

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 classifications. 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

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

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 terribly different from standard multilevel methods 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.

[†] Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598, and Department of Computer Science, Yale University, P. O. Box 2158, New Haven, CT 06520. E-mail: *bells@watson.ibm.com* or *douglas-craig@cs.yale.edu*.

[‡] 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. E-mail: *smith-roger@cs.yale.edu*.

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

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

This is not a terribly 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, $96 \leq mindim \leq 192$ 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.

All variants of Strassen's algorithm are known to possess horrendous round off properties, leading to stability problems. Simple examples exist where, due to the unnatural submatrix additions, variants of Strassen's algorithm get the wrong answer while the classical algorithm gets the right answer. We acknowledge this deficiency. Our response is as follows: for many problems, due to the reduction of arithmetic operations, Strassen-Winograd has better round off properties than the classical method. 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. 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. Stability discussions are contained in [2] and [5] along with their references. An interesting application to solving linear systems of equations is contained in [1] and [3].

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

$$(2) \quad C \leftarrow \alpha \cdot op(A)op(B) + \beta \cdot C,$$

where

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

and

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

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

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

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

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

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	<i>_copy</i>	<i>_copy</i>	<i>_f06eff/gff</i>	—	$y = x$
VAXPY	<i>_axy</i>	<i>_axy</i>	<i>_f06ecf/gcf</i>	—	$y = \alpha x + y$
VYAX	<i>_yax*</i>	<i>_yax</i>	<i>_f06fdf/hdf</i>	—	$y = \alpha x$
MATMUL	<i>_gemm</i>	<i>_gemm</i>	<i>_f06yaf/zaf</i>	<i>_gemul3*</i>	$C = op(A)op(B)$
MATADD	<i>_geadd*</i>	<i>_geadd</i>	<i>_f06ctf/cwf</i>	—	$C = op(A) + op(B)$
MATSUB	<i>_gesub*</i>	<i>_gesub</i>	<i>_f06ctf/cwf</i>	—	$C = op(A) - op(B)$

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

The actual code is organized into three 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). 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 [6], ESSL [7], NAG [8], or any other library the user chooses. We used the BLAS distributed with LAPACK. 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.

The first option actually runs slightly faster ($\approx 2 - 3\%$) than the second, but uses 2.5 times as much storage in the process. Only a hardware manufacturer (e.g., [7] or [6]) would ever consider delivering the first approach without testing the second approach first.

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.

Unlike at least one commercially available Strassen–Winograd matrix multiplication routine for complex data [7], 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 $op(B)$ 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). The latter seems to be what most parallel computer manufacturers are designing now and what many members of the scientific community are using.

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. Assuming matrix loads and stores are 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. Graphically, processors 1 – 12 collaborate on one matrix multiplication. The matrices are partitioned initially as whole matrices:

$$\begin{bmatrix} aaaa \\ aaaa \\ aaaa \end{bmatrix} \times \begin{bmatrix} bbbb \\ bbbb \\ bbbb \\ bbbb \end{bmatrix} = \begin{bmatrix} cccc \\ cccc \\ cccc \end{bmatrix}.$$

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:

$$\begin{bmatrix} aaaa \\ aaaa \\ aaaa \end{bmatrix} \times \begin{bmatrix} bbb & | & bb \\ bbb & | & bb \\ bbb & | & bb \\ bbb & | & bb \end{bmatrix} = \begin{bmatrix} ccc & | & cc \\ ccc & | & cc \\ ccc & | & cc \end{bmatrix}.$$

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:

$$\begin{bmatrix} aa & | & aa \\ aa & | & aa \\ aa & | & aa \end{bmatrix} \times \begin{bmatrix} bbb & | & bb \\ bbb & | & bb \\ \hline bbb & | & bb \\ bbb & | & bb \end{bmatrix} = \begin{bmatrix} ccc & | & cc \\ ccc & | & cc \\ \hline ccc & | & cc \\ ccc & | & cc \end{bmatrix}.$$

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:

$$\begin{bmatrix} aa & | & aa \\ aa & | & aa \\ \hline aa & | & aa \end{bmatrix} \times \begin{bmatrix} bbb & | & bb \\ bbb & | & bb \\ \hline bbb & | & bb \\ bbb & | & bb \end{bmatrix} = \begin{bmatrix} ccc & | & cc \\ ccc & | & cc \\ \hline ccc & | & cc \\ ccc & | & cc \end{bmatrix}.$$

Finally, each processor performs its matrix multiplication independent of the others.

The user stores submatrices of A and B into the parallel data base. In our example code, the Linda system [4] 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 , and N), the partitioning (\mathcal{M} , \mathcal{K} , and \mathcal{N}), and the number of processors (p) to use.

Then *matmulp* creates

$$\left(\left\lfloor \frac{M-1}{\mathcal{M}} \right\rfloor + 1 \right) \times \left(\left\lfloor \frac{N-1}{\mathcal{N}} \right\rfloor + 1 \right)$$

parallel processes to compute submatrices of C which are placed in the database. Each parallel process runs a routine *DoCij* which is completely independent of the other *DoCij*'s. Each *DoCij* process creates $(K-1)/\mathcal{K}$ 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, where

$$p = \left(\left\lfloor \frac{M-1}{\mathcal{M}} \right\rfloor + 1 \right) \times \left(\left\lfloor \frac{K-1}{\mathcal{K}} \right\rfloor + 1 \right) \times \left(\left\lfloor \frac{N-1}{\mathcal{N}} \right\rfloor + 1 \right).$$

Each parallel *DoCij*, after using Strassen-Winograd to compute its submatrix term, accumulates the outputs of its child processes. Once accumulation is completed, each *DoCij* outputs its C submatrix to the database and quits. The result, C , remains in the distributed database.

5. Numerical experiments. Simple experiments were run on a variety of machines. The ones reported here are for squaring a matrix A . (The results of the computation were verified.) In order to fairly compare our times with those of other libraries, the matrix A was copied to matrix B before calling a matrix-matrix multiplication routine. The results reported here are representative of a larger set of experiments.

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 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 the following machines:

Machine	<i>mindim</i>	<i>gemmw</i>	<i>matmulp</i>
Cray-2	128/768	X	
Cray Y-MP	64	X	
Cray Y-MP C90	128/256	X	
IBM 3090S (VM, MVS, and AIX/ESA)	192/256	X	
IBM RISC System 6000	192	X	X
Sequent Symmetry	32	X	X
SUN SparcStation	96	X	X
Intel iPSC-2	32	X	

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

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

The IBM ESSL routine *gemms* assumes $\alpha = 1$, $\beta = 0$ in (2), and no overlapping of *A, B*, or *C*.

Table 3 contains the results of the example problem for 64 bit real data. The highlights of the table as follows:

Cray Competing against the hand tuned classical parallel matrix-matrix multiplication supplied as part of the Cray Math and Scientific Library turned out to be surprisingly difficult. Cray's Strassen code seems to do the parallelization in a different manner than ours. By varying the size of *mindim* as a function of the number of processors we are able to run as fast or faster than the Cray GEMMS, however.

IBM There are two features of this table worth noting. First, the ESSL routine DGEMMS (also a Strassen-Winograd routine) uses about 2.5 times as much extra storage as our DGEMMW. Second, we compiled our Strassen-Winograd routine using a number of libraries. On RISC System 6000's, the run times were all essentially equivalent to each other and the ESSL hand tuned routine DGEMMS. Hence, the free BLAS library distributed with the machine was the price-performance winner for a companion library to our routine. On the 3090S, the run times using VM/XA, MVS, or AIX/ESA were identical. The DGEMMW-ESSL version performed comparably to DGEMMS, but consistently outperformed DGEMMS for matrices larger than 4000×4000 .

SUN The SUN 4/600 used is not licensed for NAG and ESSL does not exist for SUN's. Hence, our table has four empty columns (the NAG columns could be filled in). Note that the Strassen-Winograd is an integer factor faster than the classical algorithm even for moderately sized problems. Since most members of the scientific computing community have SUN's on their desks, they should demand that this routine be added to the Level 3 BLAS immediately.

Tables 4 and 5 contain the results of the example problem for 64 bit complex data. The highlights of the tables are as follows:

Cray CGEMMW did not do particularly well in this case. The performance appears to be lost in the Fortran-90 version of CGEMUL3. This indicates a path for further work for us. However, CGEMMW uses about one fourth as much auxiliary storage as the Cray routine CGEMMS, potentially allowing us to

multiply larger matrices.

IBM The ESSL routine ZGEMMS runs a bit faster than ZGEMMW for moderate sized matrices. However, since ZGEMMS uses 2.5 times as much memory as ZGEMMW, it quickly runs out of memory or into paging situations that are painful. There are two cases where ZGEMMS fails to perform at all (the first is noted in [7] and the second is an undocumented bug). First, when A , B , or C overlap with one or more of the other matrices, ZGEMMS calls ZGEMM instead, resulting in terrible performance (see Table 5). Second, for reasons discussed in §3, if either A or B is in a read-only segment of memory, ZGEMMS produces a core dump. In both of these cases ZGEMMW produces the correct results. Specific to the RISC System 6000's, the run times for ZGEMMW+BLAS were consistently faster than ZGEMMW+ESSL or ZGEMMW+NAG.

SUN The difference in speed between ZGEMMW and ZGEMM is just as striking as in the real data case.

Table 6 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.

Table 7 contains run times on an 18 processor Sequent Symmetry 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 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 is 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 misc* to *netlib@na-net.ornl.gov* or by ftp'ing the file *misc/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] D. H. BAILY, *Extra high speed matrix multiplication on the Cray-2*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 603–607.
- [3] D. H. BAILY, K. LEE, AND H. SIMON, *Using Strassen's algorithm to accelerate the solution of linear systems*, J. Supercomp., 4 (1990), pp. 357–371.
- [4] N. CARRIERO AND D. GELERNTER, *Linda in context*, Comm. ACM, 32 (1989), pp. 444–458.
- [5] N. J. HIGHAM, *Exploiting fast matrix multiplication within the Level 3 BLAS*, ACM Trans. Math. Soft., 16 (1990), pp. 352–368.
- [6] J. MADSEN, *Volume 3: UNICOS Math and Scientific Library Reference Manual (SR-2081)*, Cray Research, Inc., version 6.0 ed., 1990.
- [7] L. MASON AND M. E. SLIVA, *Engineering and Scientific Subroutine Library: Guide and Reference, Version 2*, IBM Corporation, Kingston, NY, 1.0 ed., 1992.
- [8] NAG, *NAG Fortran Library Manual*, Numerical Algorithms Group, Ltd., Oxford, UK, Mark 14 ed., 1990.

TABLE 3
CPU time for 64 bit real computation

Machine	Size	Classical algorithm		Strassen ¹	_GEMMW +		
		BLAS ²	NAG ³		BLAS ²	NAG ³	ESSL ⁴
Cray 2 1 CPU	206	0.04		0.04	0.04		
	511	0.59		0.55	0.59		
	882	3.02		2.66	2.71		
	1058	5.21		6.18	4.74		
	1269	8.97		8.91	7.99		
	1522	15.47		13.56	13.74		
	1826	26.72		20.33	21.98		
Cray Y-MP 6 ns clock	206	0.06		0.05	0.05		
	511	0.85		0.67	0.84		
	882	4.39		3.18	3.20		
	1058	7.58		6.87	6.58		
	1269	13.06		10.46	9.90		
	1522	22.53		16.06	15.44		
	1826	38.91		25.10	24.61		
Cray Y-MP C90 1 CPU 4.167 ns clock	206	0.02		0.02	0.02		
	511	0.30		0.24	0.25		
	882	1.57		1.11	1.13		
	1058	2.69		1.92	2.01		
	1269	4.63		3.17	3.22		
	1522	7.97		5.20	5.26		
	1826	13.80		8.60	8.76		
IBM 3090S Vector	206	0.18		0.18	0.17		0.15
	511	2.61		1.79	1.81		1.49
	882	13.74		6.22	7.65		5.72
	1058	23.50		8.71	11.93		9.26
	1269	39.98		13.90	18.32		14.42
	1522	68.79		22.15	28.93		23.16
	1826	119.34		*	53.02		37.10
IBM RISC System 6000-560	206	0.19	0.21	0.19	0.20	0.21	0.20
	511	2.98	3.06	2.68	2.66	2.87	2.67
	882	15.12	15.79	12.37	12.87	13.22	12.94
	1058	26.84	27.25	20.63	21.21	22.81	21.32
	1269	44.92	46.75	35.00	35.60	37.26	35.77
	1522	77.29	80.69	58.29	59.71	61.25	59.83
	1826	134.13	140.33	96.29	99.53	102.77	100.12
SUN 4/600 1 CPU	206	4.28			1.67		
	511	64.40			17.50		
	882	334.90			64.56		
	1058	580.58			109.38		
	1269	1007.31			160.43		

¹ DGEMMS on IBM's, SGEMMS on Cray's; not available otherwise.

² SGEMM on Cray's, DGEMM on others; ESSL version on IBM's, Cray Math and Scientific Library on Cray's, and LAPACK BLAS on others.

³ F06YAF subroutine from Mark 14 library version. The vector version was not available to us when this paper was written and the scalar results were uninteresting on vector machines. We had no access to the SUN version.

⁴ Only exists on IBM machines (370 and POWER architectures).

* Not enough memory.

TABLE 4
CPU time for 64 bit complex computation

Machine	Size	Classical algorithm		Strassen ¹	_GEMMW +		
		BLAS ²	NAG ³		BLAS ²	NAG ³	ESSL ⁴
Cray 2 1 CPU ⁵	206	0.16		0.17	0.16		
	511	2.45		1.85	2.45		
	882	12.59		8.46	11.30		
	1058	21.79		19.11	19.62		
	1269	37.44		28.28	33.46		
	1522	64.67		42.91	57.55		
	1826	111.70		62.94	90.73		
Cray Y-MP 1 CPU 6 ns clock	206	0.23		0.16	0.20		
	511	3.42		2.02	2.64		
	882	17.56		9.28	11.76		
	1058	34.24		17.46	20.79		
	1269	52.24		28.19	33.52		
	1522	90.64		45.01	54.92		
	1826	157.63		71.34	90.88		
Cray Y-MP C90 1 CPU 4.167 ns clock	206	0.08		0.06	0.07		
	511	1.19		0.74	0.93		
	882	6.10		3.36	4.37		
	1058	10.56		5.81	7.31		
	1269	18.15		9.56	12.02		
	1522	31.28		15.68	20.03		
	1826	54.15		25.96	33.84		
IBM 3090S Vector	206	0.41		0.40	0.41		0.41
	511	5.72		4.12	4.09		4.10
	882	29.26		14.70	15.79		15.77
	1058	51.15		22.23	26.73		26.73
	1269	85.04		34.81	40.03		40.03
	1522	146.19		54.60	63.07		63.07
	1826	255.83		91.74	103.44		103.44
IBM RISC System 6000-560	206	0.66	0.70	0.62	0.67	0.72	0.69
	511	9.38	9.72	8.32	8.81	9.22	8.87
	882	46.61	48.65	37.70	41.47	42.76	41.98
	1058	82.70	85.49	62.84	67.52	72.58	68.06
	1269	137.43	144.04	106.26	115.47	119.95	115.97
	1522	237.05	248.40	*	187.62	193.14	188.91
	1826	407.15	437.85	*	329.45	330.34	332.92
SUN 4/600 1 CPU	206	11.97			4.92		
	511	184.32			48.70		
	882	962.73			165.59		

¹ ZGEMMS on IBM's, CGEMMS on Cray's; not available otherwise.

² CGEMUL3 on Cray's and ZGEMUL3 on all others. Underlying that is CGEMM on Cray's, ZGEMM on others; ESSL version on IBM's, Cray Math and Scientific Library on Cray's, and LAPACK BLAS on others. Also, *mindim* = 256 was used for the IBM 3090S vector entries.

³ F06ZAF subroutine from Mark 14 library version. The vector version was not available to us when this paper was written and the scalar results were uninteresting on vector machines. We had no access to the SUN version.

⁴ Only exists on IBM machines (370 and POWER architectures).

⁵ *mindim* = 768.

* Not enough memory.

TABLE 5
CPU time for 64 bit complex computation - overlapped matrices

Machine	Size	ZGEMMS ¹	ZGEMMW ²
IBM	206	0.77	0.41
3090S	511	11.45	4.10
Vector	735	34.18	9.81
	882	59.71	15.77
	3784	4648.10	659.14
IBM	206	1.08	0.66
RISC	511	16.61	8.75
System	735	50.47	24.35
6000-560	882	85.34	41.49
	1058	148.65	67.53
	1269	253.70	115.44
	1522	437.63	188.54

¹ IBM's ESSL Strassen-Winograd subroutine.

² Using ZGEMUL3 and DGEMM.

TABLE 6
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 7
CPU time for parallel on a Sequent Symmetry (Linda-C tuple space)

Number of processors	Using <i>matmulp/DGEMMW</i>			Using classical DGEMM only		
	Time	Speedup	% Efficient	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 data, 500×500 matrices.