Recursive Array Layouts and Fast Matrix Multiplication*

Siddhartha Chatterjee† Alvin R. Lebeck‡ Praveen K. Patnala§

Mithuna Thottethodi‡

Submitted for publication to IEEE TPDS

Abstract

The performance of both serial and parallel implementations of matrix multiplication is highly sensitive to memory system behavior. False sharing and cache conflicts cause traditional column-major or row-major array layouts to incur high variability in memory system performance as matrix size varies. This paper investigates the use of recursive array layouts to improve performance and reduce variability.

Previous work on recursive matrix multiplication is extended to examine several recursive array layouts and three recursive algorithms: standard matrix multiplication, and the more complex algorithms of Strassen and Winograd. While recursive layouts significantly outperform traditional layouts (reducing execution times by a factor of 1.2–2.5) for the standard algorithm, they offer little improvement for Strassen's and Winograd's algorithms. For a purely sequential implementation, it is possible to reorder computation to conserve memory space and improve performance between 10% and 20%. Carrying the recursive layout down to the level of individual matrix elements is shown to be counter-productive; a combination of recursive

*This work supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants EIA-97-26370 and CDA-95-12356, NSF Career Award MIP-97-02547, The University of North Carolina at Chapel Hill, Duke University, and an equipment donation through Intel Corporation's Technology for Education 2000 Program. Parts of this work have previously appeared in the Proceedings of Supercomputing 1998 [50], ICS 1999 [10], and SPAA 1999 [11]. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

†IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598. Work performed when the author was a faculty member at The University of North Carolina at Chapel Hill.

‡Department of Computer Science, Duke University, Durham, NC 27708-0129.

§Nexsi Corporation, 1959 Concourse Drive, San Jose, CA 95131. Work performed when the author was a graduate student at The University of North Carolina at Chapel Hill.

layouts down to canonically ordered matrix tiles instead yields higher performance. Five recursive layouts with successively increasing complexity of address computation are evaluated, and it is shown that addressing overheads can be kept in control even for the most computationally demanding of these layouts.

Keywords: Data layout, matrix multiplication

1 Introduction

High-performance dense linear algebra codes, whether sequential or parallel, rely on good spatial and temporal locality of reference for their performance. Matrix multiplication (the BLAS 3 [14] `dgemm` routine) is a key linear algebraic kernel. The performance of this routine is intimately related to the memory layout of the arrays. On modern shared-memory multiprocessors with multi-level memory hierarchies, the column-major layout assumed in the BLAS 3 library can produce unfavorable access patterns in the memory hierarchy that cause interference misses and false sharing and increase memory system overheads experienced by the code. These effects result in performance anomalies as matrix size is varied. In this paper, we investigate recursive array layouts accompanied by recursive control structures as a means of delivering high and robust performance for parallel dense linear algebra.

The use of quad- or oct-trees (or, in a dual interpretation, space-filling curves [26, 43]) is known in parallel computing [2, 28, 29, 44, 47, 52] for improving both load balance and locality. They have also been applied for bandwidth reduction in information theory [4], for graphics applications [21, 37], and for database applications [32]. The computations thus parallelized or restructured are reasonably coarse-grained, thus making the overheads of maintaining and accessing the data structures insignificant. A series of papers by Wise *et al.* [18, 55, 56] champions the use of quad-trees to represent matrices, explores its use in matrix multiplication, and (most recently)

demonstrates the viability of automatic compiler optimization of a simple recursive algorithmic specification. This paper addresses several questions that occurred to us while reading the 1997 paper of Frens and Wise [18].

- Previous work using recursive layouts were not greatly concerned with the overhead of address computations. The algorithms described in the literature [42] follow from the basic definitions and are not particularly optimized for performance. Are there fast addressing algorithms, perhaps involving bit manipulation, that would enable such data structures to be used for fine-grained parallelism? Or, even better, might the address computations be embedded implicitly in the control structure of the program?

- Frens and Wise carried out their quad-tree layout of matrices down to the level of matrix elements. However, another result due to Lam, Rothberg, and Wolf [36]—that a tile that fits in cache and is contiguous in memory space can be organized in a canonical order without compromising locality of reference—suggested to us that the quadtree decomposition might be pruned well before the element level and be made to co-exist with tiles organized in a canonical manner. Could this interfacing of two layout functions be accomplished without increasing the cost of addressing? How does absolute performance relate to the choice of tile size?

- Frens and Wise assumed that all matrices would be organized in quad-tree fashion, and therefore did not quantify the cost of converting to and from a canonical order at the routine interface. However, as Section 2.2 of the Basic Linear Algebra Subroutine Technical (BLAST) Forum standard [3] shows, there is as yet no consensus within the scientific programming community to adopt such a layout for matrices. We felt it important to quantify the overhead

of format conversion. Would the performance benefits of quad-tree data structures be lost in the cost of building them in the first place?

- There are many variants of recursive orderings. Some of these orderings, such as Gray-Morton [38] and Hilbert [26], are supposedly better for load balancing, albeit at the expense of greater addressing overhead. How would these variants compare in terms of complexity *vs.* performance improvement for fine-grained parallel computations?

- Frens and Wise speculated about the "attractive hybrid composed of Strassen's recurrence and this one" [18, p. 215]. This is an interesting variation on the problem, for two reasons. First, Strassen's algorithm [49] achieves a lower operation count at the expense of more data accesses in less local patterns. Second, the control structure of Strassen's algorithm is more complicated than that of the standard recursive algorithm, making it trickier to work with quad-tree layouts. Could this combination be made to work, and how would it perform?

Our major contributions are as follows. First, we provide improved performance over that reported by Frens and Wise [18] by stopping their the quadtree layout of matrices well before the level of single elements. Second, we integrate recursive data layouts into Strassen's algorithm and provide some surprising performance results. Third, we test five different recursive layouts and characterize their relative performance. We provide efficient addressing routines for these layout functions that would be useful to implementors wishing to incorporate such layout functions into fine-grained parallel computations. Finally, as a side effect, we provide an evaluation of the strengths and weaknesses of the Cilk system [6], which we used to parallelize our code.

As Wise *et al.* have continued work along these lines, it is worthwhile to place this paper in the larger context. Their use of the algebra of dilated integers [55] has allowed them to automatically

unfold the divide-and-conquer recursive control to larger base cases to generate larger basic blocks and to improve instruction scheduling. Their experimental compiler [56] improves performance beyond that reported in this paper. All of these further improvements strengthens our conclusions and opens up the possibility of better support of these concepts in future languages, libraries, and compilers.

The remainder of this paper is organized as follows. Section 2 introduces the algorithms for fast matrix multiplication that we study in this paper. Section 3 introduces recursive data layouts for multi-dimensional arrays. Section 4 describes the implementation issues that arose in combining the recursive data layouts with the divide-and-conquer control structures of the algorithms. Section 5 offers measurement results to support the claim that these layouts improve the overall performance. Section 6 compares our approach with previous related work. Section 7 presents conclusions and future work.

2 Algorithms for fast matrix multiplication

Let A and B be two $n \times n$ matrices, where we initially assume that $n = 2^k$. Let $C = A \bullet B$, where the symbol $\bullet$ represents the linear algebraic notion of matrix multiplication (*i.e.*, $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$.)

We formulate the matrix product in terms of quadrant or sub-matrix operations rather than by row or column operations. Partition the two input matrices *A* and *B* and the result matrix *C* into

quadrants as follows.

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

The standard algorithm that performs $O(n^3)$ operations proceeds as shown in Figure 1(a), performing eight recursive matrix multiplication calls and four matrix additions.

Strassen's original algorithm [49] reduces the number of recursive matrix multiplication calls from eight to seven at the cost of 18 matrix additions/subtractions, using algebraic identities. This change reduces the operation count to $O(n^{\lg 7})$. It proceeds as shown in Figure 1(b).

Winograd's variant [17] of Strassen's algorithm uses seven recursive matrix multiplication calls and 15 matrix additions/subtractions; this is known [17] to be the minimum number of multiplications and additions possible for any recursive matrix multiplication algorithm based on division into quadrants. The computation proceeds as shown in Figure 1(c).

Compared to Strassen's original algorithm, the noteworthy feature of Winograd's variant is its identification and reuse of common subexpressions. These shared computations are responsible for reducing the number of additions, but can contribute to worse locality of reference.

Figure 2 illustrates the locality patterns of these three algorithms. We observe that the standard algorithm has good *algorithmic locality of reference*, accessing consecutive elements in a matrix row or column.[1] In contrast, the access patterns of Strassen's and Winograd's algorithms are much worse in terms of algorithmic locality. This is particularly evident along the main diagonal for

[1]We add the qualifier "algorithmic" to emphasize the point that we are reasoning about this issue at an algorithmic level, independent of the architecture of the underlying memory hierarchy. In terms of the 3C model [27] of cache misses, we are reasoning about capacity misses at a high level, not about conflict misses.

(a)

Pre-additions	Recursive calls	Post-additions
	$P_1 = A_{11} \bullet B_{11}$	
	$P_2 = A_{12} \bullet B_{21}$	
	$P_3 = A_{21} \bullet B_{11}$	$C_{11} = P_1 + P_2$
	$P_4 = A_{22} \bullet B_{21}$	$C_{21} = P_3 + P_4$
	$P_5 = A_{11} \bullet B_{12}$	$C_{12} = P_5 + P_6$
	$P_6 = A_{12} \bullet B_{22}$	$C_{22} = P_7 + P_8$
	$P_7 = A_{21} \bullet B_{12}$	
	$P_8 = A_{22} \bullet B_{22}$	

(b)

Pre-additions		Recursive calls	Post-additions
		$P_1 = S_1 \bullet T_1$	
$S_1 = A_{11} + A_{22}$	$T_1 = B_{11} + B_{22}$	$P_2 = S_2 \bullet B_{11}$	$C_{11} = P_1 + P_4 - P_5 + P_7$
$S_2 = A_{21} + A_{22}$	$T_2 = B_{12} - B_{22}$	$P_3 = A_{11} \bullet T_2$	$C_{21} = P_2 + P_4$
$S_3 = A_{11} - A_{12}$	$T_3 = B_{21} - B_{11}$	$P_4 = A_{22} \bullet T_3$	$C_{12} = P_3 + P_5$
$S_4 = A_{21} - A_{11}$	$T_4 = B_{11} + B_{12}$	$P_5 = S_3 \bullet B_{22}$	$C_{22} = P_1 + P_3 - P_2 + P_6$
$S_5 = A_{12} - A_{22}$	$T_5 = B_{21} + B_{22}$	$P_6 = S_4 \bullet T_4$	
		$P_7 = S_5 \bullet T_5$	

(c)

Pre-additions		Recursive calls	Post-additions	
		$P_1 = A_{11} \bullet B_{11}$	$C_{11} =$	$U_1 = P_1 + P_2$
		$P_2 = A_{12} \bullet B_{21}$		$U_2 = P_1 + P_4$
$S_1 = A_{21} + A_{22}$	$T_1 = B_{12} - B_{11}$	$P_3 = S_1 \bullet T_1$		$U_3 = U_2 + P_5$
$S_2 = S_1 - A_{11}$	$T_2 = B_{22} - T_1$	$P_4 = S_2 \bullet T_2$	$C_{21} =$	$U_4 = U_3 + P_7$
$S_3 = A_{11} - A_{21}$	$T_3 = B_{22} - B_{12}$	$P_5 = S_3 \bullet T_3$	$C_{22} =$	$U_5 = U_3 + P_3$
$S_4 = A_{12} - S_2$	$T_4 = B_{21} - T_2$	$P_6 = S_4 \bullet B_{22}$		$U_6 = U_2 + P_3$
		$P_7 = A_{22} \bullet T_4$	$C_{12} =$	$U_7 = U_6 + P_6$

Figure 1: Three algorithms for matrix multiplication. The symbol $\bullet$represents matrix multiplication. (a) Standard algorithm. (b) Strassen's algorithm. (c) Winograd variant of Strassen's algorithm.

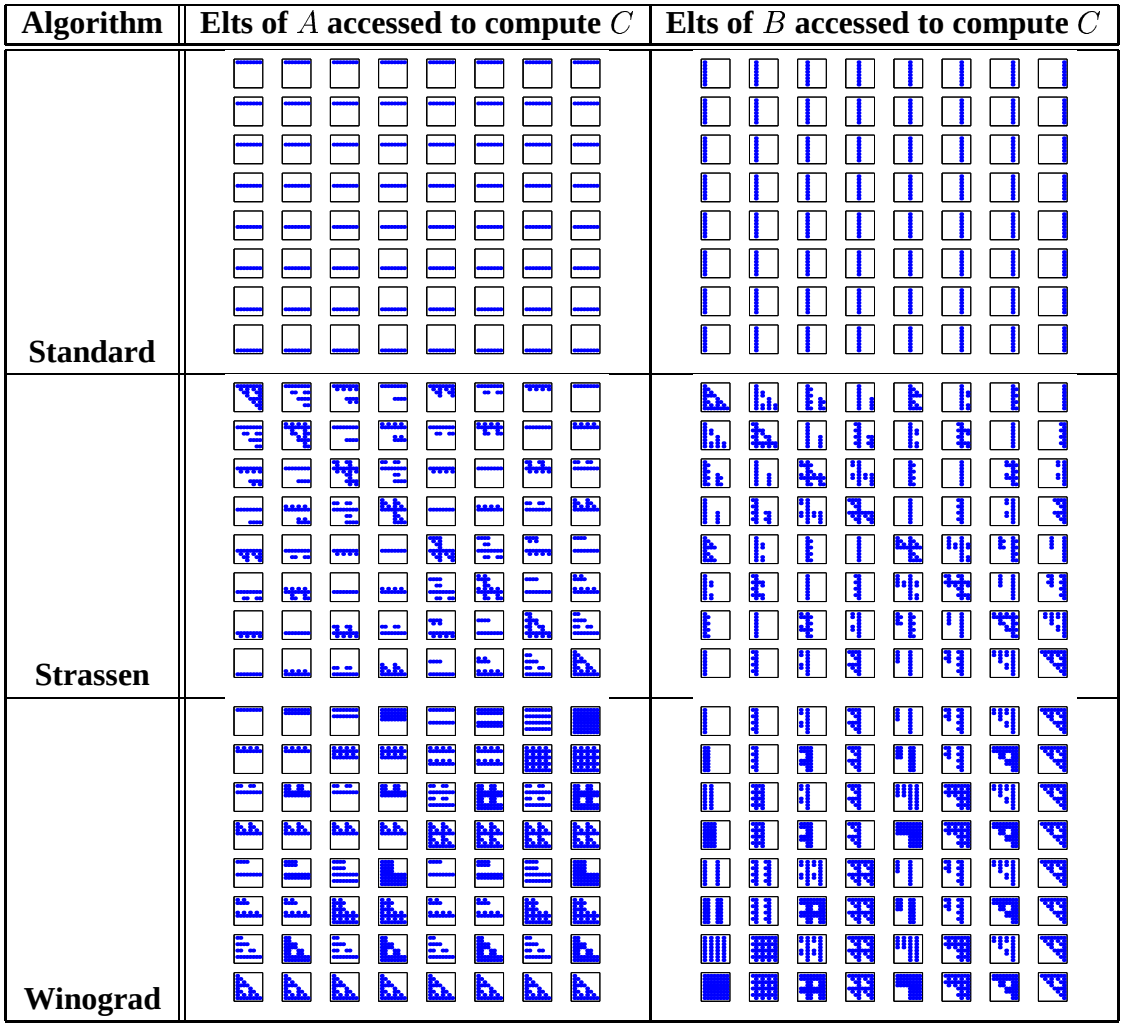


Figure 2: Algorithmic locality of reference of the three matrix multiplication algorithms. The figures show the elements of A and B accessed to compute the individual elements of $C = A \bullet B$, for 8×8 matrices. Each of the six diagrams has an 8×8 grid of boxes, each box representing an element of C. Each box contains an 8×8 grid of points, each point representing an element of matrix A or B. The grid points corresponding to accessed elements are indicated by a dot.

Strassen's algorithm and for elements $(0, 7)$ and $(7, 0)$ for Winograd's. This raises the question of whether the benefits of the reduced number of floating point operations for the fast algorithms would be lost as a result of the increased number of memory accesses.

We do not discuss in this paper numerical issues concerning the fast algorithms, as they are covered elsewhere [25].

We use Cilk [6] to implement parallel versions of these algorithms. The parallelism is exposed in the recursive matrix multiplication calls. Each of the seven or eight calls are spawned in parallel, and these in turn invoke other recursive calls in parallel. Cilk supports this nested parallelism, providing a very clean implementation.

In order to stay consistent with previous work and to permit meaningful comparisons, all our implementations follow the same calling conventions as the `dgemm` subroutine in the Level 3 BLAS library [14].

3 Recursive array layouts

Programming languages that support multi-dimensional arrays must also provide a function (the *layout* function L) to map the array index space into the linear memory address space. We assume a two-dimensional array with m rows and n columns indexed using a zero-based scheme. The results we discuss generalize to higher-dimensional arrays and other indexing schemes. Define the map L such that $L(i, j)$ is the memory offset of the array element in row i and column j from the starting memory address of the array. We list near the end of the argument list of L, following a semicolon, any "structural" parameters (such as m and n) of L, thus: $L(i, j; m, n)$.

3.1 Canonical layout functions

The default layout functions provided in current programming languages are the *row-major* layout L_R as used by Pascal and by C for constant arrays, given by

$$L_R(i, j; m, n) = n \cdot i + j$$

and the *column-major* layout L_C as used by Fortran, given by

$$L_C(i, j; m, n) = m \cdot j + i.$$

Following the terminology of Cierniak and Li [12], we refer to L_R and L_C as *canonical* layout functions. Figure 3(a) and (b) show these two layout functions.

Lemma 1 *The following equalities hold for the canonical layout functions.*

$$L_R(i, j+1; m, n) - 1 = L_R(i, j; m, n) = L_R(i+1, j; m, n) - n \tag{2}$$

$$L_C(i, j+1; m, n) - m = L_C(i, j; m, n) = L_C(i+1, j; m, n) - 1 \tag{3}$$

Proof: Follows by simple algebraic manipulation of the definitions. □

Canonical layouts do not always interact well with cache memories, because the layout function favors one axis of the index space over the other, causing neighbors in the unfavored direction to become distant in memory. This *dilation* effect, which is one interpretation of equations (2)–(3),

has implications for both parallel execution and single-node performance that can reduce program performance.

- In the shared-memory parallel environments in which we experimented, the elements of a quadrant of a matrix are spread out in shared memory, and a single shared memory block can contain elements from two quadrants, and thus be written by the two processors computing those quadrants. This leads to false sharing [13].

- In a message-passing parallel environment such as those used in implementations of High Performance Fortran [35], typical array distributions would again spread a matrix quadrant over many processors, thereby increasing communication costs.

- The dilation effect can compromise single-node memory system performance in the following ways: by reducing or even nullifying the effectiveness of multi-word cache lines; by reducing the effectiveness of translation lookaside buffers (TLBs) for large matrix sizes; and by causing cache misses due to self-interference even when a tiled loop repeatedly accesses a small array tile.

Despite the dilation effect described above, canonical layout functions have one major advantage that is heavily exploited for efficiency in address computation. A different interpretation of equations (2)–(3) reveals that these layouts allow incremental computation of memory locations of elements that are adjacent in array index space. This idiom is understood by compilers and is one of the keys to high performance in libraries such as native BLAS.[2] In defining recursive layout functions, therefore, we will not carry the recursive layout down to the level of individual elements,

[2]The algebra of dilated integers used by Wise [55] might in principle be equally efficient, but is currently not incorporated in production compilers.

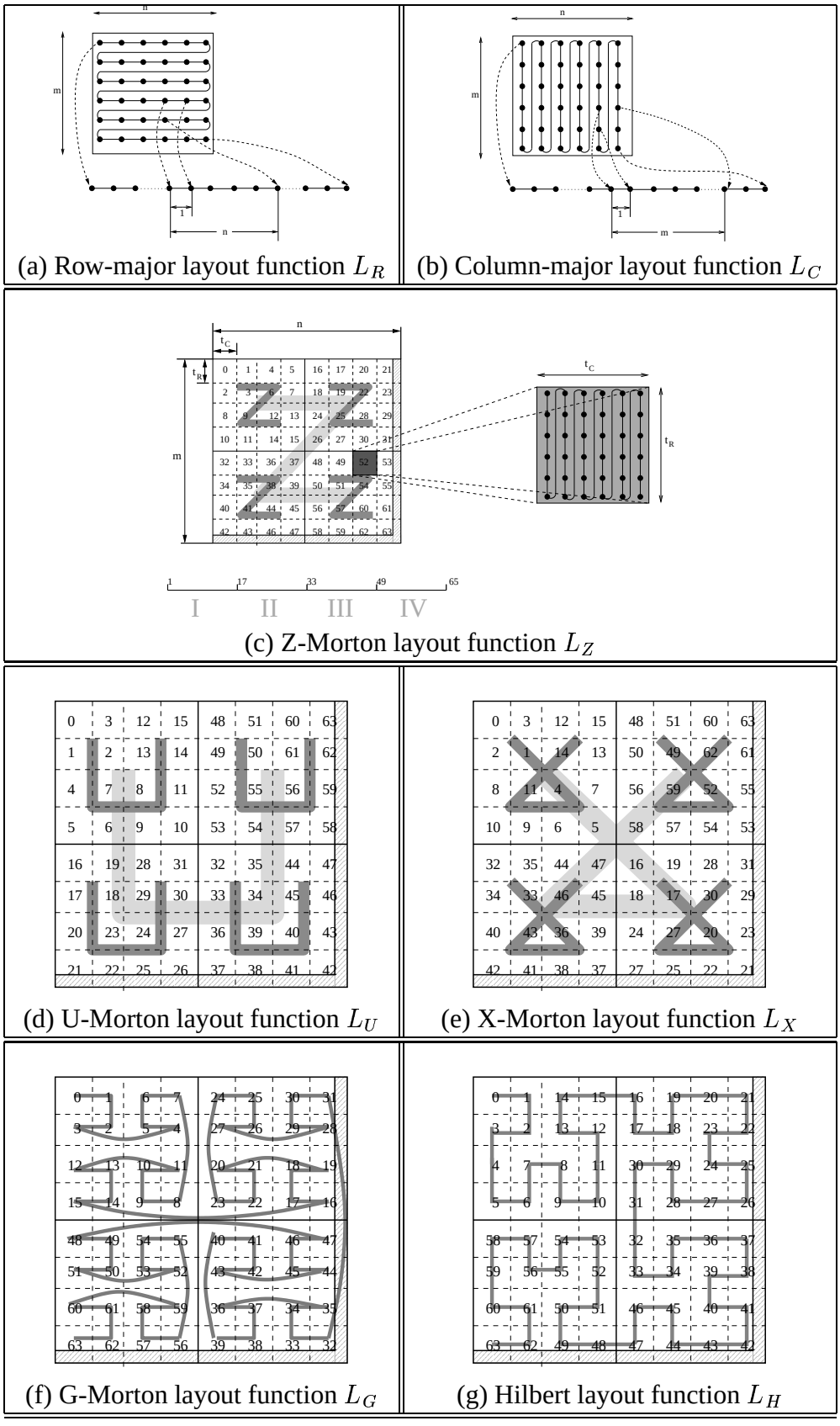


Figure 3: Graphical description of layout functions. Arrays are $m \times n$; tiles are $t_R \times t_C$.

but will instead make the base case be a $t_R \times t_C$ submatrix that fits in cache.

3.2 Recursive layout functions

Assume for the moment that we know how to choose the tile sizes t_R and t_C, and that t_R and t_C simultaneously satisfy

$$\frac{m}{t_R} = \frac{n}{t_C} = 2^d \tag{4}$$

for some positive integer d. (From a quadtree perspective, this means that the quadtree is d levels deep. We relax this constraint in Section 4.) We now view our original $m \times n$ array as a $\frac{m}{t_R} \times \frac{n}{t_C}$ array of $t_R \times t_C$ tiles. Equivalently, we have mapped the original two-dimensional array index space (i, j) into a four-dimensional space

$$(t_i, t_j, f_i, f_j) = (\mathbb{T}(i; t_R), \mathbb{T}(j; t_C), \mathbb{F}(i; t_R), \mathbb{F}(j; t_C))$$

using the (nonlinear) transformations $\mathbb{T}(i; t) = i \operatorname{div} t$ and $\mathbb{F}(i; t) = i \bmod t$. We now partition this four-dimensional space into two two-dimensional subspaces: the space T of tile co-ordinates (t_i, t_j), and the space F of tile offsets (f_i, f_j). We apply the (canonical) column-major layout function L_C in the F-space (to keep each tile contiguous in memory) and a layout function L_T in the T-space (to obtain the starting memory location of the tile), and define our recursive layout function L as their sum, thus.

$$\begin{aligned} L(i, j; m, n, t_R, t_C) &= L(t_i, t_j, f_i, f_j; m, n, t_R, t_C) \\ &= L_T(t_i, t_j; m, n, t_R, t_C) + L_C(f_i, f_j; t_R, t_C) \end{aligned}$$

$$= \quad t_R \cdot t_C \cdot \mathbb{S}(t_i, t_j) + L_C(f_i, f_j; t_R, t_C) \tag{5}$$

where $\mathbb{S}(i, j)$ gives the position along the space-filling curve (*i.e.*, the pre-image) of the element at rectangular co-ordinates (i, j). More precisely, the $2^d \times 2^d$ elements of the matrix of tiles correspond to the nodal points of the d^{th} approximating polygon for the space-filling curve [45, p. 21].

Equation (5) defines a family of layout functions parameterized by the function $\mathbb{S}$ characterizing the space-filling curve. All of these recursive layout functions have the following operational interpretation following from Hilbert's original construction [26] that defined a class of space-filling curves as the limit of a sequence of nested discrete approximations to it. Divide the original matrix into four quadrants, and lay out these submatrices in memory in an order specified by L_T. Use L_T to recursively lay out a $k_R \times k_C$ submatrix with $k_R > t_R$ and $k_C > t_C$; use L_C to lay out a $t_R \times t_C$ tile.

Space-filling curves are based on the idea of threading a region with self-similar line segments at multiple scales. The two fundamental operations involved are scaling and orienting (rotating) the line segments. We classify the five recursive layouts we consider in this paper into three classes based on the number of orientations needed. Three layouts (U-Morton, X-Morton, and Z-Morton) require a single orientation; one layout (Gray-Morton) requires two orientations; and one layout (Hilbert) requires four orientations. We now discuss the structure of these layouts and the computations involved in calculating their $\mathbb{S}$ functions.

We need the following notation to discuss the computational aspects of the recursive layouts. For any non-negative integer i, let $\mathcal{B}(i)$ be the bit string corresponding to its standard binary encoding, and let $\mathcal{G}(i)$ be the bit string corresponding to its Gray code [41] encoding. Correspondingly,

for any bit string s, let $\mathcal{B}^{-1}(s)$ be the non-negative integer i such that $\mathcal{B}(i) = s$, and let $\mathcal{G}^{-1}(s)$ be the non-negative integer i such that $\mathcal{G}(i) = s$, Also, given two bit patterns $u = u_{d-1} \cdots u_0$ and $v = v_{d-1} \cdots v_0$, each of length d, let $u \bowtie v$ be the bit pattern of length $2d$ resulting from the bitwise interleaving of u and v, *i.e.*, $u \bowtie v = u_{d-1} v_{d-1} \cdots u_0 v_0$. Finally, we adopt the convention that, for all layouts, $\mathbb{S}(0, 0) = 0$. If rotations and reflections of the layout functions are desired, they are most cleanly handled by interchanging the i and j arguments and/or subtracting them from $2^d - 1$.

3.2.1 Recursive layouts with a single orientation

The three layouts U-Morton (L_U), X-Morton (L_X), and Z-Morton (L_Z), illustrated in Figure 3(c)–(e), are based on a single repeating pattern of ordering quadrants. The mnemonics derive from the letters of the English alphabet that these ordering patterns resemble. We note that the Z-Morton layout should properly be called the Lebesgue layout, since it is based on Lebesgue's space-filling curve [45, p. 80].

The $\mathbb{S}$ functions for these layouts are easily computed with bit operations, as follows.

$$\text{For } L_U\text{:} \quad \mathbb{S}(i, j) = \mathcal{B}^{-1}(\mathcal{B}(j) \bowtie (\mathcal{B}(i) \text{ XOR } \mathcal{B}(j)))$$

$$\text{For } L_X\text{:} \quad \mathbb{S}(i, j) = \mathcal{B}^{-1}((\mathcal{B}(i) \text{ XOR } \mathcal{B}(j)) \bowtie \mathcal{B}(j))$$

$$\text{For } L_Z\text{:} \quad \mathbb{S}(i, j) = \mathcal{B}^{-1}(\mathcal{B}(i) \bowtie \mathcal{B}(j))$$

3.2.2 Recursive layouts with two orientations

The Gray-Morton layout [38] (L_G) is based on a C-shaped line segment and its counterpart that is rotated by 180 degrees. Figure 3(f) illustrates this layout. Computationally, its $\mathbb{S}$ function is defined as follows: $\mathbb{S}(i, j) = \mathcal{G}^{-1}(\mathcal{G}(i) \bowtie \mathcal{G}(j))$.

3.2.3 Recursive layouts with four orientations

The Hilbert layout [26] (L_H) is based on a C-shaped line segment and its three counterparts rotated by 90, 180, and 270 degrees. Figure 3(g) illustrates this layout. The $\mathbb{S}$ function for this layout is computationally more complex than any of the others. The fastest method we know of is based on an informal description by Bially [4], which works by driving a finite state machine with pairs of bits from i and j, delivering two bits of $\mathbb{S}(i, j)$ at each step. C code performing this computation is shown in Appendix A.

3.3 Summary

We have described five recursive layout functions in terms of space-filling curves. These layouts grow in complexity from the ones based on Lebesgue's space-filling curve to the one based on Hilbert's space-filling curve. We now state several facts of interest regarding the mathematical and computational properties of these layout functions.

- It follows from the pigeonhole principle that only two of the four cardinal neighbors of (i, j) can be adjacent to $\mathbb{S}(i, j)$. Thus, any layout function (canonical, recursive, or otherwise) must necessarily experience a dilation effect. The important difference is that the dilation occurs at multiple scales for recursive layouts. We note that this dilation effect, which is manifest in the abrupt jumps in the curves of Figure 3, gets less pronounced as the number of orientations increases.

- We do not know of a recursive layout with three orientations. There are, however, space-filling curves appropriate for triangular or trapezoidal regions. We do not know whether such curves would be useful for laying out triangular matrices.

- The L_G layout function has a useful symmetry that is easiest to appreciate visually. Refer to Figure 3(f) and observe the northwest quadrant (tiles 0–15) and the southeast quadrant (tiles 32–47), which have different orientations. If we remove the single edge between the top half and the bottom half of each quadrant (edge 7–8 for the northwest quadrant, edge 39–40 for the southeast quadrant), we note that the top and bottom halves of the two quadrants are identically oriented. That is, two quadrants of opposite orientation differ only in the order in which their top and bottom halves are "glued" together. We exploit this symmetry in Section 4.

- In terms of computational complexity of the $\mathbb{S}$ functions of the different layout functions, we observe that bits $2u+1$ and $2u$ of $\mathbb{S}(i,j)$ depend only on bit u of i and j for the layouts with a single orientation, while they depend on bits u through $d-1$ of i and j for the L_G and L_H layouts.

4 Implementation issues

Section 2 described the parallel recursive control structure of the matrix multiplication algorithms, while Section 3 described recursive data layouts for arrays. This section discusses how our implementation combines these two aspects.

4.1 A naive strategy

A naive but correct implementation strategy is to follow equation (5) and, for every reference to array element $A(i,j)$ in the code, to insert a call to the address computation routine for the appropriate recursive layout. However, this requires integer division and remainder operations to

compute (t_i, t_j, f_i, f_j) from (i, j) at each access, which imposes an unreasonably large overhead. To gain efficiency, we need to exploit the decoupling of the layout function L shown in equation (5). Once we have located a tile and are accessing elements within it, we do not recompute the starting offset of the tile. Instead, we use the incremental addressing techniques supported by the canonical layout L_C. The following lemma formalizes this observation.

Lemma 2 *Let L be as defined in equation (5).*

- *If $t_{i+1} = t_i$, then $L(i+1, j; m, n, t_R, t_C) = L(i, j; m, n, t_R, t_C) + 1$.*
- *If $t_{j+1} = t_j$, then $L(i, j+1; m, n, t_R, t_C) = L(i, j; m, n, t_R, t_C) + m$.*

Proof: It follows from the definitions of tile co-ordinate t_i and tile offset f_i that, if $t_{i+1} = t_i$, then $f_{i+1} = f_i + 1$. Then we have

$$\begin{aligned} L(i+1, j; m, n, t_R, t_C) &= t_R \cdot t_C \cdot \mathbb{S}(t_{i+1}, t_j) + L_C(f_{i+1}, f_j; t_R, t_C) \\ &= t_R \cdot t_C \cdot \mathbb{S}(t_i, t_j) + L_C(f_i + 1, f_j; t_R, t_C) \\ &= t_R \cdot t_C \cdot \mathbb{S}(t_i, t_j) + L_C(f_i, f_j; t_R, t_C) + 1 \\ &= L(i, j; m, n, t_R, t_C) + 1. \end{aligned}$$

The proof of the second equality is analogous. □

4.2 Integration of address computation into control structure

Lemma 2 allows us to exploit the incremental address computation properties of the L_C layout in the recursive setting, but requires the $\mathbb{S}$ function to be computed from scratch for every new tile.

For the matrix multiplication algorithms discussed in Section 2, we can further reduce addressing overhead by integrating the computation of the $\mathbb{S}$ function into the control structure in an incremental manner, as follows. Observe that the actual work of matrix multiplication happens on $t_R \times t_C$ tiles when the recursion terminates. At each recursive step, we need to locate the quadrants of the current trio of (sub)matrices, perform pre-additions on quadrants, spawn the parallel recursive calls, and perform the post-additions on quadrants. The additions have no temporal locality and are ideally suited to streaming through the memory hierarchy. Such streaming is aided by the fact that recursive layouts keep quadrants contiguous in memory. Therefore, all we need is the ability to quickly locate the starting points of the four quadrants of a (sub)matrix. This produces the correct $\mathbb{S}$ number of the tiles when the recursion terminates, which are then converted to memory addresses and passed to the leaf matrix multiplication routine.

For the recursive layout functions with multiple orientations, we need to retain both the location and the orientation of quadrants as we go through multiple levels of divide-and-conquer. We encode orientation in one or two most significant bits of the integers. Appendix B describes these computations for each of the five layouts.

4.3 Relaxing the constraint of equation (4)

The definitions of the recursive matrix layouts in Section 3 assumed that t_R and t_C were constrained as described in equation (4). This assumption does not hold in general, and the conceptual way of fixing this problem is to pad the matrix to an $m' \times n'$ matrix that satisfies equation (4). There are two concrete ways to implement this padding process.

- Frens and Wise keep a flag at internal nodes of their quad-tree representation to indicate

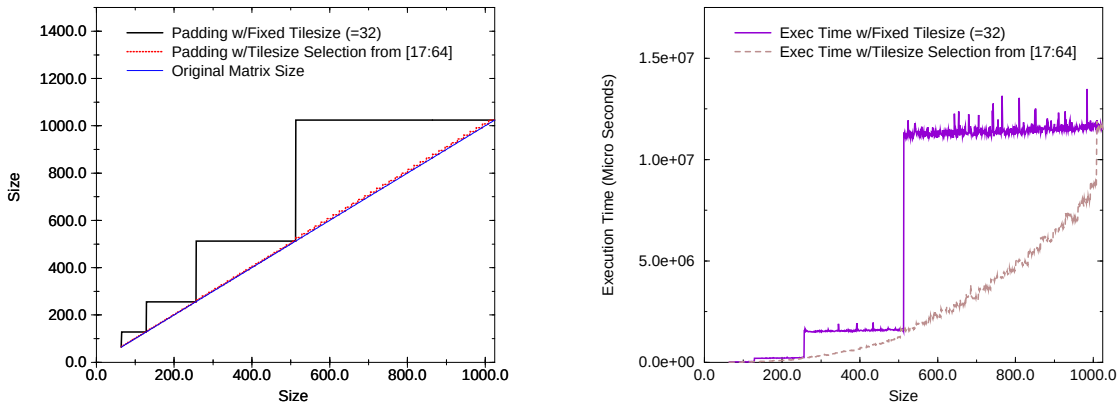
empty or nearly full subtrees, which "directs the algebra around zeroes (as additive identities and multiplicative annihilators)" [18, p. 208].

Maintaining such flags makes this solution insensitive to the amount of padding (in the sense that no additional arithmetic is performed), but requires maintaining the internal nodes of the quad-tree and inroduces additional branches at runtime. This scheme is particularly useful for sparse matrices, where patches of zeros can occur in arbitrary portions of the matrices. Note that if one carries the quad-tree decomposition down to individual elements, then $m' \approx 2m$ and $n' \approx 2n$ in the worst case.

- We choose the strategy of picking t_R and t_C from an architecture-dependent range, explicitly inserting the zero padding, and performing all the additional arithmetic on the zeros. We choose the range of acceptable tile sizes so that the tiles are neither too small (which would increase the overhead of recursive control) nor overflow the cache (which would result in capacity misses). The contiguity of tiles in memory eliminates self-interference misses, which in turn makes the performance of the leaf-level matrix multiplications almost insensitive to the tile size [50].

  Our scheme is very sensitive to the amount of padding, since it performs redundant computations on the padded portions of the matrices. However, if we choose tile sizes from the range $[T_{\min}, T_{\max}]$, the maximum ratio of pad to matrix size is $1/T_{\min}$.

Figure 4(a) shows how m' corresponding to different padding schemes tracks m for values of m between 150 and 1024. Figure 4(b) shows the execution time of matrix multiplication of two $m \times m$ matrices after padding them out to $m' \times m'$ matrices (including computations on the padded zeroes). Choosing the tilesize from a range, with $T_{min} = 17$ and $T_{max} = 64$, we see that m' tracks



(a) (b)

Figure 4: Effect of padding policy on matrix size and execution time. (a) Padded matrix size *vs.* original matrix size. (b) Execution time *vs.* original matrix size.

m quite closely. Consequently, we see that the redundant computations performed on the padded elements do not cause any significant increase in execution time. In contrast, if one were to use a fixed tilesize of 32, for example, $m' \approx 2m$ for some values of m. The additional redundant arithmetic operations implied by this large padding has a significant impact on the execution time as well.

4.4 Effect of padding on space/execution time

Our imposition of the range $[T_{\min}, T_{\max}]$ of tile sizes is guided by cache considerations, but causes a problem for rectangular matrices whose aspect ratio m/n is either too large or too small. Let $\alpha = T_{\max}/T_{\min}$, and call a matrix *wide*, *squat*, or *lean* depending on whether $\alpha < m/n$, $1/\alpha \leqslant m/n \leqslant \alpha$, or $1/\alpha > m/n$.

Lemma 3 *For wide and lean matrices, it is not possible to find tile sizes that simultaneously satisfy*

the constraints of equation (4) and lie in the prescribed range of tile sizes.

Proof: The proof is by contradiction. Let $m/n > \alpha$, and assume that we can find t_R and t_C as desired. From equation (4), we have $m/n = t_R/t_C$. Combining the range constraints and equation (4), we get $T_{\min} \leqslant t_C < t_R \leqslant T_{\max}$. But this gives $t_R/t_C < \alpha$. The case $1/\alpha > m/n$ is analogous. □

This problem can be appreciated by considering the following case: $m = 1024$, $n = 256$, $T_{\min} = 17$, and $T_{\max} = 32$.

The resolution of this problem is quite simple. We divide the wide or lean matrix into squat submatrices, and reconstruct the matrix product in terms of the submatrix products. There is no unique subdivision of a lean or wide matrix into squat submatrices. For example, if we consider $\alpha = 2.0$, and a wide matrix with dimensions $m = 10$ and $k = 50$, the 10×50 matrix can be subdivided into three squat submatrices of 10×20, 10×20, and 10×10. The same matrix can also be subdivided into four squat submatrices of dimensions 10×12, 10×13, 10×12, and 10×13. In our implementation, we repeatedlly sub-divide wide and lean matrices by factors of 2 till they become squat. Thus, our implementation would pick the latter decomposition into four submatrices for the 10×50 example described above.

Figure 5(a) and Figure 5(b) show two examples of how the input matrices A and B are divided, and how the result C is reconstructed from results of submatrix multiplications. These figures make the simplifiying assumption that subdividing the matrices to two equal halves is enough to get squat submatrices. If this assumption does not hold, we apply the subdivision procedure recursively. For brevity, we do not describe all possible cases (the cross product of $\{LeanA, SquatA, WideA\}$ and $\{LeanB, SquatB, WideB\}$). These multiple submatrix multiplications are, of course, spawned

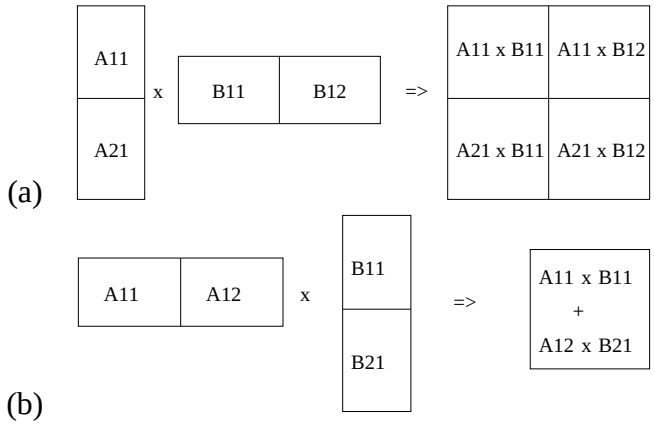


Figure 5: Handling of lean and wide matrices. (a) Lean A and wide B. (b) Wide A and lean B.

to execute in parallel.

4.5 Conversion and transposition issues

In order to stay compatible with `dgemm`, we assume that all matrices are presented in column-major layout. Our implementation internally allocates additional storage and converts the matrices from column-major to the recursive layout. The remapping of the individual tiles is again amenable to parallel execution. We incorporate any matrix transposition operations required by the operation into this remapping step. (The experimental data in Section 5 includes these format conversion times, and Figure 13 quantifies these overheads.) This is handy, because it requires only a single routine for the core matrix multiplication algorithm. The alternative solution would require multiple code versions or indirection to handle the multiple cases correctly.

4.6 Issues with pre- and post-additions

There is one final implementation issue arising from the interaction of the pre- and post-additions with the recursive layouts with more than one orientation. Consider, for example, the pre-addition

$S_1 = A_{11} + A_{22}$ in Strassen's algorithm. For recursive layouts with a single orientation, we simply stream through the appropriate number of elements from the starting locations of A_{11} and A_{22}, adding them and streaming them out to S_1. This is not true for L_G and L_H layouts, since the orientations of A_{11} and A_{22} are different. In other words, while each tile (and the entire set of tiles) is contiguous in memory, the corresponding sub-tiles and elements of A_{11} and A_{22} are not at the same relative position.

For L_G, the fix turns out to be very simple, exploiting the symmetry discussed in Section 3.3. If the orientations of the two tiles are different, and each quadrant contains $2k$ tiles, then the ordering of tiles in one orientation is $T_1, \ldots, T_{2k}$ while the ordering of tiles in the other orientation is $T_{k+1}, \ldots, T_{2k}, T_1, \ldots, T_k$. Therefore, we simply need to perform the pre- and post-additions in two half-steps.

For L_H, the situation is more complicated, because there is no simple pattern to the ordering of tiles. Instead, we simply keep global lookup tables of these orders for the various orientations, and use these tables to identify corresponding tiles in pre- and post-additions. Appendix C shows the code for initializin these tables. The added cost in loop control appears to be insignificant.

5 Experimental results

Our experimental platform was a Sun Enterprise 3000 SMP with four 170 MHz UltraSPARC processors, and 384 MB of main memory, running SunOS 5.5.1. We used version 5.2.1 of the Cilk system [6] compiled with critical path tracking turned off. The Cilk system requires the use of `gcc` to compile the C files that its `cilk2c` compiler generates from Cilk source. We used the gcc-2.7.2 compiler with optimization level `-O3`. The experimental machine was otherwise idle. We also

took multiple measurements of every data point to further reduce measurement uncertainty.

We timed the full cross-product of the three algorithms (standard, Strassen, Winograd) and the six layout functions (L_C, L_U, L_X, L_Z, L_G, L_H) running on one through four processors on square matrices with n ranging from 500 through 1500. We verified correctness of our codes by comparing their outputs with the output of vendor-supplied native version of `dgemm`, However, we could not perform the leaf-level multiplications in our codes by calling the vendor-supplied native version of `dgemm`, since we could not get Cilk to support such linkage of external libraries. Instead, we coded up a C version of a 6-loop tiled matrix multiplication routine with the innermost accumulation loop unrolled four-way. We report all our results in terms of execution time, rather than megaflop/s (which would not correctly account for the padding we introduce) or speedup (since the values are sensitive to the baseline).

In another set of experiments, we evaluated a sequential implementation in which we see a trade-off between memory and execution time. We also compare this sequential implementation with two state-of-the-art fast matrix multiplication implementations.

5.1 General comments

As predicted by theory, we observed the two fast algorithms consistently outperforming the standard algorithm. This is apparent from the different y-axis extents in the subgraphs of Figure 8. From the same figure, we observe virtually no difference between the execution times of the two fast algorithms. This suggests to us that the worse algorithmic locality of reference of Winograd's algorithm compared to Strassen's (see Figure 2) offsets its advantage of lower operation count.

We observed near-perfect scalability for all the codes, as evident from Figures 7 and 8. By en-

abling critical path tracing in Cilk, we separately determined that, for $n = 1000$, there is sufficient parallelism in the standard algorithm to keep about 40 processors busy; the corresponding number for the two fast algorithms is around 23. This is as expected, since the total work of the algorithms is $O(n^{2+\delta})$, while the critical path is $O(\lg^2 n)$.

5.2 Choice of tile size

To back our claim that, for best performance, the recursive layouts should be stopped before the matrix element level, we timed a version of the standard algorithm with the L_Z layout in which we explicitly controlled the tile size at which the recursive layout stopped. Figure 6 shows the single-processor execution times from this experiment with: $n = 1024$, $t \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$; and $n = 1536$, $t \in \{3, 6, 12, 24, 48, 96, 192, 384, 768\}$. (We used these values of n because they allow us to choose many tile sizes without incurring any padding.) The results for multiple processor runs are similar. The shape of the plot confirms our claim. The "bump"s in the curves are reproducible.

For reference, the native `dgemm` routine runs for $n = 1024$ in 17.874 seconds. Thus, our best time of 33.609 seconds (at a tile size of 16) puts us at a slowdown factor of 1.88. The numbers for $n = 1536$ are 61.555 seconds for native `dgemm`, 96.1996 seconds for our best time, and a slowdown factor of 1.56.

5.3 Robustness of performance

To study the robustness of the performance of the various algorithms, we timed the standard and Strassen algorithms using the L_C and L_Z layouts for $n \in [1000, 1048]$ on 1–4 processors. Figure 7

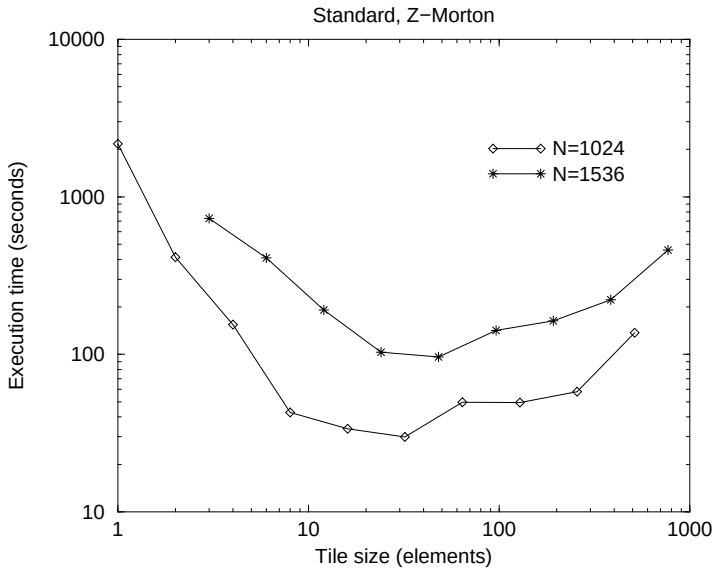


Figure 6: Effect of depth of recursive layout on performance. Standard algorithm, L_Z layout, $n = 1024$, one processor. Note that both axes are logarithmic.

shows the results, which are unlike what we had originally expected. The standard algorithm with L_C layout exhibits large performance swings which are totally reproducible. The L_Z layout greatly reduces this variation but does not totally eliminate it. In stark contrast, Strassen's algorithm does not display such fluctuation for either layout. In neither case do we observe a radical performance loss at $n = 1024$, which is what we originally expected to observe. The fluctuations for the standard algorithm appear to be an artifact of paging, although we have not yet been able to confirm this hypothesis. We offer our explanation of the robustness of Strassen's algorithm in Section 5.5.

5.4 Relative performance of different layouts

Figure 8 shows the relative performance of the various layout functions at two problem sizes: $n = 1000$ and $n = 1200$. The scaling is near-perfect for all the codes. The figures reveal two major points. First, compared to the L_C layout, the effect of recursive layouts on the standard algorithm

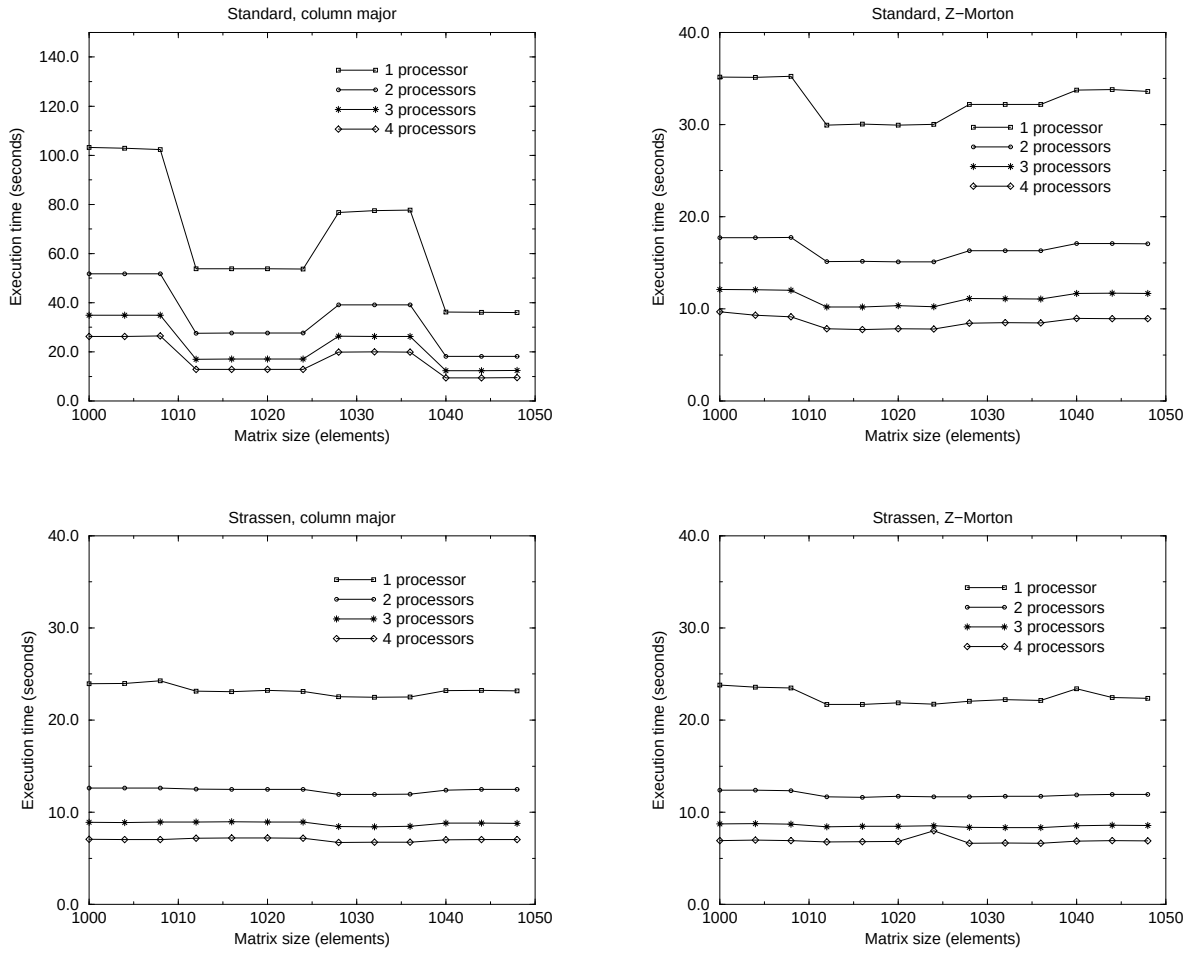


Figure 7: Performance of the standard algorithm and Strassen's algorithm, using L_C and L_Z layouts, for $n \in [1000, 1048]$, on 1–4 processors.

is dramatic, while their effect on the two fast algorithms is marginal. We offer our explanation of this effect in Section 5.5. Second, at least for these problem sizes, the performance of all the recursive layouts is approximately the same. We interpret this to mean that our implementation of the layouts is sufficiently efficient to control the addressing overheads even of L_H. An alternate explanation is that the purported benefits of, say, L_H over L_Z, do not manifest themselves until we reach even larger problem sizes.

5.5 Why the fast algorithms behave differently than the standard algorithm

Our explanation for the qualitative difference in the behavior of the fast algorithms compared to the standard algorithm is an algorithmic one. Observe that both the Strassen and Winograd algorithms perform pre-additions, which require the allocation of quadrant-sized temporary storage, while this is not the case with the standard algorithm. Therefore, when performing the leaf-level matrix products, the standard algorithm works with tiles of the original input matrices, which have leading dimension equal to n. In contrast, every level of recursion in the fast algorithms reduces the leading dimension by a factor of approximately two, *even if we do not re-structure the matrix at the top level*. This intrinsic feature of the fast algorithms makes them insensitive to the parameters of the memory system.

5.6 A sequential version : parallelism-space trade-off

For parallel execution of the recursive multiplication, it was necessary to have "live" copies of all the pre-addition results to allow the recursive calls to execute in parallel. For a sequential computation, where one wishes to conserve space, one would intersperse recursive calls with pre-

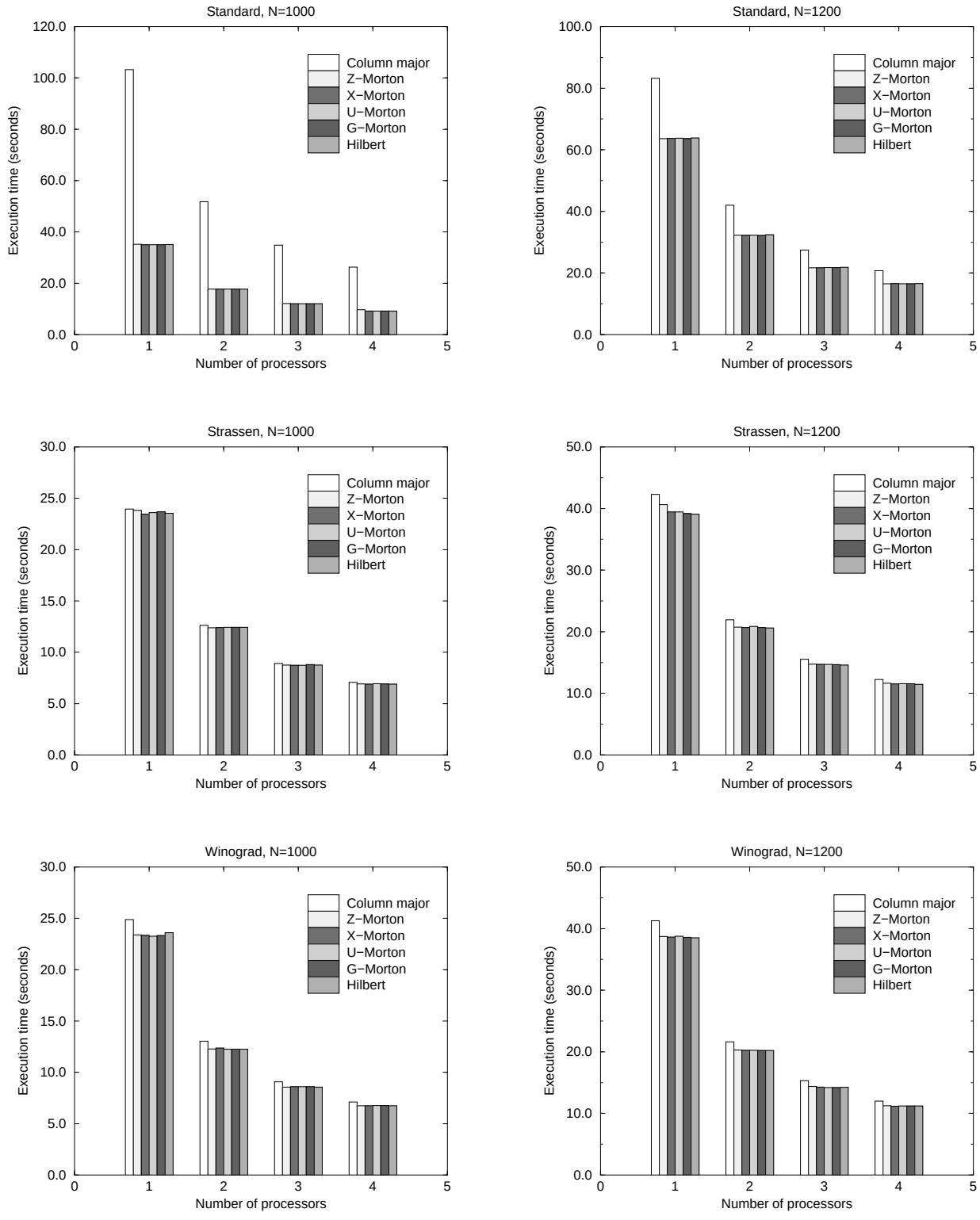


Figure 8: Comparative performance of the six layouts. The left column is for $n = 1000$, while the right column is for $n = 1200$. The rows from top to bottom are for the standard, Strassen, and Winograd algorithms.

and post-additions. Such a reordering of the schedule reduces the number of temporaries that are "live". This version also behaves more like the standard algorithm with respect to recursive layouts: L_Z reduces execution times by 10–20%.

This version is a purely sequential implementation and does not use Cilk. To understand how the performance of matrix multiplication with recursive layouts compares with other state-of-the-art sequential matrix multiplication implementations, we compare this version with two other matrix multiplication implementations. Neither of these competing implementations uses padding. Instead, they use other techniques to deal with the problem of matrix subdivision.

One implementation [30], hereafter referred to as DGEFMM, uses *dynamic peeling*. This approach peels off the extra row or column at each level, and separately adds their contributions to the overall solution in a later fix-up computation. This eliminates the need for extra padding, but reduces the portion of the matrix to which Strassen's algorithm applies, thus reducing the potential benefits of the recursive strategy. The fix-up computations are matrix-vector operations (level 2 BLAS) rather than matrix-matrix operations (level 3 BLAS), which limits the amount of reuse and reduces performance.

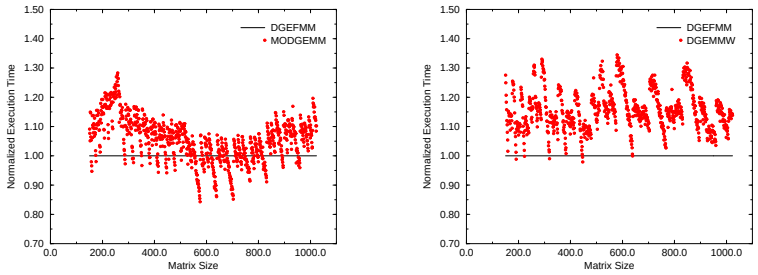
The other implementation [15], hereafter referred to as DGEMMW, uses *dynamic overlap*. This approach finesses the problem by subdividing the matrix into submatrices that (conceptually) overlap by one row or column, computing the results for the shared row or column in both subproblems, and ignoring one of the copies. This is an interesting solution, but it complicates the control structure and performs some extra computations.

We measure the execution time of the various implementations on a 500 MHz DEC Alpha Miata and a 300 MHz Sun Ultra 60. The Alpha machine has a 21164 processor with an 8KB direct-mapped level 1 cache, a 96KB 3-way associative level 2 cache, a 2MB direct-mapped level

3 cache, and 512MB of main memory. The Ultra has two UltraSPARC II processors, each with a 16 KB level 1 cache, a 2MB level 2 cache, and 512MB of main memory. We use only one processor on the Ultra 60.

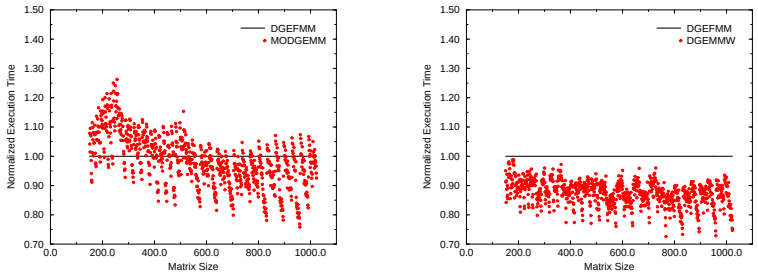
We timed the execution of each implementation using the UNIX system call `getrusage` for square matrix sizes ranging from 150 to 1024, and `dgemm` parameters $\alpha = 1$ and $\beta = 0$. We performed similar measurements for rectangular matrices with the common dimension fixed at 1000. (We measure the time to compute an $m \times m$ matrix $C = A \bullet B$, where A is an $m \times 1000$ matrix and B is an $1000 \times m$ matrix.) For DGEFMM we use the empirically determined recursion truncation point of 64. For matrices of size less than 500, we compute the average of 10 invocations of the algorithm to overcome limits in clock resolution. Execution times for larger matrices are large enough to overcome these limitations. To further reduce experimental error, we execute the above experiments three times for each matrix size, and use the minimum value for comparison. The programs were compiled with vendor compilers (cc and f77) with the -fast option. The Sun compilers are the Workshop Compilers 4.2, and the DEC compilers are DEC C V5.6-071 and DIGITAL Fortran 77 V5.0-138-3678F.

Figure 9 and Figure 10 show our results for square matrices for the Alpha and UltraSPARC, respectively. We report results in execution time normalized to the dynamic peeling implementation (DGEFMM). On the Alpha, we see that DGEFMM generally outperforms dynamic overlap (DGEMMW), see Figure 9(b). In contrast, our implementation (MODGEMM) varies from 30% slower to 20% faster than DGEFMM. We also observe that MODGEMM outperforms DGEFMM mostly in the range of matrix sizes from 500 to 800, whereas DGEFMM is faster for smaller and larger matrices. Finally, by comparing Figure 9(a) and Figure 9(b), we see that MODGEMM generally outperforms DGEMMW.



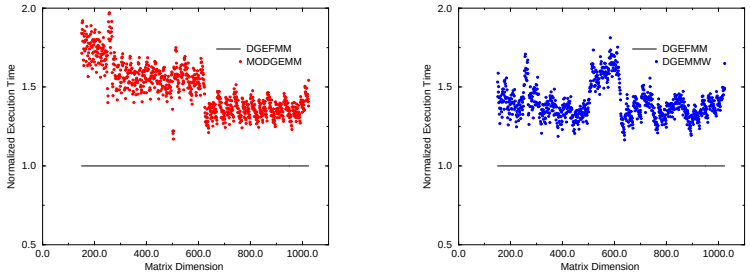
(a) (b)

Figure 9: Performance of Strassen-Winograd implementations on Dec Miata, $\alpha = 1, \beta = 0$. (a) MODGEMM *vs.* DGEFMM. (b) DGEMMW *vs.* DGEFMM.



(a) (b)

Figure 10: Performance of Strassen-Winograd implementations on Sun Ultra 60, $\alpha = 1, \beta = 0$. (a) MODGEMM *vs.* DGEFMM. (b) DGEMMW *vs.* DGEFMM.



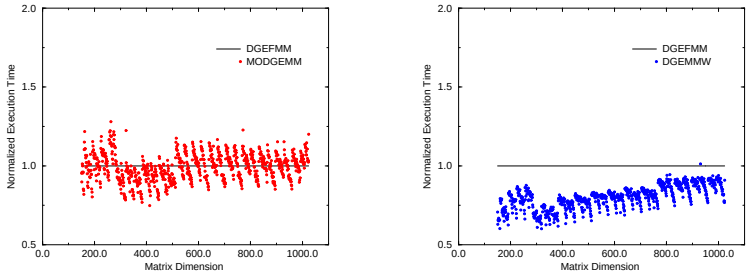
(a) (b)

Figure 11: Performance of Strassen Winograd Implementations on DEC Miata for rectangular matrices (m : Matrix Dimension, $A : m \times 1000, B : 1000 \times m, C[= A \bullet B] : m \times m$), $\alpha = 1, \beta = 0$. (a) MODGEMM *vs.* DGEFMM. (b) DGEMMW *vs.* DGEFMM.

The results are quite different on the Ultra (see Figure 10). The most striking difference is the performance of DGEMMW (see Figure 10(b)), which outperforms both MODGEMM and DGEFMM for most matrix sizes on the Ultra. Another significant difference is that MODGEMM is generally faster than DGEFMM for large matrices ($n = 500$ and larger), while DGEFMM is generally faster for small matrices.

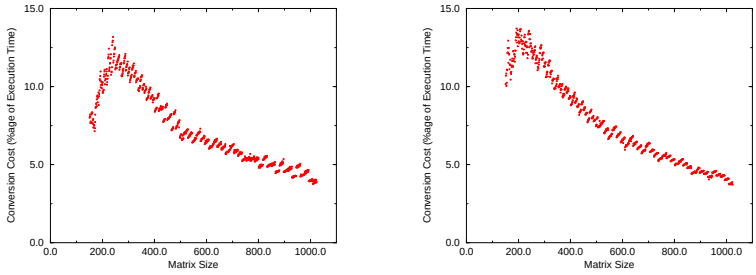
The results for the rectangular matrices are somewhat similar to that of square matrices. On the DEC Miata, DGEFMM comprehensively outperforms DGEMMW as with square matrices (see Figure 11b). But MODGEMM does not outperform DGEFMM at any matrix size (see Figure 11a). Again, on the Sun Ultra 60, DGEMMW outperforms both MODGEMM and DGEFMM as with square matrices (see Figure 12a and Figure 12b). We also see that MODGEMM is faster than DGEFMM at many sizes though no clear trend is obvious.

The final set of results shown in Figure 13 quantify the overhead of format conversion on the two machines. We represent this overhead as a fraction of the total running time. The trends on both machines are similar. The overhead is about 12% for small matrix sizes, and diminishes smoothly



(a) (b)

Figure 12: Performance of Strassen Winograd Implementations on Sun Ultra 60 for rectangular matrices (m : Matrix Dimension, $A : m \times 1000, B : 1000 \times m, C[= A \bullet B] : m \times m$), $\alpha = 1, \beta = 0$. (a) MODGEMM *vs.* DGEFMM. (b) DGEMMW *vs.* DGEFMM.



(a) (b)

Figure 13: Overhead of format conversion as percentage of running time, for square matrices. (a) Sun Ultra 60. (b) DEC Miata.

to about 4% for a matrix size of about 1000. We attribute the decreasing pattern to the fact that format conversion involves $\Theta(n^2)$ operations while the multiplication involves an asymptotically higher number of operations.

5.7 A critique of Cilk

Overall, we were favorably impressed with the capabilities of the Cilk system [6] that we used to parallelize our code. For a research system, it was quite robust. The simplicity of the language

extensions made it possible for us to parallelize our codes in a very short time. The restriction of not being able to call Cilk functions from C functions, while sound in its motivation, was the one feature that we found annoying, for a simple reason: it required annotating several intervening C functions in a call chain into Cilk functions, which appeared to us to be spurious. This problem is avoidable by using the library version of Cilk.

The intimate connections between Cilk and `gcc`, and the limitations on linking non-Cilk libraries, limit achievable system performance. In order to quantify these performance losses, we compiled our serial C codes with three different sets of compile and link options: (i) a baseline version compiled with the vendor `cc` (Sun's Workshop Compilers Version 4.2), with optimization level `-fast`, and linked against the native `dgemm` routine from Sun's perflib library for the leaf-level matrix multiplications; (ii) a version compiled with the vendor `cc` with optimization level `-fast`, but with our C routine instead of the native `dgemm`; and (iii) a version compiled with `gcc` version 2.7.2, with optimization level `-O3`, and with our C routine instead of the native `dgemm`. Figure 14 summarizes our measurements with these three versions for several problem sizes, algorithms, and layout functions. The results are quite uniform: the lack of native BLAS costs us a factor of 1.2–1.4, while the switch to `gcc` costs us a factor of 1.5–1.9. It is interesting that the incremental loss in performance due to switching compilers is comparable to the loss in performance due to the non-availability of native BLAS. The single-processor Cilk running times are indistinguishable from the running times of version (iii) above, suggesting an extremely efficient implementation of the Cilk runtime system.

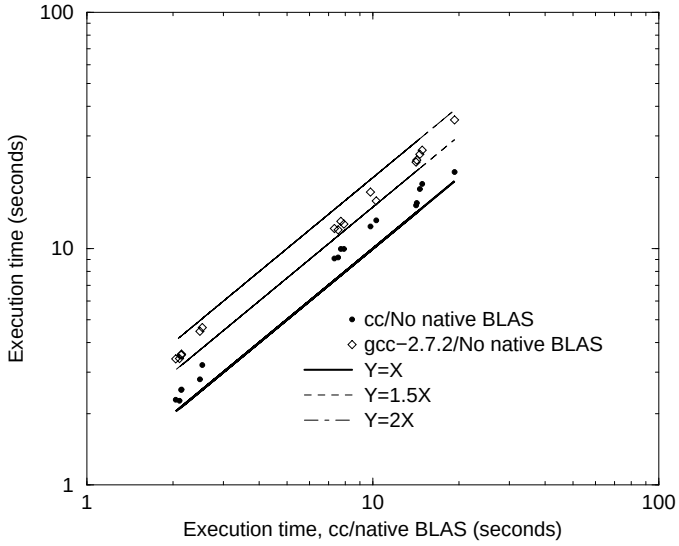


Figure 14: Overhead of gcc and of non-native BLAS.

6 Related work

We categorize related work into two categories: previous application of recursive array layout functions in scientific libraries and applications, and work in the parallel systems community related to language design and iteration space tiling for parallelism.

6.1 Scientific libraries and applications

Several projects emphasize the generation of self-tuning libraries for specific problems. We discuss three such efforts: PHiPAC [5], ATLAS [54], and FFTW [19]. The PHiPAC project aims at producing highly tuned code for specific BLAS 3 [14] kernels such as matrix multiplication that are tiled for multiple levels of the memory hierarchy. Their approach to generating an efficient code is to explicitly search the space of possible programs, to test the performance of each candidate code by running it on the target machine, and selecting the code with highest performance. It appears

that the code they generate is specialized not only for a specific memory architecture but also for a specific matrix size. The ATLAS project generates code for BLAS 3 routines based on the result that all of these routines can be implemented efficiently given a fast matrix multiplication routine. The FFTW project explores fast routines for one- and multi-dimensional fast Fourier transforms. None of these projects explicitly use data restructuring, although the FFTW project recognizes their importance.

Several authors have investigated algorithmic restructuring for dense linear algebra computations. We have already discussed the contributions of Wise *et al.* [18, 55, 56]. Toledo [51] investigated the issue of locality of reference for LU decomposition with partial pivoting, and recommended the use of recursive control structures. Gustavson *et al.* [1, 16, 23, 53] have independently explored recursive control strategies for various linear algebraic computations, and IBM's ESSL library [31] incorporates several of their algorithms. In addition, Gustavson [24] has devised recursive data structures for representing matrices. Stals and Rüde [48] investigate algorithmic restructuring techniques for improving the cache behavior of iterative methods, but do not investigate recursive data reorganization.

The goal of out-of-core algorithms [20, 39] is related to ours. However, the constraints differ in two fundamental ways from ours: the limited associativity of caches and their fixed replacement policies are not relevant for virtual memory systems; and the access latencies of disks are far greater than that of caches. These differences lead to somewhat different algorithms. Sen and Chatterjee [46] formally link the cache model with the out-of-core model.

The application of space-filling curves is not new to parallel processing, although most of the applications of the techniques have been tailored to specific application domains [2, 28, 29, 44, 47, 52]. They have also been applied for bandwidth reduction in information theory [4], for

graphics applications [21, 37], and for database applications [32]. Most of these applications have far coarser granularity than our target computations. We have shown that the overheads of these layouts can be reduced enough to make them useful for fine-grained computations.

6.2 Parallel languages and compilers

The parallel compiler literature contains much work on iteration space tiling for gaining parallelism [58] and improving cache performance [7, 57]. Carter *et al.* [8] discuss hierarchical tiling schemes for a hierarchical shared memory model. Lam, Rothberg, and Wolf [36] discuss the importance of cache optimizations for blocked algorithms. A major conclusion of their paper was that "it is beneficial to copy non-contiguous reused data into consecutive locations". Our recursive data layouts can be viewed as an *early binding* version of this recommendation, where the copying is done possibly as early as compile time.

The class of data-parallel languages exemplified by High Performance Fortran (HPF) [35] recognizes the fact that co-location of data with processors is important for parallel performance, and provides user directives such as `align` and `distribute` to re-structure array storage into forms suitable for parallel computing. The recursive layout functions described in this paper can be fitted into this memory model using the `mapped` distribution supported in HPF 2.0. Hu *et al.*'s implementation [28] of a tree-structured N-body simulation algorithm manually incorporated L_Z within HPF in a similar manner. Support for the recursive layouts could be formally added to HPF without much trouble. The more critical question is how well the corresponding control structures (which are most naturally described using recursion and nested dynamic spawning of computations) would fit within the HPF framework.

A substantial body of work in the parallel computing literature deals with layout optimization of arrays. Representative work includes that of Mace [40] for vector machines; of various authors investigating automatic array alignment and distribution for distributed memory machines [9, 22, 33, 34]; and of Cierniak and Li [12] for DSM environments. The last paper also recognizes the importance of joint control and data optimization.

7 Conclusions

We have examined the combination of five recursive layout functions with three parallel matrix multiplication algorithms. We have demonstrated that addressing using these layout functions can be accomplished cheaply, and that these address computations can be performed implicitly and incrementally by embedding them in the control structure of the algorithms. We have shown that, to realize maximum performance, such recursive layouts need to co-exist with canonical layouts, and that this interfacing can be performed efficiently. We observed no significant performance variations among the different layout functions. Finally, we observed a fundamental qualitative difference between the standard algorithm and the fast ones in terms of the benefits of recursive layouts, which we attribute to the algorithmic feature of pre-additions.

References

[1] B. S. Andersen, F. Gustavson, J. Wasniewski, and P. Yalamov. Recursive formulation of some dense linear algebra algorithms. In B. Hendrickson, K. A. Yelick, C. H. Bischof, I. S. Duff, A. S. Edelman, G. A. Geist, M. T. Heath, M. A. Heroux, C. Koelbel, R. S. Schreiber, R. F. Sincovec, and M. F. Wheeler, editors, *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, PPSC99*, San Antonio, TX, Mar. 1999. SIAM. CD-ROM.

[2] I. Banicescu and S. F. Hummel. Balancing processor loads and exploiting data locality in N-body simulations. In *Proceedings of Supercomputing'95 (CD-ROM)*, San Diego, CA, Dec. 1995. Available from http://www.supercomp.org/sc95/proceedings/594_BHUM/SC95.HTM.

[3] Basic Linear Algebra Subroutine Technical (BLAST) Forum. Basic Linear Algebra Subroutine Technical (BLAST) Forum Standard. http://www.netlib.org/blas/blast-forum/, Aug. 2001.

[4] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, IT-15(6):658–664, Nov. 1969.

[5] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, pages 340–347, Vienna, Austria, July 1997.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, CA, July 1995. Also see http://theory.lcs.mit.edu/~cilk.

[7] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, Oct. 1994.

[8] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, Apr. 1995.

[9] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Optimal evaluation of array expressions on massively parallel machines. *ACM Trans. Prog. Lang. Syst.*, 17(1):123–156, Jan. 1995.

[10] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, pages 444–453, Rhodes, Greece, June 1999.

[11] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, Saint-Malo, France, June 1999.

[12] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 205–217, La Jolla, CA, June 1995.

[13] D. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[14] J. J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, Jan. 1990.

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

[16] E. Elmroth and F. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development*, 44(4):605–624, July 2000.

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

[18] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance with source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, Las Vegas, NV, June 1997.

[19] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of ICASSP'98*, volume 3, page 1381, Seattle, WA, 1998. IEEE.

[20] G. Gibson, J. S. Vitter, and J. Wilkes. Report of the working group on storage I/O for large-scale computing. *ACM Comput. Surv.*, Dec. 1996.

[21] M. F. Goodchild and A. W. Grandfield. Optimizing raster storage: an examination of four alternatives. In *Proceedings of Auto-Carto 6*, volume 1, pages 400–407, Ottawa, Oct. 1983.

[22] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, Sept. 1992. Available as technical reports UILU-ENG-92-2237 and CRHC-92-19.

[23] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, Nov. 1997.

[24] F. G. Gustavson. New generalized data structures for matrices lead to a variety of high-performance algorithms. In B. Engquist, L. Johnsson, M. Hammill, and F. Short, editors, *Simulation and Visualization on the Grid*, volume 13 of *Lecture Notes in Computational Science and Engineering*, pages 46–61. Springer, 2000.

[25] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadephia, 1996. ISBN 0-89871-355-2 (pbk.).

[26] D. Hilbert. Über stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[27] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, C-38(12):1612–1630, Dec. 1989.

[28] Y. C. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997.

[29] S. F. Hummel, I. Banicescu, C.-T. Wang, and J. Wein. Load balancing and data locality via fractiling: An experimental study. In *Language, Compilers and Run-Time Systems for Scalable Computers*. Kluwer Academic Publishers, 1995.

[30] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of Supercomputing '96*, 1996.

[31] IBM. *Engineering and Scientific Subroutine Library Version 2 Release 2, Guide and Reference, volumes 1–3*, 1994. Order No. SC23-0526-01.

[32] H. V. Jagadish. Linear clustering of objects with multiple attributes. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 332–342, Atlantic City, NJ, May 1990. ACM, ACM Press. Published as SIGMOD RECORD 19(2), June 1990.

[33] K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Trans. Prog. Lang. Syst.*, 1998. To appear.

[34] K. Knobe, J. D. Lukas, and G. L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, Feb. 1990.

[35] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. The MIT Press, Cambridge, MA, 1994.

[36] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 1991.

[37] R. Laurini. Graphical data bases built on Peano space-filling curves. In C. E. Vandoni, editor, *Proceedings of the EUROGRAPHICS'85 Conference*, pages 327–338, Amsterdam, 1985. North-Holland.

[38] C. E. Leiserson. Personal communication, Aug. 1998.

[39] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. *J. Comput. Syst. Sci.*, 54(2):332–344, 1997.

[40] M. E. Mace. *Memory Storage Patterns in Parallel Processing*. Kluwer international series in engineering and computer science. Kluwer Academic Press, Norwell, MA, 1987.

[41] M. Mano. *Digital Design*. Prentice-Hall, 1984.

[42] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properting of Hilbert space-filling curve. Technical Report CS-TR-3611, Computer Science Department, University of Maryland, College Park, MD, 1996.

[43] G. Peano. Sur une courbe qui remplit toute une aire plaine. *Mathematische Annalen*, 36:157–160, 1890.

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

[45] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994. ISBN 0-387-94265-3.

[46] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 829–838, San Francisco, CA, Jan. 2000.

[47] J. P. Singh, T. Joe, J. L. Hennessy, and A. Gupta. An empirical comparison of the Kendall Square Research KSR-1 and the Stanford DASH multiprocessors. In *Proceedings of Supercomputing'93*, pages 214–225, Portland, OR, Nov. 1993.

[48] L. Stals and U. Rüde. Techniques for improving the data locality of iterative methods. Technical Report MRR97-038, Institut für Mathematik, Universität Augsburg, Augsburg, Germany, Oct. 1997.

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

[50] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of SC98 (CD-ROM)*, Orlando, FL, Nov. 1998. Available from http://www.supercomp.org/sc98.

[51] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal of Matrix Analysis and Applications*, 18(4):1065–1081, Oct. 1997.

[52] M. S. Warren and J. K. Salmon. A parallel hashed Oct-Tree N-body algorithm. In *Proceedings of Supercomputing'93*, pages 12–21, Portland, OR, Nov. 1993.

[53] J. Wasniewski, B. S. Anderson, and F. Gustavson. Recursive formulation of Colesky algorithm in Fortran 90. In B. Kågström, J. Dongarra, E. Elmroth, and J. Wasniewski, editors, *Proceedings of the 4th International Workshop, Applied Parallel Computing, Large Scale Scientific and Industrial Problems, PARA'98*, volume 1541 of *Lecture Notes in Computer Science*, pages 574–578, Umeå, Sweden, June 1998. Springer.

[54] R. C. Whaley. Automatically tuned linear algebra software. In *Prceedings of SC'98*, Orlando, FL, 1998.

[55] D. S. Wise. Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 774–783. Springer, Heidelberg, 2000.

[56] D. S. Wise, J. D. Frens, Y. Gu, and G. A. Alexander. Language support for Morton-order matrices. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 24–33, Snowbird, UT, June 2001.

[57] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[58] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing'89*, pages 655–664, Reno, NV, Nov. 1989.

A $\mathbb{S}$ function computation for the Hilbert layout

```
static const int out[4][4] = {{0,1,3,2},
                              {2,1,3,0},
                              {0,3,1,2},
                              {2,3,1,0}};
static const int next[4][4] = {{2,0,1,0},
                               {1,1,0,3},
                               {0,3,2,2},
                               {3,2,3,1}};

/***
   i : row index of tile
   j : column index of tile
   d : number of significant bits in i and j

   return value : the Hilbert sequence number
***/

unsigned int SCnumMO(unsigned int i, unsigned int j, int d)

{
  unsigned int in, m = 0;
  int s = 0;
  int emask = 0x01<<(--d);
  for (;d >= 0; d--) {
    in = (((i&emask)<<1)|(j&emask))>>d;
    m = 4*m + out[s][in];
    s = next[s][in];
    emask >>= 1;
  }
  return m;
}
```

B Incremental tile number computation for recursive layouts

The following C macros allow incremental computation of tile numbers for various recursive layouts. For each layout, there are five macros: `NW(a,n)`, `NE(a,n)`, `SW(a,n)`, `SE(a,n)`, and `fuzz(n)`. Given a (sub)quadrant of the matrix containing `2n`×`2n` tiles, where `a` encodes the number and orientation of the lowest-numbered tile, the first four macros gives an encoding of the number and orientation of the lowest-numbered tile in the corresponding quadrants. The last macro strips off the high order bits used to encode orientation and returns a valid tile number.

It is assumed that `n` is a power of two, as in equation (4).

Thus, if an `m`×`m` array of doubles is divided into `t`×`t` tiles and has starting address `A`, then we have:

```
int naxis = m/t;  // the number of tiles along each axis
int num = num*num;
int nn = num/4;
int sw = SW(0,nn); // the starting tile of SW quadrant
int nesw = NE(sw,nn/4); // starting tile of NE quad of SW quad
double *nesw_addr = A + fuzz(nesw)*t*t;
```

B.1 Single-orientation recursive layouts

```
/*
 * Tile numbers for recursive orders that have a single
 * orientation.  This covers Z-Morton, U-Morton, and X-Morton.
 */

#ifndef ZMORTON
#define ZMORTON 0
#endif

#ifndef UMORTON
#define UMORTON 0
#endif

#ifndef XMORTON
#define XMORTON 0
#endif

#if ZMORTON
#define QUAD1
#define _NW                0
#define _NE                1
#define _SW                2
#define _SE                3
#endif
```

```
#if UMORTON
#define QUAD1
#define _NW                0
#define _NE                3
#define _SW                1
#define _SE                2
#endif

#if XMORTON
#define QUAD1
#define _NW                0
#define _NE                3
#define _SW                2
#define _SE                1
#endif

#ifdef QUAD1
#define NW(a,n)            ((a)+_NW*(n))
#define NE(a,n)            ((a)+_NE*(n))
#define SW(a,n)            ((a)+_SW*(n))
#define SE(a,n)            ((a)+_SE*(n))
#define fuzz(n)            (n)
#endif
```

B.2 Two-orientation recursive layouts

```
/*
 * Tile numbers for recursive orders that have two orientations.
 * This covers Gray-Morton.
 *
 * Encode the (NW,NE,SE,SW) orientation with a positive sign, and
 * the (SE,SW,NW,NE) orientation with a negative sign in the "a"
 * argument in the macros below.
 */

#ifndef GMORTON
#define GMORTON 0
#endif

#if GMORTON
#define QUAD2
#define NW(a,n)     ((a)>=0? ((a)      ): (-(a)+2*(n)))
#define NE(a,n)     ((a)>=0?-((a)+1*(n)):-(-(a)+3*(n)))
#define SW(a,n)     ((a)>=0? ((a)+3*(n)): (-(a)+1*(n)))
```

```
#define SE(a,n)     ((a)>=0?-((a)+2*(n)):-(-(a)      ))
#define fuzz(n)     (abs(n))
#endif
```

B.3 Four-orientation recursive layouts

```
/*
 * Tile numbers for recursive orders that have four orientations.
 * This covers Hilbert.
 *
 * Encode the four orientations in the top two bits of the "a"
 * argument to the macros below.
 */

#ifndef HILBERT
#define HILBERT 0
#endif

#if HILBERT
#define QUAD4
#define _D 0
#define _R 1
#define _U 2
#define _L 3
#define _NW 0
#define _NE 1
#define _SW 2
#define _SE 3
static const unsigned int dir[4][4] = {{_R,_D,_U,_L},
                                       {_D,_U,_R,_L},
                                       {_L,_R,_U,_D},
                                       {_D,_R,_L,_U}};
static const unsigned int ord[4][4] = {{0,0,2,2},
                                       {1,3,3,1},
                                       {3,1,1,3},
                                       {2,2,0,0}};
#define BITS_IN_BYTE             8
#define BITS_IN_INT              (sizeof(int)*BITS_IN_BYTE)
#define SHIFT_AMT                (BITS_IN_INT-2)
#define dirmask                  (3U<<SHIFT_AMT)
#define quad(adir,aint,q,n) \
    ((dir[(q)][(adir)]<<SHIFT_AMT)| \
     (((aint)+ord[(q)][(adir)]*(n))&~dirmask))
#define xdir(a)         (((a)&dirmask)>>SHIFT_AMT)
#define xint(a)         ((a)&~dirmask)
```

```
#define NW(a,n)         quad(xdir(a),xint(a),_NW,n)
#define NE(a,n)         quad(xdir(a),xint(a),_NE,n)
#define SW(a,n)         quad(xdir(a),xint(a),_SW,n)
#define SE(a,n)         quad(xdir(a),xint(a),_SE,n)
#define fuzz(n)         (xint(n))
#endif
```

C Global lookup tables for the Hilbert layout

The following code initializes the global lookup tables needed for pre- and post-additions for the Hilbert layout, as explained in Section 4.6. It uses the `SCnumMO` function defined in Appendix A and several macros defined in Appendix B.3.

```
#define H_MAX_LEVELS 6
int *h_lut[4][H_MAX_LEVELS];

void h_lut_init()

{
  int i, j, k, len, n;

  h_lut[_U][0] = h_lut[_D][0] = h_lut[_L][0] = h_lut[_R][0] = 0L;
  for (i = 1, len = 4; i < H_MAX_LEVELS; i++, len *= 4) {
    h_lut[_U][i] = (int *)malloc(len*sizeof(int));
    h_lut[_D][i] = (int *)malloc(len*sizeof(int));
    h_lut[_L][i] = (int *)malloc(len*sizeof(int));
    h_lut[_R][i] = (int *)malloc(len*sizeof(int));
  }
  for (i = 1, len = 2; i < H_MAX_LEVELS; i++, len *= 2) {
    for (j = 0, n = 0; j < len; j++) {
      for (k = 0; k < len; k++, n++) {
         h_lut[_U][i][n] = SCnumMO(len-j-1, len-k-1, i);
         h_lut[_D][i][n] = SCnumMO(j,       k,       i);
         h_lut[_L][i][n] = SCnumMO(len-k-1, len-j-1, i);
         h_lut[_R][i][n] = SCnumMO(k,       j,       i);
      }
    }
  }
}
```