GEMMW: A PORTABLE LEVEL 3 BLAS WINOGRAD VARIANT OF STRASSEN'S MATRIX–MATRIX MULTIPLY ALGORITHM*

CRAIG C. DOUGLAS†, MICHAEL HEROUX‡, GORDON SLISHMAN§ AND ROGER M. SMITH¶

Abstract. Matrix–matrix multiplication is normally computed using one of the BLAS or a reinvention of part of the BLAS. Unfortunately, the BLAS were designed with small matrices in mind. When huge, well conditioned matrices are multiplied together, the BLAS perform like the blahs, even on vector machines. For matrices where the coefficients are well conditioned, Winograd's variant of Strassen's algorithm offers some relief, but is rarely available in a quality form on most computers. We reconsider this method and offer a highly portable solution based on the Level 3 BLAS interface.

Key Words. Level 3 BLAS, matrix multiplication, Winograd's variant of Strassen's algorithm, multilevel algorithms

AMS(MOS) subject classification. Numerical Analysis: Numerical Linear Algebra

1. Preliminaries.

Matrix–matrix multiplication is a very basic computer operation. A very clear description of how to do it can be found in many textbooks, e.g., [1]. Suppose we want to multiply two matrices

$$A : M \times K \quad and \quad B : K \times N,$$

where the elements of A and B are real or complex numbers and M, K, and N are natural numbers.

Strassen's method recursively works with sets of 2×2 submatrices to form the product using 7 matrix multiplications instead of the obvious 8. This is not very different from standard multilevel methods [7] used routinely to solve partial differential equations. Strictly speaking, we compute

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

* Yale University Department of Computer Science Report YALEU/DCS/TR-904, New Haven, CT, 1992. To appear in the Journal of Computational Physics.

† Department of Computer Science, Yale University, P. O. Box 2158, New Haven, CT 06520-2158. and Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598. E-mail: *na.cdouglas@na-net.ornl.gov*.

‡ Mathematical Software Research Group, Cray Research, Inc., 655-F Lone Oak Drive, Eagan, MN 55121, E-mail: *mamh@cray.com*.

§ Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598, E-mail: *slishmn@watson.ibm.com*.

¶ Department of Computer Science, Yale University, P. O. Box 2158, New Haven, CT 06520-2158. E-mail: *smith-roger@cs.yale.edu*.

using the following algorithm (*Strassen–Winograd*):

$$
\begin{array}{lll}
S_1 = A_{21} + A_{22} & M_1 = S_2 S_6 & T_1 = M_1 + M_2 \\
S_2 = S_1 - A_{11} & M_2 = A_{11} B_{11} & T_2 = T_1 + M_4 \\
S_3 = A_{11} - A_{21} & M_3 = A_{12} B_{21} & \\
S_4 = A_{12} - S_2 & M_4 = S_3 S_7 & \\
S_5 = B_{12} - B_{11} & M_5 = S_1 S_5 & C_{11} = M_2 + M_3 \\
S_6 = B_{22} - S_5 & M_6 = S_4 B_{22} & C_{12} = T_1 + M_5 + M_6 \\
S_7 = B_{22} - B_{12} & M_7 = A_{22} S_8 & C_{21} = T_2 - M_7 \\
S_8 = S_6 - B_{21} & & C_{22} = T_2 + M_5
\end{array} \tag{1}
$$

(see [12] and [13]). This is not a very convenient way to define this algorithm, but it is the standard textbook definition. Unlike textbook exercises, we do not require square matrices nor restrict the dimensions to 2^k for some natural number k.

While there are only 7 matrix–matrix multiplications and 15 matrix–matrix additions and subtractions in (1), there is no hint as to how to implement this efficiently. The crossover point, $mindim$, when Strassen–Winograd is more efficient than the classical algorithm, can be computed. It depends on the difference in cost between performing an arithmetic operation and loading or storing a number in memory. When arithmetic is relatively expensive, $mindim = 32$ is common. When arithmetic is less expensive relatively, $96 \leq mindim \leq 256$ is common.

Extra storage is required to hold sums of quadrants of A and B. Since the shape of C may be quite different from these two matrices, extra storage of approximately the same size as the quadrants is required. Hence, any extra storage required for the intermediate results may be a considerable percentage of the entire memory of a computer.

The obvious approaches to implementing Strassen–Winograd for general sized matrices require either padding the matrices with extra zero rows and/or columns or doing a number of rank one updates that are slow and produce spaghetti code.

In theory, while the classical method can be implemented using techniques which do the inner products as accurately as possible, the added cost of doing this step usually eliminates it from real programs. Simple examples exist where, due to the unnatural submatrix additions, variants of Strassen's algorithm get the wrong answer while the classical method gets the right answer. However, for many problems, due to the reduction of arithmetic operations, Strassen–Winograd has better round off properties than the classical method. Hence, for matrices A and B, we assume the coefficients are "well conditioned" enough so that both methods get acceptable answers. In other words, *caveat emptor* for either class of matrix–matrix multiplication in real codes. For a more complete discussion of this issue, see [3] and [8] along with their references. An interesting application to solving linear systems of equations is contained in [1] and [4].

This paper is actually interested in a highly portable Level 3 BLAS [6] interface for computing

$$C \leftarrow \alpha \cdot op(A) op(B) + \beta \cdot C, \tag{2}$$

where

$$op(X) = \begin{cases} X, \\ X\ transpose, \\ X\ conjugate\ transpose, \\ X\ conjugate, \end{cases}$$

and

$$op(A) : M \times K, \quad op(B) : K \times N, \quad and \quad C : M \times N.$$

Most of the discussion will ignore the conjugate and transpose cases, but the implementation is that of (2).

This paper addresses how Strassen–Winograd can be implemented portably with a minimum of extra storage, no rank one updates for general matrices, and whatever library the user wishes to use on a particular machine. In §2, one solution is constructed. In §3, a special case of classical matrix–matrix multiplication for complex matrices is discussed. In §4, a hybrid matrix–matrix multiplication is constructed which minimizes communication along with the justification for its existence. In §5, numerical experiments are presented for both the serial and parallel cases.

2. Serial computer implementation. In this section, we describe a practical and highly efficient implementation of (1) for serial machines. This includes the flow of computation, how odd sized matrices are handled, and the memory requirements.

Each one of the temporary variables S_i, M_i, and T_i in (1) can be considered a register. Hence, register optimization techniques can be applied based on the directed data flow graph. Each temporary variable is stored in an appropriately sized register and accessed only as needed. Clearly the dimensions of each intermediate result must be considered. Further, when M, K, or N is odd, there is considerable flexibility in choosing the dimensions of the quadrants (in fact, using irregular shaped quadrants is better than regular ones).

Since we are computing $C = AB$, parts of C can be freely used as the registers or work areas. The data for each matrix is assumed to be stored in column order as a single data area (i.e., Fortran style), rather than as a collection of row vectors with a column vector of pointers to the rows (i.e., C style). Hence, the number of elements of any work areas can be bound by a single number. As will be shown, only two large work areas, W_{MK} and W_{KN}, are needed. When M is odd, an additional short vector of length $N/2$ is required. The size of each large work area is

$$W_{MK} : \left\lfloor \frac{M \max(K,N) + M + \max(K,N) + 4}{4} \right\rfloor$$

and

$$W_{KN} : \left\lfloor \frac{KN + K + N + 4}{4} \right\rfloor,$$

where the floor symbol refers to rounding down to the nearest natural number. The total amount of extra storage required over all levels (including the possible additional short vector) is bounded by

$$\frac{1}{3}[M \max(K,N) + KN] + \frac{1}{2}[M + \max(K,N) + K + 3N] + 32. \tag{3}$$

In addition, when $\beta \neq 0$ or C overlaps with A or B in (2), an additional MN storage is required to hold AB before adding that to βC. Thus, when $K = M = N$ in (3), the auxiliary storage requirements are approximately $cN^2/3$, where $c \in \{2, 5\}$.

The actual order of operations and location of intermediate and final results for our serial computer code is in Fig. 1. There are two special cases of note.

Step	W_{MK}	C_{11}	C_{12}	C_{21}	C_{22}	W_{KN}	Operation
1.						S_7	$B_{22} - B_{12}$
2.	S_3						$A_{11} - A_{21}$
3.				M_4			$S_3 S_7$
4.	S_1						$A_{21} + A_{22}$
5.						S_5	$B_{12} - B_{11}$
6.					M_5		$S_1 S_5$
7.						S_6	$B_{22} - S_5$
8.	S_2						$S_1 - A_{11}$
9.		M_1					$S_2 S_6$
10.	S_4						$A_{12} - S_2$
11.			M_6				$S_4 B_{22}$
12.			T_3				$M_5 + M_6$
13.	M_2						$A_{11} B_{11}$
14.		T_1					$M_1 + M_2$
15.			C_{12}				$T_1 + T_3$
16.		T_2					$T_1 + M_4$
17.						S_8	$S_6 - B_{21}$
18.				M_7			$A_{22} S_8$
19.				C_{21}			$T_2 - M_7$
20.					C_{22}		$T_2 + M_5$
21.		M_3					$A_{12} B_{21}$
22.		C_{11}					$M_2 + M_3$

FIG. 1. *WINOS: Implementation of Winograd variant of Strassen's algorithm*

Step	Array	Step	Function
4.	S_1	(a)	If K is odd, then copy first column of A_{21} into W_{MK}.
		(b)	Complete S_1.
10.	S_4	(a)	If K is odd, then pretend first column of $A_{21} = 0$ in W_{MK}.
		(b)	Complete S_4.
11.	M_6	(a)	If M is odd, then save first row of M_5.
		(b)	Calculate most of M_6.
		(c)	Complete M_6 using (a) based on M odd or not.
21.	M_3	(a)	Calculate M_3 using an index shift.

FIG. 2. *Modifications to avoid corruption*

- *Odd M and/or N*: Conceptually we duplicate A's odd middle row or B's odd middle column. The product would then have a duplicated middle row or column accordingly. Letting the output quadrants overlap by a row (column) eliminates the duplicated row (column) produced from conceptual A and B. Conceptual A and B are used in the recursion.
- *Odd K*: Conceptually we duplicate B's odd middle row and insert a column of zeroes after A's odd middle column. In each operation involving A_{12} or A_{22}, the first column is either omitted (because it is zero) or is handled as a virtual column of zeroes.

For example, consider the product of two 3×3 matrices:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 3 & 2 & 1 \\ 2 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 3 \\ 9 & 8 & 5 \\ 10 & 9 & 6 \end{bmatrix}.$$

Conceptually, we use dimensions divisible by 2:

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 2 & 0 & 2 \\ 1 & 2 & 0 & 2 \\ 1 & 2 & 0 & 3 \end{bmatrix} \begin{bmatrix} 3 & 2 & 2 & 1 \\ 2 & 2 & 2 & 1 \\ 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 5 & 3 \\ 9 & 8 & 8 & 5 \\ 9 & 8 & 8 & 5 \\ 10 & 9 & 9 & 6 \end{bmatrix} \Rightarrow \begin{bmatrix} 6 & 5 & 3 \\ 9 & 8 & 5 \\ 10 & 9 & 6 \end{bmatrix}$$

Actual A and B provide easy access to conceptual A and B. For typical problems, the total elimination of rank one updates and matrix–vector operations more than compensates for the duplication of rows and/or columns. In all cases, the output matrix C is divided into four perhaps unequal quadrants as follows:

- $C_{11} : MR \times NR$, where $MR = \lceil M/2 \rceil$ and $NR = \lceil N/2 \rceil$
- $C_{12} : MR \times NT$, where $NT = \lfloor N/2 \rfloor$
- $C_{21} : MT \times NR$, where $MT = \lfloor M/2 \rfloor$
- $C_{22} : MT \times NT$,

where the ceiling symbol refers to rounding up to the nearest natural number.

Each intermediate result is computed in dimensions just sufficient to fill dependent quadrants of C. Consider the case of odd M: a single storage conflict arises within C as M_5 and M_6 are stored in the right half of C and both require MR rows. A single row copy resolves this conflict (see Fig. 2). An algorithmic revision transforms the row copy into a column (stride one) copy. Unfortunately, this introduced a number of unpleasant side effects elsewhere in the algorithm, so it was not implemented.

Conceptual A and B are of size $2MR \times 2KR$ and $2KR \times 2NR$ respectively. The first MR rows of conceptual A come from the first MR rows of actual A, and the last MR rows of conceptual A from the last MR of actual A (similarly, for KR and NR). Hence the duplication of the odd rows and columns is indeed free.

Duplication of the odd row and column of the common dimension, K, would corrupt the inner products. Simply by taking the duplicated odd column of A to be zero in A_{12} and A_{22} avoids corrupting the inner products. Hence, trivial adjustments are needed in only the four places where A_{12} and A_{22} are used: S_1, S_4, M_3, and M_7 (see Fig. 2).

The benefits of this duplication method are speed and simplicity. The solution involves no rank one updates and no matrix–vector operations to deal with odd K, M, or N. The code follows a virtually straight path from top to bottom for all M, K, and N.

TABLE 1
First letter of each routine

Letter	Data type
c	single precision complex
d	double precision real
s	single precision real
z	double precision complex

TABLE 2
Major operation macros

Macro name	BLAS/ LAPACK/ Cray SciLib	IBM ESSL	NAG	Complex only	Operation
VCOPY	*_copy*	*_copy*	*_f06eff/gff*	—	$y = x$
VAXPY	*_axpy*	*_axpy*	*_f06ecf/gcf*	—	$y = \alpha x + y$
VYAX	*_yax**	*_yax*	*_f06fdf/hdf*	—	$y = \alpha x$
MATMUL	*_gemm*	*_gemul*	*_f06yaf/zaf*	*_gemul3**	$C = op(A)op(B)$
MATADD	*_geadd**	*_geadd*	*_f06ctf/cwf*	—	$C = op(A) + op(B)$
MATSUB	*_gesub**	*_gesub*	*_f06ctf/cwf*	—	$C = op(A) - op(B)$

* provided as part of a set of extended BLAS routines.

The code is organized into four major routines. Each one takes its first letter from Table 1. There is actually only one copy of each routine; they are each compiled with a different compiler definition for floating point to get the correct name and compilation. The data types, subroutine names, and *mindim* are all defined using macros. This reduces the cost and chance of errors should any of these routines need to be modified at some later date.

The actual recursive Strassen–Winograd routine, which is not meant to be called directly by a user, is *_winos*, where one of the letters in Table 1 is substituted for the "_" symbol. The user actually calls *_gemmw*, which does error checking, memory allocation (if necessary), calls *_winos*, and completes the calculation of (2). Alternately, the user calls _gemmwb, which has the exact same arguments as the routine *_gemm* in the Level 3 BLAS. A special version of the standard Level 3 BLAS routines *cgemm* and *zgemm* is provided as *_gemul3* (see §3).

The three major routines call a collection of Level 1 and Level 3 BLAS routines to do the bulk of the computation. The code is flexible enough that by modifying the macro definitions in one header file, essentially any library can be substituted for the default ones. In this manner, it is trivial to make the code work with the BLAS, Cray Scientific Library [9], ESSL [10], NAG [11], or any other library the user chooses. We used the BLAS distributed with LAPACK [2]. A list of the macros and the routines that are actually called is contained in Table 2.

Some of the operations required by Strassen–Winograd (e.g., $op(A) + op(B)$) are not part of the Level 3 BLAS. Further, Fortran–90 does not provide adequate capability for using transposes without copying the data. Hence, an additional collection of routines (in both Fortran–77 and Fortran–90 formats) are provided for people who need to use a library without this capability.

Our approach leads to a very clean implementation that supports many numerical

libraries on a variety of platforms. Further, the numerical results in §5 demonstrate that we are competitive with hand tuned codes on various machines.

3. Complex Strassen–Winograd. Provided with *_gemmw* is a specialized version of the classical matrix multiplication algorithm for complex matrices. Let P, Q, R, and S be real matrices. A well known trick [1] calculates

$$(P + Qi) \cdot (R + Si)$$

using the formula

$$[P \cdot (R - S) + (P - Q) \cdot S)] + [(P + Q) \cdot R - P \cdot (R - S)]i.$$

Note that there are only 3 matrix multiplies instead of the usual 4.

When applying this trick to Strassen–Winograd there are two options:

1. Decompose $op(A)$ and $op(B)$ into real and complex parts first and then apply Strassen–Winograd to the three products (as real matrix multiplies).
2. Apply Strassen–Winograd directly to the complex matrices and decompose the small matrices that the classical matrix multiplication algorithm is ultimately used on.

We implemented both options and tried them on a number of different computer architectures. For us, the first option actually runs slightly faster ($\approx 2 - 3\%$) than the second, but uses 2.5 times as much storage in the process. As a result, we decided that the extra storage requirements did not justify the trivial savings in time.

Note that by using the second option and a Level 3 BLAS routine (e.g., *dgemm* or *sgemm*) that requires no extra storage, the storage requirement given in (3) is still valid. Needless to say, *_gemmw* and *_winos* use the second option in the complex cases.

Our routines do not attempt to re-order the matrices $op(A)$ and $op(B)$ in order to achieve stride 1 vectors. We note that the matrices may not always be in a part of memory that allows writes or is quick to do so. Examples include read–only shared memory segments (common when computing and input/output are overlapped in a multitasking situation) or the matrices are actually on disk (either by choice or having been paged). Another obvious advantage to not re-ordering the matrices is that $op(A)$ and and $op(B)$ can overlap without causing problems.

4. Parallel environments. There are two quite different parallel environments, namely, inexpensive data access (e.g., shared memory machines) and relatively expensive data access (e.g., distributed memory machines or clusters of workstations). The latter seems to be what most parallel computer manufacturers are designing now and what many members of the scientific community are using. In this section we are not assuming that the algorithm designers have the right to require that data reside in specific memory areas; we merely assume that the data resides somewhere in the computing environment's memory banks.

Each level of Strassen–Winograd involves twenty two matrix operations: fifteen matrix additions and seven matrix multiplications. The blocked version of the classical method requires twelve operations: four matrix additions and eight matrix multiplications. Thus, Strassen–Winograd is inferior to the classical method when the cost of matrix operand loads and stores is high enough. Assuming matrix loads and stores are quite expensive, we developed a hybrid algorithm: classical among the parallel processors but Strassen–Winograd within each processor.

Initially, a heuristic iteratively partitions the processors and matrices A, B, and C until each processor has a submatrix multiplication to perform independently and

in parallel with the others. Assume there are p processors, which can be factored into the product of primes:

$$p = \prod_{i=1}^{n} p_i.$$

Without loss of generality, we assume $p_{i-1} \leq p_i$, $2 \leq i \leq n$. Start with one set of p processors. In step i, the heuristic partitions each set of processors into p_i subsets and divides the maximum of $\{M_i, K_i, N_i\}$ by p_i. Also, the number of processor partitions grows by a factor of p_i, as does the number of submatrix multiplications. Each submatrix multiplication decreases in complexity by an identical factor of p_i. After n steps, each processor has an independent submatrix multiplication.

For example, suppose $p = 12 = 2 \cdot 2 \cdot 3$, $C = A \times B$, and A and B are 3×4 and 4×5 matrices respectively. The partitioning algorithm just described begins by assuming that all 12 processors are working on one problem, namely the original matrix–matrix multiplication. The matrices are partitioned initially as whole matrices:

$$\begin{bmatrix} aaaa \\ aaaa \\ aaaa \end{bmatrix} \times \begin{bmatrix} bbbbb \\ bbbbb \\ bbbbb \\ bbbbb \end{bmatrix} = \begin{bmatrix} ccccc \\ ccccc \\ ccccc \end{bmatrix}.$$

First, the heuristic uses the first 2 in the prime factorization of 12 so that two processor groups are formed (e.g., $1-6$ and $7-12$). These collaborate on two submatrix multiplications:

$$\left[\begin{array}{c} aaaa \\ aaaa \\ aaaa \end{array}\right] \times \left[\begin{array}{c|c} bbb & bb \\ bbb & bb \\ bbb & bb \\ bbb & bb \end{array}\right] = \left[\begin{array}{c|c} ccc & cc \\ ccc & cc \\ ccc & cc \end{array}\right].$$

Second, the heuristic uses the second 2 in the prime factorization of 12 so both processor groups are split in half (e.g., $1-3$, $4-6$, $7-9$, and $10-12$). These collaborate on four submatrix multiplications:

$$\left[\begin{array}{c|c} aa & aa \\ aa & aa \\ aa & aa \end{array}\right] \times \left[\begin{array}{c|c} bbb & bb \\ bbb & bb \\ \hline bbb & bb \\ bbb & bb \end{array}\right] = \left[\begin{array}{c|c} ccc & cc \\ ccc & cc \\ ccc & cc \end{array}\right].$$

Third, the heuristic uses the last prime (3) in the prime factorization of 12 to get 12 processor groups with one processor in each group. These collaborate on 12 submatrix multiplications:

$$\left[\begin{array}{c|c} aa & aa \\ \hline aa & aa \\ \hline aa & aa \end{array}\right] \times \left[\begin{array}{c|c} bbb & bb \\ bbb & bb \\ \hline bbb & bb \\ bbb & bb \end{array}\right] = \left[\begin{array}{c|c} ccc & cc \\ \hline ccc & cc \\ \hline ccc & cc \end{array}\right].$$

Finally, each processor performs its matrix multiplication independent of the others. Hence, we took a tuple (M,K,N) of (3,4,5) and produced a "partitioning" ($\mathcal{M}$,$\mathcal{K}$,$\mathcal{N}$) of (2,3,2) of blocks of A, B, and C.

The user stores submatrices of A and B into the parallel data base. In our example code, the Linda system [5] was used. The computation continues with a call to the parallel matrix multiplication routine, *matmulp*, with seven parameters. These include the original sizes (M,K,N), the partitioning ($\mathcal{M}$,$\mathcal{K}$,$\mathcal{N}$), and the number of processors (p) to use.

Then *matmulp* creates $\mathcal{MN}-1$ new parallel processes to compute submatrices of C which are placed in the database. All $\mathcal{MN}$ processes call a routine $DoCij$ which is completely independent of the other $DoCij$'s. Each $DoCij$ process creates $\mathcal{K}-1$ new processes to compute one term of the respective matrix inner product. Then $DoCij$ computes the remaining term itself. In total, p roughly equal, independent, and parallel Strassen–Winograd processes execute.

Each parallel $DoCij$ uses *_gemmw* to compute its submatrix term, sums the outputs of its child processes, outputs its C submatrix to the database, and quits. The result, C, remains in the distributed database.

5. Numerical experiments. Simple experiments for *_gemmw* were run on a variety of machines. The ones reported here are for multiplying two matrices A and B, both of which were initialized with random (positive and negative) numbers, starting from a known seed. The matrices were all large (1000–1826) and represent a subset of the data points, which were

$$\{1000 \cup N_i\},\ where\ N_0 = 100\ and\ N_i = \lfloor (6N_{i-1})/5 \rfloor,\ i > 0.$$

Each of the machines reported on here had at least 128Mb of memory (and usually much more), including the workstations.

Both Cray's Scientific and Math library and IBM's ESSL have a Strassen–Winograd subroutine. We used various routines from each of these libraries to do the basic block matrix operations like addition or multiplication (classical algorithm only). Since the Level 3 BLAS do not include matrix addition or subtraction, some simple routines are supplied to do these essential operations.

The basic operations are accomplished using calls to subroutines written in a variety of languages (Fortran or assembly language normally). Which subroutines are actually called and the correct order of the parameters is defined through a set of macros in a header file. In addition, the internal name mapping used by the C and Fortran compilers (e.g., underscore additions or capitalization) is included in the macro definitions. So, our code handles data typing, library names, and compiler dependent name mapping transparently.

The routines were tested on a number of machines, including those in Table 3. Note that for CPU's which can do a number of things simultaneously, *mindim* is much larger than on ones without this capability. On a 4 processor Cray 2 used with micro–tasking, $mindim = 768$ appears to be the crossover point on a system with time sharing in effect. Note that on a standalone Cray 2, $mindim = 128$ is good. On the IBM 3090S, $mindim = 192$ is good when either ESSL or NAG is used with our routine, but $mindim = 256$ is good when the BLAS are used.

In computing (2), there are two cases of interest regarding auxiliary storage. The first is when $\beta \neq 0$ and A and/or B overlaps with C in memory. The second is the opposite situation. For multiplying $N \times N$ matrices the memory requirements for Strassen implementations are in Table 4. The IBM ESSL routine *gemms* assumes $\alpha = 1$, $\beta = 0$ in (2), and no overlapping of A, B, or C.

Figs. 3–9 show a scaled set of run times for a representative subset of the machines in Table 3. The scaling factors are in Table 7.

TABLE 3
Machines tested

Machine	*mindim*	*gemmw*	*matmulp*
Cray–2	128/768	X	
Cray Y–MP	64	X	
Cray Y–MP C90	128/256	X	
DEC 3000 (OSF/1)	32	X	
DEC 5000 (Ultrix)	32	X	
IBM 3090 (VM, MVS, and AIX/ESA)	192/256	X	
IBM ES9000 (VM, MVS, and AIX/ESA)	192/256	X	
IBM RISC System 6000	192	X	X
Sequent Symmetry	32	X	X
Silicon Graphics Indigo	32	X	
SUN SparcStation	96	X	X
Intel iPSC–2	32		X

TABLE 4
Memory requirements for Strassen–Winograd routines

Implementation	$\beta \neq 0$ or A, B overlaps with C	$\beta = 0$ and A, B do not overlap with C
gemmw	$1.67N^2$	$0.67N^2$
Cray *gemms*	$2.34N^2$	$2.34N^2$
IBM ESSL *gemms*–real	not possible	$1.40N^2$
IBM ESSL *gemms*–complex	not possible	$1.70N^2$

As can be readily seen, in the real data cases, *_gemmw* is quite competitive with "hand tuned" matrix–matrix multiplication routines. It is much faster than the classical method, as is expected from complexity theory.

In the complex data case, *_gemmw* is not quite as competitive as in the real data case. It is still typically within 10% of the "hand tuned" Strassen–Winograd routines and uses much less auxiliary storage, however. An interesting aside is that the routine *_gemul3* (see §3) is faster than *_gemm* for all machines we have tried to date except ones with DEC Alpha chips as the CPU. This anomaly is due to the cache design on the current Alpha chips.

We noted in §3 that there are two different ways of implementing complex matrix–matrix multiplication. Both Cray and IBM implemented their Strassen–Winograd routines using more auxiliary storage than *_gemmw*. We implemented *_gemmw* both ways. Our two implementations never differed by more than 3% in running time on any machine, including both Cray and IBM mainframes. We suspect that whatever programming tricks were used in the Cray and IBM routines could be applied to *_gemmw* in order to speed up the complex versions of *_gemmw* while still using the theoretical minimum amount of auxiliary storage.

Table 5 contains run times on a Cray 2 with 4 processors and a Cray Y–MP C90 with 4, 8, and 16 processors. Experience on the Cray–2 shows that that *mindim* should be increased as a function of the number of processors. As can be seen from the table, SGEMMW+SGEMM did fairly well up through 8 processors. Beyond 8 Y–MP C90 processors, larger problems or the algorithm from §4 should be used.

We also tested *matmulp* (see §4) on several machines. We simulated slow access to data by using the Linda system on an 18 processor Sequent computer. Table 6 contains elapsed times for 64 bit, real data using 1, 2, 4, 8, 12, and 16 processors. The matrix A is 512×512 in this case, which was the largest problem we could run on this machine conveniently using Linda. While the classical algorithm may have a very impressive parallel efficiency (approaching 100%), it is quite slow in comparison with the hybrid algorithm. There is every reason to believe that if larger problems are run, then the parallel efficiency of the hybrid method will also approach 100%.

6. Conclusions. In this paper, we addressed three issues. The first was the design of a a highly portable version of the Winograd variant of Strassen's matrix–matrix multiplication algorithm that uses little auxiliary storage. The second was an efficient implementation of classical matrix–matrix multiplication for complex data. The third was parallel implementation.

The serial code is sufficiently flexible so that only one source code is needed for four data types: single and double precision of either real or complex. It also is capable of using the BLAS (the ones provided with LAPACK or a proprietary version), Cray's Scientific and Math Library, IBM's ESSL, NAG, or a library of the user's choice. It also handles different naming and calling conventions transparently. Numerical experiments show that this is a very good algorithm to use instead of the classical one for problems of even a moderate size.

A Linda–C specific parallel implementation of a hybrid algorithm is also described here. Logic and simple numerical experiments show that this is better than a straight forward parallel implementation of either the classical or Strassen–Winograd algorithms when loads and stores of submatrices are expensive.

Our serial code strongly supports the argument that just writing numerical libraries in Fortran and assembly language is obsolete from a software engineering or human productivity point of view. This is a case where C provides a superior

solution, particularly when combined with computational kernels of mixed languages.

The portable Strassen–Winograd solution presented here competes well against hardware specific codes, especially on larger problems which motivated this research in the first place. When auxiliary storage is used as a measure, our code is greatly superior to hardware specific codes, in some cases by as much as a factor of 4.

Acknowledgments. We would like to thank Jérôme Jaffré of INRIA for his assistance early on in this project.

The source code for GEMMW is available either by sending the message *send gemmw from linalg* to *netlib@na-net.ornl.gov* or by ftp'ing the file *linalg/gemmw.shar* directly from a netlib repository.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, and J. Du Croz, *LAPACK Users' Guide*, SIAM Books, Philadelphia, 1992.

[3] D. H. Bailey, *Extra high speed matrix multiplication on the Cray-2*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 603–607.

[4] D. H. Bailey, K. Lee, and H. Simon, *Using Strassen's algorithm to accelerate the solution of linear systems*, J. Supercomp., 4 (1990), pp. 357–371.

[5] N. Carriero and D. Gelernter, *Linda in context*, Comm. ACM, 32 (1989), pp. 444–458.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.

[7] C. C. Douglas and J. Douglas, *A unified convergence theory for abstract multigrid or multilevel algorithms, serial and parallel*, SIAM J. Numer. Anal., 30 (1993), pp. 136–158.

[8] N. J. Higham, *Exploiting fast matrix multiplication within the Level 3 BLAS*, ACM Trans. Math. Soft., 16 (1990), pp. 352–368.

[9] J. Madsen, *Volume 3: UNICOS Math and Scientific Library Reference Manual (SR-2081)*, Cray Research, Inc., version 6.0 ed., 1990.

[10] L. Mason and M. E. Sliva, *Engineering and Scientific Subroutine Library: Guide and Reference, Version 2*, IBM Corporation, Kingston, NY, 1.0 ed., 1992.

[11] NAG, *NAG Fortran Library Manual*, Numerical Algorithms Group, Ltd., Oxford, UK, Mark 14 ed., 1990.

[12] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[13] S. Winograd, *Some remarks on fast multiplication of polynomials*, in Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, New York, 1973, pp. 181–196.

TABLE 5
CPU time for simple parallel 64 bit real computation on the Cray 2 and Y–MP C90

Number of processors	Size	Classical algorithm SGEMM	Strassen SGEMMS	SGEMMW + SGEMM
Cray 2	1058	1.82	1.43	2.42
4 CPU's	1269	2.83	2.20	2.45
	1522	4.06	3.48	6.25
	1826	7.61	5.39	5.57
Y–MP C90	1058	0.66	0.54	0.69
4 CPU's	1269	1.14	0.89	0.99
4.167 ns	1522	1.96	1.47	1.53
clock	1826	3.41	2.42	2.44
Y–MP C90	1058	0.33	0.28	0.30
8 CPU's	1269	0.57	0.45	0.47
4.167 ns	1522	0.98	0.75	0.75
clock[1]	1826	1.70	1.24	1.27
Y–MP C90	1058	0.17	0.17	0.18
16 CPU's	1269	0.29	0.27	0.26
4.167 ns	1522	0.50	0.40	0.41
clock[1]	1826	0.86	0.68	0.69

[1] $mindim = 256$ for 8 and 16 processors.

TABLE 6
Elapsed time for parallel on a Sequent Symmetry (Linda–C tuple space)

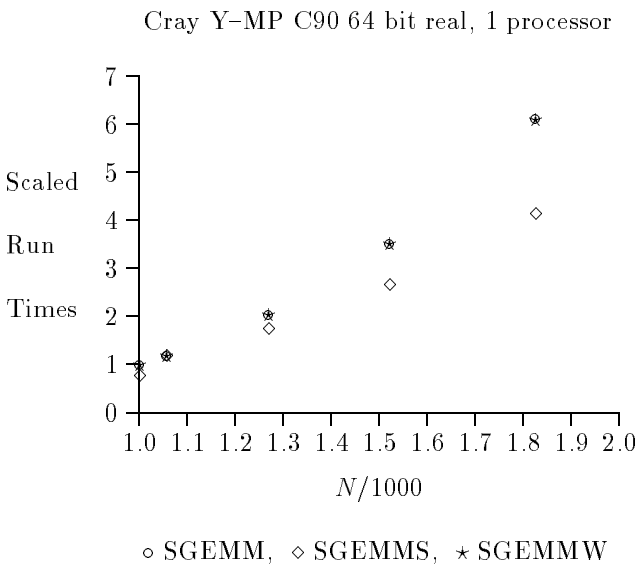
Number of processors	Using *matmulp/DGEMMW* Time	Speedup	% Efficient	Using classical DGEMM only Time	Speedup	% Efficient
1	543.28			1532.14		
2	277.71	1.9562	97.81	774.02	1.9794	98.97
4	159.49	3.4063	85.15	388.51	3.9436	98.59
8	86.84	6.2561	78.20	195.78	7.8258	97.82
12	55.93	9.7135	80.94	131.94	11.6123	96.76
16	49.55	10.9642	68.52	102.02	15.0180	93.86

64 bit real data, 500×500 matrices.

TABLE 7
Scaling factors in scaled run times figures

Machine	Real data	Complex data
Cray Y–MP C90	2.22	8.89
Cray Y–MP	6.40	25.60
DEC 3000	205.49	359.92
IBM RS/6000-560 (ESSL)	23.09	88.87
IBM RS/6000-560 (BLAS)	23.07	226.62*
IBM RS/6000-560 (NAG)	23.13	226.50*
IBM ES9000, Model 982	4.14	16.28

* Note that on machines with ESSL installed, the complex data scaling factors are the same as the ESSL one.



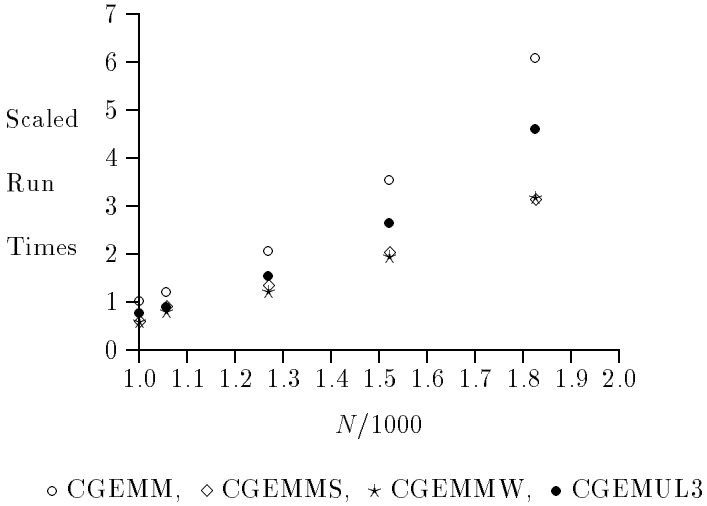
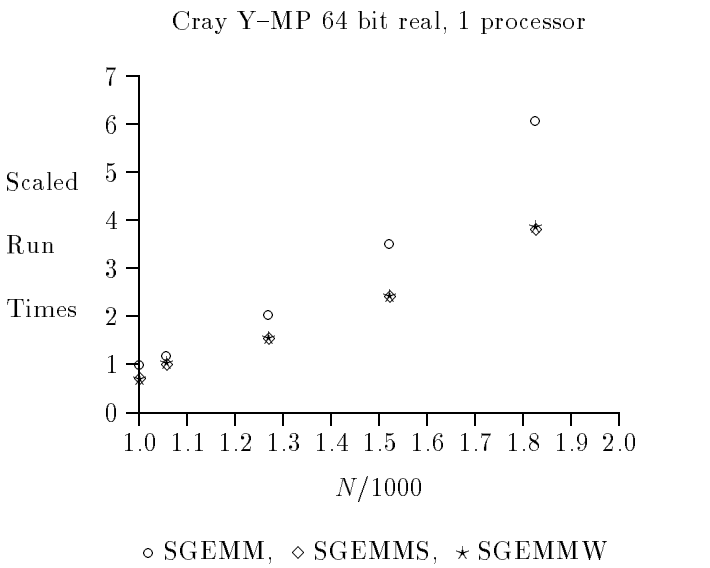


FIG. 3. *Single processor Cray Y–MP C90*



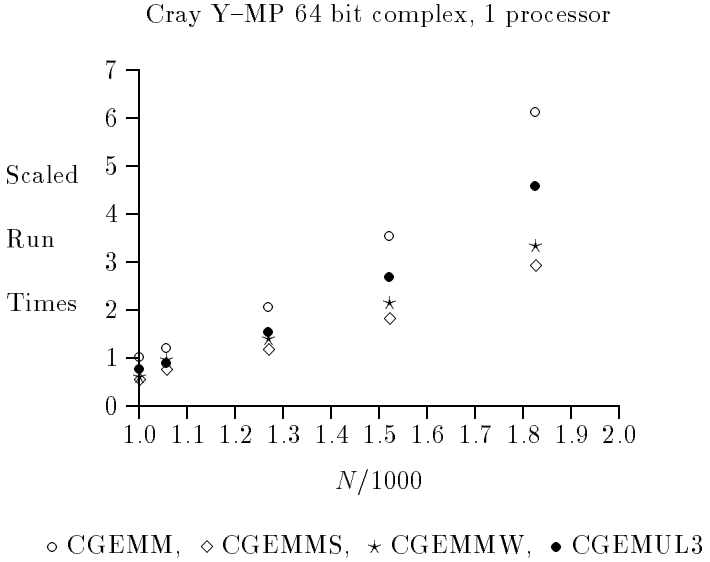
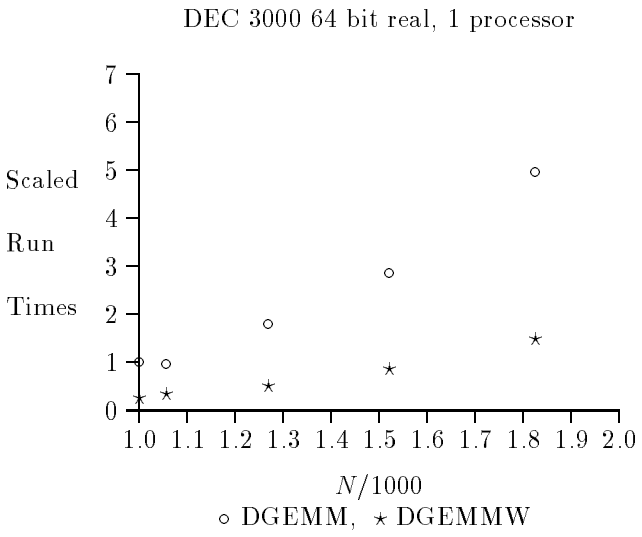


FIG. 4. *Single processor Cray Y–MP*



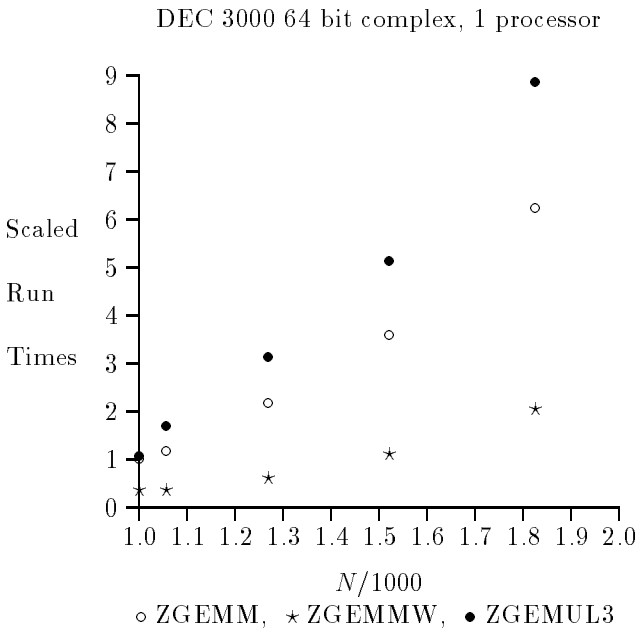
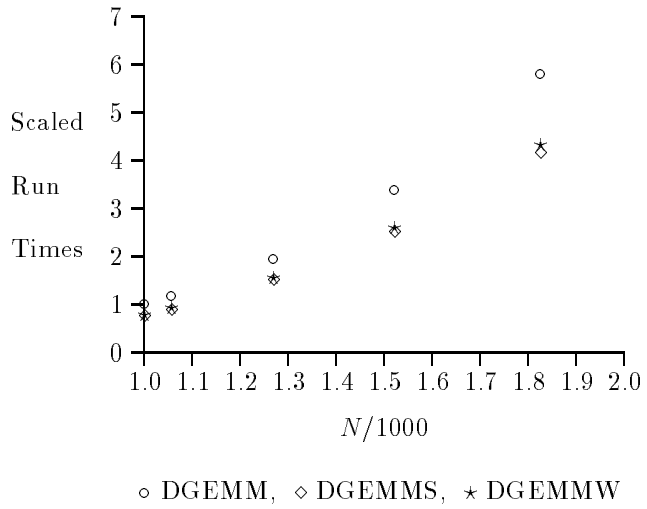


FIG. 5. *DEC 3000 (OSF/1)*

IBM RISC System/6000, Model 560 (ESSL) 64 bit real, 1 processor



IBM RISC System/6000, Model 560 (ESSL) 64 bit complex, 1 processor

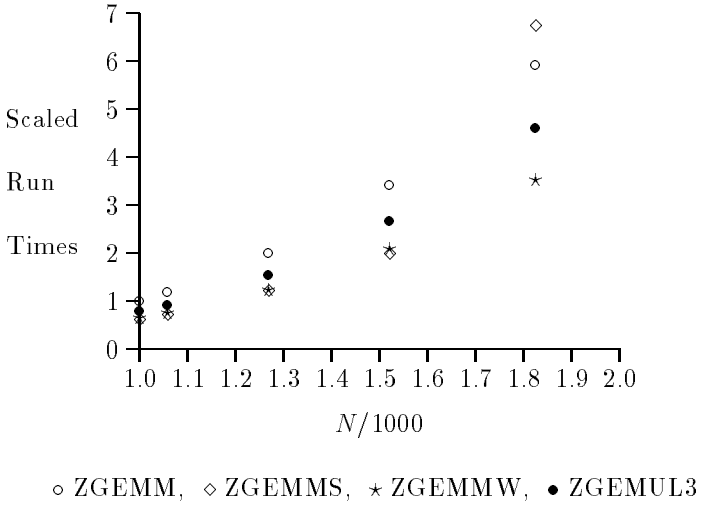
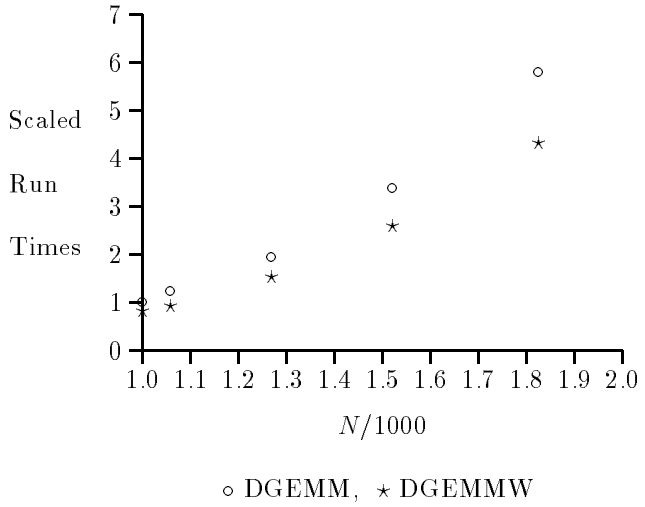


FIG. 6. *IBM RS6000/560 (ESSL)*

IBM RISC System/6000, Model 560 (BLAS) 64 bit real, 1 processor



IBM RISC System/6000, Model 560 (BLAS) 64 bit complex, 1 processor

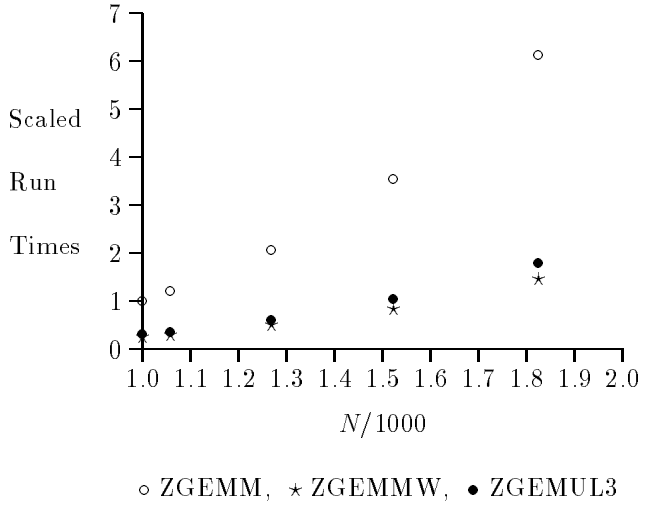
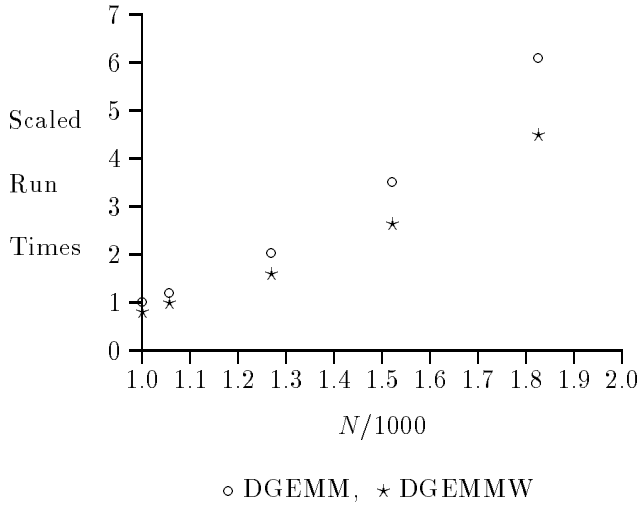


FIG. 7. *IBM RS6000/560 (BLAS)*

IBM RISC System/6000, Model 560 (NAG) 64 bit real, 1 processor



IBM RISC System/6000, Model 560 (NAG) 64 bit complex, 1 processor

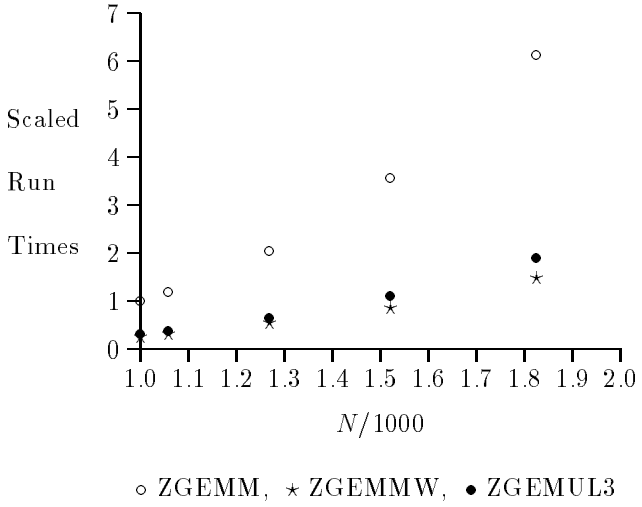
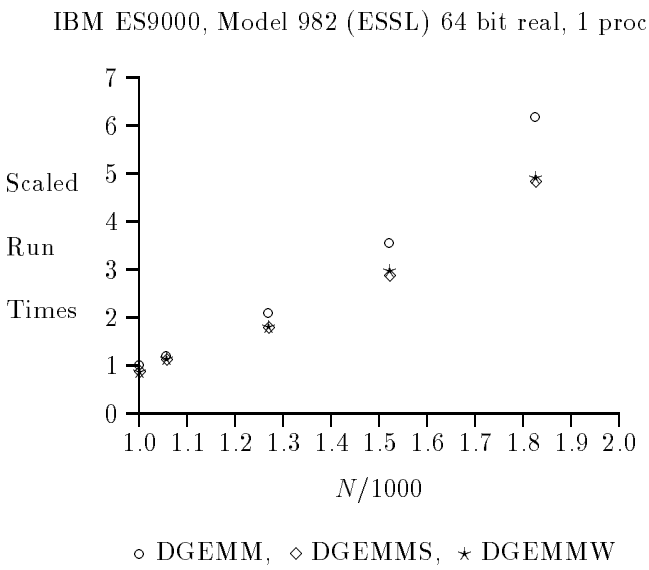


FIG. 8. *IBM RS6000/560 (NAG)*



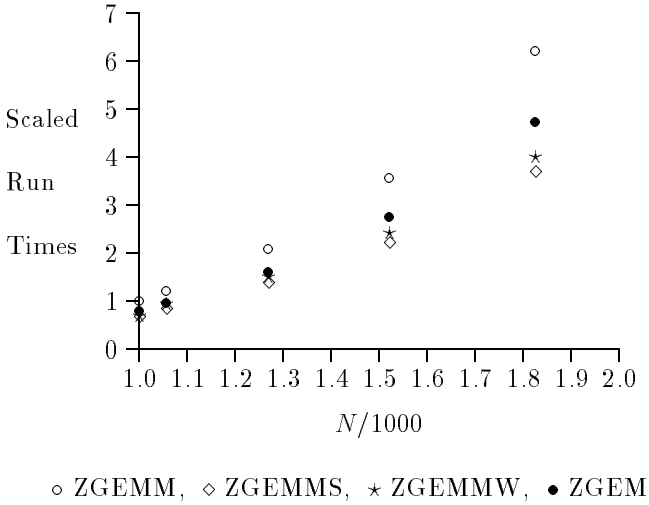


FIG. 9. *IBM ES9000/982 (ESSL)*