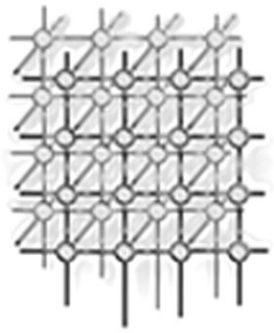
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2002; **14**:805–839 (DOI: 10.1002/cpe.630)



A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels

Vinod Valsalam and Anthony Skjellum*,†

High Performance Computing Laboratory, Department of Computer Science, Mississippi State University, MS 39762, U.S.A.

SUMMARY

Despite extensive research, optimal performance has not easily been available previously for matrix multiplication (especially for large matrices) on most architectures because of the lack of a structured approach and the limitations imposed by matrix storage formats. A simple but effective framework is presented here that lays the foundation for building high-performance matrix-multiplication codes in a structured, portable and efficient manner. The resulting codes are validated on three different representative RISC and CISC architectures on which they significantly outperform highly optimized libraries such as ATLAS and other competing methodologies reported in the literature. The main component of the proposed approach is a hierarchical storage format that efficiently generalizes the applicability of the memory hierarchy friendly Morton ordering to arbitrary-sized matrices. The storage format supports polyalgorithms, which are shown here to be essential for obtaining the best possible performance for a range of problem sizes. Several algorithmic advances are made in this paper, including an oscillating iterative algorithm for matrix multiplication and a variable recursion cutoff criterion for Strassen's algorithm. The authors expose the need to standardize linear algebra kernel interfaces, distinct from the BLAS, for writing portable high-performance code. These kernel routines operate on small blocks that fit in the L1 cache. The performance advantages of the proposed framework can be effectively delivered to new and existing applications through the use of object-oriented or compiler-based approaches. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: matrix multiplication; hierarchical matrix storage; Morton order; polyalgorithms; Strassen's algorithm; kernel interface

*Correspondence to: Anthony Skjellum, Department of Computer Science, Mississippi State University, P.O. Box 9637, Mississippi State, MS 39762, U.S.A.
†E-mail: tony@cs.msstate.edu

Received 1 June 2001
Revised 21 November 2001

1. INTRODUCTION

Multiplication of dense matrices is an operation that is extensively used in scientific computing applications and often accounts for a significant portion of the total execution time. Therefore, making matrix multiplication run faster and more predictably[‡] can go a long way towards improving such applications. Because of its prominent role in scientific computation, considerable work has been done to improve the performance of matrix multiplication. However, more improvements in performance are available despite widespread attention to this basic operation and the linear algebra based thereon.

This paper illustrates ways to obtain such added performance, especially for large matrices, through contributions in the areas of storage format, algorithms and kernels. This work also widens the range of applicability of Strassen multiplication by improving its performance and making the performance smoother as a function of problem size. The hierarchical storage format that is proposed here allows such disparate matrix-multiplication algorithms as iterative, recursive and Strassen to be unified under a common high-performance framework.

It needs to be noted here that the low-level kernels (that perform block products) used in this work were written using a semi-automated scheme that builds the kernels using C-language macros. This approach facilitates rapid generation of kernels that are reasonably well optimized based on certain processor features such as the number of floating point registers. The scheme relies heavily on the optimizing and instruction scheduling capabilities of the compiler to produce good kernels. Data prefetching is not used in the kernels. Nevertheless, the quality of the kernels thus obtained, combined with the efficacy of the other techniques described in this paper, is sufficient to produce matrix-multiplication codes that are faster than widely used high-performance libraries such as ATLAS. Because of the reasons previously mentioned, the authors are convinced that even better performance can be obtained by refining the kernel generation scheme to incorporate additional optimization techniques that exploit sophisticated processor-specific features, reduce L1 cache misses and perform prefetching.

1.1. Background

Early efforts in optimizing matrix algebra on computers with hierarchical memory led to the development of blocked computations as a means to improve data locality [1–9]. Matrix multiplication benefits significantly from this technique known as blocking or tiling. High-performance implementations of Level 3 BLAS [10,11] such as the implementations provided by platform vendors and ATLAS [12,13] use blocked algorithms to obtain good performance. Some new treatments of blocking for matrix multiplication can be found in Emmerald [14] and ITXGEMM [15]. Many optimizing compilers now use blocking as a standard code transformation technique to optimize certain loop nests (e.g. SGI's MIPSpro compilers [16]).

More recently, there has been interest in alternative storage formats to address the issue of locality. The default row/column-major order used by programming languages such as C and FORTRAN to store matrices limit locality to a single dimension, with built-in array types. Therefore, recursive

[‡]Predictability here refers to having a small upper bound on the variation in performance (Mflops) with respect to matrix size.

patterns (space-filling curves [17]) such as Morton order, that possess locality in both dimensions, have been proposed to store matrices. Frens and Wise [18] used a recursive matrix-multiplication algorithm in conjunction with a matching recursive array layout and demonstrated the beneficial effects of the latter for large matrices that force the system to page. Chatterjee *et al.* [19] improved on Frens and Wise's method by stopping recursion at the level of blocks that fit in a cache. Gustavson *et al.* [20–23] devised recursive blocked storage formats (and also recursive packed formats for triangular matrices) and developed matching recursive dense linear algebra algorithms for BLAS operations and LAPACK routines. Their results show that the recursive algorithms are faster than the standard codes for large matrices. Gustavson also described non-recursive blocked (full and packed) storage formats in [24] which are variations of the 4D array layout discussed earlier by Chatterjee *et al.* in [25].

As is well known, optimizing code to match the processor architecture is as essential for obtaining good performance as ensuring good locality properties for the algorithm. Evidence for this could be found in the low-level kernels of high-performance libraries such as ESSL [7], ATLAS [12] and Emmerald [14]. Optimal values of several platform-dependent parameters such as block sizes and loop unrolling depths must be used to maximize performance of matrix-multiplication code on a given platform. Various factors such as the processor architecture and the number and sizes of the different levels of cache must be considered, which necessitates hand-tuning of the code for each platform. Therefore, hand-optimized matrix-multiplication routines, sometimes requiring assembly level coding, are usually provided by the platform vendor as part of the native BLAS implementation or similar math library kernels.

Automatic code generation systems such as PHiPAC [26,27] and ATLAS [12,13,28] seek to provide an alternative to vendor-supplied `DGEMM` by performing some of the platform-dependent optimizations at install-time. The resulting code is usually competitive with vendor-supplied code. These systems search the design space to find the best values for the machine-dependent parameters on the given platform.

With the adoption, in some settings, of the C++ language for writing high-performance scientific applications, there have been recent attempts at using template programming to construct portable performance oriented linear algebra libraries. The Matrix Template Library (MTL) [29–31] and the Parallel Mathematical Library Project (PMLP) [32–34] are examples.

Strassen's algorithm for matrix multiplication [35] has attracted some attention because of its lower arithmetic complexity. The IBM Corporation evidently uses the algorithm in their ESSL library [36,37]. Concerns about its numerical properties [38,39] and its apparent poor locality and large temporary storage requirement abated the enthusiasm. Recently, research done with the aim of addressing the locality and temporary storage issues has yielded some positive results [39–42]. Chatterjee *et al.* also explored the use of Strassen with their recursive data layouts and obtained good performance [25,43]. The research presented in this paper enhances the performance characteristics of Strassen and thereby improves its usability.

1.2. Contributions

As indicated earlier, a large quantity of research has been devoted in diverse areas to optimize matrix multiplication. The contributions that are offered in this paper present themselves as a comprehensive strategy to enable development of high-performance matrix-multiplication codes and are summarized here.

1. A new conceptual framework is proposed for writing high-performance polyalgorithmic matrix-multiplication routines using advanced storage formats and optimized processor-specific kernels.
2. The techniques used by the authors enable their code to outperform rigorously optimized widely used linear algebra libraries such as ATLAS on all platforms tested and by a significant amount on most. The performance is also better than the results of the recursive method of Wise as reported in [44] and combinations of recursive methods and `DGEMM` used by others such as Chatterjee *et al.* [25]. The performance is comparable to that of Gustavson's method [20] on two of the platforms and better on the third.
3. A hierarchical matrix storage format is devised that incorporates Morton order and handles arbitrary-sized matrices efficiently. This storage format, while providing the benefits of improved locality of Morton ordering, can be implemented without extra memory or computation on zero padding normally required by some other recursive techniques.
4. Comparative evaluations of the performance of different algorithms reveal the importance of a polyalgorithmic[§] approach to achieve the best performance for a given matrix size and platform. Similar results have been reported in the literature for both sequential [15] and parallel [45,46] matrix multiplication.
5. Contrary to statements found in the literature such as those by Gustavson *et al.* [20] and Chatterjee *et al.* [25], the studies presented herein demonstrate that iterative algorithms, when used with an efficient Morton order location code calculation algorithm, are highly competitive with the recursive ones and even faster in certain cases. This fact bolsters the need for polyalgorithmic libraries.
6. Significant further performance enhancements can be achieved with the use of Strassen's algorithm, which is shown to work well with the hierarchical storage format. The authors' Strassen code has better performance characteristics than that reported by others in the literature on similar machines[¶] (e.g. Chatterjee *et al.* [25]). The use of better low-level kernels for block products in the Strassen code could yield even greater performance.
7. An assortment of new techniques and algorithms[‖] are presented here that can easily be incorporated in the hierarchical framework to improve the performance of matrix multiplication.

   (a) The oscillating iterative algorithm that is described here has the two-miss property that improves cache performance.
   (b) A strategy is presented for checking index bounds inside block products that enables the recursive algorithm of Frens and Wise [18] to eliminate unnecessary computation.

[§]The term *polyalgorithm* was introduced by Professor John Rice and refers to the choice of one suitable algorithm from a set of candidate algorithms, all designed to solve the same problem, with the aim of obtaining the best possible performance in a given situation. This is not to be confused with the combined, simultaneous use of several algorithmic techniques, especially in the low-level kernel routines, for optimizing performance.

[¶]Comparisons with some published results could not be made because those results were obtained on very different architectures such as Cray and IBM machines (e.g. Agarwal *et al.* [47] and Douglas *et al.* [40]).

[‖]Evidently, variations of these algorithms could have been independently discovered and put to use by others. However, the authors could not find them in the published literature.

(c) A variable block size (recursion cutoff point) determination technique is introduced for Strassen's algorithm that produces performance curves that are optimal and have minimal fluctuations than the results found in the literature (e.g. Chatterjee *et al.* [25]).

8. The need for a standard kernel interface, distinct from the BLAS, is identified so that optimized kernels tuned to specific processors can be made available to library developers for writing portable high-performance code. These kernel routines operate on small blocks that fit in the L1 cache.
9. As elaborated in Section 6, an obvious path is shown to exist for either object-based libraries or compiler-based approaches to deliver the high performance guaranteed by the hierarchical formulation to new and existing applications. Thus, the performance enhancements may be made readily available to *legacy* applications in many cases.

1.3. Organization of the paper

The remainder of the paper is organized as follows. The approach adopted here, leading to the performance framework that incorporates the hierarchical storage format, algorithms and optimized kernels is described in Section 2. The hierarchical storage format and its variants are elaborately explained in Section 3 after discussing the drawbacks of other recursive formulations found in the literature. Section 4 deals with the various matrix-multiplication algorithms considered here and the authors' contributions to their enhancement. The importance of processor-specific optimized kernels in building high-performance linear algebra codes and the need for standardizing their interfaces is discussed in Section 5. Section 6 explains how an object-oriented design strategy or a compiler-based approach can make the performance benefits of the concepts presented here readily available to users of linear algebra routines. In Section 7, performance results on three different architectures are shown, demonstrating the advantages of the hierarchical formulation in comparison to other competing methods and the relative merits of various matrix-multiplication algorithms. Finally, in Section 8, conclusions drawn from the present research and further, future work possibilities are discussed.

2. APPROACH

As is well known, CPU and memory form the two integral elements involved in performing computations. Optimal performance can be achieved only by utilizing both the memory hierarchy and the CPU efficiently. An application makes efficient use of the memory if it has an optimal schedule for transferring data to and from the CPU across the different levels of the hierarchy to achieve minimal wasted CPU cycles. Once the data required for computation are brought close to the processor (cache or registers), the application must ensure that the CPU resources are used to the fullest extent so that the computations are completed in a minimum number of CPU cycles.

The main hypothesis behind the performance framework presented here is that the impact of memory hierarchy and CPU on the performance of blocked linear algebra algorithms can be separated out and dealt with orthogonally without compromising performance. The validity of this approach is apparent from the organization of the memory hierarchy and the interaction of the CPU with it when examined in the context of the nature of the algorithm that is used. The CPU obtains data for processing from

the L1 cache, which is the closest and fastest level of the memory hierarchy. This is the only direct and immediate interaction the CPU has with the memory hierarchy. Thus, the L1 cache forms the data interface between the CPU and the rest of the memory hierarchy. Now, consider a block-based linear algebra algorithm in which the blocks are sized to fit in the L1 cache. The algorithm can be logically viewed as consisting of two tiers, the top tier forming the memory hierarchy component and the bottom tier forming the CPU component. The bottom tier operates on blocks that are assumed to be resident in the L1 cache, streaming data into the CPU, keeping all the execution units in the processor as busy as possible and making maximum use of all the available processor features. When the computation on the current set of blocks is finished, control passes to the top tier which selects the next set of blocks for computation. The block selection criterion strives to optimize the overall performance of the memory hierarchy by maximizing both spatial and temporal locality of block accesses and minimizing the potential for cache conflicts. In other words, the memory hierarchy tier attempts to minimize the number of cache misses and page faults. Because of the blocked nature of the computations the functions of the CPU and the memory hierarchy tiers of the algorithm do not interfere with each other, allowing them to be treated independently. The separation of CPU and memory concerns in this manner permits the development of a convenient framework for writing high-performance linear algebra codes that offer maximum flexibility in terms of algorithmic choice, implementation and portability as elaborated further in Section 6.

The hierarchical storage format, described in detail in Section 3, along with the particular algorithm that is applied on the blocks form the memory hierarchy tier of the performance framework. Because of its two-dimensional locality, the hierarchical storage format provides algorithms with a means to store and access matrices in memory without causing too many cache misses or page faults. This leads to efficient utilization of memory if the algorithms also possess good locality in referencing the blocks. The hierarchical storage formulation allows the use of different matrix-multiplication algorithms such as iterative, recursive and Strassen on the blocks. The best block-algorithm can be chosen and used on a given machine for a given problem size (polyalgorithm friendly). The CPU tier then performs the standard iterative matrix-multiplication operation on the blocks. The code for the block-products is embodied in what is referred to as the kernel, which is optimized for the particular processor. The kernel ensures that all the resources of the processor are efficiently utilized by optimally scheduling operations in the block product.

The proposed approach significantly improves on the current technology based on single or multi-level blocking and kernels that is used in most commonly available high-performance matrix-multiply implementations. The new framework presented here combines several advantages, including the ability to choose a suitable storage format and algorithm that provides the best performance for the given platform and problem size. The ability to use the hierarchical storage format and its variants becomes increasingly important for large matrix sizes because of its good locality properties (see results in Section 7). Portability of the code is also enhanced by restricting the use of machine-dependent parameters to the bottom tier, which consists of highly optimized processor-specific kernels. In this way, the performance and flexibility afforded by the proposed framework enhances its utility over other approaches found in the literature such as the ones used by ATLAS [12] and the Algorithms and Architecture approach described by Agarwal *et al.* [7] for the POWER2 processor.

The basic ideas of the proposed framework such as the hierarchical storage format, efficient algorithms and kernel design are described in the following sections. The concepts are demonstrated by realizing high-performance matrix-multiplication codes in the C language. Comparisons with related

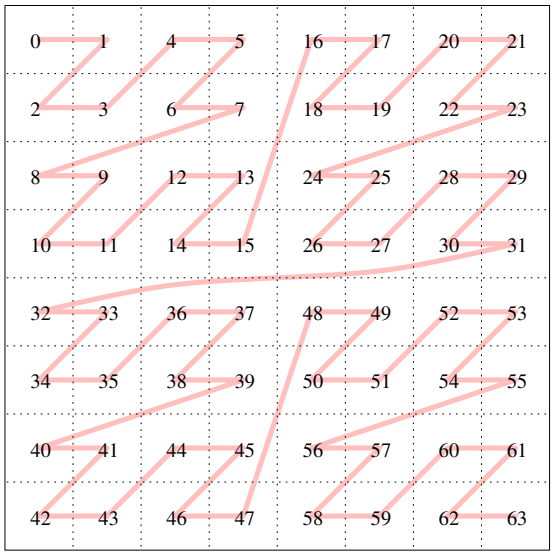


Figure 1. Morton (Z) order.

techniques reported in the literature are provided wherever relevant. The matrix-multiplication codes implemented under the proposed framework outperform rigorously optimized linear algebra libraries such as ATLAS on all platforms tested and by a significant amount on most. The performance is also better than the combinations of recursive methods and optimized `DGEMM` used by others such as Chatterjee *et al.* [25] on all platforms that were studied.

3. HIERARCHICAL MATRIX STORAGE

Algorithmic blocking [1–9] is a well known technique for improving locality that works extremely well for level 3 BLAS operations such as matrix multiplication. Blocking can be applied to each level of the memory hierarchy to improve the locality at each level. However, an alternative effective strategy is to address the poor spatial locality inherent in the one-dimensional nature of row/column-major storage format and use an alternative format that possesses locality in both dimensions. Recursive/nonlinear data layouts that impart two-dimensional locality to matrices have been shown to provide significant performance advantages for matrix multiplication in [18,19] and also by the authors in [48]. The recursive layouts are most effective when multiplying large matrices that force the system to page and incur TLB misses, but are also useful for smaller problems.

A recursive data layout based on a space-filling curve [17] known as Morton (Z) order (see Figure 1) is used in the proposed storage format. Other recursive orderings based on space-filling curves such as U, X, Gray and Hilbert orders were studied by Chatterjee *et al.* [19] along with Z-order in the context of matrix multiplication. All these orderings were found to have similar performance characteristics

for both the standard and Strassen's algorithms for matrix multiplication. Morton order was chosen to be used here because of its relative simplicity in calculating location codes (addresses), compared with the other orderings. Calculation of location codes for Morton order is more involved than that for the traditional row/column-major order. If the full benefits of improved locality offered by Morton order are to be reaped an efficient algorithm must be used for location code calculation. An algorithm described in [49] and also discussed by the authors in [48] is used for fast incremental address computation of Morton ordered matrices within iterative algorithms.

A drawback of using Morton order storage is that a straightforward application of the ordering is possible only for square matrices whose sides are an integer power of two. Other matrix sizes require special treatment. A common way of handling arbitrary-sized matrices is to apply padding such that the padded matrix can be subjected to Morton ordering. Wise [44] and Chatterjee *et al.* [19] have suggested different schemes based on padding for their recursive orderings. Both schemes have their own disadvantages from a performance standpoint as described in Section 3.1. The hierarchical data layout described here is designed to overcome these drawbacks in applying Morton ordering to arbitrary-sized matrices, bringing new opportunities for performance enhancement and practical use of the algorithms in scientific codes.

3.1. Related work and drawbacks

The quadtree representation of recursive matrix storage used by Frens and Wise [50] allocates extra memory to handle arbitrary sized matrices. Depending on the aspect ratio of the matrix the extra memory allocated could be as high as 78% (see [50]), but the extra space is not used. The authors of that work argue that the allocated, but unused memory does not affect performance because it is never fetched to the faster levels of the memory hierarchy. This works reasonably well for systems with virtual memory, but cannot be used in systems that do not have virtual memory such as most embedded computers, or even systems such as Sandia/DOE Cplant [51]. Frens and Wise's method requires bounds checking on the matrix rows and columns to bypass the padded elements, which could turn out to be expensive if continued down to the leaf nodes of the quadtree. Therefore, bounds checking is stopped at the level of, say, 8×8 blocks and any padded elements in the blocks are made to participate in the overall computation after initializing them appropriately so that the net result is not affected. The computation on padded elements compromises performance.

Chatterjee *et al.* [19] choose blocking factors from an architecture-dependent range and apply enough padding to the matrix such that the blocking factor exactly divides the padded matrix side into an integer power of two. They use the recursive algorithm proposed by Frens and Wise with their storage format. The padded elements are initialized to zero and computation performed on them blindly. Computation on the padded portions of the matrix affects performance severely, especially when the padding is large, and leads to the saw-toothed performance graphs seen in Section 7.2.

The recursive blocked data format and the matching recursive matrix multiplication algorithm used by Gustavson *et al.* [20] requires padding the matrices to make them evenly divisible by the blocking factors, but the padded elements are left out of the computation. The recursive block row (RBR) format, which is one of the recursive block orderings that they use, defaults to Morton ordering of blocks for matrices containing a power-of-two number of blocks. Their multiplication algorithm generates a binary recursion tree, whereas Frens and Wise's algorithm produces an eight-ary recursion tree. It is not apparent if the memory hierarchy friendly two-miss characteristic of Frens and Wise's recursive

algorithm can be incorporated into Gustavson's formulation. The performance of Gustavson's method is compared with the other methods in Section 7.2. From an algorithmic standpoint, the recursive blocked data format does not easily facilitate the use of any iterative methods for computing the matrix product unless tables are used. However, Strassen's algorithm can be used in those levels of the recursion that yield square matrices with even sides.

This discussion assumed that block addresses are computed dynamically from the recursive layout pattern. An alternative is to make use of tables to store the address information as suggested by Gustavson *et al.* [20]. This overcomes many of the drawbacks mentioned earlier. However, tables require extra bookkeeping effort and the associated overhead may significantly affect performance for large matrices. Since one table entry is required for each block, the table may grow undesirably large as the matrix size increases. For example, consider a typical case from the results for the Pentium III configuration discussed in Section 7. Assuming a blocking factor of 40** for both dimensions a table containing 10 000 entries is required to store a 4000×4000 matrix. The hierarchical storage formulation developed here avoids the use of tables, while minimizing the computation required to calculate block addresses.

Tables can also be eliminated if the blocks are stored in row/column order. Chatterjee *et al.* presents such a format as the 4D layout in [25]. Similar formats have also appeared recently in Gustavson's work [24]. However, using these formats results in the loss of the good locality properties of recursive layouts such as Morton order. In other words, the automatic blocking provided by the recursive storage formats for every level of the memory hierarchy [18,21] is no longer available. For small matrices that are easily accommodated in the lower levels of the memory hierarchy the manual blocking used in the blocked row/column storage formats is sufficient to afford performance comparable to the recursive formats as demonstrated by Chatterjee in [25], where results comparing the 4D layout with Morton order are presented for matrices of order up to around 1000. Gustavson does not show any performance results in [24]. Larger matrices, which are also included in the domain of the current study because of their dominance in large-scale computing problems, will benefit from the automatic blocking for the deeper levels of the memory hierarchy provided by the recursive storage formats. The Morton ordering used in the hierarchical storage format ensures that this desirable feature is available to provide steady, undiminished and predictable performance even for large matrices as shown by the results in Section 7.

3.2. Construction

The hierarchical storage format designed in this paper overcomes the limitations of the methods discussed in Section 3.1 in extending the applicability of Morton order to arbitrary-sized matrices. The key observation that leads to the hierarchical format is the fact that any integer can be expressed as a unique sum of powers of two. Using this fact, any matrix can be decomposed into square power-of-two sized submatrices as explained below. The square submatrices can then be subjected to Morton ordering individually.

The hierarchical storage format involves four levels as shown in Figure 2. The matrix is comprised of submatrices at each level. The submatrices are grouped and ordered to form bigger submatrices in the

**The I, J, K blocking factors used for the PIII are 40, 32, 32, respectively, for some algorithms.

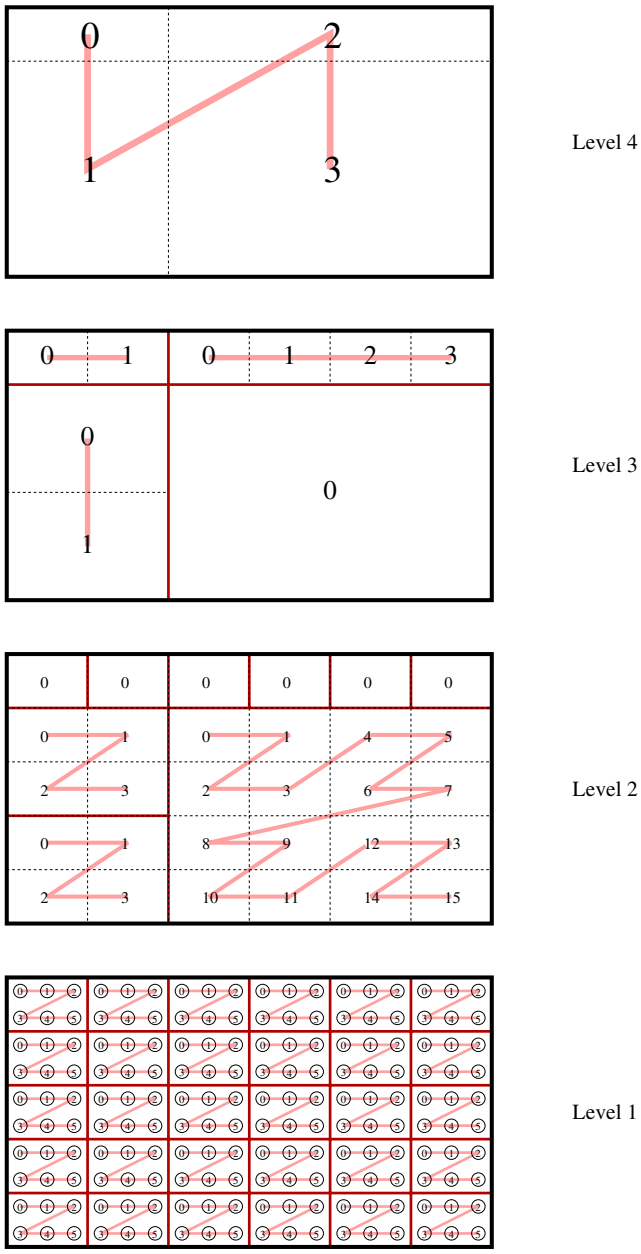


Figure 2. An example construction of the four levels of the hierarchical storage format for matrices showing the ordering of submatrices in the different levels.

next higher level. At the lowest level, the matrix is divided into blocks, inside which the elements are arranged in row-major order (column-major order could also be used, instead). The blocks are all of the same size. The size of the block is determined by cache and algorithmic considerations, as explained below. For the present discussion, it is assumed that the blocks divide the matrix exactly.

The blocks of level-one are the elements for the level-two matrix. The level-two matrix is decomposed into a minimum number of square submatrices that are powers of two in size. The level-one blocks are now Morton ordered inside these square submatrices. The decomposition is done in such a way that the smallest submatrices are on the top-left corner of the level-two matrix. The submatrix size increases in both directions, with the largest submatrices being found on the bottom-right corner. A different decomposition that reverses the progression of submatrix sizes inside the level-two matrix from one corner to the other is also possible.

The next two higher levels in the hierarchy help to uniquely organize the Morton ordered submatrices of the second level within the matrix boundaries. In the third level, the sides of the level-two matrix are expressed as sums of powers of two. Imagine lines being drawn between opposite sides of the matrix at each power of two. These vertical and horizontal lines split the matrix into tiles, whose sides are powers of two. Note that these tiles need not necessarily be square. The square submatrices from the second level fit snugly in these tiles. There is only one arrangement of the submatrices possible inside a tile: the submatrices form either a row or a column inside the tile.

At the fourth and topmost level, the tiles of the third level are arranged in column-major order. This completes a unique specification of the ordering of the elements of the original matrix. The choice of using column-major order instead of row-major order for the fourth level is arbitrary.

The construction of the four levels of the hierarchical storage format is demonstrated by means of an example in Figure 2. The full lines inside the matrix indicate submatrices in the current level while dashed lines indicate submatrices in the lower level. The numbering denotes the ordering of the elements inside the submatrices. The matrix has 10 rows and 18 columns and is divided into blocks of size 2×3 at the first level. This makes a 5×6 level-two matrix, which yields square submatrices as shown in the figure—six 1×1 submatrices form the first row followed by two 2×2 submatrices and one 4×4 submatrix below it. Inside each submatrix, the blocks are arranged in Morton order. For level-three, the sides of the level-two matrix are written as sums of powers of two ($5 = 1 + 4$ and $6 = 2 + 4$) and lines drawn across the matrix joining opposite sides at each power of two. The lines split the matrix into four tiles inside which the submatrices of level-two are arranged as a row (top tiles) or as a column (bottom tiles). The fourth and final level of the hierarchy places the four tiles from level-three in column-major order. The final ordering of matrix elements in memory is shown in Figure 3.

Blocking done at the first level has several advantages. First, it enables temporal locality to make good utilization of the cache. Second, for algorithms such as recursive and Strassen it provides a means to specify a cutoff point for recursion. Carrying recursion further down than the cutoff point can hurt performance. Third, blocking at this level allows a processor-specific multiplication kernel with custom optimizations to be used. Fourth, it reduces the overhead of using the hierarchical storage format. The complexity of the hierarchical format imposes a certain amount of overhead for address calculation. Blocking at level-one increases the granularity of the ordering for the higher levels and thereby reduces the number of complicated address calculations that are required by a significant amount.

When accessing a matrix stored in the hierarchical format the different levels are traversed top-down. Each level can be implemented as a set of two loops, one for each of the two dimensions of the matrix.

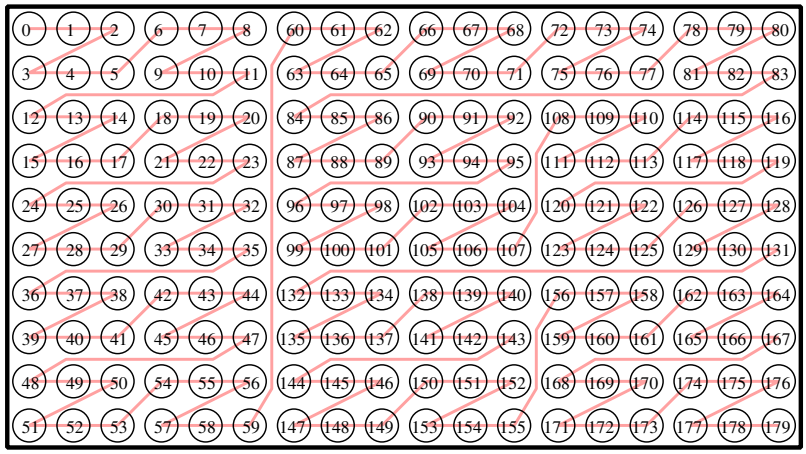


Figure 3. An example construction of the four-level hierarchical matrix storage format showing the ordering of matrix elements in memory.

The lower levels are nested within the upper levels. The loop indices are manipulated to calculate the addresses of the matrix elements. For the Morton ordered level, the fast incremental address (location code) calculation algorithm discussed in [48] is used.

So far in this discussion it has been assumed that the matrix size is evenly divisible by the block size of the first level. If this is not the case, some adjustments are made for the hierarchical storage format to be applied. Two different schemes to handle odd-sized matrices are now described.

3.3. Variant 1

An obvious way of handling matrices that are not exactly divisible by the block size is by padding. The matrix is padded up to the next size that is evenly divisible by the block size. The padding used here is better than the methods used by Chatterjee *et al.* and Frens and Wise in two respects: (a) it uses little extra memory, and (b) the algorithms here do not perform computation on the padded elements and, therefore, do not adversely affect performance.

Let T be the block size and M, the matrix size. In the hierarchical storage format, the ratio of the amount of padding to the matrix size will not exceed the ratio of the block size to the matrix size (T/M). Therefore, the pad-to-matrix size ratio decreases with increase in matrix size. Chatterjee's recursive array layouts require a maximum pad-to-matrix size ratio equivalent to the inverse of the minimum block size $(1/T)$. The ratio remains a constant with respect to the matrix size. In other words, if one side of a matrix is held constant and the other side increased in length, the maximum amount of padding required for the hierarchical ordering remains a constant whereas the maximum padding for Chatterjee's method would increase proportionally to the matrix size. If Frens and Wise's quadtree representation is used, the padding could be as high as 78% of the matrix size, as discussed in Section 3.1.

As mentioned in Section 3.1, both Chatterjee *et al.* and Frens and Wise perform computation on padded elements, thereby degrading performance. The proposed method employs bounds checking on matrix indices, thereby eliminating the need for any unnecessary computations on padded elements. The hierarchical formulation allows the use of different matrix-multiplication algorithms. If the iterative algorithm is used, detecting the matrix edges to avoid touching the padded area is straightforward and efficient. However, an efficient way for checking bounds inside the recursive algorithm is not easily apparent, which is presumably the reason why it was not done by Chatterjee or Frens and Wise. A low overhead bounds checking technique for the recursive algorithm is described in Section 4.2, which makes this variant of the hierarchical format viable for use with the recursive algorithm. It must be noted here that Frens and Wise use a bounds checking algorithm in their quadtree formulation to prune empty branches of the tree which must not be confused with the present requirement to detect matrix edges inside the Morton ordered blocks. Strassen's algorithm is not implemented with this variant of the hierarchical ordering because avoiding computation on the padded elements becomes extremely complicated.

3.4. Variant 2

The second variant of the hierarchical storage format does not use any padding for odd-sized matrices. It can be efficiently used with any matrix multiplication algorithm including Strassen. In this variant, the extra rows and columns at the end of the matrix are stripped off before the hierarchical storage format is applied. The peeled portions of the matrix are then subjected to the hierarchical ordering separately.

Peeling splits a matrix A into four submatrices: A_{11}, a_{12}, a_{21} and a_{22}. A_{11} is the main portion of the matrix left behind when the fringe submatrices, a_{12}, a_{21} and a_{22}, are removed by peeling. The hierarchical storage format is applied separately to each of the four submatrices and they are stored contiguously in the order A_{11}, a_{12}, a_{21} and a_{22}. Since the block size used in A_{11} is too big for the fringe submatrices, it is made smaller in the appropriate dimension(s) for each of a_{12}, a_{21} and a_{22} so that the blocks fit exactly in the submatrices. Since the fringe submatrices represent portions of the matrix stripped off by peeling the hierarchical storage format simplifies to a column of blocks for a_{12}, a row of blocks for a_{21} and a single block for a_{22}.

When two matrices are multiplied, the product matrix is computed as follows:

$$C = AB \tag{1}$$

$$\begin{bmatrix} C_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} B_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} C_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + a_{12}b_{21} & A_{11}b_{12} + a_{12}b_{22} \\ a_{21}B_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} \tag{3}$$

The elaborate matrix-multiplication algorithms described in Section 4 are used only for the computation of the main portion of the matrix product, namely $A_{11}B_{11}$. All the other submatrix products are computed by some fix-up code using the conventional blocked algorithm.

The two variants of the hierarchical storage format have different performance impacts on different architectures. On some architectures variant 1 performs better whereas on others variant 2 is better as seen in the performance results presented in Section 7.1.

4. MATRIX-MULTIPLICATION ALGORITHMS

A multiplication algorithm for matrices stored in the hierarchical format[††] consists of a set of three loops for each of the top two levels. The next level, which is composed of Morton ordered blocks, can use a variety of different matrix-multiplication algorithms. It is found that the same algorithm does not give the best performance for all platforms nor all matrix sizes on the same platform (see Section 7). This necessitates the adoption of a polyalgorithmic approach for high-performance matrix multiplication. The algorithms studied here are discussed in the following subsections, providing details of some of the enhancements offered by this work.

4.1. Iterative algorithm

It has been widely reported in the literature that a recursive algorithm should be used along with a recursive storage format to match the algorithm with the data layout in order to obtain maximum performance [18,20,25]. However, the authors' observations in this paper (Section 7.1) are contradictory, showing that the traditional iterative algorithm is highly competitive with the recursive one, and even better for certain (usually small) matrix sizes on some platforms.

One of the difficulties of using an iterative algorithm with Morton order is the complicated address calculation involved[‡‡]. A highly efficient incremental location code calculation algorithm for Morton order is used in this work to keep the overhead of address calculation to a minimum. The algorithm, involving algebra of dilated integers, is described by the authors in [48] and is also discussed by Schrack in [49]. If the dilation of the row index i and column index j of the matrix are known, then the Morton order location code l can be calculated as $l = 2D(i) + D(j)$, where D represents the dilation operation. The algorithm can incrementally determine the dilation of an integer using only two machine operations. In contrast, other methods for calculating dilation require many more operations and/or table lookups, resulting in significantly larger overhead for calculating addresses. For example, dilating a 16-bit integer using a 256-entry table requires six operations, including two loads [52]. For a 16-bit integer, the dilation algorithm presented by Stocco and Schrack in [52] requires at least 16 operations. The low-overhead address calculation method used here in the iterative matrix-multiplication algorithm is an important factor that contributes to its good performance.

4.2. Modified recursive algorithm

Computation of a block product involves bringing a block of each of matrices A, B and C closer to the processor in the memory hierarchy. It is always possible to compute the next block product by reusing one of the three blocks and loading the other two. The two-miss recursive algorithm of Frens and Wise [18] achieves this at all levels of recursion, thus making good use of the memory hierarchy. Their algorithm has been modified here to permit efficient bounds checking within level-one blocks to avoid computation on padded matrix regions.

[††]Conversion of matrices from row/column storage to the hierarchical format is not considered in this paper. Chatterjee *et al.* [25] have shown that such conversion costs are only about 2–5% of the total execution time for matrix multiplication.
[‡‡]Row/column major ordering has an advantage in this respect—address calculation is straightforward and efficient.

Frens and Wise and Chatterjee *et al.* used the algorithm without separating out computation on padded elements inside the block products, as was mentioned in Section 3.1. When the recursion tree is navigated down to the level of block products, information about the locations of the blocks inside the original matrices is lost. This information is required to identify and exclude padded elements from computation. From the perspective of the iterative algorithm, the values of the iteration variables i, j and k indices, must be known for each block product. The block indices when multiplied by the block sizes give the indices of the matrix elements, which permits bounds checking to be applied on the matrix elements.

The recursive algorithm multiplies square block-matrices. The number of blocks also needs to be a power of two to enable recursion to be carried down to the level of individual blocks. The algorithm proceeds by dividing the matrices into four equal quadrants and performing quadrant multiplications recursively. To each of the quadrant multiplication function calls, the three block indices that collectively identify the first blocks in the quadrants of A, B and C are passed. Figure 4 shows how the block indices i, j and k are determined for the eight quadrant products.

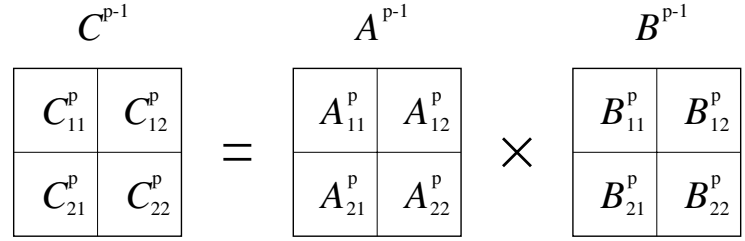
4.3. Oscillating iterative algorithm

An oscillating iterative algorithm is shown here as a modification of the regular iterative algorithm to incorporate the two-miss feature of the recursive algorithm. It always keeps one of the three blocks involved in a matrix product in cache between successive block products, thus improving locality over the regular iterative algorithm. Note that the two-miss strategy is optimal with respect to maximum utilization of the blocks that have been brought into cache. The fast address calculation method used in the case of the regular iterative algorithm is used here also to keep addressing costs as low as possible. The improved cache behavior of the oscillating iterative algorithm translates to better performance compared with the regular iterative algorithm and enables it to beat the recursive algorithm on certain platforms for most matrix sizes (see Section 7.1).

In the regular iterative algorithm the loop indices are always incremented. In other words, when the inner loop indices reach their upper limits, they are again set to the lower limits, from where they continue to be incremented. The oscillating iterative algorithm, on the other hand, increments and decrements the inner loop indices alternately between the lower and upper limits of iteration. The algorithm begins by incrementing the loop index as with the regular iterative algorithm. However, when the loop index reaches the upper limit of the loop, the iteration continues by *decrementing* the index. On reaching the lower limit of the loop, the increment–decrement cycle is repeated. This creates an oscillatory effect on the two inner loop variables. As a result, only one of the three loop indices is allowed to vary from the calculation of one product to the next. The functioning of the algorithm is illustrated in Figure 5. As seen from the figure, one of the blocks of matrices A, B or C is always reused between any two consecutive block products, improving cache utilization.

4.4. Strassen's algorithm

Strassen's algorithm for matrix multiplication is a divide and conquer approach and has a recursive structure [35]. Its lower arithmetic complexity of $\Theta(n^{\log_2 7})$ for the multiplication of two $n \times n$ matrices (compared with the $\Theta(n^3)$ complexity of traditional methods) makes it an attractive alternative to be considered. However, the reduced arithmetic complexity comes at the cost of increased memory usage



$$
\begin{aligned}
C^p_{11} &= A^p_{11}B^p_{11} \quad \langle i^p = i^{p-1}, \quad j^p = j^{p-1}, \quad k^p = k^{p-1}\rangle \\
&+ A^p_{12}B^p_{21} \quad \langle i^p = i^{p-1}, \quad j^p = j^{p-1}, \quad k^p = k^{p-1} + b\rangle \\
C^p_{12} &= A^p_{11}B^p_{12} \quad \langle i^p = i^{p-1}, \quad j^p = j^{p-1} + b, \; k^p = k^{p-1}\rangle \\
&+ A^p_{12}B^p_{22} \quad \langle i^p = i^{p-1}, \quad j^p = j^{p-1} + b, \; k^p = k^{p-1} + b\rangle \\
C^p_{21} &= A^p_{21}B^p_{11} \quad \langle i^p = i^{p-1} + b, \; j^p = j^{p-1}, \quad k^p = k^{p-1}\rangle \\
&+ A^p_{22}B^p_{21} \quad \langle i^p = i^{p-1} + b, \; j^p = j^{p-1}, \quad k^p = k^{p-1} + b\rangle \\
C^p_{22} &= A^p_{21}B^p_{12} \quad \langle i^p = i^{p-1} + b, \; j^p = j^{p-1} + b, \; k^p = k^{p-1}\rangle \\
&+ A^p_{22}B^p_{22} \quad \langle i^p = i^{p-1} + b, \; j^p = j^{p-1} + b, \; k^p = k^{p-1} + b\rangle
\end{aligned}
$$

Figure 4. Determination of the i, j and k block indices for the quadrant products in the recursive algorithm. At level p, there are b blocks in each dimension of a quadrant.

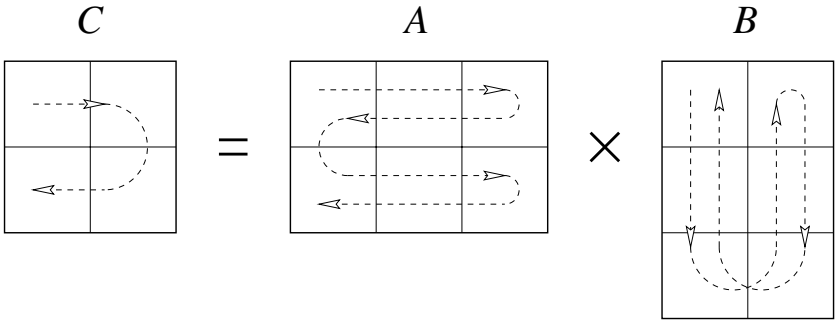


Figure 5. Access pattern of matrix elements for the oscillating iterative algorithm. The dashed arrows indicate the order in which elements are accessed when a 2×3 matrix is multiplied by a 3×2 matrix.

and poor algorithmic data locality. Moreover, since the algorithm is based on the multiplication of 2×2 matrices, it is not applicable to arbitrary matrix sizes in its pure form. Making the algorithm work for matrices of a general size would involve techniques such as padding or peeling that introduce additional overhead. Strassen's algorithm also has been shown to have poor numerical stability for certain matrix types that could limit its applicability [38,39,53]. Although the reference guide for the ESSL library [37] confirms this, according to anecdotal information that the authors received, ESSL produces results having better accuracy when Strassen is used in place of the standard algorithm for certain entries in the matrices.

Although Strassen published his algorithm in 1969 [35], it has since been largely ignored by people writing high-performance matrix-multiplication codes because of its drawbacks mentioned above. The IBM Corporation, who included the algorithm in the ESSL library [36], showed that good performance can be obtained if it is implemented carefully [47] and popularized its use on their systems. Recently, there has been a renewed interest in Strassen's algorithm and its variants, as a result of which several implementations have been developed that strive to minimize temporary storage requirement and improve locality [40,41,43]. Tensor product formulations of the algorithm have also been explored in this context for automatic optimizations [42] and implementation on parallel and vector machines [54].

The Strassen's algorithm implemented here is based on a variant due to Winograd, which uses fewer additions/subtractions than the original algorithm [55]. This implementation uses a computation schedule described by Huss-Lederman *et al.* [41,56] that minimizes the amount of temporary storage required. Strassen recursion is stopped well before it is carried down to the level of individual elements because of performance reasons. The block size of the matrix at level-one of the hierarchical storage format is set to be the same as the recursion cutoff point. The technique devised here for calculating the optimal recursion cutoff point is discussed in detail in Section 4.4.1. When the recursion stops, the conventional iterative algorithm is used to multiply the blocks, completing the computation of the final matrix product. When conventional multiplication is applied to the blocks, they are further tiled to obtain the best performance out of the cache.

The term *apparent Mflops* is used in this paper to measure the performance of Strassen's algorithm in comparison with the standard algorithm. Because of its lower arithmetic complexity, Strassen's algorithm performs fewer floating point operations than the conventional algorithm. Therefore, performance based on the number of floating point operations per second (Mflops) would not be a realistic comparison of execution speeds. To remedy this situation, the performance of Strassen is measured in terms of *apparent Mflops*, which is defined as the number of floating point operations of the standard algorithm divided by the execution time of Strassen. The apparent Mflops of Strassen can be compared with the true Mflops of the standard complexity algorithm to evaluate relative performance. Others have also used this approach in the literature under different names such as the *nominal Mflops* referred to by Agarwal *et al.* in [47].

4.4.1. Recursion cutoff point for Strassen

Stopping the recursion at an early stage instead of continuing it down to the level of individual matrix elements is imperative for obtaining good performance. The recursion cutoff criteria found in the literature (for example, in [41]) are determined empirically for row/column-major storage to be used with dynamic padding or peeling techniques and are not suitable for the present case. Moreover, the heuristic procedure presented here tunes the cutoff point to matrix size, in addition to machine

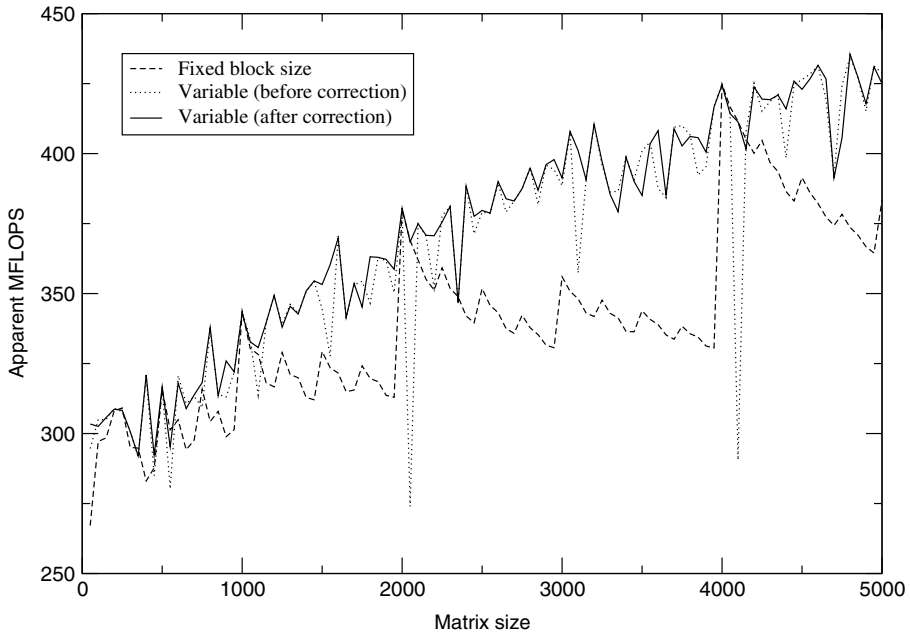


Figure 6. Performance of Strassen with fixed block size and variable block sizes before and after correction on an SGI R10k.

characteristics. The following discussion assumes that the matrices are square. If the matrices should not be square, the same procedure is simply repeated on all the sides.

The need to vary the cutoff point with respect to matrix size is clear from Figure 6 which shows the apparent Mflops of Strassen employing a constant cutoff point. The code was run on an SGI R10k machine, the details of which can be found in Section 7. The graph shows peaks repeating at exponential intervals. The peaks result from a sudden rise in performance when the number of blocks becomes a power of two followed by a gradual fall in the apparent Mflops as the number of blocks slowly diverge from the power of two. This peculiar behavior of Strassen is because of its quadrant recursive nature that requires matrices to be square and power of two for full applicability. When the number of blocks is a non-power of two, Strassen is applied to the constituent power-of-two block submatrices instead of the whole matrix. The advantage of Strassen comes from its reduced asymptotic arithmetic complexity, which is compromised when it is applied separately to smaller individual submatrices. In order to obtain maximum efficiency from Strassen, its coverage in a single invocation must be extended to as much of the matrix as possible. This can be done by making the number of blocks a power of two plus a remainder block, as is shown below. Since variant 2 of the hierarchical storage format is used, Strassen can be applied to the power-of-two part and the remainder blocks can be handled separately by conventional blocked multiplication.

A machine-dependent base block size s is first selected. Any integer matrix size $m \geq s$ obeys the relation $s \cdot 2^d \leq m < s \cdot 2^{d+1}$, where d is a non-negative integer. Therefore, a block size (cutoff point) c that varies between s and $2s$ can be obtained as $c = \lfloor m/2^d \rfloor$. If m is not evenly divisible by 2^d the remainder part of the matrix is handled separately by variant 2 of the hierarchical format. Strassen's algorithm can now be applied on the part of the matrix that has power-of-two number of blocks. An optimal value for s that provides the best performance characteristics is determined empirically.

The block size c determined using this procedure is insufficient as seen from the performance anomalies in Figure 6. Severe drops in performance occur when the block size is a power of two (e.g. 128) or a sum of relatively high powers of two (e.g. $96 = 64 + 32$), apparently because of conflict misses in the cache. In such cases the block sizes are decreased by correction factors determined empirically by experimentation. The correction factors are usually small integer numbers such as 1, 2 or 3. Performance can also be improved in some other cases, such as when the block sizes are certain odd or even numbers, by applying similar correction factors. The values of the correction factors are dependent on the architecture. In this way, the optimal block size c is determined as a function of matrix size and machine architecture. In most cases, it is possible to parameterize these characteristics and port the resulting heuristics to other architectures. The same heuristic procedure is valid on the SGI and Alpha platforms used for collecting results in Section 7. However, on the Pentium III a different heuristic needs to be used to determine the correction factors.

5. KERNEL DESIGN

Using advanced storage formats to squeeze significant performance out of the memory hierarchy is only one of the aspects involved in the construction of a high-performance code. Tuning the code to take advantage of the specific features available on a processor is also equally important, if not more. Processor-specific code is largely encapsulated in the matrix-multiply kernel that act on blocks of data from the bottom level of the hierarchical storage format and are designed to exploit full processor capabilities. This approach also increases the portability of the higher level code. Such kernel routines for multiplying fixed size blocks of data that fit in the L1 cache are evidently used in other matrix-multiply implementations also such as ATLAS [12]. Blocked implementations of other linear algebra operations also spawn similar low-level kernels that operate on cache-resident blocks of matrices. Processor architecture is becoming increasingly complex, each utilizing widely different technologies that makes development of an optimized kernel for each processor a tedious task. If a standard interface is specified for such linear algebra kernels, it would enable platform vendors and third-party software developers to supply optimized kernels, which could then be used by library developers in a portable fashion.

The BLAS `DGEMM/SGEMM` is evidently not the optimal building block for dense linear algebra performance-portable programming. It is a generalized form of matrix multiplication with options for scaling and accumulation that becomes an overhead if used as a low-level kernel. Its *fat interface* is loaded with parameters that are unnecessary and wasteful for multiplying two small matrices in a fixed storage format. Furthermore, it is designed with compromises since it handles matrices of a wide range of sizes, which may not be optimal for the small kernel sizes. This state-of-the-art consequently necessitates the design of further standard kernel interfaces for linear algebra operations to which vendors and library/application writers can conform in order to produce portable, performance-oriented

code with minimum effort. Research into the design of these linear algebra kernel interfaces and the specific techniques for their implementation is the subject of future work.

Although optimizing compilers often do a good job of tailoring code to specific processors, better performance can often be obtained by manually performing certain optimizations. Moreover, compilers sometimes fail to do even simple optimizations under certain conditions. For instance, when a three-loop matrix-multiplication code written in C is optimized using the SGI MIPSpro compiler (version 7.3) it generates blocked code with unrolled loops if the matrix sizes are known to the compiler. However, if the matrix sizes are hidden from the compiler it fails to perform these optimizations.

The Basic Linear Algebra Instruction Set (BLAIS) style abstractions, which are used in the present work, provide a convenient mechanism to construct the kernels. The BLAIS, also know as BLAS-Lite or Tiny BLAS is a language-independent specification that was proposed in the BLAS Technical Forum by the second author and colleagues, for the most basic, low-level operations in linear algebra to write high-performance kernels with useful portability [57–59]. The BLAIS macros represent RISC-type operations that act on fixed-size blocks that fit in a single line of cache. These operations are exposed as a result of unrolling the computation loops. When used with an optimizing compiler that has a good instruction scheduler, the BLAIS style macros can be very effective in building high-performance matrix-multiplication kernels.

The authors have only attempted to establish the importance of having a kernel that is optimized to specific processors. Aggressive optimizations similar to those done by ATLAS [12,28] have not been added yet, nor is hand-tuned assembly used, as is commonly done by vendors. Prefetching in the kernels is not yet performed. Kernel block sizes that have been chosen are probably not optimal either. Since enhanced performance is obtained with still further options for improvement in future work, the authors are convinced that the approach has immediate value.

The totality and integration of all these ideas constitute the new major contribution of this research. Many of these ideas are widely used in isolation without an overall structure needed for high performance and portability, such is described next.

6. INTEGRATION OF CONCEPTS

The hierarchical storage formulation provides an effective and concise framework to incorporate different matrix-multiplication algorithms and also a high-performance kernel that is optimized to particular processors. The software architecture of this matrix-multiplication framework is illustrated in Figure 7. The top two levels of the storage format are traversed iteratively to reach the Morton ordered submatrices, at which point the chosen multiplication algorithm can be applied. The block-level multiplications spawned by the previous level are performed by the optimized kernel at the lowest level of the hierarchical formulation.

The need to support different algorithms, even on the same machine, is evident from the performance results presented in Section 7. For example, Strassen performs less well than standard multiplication for small matrices, while the trend is reversed for large matrices. Similar behavior, although not as pronounced, is observable in the performance of the different algorithms implementing standard complexity matrix multiplication. This suggests the need for a polyalgorithmic treatment of matrix multiplication for optimal performance, as was pointed out by Li *et al.* for the parallel case [45] and also by Gunnels *et al.* [15,46]. Also, the two variations of the hierarchical format have slightly different

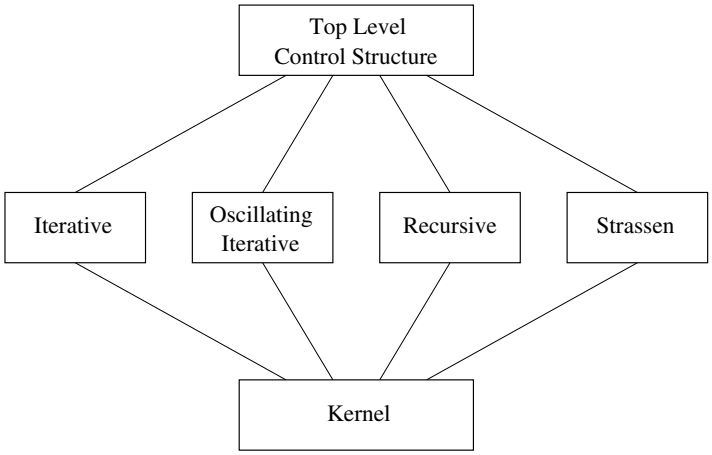


Figure 7. Software architecture of matrix-multiplication framework.

performance characteristics on different platforms, offering a choice of one or the other variant to be used on a given platform.

An object-oriented (OO) design strategy is, therefore, highly desirable to effectively manage the complexity arising from the use of the hierarchical storage formulation and the associated choices it offers for constructing an optimal design. The techniques can then be utilized by a library developer and can be incorporated into object oriented linear algebra libraries such as PMLP [33,34] and MTL [29,31]. The FLAME approach [60] is also a potential candidate for the application of this technology. The OO strategy allows the intricacies of the storage format and architecture-dependent peculiarities in the code to be hidden from the user. A convenient interface can then be made available to access the matrices stored in the hierarchical format.

Other linear algebra functions could also benefit from the good locality properties offered by Morton ordering that is used in the storage format. If all the functions required by a user have optimal implementations in the hierarchical framework, conversion costs between different storage formats can be saved. Even if conversion needs to be performed, the costs are minimal in many cases as shown by Chatterjee *et al.* [25] who measured the conversion time between row/column-major order and their non-linear layouts to be 2–5% of the total execution time for matrix multiplication. For other operations, the conversion costs relative to total execution time will also be low if their arithmetic complexity is high enough like that of matrix multiplication. A library can take advantage of the low conversion costs and use the hierarchical format internally to provide superior performance to users who want to use traditional storage formats in the rest of their code. Existing applications calling BLAS routines would immediately benefit from this approach. It is interesting to note that even when such format changes are not required some current BLAS implementations copy matrix data to avoid cache conflicts and obtain better performance [12,61,62].

An alternative to the OO approach is to hide the complexity of the storage format in the compiler and has been proposed by some researchers [50]. The compiler would then implicitly use the hierarchical format to store matrices and generate the necessary control structures and addressing schemes automatically. Legacy user code that does not make assumptions about the underlying storage format in using matrices can be compiled with the modified compiler to take advantage of the high-performance storage format. For example, the compiler would be smart enough to transform a loop nest which performs matrix operations into the hierarchical code structure required to efficiently access the hierarchical storage format. This would be similar to the transformations performed by current compilers such as SGI's MIPSpro compilers [16] on such loop nests, converting them into blocked structures with loop unrolling and other optimizations applied.

The work in this paper indicates that vendor-optimized BLAS is not the best answer for developing high-performance linear algebra libraries and applications. The emphasis of vendor optimizations must be shifted to low-level kernels, where they can really make a difference in tailoring code to specific processors. If a standard interface is used for the kernels, the portable upper layers can easily be implemented by a library writer resulting in better efficiency.

7. RESULTS

The performance results on three different architectures are now presented to demonstrate the effectiveness of the techniques that are discussed in this paper. All results are for double precision floating point arithmetic. Costs of conversion between different storage formats are not included in the performance numbers.

The systems used for experimentation include an SGI Challenge 10000, a Compaq AlphaServer and an Intel Pentium III machine. Details of these and the experimentation environment are provided in Table I. Note that although the machines are multiprocessor systems, the code is purely sequential. The Pentium III (PIII) system is a locally assembled *white box* dual processor machine and uses a ASUS P2B-D motherboard with Intel 440BX chipset*. These results are representative of the processor families covered, which include modern RISC and CISC processor architectures. The source code for all implementations presented in this section is available for download from the authors' project Web site [63].

7.1. Standard complexity algorithms

First, the performance of the various standard complexity matrix multiplication algorithms—regular iterative, recursive and oscillating iterative—with both variants of the hierarchical storage format are evaluated. The results on the different platforms are shown in Figures 8 and 9. One algorithm-variant combination that has the least significance is left out in each graph to reduce the clutter.

Except on the Alpha, variants 1 and 2 of the storage format make a clear distinction in performance. Variant 2 has better performance on the SGI, while variant 1 performs better on the PIII. On the Alpha,

*Minor variations in performance could be expected for systems having the same processor, but built using different system components.

Table I. Details of the systems used for experimentation.

SGI Challenge 10000 XL	
Processor:	195 MHz R10000
Peak Mflops:	390
Cache:	32 KB data, 32 KB instruction, 2 MB secondary
Main memory:	2 GB
OS:	Irix 6.5
Compiler:	MIPSpro Compilers 7.3
	Options: `-r10000 -O3 -64 -TARG:platform=IP25 -LNO:blocking=OFF -OPT:alias=typed`
	Options for kernel: Same as above
Compaq AlphaServer DS20	
Processor:	500 MHz Alpha 21264
Peak Mflops:	1000
Cache:	64 KB data, 64 KB instruction, 4 MB secondary
Main memory:	768 MB
OS:	Linux 2.2.12
Compiler:	`gcc` version 2.95.2
	Options: `-O3 -funroll-all-loops`
	Options for kernel: `-O1 -mcpu=ev6 -mmemory-latency=1 -fschedule-insns -fschedule-insns2 -fexpensive-optimizations`
Dual Intel Pentium III System	
Processor:	550 MHz Pentium III
Peak Mflops:	550
Cache:	16 KB data, 16 KB instruction, 512 KB secondary
Main memory:	512 MB
OS:	Linux 2.2.14
Compiler:	`gcc` version 2.95.2
	Options: `-O3 -funroll-all-loops -fomit-frame-pointer`
	Options for kernel: `-O1 -fexpensive-optimizations`

both variants have nearly the same performance with a slight advantage in favor of variant 1. The performance overhead associated with bounds checking in variant 1 is almost negligible because it is carried out at the level of Morton ordered blocks, the numbers of which are usually three to four orders of magnitude fewer than matrix elements. Since variant 1 does not use any fix-up code, its code size is smaller than that of variant 2. Again, the I-cache effects of this difference are not big enough to cause any significant performance impacts on the machines under study. The performance difference between the two variants that is observed here can be mostly attributed to how well the compiler can adapt the code for each variant to the underlying architecture.

The performance of the different algorithms for a given variant of the storage format is usually nearly the same for small and medium sized matrices. The performance difference becomes more pronounced

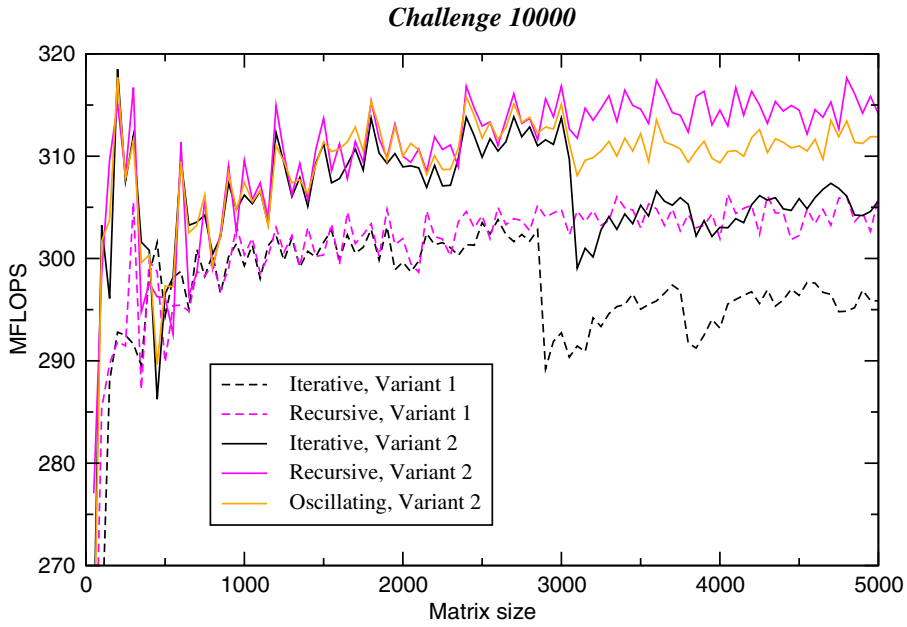


Figure 8. Performance of the various standard complexity matrix-multiplication algorithms utilizing the hierarchical storage format on the SGI Challenge 10000. The recursive algorithm using variant 2 of the hierarchical format offers the best performance. The regular iterative algorithm, although the worst performer in general, is highly competitive with the other algorithms for matrices smaller than 3000.

for larger matrices. As expected, only for large matrices does the improved locality properties offered by the oscillating iterative and the recursive algorithms become dominant. The recursive algorithm using variant 2 of the storage format is the overall best performer on the SGI. The oscillating iterative algorithm performs best for most matrix sizes on the Alpha and the PIII, with the recursive algorithm dominating in some cases.

The traditional iterative algorithm is often the worst performer for large matrices, although it is highly competitive with the other algorithms for small matrices. Another characteristic of the traditional iterative algorithm that can be seen in the graphs is the short drops in performance it displays when the matrix size is close to large powers of two (e.g. 2048 and 4096) or sum of large powers of two (e.g. $3072 = 1024 + 2048$). Note that these performance drops are minor compared with similar effects seen for the vendor-supplied (SGI) BLAS and ATLAS in the next section. After the drop, the performance starts climbing again as the matrix size is increased. These performance drops result from the linear data referencing pattern of the iterative algorithm that causes cache conflicts for matrix sizes that are closely related to powers of two in the previous manner. The oscillating iterative algorithm alleviates these performance dips because of the better locality offered by its two-miss property. The recursive algorithm is free from such variations and possesses the smoothest graph, since its data accesses match the Morton ordering inherent in the storage format. However, recursion

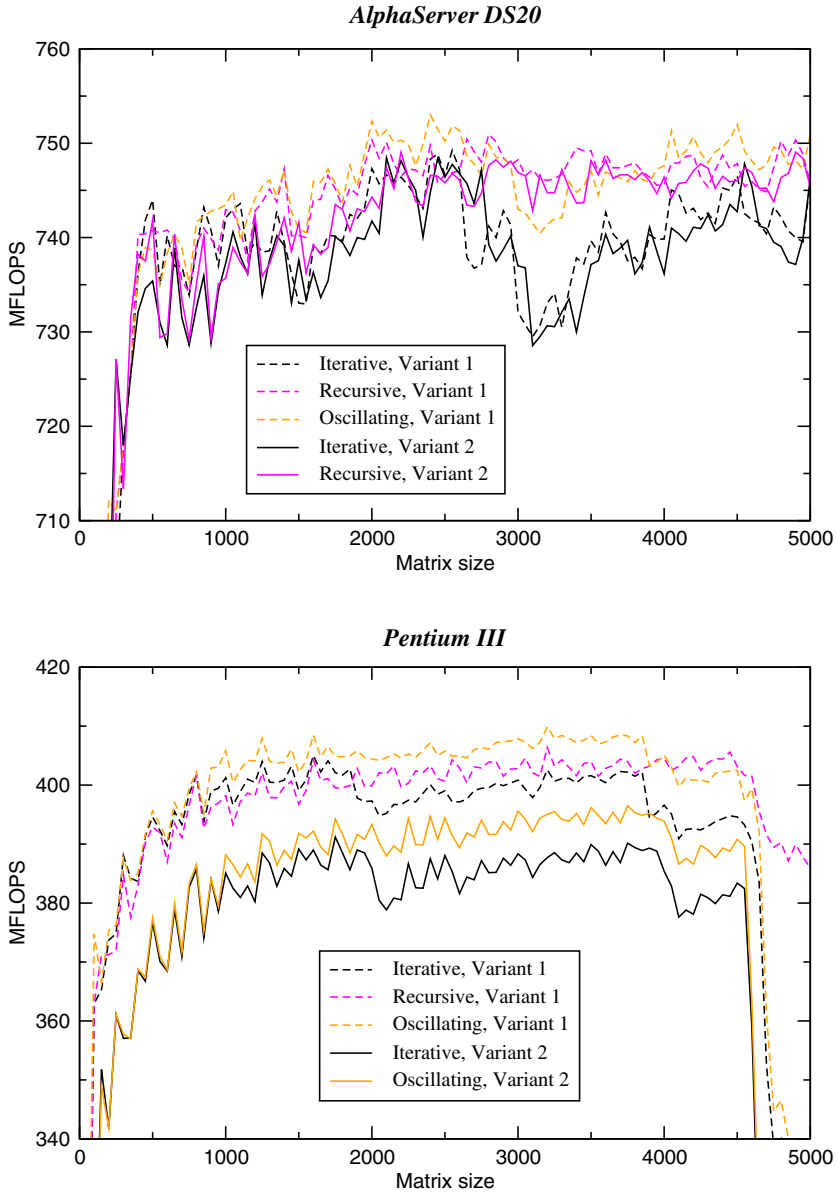


Figure 9. Performance of the various standard complexity matrix-multiplication algorithms utilizing the hierarchical storage format on the Compaq AlphaServer DS20 and the Pentium III. On the PIII, variant 1 of the hierarchical format performs better than variant 2. The same trend is also visible on the Alpha, although not as pronounced. The oscillating iterative algorithm has the best performance on both machines for most matrix sizes.

introduces some performance penalties because of function call overhead and the need to use non-cache-optimal level-one block sizes for better recursion performance which counteract the algorithm's locality benefits. The net result of these opposing effects could be in favor of or against recursion depending on the characteristics of the specific platform and matrix size. This explains its performance relative to the iterative algorithms being better or worse as a function of matrix size and machine type.

The results indicate the need for polyalgorithms to ensure optimal performance for all matrix sizes on a given platform because a single algorithm cannot provide the best performance under all execution conditions. The results also show that variations of the hierarchical storage format need to be supported for optimal performance across different platforms. This implies the need for a software environment that provides storage format independence and polyalgorithm capabilities for the implementation of high-performance linear algebra codes.

7.2. Comparisons with other methods

The best performer on each machine is chosen from the previous section and is compared with other matrix-multiplication strategies and implementations in Figures 10 and 11. The results show that the hierarchical formulation beats the matrix multiply provided by highly tuned linear algebra libraries such as ATLAS [12,13,28] on all platforms that were studied. Vendor-optimized BLAS implementation of `DGEMM` is included in the comparisons for the SGI. The implementations of Chatterjee's [25] and Gustavson's [20] methods, making use of native (vendor-supplied) BLAS or ATLAS for computing the block matrix products are listed as well[†]. The apparent Mflops of the Strassen–Winograd algorithm implemented inside the hierarchical framework is also presented. In fact, the performance numbers of Chatterjee's method are also in terms of apparent Mflops since extra computations are done on padded elements. The version of ATLAS used here is 3.0Beta.

The superior performance resulting from the hierarchical formulation is evident from the graphs. The standard complexity matrix multiplication based on the hierarchical framework is significantly faster than ATLAS on the SGI for matrices larger than 1500 and on the Alpha for matrices larger than 2700. It matches performance with native BLAS on the SGI, without the performance anomalies and fluctuations seen for the latter. The ATLAS and native BLAS performance curves sometimes experience sudden dips at or near power-of-two matrix sizes (e.g. 2048 for the SGI and 4096 for the Alpha) because of cache interference effects. Use of the hierarchical storage format eliminates such effects and smooths out the performance curves due to its inherent recursive Morton ordering, which reduces cache conflicts. The performance difference between column-major storage as used in ATLAS and the hierarchical storage becomes more pronounced as matrix size increases. Performance of ATLAS decreases significantly in most cases with increasing matrix size, whereas the hierarchical formulation keeps performance steady because of its better locality across all levels of the memory hierarchy that reduces paging and TLB misses, in addition to cache misses, for large matrices. ATLAS

[†]The authors had to implement Chatterjee's and Gustavson's algorithms themselves because of the evident absence of such implementations in the public domain. Extreme care has been taken to implement these algorithms precisely as described in their respective papers. The current authors make the source code for their implementations publicly available through their project Web site [63] so that the research community has access to these codes for further study and objective comparisons.

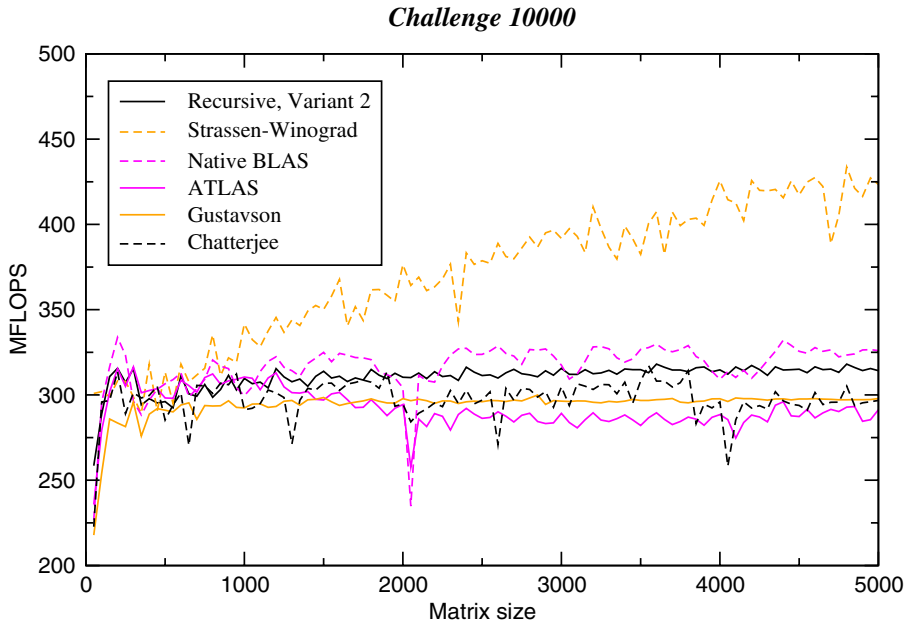


Figure 10. Performance of the standard complexity matrix multiplication and Strassen's algorithms in the hierarchical framework compared with other implementations on the SGI Challenge 10000. The hierarchical code for the standard complexity algorithm matches native BLAS performance and beats ATLAS comfortably for large matrices. The hierarchical code has a steady performance curve unlike that of native BLAS, ATLAS and Chatterjee. The relative performance of Strassen's algorithm increases steadily with matrix size, over the range considered.

and native BLAS use (multi-level) blocking to enhance locality, which is sometimes good enough to rival Morton ordered performance, as in the case of the SGI BLAS.

Since the lower arithmetic complexity of Strassen's algorithm is asymptotic in nature, the performance benefits are apparent only for large enough matrices. The additional memory requirement and the non-localized memory access pattern of Strassen, combined with the fact that multiplying small matrices does not lead to a significant savings in operation count results in poorer performance for small matrices. The point at which Strassen–Winograd starts to run faster than standard complexity algorithms is dependent on the machine. The Alpha and the SGI, with their large caches (see machine descriptions in Table I) help Strassen to outperform the other algorithms even for reasonably small matrices. The crossover point is delayed further on the PIII because of its small primary and secondary caches. Despite its clear dominance over standard matrix multiplication for large matrix sizes, Strassen's algorithm may not be suitable for certain applications because of its vulnerability to roundoff error when small off-diagonal elements are combined with large diagonal elements [39,53]. Where numerically applicable, the results shown here have widened the range over which its performance is better than the conventional algorithm. For example, using native `DGEMM` as the basis for comparison,

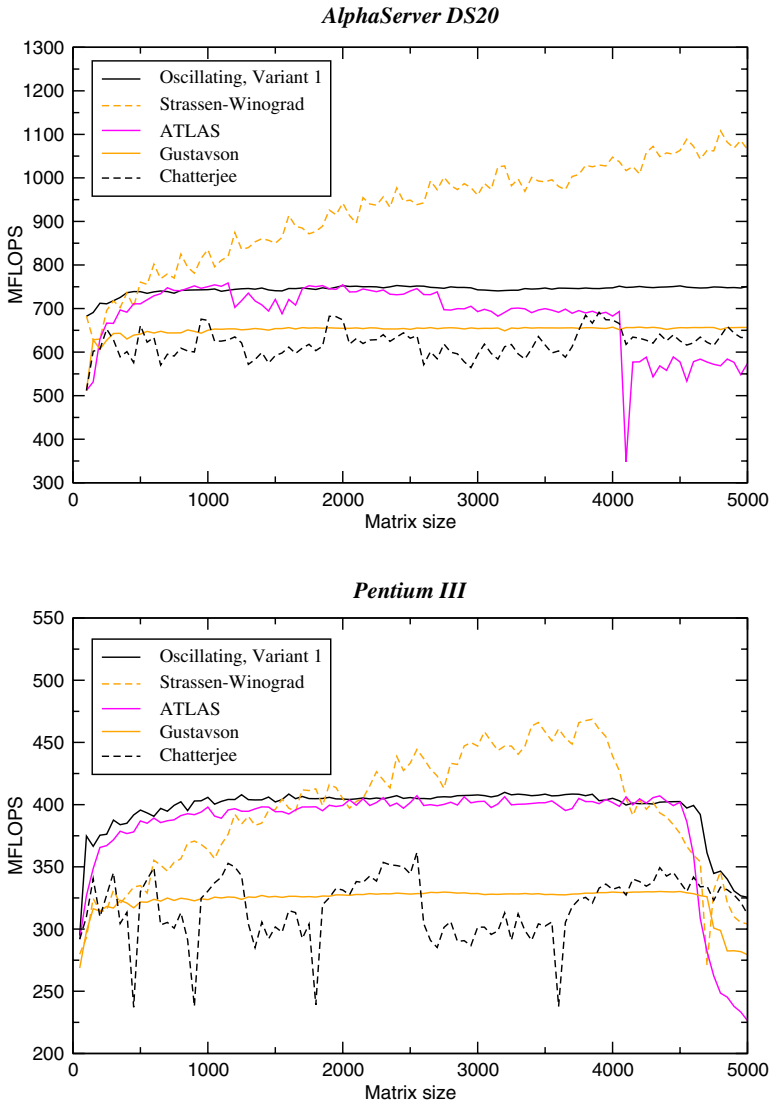


Figure 11. Performance of the standard complexity matrix multiplication and Strassen's algorithms in the hierarchical framework compared with other implementations on the Compaq AlphaServer DS20 and the Pentium III. The hierarchical code for the standard complexity matrix multiplication beats ATLAS on the Alpha and the PIII machines also, albeit narrowly for certain matrix sizes. Chatterjee's method shows wild fluctuations, evidently because of extra computations performed on padded zero elements. Performance of Strassen on the PIII is not as good as that on the other platforms tested here because of the smaller cache and main memory on the system.

the Strassen–Winograd implementation presented here performs much better than that reported by Chatterjee *et al.* in [25], especially in terms of minimizing their wide performance fluctuations. The hierarchical formulation, combined with the variable recursion cutoff criterion presented in this paper, contributes to the enhanced performance characteristics of Strassen obtained here.

When the problem size exceeds the amount of main memory available, performance drops sharply for all algorithms, as seen in the case of the Pentium III, whose 512 MB of RAM starts becoming insufficient for matrix sizes greater than 4500. For Strassen, the performance starts to degrade much earlier, when the matrix size is close to 4000 because of the additional temporary storage that the algorithm requires.

The block multiplications inside the recursive algorithms used by Chatterjee and Gustavson are performed by calling `DGEMM` of native BLAS, if available, or ATLAS[‡]. This is an important factor limiting their full performance potential since a generalized `DGEMM` implementation cannot provide good performance when called repeatedly for multiplying the small block matrices. Some `DGEMM` implementations may use techniques such as data copying, which could turn out to be detrimental to performance in this setting. The approach adopted here of addressing processor-specific issues by means of a specialized kernel that is small and does not have the overheads associated with the *fat interface* of `DGEMM` is essential for constructing an optimal matrix-multiplication routine.

This effect is evidenced by the next set of graphs, Figures 12 and 13, that shows performance of Gustavson's and Chatterjee's algorithms using the authors' low-level kernels for block products in place of `DGEMM`. The replacement of calls to `DGEMM` with calls to the low-level kernels results in significant performance improvement for Gustavson on all platforms. On the SGI and the Alpha performance of Gustavson now matches the algorithms implemented in the hierarchical storage formats. However, on the PIII, a significant gap still exists between Gustavson and the authors' hierarchical code, apparently because the overhead of Gustavson's recursive formulation has a greater impact on the PIII. Chatterjee's algorithm behaves differently than Gustavson when `DGEMM` is replaced with the low-level kernels—performance worsens on the SGI and the PIII, but improves on the Alpha. This behavior can be primarily attributed to the interaction between the blocking factors chosen by Chatterjee's algorithm [19,25] and the kernels. The block sizes chosen by Chatterjee lie in an architecture-dependent range defined by $[T_{\min}, T_{\max}]$ and are functions of the matrix size. On the SGI and the PIII, block sizes chosen in this fashion are evidently less optimal for the low-level kernel, resulting in poorer performance compared with the use of `DGEMM`. The effect is reverse on the Alpha.

The wild performance fluctuations exhibited by Chatterjee's method is evidently because of the extra computations on padded matrix elements. The performance of the hierarchical formulation is also better than that reported by Wise for SGI platforms in [44] (using the performance of vendor-supplied BLAS as the basis for comparison).

7.3. Summary of results

To summarize, the results presented here validate the performance benefits associated with the use of the proposed framework for the construction of matrix-multiplication codes. The framework

[‡]`DGEMM` is used for computing the block products because both Chatterjee and Gustavson mention its use in their respective papers [20,25].

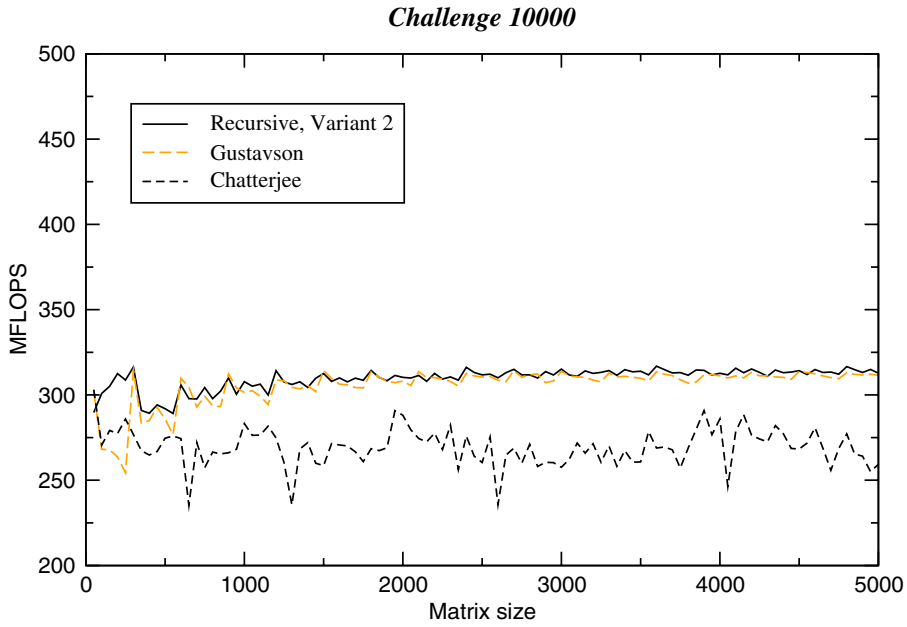


Figure 12. Performance of Gustavson's and Chatterjee's algorithms using the authors' kernels in place of `DGEMM` on the SGI Challenge 10000. An algorithm implemented in the hierarchical framework is also shown for reference. Gustavson's performance increases as a result of the replacement of `DGEMM` with the low-level kernels. But Chatterjee's performance worsens, evidently because of their particular technique for the selection of block sizes.

involves a hierarchical storage format, polyalgorithms and optimized processor-specific kernels. In addition to enabling various algorithms to approach their full performance potential, the hierarchical storage format minimizes performance fluctuations and maintains steady performance with matrix size (predictable performance) because of its good locality properties. This storage format also allows various iterative and recursive algorithms, including Strassen, to be implemented efficiently as evidenced by the results. This capability to support polyalgorithms is important because a single algorithm does not usually perform optimally for all problem sizes and platforms, as has been demonstrated. The excessive padding used in Chatterjee's algorithm and the extra computation performed on padded portions of the matrix makes it sub-optimal and imparts undesirable fluctuations to its performance graph. The results also show that iterative algorithms can perform as well as and sometimes even better than recursive algorithms if efficient techniques are used, such as for address calculation and for maintaining locality. The performance of Strassen's algorithm has been improved by devising a variable recursion cutoff point, which when used within the hierarchical storage formulation removes performance anomalies and enhances overall performance. The need to use a kernel that is tuned to the specific processor architecture is also emphasized by the results. As already mentioned, the full performance potential of the kernels have not been achieved in the current implementation.

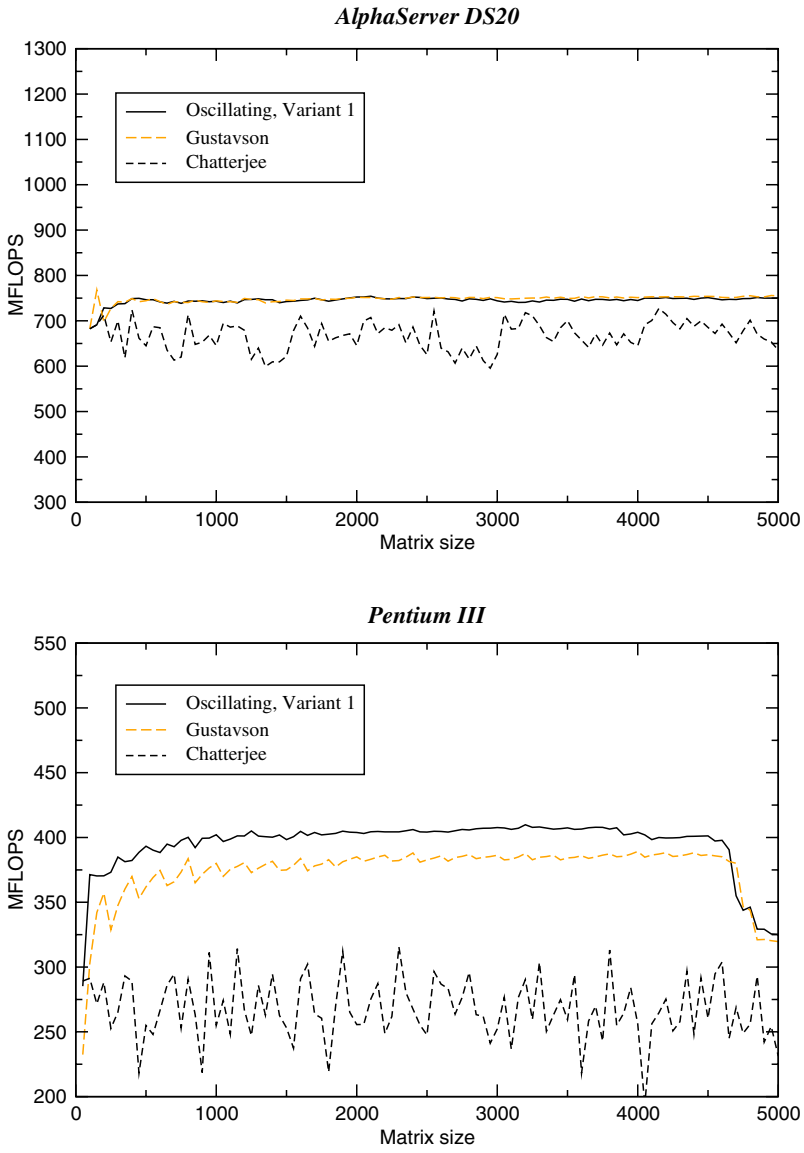


Figure 13. Performance of Gustavson's and Chatterjee's algorithms using the authors' kernels in place of `DGEMM` on the Compaq AlphaServer DS20 and the Pentium III. An algorithm implemented in the hierarchical framework is also shown in each case for reference. Gustavson shows significant improvements in performance on both machines because of the replacement of `DGEMM` with the low-level kernels. Chatterjee, on the other hand, shows small improvements on the Alpha, but has poorer performance on the PIII. This behavior is evidently because of their particular choice of blocking factors.

Therefore, there is room for further performance enhancements across the entire spectrum of matrix sizes.

8. CONCLUSIONS AND FUTURE WORK

A simple conceptual framework is presented in this paper to guide the construction of high-performance matrix-multiplication codes. The framework makes use of a hierarchical storage format that is designed to efficiently handle storage of arbitrary-sized matrices in Morton order, enhancing locality and providing opportunities for significant improvements in performance. When the storage format is combined with a well-written algorithm and an optimized kernel tuned to specific processors, the proposed techniques yield matrix multiplication routines that either surpass or match the performance of highly optimized libraries such as ATLAS and other competing methodologies reported in the literature (e.g. Chatterjee *et al.* [25]) on all platforms studied, which included three different representative RISC and CISC architectures. Incorporating other optimization techniques such as prefetching and ATLAS-like automatic search of parameter space may lead to even more improvements in performance. The authors' strategy also produces performance curves that are smooth, unlike Chatterjee's method that has radical swings in performance. The smoothness is indicative of optimization being good and balanced. This also improves predictability of performance with respect to matrix size.

The hierarchical storage format supports various algorithms efficiently including variations of iterative, recursive and Strassen. The oscillating iterative algorithm presented here improves cache behavior by incorporating the optimal two-miss property for consecutive block products. The iterative algorithms are shown to be highly competitive with the recursive algorithm and even faster in some cases, debunking the view expressed by some researchers [20,25] that the storage format should match the algorithm for optimal performance. Use of Strassen with the hierarchical format, along with the variable recursion cutoff criterion devised here, has widened the usefulness of Strassen by providing enhanced performance with minimal fluctuations. The results presented here also show that a single algorithm is not suitable for all matrix sizes and machine architectures for performance reasons. A polyalgorithmic approach is, therefore, required for obtaining the best possible performance.

An important conclusion that arises out of the current work is the need for standardizing kernel interfaces[§] so that platform vendors and independent software providers can specialize in optimizing kernels to specific processors, which can then be utilized by library developers to write portable high-performance code efficiently. It is to be noted here that BLAS `DGEMM` is not suitable for use as a kernel because of its *fat interface* and the overheads associated with a generalized implementation of matrix multiply.

Source code for all the implementations presented here is available at the project Web site [63]. Gustavson's and Chatterjee's algorithms were evidently not publicly available from their respective authors, which necessitated the independent implementation presented here to evaluate their performance relative to other competing methods. The current authors have strived to implement

[§]Note that these kernel routines are designed to operate on small blocks, often fixed-size, that fit in the L1 cache.

them as efficiently as they could, paying attention to every detail by following the description of the algorithms available in the published literature. By making the source code for these as well as the other implemented algorithms publicly available, the authors provide the scientific community with an infrastructure to facilitate further studies and comparisons on an objective basis without having to re-implement them. Porting the code to other platforms and comparing their performance with other techniques and algorithms are encouraged.

As part of future work, the hierarchical storage format and related techniques will be applied to other linear algebra operations such as matrix factorizations (e.g. LU) that have high enough complexity to benefit from this approach. An object-oriented framework will be designed to encapsulate the ideas presented here, including polyalgorithms and storage format independence, in a format that can easily be used by numerical math libraries and applications for enhanced performance. A theoretical model that includes memory hierarchy behavior, processor architectural features and algorithmic characteristics will be devised to provide the capability to analytically determine performance costs. The analytical model would also be useful in a polyalgorithmic library for choosing the algorithm that performs best for a given set of parameters. The performance framework for writing high-performance linear algebra programs will be extended to cover parallel and distributed environments. Efficient methods for parallelizing algorithms in the hierarchical formulation will be investigated. The design of kernel interfaces for various linear algebra operations will also be undertaken.

ACKNOWLEDGEMENTS

The authors wish to thank Robert van de Geijn at the Department of Computer Sciences, The University of Texas, Austin and anonymous others for their input that contributed to improvements in the paper. Acknowledgments are also due to Yoginder Dandass at the Department of Computer Science, Mississippi State University for valuable discussions that enabled clarification of certain sections of this paper.

REFERENCES

1. Wolfe M. Iteration space tiling for memory hierarchies. *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987. SIAM Press, 1987.
2. Gallivan KA, Jalby W, Meier U, Sameh AH. Impact of hierarchical memory systems on linear algebra algorithm design. *The International Journal of Supercomputer Applications* 1988; **2**(1):12–48.
3. Schreiber R, Dongarra J. Automatic blocking of nested loops. *Technical Report CS-90-108*, Department of Computer Science, University of Tennessee, May 1990.
4. Lam MS, Rothberg EE, Wolf ME. The cache performance and optimizations of blocked algorithms. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, April 1991. ACM Press, 1991.
5. Wolf ME, Lam MS. A data locality optimizing algorithm. *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, June 1991. ACM Press, 1991.
6. Chen D. Hierarchical blocking and data flow analysis for numerical linear algebra. *Proceedings of the ACM International Conference on Supercomputing*. ACM Press, 1991; 12–19.
7. Agarwal RC, Gustavson FG, Zubair M. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development* 1994; **38**(5):563–576.
8. Navarro JJ, Juan T, Lang T. MOB forms: A class of multilevel block algorithms for dense linear algebra operations. *International Conference on Supercomputing*, 1994; 354–363.
9. Golub G, Van Loan C. *Matrix Computations* (3rd edn) ch. 1. The Johns Hopkins University Press: Baltimore, MD, 1996; 43–46.

10. Dongarra J, Du Croz J, Duff I, Hammarling S. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 1990; **16**(1):1–17.
11. Basic linear algebra subprograms (BLAS). http://www.netlib.org/blas/ [31 May 2001].
12. Whaley RC, Petitet A, Dongarra JJ. Automated empirical optimization of software and the ATLAS project. *Technical Report*, University of Tennessee, September 2000. Available at: http://www.netlib.org/atlas/ [31 May 2001].
13. Automatically tuned linear algebra software (ATLAS). http://netlib2.cs.utk.edu/atlas/ [31 May 2001].
14. Aberdeen D, Baxter J. Emmerald: A fast matrix–matrix multiply using Intel SIMD technology. *Concurrency and Computation: Practice and Experience*. 2001; **13**:103–119. http://csl.anu.edu.au/ daa/research.html [31 May 2001].
15. Gunnels J, Henry G, van de Geijn R. A family of high-performance matrix multiplication algorithms. *Technical Report*, Department of Computer Science, The University of Texas, Austin, 2001.
16. MIPSpro family of compilers. http://www.sgi.com/developers/devtools/languages/mipspro.html [31 May 2001].
17. Sagan H. *Space Filling Curves*. Springer: Berlin, 1994.
18. Frens JD, Wise DS. Auto-blocking matrix multiplication or tracking BLAS3 performance with source code. *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.
19. Chatterjee S, Lebeck A, Patnala P, Thottethodi M. Recursive array layouts and fast parallel matrix multiplication. *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA '99)*, June 1999.
20. Gustavson F, Henriksson A, Kågström B, Ling P. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Proceedings of the 4th International Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems (PARA '98)* (*Lecture Notes in Computer Science*, vol. 1541). Springer: Berlin, 1998; 195–206.
21. Gustavson F. Recursion leads to automatic variable blocking for dense linear algebra algorithms. *IBM Journal of Research and Development* 1997; **41**(6):737–755.
22. Andersen BS, Gustavson F, Karaivanov A, Wasniewski J, Yalamov PY. LAWRA—linear algebra with recursive algorithms. *Proceedings of the Conference on Parallel Processing and Applied Mathematics*, September 1999; 63–76.
23. Gustavson J, Jonsson I. Minimal-storage high-performance Cholesky factorization via blocking and recursion. *IBM Journal of Research and Development* 2000; **44**(6):823–849.
24. Gustavson FG. New generalized data structures for matrices lead to a variety of high performance algorithms. *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, October 2000.
25. Chatterjee S, Jain V, Lebeck A, Mundhra S, Thottethodi M. Nonlinear array layouts for hierarchical memory systems. *Proceedings of the 13th ACM International Conference on Supercomputing (ICS '99)*, June 1999.
26. Bilmes J, Asanović K, Chin C-W, Demmel J. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. *Proceedings of the International Conference on Supercomputing*, July 1997.
27. Bilmes J, Asanovic K, Chin C-W, Demmel J. The PHiPAC v1.0 matrix-multiply distribution. *Technical Report UCB/CSD-98-1020*, CS Division, University of California at Berkeley, October 1998.
28. Whaley RC, Dongarra J. Automatically tuned linear algebra software (ATLAS). *Technical Report*, University of Tennessee, July 1997.
29. Siek J, Lumsdaine A. The matrix template library: A generic programming approach to high performance numerical linear algebra. *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE)*, December 1998.
30. Siek J, Lumsdaine A. The matrix template library: A unifying framework for numerical linear algebra. *Workshop on Parallel Object-Oriented Scientific Computing (POOSC '98), Proceedings of the 12th European Conference on Object-Oriented Programming*, July 1998.
31. The matrix template library (MTL). http://www.lsc.nd.edu/research/mtl/ [31 May 2001].
32. Birov L, Prokofiev A, Bartenev Y, Vargin A, Purkayastha A, Dandass Y, Erzunov V, Shanikova E, Skjellum A, Bangalore P, Shuvalov E, Ovechkin V, Frolova N, Orlov S, Egorov S. The parallel mathematical libraries project (PMLP): Overview, design innovations and preliminary results. *Fifth International Conference on Parallel Computing Technologies (PACT '99)* (*Lecture Notes in Computer Science*, vol. 1662). Springer: Berlin, 1999; 186–193.
33. Birov L, Bartenev Y, Vargin A, Purkayastha A, Skjellum A, Dandass Y, Bangalore P. The parallel mathematical libraries project (PMLP)—a next generation scalable, sparse, object-oriented, mathematical library suite. *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.
34. The parallel mathematical libraries project (PMLP). http://WWW.ERC.MsState.Edu/labs/hpcl/pmlp/ [31 May 2001].
35. Strassen V. Gaussian elimination is not optimal. *Numerische Mathematik* 1969; **13**:354–356.
36. IBM Corporation. Engineering and scientific subroutine library (ESSL). http://www.rs6000.ibm.com/software/Apps/essl.html [31 May 2001].
37. IBM Corporation. Engineering and scientific subroutine library (ESSL) for AIX: Guide and reference, version 3 release 2. http://www.austin.ibm.com/resource/aix_resource/sp_books/essl/ [31 May 2001].
38. Brent R. Algorithms for matrix multiplication. *Technical Report CS 157*, Computer Science Department, Stanford University, Palo Alto, CA, 1970.

39. Higham N. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software* 1990; **16**(4):352–368.
40. Douglas C, Heroux M, Slishman G, Smith RM. GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix–matrix multiply algorithm. *Journal of Computational Physics* 1994; **110**:1–10.
41. Huss-Lederman S, Jacobson EM, Johnson JR, Tsao A, Turnbull T. Implementation of Strassen's algorithm for matrix multiplication. *Proceedings of Supercomputing '96*, November 1996.
42. Pauca VP, Sun X, Chatterjee S, Lebeck AR. Architecture-efficient Strassen's matrix multiplication: A case study of divide-and-conquer algorithms. *Proceedings of the International Linear Algebra Society (ILAS) Symposium on Algorithms for Control, Signals and Image Processing*, June 1997.
43. Thottethodi M, Chatterjee S, Lebeck AR. Tuning Strassen's matrix multiplication for memory efficiency. *Proceedings of Supercomputing '98*, November 1998.
44. Wise D. Ahnentafel indexing into morton-ordered arrays, or matrix locality for free. *European Conference on Parallel Computing, Euro-Par 2000* (*Lecture Notes in Computer Science*, vol. 1900). Springer: Berlin, 2000; 774–784.
45. Li J, Skjellum A, Falgout R. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency and Computation: Practice and Experience* 1997; **9**(5):345–389.
46. Gunnels J, Lin C, Morrow G, van de Geijn R. A flexible class of parallel matrix multiplication algorithms. *Proceedings First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP '98)*. IEEE Computer Society Press, 1998; 110–116.
47. Agarwal RC, Balle SM, Gustavson FG, Joshi M, Palkar P. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 1995; **39**(5):575–582.
48. Valsalam V, Skjellum A. Fast integer dilation for structured problems. *Technical Report MSSU-COE-ERC-00-05*, Engineering Research Center, Mississippi State University, MS, March 2000.
49. Schrack G. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Understanding* 1992; **55**(3):221–230.
50. Wise D, Frens J. Morton-order matrices deserve compilers' support. *Technical Report 533*, Department of Computer Science, Indiana University, Bloomington, IN, November 1999.
51. Computational plant (Cplant). http://www.cs.sandia.gov/cplant/ [31 May 2001].
52. Stocco L, Schrack G. Integer dilation and contraction for quadtrees and octrees. *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, May 1995.
53. Golub G, Van Loan C. *Matrix Computations* (3rd edn) ch. 2. The Johns Hopkins University Press: Baltimore, MD, 1996; 66–67.
54. Kumar B, Huang C-H, Sadayappan P, Johnson RW. A tensor product formulation of Strassen's matrix multiplication algorithm. *Scientific Programming* 1995; **4**(4):275–289.
55. Fischer P, Probert R. Efficient procedures for using matrix algorithms. *Automata, Languages and Programming* (*Lecture Notes in Computer Science*, vol. 14). Springer: Berlin, 1974; 413–427.
56. Huss-Lederman S, Jacobson EM, Johnson JR, Tsao A, Turnbull T. Strassen's algorithm for matrix multiplication: Modeling, analysis and implementation. *Technical Report CCS-TR-96-147*, Center for Computing Sciences, 1996.
57. Skjellum A, Bangalore P. *Draft Document for the BLAS Lite Specification*, Department of Computer Science, Mississippi State University, February 1997.
58. Lumsdaine A, Skjellum A, Bangalore P. Standard sequential mathematical libraries: Promises and pitfalls, opportunities and challenges. *The Multicomputer Toolbox Project BLAIS Working Note 0*, Department of Computer Science, Mississippi State University, May 1996.
59. Siek J, Lumsdaine A. A rational approach to portable high performance: The basic linear algebra instruction set (BLAIS) and the fixed algorithm size template (FAST) library. *Workshop on Parallel Object-Oriented Scientific Computing (POOSC '98), Proceedings of the 12th European Conference on Object-Oriented Programming*, July 1998.
60. Gunnels J, van de Geijn R, Henry G. Formal linear algebra methods environment (FLAME) overview. *FLAME Working Note 1, Technical Report*, The University of Texas at Austin, November 2000.
61. Greer B, Henry G. High performance software on Intel Pentium Pro processors or micro-ops to TeraFLOPS. *SC97 Conference Proceedings*, 1997. Available at: http://www.supercomp.org/sc97/proceedings/TECH/GREER/INDEX.HTM [31 May 2001].
62. Gustavson FG, Henriksson A, Jonsson I, Kågström B, Ling P. Superscalar GEMM-based level 3 BLAS—the on-going evolution of a portable and high-performance library. *Applied Parallel Computing in Large Scale Scientific and Industrial Problems (PARA)* (*Lecture Notes in Computer Science*, vol. 1541). Springer: Berlin, 1998; 207–215.
63. Valsalam V, Skjellum A. Linear algebra based on hierarchical extension of recursive orderings (LAB-HERO). http://www.hpcl.cs.msstate.edu/lab-hero/ [31 May 2001].

