Tuning Strassen's Matrix Multiplication for Memory Efficiency

Mithuna Thottethodi
Department of Computer Science
Duke University
Durham, NC 27708-0129.
Phone: 919 660 6587. FAX: 919 660 6519.
mithuna@cs.duke.edu
http://www.cs.duke.edu/~mithuna

Siddhartha Chatterjee
Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27599-3175.
Phone: 919 962 1766. FAX: 919 962 1799.
sc@cs.unc.edu
http://www.cs.unc.edu/~sc

Alvin R. Lebeck
Department of Computer Science
Duke University
Durham, NC 27708-0129.
Phone: 919 660 6551. FAX: 919 660 6519.
alvy@cs.duke.edu
http://www.cs.duke.edu/~alvy

Abstract:

Strassen's algorithm for matrix multiplication gains its lower arithmetic complexity at the expense of reduced locality of reference, which makes it challenging to implement the algorithm efficiently on a modern machine with a hierarchical memory system. We report on an implementation of this algorithm that uses several unconventional techniques to make the algorithm memory-friendly. First, the algorithm internally uses a non-standard array layout known as Morton order that is based on a quad-tree decomposition of the matrix. Second, we dynamically select the recursion truncation point to minimize padding without affecting the performance of the algorithm, which we can do by virtue of the cache behavior of the Morton ordering. Each technique is critical for performance, and their combination as done in our code multiplies their effectiveness.

Performance comparisons of our implementation with that of competing implementations show that our implementation often outperforms the alternative techniques (up to 25%). However, we also observe wide variability across platforms and across matrix sizes, indicating that at this time, no single implementation is a clear choice for all platforms or matrix sizes. We also note that the time required to convert matrices to/from Morton order is a noticeable amount of execution time (5% to 15%). Eliminating this overhead further reduces our execution time.

Keywords:

matrix multiply, strassen's algorithm, cache memory, data layout

1 Introduction

The central role of matrix multiplication as a building block in numerical codes has generated a significant amount of research into techniques for improving the performance of this basic operation. Several of these efforts [3,6,12,13,14,19] focus on algorithms whose arithmetic complexity is $O(n^{\log_2 7})$ instead of the conventional $O(n^3)$ algorithm. Strassen's algorithm [23] for matrix multiplication and its variants are the most practical of such algorithms, and are classic examples of theoretically high-performance algorithms that are challenging to implement efficiently on modern high-end computers with deep memory hierarchies.

Strassen's algorithm achieves its lower complexity using a divide-and-conquer approach. Unfortunately, this technique has two potential drawbacks. First, if the division proceeds to the level of single matrix elements, the recursion overhead (measured, for instance, by the recursion depth and additional temporary storage) becomes significant and reduces performance. This overhead is generally limited by stopping the recursion early and performing a conventional matrix multiplication on submatrices that are below the *recursion truncation point* [13]. Second, the division step must efficiently handle odd-sized matrices. This can be solved by one of several schemes: by embedding the matrix inside a larger one (called *static padding*), by decomposing into submatrices that overlap by a single row or column (called *dynamic overlap*), or by performing special case computation for the boundary cases (called *dynamic peeling*).

Previous implementations have addressed these two drawbacks independently. We present a novel solution that simultaneously addresses both issues. Specifically, dynamic peeling was introduced as a method to avoid large amounts of static padding. The large amount of static padding is an artifact of using a fixed recursion truncation point. We can minimize padding by dynamically selecting the recursion truncation point from a range of sizes. However, this scheme can induce significant variations in performance when using a canonical storage scheme (such as column-major) for the matrices. By using a hierarchical matrix storage scheme, we can dynamically select the recursion truncation point within a range of sizes that both ensures high and stable performance of the computations at the leaf of the recursion tree and limits the amount of static padding.

We measured execution times of our implementation (MODGEMM) and two alternative implementations, DGEFMM uses dynamic peeling [13] and DGEMMW uses dynamic overlap [6], on both a DEC Alpha and a SUN UltraSPARC II. Our results show wide variability in the performance of all three implementations. On the Alpha, our implementation (MODGEMM) ranges from 30% slower to 20% faster than DGEFMM for matrix sizes from 150 to 1024. On the Ultra, MODGEMM is generally faster (up to 25%) than DGEFMM for large matrices (500 and larger), while DGEFMM is generally faster for small matrices (up to 25%). We also determine the time to convert matrices to/from Morton order ranges from 5% to 15% of total execution time. When eliminating this conversion time, by assuming matrices are already in Morton order, MODGEMM outperforms DGEFMM for nearly all matrix sizes on both the Alpha and Ultra, with greater benefits on the Ultra.

The remainder of this paper is organized as follows. Section 2 reviews the conventional description of Strassen's algorithm. Section 3 discusses the implementation issues that affect memory efficiency, and our solutions to these issues. Section 4 presents performance results for our code. Section 5 cites related work and compares our techniques to them. Section 6 presents conclusions and future work.

2 Background

Strassen's original algorithm [23] is usually described in the following divide-and-conquer form. Let A and B be two $n \times n$ matrices, where n is an even integer. Partition the two input matrices A and B and the result matrix C into quadrants as follows.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \bullet \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad (1)$$

The symbol $\bullet$ in equation (1) represents matrix multiplication. We then compute the four quadrants of the result matrix as follows.

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

In this paper, we discuss and implement Winograd's variant [7] of Strassen's algorithm, which uses seven matrix multiplications and 15 matrix additions. It is well-known that this is the minimum number of multiplications and additions possible for any recursive matrix multiplication algorithm based on division into quadrants. The division of the matrices into quadrants follows equation (1). The computation proceeds as follows.

$$\begin{array}{llll} C_{11} = & U_1 = P_1 + P_2 & P_1 = A_{11} \bullet B_{11} & \\ & U_2 = P_1 + P_4 & P_2 = A_{12} \bullet B_{21} & \\ & U_3 = U_2 + P_5 & P_3 = S_1 \bullet T_1 & \\ C_{21} = & U_4 = U_3 + P_7 & P_4 = S_2 \bullet T_2 & \\ C_{22} = & U_5 = U_3 + P_3 & P_5 = S_3 \bullet T_3 & \\ & U_6 = U_2 + P_3 & P_6 = S_4 \bullet B_{22} & \\ C_{12} = & U_7 = U_6 + P_6 & P_7 = A_{22} \bullet T_4 & \end{array} \qquad \begin{array}{ll} S_1 = A_{21} + A_{22} & T_1 = B_{12} - B_{11} \\ S_2 = S_1 - A_{11} & T_2 = B_{22} - T_1 \\ S_3 = A_{11} - A_{21} & T_3 = B_{22} - B_{12} \\ S_4 = A_{12} - S_2 & T_4 = B_{21} - T_2 \end{array}$$

Compared to the original algorithm, the noteworthy feature of Winograd's variant is its identification and reuse of common subexpressions. These shared computations are responsible for reducing the number of additions, but also contribute to worse locality of reference unless special attention is given to this aspect of the computation.

We do not discuss in this paper numerical issues concerning these fast matrix multiplication algorithms, as they are covered elsewhere [10].

2.1 Interface

In order to stay consistent with previous work in this area and to permit meaningful comparisons, our implementation of Winograd's variant follows the same calling conventions as the `dgemm` subroutine in the Level 3 BLAS library [5]. Thus, the implementation computes $C \leftarrow \alpha * \mathrm{op}(A) \bullet \mathrm{op}(B) + \beta * C$,where α and β are scalars, op(*A*) is an $m \times k$ real matrix, op(*B*) is a $k \times n$ real matrix, *C* is an $m \times n$ real matrix, and op(*X*) is either *X* or X^T. The matrices *A*, *B*, and *C* are stored in column-major order, with leading dimensions ldA, ldB, and ldC respectively.

3 Memory Efficiency: Issues and Techniques

To be useful in practice, an implementation of the Strassen-Winograd algorithm must answer three key questions: when to truncate the recursion, how to handle arbitrary rectangular matrices, and how to lay out the array data in memory to promote better use of cache memory. These questions are not independent, although past implementations have treated them thus. We now review these implementation issues, identify possible solution strategies, and justify our specific choices.

3.1 Recursion truncation point

The seven products can be computed by recursively invoking Strassen's algorithm on smaller subproblems, and switching to the conventional algorithm at some matrix size *T* (called the *recursion truncation point* [13]) at which Strassen's construction is no longer advantageous. If one were to estimate running time by counting arithmetic operations, the recursion truncation point would be around 16. However, the empirically observed value of this parameter is at least an order of magnitude higher. This discrepancy is a direct result of the poor (algorithmic) locality of reference of Strassen's algorithm.

All implementations of Strassen's algorithm that we have encountered use an empirically chosen cutoff criterion for determining the matrix size *T* at which to terminate recursion.

3.2 Handling arbitrary matrices

Divide-and-conquer techniques are most effective when the matrices can be evenly partitioned at each recursive invocation of the algorithm. We first note that rectangular matrices present no particular problem for partitioning if all matrix dimensions are even. The trouble arises when we encounter matrices with one or more dimensions of odd size. There are several possible solutions to this problem.

- The simplest solution, *static padding* , is to pad the $n \times n$ matrices with additional rows and columns containing zeros such that the padded $n' \times n'$ matrix satisfies the ``even-dimensions'' condition at each level of recursion, i.e., $n' = T \cdot 2^d$,where $d > 0$ is the recursion depth.

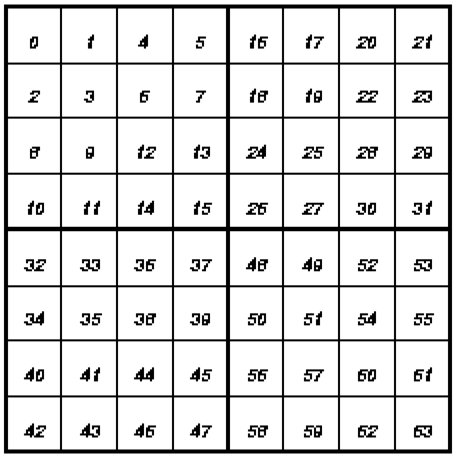
  This is Strassen's original solution and the solution most often quoted in algorithms textbooks. However, for a statically predetermined value of *T*, the overhead of static padding can become quite severe, adding almost three times the number of original matrix elements in the worst case. Furthermore, in a naive implementation of this idea, the arithmetic done on these additional zero elements is pure overhead. Finally, based on the relation between *n*' and *T*, cache interference phenomena can reduce performance. These interference effects can be mitigated using non-standard data layouts, as we discuss further in Section 3.3.

- A second solution, *dynamic peeling* , peels off the extra row or column at each level, and separately adds their contributions to the overall solution in a later fix-up computation [13]. This eliminates the need for extra padding, but reduces the portion of the matrix to which Strassen's algorithm applies, thus reducing the potential benefits of the recursive strategy. The fix-up computations are matrix-vector operations rather than matrix-matrix operations, which limits the amount of reuse and reduces performance.
- A third solution, *dynamic overlap* , finesses the problem by subdividing the matrix into submatrices that (conceptually) overlap by one row or column, computing the results for the shared row or column in both subproblems, and ignoring one of the copies. This is an interesting solution, but it complicates the control structure and performs some extra computations.

Ideally, we would like to avoid both the implementation complexity of dynamic peeling or dynamic overlap and the possibility of excessive static padding.

3.3 Data layout

Figure 1: Morton-ordered matrix layout. Each small square is a $T \times T$ tile that is laid out contiguously in column-major order. The number in each tile gives its relative position in the sequence of tiles.



0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

A significant fraction of the computation of the Strassen-Winograd algorithm occurs in the routine that multiplies submatrices when the recursion truncates. The performance of this matrix product is largely determined by its cache behavior. This issue has not been explicitly considered in previous work on implementing fast recursive matrix multiplication algorithms, where the default column-major layout of array data has been assumed.

A primary condition for performance is to choose a tile size T such that the tiles fit into the first-level cache, thus avoiding *capacity misses* [11]. This is easily achieved using tile sizes in the range shown in Figure 2. Second, to achieve performance stability as T varies, it is also important to have the tile contiguous in memory, thus avoiding *self-interference misses* [17]. Given the hierarchical nature of the algorithm (the decomposition is by quadrants within quadrants within $\ldots$), hierarchical layouts such as *Morton ordering* [8] naturally suggest themselves for storing the matrices.

An operational definition of Morton ordering is as follows. Divide the original matrix into four quadrants, and lay out these quadrants in memory in the order NW, NE, SW, SE. A submatrix larger on the side than T is laid out recursively using the Morton ordering; a $T \times T$ tile is laid out using column-major ordering. See Figure 1.

A secondary benefit of keeping tiles contiguous in memory is that the matrix addition operations can be performed with a single loop rather than two nested loops, thus reducing loop overheads.

3.4 The connections among the issues

We begin by observing that the recursion truncation point T determines the amount of padding (n'-n), since the matrix must evenly divide at all recursive invocations of the algorithm. We also note that at the recursion truncation point, we are multiplying $T \times T$ submatrices using the conventional algorithm. By carefully selecting the truncation point, we can minimize the amount of padding required.

Figure 2: Effect of tile size on padding. When minimizing padding, tiles are chosen from the range 16 to 64.

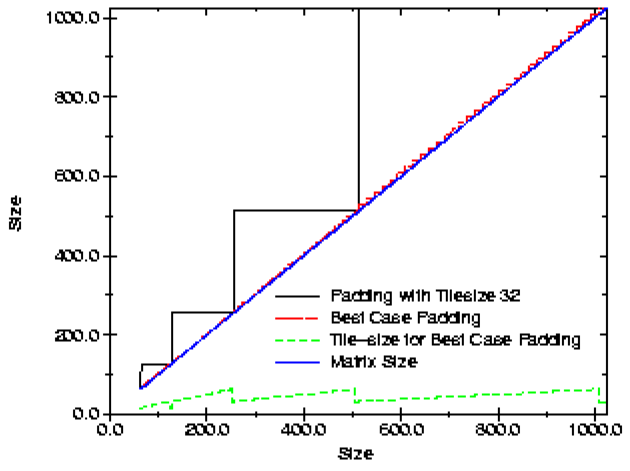
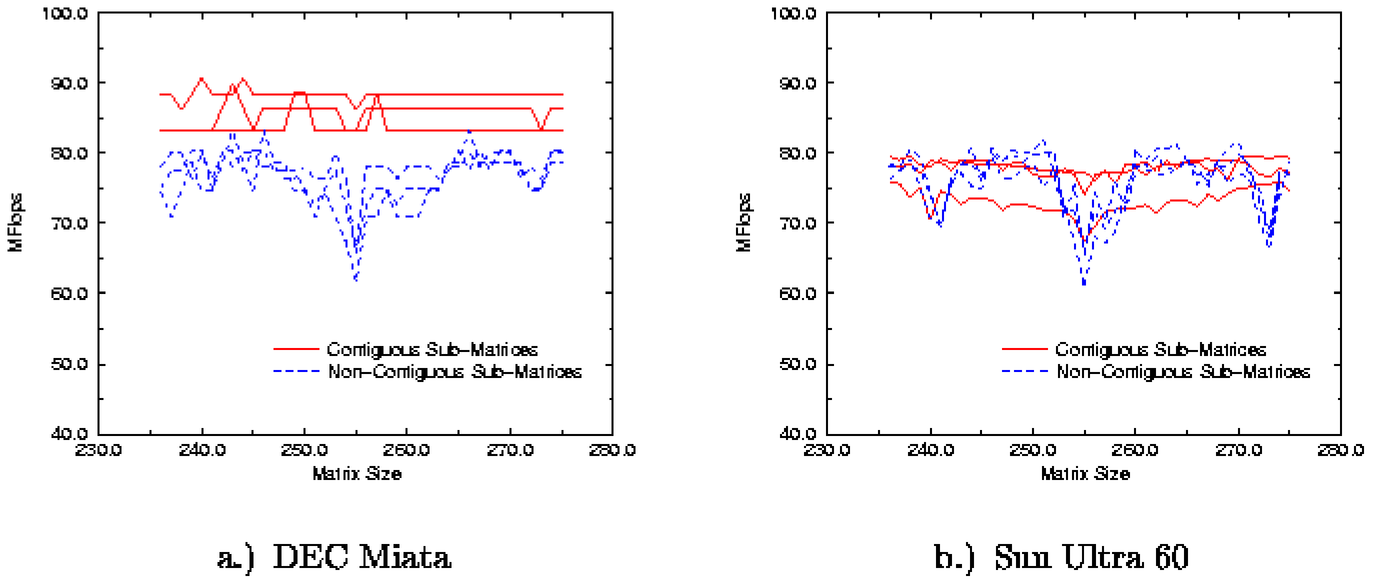


Figure 2 shows the effects of tile size selection on padding. The four lines correspond to the original matrix size *n*, the padded matrix size *n*' with the tile size *T* chosen to minimize padding, the padded matrix size *n*' for fixed tile size *T* = 32, and the tile size *T* that achieves the minimum padding. This figure demonstrates that the ability to select from a range of tile sizes can dramatically reduce the amount of extra storage, making it independent of the matrix size *n*. In contrast, a fixed tile size can require significant padding, proportional to the matrix size in the worst case.

Consider a square matrix size of 513. With a fixed tile size of 32, static padding requires a padded matrix of size 1024, nearly twice the original matrix dimension. In contrast, flexibility in choosing tile size allows us to select a tile size of 33, which requires padding with only 15 (our worst case amount) extra elements in each dimension. The padded matrix size, 528, is recursively divided four times to achieve 33×33 tiles.

Figure 3: Effect of Data Layout on Matrix Multiply Performance



However, we can exploit this flexibility in tile size selection only if we can ensure that the performance of the matrix multiplication algorithm for these tiles is not sensitive to the choice of tile size. This is an important consideration, since a significant portion of the algorithm's computation occurs in the routine that multiplies tiles. Figure 3a and Figure 3b show how the performance of matrix multiplication, $C \leftarrow A \times B$, varies as a function of the relation between tile size and leading dimension. Each line in the graph corresponds to a different submatrix size (T = 24, 28, 32). The submatrices to operate on are chosen from a base matrix (*M*) as follows: A[1,1] = M[1,1], B[1,1] = M[T+1,T+1], C[1,1] = M[2T+1,2T+1]. Non-contiguous submatrices are created by setting the leading dimension of each submatrix to the leading dimension of the base matrix, M, which corresponds to the x-axis. Contiguous submatrices are formed by setting the leading dimension of each submatrix to the tile size, T, which corresponds to each line in the graph.

From Figure 3 we see that contiguous submatrices exhibit much more stable performance than non-contiguous submatrices. As expected, the non-contiguous submatrices exhibit a dramatic drop in performance when the base matrix is a power of two (256 in this case), due to self-interference. On the Alpha, the contiguous submatrices clearly outperform the non-contiguous. The performance differential is not as pronounced on the Ultra, however the instability of the non-contiguous submatrices still exists. These results justify the use of Morton ordering within our implementation.

3.5 Implementation details

We can envision three different alternatives for storage layout: the input and output matrices are assumed to be laid out in Morton order at the interface level; the input and output matrices are copied from column-major storage to Morton order at the interface level; and the conversion to Morton order is done incrementally across the levels of recursion. The first alternative is not feasible for a library implementation. Among the two other options, converting the matrices to and from Morton order at the top level was easier to implement, and performed relatively fast in practice (5% to 15% of total execution time, see Section 4.1).

We incorporate any necessary matrix transposition operations during the conversion from column-major to Morton order. This is handy, because it requires only a single core routine for the Strassen-Winograd algorithm. The alternative solution requires multiple code versions or indirection using pointers to handle these cases correctly.

Choosing tile sizes from a range as described above will in general induce a small constant amount of padding. In our implementation, we explicitly padded out the matrix with zeros and performed redundant computation on the pad. We could afford to do this because the pad was guaranteed to be small. The alternative scheme of avoiding this overhead would have created tiles of uneven sizes and required more control overhead to keep track of tile start offsets and similar pieces of information.

We handle rectangular cases by treating the two dimensions separately. Each tile dimension is chosen to minimize padding in that dimension. This method of choosing each tile dimension independently works when the ratio of columns to rows (or rows to columns) is within certain limits. Highly rectangular matrices pose a problem because the two recommended tile dimensions may require different levels of recursion. The following example illustrates the difficulties of this method for such matrices.

Consider a highly rectangular matrix of dimensions 1024x256. We choose the tile dimensions independently. First, we consider that the matrix has 1024 rows, and choose the number of rows in the tile that minimizes row padding (i.e., the number of additional rows). In this case, 32 is chosen, and the recursion is required to unfold to a depth of 5. Next, we consider that the matrix has 256 columns. The number of columns in the tile is chosen to minimize the number of columns that are to be padded. Again, we choose 32, but the recursion must unfold to a depth of only 3. Clearly, naively choosing the two tile dimensions independently does not work for highly rectangular matrices, since we can not unfold the recursion to both a depth of 5 and to a depth of 3.

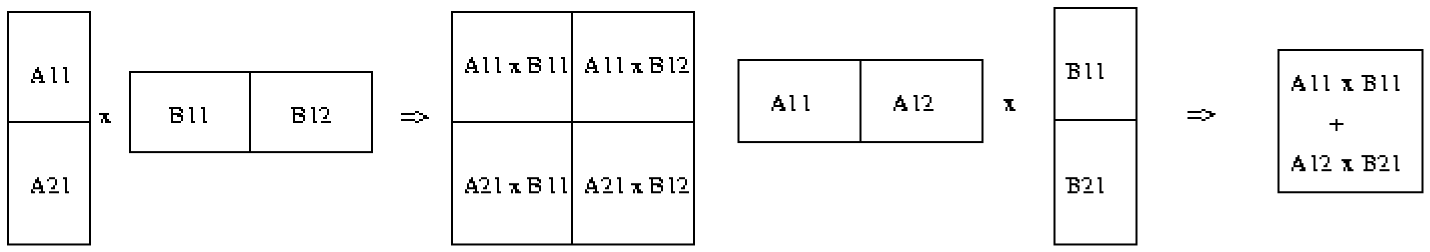
To overcome this limitation, the matrix is divided into submatrices such that all submatrices require the same depth of recursion unfolding for both dimensions. The matrix product is reconstructed in terms of the submatrix products.

A given matrix can be

- *wide*, meaning its columns-to-rows ratio exceeds the desired ratio,
- *lean*, meaning its rows-to-columns ratio exceeds the desired ratio, or
- *well-behaved* meaning both its columns-to-rows ratio and rows-to-columns ratio are within the desired ratio.

Since there are two input matrices (*A* and *B*), and each can any one of the above forms, there are a total of nine possible combinations. Figure 4a and Figure 4b show two examples of how the input matrices (*A* and *B*) are divided, and how the result (*C*) is reconstructed from results of submatrix multiplications.

Figure 4: Handling of Highly Rectangular Matrices.



a.) *Lean A and wide B.* b.) *Wide A and lean B*

Finally, we note that 1 and 0 are common values for the α and βparameters. In order to avoid performing extra arithmetic for these parameter values, the core routine for the Strassen-Winograd algorithm computes $D \leftarrow A \bullet B$,with *D* being set to *C* if $\beta = 0$ and to a temporary otherwise. We then post-process to compute $C \leftarrow \alpha * D + \beta * C$,if such post-processing is necessary.

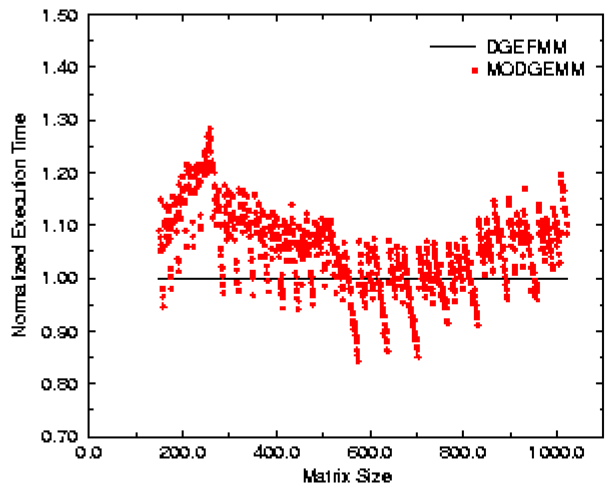
4 Results

This section compares the performance of our implementation (MODGEMM) of Strassen's algorithm to a previous implementation that uses dynamic peeling (DGEFMM) [13] (we use the author's original code), and to a previous implementation that uses dynamic overlap (DGEMMW) [6]. We measure the execution time of the various implementations on a 500 MHz DEC Alpha Miata and a 300 MHz Sun Ultra 60. The Alpha machine has a 21164 processor with an 8KB direct-mapped level 1 cache, a 96KB 3-way associative level 2 cache, a 2MB direct-mapped level 3 cache, and 512MB of main memory. The Ultra has two UltraSPARC II processors, each with a 16 K-byte level 1 cache, a 2MB level 2 cache, and 512MB of main memory. We use only one processor on the Ultra 60.

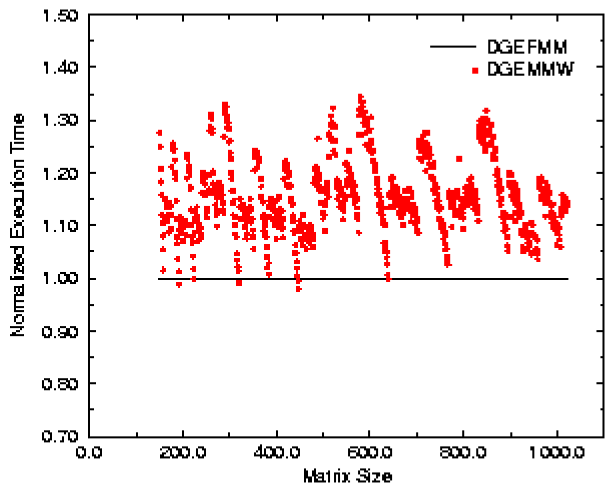
We timed the execution of each implementation using the UNIX system call `getrusage` for matrix sizes ranging from 150 to 1024, and $\alpha = 1$ and $\beta = 0$. For DGEFMM we use the empirically determined recursion truncation point of 64. For matrices less than 500 we compute the average of 10 invocations of the algorithm to overcome limits in clock resolution. Execution times for larger matrices are large enough to overcome these limitations. To further reduce experimental error, we execute the above experiments three times for each matrix size, and use the minimum value for comparison. The programs were compiled with vendor compilers (cc and f77) with the -fast option. The Sun compilers are the Workshop Compilers 4.2, and the DEC compilers are DEC C V5.6-071 and DIGITAL Fortran 77 V5.0-138-3678F.

Figure 5 and Figure 6 show our results for the Alpha and UltraSPARC, respectively. We report results in execution time normalized to the dynamic peeling implementation (DGEFMM). On the Alpha we see that DGEFMM generally outperforms dynamic overlap (DGEMMW), see Figure 5b. In contrast, our implementation (MODGEMM) varies from 30% slower to 20% faster than DGEFMM. We also observe that MODGEMM outperforms DGEFMM mostly in the range of matrix sizes from 500 to 800, whereas DGEFMM is faster for smaller and larger matrices. Finally, by comparing Figure 5a and Figure 5b, we see that MODGEMM generally outperforms DGEMMW.

Figure 5: Performance of Strassen Winograd Implementations on DEC Miata $\alpha = 1, \beta = 0$

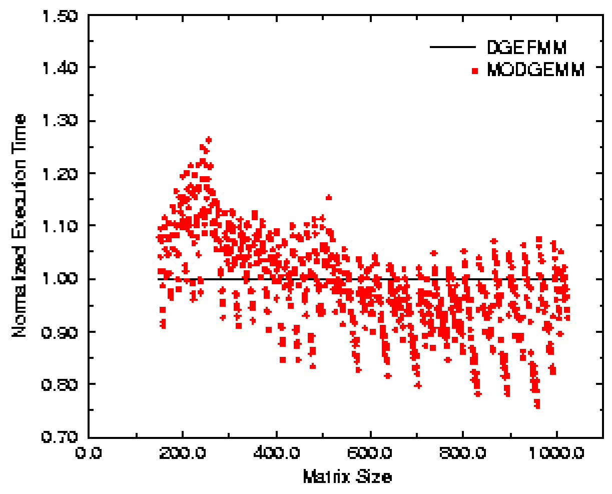


a.) MODGEMM vs DGEFMM

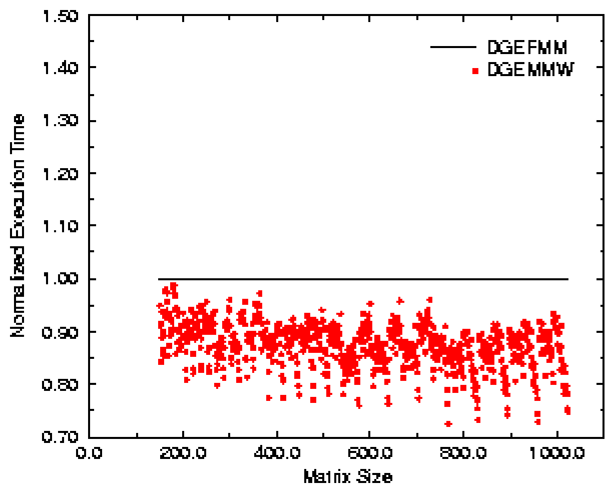


b.) DGEMMW vs DGEFMM

Figure 6: Performance of Strassen Winograd Implementations on Sun Ultra 60 $\alpha = 1, \beta = 0$



a.) MODGEMM vs DGEFMM



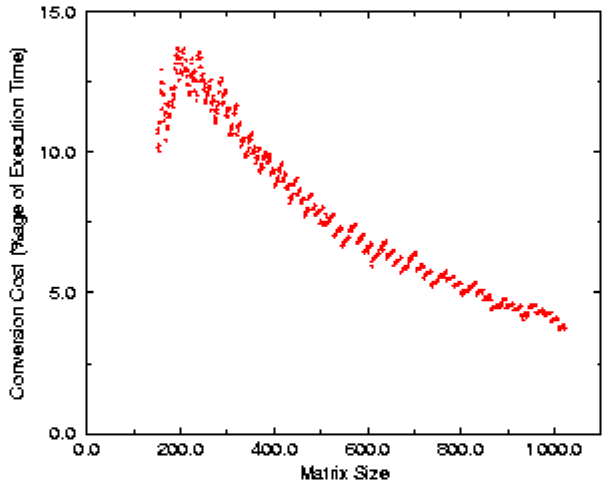
b.) DGEMMW vs DGEFMM

The results are quite different on the Ultra (see Figure 6). The most striking difference is the performance of DGEMMW (see Figure 6b), which outperforms both MODGEMM and DGEFMM for most matrix sizes on the Ultra. Another significant difference is that MODGEMM is generally faster than DGEFMM for large matrices (500 and larger), while DGEFMM is generally faster for small matrices.

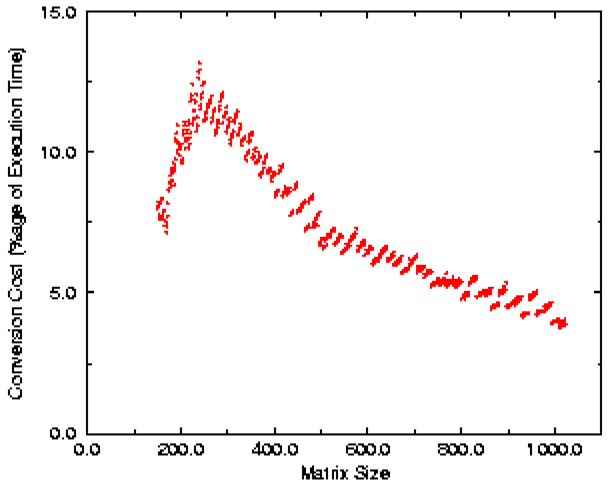
An important observation from the above results is the variability in performance both across platforms and across matrix sizes. Our ongoing research efforts are targeted at understanding these variations. Section 4.2 reports on some of our preliminary findings, and the following section analyzes the penalty of converting to Morton order.

4.1 Morton Conversion Time

Figure 7: Morton Conversion Time as Percentage of Total Execution Time



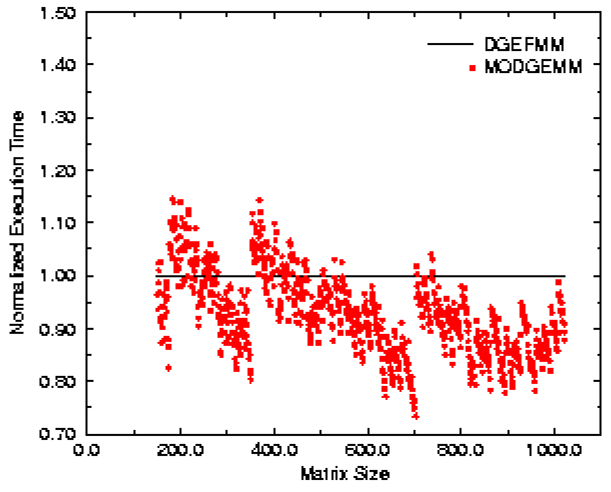
a.) DEC Miata



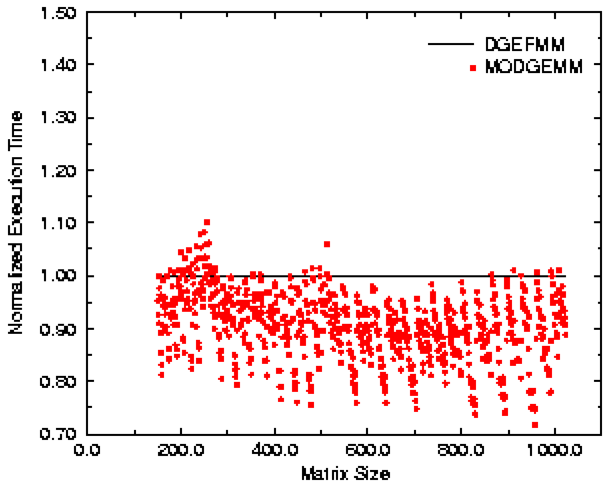
b.) Sun Ultra 60

An important aspect of our implementation is the recursive data layout, which provides stable performance for dynamic tile size selection. The previous performance results include the time to convert the two input matrices from column-major to Morton order, and to convert the output matrix from Morton order back to column-major. Figure 7a and Figure 7b show the cost of this conversion as a percentage of the entire matrix multiply for each of our platforms. From these graphs we see that Morton conversion accounts for up to 15% of the overall execution time for small matrices and approximately 5% for very large matrices.

Figure 8: Performance of MODGEMM without Matrix Conversion



a.) DEC Miata

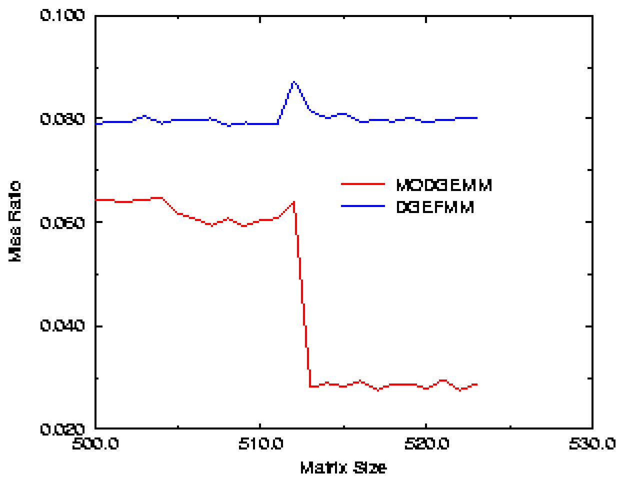


b.) Sun Ultra 60

These results show that Morton conversion is a noticeable fraction of the execution time. Eliminating the conversion cost (i.e, assuming the matrices are already in Morton order) produces commensurate improvements in the performance of our implementation. Figure 8a and Figure 8b shows that without conversion costs MODGEMM does indeed execute faster, and increases the number of matrix sizes at which it outperforms DGEFMM. Specifically, on the Alpha (Figure 8a) MODGEMM is superior for most matrix sizes larger than 500, and on the Ultra (Figure 8b) we see that for only a few matrix sizes DGEFMM outperforms MODGEMM. Furthermore, without Morton conversion, MODGEMM is very competitive with DGEMMW, and outperforms it for many matrix sizes.

4.2 Cache Effects

Figure 9: Cache Miss Ratios for 16KB Direct-Mapped with 32 Byte Blocks



Our initial efforts to gain further insight into the performance variability of our implementation begins with analysis of its cache behavior. Here, we present preliminary results. We used ATOM [22] to perform cache simulations of a 16KB direct-mapped cache with 32 byte blocks of both the DGEFMM, and MODGEMM implementations. Figure 9 shows the miss ratios of each implementation for matrix sizes ranging from 500 to 523. The first observation from this graph is that MODGEMM miss ratios (6% to 2%) are lower than DGEFMM (8%), which matches our expectations. The second observation is the unexpected dramatic drop in MODGEMM's miss ratio at a matrix size of 513. Preliminary investigations using CProf [18] reveal that this drop is due to a reduction in conflict misses.

To understand this phenomenon, consider that for matrix sizes of 505 to 512 the padded matrix size is 512 and the recursion truncation point is at tile size 32. The conventional algorithm is applied to submatrices that are each 8KB in size (32x32x8), and correspond to the four quadrants of a 64x64 submatrix. With Morton ordering the quadrants are allocated contiguously in memory, and quadrants separated by a multiple of the cache size (16KB) conflict in the cache. For example, since the NW and SW quadrants are separated by the NE quadrant, they map to the same locations in cache (i.e, the first elements of the NW and SW quadrants are separated by 16KB). Therefore, any operations involving these two quadrants will incur a significant number of cache misses. We are currently examining ways to eliminate these conflict misses.

5 Related Work

We discuss three separate areas of related work: implementations of Strassen-type algorithms, hierarchical schemes for matrix storage, and compiler technology for improving the cache behavior of loop nests.

5.1 Other implementations

Previous implementations of Strassen's algorithm include Bailey's implementation for the CRAY-2 [3], the DGEMMW implementation by Douglas *et al.* [6], and the DGEFMM implementation by Huss-Lederman *et al.* [13]. Bailey, coding in Fortran, used a static two-level unfolding of the recursion by code duplication. Douglas *et al.* introduced the *dynamic overlap* method for handling odd-sized dimensions. Huss-Lederman *et al.* introduced the *dynamic peeling* method. While all of these implementations are careful to limit the amount of temporary storage, they do not specifically consider the performance of the memory hierarchy on their code. In some cases (such as on the CRAYs), this issue did not arise because the memory system was not cache-based. Section 4 gives extensive performance comparisons of our implementation vs. DGEFMM and DGEMMW.

Kreczmar [16] proposes an elegant memory-efficient version of Strassen's algorithm based on overwriting one of the input arguments. His scheme for space savings is not directly applicable for two reasons: we cannot assume that the input matrices can be overwritten, and his scheme requires several copying operations that reduce performance.

5.2 Hierarchical schemes for matrix storage

Wise and his coauthors [1,24] have investigated the algorithmic advantages of quad-tree representations of matrices. Morton ordering has also appeared in the parallel computing literature, where is has been used for load balancing of irregular problems [20]. Most recently, Frens and Wise [8] discuss an implementation of a recursive $O(n^3)$ matrix multiplication algorithm using hierarchical matrix layout, in which they sequence the recursive calls in an unusual manner to get better reuse in cache. We do not carry the recursion to the level of single matrix elements as they do, but truncate the recursion when we reach tile sizes that fit in the upper levels of the memory hierarchy.

5.3 Cache behavior

Several authors [17,15,4,21] discuss loop transformations such as tiling that attempt to reduce the number of cache misses incurred by a loop nest and thus improve its performance. While these loop transformations are not specific to matrix multiplication, the conventional three-loop algorithm for matrix multiplication falls into the category of codes that they can handle.

Lam, Rothberg, and Wolf [17] investigated and modeled the influence of cache interference on the performance of tiled programs. They emphasized the importance of having tiles be contiguous in memory to avoid self-interference misses, and proposed data copying to satisfy this condition. Our top-level conversion between the column-major layout at the interface level and the Morton ordering used internally can be viewed as a logical extension of this proposal.

Ghosh *et al.* [9] present an analytical representation of cache misses for perfect loop nests, which they use to guide selected code optimization problems. Their work, like all of the other work cited above, relies on linear (row- or column-major) storage of arrays, and therefore does not immediately apply to our code.

6 Conclusions and Future Work

Matrix multiplication is an important computational kernel, and its performance can dictate the overall performance of many applications. Strassen's algorithm for matrix multiplication achieves lower arithmetic complexity, $O(n^{\log_2 7})$, than the conventional algorithm, $O(n^3)$, at the cost of worse locality of reference. Furthermore, since Strassen's algorithm is based on divide-and-conquer, an implementation must handle odd-size matrices, and reduce recursion overhead by terminating the recursion before it reaches individual matrix elements. These issues make it difficult to obtain efficient implementations of Strassen's algorithm.

In this paper we presented a practical implementation of Strassen's algorithm (Winograd variant) that exploits the ability to dynamically select the recursion truncation point based on matrix size and efficiently handles odd-sized matrices. We achieve this by using a non-standard array layout called Morton order; by converting from standard layouts (e.g., column-major) to internal Morton layout at the interface level; and by exploiting dynamic selection of the recursion truncation point to minimize padding.

We compare our implementation to two alternative implementations that use dynamic peeling (DGEFMM) [13] and dynamic overlap (DGEMMW) [6]. Execution time measurements on a DEC Alpha and a SUN UltraSPARC II reveal wide variability in the performance of all three implementations. On the Alpha, our implementation (MODGEMM) ranges from 30% slower to 20% faster than DGEFMM for matrix sizes from 150 to 1024. On the Ultra, MODGEMM is generally faster than DGEFMM for large matrices (500 and larger), while DGEFMM is generally faster for small matrices. When eliminating the time to convert matrices to/from Morton order (5% to 15% of total execution time), MODGEMM outperforms DGEFMM for nearly all matrix sizes on the Ultra, and for most matrices on the Alpha.

Our future work includes investigating techniques to further improve the performance and stability of Strassen's algorithm, while minimizing code complexity. We also plan to examine the effects of rectangular input matrices. Our implementation supports the same interface as Level 3 BLAS `dgemm` routine [2], we plan to examine its performance for a variety of input parameters.

References

1 S. K. Abdali and D. S. Wise.
Experiments with quadtree representation of matrices.
In P. Gianni, editor, *Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 96-108. Springer-Verlag, 1988.

2 E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Oustrouchov, and D. Sorenson.
LAPACK User's Guide.
SIAM, Philadelphia, PA, second edition, 1995.

3 D. H. Bailey.
Extra high speed matrix multiplication on the Cray-2.
SIAM Journal of Scientific and Statistical Computing, 9(3):603-607, 1988.

4 S. Coleman and K. S. McKinley.
Tile size selection using cache organization and data layout.
In *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 279-290, La Jolla, CA, jun 1995.

5 J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff.
A set of level 3 basic linear algebra subprograms.
ACM Trans. Math. Softw., 16(1):1-17, Mar. 1990.

6 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.

7 P. C. Fischer and R. L. Probert.
Efficient procedures for using matrix algorithms.
In *Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages

413-427. Springer-Verlag, 1974.

8 J. D. Frens and D. S. Wise.
Auto-blocking matrix-multiplication or tracking BLAS3 performance with source code.
In *PPoPP97*, June 1997.

9 S. Ghosh, M. Martonosi, and S. Malik.
Cache miss equations: An analytical representation of cache misses.
In *Proceedings of International Conference on Supercomputing*, pages 317-324, Vienna, Austria, July 1997.

10 N. J. Higham.
Accuracy and Stability of Numerical Algorithms.
SIAM, Philadephia, 1996.

11 M. D. Hill and A. J. Smith.
Evaluating associativity in CPU caches.
IEEE Transactions on Computers, C-38(12):1612-1630, Dec. 1989.

12 C. H. Huang, J. R. Johnson, and R. W. Johnson.
A tensor product formulation of Strassen's matrix multiplication algorithm.
Applied Mathematics Letters, 3(3):67-71, 1990.

13 S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull.
Implementation of Strassen's algorithm for matrix multiplication.
In *Proceedings of Supercomputing '96*, 1996.
http://www.supercomp.org/sc96.

14 IBM Corporation.
IBM engineering and scientific subroutine library guide and reference, 1992.
Order number: SC23-0526.

15 I. Kodukula, N. Ahmed, and K. Pingali.
Data-centric multi-level blocking.
In *PLDI97*, 1997.

16 A. Kreczmar.
On memory requirements of Strassen's algorithms.
In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science 1976*, number 45 in Lecture Notes in Computer Science, pages 404-407. Springer-Verlag, 1976.

17 M. S. Lam, E. E. Rothberg, and M. E. Wolf.
The cache performance and optimizations of blocked algorithms.
In *ASPLOS4*, pages 63-74, April 1991.

18 A. R. Lebeck and D. A. Wood.
Cache profiling and the SPEC benchmarks: A case study.
IEEE COMPUTER, 27(10):15-26, October 1994.

19 V. P. Pauca, X. Sun, S. Chatterjee, and A. R. Lebeck.
Architecture-efficient Strassen's matrix multiplication: A case study of divide-and-conquer algorithms.
In *International Linear Algebra Society(ILAS) Symposium on Algorithms for Control, Signals and Image Processing*, 1997.

20 J. R. Pilkington and S. B. Baden.
Dynamic partitioning of non-uniform structured workloads with spacefilling curves.
IEEE Transactions on Parallel and Distributed Systems, 7(3):288-300, Mar. 1996.

21 G. Rivera and C.-W. Tseng.
Data transformations for eliminating conflict misses.
In *Proceedings of SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 38-49, Montreal, Canada, jun 1998.

22 A. Srivastava and A. Eustace.
Atom a system for building customized program analysis tools.
In *PLDI94*, pages 196-205, June 1994.

23 V. Strassen.
Gaussian elimination is not optimal.
Numerische Mathematik, 13:354-356, 1969.

24 D. S. Wise and J. Franco.
Costs of quadtree representation of nondense matrices.
Journal of Parallel and Distributed Computing, 9(3):282-296, July 1990.

Author Biographies

Mithuna Thottethodi is a graduate student in the Department of Computer Science at Duke University. He is currently involved in the TUNE project and his research interests include techniques for improved memory hierarchy performance. He received his B.Tech. (Honors) in Computer Science and Engineering in 1996 from Indian Institute of Technology, Kharagpur. He is a member of ACM.

Siddhartha Chatterjee is an assistant professor of Computer Science at the University of North Carolina at Chapel Hill, where he is co-director of the TUNE project. He received his B.Tech. (Honors) in electronics and electrical communications engineering in 1985 from the Indian Institute of Technology, Kharagpur, and his Ph.D. in computer science in 1991 from Carnegie Mellon University. He has written several papers in the areas of compilers for parallel languages, computer architecture, and parallel algorithms. His research interests include the design and implementation of programming languages and systems for high-performance scientific computations, locality management in hierarchical computations, and parallel algorithms and applications. He is a member of ACM and IEEE.

Alvin R. Lebeck is an assistant professor of Computer Science and of Electrical and Computer Engineering at Duke University, where he is co-director of the TUNE project. His research interests include hardware and software techniques for improved memory hierarchy performance and efficient messaging systems for distributed operating system services. Lebeck received the BS in Electrical and Computer Engineering, and the MS and PhD in Computer Science at the Universiy of Wisconsin--Madison. He is a member of ACM and IEEE.

About this document ...

Tuning Strassen's Matrix Multiplication for Memory Efficiency

The command line arguments were:
latex2html `-split 0 -show_section_numbers -html_version 3.2 root.tex`.

The translation was initiated by Alvin R. Lebeck on 8/6/1998

Footnotes

...Efficiency
This work supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637 and CDA-95-12356, Duke University, and an equipment donation through Intel Corporation's Technology for Education 2000 Program. Chatterjee is partially supported by NSF CAREER Award CCR-95-01979. Lebeck is partially supported by NSF CAREER Award MIP-97-02547. A Postscript version of this paper is available here.

Next Up Previous

Alvin R. Lebeck
8/6/1998