run-time math library, more complicated special functions, linear algebra algorithms, and Fourier transform algorithms were included in the software layer between the hardware and the user application.

Then, in the early 1970s, there was a concerted effort to produce high-quality numerical software, with the aim of providing end users with implementations of numerical algorithms that were stable, robust, and accurate. This led to the development of several math libraries, with the public domain LINPACK and EISPACK libraries for the solution of linear and eigen systems, setting the standards for future development of math software.[1-4]

The late 1970s and early 1980s saw the availability of advanced architectures, including vector and parallel computers, as well as high-performance workstations. This added another facet to the development of math libraries, namely, the implementation of algorithms for high efficiency on an underlying architecture.

The effort to produce mathematical software thus became a task of building bridges between numerical analysts, who devise algorithms, computer architects, who design high-performance computer systems, and computer users, who need efficient, reliable software for solving their problems. Consequently, these libraries embody expert knowledge in applied mathematics, numerical analysis, data structures, software engineering, compilers, operating systems, and computer architecture and are an important programming tool in the use of high-performance computers.

Modern superscalar RISC architectures with floating-point pipelines, such as the Alpha, have deep memory hierarchies. These include floating-point registers, multiple levels of caches, and virtual memory. The significant latency and bandwidth differences between these memory levels imply that numerical algorithms have to be restructured to make effective use of the data brought into any one level. The performance of an algorithm is also susceptible to the order in which computations are scheduled as well as the higher cost associated with some operations such as floating-point square-root and division.

The architecture of the Alpha systems and the technology of the Fortran and C compilers usually provide an efficient computing environment with adequate performance. However, there is often room for improvement, especially in engineering and science applications, where vast amounts of data are processed and repeated operations are performed on each data element. One way to achieve these improvements is through the use of optimized subroutine libraries.

The Digital eXtended Math Library is a collection of routines that performs frequently occurring, numerically intensive operations. By identifying such operations and optimizing them for high performance on Alpha systems, DXML provides several benefits to the computational scientist.

- It allows definition of functions at a sufficiently high level and therefore optimization beyond the capabilities of the compiler.
- It makes the architecture of the systems more transparent to the user.
- It improves productivity by providing easy access to highly optimized, efficient code.
- It enhances the portability of user software through the support of standard libraries and interfaces.
- It promotes good software engineering practice and avoids duplication of work by identifying and optimizing common functions across several application areas.

OVERVIEW OF DXML

DXML contains almost 400 user-callable routines, optimized for Alpha systems.[5] This includes both software developed by Digital as well as the BLAS and LAPACK libraries. Most routines are available in four versions: real single precision, real double precision, complex single precision, and complex double precision.

DXML is available on both OpenVMS and DEC OSF/1 operating systems. Its routines can be called from either Fortran or C, provided the difference in array storage between these languages is taken into account. DXML is available as a shareable library, with a simple interface, enabling easy access to the routines. On DEC OSF/1 systems, DXML supports the IEEE floating-point format. On OpenVMS systems, either the IEEE floating-point format or the VAX F-float/G-float format can be selected.

DXML routines can be broadly categorized into the following four areas:

- BLAS. The Basic Linear Algebra Subroutines include the standard BLAS and Digital enhancements.
- LAPACK. The Linear Algebra PACKage includes linear and eigen-system solvers.
- Signal processing. This includes fast Fourier transforms (FFTs), convolution, and correlation.

- o Sparse linear system solvers. These include direct and iterative solvers.

Of these, the signal-processing library and the sparse linear system solvers are designed, developed, and optimized by Digital. The majority of the BLAS routines and the LAPACK library are versions of the public domain standard that were optimized for the Alpha architecture. By supporting the industry standard interfaces of these libraries, DXML provides both portability of user code and high performance of the optimized software.

We next provide a brief description of the functionality provided by each subcomponent of DXML. Further details are available in the Digital eXtended Math Library Reference Manual.[5]

VLIB

The vector library consists of seven double-precision routines that perform operations such as sine, cosine, and natural logarithm, on data stored as vectors.

BLAS 1

The Basic Linear Algebra level 1 subprograms perform low-granularity operations on vectors that involve one or two vectors as input and return either a vector or a scalar as output.[6] Examples of BLAS 1 routines include dot product, index of the maximum element in a vector, and so on.

BLAS 1 Extensions (BLAS 1E)

Digital has extended the functionality of the BLAS 1 routines by including 13 similar operations. These include index of the minimum element of a vector, sum of the elements of a vector, and so on.

BLAS 1 Sparse (BLAS 1S)

DXML also includes nine routines that are sparse extensions of the BLAS 1 routines. Of these, six are from the sparse BLAS 1 standard and three are enhancements.[7] These routines operate on two vectors, one of which is sparse and stored in a compressed form. As most of the elements in a sparse vector are zero, both computational time and memory are reduced by storing and operating on only the nonzeros. BLAS 1S routines include construction of a sparse vector from the specified elements of a dense vector, dot product, and so on.

BLAS 2

The BLAS level 2 routines perform operations of a higher granularity than the level 1 routines.[8] These include matrix-vector operations such as matrix-vector product, rank-one and rank-two updates, and solutions of triangular systems of equations. Various storage schemes are supported, including general, symmetric, banded, and packed.

BLAS 3

The BLAS level 3 routines perform matrix-matrix operations, which are of a higher granularity than the BLAS 2 operations. These routines include matrix-matrix product, rank-k updates, solution of triangular systems with multiple right-hand sides, and multiplication of a matrix by a triangular matrix. Where appropriate, these operations are defined for matrices that may be general, symmetric, or triangular.[9] The functionality of the public domain BLAS 3 library has been enhanced by three additional routines for matrix addition, subtraction, and transpose.

LAPACK

DXML includes the standard Linear Algebra PACKage, LAPACK, which supercedes the LINPACK and EISPACK packages by extending the functionality, using algorithms with higher accuracy, and improving the performance through the use of the optimized BLAS library.[10] LAPACK can be used for solving many common linear algebra problems, including solution of linear systems, linear least-squares problems, eigenvalue problems, and singular value problems. Various storage schemes are supported, including general, band, tridiagonal, symmetric positive definite, and so on.

Signal Processing

The signal-processing subcomponent of DXML includes FFTs, convolutions, and correlations. A comprehensive set of Fourier transforms is provided, including

- FFTs in one, two, and three dimensions
- FFTs in forward and inverse directions
- Multiple one-dimensional transforms

There is no limit on the number of elements being transformed, though the performance is best when the data length is a power of 2. Popular storage formats for the input and output data are supported, allowing for possible symmetry in the output data and consequent reduction in the storage required. Further efficiency is provided through the use of the three-step FFT, which separates the process of setting up and deallocating the internal data structures from the actual application of the FFT. This results in significant performance gain when repeated application of FFTs is required.

The convolution and correlation routines in DXML support both periodic (circular) and nonperiodic (linear) definition. A discrete summing technique is used for calculation. Special versions of the routines allow control of output options such as the range of coefficients computed, scaling of the output, and addition of the output to an array.

All FFT, convolution, and correlation routines are available in both single and double precision and support both real and complex data.

Sparse Iterative Solvers

DXML includes a set of routines for the iterative solution of sparse linear systems of equations using preconditioned, conjugate-gradient-like methods.[11,12] A flexible user interface, based on a matrix-free formulation of the solver, allows a choice among various solvers, storage schemes, and preconditioners. This formulation permits the user to define his or her own preconditioner and/or storage scheme for the matrix. It also allows the user to store the matrix using one of the storage schemes defined by DXML and/or use the preconditioners provided. A driver routine provides a simple interface to the iterative solvers when the DXML storage schemes and preconditioners are used.

The different iterative methods provided are (1) conjugate gradient, (2) least-squares conjugate gradient, (3) biconjugate gradient, (4) conjugate-gradient squared, and (5) generalized minimum residual. Each method supports various applications of the preconditioner: left, right, split, and no preconditioning.

The matrix can be stored in the symmetric diagonal storage scheme, the unsymmetric diagonal storage scheme or the general storage (by rows) scheme. Three preconditioners are provided for each storage scheme: diagonal, polynomial (Neumann), and incomplete LU with zero diagonals added.

A choice of four stopping criteria is provided, in addition to a user-defined stopping criterion. The iteration process can be controlled by setting various input parameters such as the maximum number of iterations, the degree of polynomial preconditioning, the level of output provided, and the tolerance for convergence. These solvers are available in real double precision only.

Sparse Skyline Solvers

The sparse skyline solver library in DXML includes a set of routines for the direct solution of a sparse linear system of equations with the matrix stored using the skyline storage scheme.[13,14] The following functions are provided.

- LDU factorization, which includes options for the evaluation of the determinant and inertia, partial factorization, statistics on the matrix, and options for handling small pivots.

- Solve, which includes multiple right-hand sides and solves systems involving either the matrix or its transpose.

- Norm evaluation, including 1-norm, infinity-norm, Frobenius norm, and the maximum absolute value of the matrix.

- Condition number estimation, which includes both the 1-norm and the infinity norm.

- Iterative refinement, including the component-wise relative backward error and the estimated forward error bound for each solution vector.

- Simple and expert drivers.

This functionality is provided for each of the following storage schemes:

- For symmetric matrices:
    - Profile-in storage mode
    - Diagonal-out storage mode

- For unsymmetric matrices:
    - Profile-in storage mode
    - Diagonal-out storage mode
    - Structurally symmetric profile-in storage mode

These solvers are available in real double precision only.

SOFTWARE CONSIDERATIONS

As with any software effort, many software engineering issues were encountered during the design and development of DXML. Some issues were specific to math libraries such as the numerical accuracy and stability of the routines, while others were more general such as the design of a user interface, testing of the software, error checking, ease of use, and portability. We next discuss some of these key design issues in further detail.

Designing the Interface

The first task in creating a library was to decide the functionality, followed by the design of the interface. This included both the naming of the subroutines as well as the design of the parameter list. For each subcomponent in DXML, the calling sequence was designed to be consistent across all routines in that subcomponent. In the case of the BLAS and LAPACK libraries, the public domain interface was maintained to enable portability of user code.

For the routines added by Digital, the routine names were chosen to indicate the function being performed as well as the precision of the data. Furthermore, the parameter lists were chosen to provide a simple interface, yet allow flexibility for the sophisticated user. For example, the sparse solvers require various real and integer parameters. By using arrays instead of scalar variables, a more concise interface that did not vary from routine to routine was obtained. In addition, all solver routines have arguments for real and integer work arrays, even if these are not used in the code. This not only provides a uniform interface but also acts as a placeholder for work arrays, should they be required in the future.

Accuracy

The numerical accuracy of the routines in DXML is dependent on the problem size as well as the algorithm used, which may vary within a routine. Since performance optimization often changes the order in which a computation is performed, identical results between the DXML routines and the public domain BLAS and LAPACK routines may not occur. The accuracy of the results obtained is checked by ensuring that the optimized versions of the BLAS and LAPACK routines pass the public domain tests to within the specified tolerance.

Error Processing

Most of the routines in DXML trap usage errors and provide sufficient information so that the user can identify and fix the problem. The low-level, fine-grained computational routines, such as the BLAS level 1, do not provide this function because the overhead of testing and error trapping would seriously degrade the performance.

In the case of BLAS 2, BLAS 3, and LAPACK, the public domain error-reporting mechanism has been maintained. If an input argument is invalid, such as a negative value for the order of the matrix, the routine prints out an error message and stops. If a failure occurs in the course of the algorithm, such as a matrix being singular to working precision, an error flag is set and control is returned to the calling program.

The signal-processing routines report success or failure using a status function value. Further information on the error can be obtained by using a user-callable routine that prints out an error message and an error flag. The user documentation indicates the actions to be taken to recover from the error.

In the case of the sparse solvers, error is indicated by setting an error flag and printing an appropriate message if the printing option is enabled. Control is always returned to the calling program.

Testing

DXML routines are tested for correctness and accuracy using a regression test suite. This includes both test code developed by Digital, as well as the public domain test codes for BLAS and LAPACK. These codes are used not only during the implementation and performance optimization of the routines, but also during the building of the complete library from each of the subcomponents.

The test codes check each routine extensively, including checks for error exits, accuracy of the results obtained, invariance of read-only data and the correctness of all paths through the code. As the complete regression tests take over 20 hours to execute, two input data sets are used: a short one that tests each routine and can be used to make a quick check that all subcomponents compiled and built correctly, and a long data set that tests each path through a routine and is thus more exhaustive.

Many of the routines, such as the FFTs and BLAS 3, are tested using random input data. However, some routines, such as the sparse solvers, operate on specific data structures or matrices with specific properties. These have been tested using matrices generated from the finite difference discretization of partial differential equations or using the matrices in the Harwell-Boeing test suite.[15]

Another aspect to the DXML regression test package is the inclusion of a performance test gauge. This software tests the performance of key routines in each component of DXML and is used to ensure that the performance of DXML routines is not adversely affected by changes in compilers or the operating systems.

Performance Trade-offs

The design and optimization of the routines in DXML often prompted a trade-off between performance on one hand, and accuracy and generality on the other. Although every effort has been made not to sacrifice accuracy for performance, the reordering of computations during performance optimization may lead to results before optimization that are not bit-for-bit identical to the results after optimization. In other cases, performance has been sacrificed to ensure generality of a routine. For example, although the matrix-free formulation of the iterative solvers permits the use of any sparse matrix storage scheme, it could result in a slight degradation in performance due to less efficient use of the instruction cache and the inability to reuse some of the data in the registers.

PERFORMANCE OPTIMIZATION

DXML routines have been designed to provide high performance on the Alpha systems.[16] These routines are tailored to take advantage of the system characteristics such as the number of floating-point registers, the size of the primary and secondary data caches, and the page size. This optimization involves changes to data structures and the use of new algorithms as well as the restructuring of computation to effectively manage the memory hierarchy.

Several general techniques are used across all DXML subcomponents to improve the performance.[17] These include the following techniques:

- Unrolling loops to make better use of the floating-point pipelines
- Reusing data in registers and caches whenever possible
- Managing the data caches effectively so that the cache hit ratio is maximized
- Accessing data using stride-1 computation
- Using algorithms that exploit the memory hierarchy effectively
- Reordering computations to minimize cache and translation buffer thrashing

Although many of these optimizations are done by the compiler, occasionally, for example in the case of the skyline solver, the data structures or the implementation of the algorithm are such that they do not lend themselves to optimization by the compiler. In these cases, explicit reordering of the computations is required.

We next discuss these optimization techniques as used in specific examples. All performance data is for the DEC 3000 Model 900 system using the DEC OSF/1 version 3.0 operating system. This workstation uses the Alpha 21064A chip, running at 275 megahertz (MHz). The on-chip data and instruction caches are each 16 kilobytes (KB) in size, and the secondary cache is 2 megabytes (MB) in size.

In the next section, we compare the performance of DXML BLAS and LAPACK routines with the corresponding public domain routines. Both versions are written in standard Fortran and compiled using identical compiler options.

Optimization of BLAS 1

BLAS 1 routines operate on vector and scalar data only. As the operations and data structures are simple, there is little opportunity to use advanced data blocking and register reuse techniques. Nevertheless, as the plots in Figure 1 demonstrate, it is possible to optimize the BLAS 1 routines by careful coding that takes advantage of the data prefetch features of the Alpha 21064A chip and avoids data-path-related stalls.[16,18]

Generally, the DXML routines are 10 percent to 15 percent faster than the corresponding public domain routines. Occasionally, as in the case of DDOT for very short, cache-resident vectors, the benefits can be much greater.

The shapes of the plots in Figure 1 rather dramatically demonstrate the benefits of data caches. Each plot shows very high performance for short vectors that reside in the 16-KB, on-chip data cache, much lower performance for data vectors that reside in the 2-MB, on-board secondary data cache, and even lower performance when the vectors reside completely in memory.

[Figure 1 (Performance of BLAS 1 Routines DDOT and DAXPY) is not available in ASCII format.]

Optimization of BLAS 2

BLAS 2 routines operate on matrix, vector, and scalar data. The data structures are larger and more complex than the BLAS 1 data structures and the operations more complicated. Accordingly, these routines lend themselves to more sophisticated optimization techniques.

Optimized DXML BLAS 2 routines are typically 20 percent to 100 percent faster than the public domain routines. Figure 2 illustrates this performance improvement for the matrix-vector multiply routine, DGEMV, and the triangular solve routine, DTRSV.[8]

[Figure 2 (Performance of BLAS 2 Routines DGEMV and DTRSV) is not available in ASCII format.]

The DXML DGEMV uses a data-blocking technique that asymptotically performs two floating-point operations for each memory access, compared to the public domain version, which performs two floating-point operations for every three memory accesses.[19] This technique is designed to minimize translation buffer and data cache misses and maximize the use of floating-point registers.[16,18,20] The same data prefetch considerations used on the BLAS 1 routines are also used on the BLAS 2 routines.

The DXML version of the DTRSV routine partitions the problem such that a small triangular solve operation is performed. The result of this solve operation is then used in a DGEMV operation to update the remainder of the vector. The process is repeated until the final triangular update completes the operation. Thus the DTRSV routine relies heavily on the optimizations used in the DGEMV routine.

As with BLAS 1 routines, BLAS 2 routines benefit greatly from data cache. Although the effect is less dramatic for the BLAS 2 routines, Figure 2 clearly shows the three-step profile observed in Figure 1. Best performance is achieved when both matrix and vector fit in the primary cache. Performance is lower but flat over the region where the data fits on the secondary board level cache. The final performance plateau is reached when data resides entirely in memory.

Optimization of BLAS 3

BLAS 3 routines operate primarily on matrices. The operations and data structures are more complicated that those of BLAS 1 and BLAS 2 routines. Typically, BLAS 3 routines perform many computations on each data element. These routines exhibit a great deal of data reuse and thus naturally lend themselves to sophisticated optimization techniques.

DXML BLAS 3 routines are generally two to ten times faster than their public domain counterparts. The plots in Figure 3 show these performance differences for the matrix-matrix multiply routine, DGEMM, and the triangular solve routine with multiple right-hand sides, DTRSM.[9]

[Figure 3 (Performance of BLAS 3 Routines DGEMM and DTRSM) is not available in ASCII format.]

All performance optimization techniques used for the DXML BLAS 1 and BLAS 2 routines are used on the DXML BLAS 3 routines. In particular, data-blocking techniques are used extensively. Portions of matrices are copied to page-aligned work areas where secondary cache and translation buffer misses are eliminated and primary cache misses are absolutely minimized.

As an example, within the primary compute loop of the DXML DGEMM routine, there are no translation buffer misses, no secondary cache misses, and, on average, only one primary cache miss for every 42 floating-point operations. Performance within this key loop is also enhanced by carefully using floating-point registers so that four floating-point operations are performed for each memory read access. Much of the DXML BLAS 3 performance advantage over the public domain routines is a direct consequence of a greatly improved ratio of floating-point operations per memory access.

The DXML DTRSM routine is optimized in a manner similar to its BLAS 2 counterpart, DTRSV. A small triangular system is solved. The resulting matrix is then used by DGEMM to update the remainder of the right-hand-side matrix. Consequently, most of the DXML DTRSM performance is directly attributable to the DXML DGEMM routine. In fact, the techniques used in DGEMM pervade DXML BLAS 3 routines.

Figure 3 illustrates a key feature of DXML BLAS 3 routines. Whereas the performance of public domain routines degrades significantly as the matrices become too large to fit in caches, DXML routines are relatively insensitive to array size, shape, or orientation.[5,9] The performance of a DXML BLAS 3 routine typically reaches an asymptote and remains there regardless of problem size.

Optimization of LAPACK

The LAPACK subroutine library derives a large part of its high performance by using the optimized BLAS as building blocks.[10] The DXML version of LAPACK is largely unmodified from the public domain version. However, in the case of the factorization routine for general matrices, DGETRF, we have introduced changes to the algorithm to improve the performance on Alpha systems.

For example, while the original public domain DGETRF routine uses Crout's method to factor a matrix, the DXML version uses a left-looking method.[11] Left-looking methods make better use of the secondary cache and translation buffers than the Crout method. Furthermore, the public domain version of the DLASWP routine swaps a single matrix row across an entire matrix. This is a very bad technique for RISC machines; it causes severe cache and translation buffer thrashing. To avoid this, the DXML version of DLASWP performs all swaps within columns, which makes much better use of the caches and the translation buffer and results in a much improved performance of the DXML DGETRF routine.

The DGETRS routine was not modified. Its performance is solely attributable to use of optimized DXML routines.

Figure 4 shows the benefits of the optimizations made to DGETRF and the BLAS routines. DGETRF makes extensive use of the BLAS 3 DGEMM and DTRSM routines. The performance of DXML DGETRF improves with increasing problem size largely because DXML BLAS 3 routines do not degrade in the face of larger problems.

The plots of Figure 4 also show the performance of DGETRS when processing a single right-hand-side vector. In this case, DTRSV is the dominant BLAS routine, and the performance differences between the public domain and DXML DGETRS routines reflect the performance of the respective DTRSV routines. Finally, although not shown, we note that the performance of DXML DGETRS is much better than the public domain version when many right-hand sides are used and DTRSM becomes the dominant BLAS routine.

[Figure 4 Performance of LAPACK Routines DGETRF and DGETRS (LDA = N+1) is not available in ASCII format.]

Optimization of the Signal-processing Routines

We illustrate the techniques used in optimizing the signal-processing routines using the one-dimensional, power-of-2, complex FFT.[21] The algorithm used is a version of Stockham's autosorting algorithm, which was originally designed for vector computers but works well, with a few modifications, on a RISC architecture such as Alpha.[22,23]

The main advantage in using an autosorting algorithm is that it avoids the initial bit-reversal permutation stage characteristic of the Cooley-Tukey algorithm or the Sande-Tukey algorithm. This stage is implemented by either precalculating and loading the permutation indices or calculating them on-the-fly. In either case, substantial amounts of integer multiplications are needed. By avoiding these multiplications, the autosorting algorithm provides better performance on Alpha systems.

This algorithm does have the disadvantage that it cannot be done in-place, resulting in the use of a temporary work space, which makes more demands on the cache than an algorithm that can be done in-place. However, this disadvantage is more than offset by the avoidance of the bit-reversal stage.

The implementation of the FFT on the Alpha makes effective use of the hierarchical memory of the system, specifically, the 31 usable floating-point registers, which are at the lowest, and therefore the fastest, level of this hierarchy. These registers are utilized as much as possible, and any data brought into these registers is reused to the extent possible. To accomplish this, the FFT routines implement the largest radices possible for all stages of the power-of-2 FFT calculation. Radix-8 was used for all stages except the first, utilizing 16 registers for the data and 14 for the twiddle factors.[21] For the first stage, as all twiddle factors are 1, radix-16 was used.

Figure 5 illustrates the performance of this algorithm for various sizes. Although the performance is very good for small data sizes that fit into the primary, 16-KB data cache, it drops off quickly as the data exceeds the primary cache. To remedy this, a blocking algorithm was used to better utilize the primary cache.

[Figure 5 (Performance of 1-D Complex FFT) is not available in ASCII format.]

The blocking algorithm, which was developed for computers with hierarchical memory systems, decomposes a large FFT into two sets of smaller FFTs.[24] The algorithm is implemented using four steps:

1. Compute N1 sets of FFTs of size N2

2. Apply twiddle factors

3. Compute N2 sets of FFTs of size N1

4. Transpose the N1 by N2 matrix into an N2 by N1 matrix

In the above, N = N1 X N2. Steps (1) and (3), use the autosorting algorithm for small sizes. In step (2), instead of precomputing all N twiddle factors, a table of selected twiddle factors is computed and the rest calculated using trigonometric identities.

Figure 5 compares the performance of the blocking algorithm with the autosorting algorithm. Due to the added cost of steps (2) and (4), the maximum computation speed for the blocking algorithm (115 million floating-point operations per second [Mflops] at N=2**12) is lower than the maximum computation speed of the autosorting algorithm (192 Mflops at N = 2**9). The crossover point between the two algorithms is at a size of approximately 2K, with the autosorting algorithm performing better at smaller sizes. Based on the length of the FFT, the DXML routine automatically picks the faster algorithm. Note that at N=2**16, as the size of the data and workspace exceeds the 2-MB secondary cache, the performance of the blocking algorithm drops off.

Optimization of the Skyline Solvers

A skyline matrix (Figure 6) is one where only the elements within the envelope of the sparse matrix are stored. This storage scheme exploits the fact that zeros that occur before the first nonzero element in a row or column of the matrix, remain zero during the factorization of the matrix, provided no row or column interchanges are made.[14] Thus, by storing the envelope of the matrix, no additional storage is required for the fill-in that occurs during the factorization. Though the skyline storage scheme does not exploit the sparsity within the envelope, it allows for a static data structure, and is therefore a reasonable compromise between organizational simplicity and computational efficiency.

[Figure 6 (Skyline Column Storage of a Symmetric Matrix) is not available in ASCII format.]

In the skyline solver, the system, Ax = b, where A is an N by N matrix, and b and x are N-vectors, is solved by first factorizing A as A = LDU, where L and U are unit lower and upper triangular matrices, and D is a diagonal matrix. The solution x is then calculated by solving in order, Ly = b, Dz = y, and Ux = z, where y and z are N-vectors.

In our discussion of performance optimization, we concentrate on the factorization routine as it is often the most time-consuming part of an application. The algorithm implemented in DXML uses a technique that generates a column (or row) of the factorization using an inner product formulation. Specifically, for a symmetric matrix A, let

[Equation 1 is not available in ASCII format.]

where the symmetric factorization of the leading (N-1) by (N-1) leading principal submatrix M has already been obtained as

[Equation 2 is not available in ASCII format.]

Since the vector v, of length (N-1), and the scalar s are known, the vector w, of length (N-1) and the scalar d can be determined as

[Equation 3 is not available in ASCII format.]

and

[Equation 4 is not available in ASCII format.]

The definition of w indicates that a column of the factorization is obtained by taking the inner product of the appropriate segment of that column with one of the previous columns that has already been calculated. Referring to Figure 7, the value of the element in location (i,j) is calculated by taking the inner product of the elements in column j above the element in location (i,j) with the corresponding elements in column i. The entire column j is thus calculated starting with the first nonzero element in the column and moving down to the diagonal entry.

[Figure 7 (Unoptimized Skyline Computational Kernel) is not available in ASCII format.]

The optimization of the skyline factorization is based on the following two observations [25,26]:

- The elements of column j, used in the evaluation of the element in location (i,j), are also used in the evaluation of the element in location (i+1,j).

- The elements of column i, used in the evaluation of the element in location (i,j), are also used in the evaluation of the element in location (i,j+1).

Therefore, by unrolling both the inner loop on i and the outer loop on j, twice, we can generate the entries in locations (i,j), (i+1,j), (i,j+1), (i+1,j+1) at the same time, as shown in Figure 8. These four elements are generated using only half the memory references made by the standard algorithm. The memory references can be reduced further by increasing the level of unrolling. This is, however, limited by two factors:

- The number of floating-point registers required to store the elements being calculated and the elements in the columns.

- The length of consecutive columns in the matrix, which should be close to each other to derive full benefit from the unrolling.

Based on these factors, we have unrolled to a depth of 4, generating 16 elements at a time.

A similar technique is used in optimizing the forward elimination and the backward substitution.

[Figure 8 (Optimized Skyline Computational Kernel) is not available in ASCII format.]

Table 1 gives the performance improvements obtained with the above techniques for a symmetric and an unsymmetric matrix from the Harwell-Boeing collection.[15] The characteristics of the matrix are generated using DXML routines and were included because the performance is dependent on the profile of the skyline. The data presented is for a single right-hand side, which has been generated using a known random solution vector.

The results show that for the matrices under consideration, the technique of reducing memory references by unrolling loops at two levels leads to a factor of 2 improvement in performance.

Table 1 Performance Improvement in the Solution of Ax = b, Using the Skyline Solver on the DEC 3000 Model 900 System

	Example 1	Example 2
Harwell-Boeing matrix[15]	BCSSTK24	ORSREG1
Description	Stiffness matrix of the Calgary Olympic	Jacobian from a model of an oil

	Saddledome Arena	reservoir
Storage scheme	Symmetric diagonal-out	Unsymmetric profile-in
Matrix characteristics		
Order	3562	2205
Type	Symmetric	Unsymmetric with structural symmetry
Condition number estimate	6.37E+11	1.54E+4
Number of nonzeros	81736	14133
Size of skyline	2031722	1575733
Sparsity of skyline	95.98%	99.10%
Maximum row (column) height	3334	442 (442)
Average row (column) height	570.39	357.81 (357.81)
RMS row (column) height	1135.69	395.45 (395.45)
Factorization time (in seconds)		
Before optimization	66.80	23.12
After optimization	35.02	13.02
Solution time (in seconds)		
Before optimization	0.82	0.32
After optimization	0.43	0.17
Maximum component-wise relative error in solution (See equation below.)	0.16E-5	0.50E-10

$$\max_i \frac{| x(i) - \bar{x}(i) |}{| x(i) |}$$

, where x(i) is the i-th component of the true solution, and $\bar{x}(i)$ is the i-th component of the calculated solution.

SUMMARY

In this paper, we have shown that optimized mathematical subroutine libraries can be a useful tool in improving the performance of science and engineering applications on Alpha systems. We have described the functionality provided by DXML, discussed various software engineering issues and illustrated techniques used in performance optimization.

Future enhancements to DXML include symmetric multiprocessing support for key routines, enhancements in the areas of signal processing and sparse solvers, as well as further optimization of routines as warranted by changes in hardware and system software.

ACKNOWLEDGMENT

DXML is the joint effort of a number of individuals over the past several years. We would like to acknowledge the contributions of our colleagues, both past and present. The engineers: Luca Broglio, Richard Chase, Claudio Deiro, Laura Farinetti, Leo Lavin, Ping-Charng Lue, Joe O'Connor, Mark Schure, Linda Tella, Sisira Weeratunga and John Wilson; the technical writers: Cheryl Barabani, Barbara Higgins, Marll McDonald, Barbara Schott and Richard Wolanske; and the management: Ned Anderson, Carlos Baradello, Gerald Haigh, Buren Hoffman, Tomas Lofgren, Vehbi Tasar and David Velten. We would also like to thank Roger Grimes at Boeing Computer Services for making the Harwell-Boeing matrices so readily available.

REFERENCES

1. W. Cowell, ed., Sources and Development of Mathematical Software (Englewood Cliffs, NJ: Prentice-Hall, 1984).

2. D. Jacobs, ed., Numerical Software -- Needs and Availability (New York: Academic Press, 1978).

3. J. Dongarra, J. Bunch, C. Moler, and G. Stewart, LINPACK Users' Guide (Philadelphia: Society for Industrial and Applied Mathematics [SIAM], 1979).

4. B. Smith et al., Matrix Eigensystem Routines -- EISPACK Guide (Berlin: Springer-Verlag, 1976).

5. Digital eXtended Math Library Reference Manual (Maynard, MA: Digital Equipment Corporation, Order No. AA-Q0MBB-TE for VMS and AA-Q0NHB-TE for OSF/1).

6. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," ACM Transactions on Mathematical Software, vol. 5, no. 3 (September 1979): 308-323.

7. D. Dodson, R. Grimes, and J. Lewis, "Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms," ACM Transactions on Mathematical Software, vol. 17, no. 2 (June 1991): 253-263.

8. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," ACM Transactions on Mathematical Software, vol. 14, no. 1 (March 1988): 1-17.

9. J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," ACM Transactions on Mathematical Software, vol. 16, no. 1 (March 1990): 1-17.

10. E. Anderson et al., LAPACK Users' Guide (Philadelphia: Society for Industrial and Applied Mathematics [SIAM], 1992).

11. J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, Solving Linear Systems on Vector and Shared Memory Computers (Philadelphia: Society for Industrial and Applied Mathematics [SIAM], 1991).

12. R. Barrett et al., Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods (Philadelphia: Society for Industrial and Applied Mathematics [SIAM], 1993).

13. C. Felippa, "Solution of Linear Equations with Skyline Stored Symmetric Matrix," Computer and Structures, vol. 5, no. 1 (April 1975): 13-29.

14. I. Duff, A. Erisman, and J. Reid, Direct Methods for Sparse Matrices (New York: Oxford University Press, 1986).

15. I. Duff, R. Grimes, and J. Lewis, "Sparse Matrix Test Problems," ACM Transactions on Mathematical Software, vol. 15, no. 1 (March 1989): 1-14.

16. Alpha AXP Architecture and Systems, Digital Technical Journal, vol. 4, no. 4 (Special Issue 1992).

17. K. Dowd, High Performance Computing (Sebastopol, CA: O'Reilly & Associates, Inc., 1993).

18. DECchip 21064-AA Microprocessor -- Hardware Reference Manual (Maynard, MA: Digital Equipment Corporation, Order No. EC-N0079-72, October 1992).

19. J. Dongarra and S. Eisenstat, "Squeezing the Most Out of an Algorithm in CRAY FORTRAN," ACM Transactions on Mathematical Software, vol. 10, no. 3 (September 1984): 219-230.

20. R. Sites, ed., Alpha Architecture Reference Manual (Burlington, MA: Digital Press, 1992).

21. H. Nussbaumer, Fast Fourier Transforms and Convolution Algorithms, Second Edition (New York: Springer Verlag, 1982).

22. D. Bailey, "A High-performance FFT Algorithm for Vector Supercomputers," The International Journal of Supercomputer Applications, vol. 2, no. 1 (Spring 1988): 82-87.

23. P. Swarztrauber, "FFT Algorithms for Vector Computers," Parallel Computing, vol. 1, no. 1 (August 1984): 45-63.

24. D. Bailey, "FFTs in External or Hierarchical Memory," The Journal of Supercomputing, vol. 4, no. 1 (March 1990): 23-35.

25. O. Storaasli, D. Nguyen, and T. Agarwal, "Parallel-Vector Solution of Large-Scale Structural Analysis Problems on Supercomputers," American Institute of Aeronautics and Astronautics (AIAA) Journal, vol. 28, no. 7 (July 1990): 1211-1216.

26. H. Samukawa, "A Proposal of Level 3 Interface for Band and Skyline Matrix Factorization Subroutine," Proceedings of the 1993 ACM International Conference on Super Computing, Tokyo, Japan (July 1993): 397-406.

BIOGRAPHIES

Chandrika Kamath Chandrika Kamath is a member of the Applied Computational Mathematics Group. She has designed and implemented the sparse linear solver packages that are included in DXML. She has also optimized customer benchmarks for Alpha systems. Chandrika holds a Bachelor of Technology in electrical engineering (1981) from the Indian Institute of Technology, an M.S. in computer science (1984) and a Ph.D. in computer science (1986), both from the University of Illinois at Urbana-Champaign. She has published several papers on numerical algorithms for parallel computers.

Roy Ho As a principal software engineer in Digital's High Performance Computing Group, Roy Ho developed the signal-processing routines used in DXML. Prior to this work, he was a member of the High Performance Computing Technology Group. There he designed the clock distribution system for the VAX fault tolerant system and the delay estimation software package for the VAX 9000 system boards. Roy has B.S. (1985) and M.S. (1987) degrees in electrical engineering from the Rensselaer Polytechnic Institute. He joined Digital in 1987.

Dwight P. Manley Dwight Manley is a consulting software engineer in the Applied Computational Mathematics Group. He joined the DXML Group at its inception in 1989 and continues to support and enhance the DXML and KAPF products. Since joining Digital in

1979, he has worked on system measurement and modeling projects and was responsible for all performance modeling of the VAX 9000 CPU design. He is listed as a coinventor on 11 patents and as a coauthor of a paper on matrix computation theory. Dwight has a B.S. in mathematics from the University of Massachusetts and an M.S. in operations research from Northeastern University.