Strassen's Algorithm for Matrix Multiplication: Modeling, Analysis, and Implementation

Steven Huss-Lederman
Elaine M. Jacobson
J. R. Johnson
Anna Tsao
Thomas Turnbull

November 15, 1996

Abstract

In this paper we report on the development of an efficient and portable implementation of Strassen's matrix multiplication algorithm for matrices of arbitrary size. Our implementation is designed to be used in place of DGEMM, the Level 3 BLAS matrix multiplication routine. Our code is designed so that efficient performance will be obtained for all matrix sizes and shapes and that the additional memory needed for temporary variables is minimized. Replacing DGEMM with our routine should provide 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.

Our implementation is based on a detailed theoretical study of the design choices that are available for various aspects of Strassen's algorithm. These choices include the method of handling matrices with odd dimensions, the criterion for switching to the standard algorithm for matrix multiplication, and how to organize the intermediate computations to minimize the amount of memory required for temporary variables. Our study of techniques for dealing with odd dimensions shows that a method previously dismissed is a viable alternative. Furthermore, our theoretical study leads to a new method for determining when to switch to the standard matrix multiplication algorithm for rectangular matrices. The new criterion performs better in practice than existing criteria. Finally, our implementation uses less memory for temporary variables than existing implementations. In certain cases our memory requirements are reduced by 40 to more than 70 percent over these other codes. A theoretical discussion of temporary variables shows that our memory usage is the smallest possible.

In addition to discussing the important design choices that were made, several models are developed to help predict the effects of various decisions. The first model uses operation count; however, operation count is not an accurate predictor of actual performance. Therefore a model that takes into account the performance of fundamental matrix operations on different computer architectures is introduced. The costs for the basic operations are obtained from a best-fit to actual performance data. Using the data for a specific machine, we are able to accurately predict performance. Furthermore, using this model, design decisions can be tested and performance parameters can be adjusted to a particular machine. For example, we are able to determine, from the model, the optimal criterion for stopping the recursive calls to Strassen's algorithm and switching to the standard algorithm. The model's accuracy is illustrated using performance data on the IBM RS/6000, Cray YMP C90, and Cray T3D single processor.

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

Contents

1 Introduction **1**

2 Strassen's Algorithm **3**

3 Choosing an Appropriate Cutoff **8**

4 Odd-Sized Matrices **13**
- 4.1 Comparison of Padding and Peeling . . . 15
- 4.2 Cutoff criteria for square matrices with odd dimension . . . 18
- 4.3 The number of recursive levels in Strassen's algorithm . . . 20
- 4.4 Dynamic padding verses dynamic peeling . . . 20

5 Implementation Issues **25**
- 5.1 Interface . . . 25
- 5.2 Temporary Allocation and Memory Usage . . . 25
- 5.3 Odd-sized Matrices . . . 32
- 5.4 Setting Cutoff Criteria . . . 33
- 5.5 Program Structure . . . 36

6 Further Model Refinements **38**

7 Performance **42**
- 7.1 Machines and Testing . . . 42
- 7.2 Determining Model Coefficients . . . 43
- 7.3 Behavior of Primitive Operations . . . 47
- 7.4 Measuring and Modeling the Cutoff . . . 49
  - 7.4.1 Square Matrices . . . 49
  - 7.4.2 Rectangular Matrices . . . 52
- 7.5 Modeling Runtimes . . . 55
- 7.6 Using the Model for Further Analysis . . . 55
- 7.7 Performance Comparisons . . . 58
- 7.8 Sample Application: Eigensolver . . . 64

8 Summary and Conclusions **65**

9 Acknowledgements **65**

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 [10], where it can then be used as a building block in the implementation of many other routines, as done in LAPACK ([1], [2]). 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 [28], which has complexity $\Theta(m^{\lg(7)})$, where $\lg(x)$ denotes the base 2 logarithm of x and $\lg(7) \approx 2.807$. Other methods have been developed which reduce the complexity even closer to the lower bound of $\Omega(m^2)$ ([7], [22]), but so far none of these have lent themselves to practical implementations for typical-size matrices, say of order 10,000 or less.

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 myths 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 [3], Douglas, et al.'s [11] C implementation of the Winograd variant of Strassen's algorithm on various machines, and IBM's ESSL library routine [18]. In addition, the stability analyses of Brent [5] and then Higham [14, 15] 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. In addition, 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. Previous work addressing these issues can be found in [3, 5, 6, 4, 11, 12, 13, 14, 19, 28]. We note that two periods of work on Strassen's algorithm are seen here; an early period from 1969 to 1976, and a recent one from 1988 to the present.

In this paper we expand on this previous work by developing detailed models of computation cost for multiple variants of Strassen's algorithm on matrices of arbitrary size. Since the Strassen recursion relies on dividing each matrix dimension in half, these techniques differ mainly in their treatment of odd dimensions. Basing the modeling solely on full operation counts suggests that one technique for dealing with odd-sized matrices, known as dynamic peeling, could in fact be a viable alternative even though it had in the past been dismissed as undesirable [11]. Using this approach we develop a general, efficient, and portable implementation of Strassen's algorithm which is usable in any program in place of calls to DGEMM, the Level 3 BLAS matrix multiplication routine. In our design, careful consideration has been given to all of the issues mentioned above. In addition to demonstrating the viability of dynamic peeling, our analysis also provides an improved cutoff condition for rectangular matrices 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 RS/6000 ESSL 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 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. In addition, refining our model by recosting the basic operations using a best-fit to actual performance on a given architecture leads to a good estimator of execution times and predictor of optimal stopping points for the recursion. We illustrate the model's accuracy by comparing its predictions to performance results.

The remainder of this paper is organized as follows. Section 2 reviews Strassen's algorithm and introduces our cost models. In Section 3 we use a full operation count model to analyze cutoffs for both square and rectangular input matrices. Section 4 also uses this model to examine various options for dealing with odd matrix dimensions. In Section 5 we address implementation issues related to cutoff, odd dimensions, and reduction of the amount of additional memory space required for temporary computations. A detailed description of our design choices and code structure is given. Section 6 updates our cost models to more closely match this actual code. Performance of our implementation is examined in Section 7, including comparisons to other codes and across machines. We complete our cost models with fits to performance data and compare predicted parameters to actual measurements. In this section we also report on using our Strassen code for the matrix multiplications in an eigensolver application. A summary and conclusions is offered in Section 8.

2. Strassen's Algorithm

In Sections 2, 3, and 4, we review many important aspects of Strassen's matrix multiplication algorithm for matrices of arbitrary size. The algorithm can be applied in a straightforward fashion to square matrices with order a power of 2, but issues arise for all other matrices. Earlier work addressing these issues can be found in [3, 5, 6, 4, 11, 12, 13, 14, 19, 28], but to our knowledge our development in this paper is the first to address the issues fully through both theoretical modeling and practical implementation. We begin in this section by reviewing the algorithm and discussing a framework for modeling its performance.

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 Volker Strassen's now famous 1969 paper [28], 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. Thus, if A and B are both $m \times m$ matrices with m even, we partition A and B into 2×2 block matrices, each block of size $m/2 \times m/2$. Then the product $C = AB$ is naturally partitioned in the same way

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

and can be computed by forming the seven products

$$\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\
P_2 &= (A_{21} + A_{22})B_{11}, \\
P_3 &= A_{11}(B_{12} - B_{22}), \\
P_4 &= A_{22}(B_{21} - B_{11}), \\
P_5 &= (A_{11} + A_{12})B_{22}, \\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

and observing that $C_{11} = P_1 + P_4 - P_5 + P_7$, $C_{12} = P_3 + P_5$, $C_{21} = P_2 + P_4$, and $C_{22} = P_1 + P_3 - P_2 + P_6$.

If we would now compute the products $P_i, i = 1, 2, \ldots 7$ using the standard algorithm, we can calculate the total number of arithmetic operations required by this single application of Strassen's technique as $7(2(m/2)^3 - (m/2)^2) + 18(m/2)^2 = (7/4)m^3 + (11/4)m^2$. The ratio of this operation count to that required by the standard algorithm alone is seen to be

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

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.

This alone would not be extraordinary, but Strassen's algorithm can continue, applying this construction recursively to compute each $P_i, i = 1, 2, \ldots 7$, and so on. The recursion can be carried as far as possible, being terminated at the bottom level when the multiplications are scalar. This of course assumes that at every level of recursion matrix dimensions are even, so that the simplest version of Strassen's algorithm actually requires matrix order $m = 2^d$. We will soon see that the technique is applicable in much more general cases. Letting $\mathcal{T}(m)$ be the arithmetic operation count required to compute C by Strassen's method with full recursion as described above, we have

$$\mathcal{T}(m) = 7\mathcal{T}(m/2) + 18((m/2)^2).$$

This is easily solved [8], giving $\mathcal{T}(m) = \Theta(m^{\lg(7)}) = \Theta(m^{2.807})$. Strassen in fact observed that $\mathcal{T}(m)$ was less than $4.7m^{\lg(7)}$, showing that the constant is not prohibitive. Thus, compared to the standard method, Strassen's technique provides an asymptotically faster algorithm for multiplying matrices.

One can ask if it is possible to improve upon Strassen's construction to obtain further savings. Hopcroft and Kerr [16] showed that at least seven multiplications are required by any algorithm for multiplying 2×2 matrices that does not depend on commutativity. Thus a recursive algorithm based on 2×2 matrix multiplication cannot be asymptotically faster than Strassen's algorithm. We note that it remains an open problem whether a faster algorithm can be obtained using 3×3 matrices.

Even though Strassen's algorithm leads to an asymptotic improvement there is substantial overhead due to the extra additions and subtractions. While reducing the number of additions does not lead to an asymptotic improvement it may provide a practical improvement. Probert [25] proved that the minimal number of additions and subtractions used by any bilinear algorithm for multiplying 2×2 matrices with seven multiplications is 15. A bilinear algorithm for computing the product $C = AB$, consists of four stages: (1) form linear combinations of the elements of A, (2) form linear combinations of the elements of B, (3) form products where the first operand is from (1) and the second operand is from (2), (4) form linear combinations of elements obtained from (3).

Winograd's variant of Strassen's algorithm (credited to M. Paterson) uses the required 7 multiplications and this minimal number of 15 additions/subtractions [13]. Stages (1) and (2) of the algorithm compute

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

Stage (3) computes the seven products

$$\begin{aligned}
P_1 &= A_{11}B_{11}, \\
P_2 &= A_{12}B_{21},
\end{aligned}$$

$$
\begin{aligned}
P_3 &= S_1 T_1, \\
P_4 &= S_2 T_2, \\
P_5 &= S_3 T_3, \\
P_6 &= S_4 B_{22}, \\
P_7 &= A_{22} T_4,
\end{aligned}
$$

and stage (4) computes

$$
\begin{aligned}
U_1 &= P_1 + P_2, \\
U_2 &= P_1 + P_4, \\
U_3 &= U_2 + P_5, \\
U_4 &= U_3 + P_7, \\
U_5 &= U_3 + P_3, \\
U_6 &= U_2 + P_3, \\
U_7 &= U_6 + P_6.
\end{aligned}
$$

It is easy to verify that $C_{11} = U_1$, $C_{12} = U_7$, $C_{21} = U_4$, and $C_{22} = U_5$. 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; at any point all of the required matrix products can be computed with any other technique available, usually the standard algorithm. As noted in the introduction, one key element for obtaining an efficient implementation of Strassen's algorithm is to 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*; see Section 3 for a detailed discussion of cutoff and its determination. 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. These techniques, often referred to as 'padding', generally involve either adding or deleting rows and/or columns to/from the original matrix. In Section 4 we will discuss various padding techniques. For now we simplify the present discussion by assuming matrix dimensions are always even.

In modeling the actual performance of Strassen's algorithm on real machines we cannot rely on operation count alone. While revealing asymptotic behavior, it can be a poor predictor of actual cost on specific problem sizes and machines for it does not take into account memory access patterns, possible data reuse, and differences in speed between different arithmetic operations. Thus, we refine our cost model by letting $M(m,k,n)$ be the cost of multiplying an $m \times k$ matrix by a $k \times n$ matrix using the standard matrix multiplication algorithm and $G(m,n)$ be the cost of adding or subtracting two $m \times n$ matrices. 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

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

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 the cost of Strassen's algorithm in terms of M and G is

$$\begin{aligned} W(2^d m', 2^d k', 2^d n') \quad = \quad & 7^d M(m', k', n') \; + \\ & 4 \sum_{i=0}^{d-1} 7^i G(2^{d-i-1} m', 2^{d-i-1} k') \; + \\ & 4 \sum_{i=0}^{d-1} 7^i G(2^{d-i-1} k', 2^{d-i-1} n') \; + \\ & 7 \sum_{i=0}^{d-1} 7^i G(2^{d-i-1} m', 2^{d-i-1} n'). \end{aligned} \tag{1}$$

In the remainder of this section we discuss three different models for estimating the computing time of Strassen's algorithm. These are based on three different assumptions for the costs $M(m, k, n)$ and $G(m, n)$. The first and simplest model stays with operation count, thus setting $M(m, k, n) = 2mkn - mn$ and $G(m, n) = mn$.

Our second model represents a simple attempt to differentiate between the compute rates for matrix addition/subtraction and multiplication on actual machines. Clearly $M(m, k, n)$ and $G(m, n)$ should be, respectively, cubic and quadratic functions of their parameters. Thus, we use μmkn for M, and γmn for G. The coefficients μ and γ can be chosen to correspond to MFLOP rates of particular algorithms for matrix multiplication and addition on a given computer.

Even the second model is lacking detail since it does not take into account different shapes of matrices and effects which may be dominant for small matrices. Thus, our third model includes low order terms, using a full cubic polynomial in m, k, and n for M, and a quadratic polynomial in m and n for G. The second model can be viewed as an asymptotic estimate of the third model. In the third model,

$$M(m, k, n) = \mu_7 mkn + \mu_6 mk + \mu_5 mn + \mu_4 kn + \mu_3 m + \mu_2 k + \mu_1 n + \mu_0 \tag{2}$$

and

$$G(m, n) = \gamma_3 mn + \gamma_2 m + \gamma_1 n + \gamma_0. \tag{3}$$

The coefficients μ_i, $i = 0, 1, \ldots, 7$ and γ_j, $j = 0, 1, 2, 3$, can be obtained empirically for a given implementation of the operations on a given computer by using a least squares fit of the polynomials $M(m, k, n)$ and $G(m, n)$ to measured computing times. This model should be reasonably accurate provided the memory hierarchy is used effectively. If the memory hierarchy is not used effectively there will be discontinuities when matrix sizes prevent data from fitting in a level of the memory hierarchy such as cache or main memory causing the algorithm to exhibit increased memory traffic across the memory hierarchy. If this is the case no single polynomial will model the computing time over a range of matrix sizes that jumps across such boundaries.

Substituting any of the three models' expressions for M and G into (1) we can obtain closed form expressions for $W(2^d m', 2^d k', 2^d n')$. Using the operation count model we obtain

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

The high-order-terms model gives

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

and the full-polynomial model leads to

$$
\begin{aligned}
W(2^d m', 2^d k', 2^d n') \;&=\; 7^d(\mu_7 m'k'n' + \mu_6 m'k' + \mu_5 m'n' + \mu_4 k'n' \\
&\qquad +\mu_3 m' + \mu_2 k' + \mu_1 n' + \mu_0) \;+ \\
&\quad \gamma_3(7^d - 4^d)(4m'k' + 4k'n' + 7m'n')/3 \;+ \\
&\quad (7^d - 2^d)(\gamma_2(11m' + 4k') + \gamma_1(4k' + 11n'))/5 \;+ \\
&\quad 15\gamma_0(7^d - 1)/6.
\end{aligned}
\tag{5}
$$

For the square matrix case ($m' = k' = n'$) we will shorten our notation to $W(m)$, $M(m)$, and $G(m)$. We note that for this case (4) simplifies to

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

and, if Strassen's original version of the algorithm had been used in the above development, (6) would become

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

Many early papers examined Strassen's algorithm using an operation count model. While most consider only the square case, Speiß [27] addresses the rectangular case and comes closest to our development here. While not explicitly computed in his paper, his equations (4.3) and (4.6) could lead to our equation (1). He explores a slightly different operation count model than ours, but again could have obtained (4) from his basic formulas. In more recent work, Knight [19] obtains the equivalent of (4) for Strassen's original algorithm; this also could be obtained from Speiß's results. Here, in (4), and the corresponding result for Strassen's original algorithm, we have obtained closed form expressions for operation count costs of both the original and the Winograd variant of Strassen's algorithm. In our development we have made no assumptions about the cutoff dimensions, have clearly separated the general modeling ideas from any specific model, and have carried to completion ideas only partially explored by others. We will continue in the following sections to use the above modeling schemes in our further analysis of Strassen's algorithm, and in Section 7, they will be used to estimate the performance of our implementation.

2^d	Standard MM	Strassen (Winograd) (Full Recursion)	Strassen (Original) (Full Recursion)
2	12	22	25
8	960	1,738	2,017
16	7,936	13,126	15,271
32	64,512	95,722	111,505
128	4,177,920	4,859,338	5,666,497
256	33,488,896	34,261,126	39,960,391
512	268,173,312	240,810,922	280,902,385
1024	2,146,435,072	1,689,608,614	1,971,035,287
2048	17,175,674,880	11,842,988,938	13,816,121,377

Table 1: *Operation Count Costs for* $2^d \times 2^d$ *Matrices, Full Recursion.*

3. Choosing an Appropriate Cutoff

In this section we discuss stopping the Strassen recursions early and performing the remaining matrix multiplications using the standard algorithm. Using the operation count model, we present theoretical techniques for determining the optimal stopping point. The ideas naturally lead to a corresponding empirical technique.

One level of Strassen's algorithm requires 7 multiplications and 15 additions/subtractions. Thus, applying the algorithm to 2×2 matrices requires 22 arithmetic operations (25 for the original version), while regular matrix multiplication only requires 8 multiplications and 4 additions or 12 operations. Thus we see that for small matrices, the extra additions/subtractions can outweigh the benefit of the reduced number of multiplications. Furthermore, this phenomenon affects the algorithm even when applied to larger matrices, for if the recursions are carried to the bottom (scalar) level very large matrices are needed before Strassen's algorithm becomes faster than regular matrix multiplication. For example, if we use (6) to compute the arithmetic operation count cost of Strassen's algorithm with full recursion for various square matrix sizes by setting $m' = 1$ and varying d, we obtain the results shown in Table 1. We also show the operation count costs for standard matrix multiplication and Strassen's original algorithm (7) for this case. Observe that the smallest square power-of-2 matrix order for which Strassen's algorithm has a lower operation count than regular matrix multiplication is 512, or 1024 for the original version. It is likely that this is one of the reasons that for many years Strassen's algorithm was thought to be of merely theoretical interest [24, 26].

The key to surmounting this difficulty is to not continue the Strassen recursions past the point where it is more efficient to use regular matrix multiplication. To determine the optimal point, or cutoff, at which to switch from Strassen's algorithm to regular matrix multiplication we need to consider when one application of Strassen's recursion leads to an improvement over completing the remaining required matrix multiplications using the standard algorithm. Clearly, if at any point in Strassen's algorithm, one more application of Strassen's technique leads to an improvement, it should be applied. Therefore, the optimal cutoff is independent of overall matrix size and can be determined strictly locally. We will see that this small change dramatically improves the practical performance of Strassen's algorithm.

In the remainder of this section we will show how to theoretically determine the optimal cutoff for both square and non-square matrices. The discussion can be based on any of the three performance models proposed in Section 2, each in turn being expected to give a more accurate prediction of the cutoff to be used in practice. In addition, the theoretical ideas will lead naturally to an empirical procedure for determining the optimal cutoff, which will be applied to our implementation in Section 7 and compared to the model predictions.

We again simplify the present discussion by assuming that matrix dimensions are always even, reserving treatment of the fully general case until the next section. In order to establish the proper cutoff point for both the square and rectangular case, we need to characterize the set of positive integers (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

$$M\left(m,k,n\right) \le 7M\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right) + 4G\left(\frac{m}{2},\frac{k}{2}\right) + 4G\left(\frac{k}{2},\frac{n}{2}\right) + 7G\left(\frac{m}{2},\frac{n}{2}\right). \tag{8}$$

Any of the different models for M and G may be used. The set of solutions to this inequality will be the values for m, k, and n for which one should switch to the standard algorithm; at values in the complement of this set, one more level of Strassen's algorithm should be applied.

We first consider the operation count model. Then (8) becomes

$$mkn \le 4(mk + kn + mn), \tag{9}$$

which is equivalent to

$$1 \le 4(1/n + 1/m + 1/k). \tag{10}$$

It is easy to completely characterize the positive integer solutions to (9) and (10). First we examine the case where $m = k = n$, i.e., square matrices. In this case we obtain $m \le 12$. In other words, the operation count is reduced, using Strassen's algorithm, when applied to matrices of order greater than 12, and not reduced when applied to matrices of order less than or equal to 12.

This suggests that, for the case so far considered, instead of recursing all the way down to matrices of dimension 1 (scalars), we should instead switch to regular matrix multiplication whenever the remaining matrix multiplications involve matrices whose order is 12 or less. To see how important this change is, we can compute from (6) the operation count cost of Strassen's algorithm with cutoff at order 12 for various square power-of-2 order matrix sizes by setting $m' = 8$ and varying d, thus obtaining the results shown in Table 2. Note that for these examples the recursion actually stops for matrices of order 8, the first size that is 12 or less. For comparison, Table 2 also shows operation count costs for Strassen's algorithm with full recursion and for standard matrix multiplication. Observe that Strassen's algorithm is already faster than regular matrix multiplication for 16×16 matrices and is 2.05 times faster for 1024×1024 matrices.

The situation is more complicated for rectangular matrices. We first note that (10) (equivalently, (9)) is symmetric in m, k, and n. Thus we may, without loss of generality, assume $m \le k \le n$. Given this assumption, we list all solutions of (10) in Table 3. For these (and all

$8 * 2^d$	Standard MM	Strassen (Winograd) (Cutoff = 12)	Strassen (Winograd) (Full Recursion)
8	960	960	1,738
16	7,936	7,680	13,126
32	64,512	57,600	95,722
128	4,177,920	2,991,360	4,859,338
256	33,488,896	21,185,280	34,261,126
512	268,173,312	149,280,000	240,810,922
1024	2,146,435,072	1,048,892,160	1,689,608,614
2048	17,175,674,880	7,357,973,760	11,842,988,938

Table 2: *Operation Count Costs for* $8 * 2^d \times 8 * 2^d$ *Matrices, Cutoff 12.*

m	k	n
2	$k \geq 2$	$n \geq k$
4	$k \geq 4$	$n \geq k$
6	$6 \leq k \leq 12$	$n \geq k$
6	14	$14 \leq n \leq 84$
6	16	$16 \leq n \leq 48$
6	18	$18 \leq n \leq 36$
6	20	$20 \leq n \leq 30$
6	22	$22 \leq n \leq 26$
6	24	24
8	8	$n \geq k$
8	10	$10 \leq n \leq 40$
8	12	$12 \leq n \leq 24$
8	14	$14 \leq n \leq 18$
8	16	16
10	10	$10 \leq n \leq 20$
10	12	$12 \leq n \leq 14$
12	12	12

Solutions with m, k, n *even,* $m \leq k \leq n$*. Symmetric cases are also solutions.*

Table 3: *Cutoff Dimensions for Non-square Matrices.*

symmetric) cases, Strassen's algorithm should not be applied. Thus, regular matrix multiplication should be used when multiplying 6×22 and 22×26 or 22×6 and 6×26 matrices. When (10) is not satisfied, another level of recursion should be applied. As examples, observe that the condition is not met if m, k, and n are all greater than 12, and that $m = 6, k = 14, n = 86$ also does not satisfy the condition. Thus, a Strassen recursion should be applied when multiplying matrices with all dimensions greater than 12, or when multiplying 6×14 and 14×86 matrices.

The analysis for rectangular matrices 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 (9) should be used instead of a simpler condition derived from the square cutoff, ($m \leq 12$ or $k \leq 12$ or $n \leq 12$), which has been used by others [4, 11].

Cutoffs are addressed by almost all work on Strassen's algorithm. Strassen, in [28], used the flexibility of cutoff to reduce the cost of his algorithm for square matrices whose dimensions are not all powers of 2, thereby obtaining the constant of 4.7 on his upper bound for complexity in the square case. Other early theoretical work on the square case includes [12] and [13], where this constant is reduced further. Speiß [27] seems to be the first to give an operation count theory of optimal cutoffs on even inputs for both square and rectangular matrices, in an equation similar to our (8). Thus, his development also expresses that strictly local analysis suffices to theoretically determine optimal cutoff for even inputs. More recently Higham [14] and Knight [19] explore minimization over full operation count formulas rather than solution of any local inequality such as (8). Going beyond operation count theory, implementations supporting early work are generally research codes, with limited reporting of results. Cohen et al. [6] address only the square case, and timing results are only given for the square case in both Brent [5] and Speiß [27]. Recent work emphasizes either theory [14, 19] or practice [3, 4, 11], but not both.

In this work we extend the theory to other models and tie this to the development and examination of a practical implementation. The deficiencies of the operation count model, especially the fact that it does not take into account the higher efficiency of standard matrix multiplication as compared to matrix addition/subtraction, will most likely lead to underestimates in its computation of optimal cutoff values. We can therefore carry out an analysis similar to the one above using either of our two alternate performance models to obtain more accurate predictions for the optimal cutoff.

For example, the second performance model leads to the condition

$$mkn \leq 2\gamma/\mu(4mk + 4kn + 7mn),$$

which reduces to

$$m \leq 30\gamma/\mu,$$

when the input matrices are square. Recall that γ and μ correspond to costs for matrix multiplication and addition, respectively. Here we can even more clearly see that the relative efficiency of matrix multiplication and addition/subtraction plays a crucial role in determining optimal cutoff criteria. This model should give a more accurate prediction of the value for the optimal cutoff to be used in practice. Our third model, which includes low order terms, should produce even more accurate predictions for an even larger range of matrix shapes and sizes.

Alternatively, instead of using the various models to predict the proper cutoff condition, one can empirically determine the appropriate cutoff in a manner very similar to the theoretical analysis. The idea is to determine, via measured compute times, the smallest matrix dimensions such that one application of Strassen's algorithm is faster than regular matrix multiplication. This will require a more complicated set of experiments for rectangular matrices than for square ones. A discussion of empirical results of this type can be found in Section 5.4; empirical measurements and comparisons of model predictions to empirical findings will be presented in Section 7.4.

4. Odd-Sized Matrices

In order to apply one level of Strassen's algorithm the row and column dimensions of the input matrices must be even. If any of the dimensions are odd, some technique must be applied to make the dimensions even before Strassen's algorithm can be applied. This is achieved by adding or deleting rows and/or columns to or from the input matrices. After Strassen's algorithm is applied a correction is necessary to obtain the desired matrix product.

In this section we discuss two techniques for handling matrices with odd-sized dimensions. The first approach, called *padding*, embeds the input matrices in larger matrices, by adding extra rows and/or columns typically containing zeros, to obtain matrices whose dimensions are even. After Strassen's algorithm is applied to this larger matrix, the desired product is obtained by deleting the extra rows and columns.

There is some flexibility in the elements and location of the extra rows and columns. Let $C = AB$ be the desired matrix product, and let

$$\tilde{A} = \left(\begin{array}{ccc|c} & & & 0 \\ & A & & \vdots \\ & & & 0 \\ \hline * & \cdots & * & * \end{array}\right) \text{ and } \tilde{B} = \left(\begin{array}{ccc|c} & & & * \\ & B & & \vdots \\ & & & * \\ \hline * & \cdots & * & * \end{array}\right)$$

be matrices whose dimensions are even, where * indicates an unspecified value. Then Strassen's algorithm can be used to compute

$$\tilde{A}\tilde{B} = \left(\begin{array}{ccc|c} & & & * \\ & AB & & \vdots \\ & & & * \\ \hline * & \cdots & * & * \end{array}\right).$$

The product $C = AB$ appears in the upper-left block independent of the values substituted for the *'s in A and B. The same would be true if the zeroes would have been placed in the bottom row of B instead of the last column of A. In the above diagram one extra row and column has been inserted; however, if desired, additional rows and columns can be inserted so long as the resulting matrix has even dimensions. Furthermore, the extra rows and columns could have been inserted in arbitrary locations so long as the column index in the first matrix matches up with the row index in the second matrix. Typically the extra rows and columns contain zeros and are inserted so that A and B are in the top-left or bottom-right block (see [11] for an exception).

The second approach to dealing with odd-sized matrices, called *peeling*, applies Strassen's algorithm to smaller matrices obtained by deleting rows and columns. In this case, after Strassen's algorithm is applied, some additional work, called *fixup*, is required to include the contribution of the rows and columns that were deleted.

The peeling approach corresponds to block matrix multiplication. The input matrices are partitioned into blocks, possibly of different sizes, such that the matrix product is compatible with the dimensions of the blocks, i.e., if A_{ij} is the ij-block of the matrix A and B_{jk} is the jk-block of the matrix B, then the column dimension of A_{ij} must equal the row dimension of B_{jk} for all i, j, and k. If the dimensions of the blocks are compatible in this sense, then

C_{ik}, the ik-block of $C = AB$, is equal to $\sum_j A_{ij} B_{jk}$, where j runs over the blocks in the i-th column of the block matrix obtained by partitioning A. After partitioning the input matrices into blocks, Strassen's algorithm can then be applied to compute the block products of matrices whose dimensions are even. The fixup work includes the block products that are not computed with Strassen's algorithm and the work required to combine the block products.

One partitioning scheme chooses the upper left-hand block to be the largest submatrix, with even dimensions, containing the element in the upper left-hand corner. In this scheme there can be at most four blocks, which occurs when both the row and column dimensions are odd. In this case,

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

where A_{11} has even dimensions, a_{12} is a column vector, a_{21} is a row vector, and a_{22} is a single element. Similarly,

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

where B_{11} has even dimensions, b_{12} is a column vector, b_{21} is a row vector, and b_{22} is a single element. Using this partitioning, the product $C = AB$ is computed as

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

where $A_{11}B_{11}$ is computed using Strassen's algorithm, and the other computations constitute the fixup work.

When padding is used, it is possible to add sufficiently many rows and columns so that only even dimensions are encountered in all of the recursive applications of Strassen's algorithm. Similarly, with peeling, it is possible to delete sufficiently many rows and columns so that only even dimensions are encountered in all of the recursive applications of Strassen's algorithm. When padding or peeling is applied in this manner to the initial matrices, before any applications of Strassen's algorithm, the padding or peeling method will be called *static*. This is contrasted with *dynamic* padding and peeling, where the padding or peeling method is applied at each recursive level, typically adding or removing a single row and/or column at a time.

In many textbooks it is suggested that static padding be used. When Strassen's algorithm is recursively applied to square matrices until 1×1 matrices are encountered, this is referred to as "padding to the next power of two". Unfortunately padding to the next power of two is unacceptable in an implementation that must be efficient for all matrix dimensions, since the resulting computing times behave like a step function with large jumps in computing time following powers of two. In this scenario, the cost of multiplying square matrices of dimension $2^d + 1$ is approximately seven times the cost of multiplying matrices of size 2^d.

The large steps in computing time at powers of two can be diminished when Strassen's algorithm does not recurse all the way to matrices of dimension 1. The reason for this is that there is flexibility in the dimension of the matrices in the base case of the recursion and standard matrix multiplication can be used independent of whether the matrices at the base case have even or odd dimensions. For example, when using the cutoff criterion in (9) to multiply two square matrices of dimension 33, it is only necessary to pad to matrices of dimension 36 instead of 64, the next power of two. Strassen's algorithm is then applied for two levels of recursion after which standard matrix multiplication is used to multiply 9×9 matrices. In general, matrices of dimension $m \times m$ can be embedded in matrices of dimension $2^d q$, where $m \leq 2^d q$ and $c/2 < q \leq c$, where c is set to the optimal cutoff for square matrices implied by (9). Strassen's original paper [28] used this flexibility to obtain an algorithm whose operation count is less than $4.7m^{\lg(7)}$ for all $m \times m$ matrices. Using Winograd's variant of Strassen's algorithm and a more careful application of this strategy, Fischer and Probert [13] improved this bound to $4.537m^{\lg(7)}$. Even with this improvement the computing time with static padding suffers from large jumps.

The corresponding situation for static peeling is even worse. One approach to static peeling is to partition the input matrices into 2×2 block matrices such that the upper left-hand block is the largest possible matrix whose dimensions are powers of two. The product of the upper left-hand matrices are computed with Strassen's algorithm and the remaining products are computed with the standard algorithm. The maximum computing time of this algorithm is asymptotically the same as the cost of standard matrix multiplication. To see this, observe that matrices of dimension $m = 2^d - 1$ are partitioned into 4 blocks and the cost of the three block products that use standard matrix multiplication is approximately 3/8 the cost of using standard matrix multiplication to multiply the entire matrices. Alternative algorithms using static peeling can be designed so that their asymptotic computing time is $\Theta(m^{\lg(7)})$; however these algorithms involve more complicated partitioning schemes. Due to these remarks, static peeling will not be considered further in this paper.

4.1. Comparison of Padding and Peeling

In this section, the operation count model is used to compare various padding and peeling algorithms. In order to simplify the analysis, we assume that all matrices are square of order m. Recurrence relations are derived for the cost of the different algorithms and the techniques used to analyze them are applicable to all three cost models discussed in Section 2; however, the specific conclusions obtained will vary depending on the particular model and the parameters used in the model. Using operation count, it is shown that dynamic peeling is the best method for dealing with odd dimensions. Since operation count is not always indicative of actual performance, it cannot be concluded that dynamic peeling is the best method in practice. Nonetheless, the conclusions obtained using operation count justify our decision to study dynamic peeling in a serious implementation. In addition to providing a comparison of padding and peeling, the results in this section show that the cost of Strassen's algorithm depends on the particular input sizes in a fundamental way. This suggests that the subtle dependence of the observed computing time of Strassen's algorithm on the dimension of the inputs is due to the algorithms themselves and not just the effects of the computer architecture or variability of the clock used to measure computing time.

First we present recurrence relations for the cost of Strassen's algorithm using static and dynamic padding. Let $P^*(m)$ denote the cost of the matrix multiplication algorithm that uses Strassen's algorithm with static padding. When static padding is used, the input matrices are embedded into matrices of dimension $2^d q$, where $c/2 < q \leq c$ and $2^d(q-1) < m \leq 2^d q$. This implies that q and d are uniquely defined with $d = \lceil \lg(m/c) \rceil$ and $q = \lceil m/2^d \rceil$. For example, if $m = 81$ and $c = 12$ then $8 \cdot 10 = 80 < m \leq 8 \cdot 11 = 88$, so that $q = 11$ and $d = 3$. By definition of q and d, Strassen's algorithm will be recursively applied d times until $q \times q$ matrices are encountered, and the computing time is therefore,

$$P^*(m) = \begin{cases} M(m) & \text{when } m \leq c \\ W(2^d q) & \text{where } 2^d(q-1) < m \leq 2^d q \text{ and } c/2 < q \leq c. \end{cases} \tag{11}$$

This implies that $P^*(m)$ is a step function with steps of size 2^d for all matrix dimensions m such that $2^{d-1}c < m \leq 2^d c$. When operation count is used for cost (11) becomes

$$P^*(m) = \begin{cases} 2m^3 - m^2 & \text{when } m \leq c \\ 7^d(2q^3 - q^2) + 5q^2(7^d - 4^d) & \text{where } 2^d(q-1) < m \leq 2^d q \text{ and } c/2 < q \leq c. \end{cases}$$

Unlike static padding, the cost of dynamic padding is not constant for matrix dimensions $2^d(q-1) < m \leq 2^d q$. There is a gradual increase in the cost function over this range. It will be shown that the cost of dynamic padding is always less than or equal to the cost of static padding. Nonetheless, there remains a fairly large jump between the cost of dynamic padding at $m = 2^d q$ and $m = 2^d q + 1$.

Let $P^+(m)$ denote the cost of the matrix multiplication algorithm that uses Strassen's algorithm with dynamic padding. It is assumed that a single row and column is added whenever a matrix with odd dimensions is encountered. In the previous example, where $m = 81$ and $c = 12$, the following matrix dimensions are encountered as Strassen's algorithm is recursively applied and padding is performed: $81 \uparrow 82 \rightarrow 41 \uparrow 42 \rightarrow 21 \uparrow 22 \rightarrow 11$, where $\uparrow$ indicates a padding step and $\rightarrow$ indicates a recursive step of Strassen's algorithm. The cost function for dynamic padding is defined by

$$P^+(m) = \begin{cases} M(m) & \text{when } m \leq c \\ 7P^+(m/2) + 15G(m/2) & \text{when } m \text{ is even} \\ P^+(m+1) = 7P^+((m+1)/2) + 15G((m+1)/2) & \text{when } m \text{ is odd.} \end{cases}$$

When operation count is used for the cost, this becomes

$$P^+(m) = \begin{cases} 2m^3 - m^2 & \text{when } m \leq c \\ 7P^+(m/2) + 15(m/2)^2 & \text{when } m \text{ is even} \\ P^+((m+1)/2) = 7P^+((m+1)/2) + 15((m+1)/2)^2 & \text{when } m \text{ is odd.} \end{cases} \tag{12}$$

Observe that for m odd, $P^+(m) = P^+(m+1)$ and that $P^+(m)$ is a monotonic non-decreasing function. Furthermore, $P^+(2^d q) = P^*(2^d q)$, when $q \leq c$. These observations, along with the fact that $P^*(m) = P^*(2^d q)$ for $2^d(q-1) < m \leq 2^d q$ imply the following lemma.

Lemma 1. *If static and dynamic padding use the same cutoff criterion, then* $P^+(m) \leq P^*(m)$.

The savings provided by dynamic padding comes entirely from a reduction in the number of additions/subtractions performed during stages (1), (2), and (4) of Strassen's algorithm, since the number and size of the calls to standard matrix multiplication is the same for both algorithms. To see this observe that if $2^d(q-1) < m \leq 2^d q$, then the static padding algorithm requires d recursions and the dimension of the resulting standard matrix multiplications is q. If the dynamic padding algorithm is applied to the same input, after r recursions the dimension of the resulting matrices are $\lceil (m/2^r) \rceil$. Since $m = (q-1)2^d + j$, where $0 < j \leq 2^d$, the dimension of the matrices after d recursions is $\lceil ((q-1) + j/2^d) \rceil = q$. This implies that the dynamic padding algorithm, like the static padding algorithm, uses d recursions, resulting in $q \times q$ matrices at the base of the recursion.

The difference between dynamic and static padding can be seen in the example where $m = 81$. Observe that $P^*(81) = 1,040,358$ and $P^+(81) = 1,032,018$. The ratio $P^*(81)/P^+(81) \approx 1.008$, shows that dynamic padding provides a modest improvement over static padding. Additionally, we find that the ratio $P^+(81)/P^+(80) \approx 1.304$ and the ratio $P^*(81)/P^*(80) \approx 1.315$. Thus, there is still a fairly large jump in the operation count for dynamic padding between $m = 2^d q$ and $m = 2^d q + 1$, the point where the step occurred in the cost of static padding.

Dynamic peeling provides a significant reduction in the cost of Strassen's algorithm for inputs of size $2^d q + 1$. Recall that dynamic peeling uses block matrix multiplication with Strassen's algorithm to compute the product of the blocks in the upper-left corner. Assume that the $m \times m$ input matrices are partitioned into 2×2 block matrices, where the blocks have dimension $(m-f) \times (m-f)$, $(m-f) \times f$, $f \times (m-f)$, and $f \times f$, where $0 \leq f < m$. The fixup cost, denoted by $F(m-f, f)$, is the cost to compute all but the product of the two $(m-f) \times (m-f)$ blocks. If f is a constant, then $F(m-f, f) = O(m^2)$.

Let $P^-(m)$ denote the cost of the matrix multiplication algorithm that uses Strassen's algorithm with dynamic peeling with $f = 1$, then

$$P^-(m) = \begin{cases} M(m) & \text{when } m \leq c \\ 7P^-(m/2) + 15G(m/2) & \text{when } m \text{ is even} \\ P^-(m-1) + F(m-1, 1) & \text{when } m \text{ is odd.} \end{cases} \tag{13}$$

In general, when operation count is used, $F(m-f, f) = M(m) - M(m-f)$. Substituting operation count for the functions G, M, and F, (13) becomes

$$P^-(m) = \begin{cases} 2m^3 - m^2 & \text{when } m \leq c \\ 7P^-(m/2) + 15(m/2)^2 & \text{when } m \text{ is even} \\ P^-(m-1) + 6(m-1)^2 + 4(m-1) + 1 & \text{when } m \text{ is odd.} \end{cases} \tag{14}$$

The cost of dynamic padding at $m = 81$ is $P^-(81) = 829,921$, which is substantially smaller than $P^+(81) = 1,032,018$. The ratio $P^+(81)/P^-(81) \approx 1.244$. Also the ratio $P^-(81)/P^-(80) \approx 1.049$ which shows that the jump from $m = 80$ to $m = 81$ is quite reasonable. While the performance of dynamic peeling at dimensions one larger than $2^d q$ is significantly better than the performance of dynamic padding, there are dimensions where dynamic padding outperforms dynamic peeling. For example, consider $m = 87 = 11 \cdot 2^3 - 1$. Here $P^-(87) = 1,040,801$ and $P^+(87) = 1,040,358$. The dimension $m = 87$ is particularly bad for dynamic peeling. There are three recursive applications of Strassen's algorithm and peeling is required at each recursive stage. The sequence of matrix dimensions is $87 \downarrow 86 \rightarrow 43 \downarrow 42 \rightarrow 21 \downarrow 20 \rightarrow 10$, where $\downarrow$ indicates an application of peeling and, as before, $\rightarrow$ indicates a reduction in matrix dimension due to an application of Strassen's algorithm. In this example, only one padding is required for

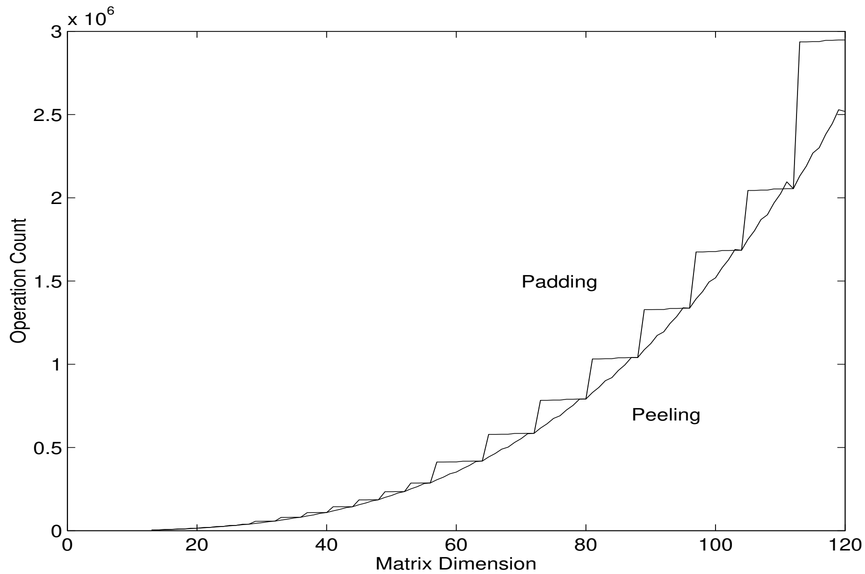

Figure 1: *Comparison of dynamic padding and dynamic peeling ($c = 13$).*

dynamic padding which has a sequence of matrix dimensions equal to $87 \uparrow 88 \rightarrow 44 \rightarrow 22 \rightarrow 11$. The ratio $P^+(87)/P^-(87) \approx 0.9996$, which is much smaller than the previous example where dynamic peeling outperformed dynamic padding.

In general, dynamic peeling outperforms dynamic padding most of the time and does so by a larger ratio than when dynamic padding outperforms dynamic peeling. The analysis of this phenomenon is complicated by the fact that $P^-(m)$ is not a monotonic function. The non-monotonic behavior of $P^-(m)$ is first observed at $m = 87$, where $P^-(87) = 1,040,801 > P^-(88) = 1,040,358$. Figure 1 shows a graph of the functions $P^+(m)$ and $P^-(m)$, where a cutoff of $c = 13$ is used instead of the cutoff $c = 12$ from condition (9). The reason for this change will be explained in Section 4.2.

The remainder of this section compares various aspects of dynamic peeling and dynamic padding. First, the effects of odd dimensions on square cutoff selection are investigated; second, the number of recursive applications in dynamic padding and peeling are compared; third, the amount of padding and peeling is determined as a function of the input dimensions; and finally, the cost functions of the two algorithms are compared.

4.2. Cutoff criteria for square matrices with odd dimension

The inclusion of odd-sized matrices requires a modification to the analysis of the cutoff criterion. In Section 3 we showed that for square matrices Strassen's algorithm should be applied to all matrices with even dimension greater than or equal to 12. Since $P^-(13) = M(13)$ when $c = 12$, c might as well be set to 13. Moreover, if m is odd,

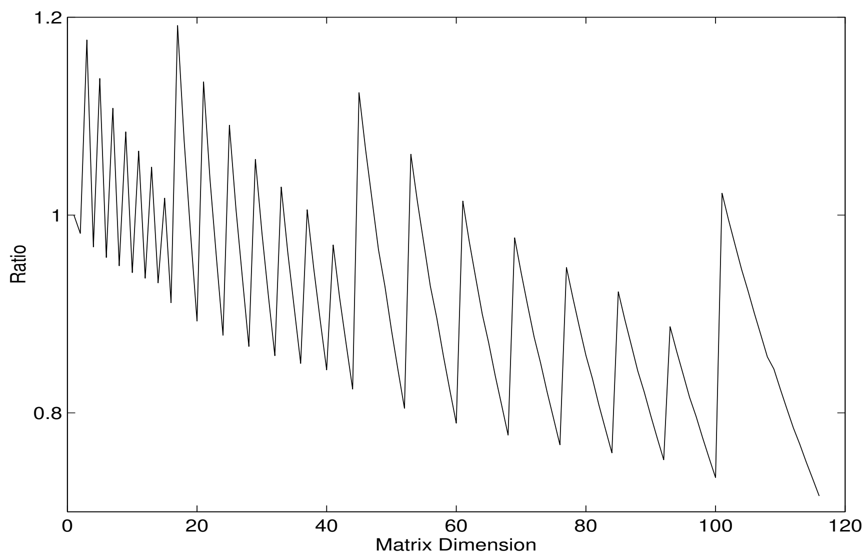

Figure 2: *Ratio $P^+(m)/M(m)$ of dynamic padding and standard matrix multiplication.*

$$\begin{aligned} P^-(m) &= P^-(m-1) + F(m-1,1) \\ &= P^-(m-1) + (M(m) - M(m-1)) \\ &= M(m) + (P^-(m-1) - M(m-1)). \end{aligned}$$

Since $P^-(m) < M(m)$ for even $m > 12$, we conclude that $P^-(m) < M(m)$ for all $m > 13$. This argument depends on the fact that $F(m-1,1) = M(m) - M(m-1)$, which is true for operation count; however, it is likely not to be true for other computational models. Thus, if the model is changed, it is possible for standard matrix multiplication to be faster than Strassen's algorithm for larger odd-sized matrices and the appropriate choice of cutoff would change accordingly.

Even when operation count is used, the analysis for dynamic padding is more complicated. When $c = 13$, there are both even and odd values of m greater than 12 such that $P^+(m) > M(m)$. Figure 2 shows that the ratio $P^+(m)/M(m)$ oscillates above and below one. The largest value of m such that $P^+(m) > M(m)$ is 113, where $P^+(113) = 2,936,655$ and $M(113) = 2,873,025$. The number 113 is such that, at each recursive stage of the dynamic padding algorithm, padding occurs. The sequence of matrix sizes that occur is $113 \uparrow 114 \rightarrow 57 \uparrow 58 \rightarrow 29 \uparrow 30 \rightarrow 15 \uparrow 16 \rightarrow 8$. These observations suggest, that when dynamic padding is used, it may be beneficial to use standard matrix multiplication for odd matrices of dimension greater than 13. Therefore the cutoff in condition (9) should either be made larger or separate stopping criteria should be used for matrices with even and odd dimensions.

Let $P_c^+(m)$ denote the value of $P^+(m)$ when standard matrix multiplication is used for all matrices of dimension c or less. The average of the ratio $P_{13}^+(m)/P_c^+(m)$ for m between 13 and 120 was computed for several values of c. When $c = 21$ the average ratio is approximately 1.0322, when $c = 23$ the ratio is approximately 1.0326, and when $c = 25$ the ratio is approximately

1.0316. As c increases further the ratio continues to decrease. Therefore, in order to minimize total operation count when using dynamic padding, c should be set to 23.

4.3. The number of recursive levels in Strassen's algorithm

The number of recursive levels in Strassen's algorithm can be different when dynamic padding and dynamic peeling are used to handle matrices with odd dimensions. Assume that standard matrix multiplication is used for matrices whose dimension is less than or equal to c. When dynamic peeling is used, exactly d levels of recursion are used to multiply $m \times m$ matrices when

$$2^{d-1}(c+1) \leq m < 2^d(c+1),$$

and when dynamic padding is used, exactly d levels of recursion are used when

$$2^{d-1}c < m \leq 2^d c.$$

If $c = 13$ for both algorithms then one recursion is required by dynamic peeling for $m = 14$ through 27 and one recursion is required by dynamic padding for $m = 14$ through 26; two recursions are required by dynamic peeling for $m = 28$ through 55 and two recursions are required by dynamic padding for $m = 27$ through 52; three recursions are required by dynamic peeling for $m = 56$ through 111 and three recursions are required by dynamic padding for $m = 53$ through 104. Notice that in this scenario, dynamic padding requires one extra recursion for $m = 27$, $m = 53$ through 55, and $m = 105$ through 111. The following lemma summarizes these observations.

Lemma 2. *Assume that Strassen's algorithm uses standard matrix multiplication for all matrices of dimension c or less. Then, the number of levels of recursion used by Strassen's algorithm to multiply $m \times m$ matrices is*

$$\lfloor \lg(m/(c+1)) \rfloor + 1$$

when dynamic peeling is used, and is

$$\lceil \lg(m/c) \rceil$$

when dynamic padding is used.

4.4. Dynamic padding verses dynamic peeling

In this section the ratio $P^+(m)/P^-(m)$ is computed. As previously remarked there are inputs m where this ratio is less than one and other inputs m where the ratio is greater than one. Figure 3 plots $P^+(m)/P^-(m)$ when $c = 13$. This figures shows that dynamic peeling is better for most of the matrix dimensions in the plot.

Additional data concerning the ratio $P^+(m)/P^-(m)$ is summarized in Table 4. The ratio was computed for all matrix dimensions in the range from 13 to 1,000, 10,000, and 20,000 respectively. For each range the average value for the ratio and the percentage of inputs for which the ratio was greater than one were computed. This data shows that dynamic peeling is superior to dynamic padding about 86% of the time. Moreover dynamic padding requires 1.12 times as many operations on average. The data suggests that both the average ratio and the percentage of times the ratio is greater than one approach a limit. The percentages and

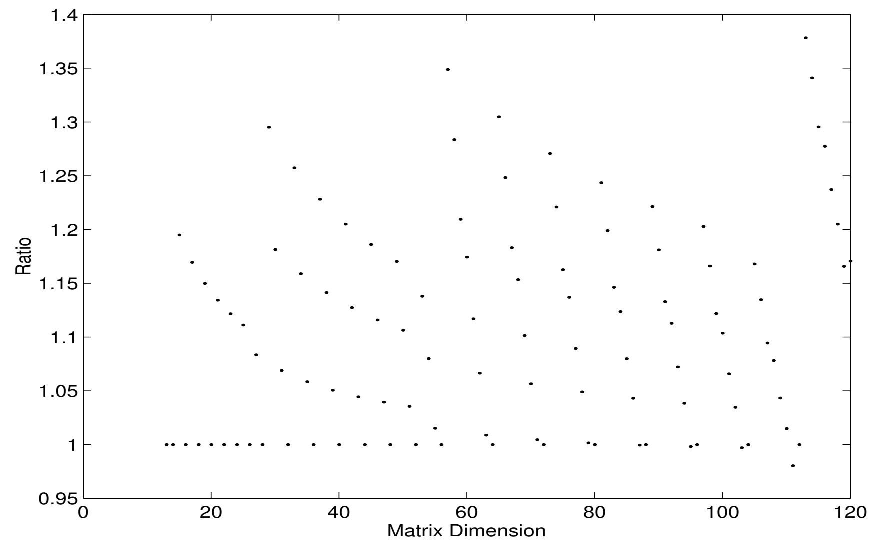

Figure 3: *Ratio of dynamic padding to dynamic peeling ($c = 13$).*

average ratios were computed over the same ranges when $c = 23$ for dynamic padding as shown in Table 5. Recall from Section 4.2 that the optimal cutoff for dynamic padding turned out to be $c = 23$. Changing the value of the cutoff for dynamic padding leads to a reduction in the average ratio from 1.120 to 1.087. Note however, that the percentage of times dynamic peeling is superior increases from 86.2% to 91.2%.

Range for m	13,...,1000	13,...,10000	13,...,20000
Percentage $m : P^+(m)/P^-(m) > 1$	84.9%	86.2%	86.2%
Avg. $P^+(m)/P^-(m)$	1.118	1.120	1.120

Table 4: *Statistics for $P^+(m)/P^-(m)$ with $c = 13$.*

Range for m	13,...,1000	13,...,10000	13,...,20000
Percentage $m : P^+(m)/P^-(m) > 1$	91.9%	91.4%	91.2%
Avg. $P^+(m)/P^-(m)$	1.087	1.087	1.087

Table 5: *Statistics for $P^+(m)/P^-(m)$ with $c = 13$ for dynamic peeling and $c = 23$ for dynamic padding.*

For certain sequences it is easy to determine limiting values for the ratio $P^+(m)/P^-(m)$. Two sequences are considered. The first is the sequence $c2^d + 1$, which is favorable to dynamic peeling, and the second is the sequence $c2^d - 1$, which is favorable to dynamic padding. For both sequences, it is assumed that standard matrix multiplication is used for $c \times c$ matrices and $(c + 1) \times (c + 1)$ matrices.

Note that for certain values of c dynamic padding could obtain a lower operation count if the Strassen recursions stopped before reaching matrices of dimension $c+1$. For example, when $c = 11$, $2c + 1 = 23$, and as was seen in Section 4.2, the number of operations for the standard matrix multiplication algorithm for 23×23 matrices is less than the cost for the dynamic padding algorithm. Therefore, there are some values of c for which the ratios $P^+(c2^d + 1)/P^-(c2^d + 1)$ would be smaller if dynamic padding did fewer recursions. To simplify the calculations, we do not perform separate ratio computations for different values of c except in one case.

The sequence $c2^d+1$ is closed under dynamic padding. To see this observe that $(c2^d+2)/2 = c2^{d-1} + 1$. Because, $c2^d + 1$ is odd, dynamic padding is required at each recursive level of Strassen's algorithm. In contrast only one application of dynamic peeling is required since all of the remaining Strassen recursions encounter matrices with even dimensions. Similarly, the sequence $c2^d - 1$ is closed under dynamic peeling, since $c2^d - 1 = (c2^d - 2) + 1$ and $(c2^d - 2)/2 = c2^{d-1} - 1$. Moreover, since $c2^d - 1$ is odd, dynamic peeling is required at each recursive level of Strassen's algorithm. In contrast only one application of dynamic padding is required since $(c2^d - 1) + 1 = c2^d$.

The behavior illustrated by these two sequences is at the two extremes in terms of the amount of padding and peeling that is required. In general, if standard matrix multiplication is used to multiply $c \times c$ matrices, the number of recursive levels that require peeling for matrices of dimension $c2^d + j$, $0 \leq j < 2^d$, is equal to the number of one bits in the binary representation of j. For example, $c2^d - 1 = (c-1)2^d + 2^d - 1$ and the number of bits in $2^d - 1$ is d so that d peelings are required (one per recursive level). Assume that the value cutoff is the first dimension where standard matrix multiplication is used instead of Strassen's algorithm. Then all matrices of dimension $c2^d + j$, for c between cutoff and cutoff/2, will exhibit similar peeling behavior, and since there are 2^d possible bit patterns there will be 2^d possible peeling patterns for matrices requiring d levels of recursion. This phenomenon can be observed in the bands appearing in Figure 3.

The number of recursive levels that require padding can also be determined by looking at the bit pattern of j. The number of paddings is equal to the number of significant zeros, i.e., those not including the trailing zeros, when j is written as a $d + 1$ bit number. For example, $c2^d + 1$ requires d paddings and there are d significant zeros when 1 is written as $b_d \cdots b_1 b_0$ with $b_i = 0$ for $i > 0$ and $b_0 = 1$.

The number of operations required by the dynamic padding and dynamic peeling algorithms can be computed for the sequences $c2^d \pm 1$. This is done in the following lemma. The proof of the lemma uses the fact that the corresponding recurrences can be unwound into geometric sequences.

Lemma 3. *The following equations give the operation count for dynamic padding and dynamic peeling for matrices of size $c2^d \pm 1$, when standard matrix multiplication is used for matrices of dimension $c - 1$, c, and $c + 1$.*

$$\begin{aligned}
P^-(c2^d - 1) &= (2c^3 + 6c^2 - 6c + \frac{7}{3})7^d - 13c^2 4^d + 14c2^d - \frac{16}{3} \\
P^+(c2^d - 1) &= (2c^3 + 4c^2)7^d - 5c^2 4^d \\
P^-(c2^d + 1) &= (2c^3 + 4c^2)7^d + c^2 4^d + 4c2^d + 1 \\
P^+(c2^d + 1) &= (2c^3 + 10c^2 + 10c + \frac{7}{2})7^d - 5c^2 4^d - 6c2^d - \frac{5}{2}
\end{aligned}$$

Proof. Using recurrence (14), $P^-(c2^d-1) = P^-(c2^d-2) + 6(c2^d-2)^2 + 4(c2^d-2) + 1$, which is equal to $7P^-(c2^{d-1}-1) + 39(c2^{d-1}-1)^2 + 8(c2^{d-1}-1) + 1$. Unwinding this recurrence until $d = 0$, the following sum is obtained.

$$\begin{aligned}
P^-(c2^d-1) &= 7^d M(c-1) + 39\sum_{i=0}^{d-1} 7^i (c2^{d-i-1}-1)^2 + 8\sum_{i=0}^{d-1} 7^i (c2^{d-i-1}-1) + \sum_{i=0}^{d-1} 7^i \\
&= (M(c-1) + 13c^2 - 14c + \frac{16}{3})7^d - 13c^2 4^d + 14c2^d - \frac{16}{3} \\
&= (2c^3 + 6c^2 - 6c + \frac{7}{3})7^d - 13c^2 4^d + 14c2^d - \frac{16}{3}.
\end{aligned}$$

After one padding step is applied to matrices of dimension $c2^d - 1$ the dimension becomes $c2^d$ and no further padding steps will be required during Strassen's algorithm. Therefore,

$$\begin{aligned}
P^+(c2^d-1) &= W(c2^d) \\
&= 7^d M(c) + 5c^2(7^d - 4^d) \\
&= (2c^3 + 4c^2)7^d - 5c^2 4^d.
\end{aligned}$$

Similiarly, when dynamic peeling is applied to matrices of dimension $c2^d + 1$, only one peeling step is required and

$$\begin{aligned}
P^-(c2^d+1) &= W(c2^d) + 24(c2^{d-1})^2 + 8c2^{d-1} + 1 \\
&= (M(c) + 5c^2)7^d + c^2 4^d + 4c2^d + 1 \\
&= (2c^3 + 4c^2)7^d + c^2 4^d + 4c2^d + 1.
\end{aligned}$$

However, if dynamic padding is applied to matrices of dimension $c2^d + 1$ recurrence (12) implies that

$$P^+(c2^d+1) = 7P^+(c2^{d-1}+1) + 15(c2^{d-1}+1)^2.$$

Unwinding this recurrence until $d = 0$ the following sum is obtained.

$$\begin{aligned}
P^+(c2^d+1) &= 7^d M(c+1) + 15\sum_{i=0}^{d-1} 7^i (c2^{d-i-1}+1)^2 \\
&= (M(c+1) + 5c^2 + 6c + \frac{5}{2})7^d - 5c^2 4^d - 6c2^d - \frac{5}{2} \\
&= (2c^3 + 10c^2 + 10c + \frac{7}{2})7^d - 5c^2 4^d - 6c2^d - \frac{5}{2}.
\end{aligned}$$

∎

Using the previous lemma, we can compute the ratio of the operation count $P^+(m)/P^-(m)$ when m is equal to $c2^d \pm 1$.

$$\begin{aligned}
\lim_{d\to\infty} P^+(c2^d-1)/P^-(c2^d-1) &= \frac{6c^3+12c^2}{6c^3+18c^2-18c+7} \\
&= \frac{18}{13} \approx 1.3846 \quad \text{when } c=1 \\
&= \frac{7200}{7627} \approx 0.9440 \quad \text{when } c=10 \\
&= \frac{15210}{15997} \approx 0.9508 \quad \text{when } c=13 \\
\lim_{d\to\infty} P^+(c2^d+1)/P^-(c2^d+1) &= \frac{4c^3+20c^2+20c+7}{4c^3+8c^2} \\
&= \frac{17}{4} \approx 4.25 \quad \text{when } c=1 \\
&= \frac{2069}{1600} \approx 1.2931 \quad \text{when } c=10 \\
&= \frac{829}{676} \approx 1.2263 \quad \text{when } c=13
\end{aligned}$$

The values $c=1$ and $c=13$ are special, and for this reason a value between 1 and 13 was considered. When $c=1$ in $c2^d-1$, dynamic peeling must stop after $d-1$ steps. Nonetheless, the limit computed above, $18/13$, is the same when this is taken into account. Also when $c=13$, dynamic padding will benefit from an additional recursion. The limit $P^+(c2^d+1)/P^-(c2^d+1)$ is recalculated with an additional recursive step taken for dynamic padding. The new limit becomes

$$\frac{\frac{7}{4}c^3+\frac{49}{4}c^2+\frac{61}{4}c+\frac{25}{4}}{2c^3+4c^2}$$

and when $c=13$ this becomes $12239/10140 \approx 1.2070$. Note that when $c=1$ the operation count for dynamic padding is worse with the extra recursion. The above limit when $c=1$ is $71/12 \approx 5.9167$.

These calculations show that dynamic peeling is better than dynamic padding in situations particularly favorable to dynamic peeling and dynamic padding is better than dynamic peeling in situations particularly favorable to dynamic padding, except when $c=1$, in which case dynamic peeling is better in both cases. In fact, it can be shown, that if the cutoff is equal to one, dynamic peeling is always superior to dynamic padding. In general, when dynamic peeling is superior it is so by a larger ratio than when dynamic padding is superior.

5. Implementation Issues

In this section we discuss DGEFMM, our implementation of Strassen's algorithm. As mentioned earlier, key implementation issues are the determination of cutoff criteria, appropriate handling of odd-sized matrices, and controlling the amount of additional memory for temporary computations that is inherent to the algorithm. Our objective was to provide a portable, high-performance, general implementation. Thus, coding was done in a high level language with calls to high performance BLAS primitives for the required matrix operations. Section 5.1 below describes the high level interface to DGEFMM, and Section 5.2 shows how the equations in the four stages of Strassen's algorithm are implemented. The latter section also discusses memory usage in detail. Section 5.3 discusses our treatment of odd matrix dimensions, and in Section 5.4, we provide a method for determining the appropriate cutoff criterion for a particular machine. All of this information is put together in Section 5.5 where we summarize the overall program structure.

5.1. Interface

Since our implementation of Strassen's algorithm, called FMM, 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. We will refer to FMM to mean the algorithm described in this section, and DGEFMM to refer to our actual code. 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). We will later see that the more general input/output specifications of DGEMM can in fact be used advantageously in the recursive calls of Strassen's algorithm.

Unlike the standard matrix multiplication algorithm, Strassen's algorithm requires additional space, proportional to the size of the input matrices, to hold intermediate computations. We provide two interfaces to DGEFMM. One has a parameter for the user to pass scratch space, and one is identical to DGEMM, which allocates temporary space internally. In this section we will discuss the version having the explicit scratch space parameter to highlight its memory requirements.

All code except a few kernel routines were written in C. However, we chose to use column-based storage (FORTRAN style) for all of our matrices. This was done in order to easily call other BLAS routines, which have a FORTRAN interface. Also, the code is thereby compatible with the current BLAS DGEMM routine. Any program which calls DGEMM can easily call DGEFMM without any change in the calling sequence.

5.2. Temporary Allocation and Memory Usage

Intermediate results required by any variant of Strassen's algorithm need to be stored in temporary locations. Early works addressing the issue of minimizing the amount of auxiliary memory needed includes [5, 6, 20, 27]. All recent implementations have tried to reduce memory usage. In this section we obtain lower bounds on the auxiliary memory required for the Winograd variant of Strassen's algorithm and develop a computation schedule for the general case that achieves this lower bound. In all cases our requirements are as good or better than those of other codes.

Winograd's variant of 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. If stages (1) and (2) are intermingled with stage (3) then it suffices to have one temporary for stage (1) operations, one temporary for stage (2) operations, and seven temporaries for both stage (3) and (4) operations. The total memory required in this scenerio is bounded by

$$(mk + kn + 7mn)/3$$

If the temporaries used in stages (1) and (2) are reused in stages (3) and (4) this can be reduced to

$$7 \max(m, k) \max(k, n)/3,$$

which is the memory requirement for CRAY's implementation of Strassen's algorithm [9].

Further reduction is possible if all of the stages are allowed to overlap. Using Strassen's original algorithm, Bailey, et al. [4] devised a straightforward scheme which 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 present 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 has $\beta = 0$. This scheme is similar to the one used in the implementation of Winograd's variant reported in Douglas, et al., [11], 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 $(mk + kn + mn)/3$.

Figure 4 shows the computation graph for the Winograd variant of Strassen's algorithm presented in Section 2. The subblocks of matrices A and B are the initial inputs, shown labeling the vertices at the top of the graph. All edges are directed from inputs to outputs, higher vertices to lower ones below. Thus, for example, A_{11} and A_{21} are initial inputs, which are combined to produce S_3. S_3 is subsequently combined with T_3 to produce P_5, and so on, until the final results, subblocks of C, are produced at the bottom level.

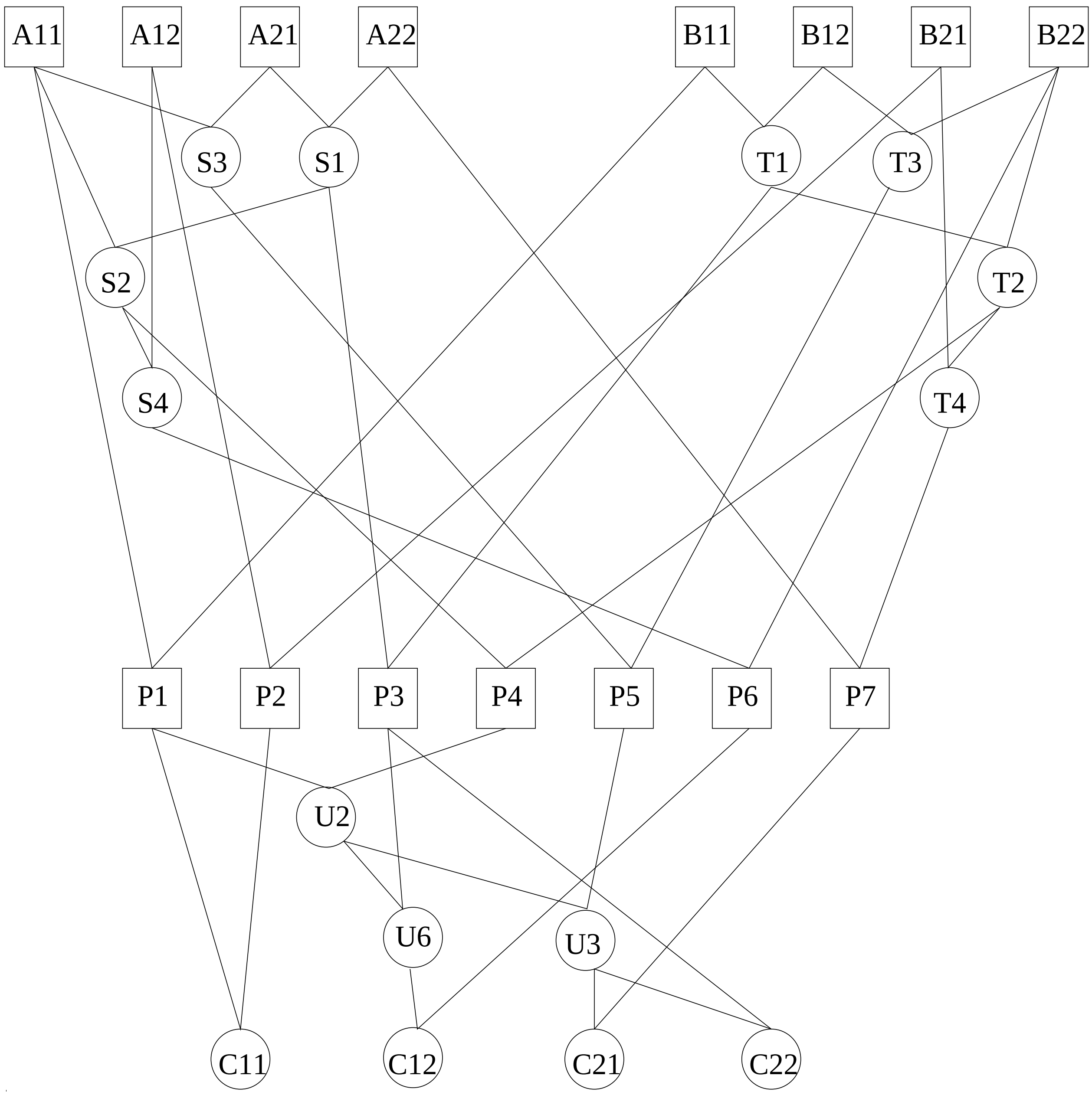

Figure 4: *Dependency graph for Winograd's variant of Strassen's Algorithm*

A schedule for completing these computations and the temporary variables needed for intermediate results can be established by playing the pebble game [23] on this directed graph. A pebble represents a temporary memory location. Placing a pebble on a vertex corresponds to performing the operation that computes the variable labeling that vertex and storing the result in the memory location represented by the pebble. We in fact only play the game on the subgraph obtained by deleting the top row of initial inputs, and the bottom row of final outputs, since these already have memory space allocated. In our pebble game we have the following rules:

1. A pebble may be removed from any vertex.

2. If all the immediate predecessors of a vertex are either initial inputs or have pebbles on them, a pebble may be placed on that vertex.

3. If all the immediate predecessors of a vertex have pebbles on them, and the computation to be performed at that vertex is not matrix multiplication, then a predecessor's pebble can be *moved* onto that vertex.

4. Rules 2 or 3 are applied to each vertex exactly once.

Each application of the second or third rules is a computation step, and the maximum number of pebbles on the graph at any one time is the number of temporary memory locations required for the complete computation. The third rule corresponds to being able to reuse space allocated for inputs (operands) to store the result. In the game defined above we allow this for computations of the form $X \leftarrow X + Y$ and $X \leftarrow X - Y$, but not for $X \leftarrow X * Y$, where X and Y are matrices. Observe that because of the third rule, at least three pebbles are required to pebble this graph, since obtaining result P_3 requires one pebble on S_1, one on T_1, and a third on P_3 since the third rule cannot be applied for this computation.

One pebbling of the graph using the above rules gives rise to our first schedule for performing these computations, STRASSEN1, described in Figure 5. With each computation step indicated, we show the temporary variable stored to, as well as the corresponding algorithmic variable being computed. We also indicate where the scaling by α and β is done. In the general case, we require space for six temporary variables, R_1 through R_6. R_1 stores A subblock sums and A subblock times B subblock products, and thus must be of size $m\max(k,n)/4$. R_2 stores B subblock sums, and hence is of size $kn/4$. The remaining four temporary variables store products only, thus each requires size $mn/4$. The space required for temporaries in the algorithm STRASSEN1 is thus

$$mn + m\max(k,n)/4 + kn/4.$$

If the seven products are computed recursively using the same algorithm, then the total additional memory is bounded by

$$4mn\sum_{i=1}^{\infty} 1/4^i + m\max(k,n)\sum_{i=1}^{\infty} 1/4^i + kn\sum_{i=1}^{\infty} 1/4^i = 4mn/3 + m\max(k,n)/3 + kn/3.$$

Note that if $m = k = n$, then the required space is $2m^2$. Also observe that when $\beta = 0$, the computation order, designed in fact to this purpose, allows use of the memory space allocated

STRASSEN1($A, B, \alpha, \beta; C$)

Implementation of Winograd's variant of Strassen's matrix multiplication algorithm.
Inputs: A is an $m \times k$ matrix. 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 $= m \max(k, n)/4$), R_2 (size $= kn/4$), R_3, R_4, R_5, R_6 (size $mn/4$). Note that if $\beta = 0$, we can use C for temporary space as follows: $R_6 = C_{12}$, $R_5 = C_{21}$, $R_4 = C_{22}$, $R_3 = C_{11}$.

Store to :		*Computation*	*Algorithmic Variables*
R_1	$\leftarrow$	$\alpha(A_{11} - A_{21});$	αS_3
R_2	$\leftarrow$	$B_{22} - B_{12};$	T_3
R_3	$\leftarrow$	$R_1 R_2;$	αP_5
R_1	$\leftarrow$	$\alpha(A_{21} + A_{22});$	αS_1
R_2	$\leftarrow$	$B_{12} - B_{11};$	T_1
R_4	$\leftarrow$	$R_1 R_2;$	αP_3
R_1	$\leftarrow$	$R_1 - \alpha A_{11};$	αS_2
R_2	$\leftarrow$	$B_{22} - R_2;$	T_2
R_5	$\leftarrow$	$R_1 R_2;$	αP_4
R_1	$\leftarrow$	$\alpha A_{12} - R_1;$	αS_4
R_2	$\leftarrow$	$\alpha(B_{21} - R_2);$	αT_4
R_6	$\leftarrow$	$R_1 B_{22};$	αP_6
R_6	$\leftarrow$	$R_6 + R_4;$	$\alpha P_6 + \alpha P_3$
R_1	$\leftarrow$	$\alpha(A_{11} B_{11});$	αP_1
R_5	$\leftarrow$	$R_5 + R_1;$	αU_2
R_6	$\leftarrow$	$\beta C_{12} + R_5 + R_6;$	$\beta C_{12} + \alpha U_2 + \alpha P_6 + \alpha P_3$
R_5	$\leftarrow$	$R_5 + R_3;$	αU_3
R_4	$\leftarrow$	$\beta C_{22} + R_4 + R_5;$	$\beta C_{22} + \alpha U_5$
R_3	$\leftarrow$	$\alpha A_{12} B_{21};$	αP_2
R_3	$\leftarrow$	$\beta C_{11} + R_3 + R_1;$	$\beta C_{11} + \alpha U_1$
R_1	$\leftarrow$	$A_{22} R_2;$	αP_7
R_5	$\leftarrow$	$\beta C_{21} + R_5 + R_1;$	$\beta C_{21} + \alpha U_4$

Figure 5: *STRASSEN1*

Implementation	$\beta = 0$	$\beta \neq 0$
CRAY `SGEMMS`	$7m^2/3$	$7m^2/3$
IBM ESSL `DGEMMS`	$1.40m^2$	$2.40m^2$
`DGEMMW` ([11])	$2m^2/3$	$5m^2/3$
STRASSEN1	$2m^2/3$	$2m^2$
STRASSEN2	m^2	m^2
FMM/`DGEFMM`	$2m^2/3$	m^2

Table 6: *Memory Requirements for Winograd's Strassen variant on $m \times m$ matrices.*

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 detailed in Figure 6. Again we show each computation, memory space used, scaling by α and β, and the corresponding algorithmic computation for each step. Like STRASSEN1, STRASSEN2 does not allow operations of the form $X \leftarrow X * Y$, where X and Y are matrices, but does allow $X \leftarrow X + Y$ and $X \leftarrow X - Y$. In addition, we now also allow multiply-accumulate operations, i.e., computations of the form $Z \leftarrow Z \pm X * Y$ for matrices X, Y, Z. Allowing this additional operation would change the computation graph from that shown in Figure 4, primarily by removing some of the vertices whose values do not now have to be separately computed and stored. However, by Rule 4, we do not allow recomputation of any variables. Thus, observe that S_1, T_1, and P_3 all must still be explicitly saved to memory since they are used in more than one subsequent computation. Thus, at least three pebbles are still required for the resulting graph, again based on the computation and storage of P_3.

By allowing the multiply-accumulate operation we can rearrange the computations in Winograd's variant of 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. 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

$$mn \sum_{i=1}^{\infty} 1/4^i + mk \sum_{i=1}^{\infty} 1/4^i + nk \sum_{i=1}^{\infty} 1/4^i = mn/3 + mk/3 + nk/3 = (mn + km + kn)/3.$$

When $m = k = n$, this gives m^2. In Table 6 we show the memory requirements for multiplying $m \times m$ matrices for various implementations of Winograd's Strassen variant. (As discussed in Section 7.1, IBM's ESSL Strassen implementation does not directly support the case where $\beta \neq 0$. Consequently, an additional copy of matrix C must be kept to perform the accumulate.)

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 6: *STRASSEN2*

Computation	α	β
P_3	1	0
αP_1	α	0
U_2	1	1
$C_{11} + \alpha P_2$	α	1
$C_{12} + P_6$	1	1
$\beta C_{21} + P_7$	1	β
U_3	1	1

Table 7: *Values for α and β in recursive calls.*

We can also see from Table 6 that the best use of our two computation schemes might be to use STRASSEN1 when $\beta = 0$, and STRASSEN2 otherwise. With this usage, however, STRASSEN2 requires a greater number of arithmetic operations. This raises the possibility that the savings achieved in memory usage would be at the expense of increased computation time. However, we first note that the multiply-accumulate operation is often optimized on high performance machines, leading us to expect no degradation. In addition, even if STRASSEN2 is passed a general α and β at the top level of recursion, none of the seven recursive calls at the next level require general α and β; in Table 7 we show the values of α and β needed for each recursive call in STRASSEN2. Furthermore, the total number of extra arithmetic operations throughout all of the recursive calls is just $O(mk+kn+mn)$. Finally, we have observed that the actual computing time for STRASSEN2 is in fact less than the computing time for STRASSEN1 (see Section 7). We conjecture that the reason for this is the increased data locality obtained be fusing the multiply-accumulate operations. In general, we have observed better performance as a result of reorganizing the computation so that data stored in a memory location is used as soon as it becomes available. Thus, we are able to stay with the design decision of using STRASSEN1 when $\beta = 0$, and STRASSEN2 for the more general case. This is reflected in the final entry in Table 6, 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 other two. In the case $\beta \neq 0$, our requirement of m^2 is lower than all the others, representing a reduction of 40, 57 and 58 percent, respectively, over that required by DGEMMW, CRAY SGEMMS, and IBM DGEMMS.

5.3. Odd-sized Matrices

As we saw in Section 4, odd matrix dimensions must be dealt with using some sort of padding or peeling technique. Our analysis there shows that dynamic peeling could be a useful method, but it had not been previously tested through actual implementation. Thus, we chose to use this technique in our implementation of Strassen's algorithm. To deal with odd dimensions we therefore strip off the extra column and/or row and add their contributions to the final result in a later round of fixup work. If m is odd, we must compute the contribution from the last row of A to the product $C = AB$. Similarly if k is odd the contribution from the last column of A and the last row of B must be included, and if n is odd the contribution from the last column of B must be included.

This approach has been dismissed by others [11] because of the inefficiency of some of these required fixup operations. Our implementation partially deals with this concern by combining the required operations 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-1 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 from our theoretical analysis, 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.

5.4. Setting Cutoff Criteria

As was shown theoretically in Section 3, selecting an appropriate cutoff can significantly improve the operation counts, and thereby performance, of any implementation of Strassen's algorithm. However, the operation count model probably seriously underestimates this value, thus necessitating timings of actual machine performance to determine a cutoff for practical use. 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 rectangular cutoff criterion whose parameters are set from empirical performance measurements. While all earlier work ties such a criterion to a single machine parameter based solely on square-case timings, we develop a new criterion based on multiple parameters and non-square, as well as square case, measurements.

To do this we compare measured timings for performing Strassen's algorithm with one level of recursion to timings for calling DGEMM on a given matrix size. In the case that input matrices are square, analysis is simpler since measurements will be a function of just one parameter, m. One set of tests can be done, with the matrix order m varying over a large range of typical input sizes. Results will point to a single crossover value, the matrix order τ, where Strassen's algorithm becomes faster than DGEMM. This leads to the cutoff criterion

$$m \leq \tau, \tag{15}$$

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 number 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. Letting matrix dimensions be given by $m \times k$ and $k \times n$, these are:

$$m \leq \tau \quad \text{or} \quad k \leq \tau \quad \text{or} \quad n \leq \tau, \tag{16}$$

and

$$mkn \leq \tau(nk + mn + mk)/3. \tag{17}$$

Condition (16), used in [4, 11], 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 3, is when one of the matrix dimensions is below the square cutoff and one of the other dimensions is large. Condition (17), suggested by Brent [5] and then Higham [14], scales the theoretical condition (9) by $\tau/12$, so that it reduces to the square condition (15) 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 7.4, 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 (9) is extended by including three parameters to take into account this asymmetry. Second, this extended condition is combined with the square condition (16) to preserve square-case behavior. The parameterized extended rectangular condition is

$$mkn \leq \rho_m nk + \rho_k mn + \rho_n mk, \tag{18}$$

which is equivalent to

$$1 \leq \rho_m/m + \rho_k/k + \rho_n/n. \tag{19}$$

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, (19) 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 (19) 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 (18). 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 (15) when $m = k = n$. Our compromise is to use the hybrid condition

$$\begin{gathered}\big(\, (mkn \leq \rho_m nk + \rho_k mn + \rho_n mk) \quad \text{and} \quad (m \leq \tau \quad \text{or} \quad k \leq \tau \quad \text{or} \quad n \leq \tau)\,\big) \\ \text{or} \\ (m \leq \tau \quad \text{and} \quad k \leq \tau \quad \text{and} \quad n \leq \tau),\end{gathered} \tag{20}$$

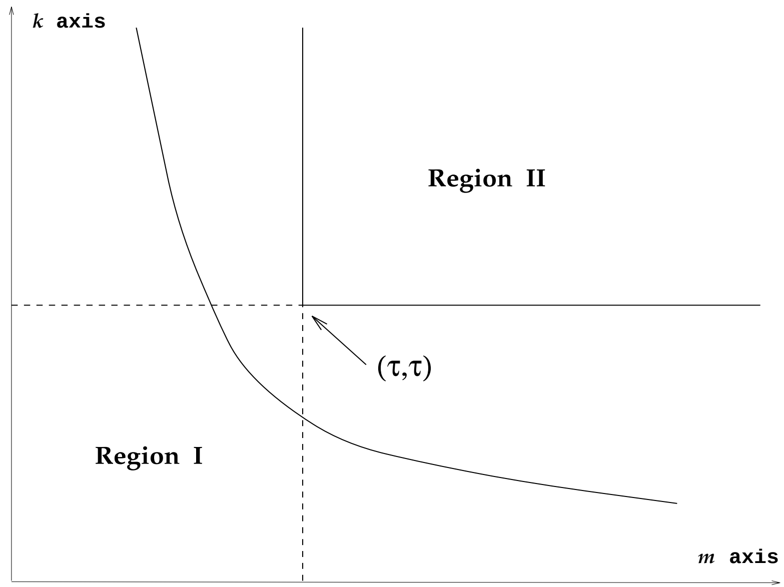

Figure 7: *Regions in positive octant for* $n = n_0 > \tau$.

which inherently allows recursion, via condition (16), 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 (18) rules in order to allow an extra application of Strassen's algorithm where it may be beneficial.

To visualize these various regions, we label as Region I the area specified by the condition ($m \leq \tau$ and $k \leq \tau$ and $n \leq \tau$) and that given by ($m > \tau$ and $k > \tau$ and $n > \tau$) we label Region II. Note that we always assume $m, k, n > 0$. In the schematic diagram in Figure 7 we show a two-dimensional slice of the positive octant where we have fixed n at $n = n_0$. Such a horizontal slice will intersect exactly one of Region I and Region II. Here we assume $n_0 > \tau$ so that we show the boundary of Region I with dotted lines and that of Region II with solid lines. In Figure 7 we also show the intersection in the first octant of this $n = n_0$ plane with the surface $mkn = \rho_m nk + \rho_k mn + \rho_n mk$. This curve is given by the equation

$$mk = \left(\frac{n_0}{n_0 - \rho_n} \right) (\rho_m k + \rho_k m), \quad n_0 > \rho_n, \tag{21}$$

has asymptotes

$$m = \left(\frac{n_0}{n_0 - \rho_n} \right) \rho_m \qquad \text{and} \qquad k = \left(\frac{n_0}{n_0 - \rho_n} \right) \rho_k,$$

and its intersection with the line $m = k$ is the point

$$\left(\left(\frac{n_0}{n_0 - \rho_n} \right) (\rho_m + \rho_k), \left(\frac{n_0}{n_0 - \rho_n} \right) (\rho_m + \rho_k) \right). \tag{22}$$

Note that the position of this curve, in particular with respect to Regions I and II of the positive octant, will change for different values of n_0. The case shown in Figure 7 assumes $0 < (n_0/(n_0 - \rho_n))\,(\rho_m + \rho_k) < \tau$, putting (22) below the point (τ, τ). Condition (20) prevents recursion in Region I and allows it in Region II independent of condition (18). Elsewhere recursion is prevented anywhere that (18) is true and allowed where (18) is false. For the plane shown in Figure 7, recursion is thus controlled solely by (18), which prevents or allows recursion in the area on or below the curve (21), and above this curve, respectively.

In Section 7.4 we will perform the measurements to establish τ, ρ_m, ρ_k, and ρ_n for various machines.

5.5. Program Structure

Algorithm FMM, shown in Figure 8, summarizes our implementation of Strassen's algorithm. There are four main steps performed at each level of recursion; for more details on each step see the indicated sections.

1. Check to see if the cutoff criterion is satisified; if so apply DGEMM (Sections 3 and 5.4); otherwise, continue with Step (2).
2. Peel off a row and/or column if any of the dimensions are odd. (Section 4)
3. Apply Strassen's algorithm via calls to STRASSEN1 or STRASSEN2 . (Sections 2 and 5.2)
4. Insert the contribution of the pieces that were peeled off in Step (2). (Sections 4 and 5.3)

$$\mathrm{FMM}(A, B, \alpha, \beta, C, W)$$

Inputs: A is an $m \times k$ matrix. B is an $k \times n$ matrix. α and β are scalars. W is a block of memory used for intermediate computations. If $\beta \neq 0$ then the size of W is at most $(mk + kn + mn)/3$. If $\beta = 0$ then the size of W is at most $(m \cdot \max(k, n) + kn)/3$.
Output: C is an $m \times n$ matrix. $C = \alpha AB + \beta C$.

(1) [Check cutoff criterion]
If cutoff criterion is satisfied, then call DGEMM.

(2) [Peel off row and column if odd]
$m_0 \leftarrow (m\%2)$;
$k_0 \leftarrow (k\%2)$;
$n_0 \leftarrow (n\%2)$;
$m_1 \leftarrow m - m_0$;
$k_1 \leftarrow k - k_0$;
$n_1 \leftarrow n - n_0$;
Let

$$A = \begin{pmatrix} A_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix};$$

$$B = \begin{pmatrix} B_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix};$$

where A_{11} is an $m_1 \times k_1$ matrix, a_{12} is an $m_1 \times k_0$ matrix, a_{21} is an $m_0 \times k_1$ matrix, a_{22} is an $m_0 \times k_0$ matrix, B_{11} is an $k_1 \times n_1$ matrix, b_{12} is an $k_1 \times n_0$ matrix, b_{21} is an $k_0 \times n_1$ matrix, and b_{22} is an $k_0 \times n_0$ matrix.

(3) [Call Strassen's algorithm.]
If $\beta = 0$ then
STRASSEN1$(A_{11}, B_{11}, \alpha; 0, C_{11})$
else
STRASSEN2$(A_{11}, B_{11}, \alpha, \beta; C_{11})$;

(4) [Fixup]

$$C_{11} = \alpha\,(a_{12}b_{21}) + C_{11} \qquad \text{(via DGER)};$$

$$c_{12} = \alpha \begin{pmatrix} A_{11} & a_{12} \end{pmatrix} \begin{pmatrix} b_{12} \\ b_{22} \end{pmatrix} + \beta c_{12} \qquad \text{(via DGEMV)};$$

$$\begin{pmatrix} c_{21} & c_{22} \end{pmatrix} = \alpha \begin{pmatrix} a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} B_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} + \beta \begin{pmatrix} c_{21} & c_{22} \end{pmatrix} \qquad \text{(via DGEMV)};$$

Figure 8: *Algorithm FMM.*

6. Further Model Refinements

Section 2 introduced three models to represent, with increasing precision, the computation costs associated with Strassen's algorithm. All of these were based on the two key primitive operations of standard matrix multiplication and matrix addition/subtraction. However, in Section 5 we have seen that the actual implementation involves a more complex set of basic computations. In this section we refine our models so as to be able to accurately represent the differing performance expected on this larger class of primitives.

We first deal with inputs having even dimensions. Examination of our computation schemes STRASSEN1 and STRASSEN2 in Figures 5 and 6, for the cases $\beta = 0$ and $\beta \neq 0$, respectively, reveals that there are four types of operations actually used on which preliminary timing data indicated possibly different execution speeds. These are (1) matrix multiplication (recursive calls), such as the computation R_1R_2, or scaled matrix multiplication, such as $\alpha A_{12}B_{21}$, (2) multiply-accumulate, with various scalings, such as $C_{11} + \alpha A_{12}B_{21}$ or $\beta C_{21} + A_{22}R_2$, (3) matrix add/subtract, such as $R_6 + R_4$ or $C_{21} + R_3$, or scaled matrix add/subtract, such as $\alpha(A_{21} + A_{22})$, and (4) matrix linear update (matrix add/subtract with only one scaled term), such as $\beta C_{12} + R_3$. We expect that (1) and (2) have similar timings, since both would be executed as calls to DGEMM. This leaves three basic operations: matrix multiplication and matrix add/subtract which were used in our original models, represented by $M(m,k,n)$, and $G(m,n)$, respectively, and matrix linear update, which we express as $L(m,n)$. When $\beta = 0$, STRASSEN1 incurs 13 add/subtracts, 2 linear updates, and the 7 recursive matrix multiplications, while STRASSEN2 in general incurs 9 add/subtracts, 5 linear updates, and the 7 multiplications. Separating these into calls with the same input sizes we obtain the following cost function for one level of recursion:

$$W(m,k,n) =$$

when $\beta = 0$

$$\begin{cases} M(m,k,n), & \text{when } (m,k,n) \text{ satisfies cutoff criterion} \\ 7W(\frac{m}{2},\frac{k}{2},\frac{n}{2}) + 2G(\frac{m}{2},\frac{k}{2}) + 2L(\frac{m}{2},\frac{k}{2}) + \\ \qquad 4G(\frac{k}{2},\frac{n}{2}) + 7G(\frac{m}{2},\frac{n}{2}), & \text{otherwise} \end{cases}$$

when $\beta \neq 0$

$$\begin{cases} M(m,k,n), & \text{when } (m,k,n) \text{ satisfies cutoff criterion} \\ 7W(\frac{m}{2},\frac{k}{2},\frac{n}{2}) + 2G(\frac{m}{2},\frac{k}{2}) + 2L(\frac{m}{2},\frac{k}{2}) + \\ \qquad 4G(\frac{k}{2},\frac{n}{2}) + 3G(\frac{m}{2},\frac{n}{2}) + 3L(\frac{m}{2},\frac{n}{2}), & \text{otherwise} \end{cases}$$

As shown in Table 7, subsequent recursions will have different values for β. However, we base our modeling on the value for β at the top level call. We will see in Section 7 that the model produced by this simplifying assumption is highly accurate, so that a more complex analysis proved unnecessary. Thus, if A and B are of size $2^d m' \times 2^d k'$ and $2^d k' \times 2^d n'$, respectively, with cutoff arbitrarily set at size $m' \times k'$ for subblocks of A and size $k' \times n'$ for subblocks of B, the cost of Strassen's algorithm in terms of M, G and L is

$$\begin{aligned}
&\text{when } \beta = 0 \\
W(2^d m', 2^d k', 2^d n') \;&=\; 7^d M(m', k', n') \;+ \\
&\quad 2\sum_{i=0}^{d-1} 7^i G(2^{d-i-1}m', 2^{d-i-1}k') \;+ \\
&\quad 2\sum_{i=0}^{d-1} 7^i L(2^{d-i-1}m', 2^{d-i-1}k') \;+ \\
&\quad 4\sum_{i=0}^{d-1} 7^i G(2^{d-i-1}k', 2^{d-i-1}n') \;+ \\
&\quad 7\sum_{i=0}^{d-1} 7^i G(2^{d-i-1}m', 2^{d-i-1}n')
\end{aligned} \tag{23}$$

$$\begin{aligned}
&\text{when } \beta \neq 0 \\
W(2^d m', 2^d k', 2^d n') \;&=\; 7^d M(m', k', n') \;+ \\
&\quad 2\sum_{i=0}^{d-1} 7^i G(2^{d-i-1}m', 2^{d-i-1}k') \;+ \\
&\quad 2\sum_{i=0}^{d-1} 7^i L(2^{d-i-1}m', 2^{d-i-1}k') \;+ \\
&\quad 4\sum_{i=0}^{d-1} 7^i G(2^{d-i-1}k', 2^{d-i-1}n') \;+ \\
&\quad 3\sum_{i=0}^{d-1} 7^i G(2^{d-i-1}m', 2^{d-i-1}n') \;+ \\
&\quad 3\sum_{i=0}^{d-1} 7^i L(2^{d-i-1}m', 2^{d-i-1}n').
\end{aligned}$$

As before, we can substitute various expressions for M, G, and now L, to obtain closed form expressions for $W(2^d m', 2^d k', 2^d n')$. To do this for our full polynomial model, we would use (2) and (3), and let $L(m,n)$ be expressed as

$$L(m,n) \;=\; \lambda_3 mn + \lambda_2 m + \lambda_1 n + \lambda_0.$$

Substitution into (23) would yield results similar to (5). For the square case it is simpler to start over with new expressions for M, G, and L requiring fewer coefficients. Thus, we let

$$M(m) \;=\; \mu'_3 m^3 + \mu'_2 m^2 + \mu'_1 m + \mu'_0, \tag{24}$$

$$G(m) \;=\; \gamma'_2 m^2 + \gamma'_1 m + \gamma'_0, \tag{25}$$

and

$$L(m) \equiv L(m,m) = \lambda_2' m^2 + \lambda_1' m + \lambda_0'. \tag{26}$$

Substitution into (23) now gives

$$\begin{aligned} W(2^d m') &= 7^d(\mu_3'(m')^3 + \mu_2'(m')^2 + \mu_1'(m') + \mu_0') + \\ &\quad (7^d - 4^d)(g_\beta \gamma_2' + l_\beta \lambda_2')(m')^2/3 + \\ &\quad (7^d - 2^d)(g_\beta \gamma_1' + l_\beta \lambda_1')(m')/5 + \\ &\quad (7^d - 1)(g_\beta \gamma_0' + l_\beta \lambda_0')/6 \end{aligned} \tag{27}$$

where

$$g_\beta = \begin{cases} 13, & \text{if } \beta = 0; \\ 9, & \text{if } \beta \neq 0 \end{cases} \quad \text{and} \quad l_\beta = \begin{cases} 2, & \text{if } \beta = 0; \\ 5, & \text{if } \beta \neq 0. \end{cases}$$

To obtain a fully general model we must extend this to odd-sized input dimensions and account for fixup costs. We do this by adding costs for the DGEMV and DGER calls used to perform the fixup operations. Since the calls to DGEMV are one matrix-vector product and one vector-matrix product, one call involves a transposed input matrix and one does not. The different memory access patterns of these two calls leads to different performance, so we model them separately. All three of these operations are quadratic, so for the square case in the full polynomial model we can let

$$R(m) = \sigma_2' m^2 + \sigma_1' m + \sigma_0', \tag{28}$$

$$V(m) = \delta_2' m^2 + \delta_1' m + \delta_0', \tag{29}$$

and

$$VT(m) = \epsilon_2' m^2 + \epsilon_1' m + \epsilon_0' \tag{30}$$

represent costs for DGER, DGEMV with no transpose, and DGEMV with transpose, respectively.

Obtaining expressions such as (27) would prove more difficult in this general case since the recursive patterns would not be uniform. However, we can still measure computing times for the six primitives M, G, L, R, V, and VT, and use a least squares fit to empirically determine the coefficients of their respective polynomials. An estimate of runtime cost of our Strassen implementation can then be obtained via the pseudo-code shown in Figure 9. Measurements of this type will be done in Section 7.5.

```
time = 0
factor = 1
while (m > cutoff)
  m_half = ⌊m/2⌋  !  integer division with truncation
  m_even = m_half × 2
  time = time + factor × (g_β G(m_even) + l_β L(m_even))
        !  add and linear update time
  if (m ≠ m_even) then
    time = time + factor × (R(m_even) + V(m) + VT(m))  !  fixup time
  end if
  factor = factor × 7
  m = m_half
end
time = time + factor × M(m)  !  matrix multiplication time
```

Figure 9: *Pseudo-code for modeling costs of DGEFMM on square inputs*

7. Performance

This section discusses the results of running and analyzing Strassen's matrix multiplication on a variety of computers. We first describe the machines used and how the results were obtained. Next, the polynomial fits for the computational primitives discussed in Section 6 are applied to each machine. These fits are then used to create a model of the complete running time of our Strassen implementation. This is followed by a general analysis of the costs of the primitive operations to show which ones are significant. Empirical measurements are then used to set the cutoff conditions for both square and rectangular inputs, and our new criterion is compared to other methods. On each machine the square case empirical result is compared to model predictions. We then examine the performance of our new routine. The full polynomial model is shown to be accurate in predicting actual run times on each machine, and we present a comparison 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.

7.1. Machines and Testing

Though our code is portable and has been run on many machines, this paper will focus on three machines: IBM RS/6000, Cray YMP C90, and Cray T3D (single processor). These three machines were chosen because they have a wide range of characteristics which show how Strassen's matrix multiplication is affected. This allows us to demonstrate the robustness of our models and analyze the differences across machines. This should make understanding the potential performance of our Strassen code on alternative machines fairly easy. Furthermore, highly-tuned vendor implementations of Strassen's algorithm exist on the RS/6000 and C90, thus allowing us to demonstrate the absolute performance of our routine.

The exact model and OS of the three machines is described in Table 8. It is important to note that we are only presenting serial results. The T3D numbers are from running on a single processor. 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. On the RS/6000 the IBM-supplied BLAS library, *libblas.a*, 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). 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*, was utilized. On the C90, the Cray implementation of Strassen's algorithm, SGEMMS, was used for comparison. 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.

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. In particular, there are no matrix addition or subtraction routines contained in the BLAS library. One needs first to do

Machine	OS	C	FORTRAN	BLAS
IBM RS/6000	AIX 3.25	1.12	1.1.2.1	libblas.a (v2.2)
Cray YMP C90	UNICOS 9.0.1.2	4.0.2.7	CF77 ver 6.0.4.0	scilib.a (v2.0)
Cray T3D	UNICOS MAX 1.3.0.0	4.0.4	CF77_M ver 6.0.4.1	scilib.a (v2.0)

Table 8: *Machine characteristics.*

a DCOPY, then a DAXPY operation. We found that generally a two-loop FORTRAN routine was more efficient. On the RS/6000 and C90, the FORTRAN matrix addition and subtraction routine was the fastest. (Although the IBM ESSL contains a matrix addition and subtraction routine, we chose not to use it since we did not notice any appreciable gain, and it would have required users of our package to purchase that library separately.)

To perform the linear update operation we created a new linear update routine and used a FORTRAN version. Though all of the operations performing linear combinations of matrices could be incorporated into a single routine, we have opted to keep matrix add/subtract and linear update separate for two reasons. First, it breaks up the operations into more traditional Strassen categories, and secondly, the performance of these two operations is generally different, so that keeping them separate enhances the modeling effort.

Timing was accomplished by starting a CPU 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.

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/`.

7.2. Determining Model Coefficients

As described in Section 6, to model our Strassen implementation on a given architecture requires formulas for the cost of the six primitive operations as a function of matrix size. Recall that these operations are matrix multiplication, matrix add/subtract, matrix linear update, DGER, DGEMV with no transpose, which we denote as DGEMV_n, and DGEMV with transpose, denoted DGEMV_t.

Our goals in modeling our Strassen implementation are to allow us to show how one can predict performance from primitive costs and to allow analysis of the algorithm without requiring actual runs of the code. Both of these goals can be met by restricting our analysis to square matrices. Thus, we wish to use a least squares fit to determine the coefficients of the polynomials in (24) through (26) and (28) through (30). This allows us to perform simple fits to a single variable.

Trying to accurately measure the costs of these primitives leads to a number of issues. For example, the absolute times for small matrices can be quite small. As a result, we average over several calls to get more stable results. However, repeatedly calling the routine with the same parameters raises the issue of cache reuse. In some parts of the full code we get cache reuse between calls and in others we do not. As a compromise, we flush the cache before we start the loop for averaging but not during each iteration of the loop. We have also noticed on some machines that the first call to a routine or the first time dynamically allocated memory is touched can be more expensive than future uses. To avoid this problem we call all routines before timing and touch all array elements before their first use. Those interested in the exact details of our timing methods should contact one of the authors.

To perform the polynomial fits, we set up a Vandermonde system $Vc = t$, where V is a Vandermonde system of data points, c is the unknown coefficient vector, and t is a vector of empirical computing times. The system will be overdetermined and a least squares solution computed to account for the error in the timing measurements. For matrix multiplication we obtain the system

$$\begin{pmatrix} m_1^3 & m_1^2 & m_1 & 1 \\ m_2^3 & m_2^2 & m_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ m_r^3 & m_r^2 & m_r & 1 \end{pmatrix} \begin{pmatrix} \mu_3' \\ \mu_2' \\ \mu_1' \\ \mu_0' \end{pmatrix} = \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_r \end{pmatrix}$$

where M_i is the empirical time for a matrix of order m_i. One question is what range of m_i should be used? Clearly, one wants to use the sizes of matrices that are important for the modeling. However, since one goal of our fits is to allow someone to predict the square cutoff and performance of Strassen's algorithm on a new machine, one should not use a priori information in determining the range of values used. We chose to use the range $m_i = 12, \cdots, 500$. We use 12 as the lower limit since it is the theoretical square cutoff based on operation count. The actual cutoff has always been much larger than this lower limit value so this gives a range of points below the actual cutoff. We have found that using too large a value can lead to errors in modeling our Strassen code. This is due to lower order terms being important for small values of m and needed in the models as described later. The upper limit of 500 was used since it was large but still allowed one to acquire the needed data quickly. Unlike the lower limit, the exact value of the upper limit is not as important. The only criteria is that it is large enough to be much larger than the square cutoff. The reason that changing it does not effect the results greatly is that for large m one is seeing the asymptotic behavior and any reasonably large range of m will get a good fit.

An example of the fit obtained for the matrix add/subtract primitive on the RS/6000 is shown in Figure 10. The plot shows the normalized times as a function of the matrix dimension. The normalized times are obtained by taking the time in seconds and dividing by m^2. This is done because the times for larger matrices are significantly larger than smaller matrices. There are several things to note. First, the fit, excluding the anomalous points, is very good. Second, the fit for small matrix dimensions is quite poor. This is due to the fact that the actual run times are not a smooth quadratic polynomial. However, the poor fit in this region will not hurt our models because it is for matrix sizes outside the range needed. Third, note that there are a handful of empirical points that have times that are significantly different than nearby points. These points are true anomalies and can be readily reproduced. Though we have not investigated the reason for their existence in detail, we feel they are due to cache effects as has been studied previously [21]. Since our primary focus in this paper is on the general tradeoffs in Strassen's algorithm, which depend on the average behavior and not anomalous behavior that varies by machine, these points are not of great concern.

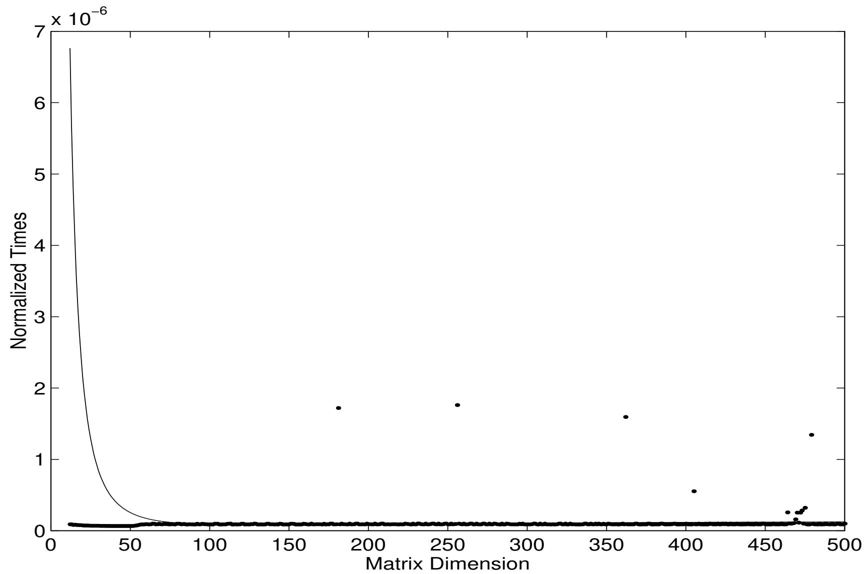

Figure 10: *Comparison of actual run times to polynomial fit for matrix add/subtract on the RS/6000 for* $\alpha = 1$ *and* $\beta = 0$. *The dots are the actual run times and the solid line is the polynomial fit. The times have been normalized by* m^2.

An issue related to the anomalous points is the layout in memory of the matrices used in our code. Because of the nature of Strassen's algorithm, the majority of the work in this implementation occurs on matrices which are of even dimension. On some memory systems this can lead to potential performance degradations. For example, on a computer with b banks of memory, every b^{th} element of a matrix is stored in the same bank. Rapidly doing memory operations to the same bank can significantly slow memory accesses. Since b is generally a power of two, stepping across the columns of a matrix which is even will not hit all the banks. As a result, there can be a performance degradation. In the worst case, the matrix size is a multiple of b. To avoid this potential problem, we experimented with storing all even-sized matrices with r rows in an array allocated for $r + 1$ rows. Since $r + 1$ is odd, it must be relatively prime to b, which is a power of two. In the end we opted not to use this option for several reasons. First, it is more complicated to compute the minimum amount of temporary memory needed, since matrices encountered during the recursion may be odd or even. Second, the performance gain was negligible except in extreme cases, i.e., the given matrix had a power-of-two number of rows. This is because the code is carefully designed to always process in column-major order whenever possible. Consequently, the code rarely strides across columns. This is consistent with the fact that before we optimized our memory usage and access, we found this modification to have a greater impact on performance. Users who have applications where the matrices are often powers of two, may wish to store those matrices in arrays with an odd leading dimension. This will achieve the same advantage as if done within our code and may also improve memory access within the user's code.

Tables 9 and 10 show the coefficients of the polynomial fits on the RS/6000 for both types of α and β. These fits show that the different $O(m^2)$ operations take different amounts of time. For example, Table 9 for the RS/6000, shows that the linear update is typically 50% faster than add/subtract. Also, the fixup costs are faster than matrix add/subtract. However, these results are very machine specific. For example, the results on the C90 show that fixup is faster than add/subtract, but linear update is the most expensive. On the T3D, linear update and fixup are significantly cheaper than add/subtract. Comparing Table 9 to Table 10 shows that results also depend on whether α and β are 1 and 0 or more general values. Times are independent of the specific value of α and β once they are not the special values of 1 and 0. It was a surprise on the RS/6000 that the timings for general α and β are faster than the special values of 1 and 0 for the primitives of add/subtract and linear update. On the C90 they are similar except for linear update and on the T3D the general case is significantly slower. These results show why it is necessary to model each primitive component in Strassen. Not doing this leads to errors in predicting the actual behavior of the Strassen code.

Figure 10 shows that there are differences between the polynomial fits and the actual data. Table 11 shows an estimate of the quality of the fit for the plot shown in Figure 10. To represent the errors, we define the difference for each point to be

$$\Delta_i = 100 \times (P_i - D_i)/D_i \tag{31}$$

where P_i is the value from the polynomial fit at size m_i and D_i is the measured value at m_i. As Figure 10 demonstrated, there are measured data that is anomalous. To show these points, we give

$$\text{max} = max_i \Delta_i$$

$$\text{min} = min_i \Delta_i$$

where i ranges from 89, 53, and 151 for the RS/6000, C90, and T3D, respectively, and goes to 500. We exclude the small data points from the quality of fit analysis since they yielded a poor fit in general and they are never used in the models. The low value chosen for each machine is 10% less than half the square cutoff (which is discussed in Section 7.4) since this covers the smallest values ever used in the models. Also note that since the measured data is "correct", we divide by D_i. This causes Δ_i to be near -100 when $D_i \gg P_i$ which is why, for example, the minimum add/subtract error is -94.8 even though there are measured points which are an order of magnitude larger than the polynomial fit. To show the typical error, we define

$$\text{ave} = \frac{1}{k} \sum_i \Delta_i$$

where k is the number of data points (range of i). The average can be somewhat deceptive if Δ_i is both positive and negative, so we also give

$$\text{absolute average} = \frac{1}{k} \sum_i |\Delta_i| \ .$$

Note that for add/subtract in Table 11, the average is 11.9 but the absolute average is 18.6. This is consistent with Figure 10 where the polynomial fit is generally greater than the actual values.

Primitive Operation	Coefficients cubic	of quadratic	Polynomial linear	constant
matrix multiplication	1.76e-08	6.60e-08	1.12e-05	-4.94e-04
matrix add/subtract	–	1.57e-07	-1.86e-05	1.17e-03
matrix linear update	–	9.33e-08	-1.16e-05	7.53e-04
DGER	–	5.99e-08	-4.41e-06	6.93e-04
DGEMV_t	–	3.50e-08	-7.64e-07	1.68e-04
DGEMV_n	–	4.50e-08	-6.74e-06	8.15e-04

Table 9: *Coefficients of polynomial fits of primitive operations in our Strassen code on the RS/6000 for* $\alpha = 1$ *and* $\beta = 0$.

Primitive Operation	Coefficients cubic	of quadratic	Polynomial linear	constant
matrix multiplication	1.75e-08	1.33e-07	-2.62e-06	2.18e-4
matrix add/subtract	–	1.16e-07	-3.21e-06	1.95e-04
matrix linear update	–	8.09e-08	1.77e-07	-3.73e-05
DGER	–	5.93e-08	-3.50e-06	4.66e-04
DGEMV_t	–	4.24e-08	-4.98e-06	6.99e-04
DGEMV_n	–	4.52e-08	-7.00e-06	8.74e-04

Table 10: *Coefficients of polynomial fits of primitive operations in our Strassen code on the RS/6000 for* $\alpha = 1/3$ *and* $\beta = 1/4$.

Looking at the minimum and maximum for the other machines shows that anomalous points exist on all three machines. The average error on all the machines is typically less than 1% and the absolute average is typically less than 7%.

7.3. Behavior of Primitive Operations

The polynomial fits from the previous section can be used to examine the relative time per operation of the primitive parts. Comparing matrix multiplication to add/subtract shows that the former is 9 times cheaper per operation at cutoff/2 ($m = 100$) and is asymptotically (large m) over 17 times cheaper on the RS/6000 for $\alpha = 1$ and $\beta = 0$. The ratio does not change greatly for other α and β. In Strassen's algorithm, the matrix multiplication only occurs for matrices of size between cutoff/2 and cutoff. Matrix add/subtract occurs for many sizes of matrices. Since the matrix multiplication is on smaller matrices then the matrix adds/subtracts, one might expect that the cost per operation would be better than 9 to 17 which is for matrices of the same size. However, the fits show that matrix multiplication of matrices of order cutoff/2 is within 7% of the asymptotic cost per operation so the ratio is not changed significantly. Thus, the ratio of 9 to 17 between matrix multiplication and add/subtraction is significant in this case and it is overlooked when using the operation count model. For the T3D the ratio ranges from

Primitive Operation	max	min	average	abs ave
matrix multiplication	2.20	-17.17	0.13	0.55
matrix add/subtract	39.91	-94.78	11.99	18.62
matrix linear update	21.10	-69.10	-3.40	13.04
DGER	48.41	-56.44	0.53	4.90
DGEMV_t	16.58	-85.06	1.47	2.90
DGEMV_n	67.30	-82.37	-0.47	8.13

Table 11: *Error estimate of polynomial fits of primitive operations in our Strassen code on the RS/6000 for $\alpha = 1$ and $\beta = 0$.*

21 to 22 and the C90 has a ratio of 4.4 to 2.9.

The relative cost of matrix multiplication to add/subtract is an important factor in determining the optimal use of Strassen's algorithm. When Strassen's algorithm was first introduced, the CPU cost of adding two elements was typically much less than multiplying the same two elements. Changes in computers have mitigated this situation. First, the number of cycles to perform a multiplication has been reduced due to the availability of greater logic on the chip. Second, the pipelining of functional units means that if a group of multiplies is performed (often a vector), the cost per multiply approaches a single cycle. These types of effects have changed the situation to where the number of cycles for an add/subtract and a multiplication are often comparable.

The CPU costs must be combined with the fact that neither matrix multiplication nor add/subtract performs a single add/subtract but rather works on matrices. When working on matrices, it is often the memory bandwidth to a processor that determines the efficiency of the operation being performed. To see the expected memory bandwidth requirements, typical code for performing a matrix add/subtract is shown in Figure 11 and matrix multiplication in Figure 12. It is easily seen that matrix add/subtract requires 2 loads, 1 store, and 1 add/subtract for each execution of the innermost loop. (Here a load refers to bringing in a value from main memory. It is assumed that values are not in cache when the computation begins.) Matrix multiplication, on the other hand, requires only 1 load, 1 add/subtract, and 1 multiplication in the limit of large m. (Since the typical cutoff for an implementation of Strassen has $m \gg 1$, this is a good assumption.) This is because the j^{th} column of c[i, j] can be reused for many values of k. If this is done intelligently, the necessary values can be kept in cache which significantly reduces the memory cost. In addition, temp is reused for all m values in the inner loop.

Comparing matrix add/subtract to matrix multiplication shows that the former requires 3 memory references as compared to 1 for the latter. If memory costs dominate the total cost of the loops (which is true on most microprocessors), then the add/subtract will take 3 times longer per inner loop execution. In the operation counts given previously, each inner loop execution does 1 operation for matrix add/subtract but 2 for matrix multiplication. Taking these together, the operation counts would have to have the matrix add/subtract operations weighted by a factor of 6 to make them represent these costs. Furthermore, on many systems, the vendor supplies high quality BLAS routines which covers matrix multiplication but not matrix add/subtract. As a

```
j = 1:m
  i = 1:m
    c[i,j] = a[i,j] + b[i,j]
  endloop
endloop
```

Figure 11: *Pseudo-code of typical matrix add/subtract algorithm.*

```
j = 1:m
  k = i:m
    temp = b[k,j]
    i = 1:m
      c[i,j] = c[i,j] + temp * a[i,k]
    endloop
  endloop
endloop
```

Figure 12: *Pseudo-code of typical matrix multiplication algorithm.*

result, the matrix multiplication is even faster than this when compared to matrix add/subtract.

This analysis leads to the conclusion that it is usually substantially more expensive per operation to perform a matrix add/subtract than a matrix multiplication. As stated above, the ratio is 9 to 17 on the RS/6000 and 21 to 22 on the T3D. These ratios are consistent with the discussion above. The C90 has a lower ratio of 4.4 to 2.9. This lower ratio is expected since the C90 has multiple memory ports so it can do simultaneous loads and stores. It is important to note that this is in contrast to the expected situation, stated above, when Strassen's algorithm was first introduced. This is one reason why the reconsideration of Strassen's algorithm based on modern computer architectures is important.

7.4. Measuring and Modeling the Cutoff

As described in Section 5.4, empirical testing will determine the actual cutoff criteria to be used in practice. We report on this testing first for square and then rectangular matrices. We also compare empirical results to predictions of our square-case model developed in Sections 6 and 7.2.

7.4.1. Square Matrices

Recall from Section 5.4 that determining the square cutoff is simply a matter of determining the crossover point where doing one level of Strassen becomes faster than calling DGEMM. Figure 13 shows the ratio of DGEMM to our DGEFMM code with one level of recursion as a function of square matrix order for the RS/6000. When this ratio is greater than 1, using

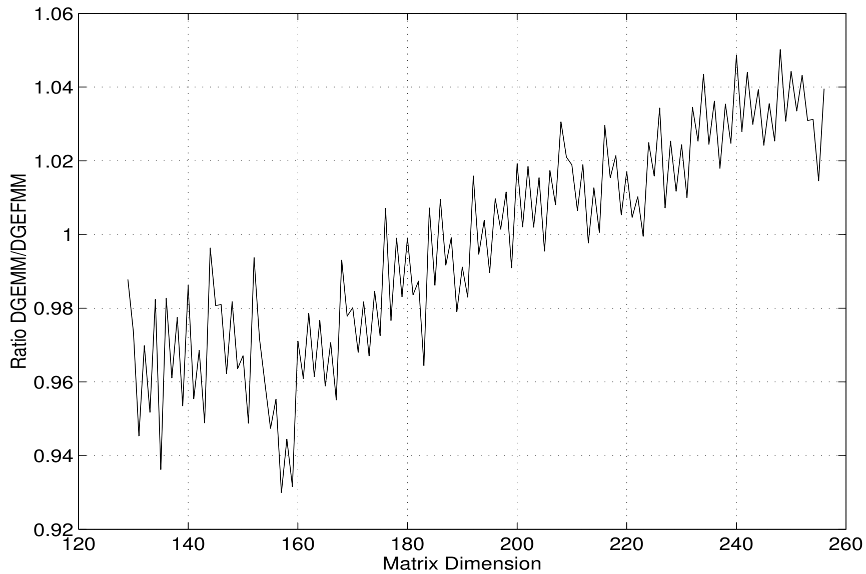

Figure 13: *Experimentally determined square cutoff for RS/6000 with* $\alpha = 1$ *and* $\beta = 0$.

Strassen is a win. For our implementation, Strassen is first a win for $n = 176$ and always a win if $n \geq 214$. The range is caused by the fact that the primitive costs for the general case are not monotonic, with the varying requirements for fixup work on odd-sized inputs creating the saw-toothed look of the graph. However, the range is fairly small and the performance difference between DGEFMM 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 doing a Strassen recursion is almost always better than DGEMM for larger values and when it is slower it is so by a very small amount. This technique can be applied to each machine to experimentally determine its cutoff value for square matrices. The results, noted in the second column of Table 12, show the size at which our Strassen implementation becomes more efficient for square matrices. The range of cutoffs observed is one reason we chose to report on these three machines in this paper. Also note 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.

Cutoffs for the square case can also be estimated from the model of our Strassen code presented in Sections 6 and 7.2. If the general model (with fixup costs) is used, a simple closed form result cannot be easily found. The fixup costs increase the cost of our implementation for odd-sized matrices. This is seen by the lower ratios for odd-sized matrices in the saw-toothed plot in Figure 13. Removing the fixup cost from consideration would lower the cutoff determined. However, we will see in Section 7.6 that the fixup cost is never too high. As a result, we expect a model that ignores fixup costs to yield good results and we have found that this is the case.

As described in Section 3, we need to determine the smallest input size for which one more

Machine	Empirical Square Cutoff τ	Modeled		
		Full		High order
		$\alpha = 1,\ \beta = 0$	$\alpha = 1/3,\ \beta = 1/4$	$\alpha = 1,\ \beta = 0$
RS/6000	199	196	198	255
C90	129	136	169 & 60	44
T3D	325	304	418	296

Table 12: *Table of empirical and modeled square cutoff for various machines.*

Strassen recursion is better than applying DGEMM. In the full polynomial model for the square case developed in Section 6 we need to find the smallest m such that $M(m) > W(m)$. From (24) and (27) this produces the equation

$$\begin{aligned} M(m) - W(m) &= M(m) - W(2^1 \cdot m/2) \qquad (32) \\ &= \mu_3' m^3/8 - (3\mu_2' + g_\beta \gamma_2' + l_\beta \lambda_2')m^2/4 \ - \\ &\qquad (5\mu_1' + g_\beta \gamma_1' + l_\beta \lambda_1')m/2 - (6\mu_0' + g_\beta \gamma_0' + l_\beta \lambda_0') \\ &= 0, \end{aligned}$$

which can easily be solved. The real, positive root(s) yields a square cutoff given by input size m. The solutions for all three machines are given in the third column of Table 12 when $\alpha = 1$ and $\beta = 0$. The empirical cutoff and the model predictions agree within 7% in all cases. Thus, one can accurately model the square cutoff from the full polynomial fits of the cost of the primitive operations.

We have already seen in Section 7.2 that the costs of the primitive operations changes when general α and β are used instead of 1 and 0. Thus, one might expect that the cutoff would also depend on these values. However, we have observed empirically that the cutoff is independent of the value of α and β. Column four of Table 12 shows that the result of using (32) with the polynomial fits for $\alpha = 1/3$ and $\beta = 1/4$ yields different cutoffs for the C90 and T3D. This is caused by very modest errors in the fits. For example, on the C90, the polynomial fit of matrix multiplication overestimates the cost for matrices of size near (cutoff/2) but underestimates the cost at cutoff. Estimating the cost of DGEFMM depends on the former whereas DGEMM depends on the latter. As a result, the cost of DGEFMM is overestimated and DGEMM is underestimated. The errors in each fit are only 5 to 8%, which means the fits are fairly accurate. However, estimating the cutoff is a difference of the two, thereby doubling the error in this case. Changing the relative cost by 10% causes the cutoff at 136 to change to 169. There is also a second root of (32) at 60. This one occurs because of problems in fitting the empirical data for small matrix sizes. Here, the matrix add/subtract and multiplication errors are different and this causes a second solution to the cutoff equation. A similar analysis could be carried out on the T3D results. The above analysis shows that even modest errors in the polynomial fits can cause substantially larger errors in estimating the cutoff. This cannot be avoided due to the nature of the equations involved.

If only the higher-order terms are included, then the cutoff is determined by the relative MFLOP rate of matrix multiplication to matrix add/subtract and linear update. This model

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

yields

$$M(m) - W(m) \;=\; \mu_3' m^3/8 - (g_\beta \gamma_2' + l_\beta \lambda_2')m^2/4 \;=\; 0,$$

which implies

$$m = 2(g_\beta \gamma_2' + l_\beta \lambda_2')/\mu_3'.$$

The results from this model for $\alpha = 1$ and $\beta = 0$ are given in column five of Table 12. There is no longer a quantitative fit between these estimates of the cutoff and empirical results. The qualitative trends, however, are correct. Thus, the simple MFLOP model can give one a rough idea of the cutoff and efficiency of DGEFMM but cannot provide quantitative results.

This result shows the need to accurately model primitive operations in a Strassen implementation to get quantitative results. Analyzing the cause of the errors in the high-order term model shows that the errors are due to poor fit for smaller values of matrix sizes. It is for smaller matrices that overheads are substantially increased and the asymptotic MFLOP rate is no longer valid. It is these overheads that affect where the actual cutoff occurs. This can also be seen from using the relative cost of matrix add/subtract to multiplication found in Section 7.3. When smaller size matrices are used, the ratio is 7, 21, and 4.4 on the RS/6000, T3D, and C90, respectively. (Compared to 17, 22, and 2.9 for the asymptotic values given previously.) The changing ratios in the empirically-determined costs track the cutoff much more accurately than the high-order model which has a fixed ratio. This shows that the relative cost of matrix add/subtracts to multiplications is important in determining the actual cutoff.

7.4.2. Rectangular Matrices

As was discussed in Sections 3 and 5.4, determining the appropriate cutoff condition for rectangular matrices is much more complicated. The benefit of enhancing the cutoff criteria for rectangular matrices can be seen by noting that use of criterion (16) 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 13 summarizes, for each machine, the values obtained for the parameters ρ_m, ρ_k, and ρ_n, which describe the rectangular condition (18). These were computed using the method described in Section 5.4. 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. 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.

Machine	Comparison Ratio	Random Sample Size	Range	Quartiles	Average
RS/6000	(20)/(16)	100	0.9128–1.0169	0.9403;0.9566;0.9634	0.9529
	(20)/(17)	1000	0.9083–1.1286	0.9863;1.0038;1.0217	1.0017
	(20)/(17), two dims large	100	0.9126–1.0462	0.9688;0.9938;1.0052	0.9888
C90	(20)/(16)	100	0.8275–1.0294	0.9137;0.9488;0.9682	0.9375
	(20)/(17)	1000	0.7803–1.1095	0.9198;0.9457;0.9769	0.9428
	(20)/(17), two dims large	100	0.7908–0.9859	0.9024;0.9189;0.9454	0.9098
T3D	(20)/(16)	100	0.7957–1.1026	0.9335;0.9508;0.9667	0.9518
	(20)/(17)	1000	0.6868–1.1393	0.9528;0.9792;0.9996	0.9777
	(20)/(17), two dims large	100	0.9120–1.0328	0.9200;0.9262;0.9474	0.9340

Table 14: *Comparison of Cutoff Criteria for $\alpha = 1$ and $\beta = 0$. (16) = Simple Criterion from Square Cutoff, (17) = Theoretically Scaled Criterion, (20) = New Rectangular Criterion.*

The data in Table 13 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 14 summarizes a series of experiments comparing the various rectangular cutoff criteria described in Section 5.4. 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 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 enjoys better overall performance.

Entries in Table 14 are given as ratios of DGEFMM timings using our new criterion (20) 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

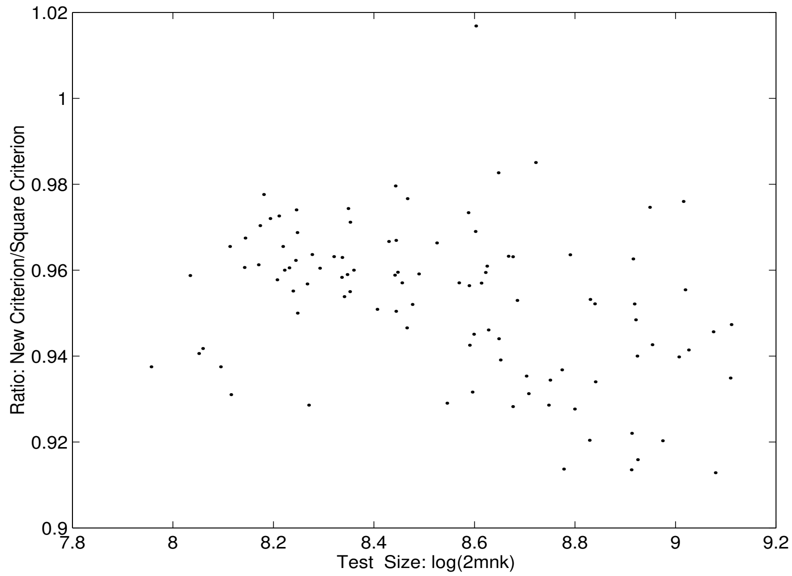

Figure 14: *Ratio of Time for New Rectangular Criterion to Square Criterion as a function of matrix size on the RS/6000 for* $\alpha = 1$ *and* $\beta = 0$.

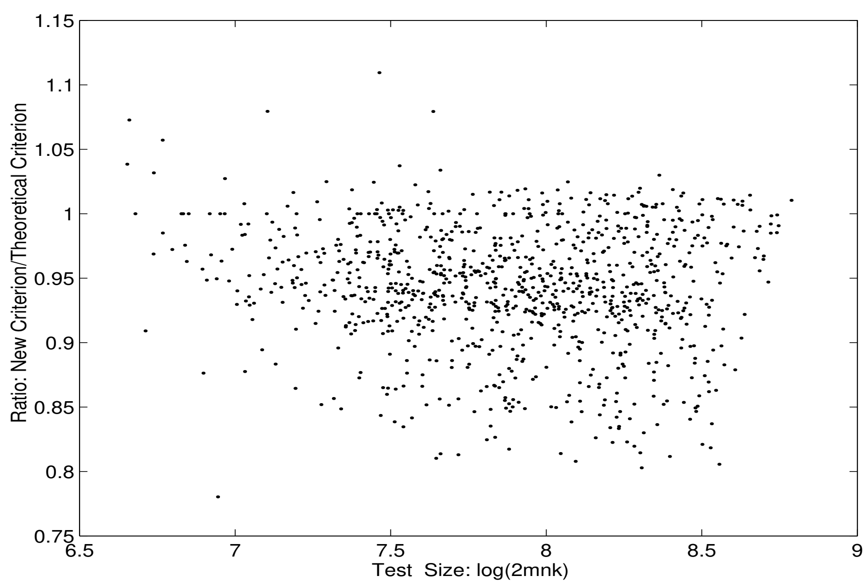

Figure 15: *Ratio of Time for New Rectangular Criterion to Theoretical Criterion as a function of matrix size on the CRAY C90 for* $\alpha = 1$ *and* $\beta = 0$.

compare (20) to (16), (20) to (17) on general input sizes, and (20) to (17) on problems where two of the three matrix dimensions are large. We define large to be at least 1800 for the RS/6000 and CRAY C90, and 1350 for the CRAY T3D. We also show in Figure 14 a plot of the complete set of ratios where we are comparing (20) to (16) on the RS/6000. Figure 15 is a similar plot on the CRAY C90 comparing (20) to (17) on general input sizes. In these figures $\log(x)$ denotes the logarithm base 10 of x.

These results 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, on average, 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 5.4, the values for ρ_m, ρ_k, and ρ_n could be even more finely tuned, and, for instance, be used in application areas where performance on matrices with large aspect ratios is critical.

7.5. Modeling Runtimes

The model given by the pseudo-code in Figure 9 can be used to readily estimate the execution time of our Strassen code. Figure 16 shows the relative error, defined in analogy to (31), between this model and the actual running times for the RS/6000 when $\alpha = 1$ and $\beta = 0$. This graph shows that the model is very accurate. The larger errors are due to anomalous times for our Strassen code (as was seen in the primitive parts times in Figure 10) and cannot be accurately modeled. Figure 17 shows the same result when $\alpha = 1/3$ and $\beta = 1/4$. Here the error is not as close to zero and the scatter is larger. This increase in error is expected, as discussed in Section 6, since the model does not take into account that recursive calls exist with different α and β. However, the overall fit is still very good.

This shows that our model, which uses the full polynomial fits and accounts for fixup costs, is very accurate on the RS/6000. Table 15 shows the quality of the model on all three machines and for both types of α and β. The statistics used are the same as those fits for the primitive parts. The table shows that the fits are quite good on all three machines. The typical errors, seen by average and absolute average, are generally better for $\alpha = 1$ and $\beta = 0$. The exception is the T3D and this is due to the fact that the $\alpha = 1$ and $\beta = 0$ fit is too low whereas the general case is too high. The average error is thus smaller for the general case. Overall, we can accurately model our Strassen code on all machines and the errors are typically only a few percent.

7.6. Using the Model for Further Analysis

The quality of the model of our Strassen code allows us to use it to analyze the underlying algorithm. Using the model instead of actual run times has several advantages. First, the model shows the normal behavior without the anomalous points seen in the empirical data. Another is that the speed of modeling allows one to quickly analyze the algorithm over many possible parameters including matrix sizes which cannot be readily run today.

Simple analysis of Strassen's algorithm typically considers the asymptotic speedups that can be achieved. In this analysis, the cost of matrix add/subtract is assumed to be small. Furthermore, the cost of dealing with the odd-sized matrices is not considered. In Figure 18, the percentage of time spent on the primitive operations is given based on the full model. The

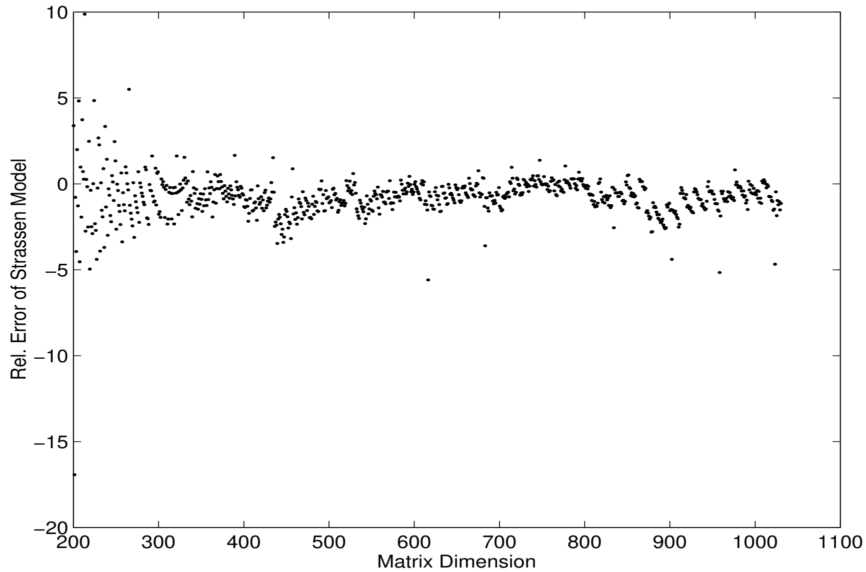

Figure 16: *Relative error between our model and the actual running times of our code on the RS/6000 for* $\alpha = 1$ *and* $\beta = 0$.

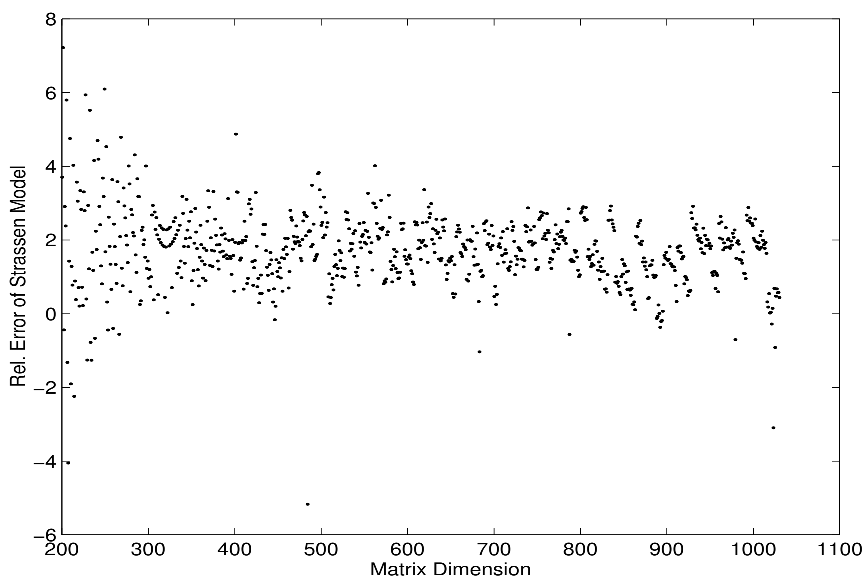

Figure 17: *Relative error between our model and the actual running times of our code on the RS/6000 for* $\alpha = 1/3$ *and* $\beta = 1/4$.

Machine	$\alpha/beta$	max	min	average	abs ave
RS6000	1/0	9.96	-16.85	-0.73	0.97
RS6000	0.33/0.25	7.16	-5.17	1.66	1.73
C90	1/0	4.33	-19.8	0.0125	1.37
C90	0.33/0.25	11.5	-15.1	5.84	6.01
T3D	1/0	4.06	-17.7	-5.20	5.26
T3D	0.33/0.25	10.4	-12.7	1.29	2.13

Table 15: *Error estimate of model of our Strassen code on all three machines and different* α *and* β.

operations are grouped by matrix multiplication; matrix addition, subtraction, and linear update (addition); and, fixup costs.

Beginning by overlooking the detailed structure in the figure, one can see several features. One trend is that the time spent on matrix multiplication decreases with increasing matrix size whereas the time for the other two increases. In the extreme case, addition takes 18.4% of the total time and the fixup is 6.5%. This shows that the cost of addition, though not dominant, plays an important consideration in the total cost. Fixup plays a smaller but noticeable role in the total time. The cost of fixup relative to addition is less than expected from a simple operations count model. This is due to the fact that the fixup primitives are more efficient per operation than matrix addition as discussed in Section 7.2.

On a finer grain scale, the percentages for all three groups are broken up into regions by matrix size. Each region represents matrix sizes where the same number of recursions of the Strassen algorithm is applied. Within a region, the matrix multiplication and fixup costs are divided into curves. Each curve represents one of the 2^d possible fixup choices when doing d recursions, i.e., the bit patterns for the last d bits described in Section 4.4. To show this effect more clearly, the percentage for fixup for matrices up to order 1050 is shown in Figure 19. (See Figure 3 for the same pattern which occurs for similar reasons.) For example, in the region of matrix dimension of 400 to 800, there are 4 curves. These represent the four possible ways to do the fixup work depending on the value of m (mod 4). When the remainder is zero, no fixup work is performed. As the remainder increases, the amount of fixup work increases because more odd-sized matrices are encountered during the recursion, and the percentage of the total time spent on fixup increases. Note that within one region for a single curve (fixed value of mod), the percentage of time spent in fixup decreases as the matrices get larger. This is due to the fact that the type of fixup work is the same and only the size of the matrices and vectors involved changes. Since the fixup costs scale as m^2 but the matrix multiplication times scales as m^3, the percentage of time spent in fixup decreases within a region because that of matrix multiplication, as seen in Figure 18, increases more rapidly. When crossing between regions, there is a jump in the worst case for the fixup cost. This occurs because the number of odd-sized matrices encountered during the recursions is larger so the total cost is larger.

The same type of analysis can be carried out on the other two machines. The qualitative behavior is the same as the RS/6000. The only differences is the actual percentage of time spent on the primitive operations. On the C90, a maximum of 14% is spent on matrix addition and

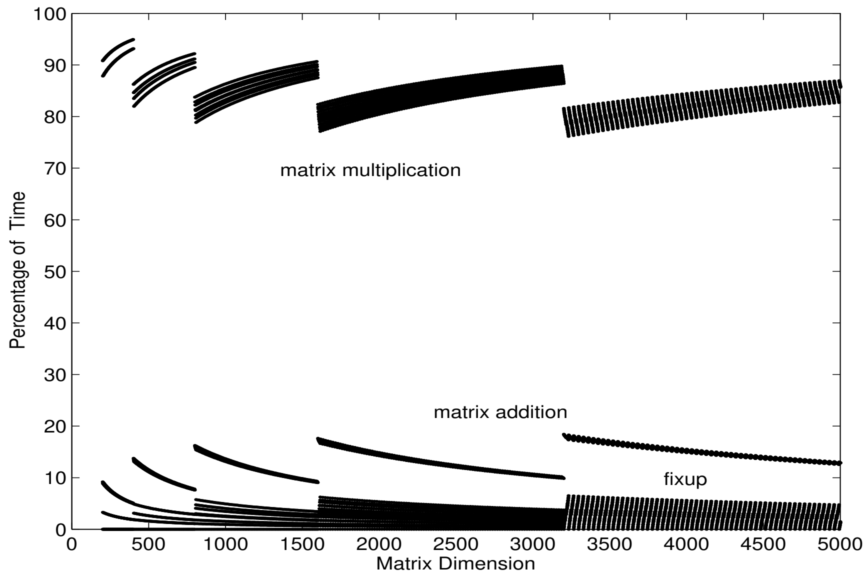

Figure 18: *Model prediction of percentage of time spent on matrix multiplication, matrix addition, and fixup as a function of matrix size on the RS/6000.*

5% on fixup. On the T3D, a maximum of 22% is spent on matrix addition and 6% on fixup. The trends match the relative costs determined in Section 7.3

7.7. Performance Comparisons

In this section we discuss the absolute performance of our Strassen code, DGEFMM, and compare it to the performance of other implementations.

Figure 20 shows the absolute performance of our DGEFMM routine to DGEMM on the RS/6000 for general α and β. The figure shows that DGEFMM is an increasing win over conventional matrix multiplication and, also, that it scales differently. Table 16 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. (Refer to Lemma 2). The scaling of DGEFMM with matrix size 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's algorithm can be substantially faster than conventional matrix multiplication. For the largest size matrix given in Table 16, 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.

Because of the wide range of times in Figure 20, it is difficult to see small effects. To bring these out, Figure 21 shows the DGEFMM time in the previous figure normalized by (time / $m^{lg(7)}$). This figure shows that there is a noticeable variation in times for nearby matrix sizes. This figure has the same saw-toothed feature seen previously in Figure 13. The variation

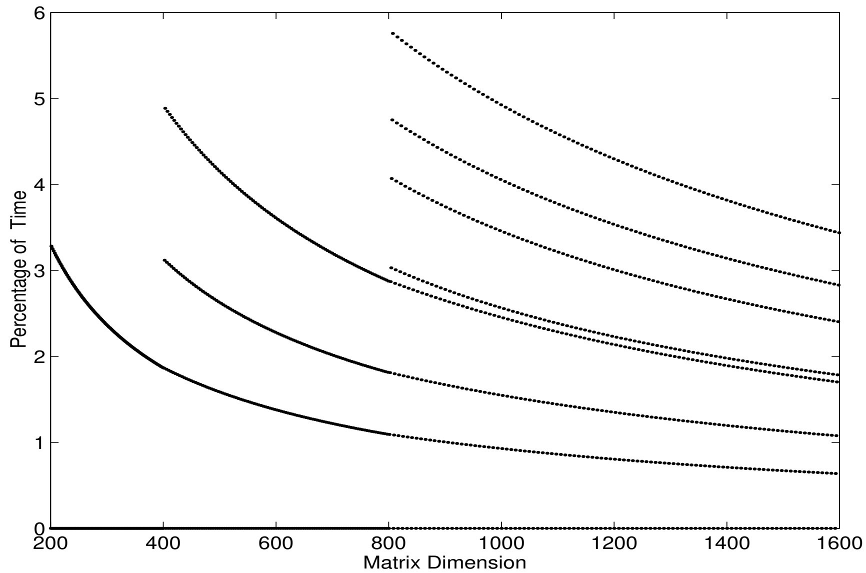

Figure 19: *Model prediction of percentage of time spent on fixup as a function of matrix size on the RS/6000.*

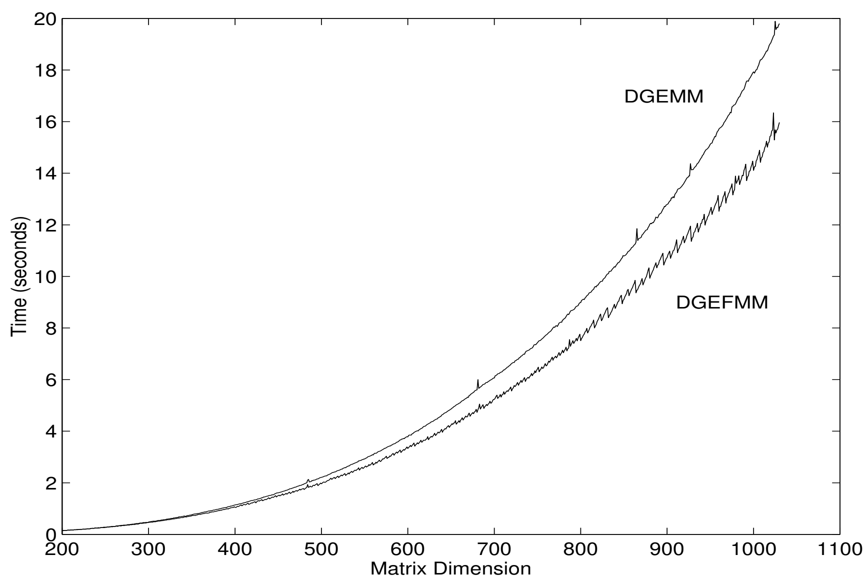

Figure 20: *Time for DGEMM and DGEFMM as a function of matrix size on RS/6000 for general* α *and* β*.*

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 16: *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 11 for each machine.*

is caused by the fact that the odd-sized matrices need to do fixup work. The pattern is not a simple odd versus even initial input matrix size however, since multiple levels of recursion occur for the larger matrices so fixup can also occur for submatrices within the code. Also note that the scaled times show that DGEFMM quickly approaches the asymptotic scaling. This agrees with the data in Table 16.

Figure 22 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 $\alpha = 1$ and $\beta = 0$. In the general case, which the IBM routine does not directly support, the average drops to 1.028. This result supports our design of including the general case directly in our code.

Figure 23 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. Also, 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 however, we have not included these machine-specific techniques in our code.

Finally, we compare to a public domain implementation from Douglas, et al. [11], DGEMMW. We see in Figure 24 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 current package.

We now turn our attention to rectangular matrices. We tested randomly-generated rectan-

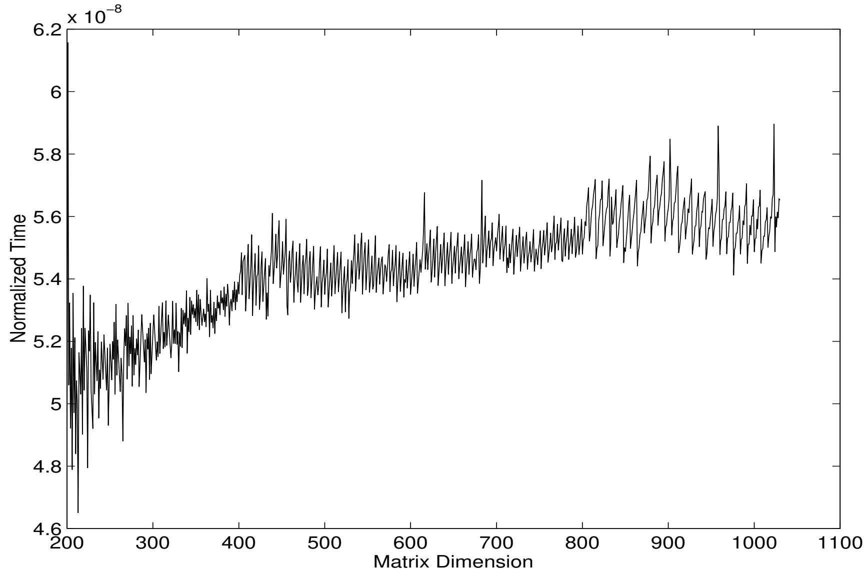

Figure 21: *Normalized time for DGEFMM from Figure 20.*

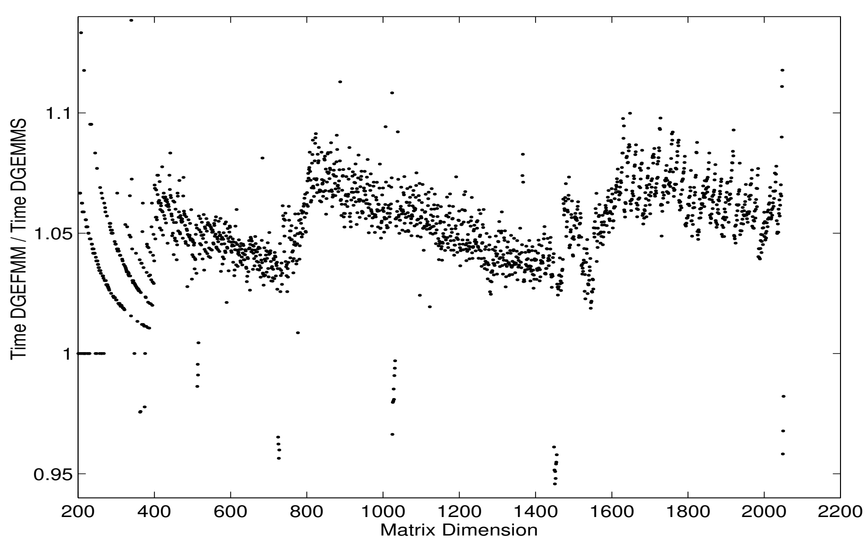

Figure 22: *Ratio of DGEFMM to IBM DGEMMS as a function of matrix size on RS/6000 for* $\alpha = 1$ *and* $\beta = 0$.

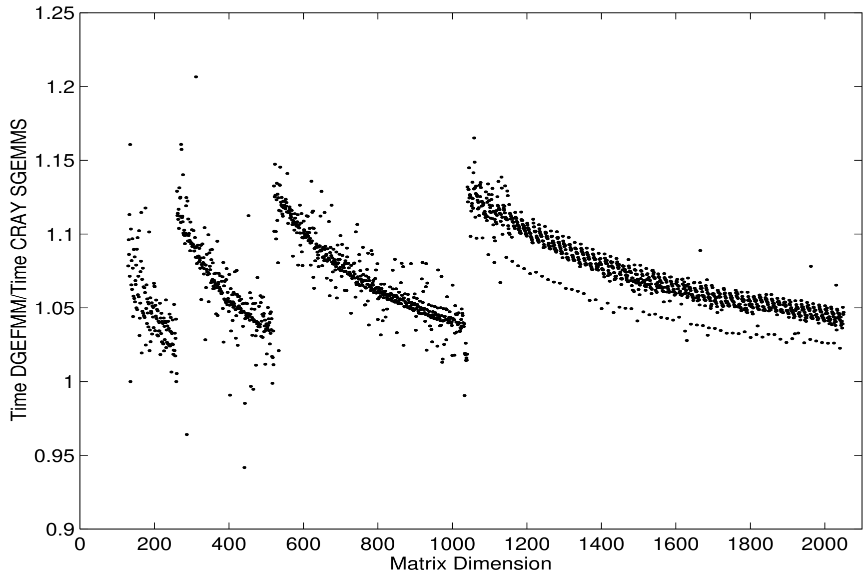

Figure 23: *Ratio of DGEFMM to Cray SGEMMS as a function of matrix size on C90 for* $\alpha = 1$ *and* $\beta = 0$.

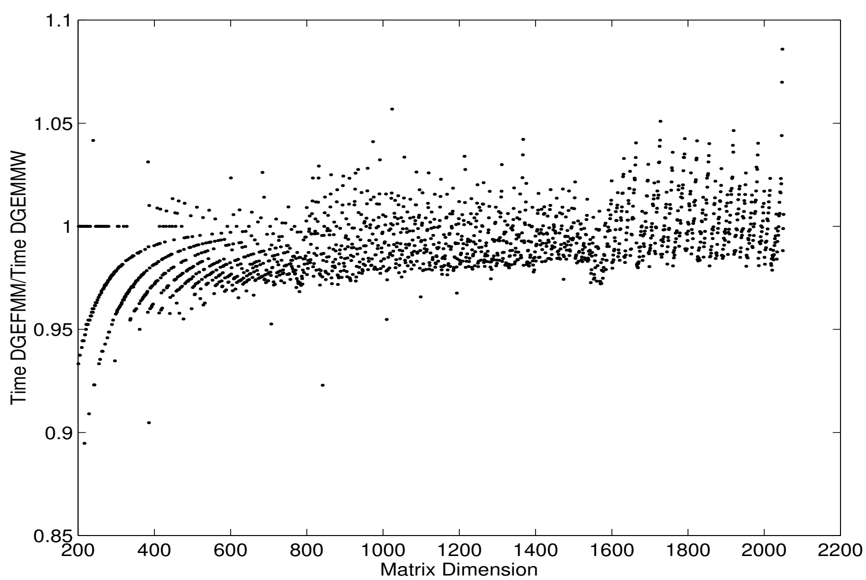

Figure 24: *Ratio of DGEFMM to DGEMMW as a function of matrix order on RS/6000 for general* α *and* β.

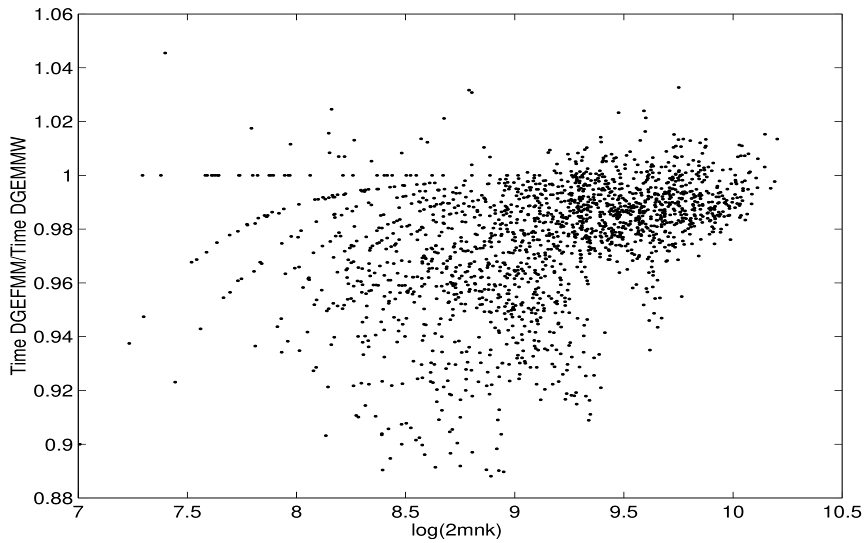

Figure 25: *Ratio of DGEFMM to DGEMMW for random rectangular matrices on RS/6000 with general α and β.*

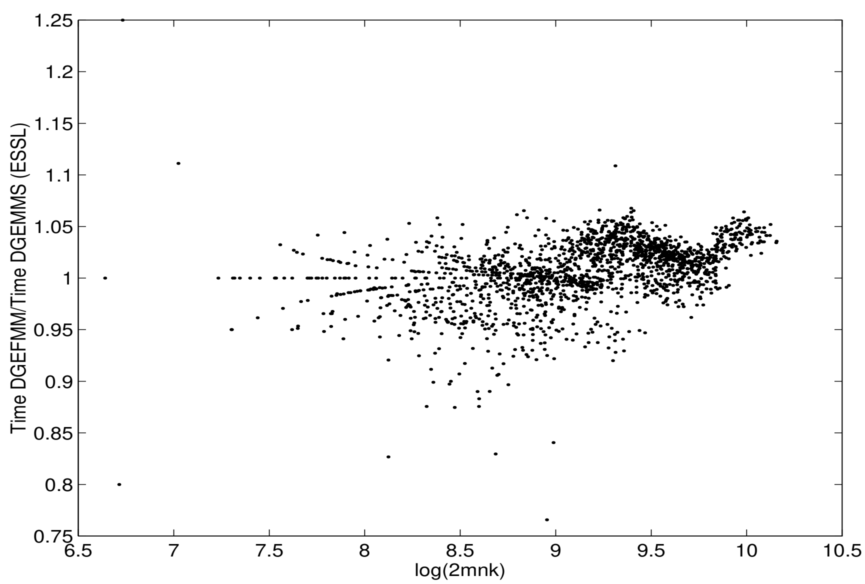

Figure 26: *Ratio of DGEFMM to IBM DGEMMS for random rectangular matrices on RS/6000 with general α and β.*

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

Table 17: *Eigensolver timings for* 1000×1000 *matrix on IBM RS/6000.*

gular matrices and computed the ratio of our routine, DGEFMM, to the routine DGEMMW on the RS/6000 as shown in Figure 25, where $\log(x)$ 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 (16). 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 7.4, since we do not gain an extra level of recursion in many cases.

As a final comparison, we compared DGEFMM to IBM's Strassen algorithm, DGEMMS, on randomly-generated rectangular matrices for general α and β as shown in Figure 26. The average ratio for these runs is 1.0066. Recall that the ratio for the square case was 1.028, so we see that the rectangular case is again a slight improvement. However, IBM's implementation is still slightly faster.

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. Also, our model has shown that our code will continue to scale for larger matrices so the performance will be maintained in this case.

7.8. 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) [17] 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 17 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.

8. Summary and Conclusions

In this paper we have made mathematically precise a methodology for implementing a well-known algorithm for matrix multiplication. Starting with a full operation count model we have analyzed various issues involved in applying Strassen's algorithm to arbitrary matrices, including optimal stopping points for the recursion and different padding or peeling techniques for dealing with odd matrix dimensions. We have developed a new criterion for determining when to switch to the standard algorithm for matrix multiplication. This criterion, based on our theoretical work, has lead to improved performance for rectangular matrices. In addition to the use of this new criterion, our implementation uses dynamic peeling with rank-1 updates, a technique whose usefulness had previously been in question and had not so far been demonstrated. Our analysis showed that this technique is in fact superior to other techniques when operation count is used to compare performance. When other computational models are used, the performance of the two methods are usually competitive. For some inputs one method is better and, for other inputs, another method is better. The behavior on different input sizes can be predicted using our models. Which method is best depends upon the model or machine being used for the comparison.

Our implementation is portable, efficient, and has low memory requirements. 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. When used to perform the matrix multiplications in a sample application, answers are correct and performance is enhanced. The competitive performance of our code illustrates the usefulness of design decisions based on qualitative mathematical analysis and shows the rank-1 update technique to indeed be a viable alternative.

In addition, we have further refined our high-level model using fits to actual performance on the IBM RS/6000, CRAY YMP C90, and CRAY T3D single processor. The resulting quantitative models clearly delineate how the relative speeds of key computational primitives, such as standard matrix multiplication and matrix add/subtract, form the primary factor influencing runtime parameters. Our enhanced model is also able to correctly estimate these parameters, including practical cutoff values and execution timings.

Further work could include completing our models for the rectangular case, as well as applying our methodology to other machines and/or other algorithms. The qualitative models allow high level design decisions to be made before doing an implementation. After development of such an implementation, its behavior can be easily examined using the best-fit models without requiring actual runs of the code. The ability to do both of these for a wide range of important computations would be of great value in many areas of high performance computing.

9. Acknowledgements

Special thanks to R. W. Johnson who helped introduce us to this problem. The authors would also like to thank Dave Carlson, who provided valuable discussions and hard-to-find references on pebbling.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings, Supercomputing '90*, pages 2–11, Los Alamitos, CA, 1990. IEEE Computer Society Press.

[2] 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.

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

[4] 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.

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

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

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Symbolic Computation*, 9(3):251–280, 1990. Shorter version appears in Proceedings, 19th Annual ACM Symposium on the Theory of Computing (STOC), 1987, New York, NY, pp. 1–6.

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

[9] Cray Research, Inc. Volume 3: (UNICOS) Math and Scientific Library Reference Manual (SR-2081), Release 5.0. Mendota Heights, MN, 1989.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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).

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

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

[20] A. Kreczmar. On memory requirements of Strassen algorithms. In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science 1976*, volume 45 of *Lecture Notes in Computer Science*, pages 404–407. Springer-Verlag, Berlin, 1976.

[21] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings, ASPLOS IV*, pages 63–74, New York, NY, 1991. ACM.

[22] V. Pan. Strassen algorithm is not optimal. trilinear technique of aggregating, uniting, and canceling for constructing fast algorithms for matrix multiplication. In *Proceedings, 19th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 166–176, Ann Arbor, MI, 1978.

[23] N. Pippenger. Pebbling. Unpublished manuscript, May 1980.

[24] W. H. Press, S. A. Teukolsy, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN: The Art of Scientific Computing.* Cambridge University Press, Cambridge, UK, second edition, 1992.

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

[26] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, second edition, 1988.

[27] J. Speiß. Untersuchungen des zeitgewinns durch neue algorithmen zur matrix-multiplikation. *Computing*, 17:23–36, 1976.

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