A COMPARISON OF ALGORITHMS FOR BANDED MATRIX MULTIPLICATION*

A. Tsao and T. Turnbull

Supercomputing Research Center
17100 Science Drive
Bowie, MD 20715-4300

SRC-TR-93-092

21 April 1993

Abstract. We present and compare several methods for multiplying banded square matrices. Various storage schemes and their implementations are discussed. Of particular interest is an algorithm for multiplying matrices by diagonals, which always references contiguous matrix elements. Two blocked implementations also are presented. These specialized routines are attractive for multiplying matrices whose bandwidths are known to be small relative to the size of the matrices. Results from tests performed on a Cray-2, Cray Y-MP, and RS/6000 are given. It is shown that, for specialized applications, a substantial savings can be realized over the standard three-loop multiplication algorithm.

1. Motivation

During the development of the Invariant Subspace Decomposition Algorithm (ISDA) [3], which computes the eigenvalues and eigenvectors of matrices, we began an investigation into the issues relating to banded matrices. In particular, since the ISDA relies heavily on matrix multiplication to decompose the matrix into smaller subproblems, we are exploring ways to decrease the amount of time spent performing matrix multiplication. One way to accomplish this is to perform periodic reductions to banded form [5]. This allows us to work mostly with banded matrices and to perform only those portions of the matrix multiplications necessary to compute the elements within the resulting band. (By band or bandwidth, we mean the upper or lower bandwidth of a matrix, as defined in [2]. In this paper we deal only with square matrices that have the same upper and lower bandwidth, although all of the algorithms can be extended to accommodate differing upper and lower bandwidths, as well as non-square matrices.)

On many architectures, dense matrix multiplication can be implemented to sustain computational speeds close to the achievable peak for the machine. Therefore, a banded variant of ISDA will be worthwhile only if banded matrix multiplication can be implemented efficiently. We have, therefore, been investigating a variety of methods for performing this computational primitive. If the bandwidth of a square matrix can be kept small relative to the order of the matrix, then a substantial savings in arithmetic operations can be realized by performing a banded matrix multiplication instead of a dense matrix multiplication. If the bandwidths of the two matrices to be multiplied together are known in advance, then the bandwidth of the resulting matrix will be merely the sum of the bandwidths of the two original matrices. Since the bandwidths of the matrices used in the ISDA are always known, we should be able to take advantage of a matrix multiplication routine specially tailored for banded matrices.

*This research is being supported in part by ARPA's Applied and Computational Mathematics Program, under contract DM28E04120.

1991 *Mathematics Subject Classification.* 65F30 65Y20.

Key words and phrases. matrix multiplication, banded matrices, symmetric matrices.

In Sections 2 and 3, we describe the data layout schemes and algorithms we considered in our investigations. A description of our testing methodology is given in Section 4, followed by timing results in Section 5. A brief discussion of the important cases where the matrices being multiplied are either symmetric or powers of one another is given in Section 6. We summarize our conclusions in Section 7.

2. Banded Storage Schemes

In the standard three-loop algorithm for multiplying general matrices, each loop references all elements of a particular row or column. This is needlessly expensive when the matrices to be multiplied are sparse. In xGEMM, the general multiplication subroutine provided in the Basic Linear Algebra Subroutines (BLAS), some optimization is performed by doing limited zero checking to avoid needless computations. Therefore, multiplication of fairly sparse matrices is somewhat faster than multiplication of dense matrices of the same order. Many computer vendors have written assembly-coded routines to perform this operation. While performing the timings for this paper, we noticed that for the Cray and the RS/6000 assembly-coded multiplication routine xGEMM, this step of checking for zeros was not performed. Consequently, it takes as long to multiply two dense matrices together as it does to multiply two matrices each with bandwidth one.

Banded matrices can not only be very sparse; their sparsity pattern is also deterministic. Therefore, a specialized algorithm should lead to better performance than a more general algorithm. Since most of the standard packages do not include specialized matrix multiplication routines for banded matrices, we needed to write our own. This led us to compare several methods, each of which uses a different storage scheme. The most straightforward method is to store the matrices in $n \times n$ arrays, modifying the standard three-loop algorithms so that each loop references only those elements within the bands that contribute to the final product. This involves only a slight modification of the loop indices, with the inner loop still vectorizing. For matrices with small bandwidth, this storage scheme is not particularly appealing, since it requires storing the full n^2 elements.

The next obvious scheme is to store the $n \times n$ matrix, say with bandwidth b, into a $(2b+1) \times n$ array. Such a storage scheme is much more efficient in terms of memory utilization but requires a slightly more complicated indexing scheme to perform the correct multiplications. Nevertheless, the inner loop can still be vectorized. How one actually performs packing of the matrix also can vary. Consider a 4×4 matrix with bandwidth $b = 2$:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & a_{42} & a_{43} & a_{44} \end{bmatrix}.$$

One of the LAPACK [1] storage schemes packs this matrix by diagonals as follows:

$$\begin{bmatrix} 0 & 0 & a_{13} & a_{24} \\ 0 & a_{12} & a_{23} & a_{34} \\ a_{11} & a_{22} & a_{33} & a_{44} \\ a_{21} & a_{32} & a_{43} & 0 \\ a_{31} & a_{42} & 0 & 0 \end{bmatrix},$$

which keeps the column orientation intact. (Here the zeros are really just placeholders in the array and do not get referenced.) This example also shows why a banded implementation is not desirable unless $b \ll n$. In this case, the band storage actually requires more memory locations than the original matrix. (This is true in FORTRAN where the arrays are statically defined. In C, one could use the idea of an array of pointers, where each pointer points to the correct diagonal, and each diagonal contains only the exact number of elements required. In this way, only the non-zero entries would need to be stored. However, in the interest of keeping the FORTRAN and C data structures as similar as possible for these comparisons, we did not implement this approach.)

The blocked algorithms use the same storage scheme as LAPACK above, except that instead of individual elements, blocks of elements are stored by diagonals. Here again, the bandwidth of the matrix, along with the blocksize, will determine whether this blocked storage scheme uses more memory than the unblocked version. Since these blocked versions were generally the faster algorithms, there is the usual trade-off between the speed of a particular algorithm and its memory usage. We found that many times the blocksize that produced the best performance also required more memory than other, less efficient blocksizes.

Another alternative is similar but does not preserve the column orientation. Using this method, the original sample matrix would be stored as follows:

$$\begin{bmatrix} a_{13} & a_{24} & 0 & 0 \\ a_{12} & a_{23} & a_{34} & 0 \\ a_{11} & a_{22} & a_{33} & a_{44} \\ a_{21} & a_{32} & a_{43} & 0 \\ a_{31} & a_{42} & 0 & 0 \end{bmatrix},$$

where the zeros are again placeholders. We found this method particularly attractive since the indexing is much simpler—the beginning of each row is also the beginning of each diagonal. This came in handy for the implementation of an algorithm designed by Madsen, Rodrigue, and Karush [4] to perform matrix multiplication by diagonals on vector and parallel computers. For ease of identification, we will refer to this as the MRK algorithm.

Additionally, for performance considerations, there are differences between storage schemes for FORTRAN and C. Since FORTRAN is column-oriented, one should place the diagonals in columns so that the elements are contiguous; for C, diagonals should be placed in rows.

3. Algorithms

We implemented several algorithms for banded matrix multiplication in both C and FORTRAN. Our feeling from the beginning was that the FORTRAN timings would be slightly better since, in general, FORTRAN compilers are more mature. Our results indicate that this is actually the case, even on Crays. We were led to believe that since C and FORTRAN programs are supposed to produce the same intermediate code, the two languages would perform the same on Crays. We did not find this to be the case. For one of the C implementations (SGBMM), we coded a version using explicit array references, i.e., $a[i][j]$, as well as a version using pointers (SGBMMP). As we expected, the pointer implementation was superior.

We implemented, in both FORTRAN and C, variants of the standard three-loop algorithm for both the full storage and the LAPACK storage scheme. These variants use and compute only the non-zero elements of the matrices involved. We also implemented blocked versions of these two algorithms in C only, using pointers instead of explicit array references. Although the blocked routines were not implemented in FORTRAN, based on the results of the unblocked versions, we would expect the FORTRAN blocked versions to be only slightly better than the C implementations. The blocked versions break the matrices up into smaller blocks and use a call to the assembly-coded BLAS routine SGEMM (supplied by the computer vendor) to multiply these blocks. As expected, the performance of these blocked versions varies greatly with the blocksize used. For matrices with small bandwidth, we found the optimum blocksize to be the bandwidth. As the matrices became more dense, the best blocksize was half the bandwidth; when the matrix was almost completely dense, using the order of the matrix as the blocksize was best. This last case is essentially equivalent to making a single call to SGEMM to multiply the two matrices, which is known to be optimum for dense matrices.

The MRK algorithm rewrites the order of the multiplication in such a way that contiguous elements from diagonals are multiplied rather than elements from a row by elements from a column. This alleviates the problem of irregular access of column elements, if the matrix is stored row-wise, or irregular access of row elements, if the matrix is stored column-wise. Following the notation in [4], the main diagonal of A is denoted a_0, the super-diagonals are denoted a_1, a_2, etc., and the sub-diagonals as a_{-1}, a_{-2}, etc. Additionally, $a_k(i)$ refers to the ith element of diagonal a_k, and the colon notation $a_k(i:j)$ refers to the ith through jth elements of diagonal a_k. The product $a_k(i:j)b_\ell(i:j)$ results in the vector whose entries are just the componentwise products of the entries of $a_k(i:j)$ and $b_\ell(i:j)$. Our sample matrix would contain five non-zero diagonals, since its bandwidth is 2. As an example of how the MRK algorithm works, consider the following matrix multiplication:

$$\begin{bmatrix} a_0 & a_1 & a_2 & 0 \\ a_{-1} & \ddots & \ddots & \ddots \\ a_{-2} & \ddots & & \\ 0 & \ddots & & \end{bmatrix} \begin{bmatrix} b_0 & b_1 & b_2 & 0 \\ b_{-1} & \ddots & \ddots & \ddots \\ b_{-2} & \ddots & & \\ 0 & \ddots & & \end{bmatrix} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_{-1} & \ddots & \ddots & \ddots \\ c_{-2} & \ddots & & \\ c_{-3} & \ddots & & \end{bmatrix}$$

The elements in the main diagonal c_0 are computed by the rules:

$$\begin{aligned} c_0(2:4) &= c_0(2:4) + a_{-1}(1:3)b_1(1:3) \\ c_0(3:4) &= c_0(3:4) + a_{-2}(1:2)b_2(1:2) \\ c_0(1:3) &= c_0(1:3) + a_1(1:3)b_{-1}(1:3) \\ c_0(1:2) &= c_0(1:2) + a_2(1:2)b_{-2}(1:2) \\ c_0(1:4) &= c_0(1:4) + a_0(1:4)b_0(1:4). \end{aligned}$$

Here, c_0 is assumed to have been initialized to zeros. Since the diagonal elements are stored by row (or column), they are in contiguous memory locations, and one simply has to compute the starting address in the diagonal.

The single precision unblocked versions implemented were:

- SGEMMB - banded matrix multiplication using full storage.
- SGBMM - banded matrix multiplication using the LAPACK band storage.
- SGBMMP - banded matrix multiplication using the LAPACK band storage with pointers (C only).
- SGBMMD - banded matrix multiplication using the MRK algorithm with band storage (C implementation uses pointers).

The single precision blocked versions implemented in C were:

- SGEMMB(B) - blocked banded matrix multiplication using full storage.
- SGBMMP(B) - blocked banded matrix multiplication using the LAPACK band storage with pointers.

4. Test Cases

We tested a variety of matrices with varying bandwidths to demonstrate the savings and penalties of using a tailored routine to multiply banded matrices together. As the size of the bandwidths increases, the cost of doing the index calculations adds significantly to the overall multiplication cost. Thus, one reaches a point where it is more efficient to perform the full matrix multiplication than to try to determine which multiplications are actually necessary and only performing those. However, if one's application is such that one knows in advance that the bandwidths are small relative to the order of the matrices, then there are distinct advantages to using such tailored routines.

The routines were tested on a Cray-2 using 1000×1000 matrices with bandwidths ranging from 1 to 999 and on a Cray Y-MP and an RS/6000 Model 550 with matrices of order 500 and bandwidths ranging from 1 to 499. The Cray-2 compilers used were the Cray Standard C compiler, Release 3.0.1.6, and the CFT77 FORTRAN compiler, Version 5.0.2.2. On the Cray Y-MP, we used the Cray Standard C compiler, Release 3.0.4.1, and the CFT77 FORTRAN compiler, Version 5.0.3.14. For maximum optimization, the '-o aggress' option was used with the FORTRAN compilers, while the '-h aggress' option was used with the C compilers. On the IBM RS/6000, the IBM AIX XL C Compiler/6000, Version 01.02.0000.0000, and the IBM AIX XL FORTRAN Compiler/6000, Version 02.02.0100.0003 were used. The '-O' option was used for optimization with both the FORTRAN and C compilers. All results were checked against SGEMM, and all routines produced identical results. Inner loops for all routines, both C and FORTRAN, vectorized, although some of the C routines required extra care over their FORTRAN counterparts to ensure that the inner loop did, in fact, vectorize. All tests were run on non-dedicated systems, which accounts for the slight variations in execution times, particularly for SGEMM.

5. Timing Results

The results of the test cases follow. The x-axis represents the bandwidths of the test matrices. The y-axis represents the time, in seconds, for the programs to execute. Since the times differ greatly between those matrices with bandwidths less than 100 and those with bandwidths greater than 100, we present different graphs for these two cases. First the results for the C programs are presented, followed by the results for the FORTRAN programs. In almost all cases, the FORTRAN programs ran slightly faster than the C programs for the same routines.

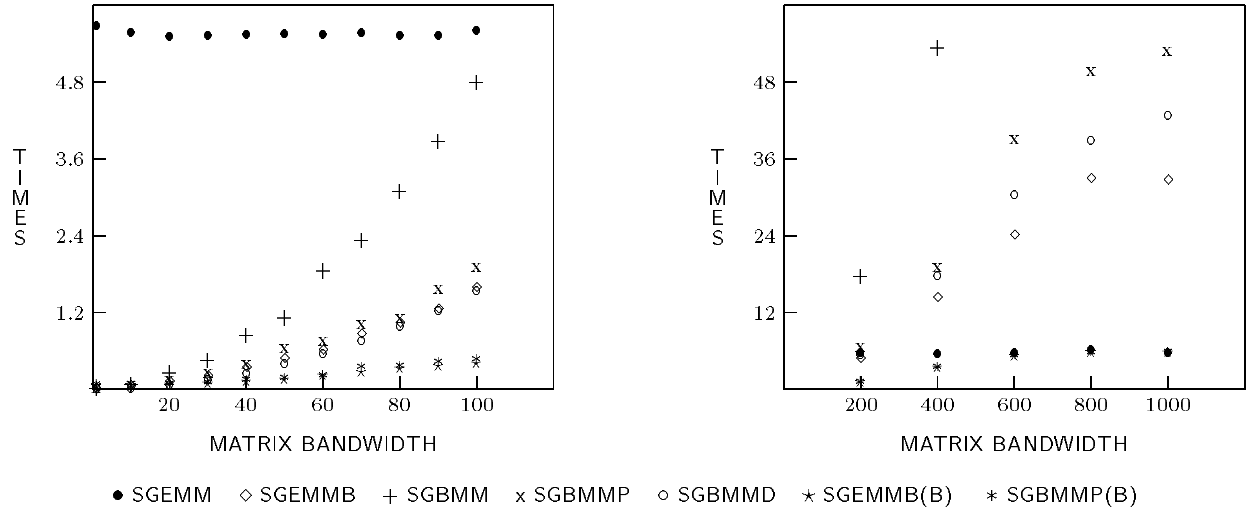

Figure 1: Timing Comparison of Cray-2 C Programs (1000x1000)

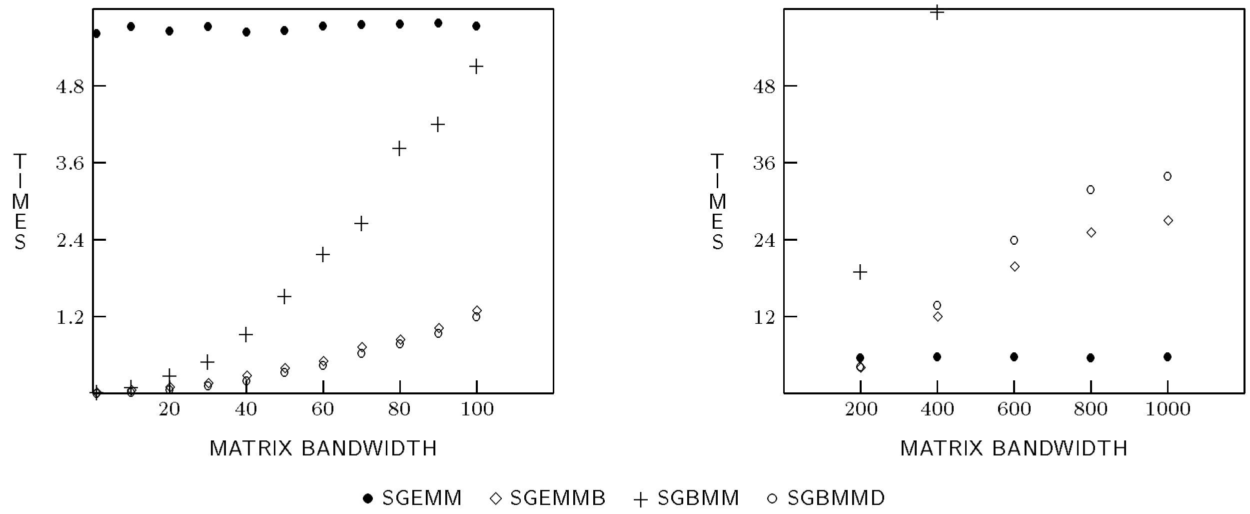

Figure 2: Timing Comparison of Cray-2 FORTRAN Programs (1000x1000)

The following symbols are used to denote the different versions of the multiplication routines:

- • SGEMM - assembly-coded BLAS matrix multiplication using full storage.
- ◇ SGEMMB - banded matrix multiplication using full storage.
- + SGBMM - banded matrix multiplication using the LAPACK band storage.

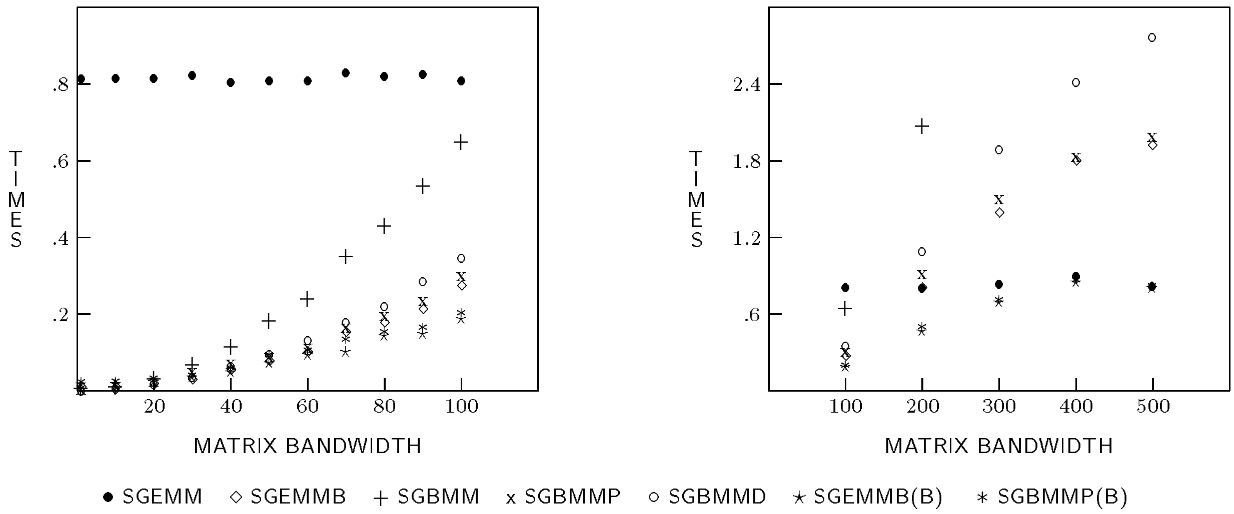

FIGURE 3: TIMING COMPARISON OF CRAY Y-MP C PROGRAMS (500X500)

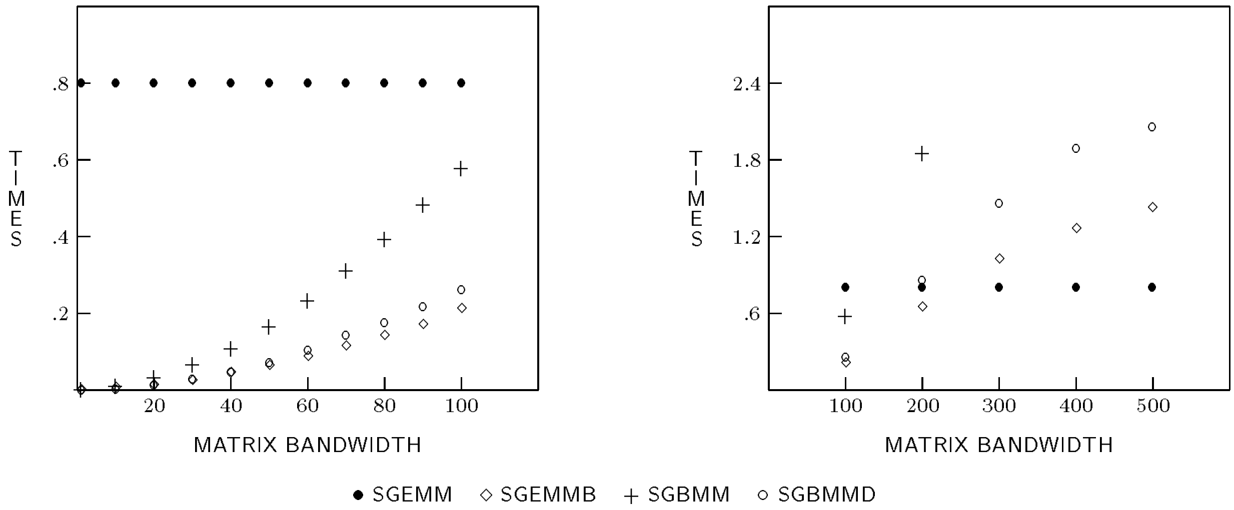

FIGURE 4: TIMING COMPARISON OF CRAY Y-MP FORTRAN PROGRAMS (500X500)

x SGBMMP - banded matrix multiplication using the LAPACK band storage with pointer.
∘ SGBMMD - banded matrix multiplication using the MRK algorithm.
⋆ SGEMMB(B) - blocked banded matrix multiplication using full storage.
∗ SGBMMP(B) - blocked banded matrix multiplication using the LAPACK band storage.

In many cases the times for SGBMM and/or SGBMMP were disproportionately larger than the times for the other routines. When that occurred, in order to keep the range on the plots reasonable, those times were not plotted. In all cases where the results are not plotted, the times were greater than those presented and, consequently, do not represent feasible alternatives. Although the inner loop for SGBMM vectorizes on the Crays, there could be memory contention due to the irregular memory access pattern for the array elements. Additionally, this storage scheme requires more operations to compute the index of a particular element.

The fact that the times for the assembly-coded BLAS routine SGEMM on all machines tested are constant with respect to the bandwidth of the matrix illustrates that this code does not take into account any of the zeros in the matrices. The FORTRAN version of SGEMM supplied with the BLAS does check for zeros;

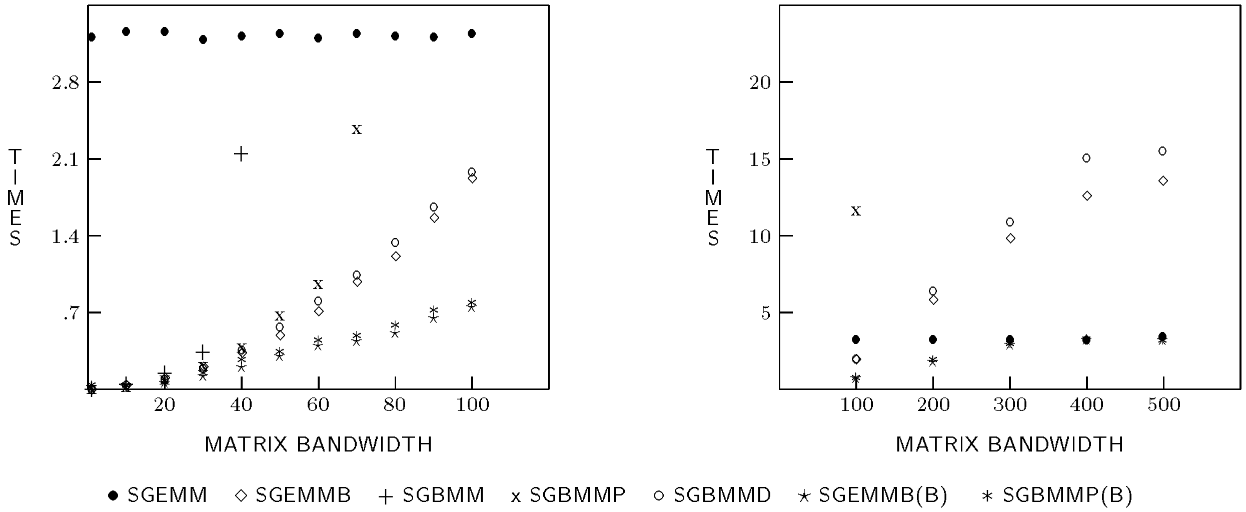

FIGURE 5: TIMING COMPARISON OF RS/6000 C PROGRAMS (500X500)

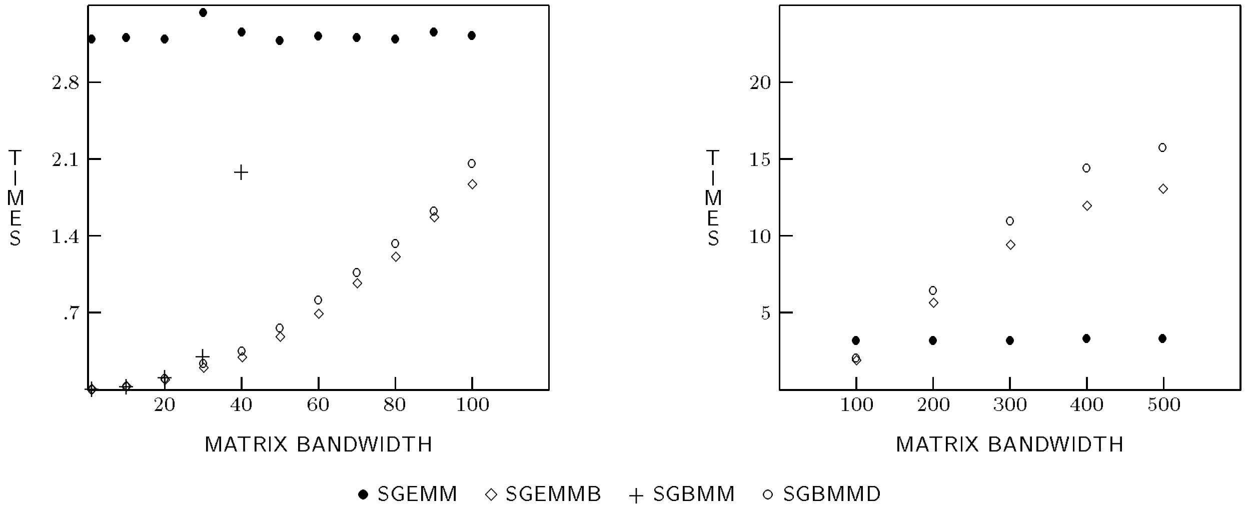

FIGURE 6: TIMING COMPARISON OF RS/6000 FORTRAN PROGRAMS (500X500)

consequently, those times will scale with the problem size. However, here we only report the times for the assembly-code SGEMM routine, since it is superior to the FORTRAN version for full matrices.

For matrices with bandwidths greater than 10, the blocked versions are superior to the corresponding unblocked version *for the optimum or near-optimum blocksize.* Notice that for matrices with bandwidths less than 300 – 400, the blocked versions are also superior to the assembly-code BLAS routine SGEMM. These blocked versions take the sparsity of the matrices into account and call SGEMM to multiply almost dense matrices. Consequently, there is very little time wasted multiplying mostly zeros.

In general, the times for SGEMMB and SGBMMD are very similar. Recall that the major difference between these two routines is memory usage. SGEMMB uses the full $n \times n$ array, while SGBMMD stores the matrix by diagonals, utilizing $(2b+1) \times n$ arrays. Both routines compute only the non-zero entries of the resulting product.

As mentioned before, the FORTRAN routines generally performed slightly better than their C counterparts. In all cases, SGEMMB and SGBMMD and the two blocked routines SGEMMB(B) and SGBMMP(B) performed considerably better than the assembly-coded routine SGEMM for bandwidths less than 100.

Figure 7 provides a comparison between the theoretical savings from using a banded matrix multiplication routine and the actual performance observed when multiplying two 1000×1000 matrices with varying bandwidths together. Following [2], the cost to perform a dense matrix multiplication is $2n^3$, and although an approximate cost for the banded case can be derived, we used the actual number of operations performed during the multiplication. This ratio gives the maximum improvement possible. Then, the ratio of the actual runtimes from the Cray-2 tests for the dense versus the banded implementation was calculated. The actual amount of savings realized is dependent on many factors, including the data structure utilized, the amount of memory available, and the machine architecture. For most cases, our fastest implementations were within a factor of two of the theoretical savings.

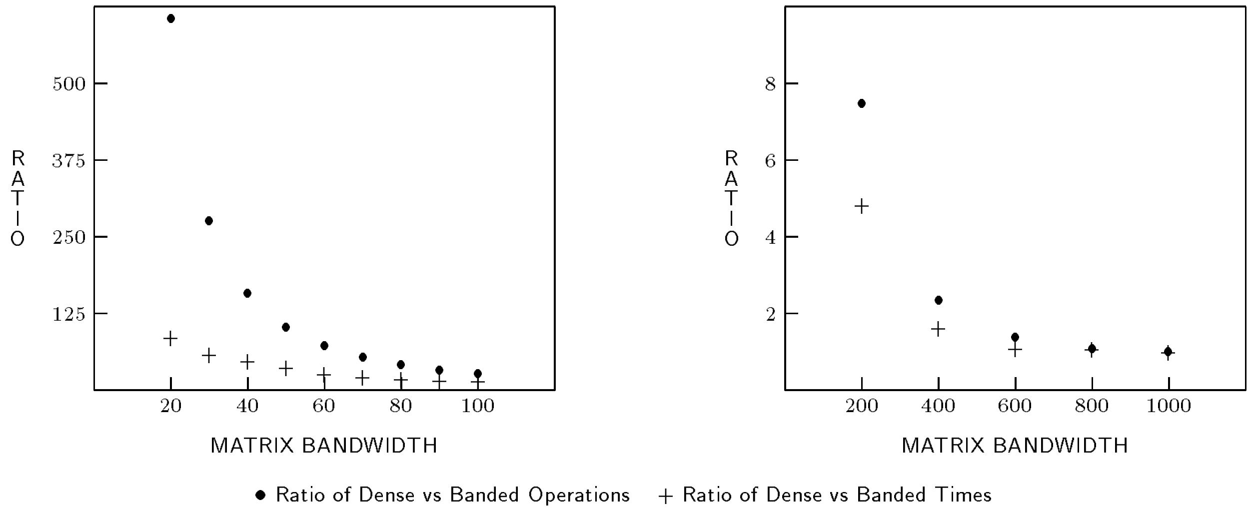

FIGURE 7: RATIOS FOR DENSE VS BANDED MATRIX MULTIPLY (1000X1000)

6. SYMMETRIC MATRICES AND POWERS OF SYMMETRIC MATRICES

Since the product of two symmetric matrices is not, in general, symmetric, there is usually no savings in the number of multiplications required. There is, however, a savings in the amount of memory that is needed to store the symmetric matrices. Usually only the upper or lower half of the matrices needs to be stored. Since only half the matrix is stored, the loop structure is slightly more complicated than the standard three-loop algorithm. Consequently, there is a point at which the bandwidth is too large, and it will be more efficient to use the full matrix rather than try to save storage. This point will vary depending on the architecture and data structure used.

Although products of symmetric matrices are not necessarily symmetric, powers of symmetric matrices are symmetric. Not only is there a savings in the storage required, for both the original matrix and the product matrix, but there is also a savings in the number of computations required. Since the resulting matrix will be symmetric, only half of the entries are required. This will generally equate to a savings of 50%, although vectorization and pipelining can affect the overall improvement. For problems where it is known that the bandwidth is not too large, specialized routines seem particularly well-suited. However, if the bandwidth is too large, the savings realized in a specialized routine will be lost, and it will be more efficient to do the full matrix computations. Again, this crossover point will depend on the data structure used and the machine architecture.

7. Conclusions

We have presented several specialized routines for multiplying banded square matrices together. For certain applications where the bandwidth of the matrix is known to be small relative to the order of the matrix, these tailored routines can result in substantial savings, both in terms of memory usage in storing the banded matrix and in computational time required to perform the multiplication. These routines are general, but can be modified for even more specialized applications, such as multiplying symmetric banded matrices or computing powers of symmetric banded matrices. For our particular application of solving the symmetric eigenvalue problem using the ISDA, which relies heavily on matrix multiplication of very specialized matrices, we are able to reduce the overall time considerably by reducing the time spent performing matrix multiplication. However, if the bandwidth grows too large, the best method remains the standard three-loop multiplication algorithm, particularly if it has been implemented in assembly code.

8. Acknowledgements

We would like to thank Cray Research Inc. for use of the Y-MP and Jack Dongarra of Oak Ridge National Labs for use of the RS/6000 Model 550.

References

1. Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, & D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.
2. Golub, G. & C. F. Van Loan, *Matrix Computations, 2nd ed.*, Johns Hopkins University Press, Baltimore, 1989.
3. Lederman, S., A. Tsao, & T. Turnbull, *A parallelizable eigensolver for real diagonalizable matrices with real eigenvalues*, Technical Report TR-91-042, Supercomputing Research Center (1991).
4. Madsen, N. K., G H Rodrigue, J Karush, *Matrix multiplication by diagonals on a vector/parallel processor*, Info. Proc. Letters. **5** (1976), no. 2, 41-5.
5. Tsao, A. & T. Turnbull, *A study of the Invariant Subspace Decomposition Algorithm for banded symmetric matrices*; (in preparation).