Implementation of Strassen's Algorithm for Matrix Multiplication [1]

Steven Huss-Lederman [2]
Elaine M. Jacobson [3]
Jeremy R. Johnson [4]
Anna Tsao [5]
Thomas Turnbull [6]

August 1, 1996

[1] This work was partially supported by the Applied and Computational Mathematics Program, Defense Advanced Research Projects Agency, under Contract P-95006.

[2] Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706, *Phone:* (608) 262-0664, *FAX:* (608) 262-9777, *email:* `lederman@cs.wisc.edu`

[3] Center for Computing Sciences, 17100 Science Dr., Bowie, MD 20715, *Phone:* (301) 805-7435, *FAX:* (301) 805-7602, *email:* `emj@super.org`

[4] Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA 19104, *Phone:* (215) 895-2893, *FAX:* (610) 647-8633, *email:* `jjohnson@king.mcs.drexel.edu`

[5] Center for Computing Sciences, 17100 Science Dr., Bowie, MD 20715, *Phone:* (301) 805-7432, *FAX:* (301) 805-7602, *email:* `anna@super.org`

[6] Center for Computing Sciences, 17100 Science Dr., Bowie, MD 20715, *Phone:* (301) 805-7358, *FAX:* (301) 805-7602, *email:* `turnbull@super.org`

Abstract

In this paper we report on the development of an efficient and portable implementation of Strassen's matrix multiplication algorithm. Our implementation is designed to be used in place of DGEMM, the Level 3 BLAS matrix multiplication routine. Efficient performance will be obtained for all matrix sizes and shapes and the additional memory needed for temporary variables has been minimized. Replacing DGEMM with our routine should provide a significant performance gain for large matrices while providing the same performance for small matrices. We measure performance of our code on the IBM RS/6000, CRAY YMP C90, and CRAY T3D single processor, and offer comparisons to other codes. Our performance data reconfirms that Strassen's algorithm is practical for realistic size matrices. The usefulness of our implementation is demonstrated by replacing DGEMM with our routine in a large application code.

Keywords: *matrix multiplication, Strassen's algorithm, Winograd variant, Level 3 BLAS*

1 Introduction

The multiplication of two matrices is one of the most basic operations of linear algebra and scientific computing and has provided an important focus in the search for methods to speed up scientific computation. Its central role is evidenced by its inclusion as a key primitive operation in portable libraries, such as the Level 3 BLAS [7], where it can then be used as a building block in the implementation of many other routines, as done in LAPACK [1]. Thus, any speedup in matrix multiplication can improve the performance of a wide variety of numerical algorithms.

Much of the effort invested in speeding up practical matrix multiplication implementations has concentrated on the well-known standard algorithm, with improvements seen when the required inner products are computed in various ways that are better-suited to a given machine architecture. Much less effort has been given towards the investigation of alternative algorithms whose asymptotic complexity is less than the $\Theta(m^3)$ operations required by the conventional algorithm to multiply $m \times m$ matrices. One such algorithm is Strassen's algorithm, introduced in 1969 [19], which has complexity $\Theta(m^{\lg(7)})$, where $\lg(7) \approx 2.807$ and $\lg(x)$ denotes the base 2 logarithm of x.

Strassen's algorithm has long suffered from the erroneous assumptions that it is not efficient for matrix sizes that are seen in practice and that it is unstable. Both of these assumptions have been questioned in recent work. By stopping the Strassen recursions early and performing the bottom-level multiplications using the traditional algorithm, competitive performance is seen for matrix sizes in the hundreds in Bailey's FORTRAN implementation on the CRAY 2 [2], Douglas, et al.'s [8] C implementation of the Winograd variant of Strassen's algorithm on various machines, and IBM's ESSL library routine [16]. In addition, the stability analyses of Brent [4] and then Higham [11, 12] show that Strassen's algorithm is stable enough to be studied further and considered seriously in the development of high-performance codes for matrix multiplication.

A useful implementation of Strassen's algorithm must first efficiently handle matrices of arbitrary size. It is well known that Strassen's algorithm can be applied in a straightforward fashion to square matrices whose order is a power of two, but issues arise for matrices that are non-square or those having odd dimensions. Second, establishing an appropriate cutoff criterion for stopping the recursions early is crucial to obtaining competitive performance on matrices of practical size. Finally, excessive amounts of memory should not be required to store temporary results. Earlier work addressing these issues can be found in [2, 3, 4, 5, 8, 9, 10, 11, 17, 19].

In this paper we report on our development of a general, efficient, and portable implementation of Strassen's algorithm that is usable in any program in place of calls to DGEMM, the Level 3 BLAS matrix multiplication routine. Careful consideration has been given to all of the issues mentioned above. Our analysis provides an improved cutoff condition for rectangular matrices, a demonstration of the viability of dynamic peeling, a simple technique for dealing with odd matrix dimensions that had previously been dismissed [8], and a reduction in the amount of memory required for temporary variables.

We measure performance of our code on the IBM RS/6000, CRAY YMP C90, and CRAY T3D single processor and examine the results in several ways. Comparisons with machine-specific implementations of DGEMM reconfirm that Strassen's algorithm can provide an improvement over the standard algorithm for matrices of practical size. Timings of our code using several different cutoff criteria are compared, demonstrating the benefits of our new technique. Comparisons to the Strassen routine in the ESSL RS/6000 and the CRAY C90 libraries and the implementation of Douglas, et al., show that competitive performance can be obtained in a portable code that uses the previously untried dynamic peeling method for odd-sized matrices. This is especially significant since for certain cases our memory requirements have been reduced by 40 to more than 70 percent over these other codes.

The remainder of this paper is organized as follows. Section 2 reviews Strassen's algorithm. In Section 3 we describe our implementation and address implementation issues related to cutoff, odd dimensions, and memory usage. Performance of our implementation is examined in Section 4, where we also report on using our Strassen code for the matrix multiplications in an eigensolver application. We offer a summary and conclusions in Section 5.

2 Strassen's Algorithm

Here we review Strassen's algorithm and some of its key algorithmic issues within the framework of an operation count model. The interested reader is referred to [14] for more details on this and other models, some of which also take into account memory access patterns, possible data reuse, and differences in speed between different arithmetic operations. In this paper the simpler operation count model will meet our needs for discussion of the various issues that had an effect on our code design.

The standard algorithm for multiplying two $m \times m$ matrices requires m^3 scalar multiplications and $m^3 - m^2$ scalar additions, for a total arithmetic operation count of $2m^3 - m^2$. In Strassen's now famous 1969 paper [19], he introduced an algorithm, stated there for square matrices, which is based on a clever way of multiplying 2×2 matrices using 7 multiplications and 18 additions/subtractions. His construction does not depend on the commutativity of the component multiplications and hence can be applied to block matrices and then used recursively.

If one level of Strassen's algorithm is applied to 2×2 matrices whose elements are $m/2 \times m/2$ blocks and the standard algorithm is used for the seven block matrix multiplications, the total operation count is $7(2(m/2)^3 - (m/2)^2) + 18(m/2)^2 = (7/4)m^3 + (11/4)m^2$. The ratio of this operation count to that required by the standard algorithm alone is seen to be

$$\frac{7m^3 + 11m^2}{8m^3 - 4m^2}, \tag{1}$$

which approaches 7/8 as m gets large, implying that for sufficiently large matrices one level of Strassen's construction produces a 12.5% improvement over regular matrix multiplication. Applying Strassen's construction recursively leads to the complexity result stated in the introduction [6]. We remark that the asymptotic complexity does not depend on the number of additions/subtractions; however, reducing the number of additions/subtractions can have practical significance.

Winograd's variant of Strassen's algorithm (credited to M. Paterson) uses 7 multiplications and 15 additions/subtractions [10]. The algorithm partitions input matrices A and B into 2×2 blocks and computes $C = AB$ as

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

Stages (1) and (2) of the algorithm compute

$$\begin{array}{ll} S_1 = A_{21} + A_{22}, & T_1 = B_{12} - B_{11}, \\ S_2 = S_1 - A_{11}, & T_2 = B_{22} - T_1, \\ S_3 = A_{11} - A_{21}, & T_3 = B_{22} - B_{12}, \\ S_4 = A_{12} - S_2, & T_4 = B_{21} - T_2, \end{array}$$

and stages (3) and (4) compute, respectively, the seven products and seven sums

$$\begin{array}{llll}
P_1 = A_{11}B_{11}, & P_5 = S_3T_3, & U_1 = P_1 + P_2, & U_5 = U_3 + P_3, \\
P_2 = A_{12}B_{21}, & P_6 = S_4B_{22}, & U_2 = P_1 + P_4, & U_6 = U_2 + P_3, \\
P_3 = S_1T_1, & P_7 = A_{22}T_4, & U_3 = U_2 + P_5, & U_7 = U_6 + P_6. \\
P_4 = S_2T_2, & & U_4 = U_3 + P_7, &
\end{array}$$

It is easy to verify that $C_{11} = U_1$, $C_{12} = U_7$, $C_{21} = U_4$, and $C_{22} = U_5$. Further reduction in the number of multiplications and additions/subtractions of any Strassen-like algorithm based on 2×2 matrices is not possible [13, 18]. In the remainder of this paper, unless otherwise specified, we will mean the Winograd variant described above when referring to Strassen's algorithm.

To apply Strassen's algorithm to arbitrary matrices, first note that one level of recursion can easily be applied to rectangular matrices, provided that all of the matrix dimensions are even. Next, observe that we do not have to carry the recursions all the way to the scalar level; as noted in the introduction, one key element for obtaining an efficient implementation of Strassen's algorithm is to in fact stop the recursions early, switching to the standard algorithm when Strassen's construction no longer leads to an improvement. The test used to determine whether or not to apply another level of recursion is called the **cutoff criterion**.

Let $G(m,n)$ be the cost of adding or subtracting two $m \times n$ matrices and let $M(m,k,n)$ be the cost of multiplying an $m \times k$ matrix by a $k \times n$ matrix using the standard matrix multiplication algorithm. Then, assuming m, k, and n are even, the cost of Strassen's algorithm, $W(m,k,n)$, to multiply an $m \times k$ matrix A by a $k \times n$ matrix B satisfies the following recurrence relation

$$W(m,k,n) = \begin{cases} M(m,k,n), & \text{when } (m,k,n) \text{ satisfies the cutoff criterion} \\ 7W\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right) + 4G\left(\frac{m}{2},\frac{k}{2}\right) + 4G\left(\frac{k}{2},\frac{n}{2}\right) + 7G\left(\frac{m}{2},\frac{n}{2}\right), & \text{otherwise.} \end{cases} \tag{2}$$

Throughout the remainder of this section we will model costs by operation counts, implying that $M(m,k,n) = 2mkn - mn$ and $G(m,n) = mn$.

If A and B are of size $2^d m' \times 2^d k'$ and $2^d k' \times 2^d n'$, respectively, then Strassen's algorithm can be used recursively d times. If we choose to stop recursion after these d applications of Strassen's algorithm, so that the standard algorithm is used to multiply the resulting $m' \times k'$ and $k' \times n'$ matrices, then

$$\begin{aligned} W(2^d m', 2^d k', 2^d n') &= 7^d(2m'k'n' - m'n') + \\ &\quad (7^d - 4^d)(4m'k' + 4k'n' + 7m'n')/3. \end{aligned} \tag{3}$$

For the square matrix case ($m' = k' = n'$), (3) simplifies to

$$W(2^d m') \equiv W(2^d m', 2^d m', 2^d m') = 7^d(2(m')^3 - (m')^2) + 5(m')^2(7^d - 4^d). \tag{4}$$

If Strassen's original version of the algorithm had been used in the above development, (4) would become

$$S(2^d m') \;=\; 7^d(2(m')^3 - (m')^2) \;+ 6(m')^2(7^d - 4^d). \tag{5}$$

In order to establish the proper cutoff point for both the square and rectangular case, we need to characterize the set of positive (m, k, n) such that using the standard algorithm alone is less costly than applying one level of Strassen's recursion followed by the standard method. This is equivalent to finding the solutions to the inequality

$$M(m,k,n) \le 7M\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right) + 4G\left(\frac{m}{2},\frac{k}{2}\right) + 4G\left(\frac{k}{2},\frac{n}{2}\right) + 7G\left(\frac{m}{2},\frac{n}{2}\right). \tag{6}$$

Using operation counts this becomes

$$mkn \le 4(mk + kn + mn), \tag{7}$$

which is equivalent to

$$1 \le 4(1/n + 1/m + 1/k). \tag{8}$$

It is easy to completely characterize the positive integer solutions to (7) and (8). In the square ($m = k = n$) case we obtain $m \le 12$. Thus, we should switch to regular matrix multiplication whenever the remaining matrix multiplications involve matrices whose order is 12 or less.

Utilizing equations (4) and (5) we can now begin to see the improvement possible using cutoffs and using Winograd's variant instead of Strassen's original algorithm. First, observe that (4) is an improvement over (5) for all recursion depths d and all m', since their difference is $(m')^2(7^d - 4^d)$. The limit as d goes to infinity of the ratio of equation (5) to equation (4) is $(5+2m')/(4+2m')$. Thus, for large square matrices, improvement of (4) over (5) is 14.3% when full recursion is used ($m' = 1$), and between 5.26% and 3.45% as m' ranges between 7 and 12. These are the values of m' that would occur when the optimal cutoff value of 12 for square matrices is employed. To see how valuable the use of cutoffs can be, we can also compute, using equation (4), the ratio of the operation counts for Winograd's variant on square matrices without cutoff to that with cutoff 12. For matrices of order 256 this means we compute the ratio (4) with $d = 8, m' = 1$ to (4) with $d = 5, m' = 8$, obtaining a 38.2% improvement using cutoffs.

Returning to establishing the cutoff criteria, the situation is more complicated for rectangular matrices. For more details see [14]; here we illustrate with an example. If $m = 6, k = 14, n = 86$, (7) is not satisfied; thus recursion should be used when multiplying 6×14 and 14×86 matrices. This shows that there are situations where it is beneficial to apply Strassen's algorithm even though one of the matrix dimensions is smaller than the optimal cutoff of 12 for square matrices. Therefore, at least theoretically, when considering rectangular matrices, the cutoff criterion (7) should be used instead of the simpler condition, $m \leq 12$ or $k \leq 12$ or $n \leq 12$, which has been used by others [3, 8]. Alternatively, instead of using the operation count model to predict the proper cutoff condition, one can empirically determine the appropriate cutoff in a manner very similar to the theoretical analysis. This will require a more complicated set of experiments for rectangular matrices than for square. A discussion of empirical results of this type can be found in Section 3.4; the actual measurements will be done in Section 4.2.

Finally, for matrices with odd dimensions, some technique must be applied to make the dimensions even, apply Strassen's algorithm to the altered matrix, and then correct the results. Originally, Strassen suggested padding the input matrices with extra rows and columns of zeros, so that the dimensions of all the matrices encountered during the recursive calls are even. After the product has been computed, the extra rows and columns are removed to obtain the desired result. We call this approach **static padding**, since padding occurs before any recursive calls to Strassen's algorithm. Alternatively, each time Strassen's algorithm is called recursively, an extra row of zeros can be added to each input with an odd row-dimension and an extra column of zeros can be added for each input with an odd column-dimension. This approach to padding is called **dynamic padding** since padding occurs throughout the execution of Strassen's algorithm. A version of dynamic padding is used in [8].

Another approach, called **dynamic peeling**, deals with odd dimensions by stripping off the extra row and/or column as needed, and adding their contributions to the final result in a later round of fixup work. More specifically, let A be an $m \times k$ matrix and B be a $k \times n$ matrix. Assuming that m, k, and n are all odd, A and B are partitioned into the block matrices

$$A = \left(\begin{array}{c|c} A_{11} & a_{12} \\ \hline a_{21} & a_{22} \end{array}\right) \qquad \text{and} \qquad B = \left(\begin{array}{c|c} B_{11} & b_{12} \\ \hline b_{21} & b_{22} \end{array}\right),$$

where A_{11} is an $(m-1) \times (k-1)$ matrix, a_{12} is a $(m-1) \times 1$ matrix, a_{21} is a $1 \times (k-1)$ matrix, a_{22} is a 1×1 matrix and B_{11} is an $(k-1) \times (n-1)$ matrix, b_{12} is a $(k-1) \times 1$ matrix, b_{21} is a $1 \times (n-1)$ matrix, b_{22} is a 1×1 matrix. The product $C = AB$ is computed as

$$\left(\begin{array}{c|c} C_{11} & c_{12} \\ \hline c_{21} & c_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11}B_{11} + a_{12}b_{21} & A_{11}b_{12} + a_{12}b_{22} \\ \hline a_{21}B_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{array}\right), \tag{9}$$

where $A_{11}B_{11}$ is computed using Strassen's algorithm, and the other computations constitute the fixup work. To our knowledge, the dynamic peeling method had not been previously tested through actual implementation, and in fact its usefulness had been questioned [8]. However, our operation count analysis in [14] showed it to be superior to dynamic padding. This indicated that it could be competitive in practice with other approaches, and that further study was needed. Thus, we chose to use this technique in our implementation of Strassen's algorithm. In Section 3.3 we will discuss how we actually coded these computations.

3 Implementation Issues

In this section we discuss DGEFMM, our implementation of Strassen's algorithm. Section 3.1 describes the interface to DGEFMM, and Section 3.2 shows how the equations in the four stages of Strassen's algorithm are implemented. The latter section also discusses the amount of memory needed for temporary storage. Section 3.3 discusses our treatment of odd matrix dimensions. Finally, in Section 3.4, a method for determining the appropriate cutoff criterion for a particular machine is presented.

3.1 Interface

Since our implementation of Strassen's algorithm is intended to be used in place of the Level 3 BLAS matrix multiplication routine DGEMM, we adopt the input and output specifications of DGEMM. DGEMM computes $C \leftarrow \alpha * \mathrm{op}(A) * \mathrm{op}(B) + \beta * C$, where α and β are scalars, $\mathrm{op}(A)$ is an $m \times k$ matrix, $\mathrm{op}(B)$ is an $k \times n$ matrix, C is an $m \times n$ matrix, and $\mathrm{op}(X)$ can either be X or X^T (transpose of X).

All code was written in C utilizing the BLAS, except for some kernel operations on some machines where we found the FORTRAN implementation to be superior. Matrices are stored in column-major order, as in FORTRAN, to facilitate the interface with the BLAS.

3.2 Temporary Allocation and Memory Usage

Intermediate results required by Strassen's algorithm need to be stored in temporary locations. Strassen's algorithm as stated in Section 2 uses four temporary variables in stage (1), four temporary variables in stage (2), seven temporary variables in stage (3), and seven temporary variables in stage (4). The temporary variables in stage (1) hold $m/2 \times k/2$ matrices, those in stage (2) hold $k/2 \times n/2$ matrices, and those in stages (3) and (4) hold $m/2 \times n/2$ matrices. A straightforward implementation of one level of recursion of Strassen's algorithm thus requires $mk + kn + (14/4)mn$ temporary memory space in addition to the input and output variables. If all recursive calls in Strassen's algorithm use this same approach, then the total amount of extra storage is bounded by $(4mk + 4kn + 14mn)\sum_{i=1}^{\infty} 1/4^i = (4mk + 4kn + 14mn)/3$.

Clearly this can be greatly reduced if temporary memory space is effectively reused. Using Strassen's original algorithm, Bailey, et al. [3] devised a straightforward scheme that reduces the total memory requirements to $(mk+kn+mn)/3$. Since Winograd's variant nests the additions/subtractions in stage (4) it is not clear that a similar reduction is possible. Below we discuss two computation schemes, both of which are used in our implementation. The first will demonstrate that an even lower memory requirement is attainable for the case where the input parameter $\beta = 0$. This scheme is similar to the one used in the implementation of Winograd's variant, DGEMMW, reported in Douglas, et al., [8], where additional storage requirements when $\beta = 0$ are slightly more than $(m \max(k,n) + kn)/3$. Our technique requires $(m \max(k,n) + kn)/3$ in this case. For the general case, i.e., $\beta \neq 0$, the temporary storage requirement in the Douglas, et al., implementation is approximately $mn + (mk + kn)/3$. Using our second computation scheme allows us to reduce this to the lower amount attained for the simpler case by Bailey, et al., namely $(mk + kn + mn)/3$. We will see later that the key to obtaining this memory reduction is the recursive use of the multiply-accumulate operation that DGEFMM supports.

Our first scheme for performing Strassen's algorithm, STRASSEN1, is a straightforward schedule for performing the computations as described in Section 2, and is similar to that used in [8]. See [14] for a detailed description. In the general case, we require space for six temporary variables, R_1 through R_6, with total size $m \max(k,n)/4 + mn + kn/4$. If the seven products are computed recursively using the same algorithm, then the total additional memory is bounded by $(4mn + m \max(k,n) + kn)/3$. Note that if $m = k = n$, then the required space is $2m^2$. Also when $\beta = 0$, the computation order, designed in fact to this purpose, allows use of the memory space allocated for matrix C to hold the temporary variables R_3, R_4, R_5, and R_6. Thus, only two temporary variables are required, and the bound for extra space is reduced to $(m \max(k,n) + kn)/3$. For the square case $(m = k = n)$, this becomes $2m^2/3$.

Our alternate computation scheme, STRASSEN2, is presented in Figure 1. With each computation step indicated, we show the temporary variable stored to, as well as the relationship to the variables used in the description of the algorithm in Section 2. Although STRASSEN2 performs some additional arithmetic, multiplication by α and β, and some accumulation operations, our analysis [14] and empirical results (Section 4) suggest that no time penalty is paid for these additional operations. By allowing the multiply-accumulate operation, such as $C_{11} \leftarrow C_{11} + \alpha A_{12} B_{21}$, we can rearrange the computations in Strassen's algorithm so as to essentially use C's memory space for intermediate results even when $\beta \neq 0$. We can thus complete the computations using only three temporaries, R_1, R_2, and R_3, the minimum number possible [14]. Notice that R_1 only holds subblocks of A, R_2 only holds subblocks of B, and R_3 only holds subblocks of C, so that their sizes are $mk/4$, $nk/4$, and $mn/4$, respectively. The total memory requirement for STRASSEN2 is thus bounded by $(mn + km + kn)/3$. When $m = k = n$, this gives m^2.

In Table 1 we show the memory requirements for multiplying order m matrices for various implementations of Winograd's Strassen variant. We can also see from Table 1 that the best use of our two computation schemes might be to use STRASSEN1 when $\beta = 0$, and STRASSEN2 otherwise. This is reflected in the final entry of the table, where we see that our memory requirement of $2m^2/3$ in the case $\beta = 0$ equals the lowest requirement seen for the other implementations (DGEMMW), which is a 48 to 71 percent reduction over the

STRASSEN2$(A, B, \alpha, \beta; C)$

Implementation of Winograd's variant of Strassen's matrix multiplication algorithm.
Inputs: A is an $m \times k$ matrix and B is a $k \times n$ matrix, where m, k, and n are even. α and β are scalars.
Output: $C = \alpha AB + \beta C$ is an $m \times n$ matrix.
Temporary Variables: R_1 (size $mk/4$), R_2 (size $kn/4$), and R_3 (size $mn/4$).

Store to :		*Computation*	*Algorithmic Variables*
R_1	$\leftarrow$	$\alpha(A_{21} + A_{22})$;	αS_1
R_2	$\leftarrow$	$B_{12} - B_{11}$;	T_1
R_3	$\leftarrow$	$R_1 R_2$;	αP_3
C_{12}	$\leftarrow$	$\beta C_{12} + R_3$;	$\beta C_{12} + \alpha P_3$
C_{22}	$\leftarrow$	$\beta C_{22} + R_3$;	$\beta C_{22} + \alpha P_3$
R_1	$\leftarrow$	$R_1 - \alpha A_{11}$;	αS_2
R_2	$\leftarrow$	$B_{22} - R_2$;	T_2
R_3	$\leftarrow$	$\alpha A_{11} B_{11}$;	αP_1
C_{11}	$\leftarrow$	$\beta C_{11} + R_3$;	$\beta C_{11} + \alpha P_1$
R_3	$\leftarrow$	$R_3 + R_1 R_2$;	αU_2
C_{11}	$\leftarrow$	$C_{11} + \alpha A_{12} B_{21}$;	$\beta C_{11} + \alpha P_2$
R_1	$\leftarrow$	$\alpha A_{12} - R_1$;	αS_4
R_2	$\leftarrow$	$\alpha(B_{21} - R_2)$;	αT_4
C_{12}	$\leftarrow$	$C_{12} + R_1 B_{22}$;	$\beta C_{12} + \alpha P_6$
C_{12}	$\leftarrow$	$C_{12} + R_3$;	$\beta C_{12} + \alpha U_2$
C_{21}	$\leftarrow$	$\beta C_{21} + A_{22} R_2$;	$\beta C_{21} + \alpha P_7$
R_1	$\leftarrow$	$\alpha(A_{11} - A_{21})$;	αS_3
R_2	$\leftarrow$	$B_{22} - B_{12}$;	T_3
R_3	$\leftarrow$	$R_3 + R_1 R_2$;	αU_3
C_{21}	$\leftarrow$	$C_{21} + R_3$;	$\beta C_{21} + \alpha U_3$
C_{22}	$\leftarrow$	$C_{22} + R_3$;	$\beta C_{22} + \alpha U_3$

Figure 1: STRASSEN2

Implementation	$\beta = 0$	$\beta \neq 0$
CRAY SGEMMS	$7m^2/3$	$7m^2/3$
IBM ESSL DGEMMS	$1.40m^2$	not directly supported
DGEMMW ([8])	$2m^2/3$	$5m^2/3$
STRASSEN1	$2m^2/3$	$2m^2$
STRASSEN2	m^2	m^2
DGEFMM	$2m^2/3$	m^2

Table 1: *Memory Requirements for Strassen codes on order m matrices*

other two. In the case $\beta \neq 0$, our requirement of m^2 is lower than the others that can handle this case, representing a reduction of 40 and 57 percent, respectively, over that required by DGEMMW and CRAY SGEMMS.

3.3 Odd-sized Matrices

As we saw in Section 2, our implementation uses dynamic padding when odd dimensions are encountered. This approach has been dismissed by others [8] because of the inefficiency of some of the required fixup operations. Our implementation partially deals with this concern by combining the required operations in (9) and computing them with BLAS routines. After combining operations, there are potentially three fixup steps:

$$\begin{aligned} C_{11} &= \alpha\,(a_{12}b_{21}) + C_{11}, \\ c_{12} &= \alpha \begin{pmatrix} A_{11} & a_{12} \end{pmatrix} \begin{pmatrix} b_{12} \\ b_{22} \end{pmatrix} + \beta c_{12}, \\ \begin{pmatrix} c_{21} & c_{22} \end{pmatrix} &= \alpha \begin{pmatrix} a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} B_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} + \beta \begin{pmatrix} c_{21} & c_{22} \end{pmatrix}. \end{aligned}$$

The first step can be computed with the BLAS routine DGER (rank-one update), and the second and third steps can be computed by calling the BLAS routine DGEMV (matrix-vector product).

Further justification for our decision to use dynamic padding comes, as mentioned in Section 2, from our theoretical analysis in [14], as well as what we believe is a simplified code structure, where no special cases are embedded in the routine that applies Strassen's algorithm, and no additional memory is needed when odd dimensions are encountered.

3.4 Setting Cutoff Criteria

In Section 2 we determined the optimal cutoff criterion using operation count as a cost function. In practice operation count is not an accurate enough predictor of performance to be used to tune actual code. The interested reader is referred to [14] for a discussion of the limitations of operation count and development of other performance models that can more accurately predict performance parameters. In addition, performance of our code varies from machine to machine, so an effective cutoff criterion must be adaptable. In this section we describe a parameterized cutoff criterion whose parameters are set from empirical performance measurements.

A parameterized criterion for square matrices can be determined in a manner similar to the way the theoretical condition in Section 2 was obtained. The time for DGEMM is compared to the time required by applying Strassen's algorithm with one level of recursion. The cutoff is obtained from the crossover, the matrix order τ, where Strassen's algorithm becomes faster than DGEMM. This leads to the condition

$$m \leq \tau, \tag{10}$$

where τ is the empirically determined parameter.

Obtaining a general condition for rectangular matrices is much more difficult since the boundary between the regions where DGEMM is faster and Strassen's algorithm is faster is most likely no longer described by a simple equation and will certainly require many more computer experiments to determine. This forces us to compromise, both in the amount of experiments that are performed and the model used to describe the boundary, when choosing the cutoff criterion for rectangular matrices. Also it is important that the square condition be preserved since it is essential that the performance of DGEFMM on square matrices is not sacrificed.

There are two natural ways to create a rectangular criterion from the condition for square matrices:

$$m \leq \tau \quad \text{or} \quad k \leq \tau \quad \text{or} \quad n \leq \tau, \tag{11}$$

and

$$mkn \leq \tau(nk + mn + mk)/3. \tag{12}$$

Condition (11), used in [8], prevents Strassen's algorithm from being applied in certain situations where it would be beneficial. One situation where this can occur, as was illustrated in Section 2, is when one of the matrix dimensions is below the square cutoff and one of the other dimensions is large. Condition (12), proposed by Higham in [11], scales the theoretical condition (7) by $(4/3)\tau$, so that it reduces to the square condition (10) when $m = k = n$. This condition suffers from two drawbacks: (1) it assumes that the performance of DGEMM for general rectangular matrices is determined by the performance on square matrices, and

(2) it assumes that the performance of DGEMM is symmetric in the matrix dimensions. Our empirical investigations, which will be presented in Section 4.2, indicate that these assumptions are not generally valid.

We propose a model for the boundary that takes into account both square and rectangular matrices and allows for asymmetry in the performance of DGEMM. First, condition (7) is extended by including three parameters to take into account this asymmetry. Second, this extended condition is combined with the square condition (11) to preserve square-case behavior. The parameterized rectangular condition is

$$mkn \leq \rho_m nk + \rho_k mn + \rho_n mk, \tag{13}$$

which is equivalent to

$$1 \leq \rho_m/m + \rho_k/k + \rho_n/n. \tag{14}$$

The parameters, ρ_m, ρ_k, ρ_n, are computed empirically from three separate experiments, where two of the dimensions are fixed at a large value and the third varies. On the face of it, we must solve a system of three linear equations; however, (14) suggests an alternative approach. In each of the three experiments, as in the square case, we search for the point at which one application of Strassen's algorithm becomes faster than DGEMM. When k and n are large, their contribution in (14) is negligible, so that the parameter ρ_m can be set to the crossover point determined from the experiment where k and n are fixed. The other two parameters are set in the same way.

If alternative values of m, k, and n are used to compute ρ_m, ρ_k, and ρ_n, different values for the parameters may be obtained. This is possible since it is unlikely that the entire boundary can be described by an equation of the form (13). In particular, since long thin matrices are used to determine the parameters ρ_m, ρ_k, and ρ_n, it is likely that $\rho_m + \rho_k + \rho_n \neq \tau$, as would be required if the new condition were to reduce to (10) when $m = k = n$. Our compromise is to use the hybrid condition

$$\begin{gathered}\big(\,(mkn \leq \rho_m nk + \rho_k mn + \rho_n mk) \quad \text{and} \quad (m \leq \tau \quad \text{or} \quad k \leq \tau \quad \text{or} \quad n \leq \tau)\,\big) \\ \text{or} \\ (m \leq \tau \quad \text{and} \quad k \leq \tau \quad \text{and} \quad n \leq \tau),\end{gathered} \tag{15}$$

which inherently allows recursion, via condition (11), when all three dimensions are greater than τ, and stops recursion, via the last condition, when all three dimensions are less than or equal to τ. In other regions of the positive octant condition (13) rules in order to allow an extra application of Strassen's algorithm where it may be beneficial. Actual tests to establish τ, ρ_m, ρ_k, and ρ_n for various machines will be done in Section 4.2.

4 Performance

This section discusses the results of running our implementation of Strassen's matrix multiplication algorithm on a variety of computers. We first describe the machines used and how the results were obtained. Next, empirical measurements are used to set the cutoff conditions for both square and rectangular inputs, and our new criterion is compared to other methods. We then examine the performance of our new routine, including comparisons to other available routines for performing Strassen's matrix multiplication. Finally, we use our routine to perform the matrix multiplications in a large application code.

4.1 Machines and Testing

Though our code is portable and has been run on many machines, this paper will focus on the IBM RS/6000 (AIX 3.25), with some results for the Cray YMP C90 (UNICOS 9.0.1.2) and Cray T3D (single processor/UNICOS MAX 1.3.0.0). It is important to note that we are presenting only serial results. Furthermore, we limited ourselves to sizes of matrices where the entire problem fits into the machine's memory without using virtual memory.

Wherever possible and beneficial, we utilized optimized versions of core routines. The IBM-supplied BLAS library, *libblas.a* (Version 2.2), was used for all computational kernels, except for IBM's implementation of Strassen's algorithm, DGEMMS, which is contained in their Engineering Scientific Subroutine Library (ESSL - Version 2.2) . Unlike all other Strassen implementations we have seen, IBM's DGEMMS only performs the multiplication portion of DGEMM, $C = \mathrm{op}(A) * \mathrm{op}(B)$. The update of C and scaling by α and β must be done separately by the calling routine whenever $\alpha \neq 1.0$ or $\beta \neq 0.0$. On the C90 and T3D, Cray's BLAS, contained in their scientific routines library, *scilib.a* (Version 2.0), was utilized. All results presented in this paper are for 64-bit values. This is double precision on the RS/6000, but single precision for the routines on the Cray machines.

Timing was accomplished by starting a clock just before the call to the matrix multiplication routine of interest and stopping the clock right after the call, with the exception of the call to IBM's DGEMMS, which contained an extra loop for the scaling and update of C as described above. The times reported here are CPU times on non-dedicated machines.

All routines were tested with the same initial matrices, and the correctness of the results was verified. All of our routines, including our Strassen library and test codes used in this paper, are available on the Web at `http://www.mcs.anl.gov/Projects/PRISM/lib/`.

4.2 Measuring the Cutoff

In this section we report on the empirical testing needed to determine the actual cutoff criteria used on all three of our test machines. This is done first for square matrices and

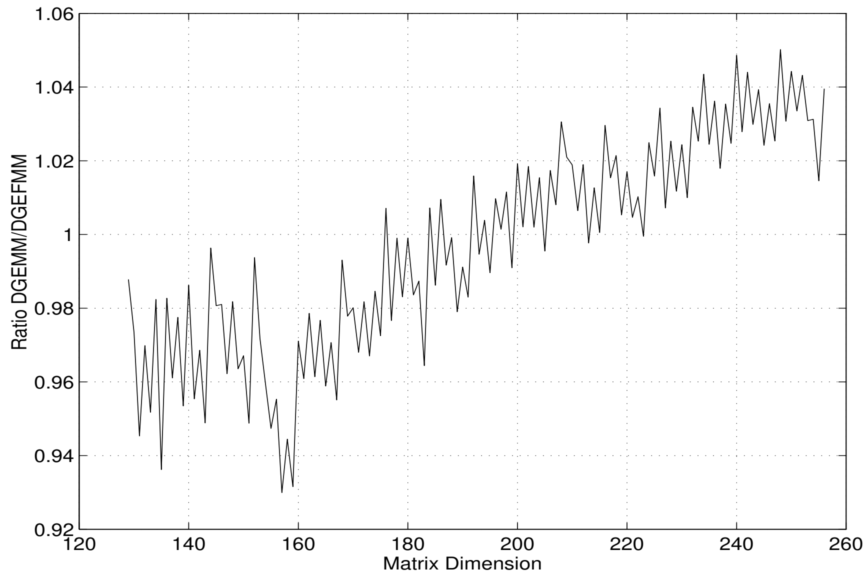

Figure 2: *Experimentally determined cutoff for RS/6000 with* $\alpha = 1$ *and* $\beta = 0$*.*

then for rectangular matrices.

Recall from Section 3.4 that determining the square cutoff is simply a matter of finding where calling DGEMM is faster than applying Strassen's algorithm with one level of recursion. Figure 2 shows the ratio of DGEMM to our DGEFMM code with one level of recursion as a function of the square matrix order m for the RS/6000. When this ratio is greater than 1, using Strassen is more efficient. Strassen becomes better at $m = 176$ and is always more efficient if $m \geq 214$. The range is caused by the fact that the costs in our Strassen code are not monotonic, and the varying requirements for fixup work on odd-sized inputs create the saw-toothed look of the graph. However, the range is fairly small, and the performance difference between Strassen and DGEMM is small. Thus, it is reasonable and acceptable to chose any of the points in the range. We opted to choose the cutoff to be $\tau = 199$, i.e., use DGEMM if $m \leq 199$, since Strassen is almost always better than DGEMM for larger values, and when it is slower it is so by a very small amount. Table 2 shows the results of performing this experiment for our test machines. The cutoffs in Table 2 show the size at which our Strassen implementation becomes more efficient for square matrices. The modest size of the cutoffs show that using our Strassen code instead of standard matrix multiplication is advantageous in many applications. These results, along with those of others, shows that Strassen's algorithm is of practical interest.

As was discussed in Sections 2 and 3.4, determining the condition for rectangular matrices is much more complicated. The benefit of enhancing the cutoff criteria for rectangular

Machine	Empirical Square Cutoff τ
RS/6000	199
C90	129
T3D	325

Table 2: *Experimentally determined square cutoffs on various machines.*

Machine	ρ_m	ρ_k	ρ_n	$\rho_m + \rho_k + \rho_n$
RS/6000	75	125	95	295
C90	80	45	20	145
T3D	125	75	109	309

Table 3: *Experimentally determined rectangular cutoff parameters, when $\alpha = 1$ and $\beta = 0$.*

matrices can be seen by noting that use of criterion (11) on the RS/6000 prevents Strassen's algorithm from being applied when $m = 160$, $n = 957$, and $k = 1957$. However, applying an extra level of Strassen's algorithm gives an 8.6 percent reduction in computing time.

Table 3 summarizes, for each machine, the values obtained for the parameters ρ_m, ρ_k, and ρ_n, which describe the rectangular condition (13). These were computed using the method described in Section 3.4. On the CRAY C90 and the RS/6000, the two fixed variables in each of the experiments were set to 2000, whereas on the CRAY T3D a value of 1500 was used to reduce the time to run the tests. Finally, we remark that the experiments were run using $\alpha = 1$ and $\beta = 0$ in the calls to DGEFMM, and that the values of ρ_m, ρ_k, and ρ_n may change for the general case. Our code allows user testing and specification of two sets of parameters to handle both cases.

The data in Table 3 shows that the performance of DGEMM is not symmetric in the matrix dimensions. Moreover, the asymmetry varies from machine to machine. Also observe that on the RS/6000 the sum $\rho_m + \rho_k + \rho_n$ is significantly different from the corresponding square cutoff τ. This illustrates that the performance of DGEMM on long thin matrices can be very different from its performance on square matrices.

Table 4 summarizes a series of experiments comparing the various rectangular cutoff criteria described in Section 3.4. For each comparison a set of random problems was generated on which the two criteria being compared would make different cutoff decisions. To do this we randomly selected the input dimensions m, k, and n, and then tested for those on which the two criteria would make opposite determinations on whether to apply recursion at the

Machine	Comparison Ratio	Random Sample Size	Range	Quartiles	Average
RS/6000	(15)/(11)	100	0.9128–1.0169	0.9403;0.9566;0.9634	0.9529
	(15)/(12)	1000	0.9083–1.1286	0.9863;1.0038;1.0217	1.0017
	(15)/(12), two dims large	100	0.9126–1.0462	0.9688;0.9938;1.0052	0.9888
C90	(15)/(11)	100	0.8275–1.0294	0.9137;0.9488;0.9682	0.9375
	(15)/(12)	1000	0.7803–1.1095	0.9198;0.9457;0.9769	0.9428
	(15)/(12), two dims large	100	0.7908–0.9859	0.9024;0.9189;0.9454	0.9098
T3D	(15)/(11)	100	0.7957–1.1026	0.9335;0.9508;0.9667	0.9518
	(15)/(12)	1000	0.6868–1.1393	0.9528;0.9792;0.9996	0.9777
	(15)/(12), two dims large	100	0.9120–1.0328	0.9200;0.9262;0.9474	0.9340

Table 4: *Comparison of Cutoff Criteria for $\alpha = 1$ and $\beta = 0$. (11) = Simple Criterion from Square Cutoff, (12) = Theoretically Scaled Criterion, (15) = New Rectangular Criterion.*

top level. The range of dimensions for the generated problems ran from the smaller of $\tau/3$ and ρ_m, ρ_k, or ρ_n for m, k, and n, respectively, to 2050 on the RS/6000 and CRAY C90 and 1550 on the T3D. A sample size of 100 was used when the design of our new criterion clearly implied an expected outcome for the experiment; a larger sample size of 1000 was used for the case where the outcome was less clear. We note that it is sufficient to compare each pair of criteria on only their set of such problems; for on other problems, where they would make identical decisions on when to apply recursion, they would have inherently identical performance. Thus, any criterion that shows better performance on these test problems in fact enjoys better overall performance.

Entries in Table 4 are given as ratios of DGEFMM timings using our new criterion (15) to those obtained using other criteria. We show the range, average, and quartiles, values that mark the quarter, half (or median), and three-quarter points in the data. For each machine, we compare (15) to (11), (15) to (12) on general input sizes, and (15) to (12) on problems where two of the three matrix dimensions are large. We define large to be at least

Matrix Order: Number of Recursions	RS/6000 DGEMM	RS/6000 DGEFMM	C90 DGEMM	C90 DGEFMM	T3D DGEMM	T3D DGEFMM
$\tau + 1 :\ 1$	.150	.150	.0060	.0055	.694	.669
$2\tau + 2 :\ 2$	1.14	1.05	.0431	.0410	5.40	4.91
$4\tau + 4 :\ 3$	9.06	7.59	.332	.312	42.6	33.3
$8\tau + 8 :\ 4$	72.2	54.1	2.54	2.10		
$16\tau + 16 :\ 5$			20.1	13.3		

Table 5: *Times for DGEMM and DGEFMM for different numbers of recursions on all three machines with $\alpha = 1/3$ and $\beta = 1/4$. τ is the square cutoff value given in Table 2 for each machine.*

1800 for the RS/6000 and CRAY C90 and 1350 for the CRAY T3D.

The results in Table 4 demonstrate that in some cases significant performance improvements can be obtained through careful refinement of the cutoff criterion used. We see that our new criterion nearly meets or in general exceeds the performance of other cutoff criteria that have been used or proposed. For matrices with large aspect ratios, performance is always improved. We also note that, as indicated in Section 3.4, the values for ρ_m, ρ_k, and ρ_n could be even more finely tuned, for instance to use in application areas where performance on matrices with large aspect ratios is critical.

4.3 Performance Comparisons

Table 5 shows the times for DGEMM and DGEFMM on all three machines for the smallest matrix order that does a given number of recursions in DGEFMM. The scaling of DGEFMM with matrix order is very close to the theoretical factor of 7 for each doubling in matrix size. All are within 10% of this scaling, with the C90 showing the largest variance. The table also shows that Strassen can be substantially faster than conventional matrix multiplication. For the largest size matrix given for each machine in Table 5, the time for DGEFMM is between 0.66 and 0.78 the time for DGEMM. Note that the matrix sizes needed for savings are well within the range of matrices of interest in real applications.

Figure 3 shows the ratio of DGEFMM to the IBM Strassen routine DGEMMS for square matrices. The plot shows that on average the IBM routine is faster than DGEFMM. The average ratio is 1.052. These results are for the case where $\alpha = 1$ and $\beta = 0$. In the general

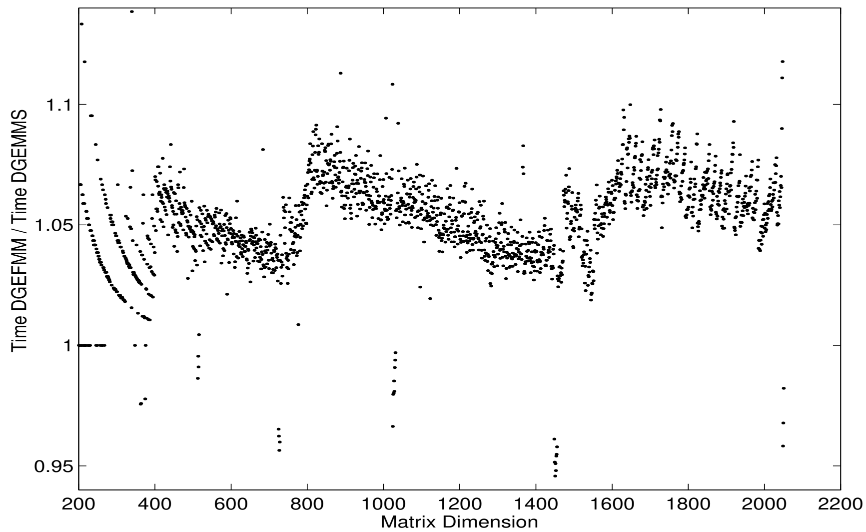

Figure 3: *Ratio of DGEFMM to IBM DGEMMS as a function of matrix order on RS/6000 for $\alpha = 1$ and $\beta = 0$.*

case (where $\alpha \neq 1.0$ and $\beta \neq 0.0$), which the IBM routine does not directly support, the average drops to 1.028. This supports our design of including the general case directly in our code.

Figure 4 shows the ratio of DGEFMM to the Cray Strassen routine, SGEMMS, on the C90. As with the IBM results, our performance is slightly worse than the vendor-supplied routine. The average ratio is 1.066. As with the RS/6000, we do better for general α and β, where the average drops to 1.052.

Considering that we did not tune our code to either the RS/6000 or C90, we think our relative performance is very good. We have observed that one can optimize the primitives and methods to typically get a several percent gain. To keep our code general, we have not included these machine-specific techniques in our code.

Next, we compare to a public domain implementation from Douglas, et al. [8], DGEMMW. We see in Figure 5 that, for general α and β on square matrices, there are matrix sizes where each code does better. The average ratio is 0.991, which shows that we are slightly better. In the case where $\alpha = 1$ and $\beta = 0$ the average is 1.0089. This shows that our STRASSEN2 construction (general α, β) not only saves temporary memory but yields a code that has higher performance both absolutely and relative to STRASSEN1. This is due to better locality of memory usage. It should be noted that DGEMMW also provides routines for multiplying complex matrices, a feature not contained in our package.

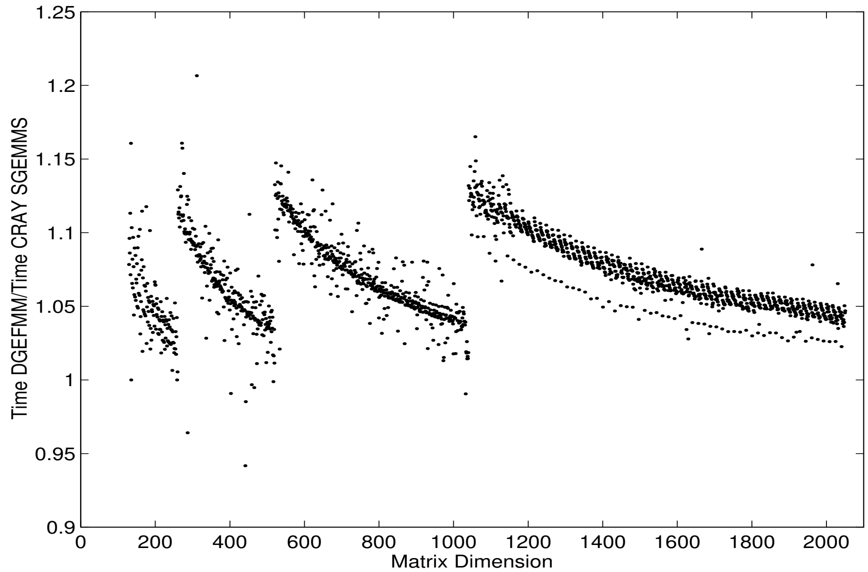

Figure 4: *Ratio of DGEFMM to Cray SGEMMS as a function of matrix order on C90 for* $\alpha = 1$ *and* $\beta = 0$.

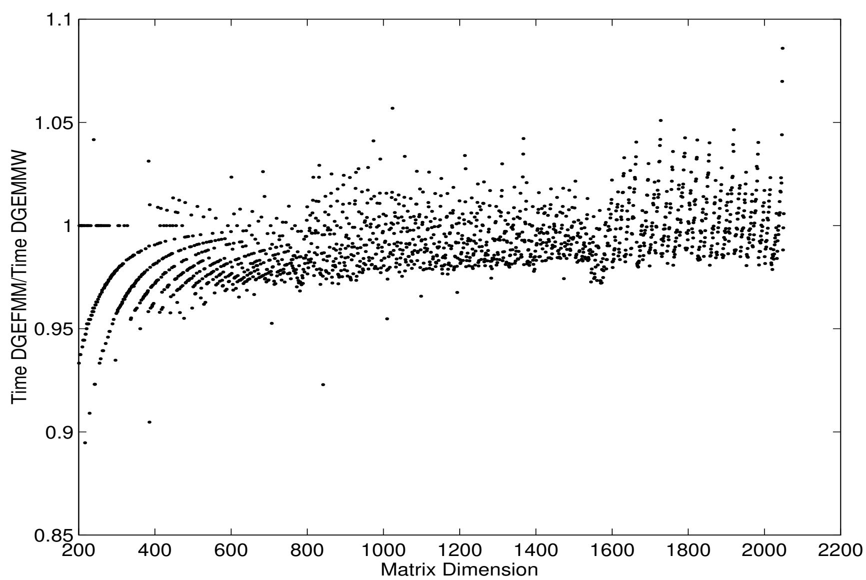

Figure 5: *Ratio of DGEFMM to DGEMMW as a function of matrix order on IBM RS/6000 for general* α *and* β.

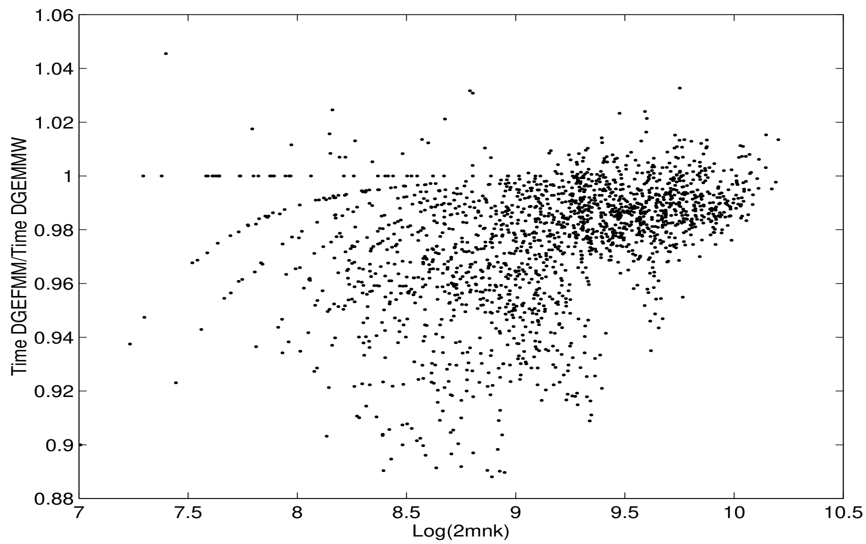

Figure 6: *Ratio of DGEFMM to DGEMMW for random rectangular matrices on RS/6000 with general α and β.*

We now turn our attention to rectangular matrices. We tested randomly-generated rectangular matrices and computed the ratio of our routine, DGEFMM, to the routine DGEMMW on the RS/6000 as shown in Figure 6, where $\text{Log}(x)$ denotes the logarithm base 10 of x. By randomly-generated, we mean randomly selecting the input dimensions m, k, and n in the range from $\rho_m = 75$, $\rho_k = 125$, or $\rho_n = 95$, respectively, to 2050. The average ratio has decreased to 0.974 compared to 0.991 for square matrices. This could be due to the fact that DGEMMW uses the simplified cutoff test given by (11). The average ratio for $\alpha = 1$ and $\beta = 0$ is 0.999, which is also an improvement over the square case. The average improvement over random rectangular matrices is smaller than that seen in Section 4.2, since we do not gain an extra level of recursion in many cases.

Overall, these results show that our DGEFMM code performs very well compared to other implementations. This is especially significant considering we use less memory in many cases. This also shows the dynamic peeling technique using rank-one updates is indeed a viable alternative. The reader is referred to [14], where our enhanced models are given that quantitatively describe the behavior seen here. They can also be used to further examine these and other implementations of Strassen's algorithm.

	Using DGEMM	Using DGEFMM
Total Time (secs)	1168	974
MM Time (secs)	1030	812

Table 6: *Eigensolver timings for* 1000×1000 *matrix on IBM RS/6000.*

4.4 Sample Application: Eigensolver

In order to measure the efficiency of our implementation of Strassen's algorithm, we have incorporated it into a divide-and-conquer-based symmetric eigensolver, whose kernel operation is matrix multiplication. This eigensolver is based on the Invariant Subspace Decomposition Algorithm (ISDA) [15] and is part of the PRISM project. The ISDA uses matrix multiplication to apply a polynomial function to a matrix until a certain convergence criterion is met. At that point, the range and null space of the converged matrix is computed, which provides the subspaces necessary for dividing the original matrix into two subproblems. Both of these operations depend heavily on matrix multiplication. The algorithm is repeated until all subproblems have been solved.

Incorporating Strassen's algorithm into this eigensolver was accomplished easily by renaming all calls to DGEMM as calls to DGEFMM. Table 6 provides the resulting performance gain for finding all the eigenvalues and eigenvectors of a randomly-generated 1000×1000 test matrix on a RS/6000 using both DGEMM and DGEFMM. Note that in this application the running time for random matrices is typically the same as that seen for other matrices of the same size. Thus, it is sufficient to test on a randomly-generated input matrix. We see that there is an approximate 20% savings in the matrix multiplication time by using DGEFMM. This shows that real applications can easily realize the performance gain from our new Strassen implementation.

5 Summary and Conclusions

In this paper we have described our implementation of Strassen's algorithm for matrix multiplication and reported on running our code on the IBM RS/6000, CRAY YMP C90, and CRAY T3D single processor. Our empirical data shows that Strassen's algorithm provides improved performance over the standard algorithm for matrix sizes that occur in practice. Moreover, our implementation, DGEFMM, is designed to replace DGEMM, the Level 3 BLAS matrix multiplication routine, thus providing enhanced performance in existing application codes. This was exhibited by the use of DGEFMM for the matrix multiplications required in an eigensolver code.

Our implementation is written in C and uses the BLAS for important kernel routines. A cutoff criterion is used to determine whether to apply Strassen's algorithm or to use DGEMM. The cutoff criterion uses parameters which can be set based on empirical performance measurements, allowing our code to be tuned to different machines.

Previous implementations of Strassen's algorithm [2, 3, 8] offer similar functionality and performance characteristics, though our study and analysis is more thorough. Aspects of our work that are unique to our implementation are the following. To deal with odd matrix dimensions, we use dynamic peeling with rank-one updates, a technique whose usefulness had previously been in question and had not so far been demonstrated. Also, we have carefully analyzed cutoff criteria for rectangular matrices and have developed a new technique that leads to a performance improvement. In addition, for certain cases, we have reduced the amount of memory needed for temporary computations by 40 to more than 70 percent compared to other implementations. Performance of our implementation is competitive with all other similar codes known to us. This is especially significant in view of our reduced memory requirements.

In addition to the work presented in this paper we have developed several models whose goal is to predict and compare the performance of various alternatives that arise in the implementation of Strassen's algorithm. A description of the models and their use in predicting performance and comparing design alternatives can be found in [14]. Future work could make use of these models to further refine our criteria for stopping recursions, investigate alternate peeling techniques, extend our implementation to use virtual memory and parallelism, and/or extend our models to account for these new architectural features.

6 Acknowledgements

The authors thank the reviewer who provided extensive comments and valuable suggestions.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. **LAPACK Users' Guide**. SIAM, Philadelphia, 1992.

[2] D. H. Bailey. Extra high speed matrix multiplication on the Cray-2. **SIAM J. Sci. Stat. Computing**, 9:603–607, 1988.

[3] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's Algorithm to Accelerate the Solution of Linear Systems. **Journal of Supercomputing**, 4(5):357–371, 1990.

[4] R. P. Brent. Algorithms for matrix multiplication. Technical report, Stanford University, 1970. Technical Report CS 157, (Available from National Technical Information Service, # AD705509).

[5] J. Cohen and M. Roth. On the implementation of Strassen's fast multiplication algorithm. **Acta Informatica**, 6:341–355, 1976.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. **Introduction to Algorithms**. MIT Press, Cambridge, Massachusetts, 1990.

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. **ACM Trans. Math. Software**, 16:1–17, 1990.

[8] C. Douglas, M. Heroux, G. Slishman, and R. M. Smith. GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm. **Journal of Computational Physics**, 110:1–10, 1994.

[9] P. C. Fischer. **Further schemes for combining matrix algorithms, in Automata, Languages and Programming**, volume 14 of **Lecture Notes in Computer Science**, pages 428–436. Springer-Verlag, Berlin, 1974.

[10] P. C. Fischer and R. L. Probert. **Efficient procedures for using matrix algorithms, in Automata, Languages and Programming**, volume 14 of **Lecture Notes in Computer Science**, pages 413–427. Springer-Verlag, Berlin, 1974.

[11] N. J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. **ACM Trans. Math. Software**, 16:352–368, 1990.

[12] N. J. Higham. Stability of block algorithms with fast level 3 BLAS. **ACM Trans. Math. Software**, 18:274–291, 1992.

[13] J. E. Hopcroft and L. R. Kerr. Some techniques for proving certain simple programs optimal. In **Proceedings, Tenth Annual Symposium on Switching and Automata Theory**, pages 36–45, 1969.

[14] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Strassen's algorithm for matrix multiplication: Modeling, analysis, and implementation. Technical report, Center for Computing Sciences, 1996. Technical Report CCS-TR-96-147.

[15] S. Huss-Lederman, A. Tsao, and T. Turnbull. A parallelizable eigensolver for real diagonalizable matrices with real eigenvalues. **SIAM J. Sci. Computing**, 18(2), 1997. (to appear).

[16] IBM Engineering and Scientific Subroutine Library Guide and Reference, 1992. Order No. SC23-0526.

[17] P. A. Knight. Fast rectangular matrix multiplication and QR decomposition. **Linear Algebra Appl.**, 221:69–81, 1995.

[18] R. L. Probert. On the additive complexity of matrix multiplication. **SIAM Journal of Computing**, 5(6):187–203, 1976.

[19] V. Strassen. Gaussian elimination is not optimal. **Numer. Math.**, 13:354–356, 1969.