Nonlinear Array Layouts for Hierarchical Memory Systems[*]

Siddhartha Chatterjee[†] Vibhor V. Jain[†] Alvin R. Lebeck[‡] Shyam Mundhra[†] Mithuna Thottethodi[‡]

Abstract

Programming languages that provide multidimensional arrays and a flat linear model of memory must implement a mapping between these two domains to order array elements in memory. This layout function is fixed at language definition time and constitutes an invisible, non-programmable array attribute. In reality, modern memory systems are architecturally hierarchical rather than flat, with substantial differences in performance among different levels of the hierarchy. This mismatch between the model and the true architecture of memory systems can result in low locality of reference and poor performance. Some of this loss in performance can be recovered by re-ordering computations using transformations such as loop tiling. We explore nonlinear array layout functions as an additional means of improving locality of reference. For a benchmark suite composed of dense matrix kernels, we show by timing and simulation that two specific layouts (4D and Morton) have low implementation costs (2–5% of total running time) and high performance benefits (reducing execution time by factors of 1.1–2.5); that they have smooth performance curves, both across a wide range of problem sizes and over representative cache architectures; and that recursion-based control structures may be needed to fully exploit their potential.

1 Introduction

The performance of programs on modern computer systems depends as much on the capabilities of the memory system as on the instruction issue rate of the processor. For scientific computations that repeatedly access large data sets, good locality of reference is essential at the algorithm level for high performance. Such locality can either be *temporal*, in which a data item is reused repeatedly, or *spatial*, in which a group of data items "adjacent" in space are used in temporal proximity. Architectural components such as registers and caches support such algorithm-level locality: registers naturally support temporal locality, cache replacement policies favor temporal locality, and multi-word cache lines support spatial locality. The best performance usually results when the algorithmic patterns of locality of reference matches the patterns that the cache organization supports well. Unfortunately, such a match cannot always be taken for granted: features of the cache architecture such as limited capacity and limited associativity can often interfere with high performance.

The thesis of this paper is that joint restructuring of the control flow of a program and of the data structures it uses results in the highest level of performance. We focus on dense matrix codes for which *loop tiling* (also called loop blocking) [63] is an appropriate means of high-level control flow restructuring to improve locality. Loop tiling is a program transformation that tessellates the iteration space of a loop nest with uniform tiles of a given size and shape and schedules the tiles for execution in an order consistent with the original data dependences. Computations associated with a given tile are executed as a group. This computation re-ordering is typically expressed in the optimized program as a deeper loop nest. Loop tiling is usually accompanied by loop unrolling and software pipelining to enable better use of registers and the processor pipeline. The theory of these techniques is well-developed [43], and implementations of the techniques are available as optimization options in several high-performance compilers. Tiling techniques are also often applied at the source level in numerical libraries, *e.g.*, in the level-3 BLAS [16] and in LAPACK [1]. In this paper, we assume that loop tiling has been performed either by the programmer or by the compiler, and examine the additional performance gains achievable using nonlinear array layout functions. Viewed in another formalism where we model the program as a stream of references to array elements, tiling alters the reference stream in the array index space by restructuring the iteration space, while the array layout function changes the mapping from the array index space to the virtual address space. In this paper, we evaluate the marginal benefit of the latter transformation given that the former transformation has been performed.

While programmers and compilers are familiar and comfortable with transforming the control flow of programs, they are less likely to restructure multidimensional arrays to be "cache-conscious" or "memory-friendly". Such restructuring techniques have been studied for pointer-based data structures, such as heaps [35, 37, 38] and trees [13]; for profile-driven object placement [8]; for matrices with special structure (*e.g.*, banded matrices in LAPACK [1], or sparse matrices [20]); and in parallel computing [5, 28, 29, 45, 51, 61]. But when working with general dense matrices in a uniprocessor environment, most programmers are reluctant to alter the default row-major or column-major linearization of multidimensional arrays that high-level languages provide, even when such ordering degrades cache performance. We suspect several reasons for this *status quo*: the special support accorded to arrays in modern programming languages; the ability to perform fast address computation incrementally with the default layout; and the well-understood (and often abused) idioms of "efficient" usage patterns. We believe that

[*]This work supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637 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. 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.

[†]Department of Computer Science, The University of North Carolina, Chapel Hill, NC 27599-3175. Email: {sc,vibhor,mundhra}@cs.unc.edu

[‡]Department of Computer Science, Duke University, Durham, NC 27708-0129. Email: {alvy,mithuna}@cs.duke.edu

the non-standard data layouts used by library writers and in parallel computing are applicable in more general situations, but are not readily exploitable given existing programming language design.

In this paper, we consider whether user-definable layout functions can and should be offered in a general way for dense matrices. For two families of nonlinear layout functions, we quantify both the additional costs associated with a non-standard array layout (such as the cost of format conversion and the overhead of address computation) and the improved performance that such a layout function provides. We make the following contributions in this paper.

- We demonstrate by measurement of execution time that the costs associated with these layout functions are minimal (§3.4), that they are more than offset by the performance gains resulting from improved locality (§3.1), and that the absolute performance levels are acceptably high (§3.2).
- We show by measurement that such layouts reduce the variability of performance as the input matrix size is smoothly varied (§3.3).
- We show by measurement that the performance of such layouts is largely insensitive to the choice of tile size (§3.5).
- We show by simulation that such layouts improve performance for representative multi-level memory hierarchy architectures (§3.7).
- We show by measurement that alternative control structures based on recursion rather than loop tiling often better exploit the potential of these layout functions (§3.6).

The remainder of this paper is organized as follows. §2 discusses the problems with the default mappings of multidimensional arrays, and introduces two nonlinear mappings that we expect to demonstrate better cache performance. §3 offers measurement and simulation results that support the claim that these layouts improve both locality of reference and the overall performance. §4 compares our approach with previous related work. §5 presents conclusions and future work.

2 Nonlinear array layouts

Programming languages that support multidimensional arrays must also provide a function (the *layout* function L) to map the array index space into the linear memory address space. For ease of exposition, we assume a two-dimensional array with m rows and n columns, which we index using a zero-based scheme. The results we discuss generalize to higher-dimensional arrays and other indexing schemes. We define L such that $L(i,j)$ is the memory location of the array element in row i and column j relative to the starting memory location of the array, in units of array elements. 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: $L(i,j;m,n)$.

We require that the layout function be one-to-one (so that different array elements map to different memory locations), that its image be dense (so that there are no holes in the memory footprint of the array), and that it be easily computable. If we restrict our attention to layout functions that are linear and monotonically increasing in the arguments i and j (such functions certainly being easy to compute), it is easy to prove that there are only two such layout functions [14]: the *row-major* layout L_{RM} as used in Pascal, given by $L_{RM}(i,j;m,n) = n \cdot i + j$; and the *column-major* layout L_{CM} as used in Fortran, given by $L_{CM}(i,j;m,n) = m \cdot j + i$. We refer to these two layouts as *canonical* layouts. Figure 1(a)–(b) show these two layouts. Simple algebraic manipulation of the defining formulas, as shown below in equations (1)–(2), reveals that these layouts have the further desirable property of allowing incremental computation of memory locations of elements that are adjacent in array index space.

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

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

Canonical layouts do not always interact well with cache memories, because the layout function favors one axis of the index space over the other: neighbors in the unfavored direction become distant in memory. This is an alternative interpretation of equations (1)–(2). This *dilation* effect can reduce program performance in several ways. First, it may reduce or even nullify the effectiveness of multi-word cache lines, as in the case of a Fortran loop that accesses successive elements of an array row. Such low spatial locality can usually be corrected by appropriate loop transformations (such as interchange, reversal, or skewing) when such transformations are legal [4]. Second, for large matrix sizes, it may even reduce the effectiveness of translation lookaside buffers (TLBs), because the dilation effect extends to virtual memory pages [3, 56]. Finally, it may cause cache misses due to self-interference even when a tiled loop repeatedly accesses a small tile in the array index space, because the canonical layout depends on the matrix size rather than the tile size. Such interference misses are a complicated and nonsmooth function of the array size, the tile size, and the cache parameters [19]. These considerations lead us to investigate other array layout functions.

We begin with the result of Lam *et al.* [36] that a $t_R \times t_C$ array that is contiguous in memory and fits in cache causes no self-interference misses. If we fix values for t_R and t_C, we can now conceptually view our original $m \times n$ array as a $\lceil \frac{m}{t_R} \rceil \times \lceil \frac{n}{t_C} \rceil$ 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

$$\begin{aligned} (t_i, t_j, f_i, f_j) &= (\mathbb{T}(i;t_R), \mathbb{T}(j;t_C), \mathbb{F}(i;t_R), \mathbb{F}(j;t_C)) \\ \mathbb{T}(i;t) &= i \operatorname{div} t \\ \mathbb{F}(i;t) &= i \bmod t \end{aligned}$$

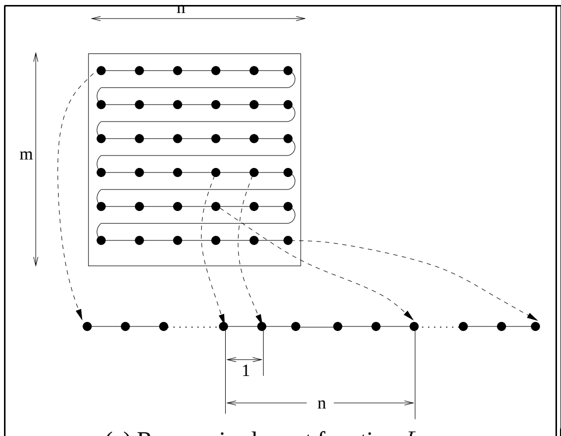
(The transformations $\mathbb{T}$ and $\mathbb{F}$ are *nonlinear*, explaining the use of this qualifier in our description of these array layout functions.) We then create two subspaces: the space T of tile co-ordinates (t_i, t_j), and the space F of tile offsets (f_i, f_j). We apply a canonical layout function L_F in the F-space (to keep each tile contiguous in memory) and a layout function L_T of our choice in the T-space (to obtain the starting memory location of the tile), and define our nonlinear layout function L as their sum:

$$\begin{aligned} L(i,j;m,n,t_R,t_C) &= L(t_i,t_j,f_i,f_j;m,n,t_R,t_C) \\ &= L_T(t_i,t_j;m,n,t_R,t_C) \\ &\quad + L_F(f_i,f_j;t_R,t_C). \end{aligned} \tag{3}$$

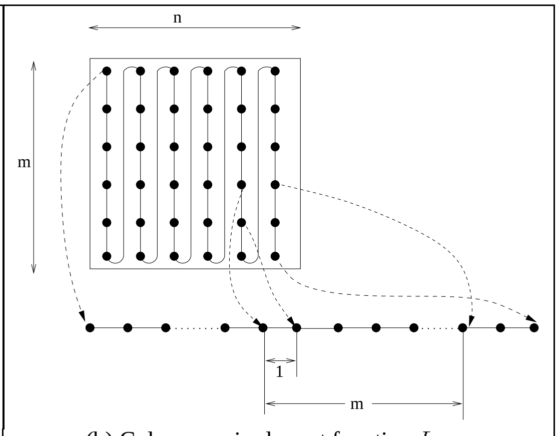
Equation (3) defines a two-parameter family of layout functions, parameterized by L_T and L_F. In this paper, we fix L_F to be the column-major layout function, and investigate two specific choices for L_T.

2.1 The 4D layout, L_{4D}

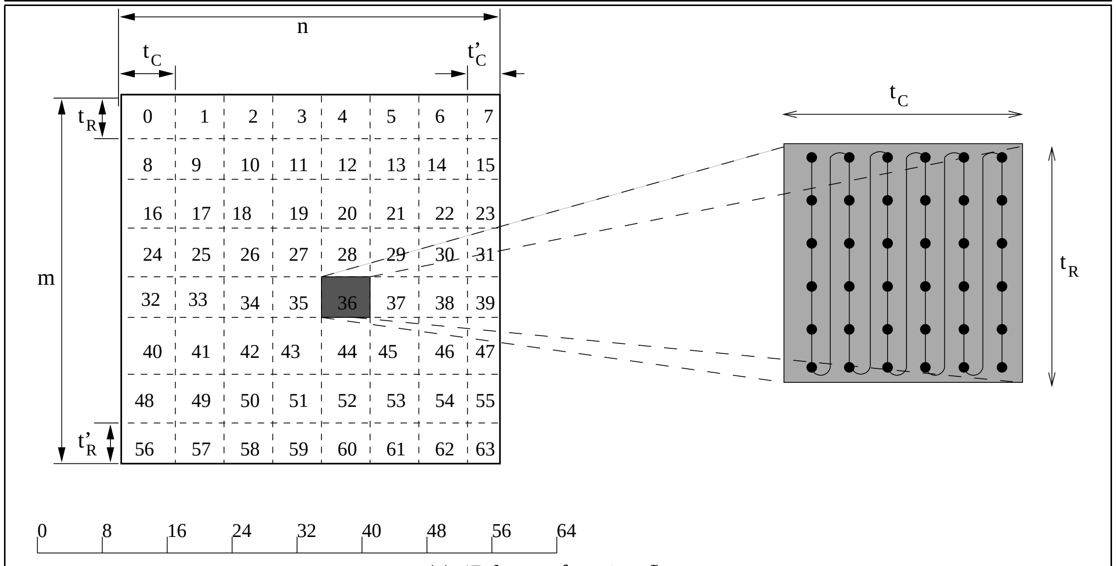
If we choose both L_T and L_F to be canonical layout functions, we call the resulting layout function the *4D* layout function L_{4D}. In



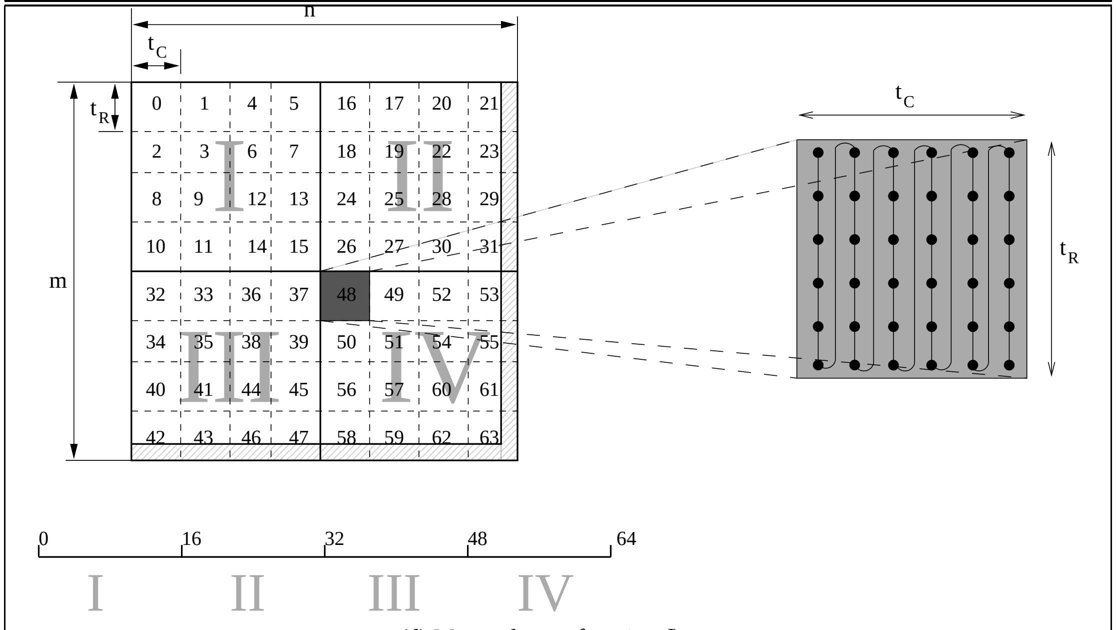
(a) Row-major layout function L_{RM}



(b) Column-major layout function L_{CM}



(c) 4D layout function L_{4D}



(d) Morton layout function L_{MO}

Figure 1: Graphical description of layout functions. Arrays are $m \times n$; tiles are $t_R \times t_C$.

this paper, we choose L_T to be L_{RM} and L_F to be L_{CM}. The following equations summarize the mathematical details of this mapping, accounting for the cases where a tile size does not exactly divide the corresponding matrix dimension. Figure 1(c) illustrates this layout.

$$\begin{array}{rcl}
(k_R, k_C) & = & (\lceil m/t_R \rceil, \lceil n/t_C \rceil) \\
t'_R & = & m - t_R \cdot (k_R - 1) \\
t'_C & = & n - t_C \cdot (k_C - 1) \\
\mathbb{U}(i,k;t_1,t_2) & = & \left\{ \begin{array}{ll} t_1, & i \neq k-1 \\ t_2, & i = k-1 \end{array} \right. \\
\mathbb{S}(i,j;m,n,t_R,t_C) & = & i \cdot n \cdot t_R + \mathbb{U}(i,k_R;t_R,t'_R) \cdot t_C \cdot j \\
L_{4D}(i,j;m,n,t_R,t_C) & = & \mathbb{S}(t_i,t_j;m,n,t_R,t_C) \\
 & & +L_{CM}(f_i,f_j;\mathbb{U}(t_i,k_R;t_R,t'_R),\mathbb{U}(t_j,k_C;t_C,t'_C))
\end{array}$$

2.2 The Morton layout, L_{MO}

Our second nonlinear layout function has been variously described as being based either on quadtrees [18] or on space-filling curves [27, 44, 49]. This layout is known in parallel computing as the *Morton ordering* and has been used for load balancing purposes [5, 28, 29, 45, 51, 61]. It has also been applied for bandwidth reduction in information theory [6], for graphics applications [24, 39], and for database applications [30]. Figure 1(d) illustrates this layout.

Morton ordering has the following operational interpretation. Divide the original matrix into four quadrants, and lay out these submatrices in memory in the order NW, NE, SW, SE. A $k_R \times k_C$ submatrix with $k_R > t_R$ and $k_C > t_C$ is laid out recursively using the Morton ordering; a $t_R \times t_C$ tile is laid out using the L_F-ordering.

To formally define this layout function, we require t_R and t_C to simultaneously satisfy

$$\frac{m}{t_R} = \frac{n}{t_C} = 2^d \quad (4)$$

for some positive integer d. We define

$$L_T(t_i, t_j; t_R, t_C) = t_R \cdot t_C \cdot \mathbb{M}(t_i, t_j)$$

where $\mathbb{M}(i, j)$ is the integer whose binary representation is the bitwise interleaving of the binary representations of i and j. Then,

$$L_{MO}(i,j;m,n,t_R,t_C) = t_R \cdot t_C \cdot \mathbb{M}(t_i,t_j) + L_{CM}(f_i,f_j;t_R,t_C). \quad (5)$$

There is, in fact, a family of Morton and Morton-like layout functions, of varying degrees of complexity. The variant describes above is more accurately called the Z-Morton layout. Chatterjee *et al.* [12] discuss the details of this family in greater detail.

2.3 Practical issues

The mathematical description of L_{4D} and L_{MO} above glosses over several practical details that are critical for efficient implementation of these ideas. We discuss these issues in this section.

Holes The L_{4D} mapping imposes no restrictions on tile sizes or matrix sizes. However, the definition of L_{MO} works only when t_R and t_C are 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 their solution insensitive to the amount of padding, but requires maintaining the internal nodes of the quad-tree. 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 tile sizes t_R and t_C from an architecture-dependent range, explicitly inserting the zero padding, and blindly performing all the arithmetic on the zeros. We choose the range of acceptable tile sizes so that the tiles are neither too small (which increases the overhead of recursive control) nor overflow the cache (which results in capacity misses). Since tiles are contiguous, there are no self-interference misses. This makes the performance of the leaf-level computations almost insensitive to the tile size [59].

  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 $1/T_{\min}$.

Address computation costs The naïve computation of the 4-tuple (t_i, t_j, f_i, f_j) from (i, j) involves integer division and remainder operations and is therefore expensive. These layout functions are inappropriate for programs that access array elements randomly. For structured accesses, however, we can often eliminate the explicit computation of the 4-tuple and provide efficient incremental addressing. The decoupling of the layout function is critical to such incremental address computation: once we have located a tile and are accessing elements within it, we need not recompute the starting offset of the tile and can use the incremental addressing techniques developed for the canonical layouts.

Even more than for the 4D layout, the Morton layout function is expensive to compute naïvely. It requires bit manipulation operations, or can alternatively be computed in $O(d)$ integer operations without any bit manipulations [42]. Yet another option is to pre-compute the required $\mathbb{M}(i, j)$ values in a lookup table. In our implementation, we use a combination of both techniques. We process the inputs four bits at a time, generating partial contributions to $\mathbb{M}$ using a pre-computed lookup table.

A naïve but correct implementation strategy is to follow equation (3) and to replace every reference to array element $A(i, j)$ in the code with a call to the address computation routine for the appropriate nonlinear layout. However, this requires integer division and remainder operations to compute (t_i, t_j, f_i, f_j) from (i, j), which imposes an unreasonably large overhead. To gain efficiency, we need to exploit the decoupling of the layout function L shown in equation (3): once we have located a tile and are accessing elements within it, we do not recompute the starting offset of the tile and instead use the incremental addressing techniques supported by the canonical layout L_{CM}.

Format conversion costs If a nonlinear layout function is used internally within a library, an honest accounting of costs must include the cost of converting to and from a canonical representation at the library interface. We demonstrate in §3.4 that such conversions can indeed be performed efficiently.

Tile size selection The tile size chosen for loop tiling has a significant impact on performance, and previous work (*e.g.*, [2, 15, 17, 19, 62]) has addressed various questions related to tile size selection. The nonlinear layout schemes are based on the result that a small contiguous tile exhibits no self-interference misses. This makes their performance largely insensitive to the tile size, as long as we choose the tile size in a range so that the tile is neither too small nor overflows the cache. We demonstrate this fact in §3.5.

Control flow We need to restructure control flow in order to reap the benefits of Morton ordering. The natural control flow that makes good use of this ordering is based on recursion rather than on tiled loop nests. We demonstrate this point in §3.6.

Multi-level memory hierarchies If the control flow of the program can exploit Morton ordering, we expect the layout function to be particularly beneficial for multi-level memory hierarchies. This expectation arises from the recursive nature of the ordering, which keeps the elements of a quadrant together. This means that an element or tile of the matrix that is absent from a level of the memory hierarchy is likely to be found in the next level. Our simulations in §3.7 confirm this point.

2.4 Summary

The layout of a multi-dimensional array in memory is a key determinant of the performance of code that uses that array. Current high-level programming languages do not make this array attribute easily accessible to programmers or compilers. Even the use of canonical orderings in implementations of arrays is artificial. A d-dimensional array has $d!$ canonical orderings [14], but a given programming language uses a single ordering that was fixed at language design time. Within a single module or compilation unit, the effect of an arbitrary canonical layout can be simulated by permuting the order of indices. This process is tedious and error-prone, leads to obfuscated code, and does not extend across modules. A more reliable and scalable solution is to discard the flat linear memory model and to make the layout of multi-dimensional arrays an explicitly programmable attribute.

Making the layout attribute of arrays explicitly programmable does raise the obvious question of how one chooses the desired layout. This problem is related to the problem of choosing good data decompositions in languages such as High Performance Fortran [34]. We expect that the techniques for layout optimization developed in the vector and parallel computing context (*e.g.*, [11, 25, 31, 32, 41]) can be adapted to the hierarchical memory situation.

3 Experimental results

Table 2 describes benchmark suite that we use to evaluate nonlinear array layout functions. Three of these codes call the BLAS 3 [16] routine `dgemm` to perform matrix multiplication on tile-sized chunks of data. We implemented the various versions of these codes in ANSI C, then compiled and executed them on the three platforms described in Table 3. The versions of these codes using canonical layouts are blocked where appropriate (*i.e.*, not in the recursive codes). The CHOL code is the "shackled" version due to Kodukula *et al.* [33]. The BMXM, RECMXM, and STRASSEN codes were run on matrices initialized with random data. The CHOL code was run on a matrix initialized with random data while keeping it symmetric and diagonally dominant. The STDHAAR and NONHAAR codes were run on two real images cropped to the appropriate size.

Benchmark	**Ultra 10**		**Ultra 60**		**Miata**	
	L_{4D}	L_{MO}	L_{4D}	L_{MO}	L_{4D}	L_{MO}
BMXM	0.93	1.06	0.95	1.05	0.97	0.95
RECMXM	—	0.94	—	0.94	—	0.95
STRASSEN	—	0.87	—	0.79	—	0.91
CHOL	0.78	—	0.85	—	0.67	—
STDHAAR	0.68	0.67	0.64	0.64	0.42	0.43
NONHAAR	0.62	0.61	0.58	0.58	0.40	0.40

Table 1: Summary of the performance improvement of nonlinear data layouts. Each table entry is the arithmetic mean over a range of problem sizes of the ratio of the execution time of the nonlinear layout code (including conversion from a canonical layout) to the execution time of the canonical layout code. Smaller numbers are better.

We were the only user on the systems for the duration of the runs. We use a self-calibrating timing loop to ensure that the computation ran for at least 2 seconds, to eliminate the effects of low timer resolution; we measure elapsed time for the computation (excluding such things as allocation and initialization) using the system call `getrusage()`; and we execute multiple trials to further reduce measurement error. We evaluate cache performance using ATOM [52] and TLB performance using `fast-cache` [40].

The following sections highlight our major results. The data presented below is a highly condensed version of our complete data, due to length limitations.

3.1 Performance improvement over canonical layouts

Table 1 shows normalized execution time (time with a nonlinear layout/ time with a canonical layout) for each benchmark, averaged over matrix sizes in the range 100–1000 (specific values depend on the benchmark). We see that nonlinear layouts can dramatically reduce execution time, with some benchmarks finishing in half the time of the code with a canonical layout.

3.2 High performance

In order to allay concerns about the absolute level of performance of our codes, Figure 2 presents execution times for RECMXM and STRASSEN normalized to the execution time of the vendor-supplied native BLAS routine `dgemm`, and Figure 3 presents MFLOP/s for CHOL. We observe that, for the matrix multiplication codes, both L_{4D} and L_{MO} are competitive with the native library, and even outperform it for large problem sizes. The sawtooth pattern in the curves is due to a mismatch between the tile size and the amount of loop unrolling performed by the compiler.

3.3 Robust performance

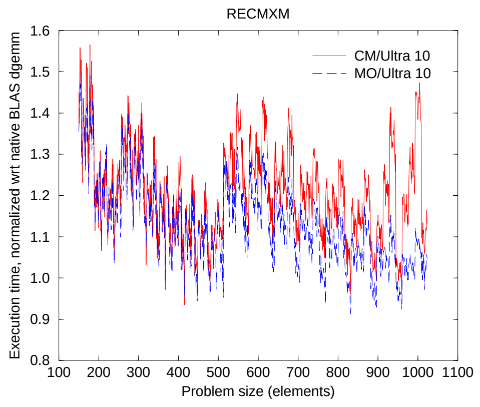
Figure 4 shows the execution time of CHOL for various tile sizes (a) and matrix sizes (b) using both L_{CM} and L_{4D} on the DEC Miata. These graphs clearly show that L_{4D} is superior to L_{CM}. First, Figure 4(a) shows that L_{4D} produces lower execution time for all tile sizes examined. Similarly, Figure 4(b) shows that the performance of L_{4D} is extremely robust as a function of matrix size. In contrast, self-interference misses degrade performance for L_{CM} when the matrix is a multiple of the cache size. The other platforms and benchmarks exhibit similar behavior.

Name	**Description**	**BLAS?**	**Layouts used**		
			L_{CM}	L_{4D}	L_{MO}
BMXM	Tiled 6-loop matrix multiplication	$\surd$	$\surd$	$\surd$	$\surd$
RECMXM	Recursive matrix multiplication [18]	$\surd$	$\surd$		$\surd$
STRASSEN	Strassen's matrix multiplication [55]	$\surd$	$\surd$		$\surd$
CHOL	Right-looking Cholesky factorization [33]		$\surd$	$\surd$	
STDHAAR	Standard wavelet compression of image (Haar basis) [54]		$\surd$	$\surd$	$\surd$
NONHAAR	Non-standard wavelet compression of image (Haar basis) [54]		$\surd$	$\surd$	$\surd$

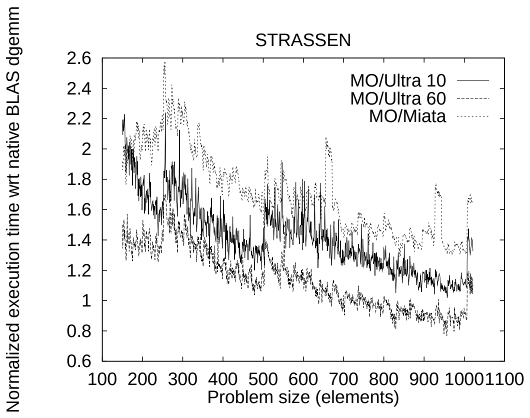
Table 2: Description of benchmark suite.

Parameter	**Ultra 10**	**Ultra 60**	**Miata**
CPU	UltraSPARC-IIi	UltraSPARC-II	Alpha 21164
Clock rate	300 MHz	300 MHz	500 MHz
L1 cache	16KB/32B/1, on-chip	16KB/32B/1, on-chip	8KB/32B/1, on-chip
L2 cache	512KB/64B/1, off-chip	2MB/64B/1, off-chip	96KB/64B/3, on-chip
L3 cache	none	none	2MB/64B/1, off-chip
RAM	320MB	512MB	512MB
Data TLB entries	64	64	64
VM page size	8KB	8KB	8KB
cc version	Workshop Compilers 4.2	Workshop Compilers 4.2	DEC C V5.6-075
cc switches	-fast	-fast	-fast
Native BLAS	libsunperf.so	libsunperf.so	libdxml.a
Operating system	SunOS 5.6	SunOS 5.6	OSF1 V4.0 878

Table 3: Machine configurations. The entries for the cache configurations are the cache parameters C/B/A.

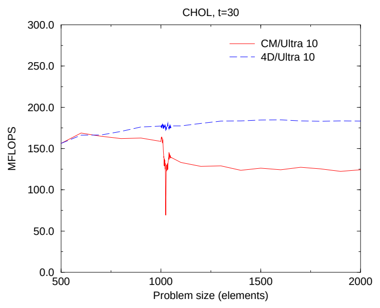


(a)

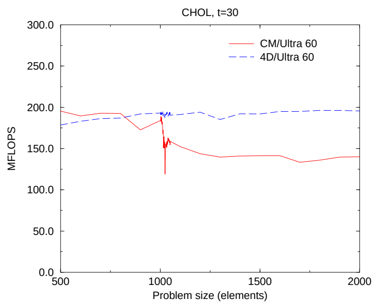


(b)

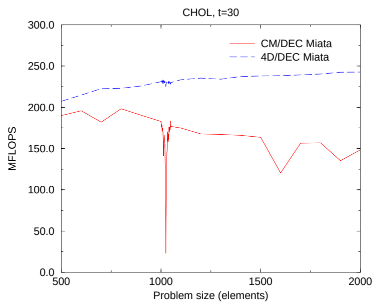
Figure 2: Absolute performance level of our codes. (a) Execution time of RECMXM on **Ultra 10** normalized with respect to execution time of native BLAS routine `dgemm`. (b) Execution time of STRASSEN on **Ultra 10**, **Ultra 60**, and **Miata**, normalized with respect to execution time of native BLAS routine `dgemm`.



(a)



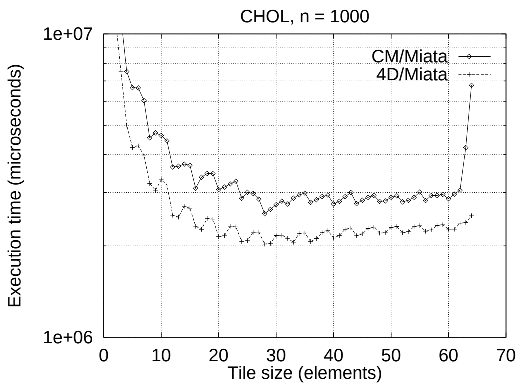
(b)



(c)

Figure 3: Performance of CHOL, with tile size of 30. Data points are at multiples of 100 elements, except for the dense range [1000:1050]. This dense range is shown in greater detail in Figure 4(b). The MFLOP/s numbers are based on a flop count of $n^3/2$. (a) MFLOP/s on **Ultra 10**. (b) MFLOP/s on **Ultra 60**. (c) MFLOP/s on **Miata**.

(a)



(b)

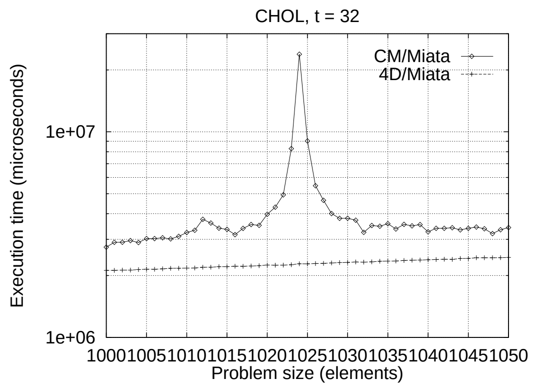
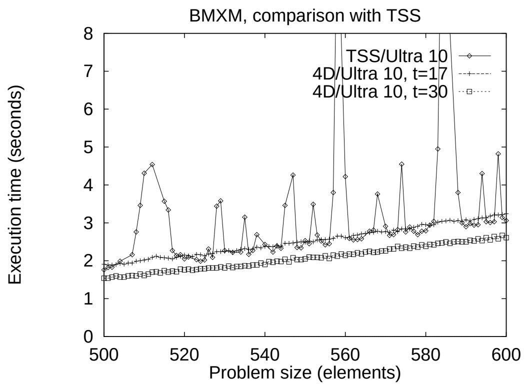


Figure 4: Performance variation of CHOL on **Miata** for different data layout functions. (a) Problem size 1000×1000, varying tile size. (b) Tile size 32, varying problem size.

(a)



(b)

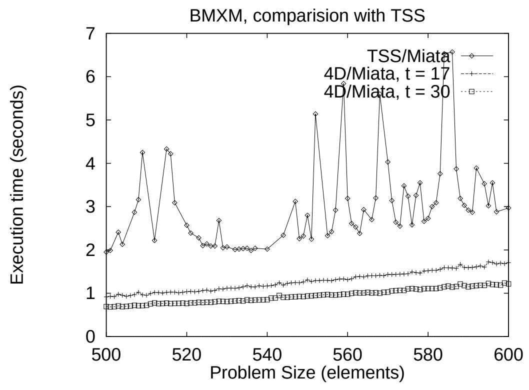


Figure 5: Running times for BMXM for the TSS algorithm of Coleman and McKinley [15] and the L_{4D} layout with fixed tile sizes of 17 and 30. (a) On **Ultra 10**. (b) On **Miata**.

3.4 Format conversion cost

All measurements provided thus far include the cost of converting from canonical layout to the appropriate nonlinear layout. We measured the conversion cost to be 2–5% of the total execution time. We conclude that the benefits of nonlinear layouts outweigh the conversion cost.

3.5 Tile size selection issues

Figure 5 compares our work to the TSS algorithm of Coleman and McKinley [15]. We compare execution times for a range of problem sizes of the L_{4D} layout with tile sizes set to 17 and 30 with the execution time of the TSS algorithm. For the comparison, we obtained the authors' (corrected) Fortran code and supplied it with L1 cache parameters for the machine. We do not time their overhead of computing the tile sizes. The shapes of the graphs reveal that nonlinear layouts significantly reduce the variability in performance. There is a strong correlation between the measured execution time and the number of elements contained in the tile size selected by the TSS algorithm. We conjecture that any tile size selection algorithm based on GCD computations (as is the case with the TSS algorithm) will show similar non-smooth performance characteristics.

3.6 Control flow issues

Table 1 shows that in some cases the full benefit of nonlinear layouts can be achieved only when the control structure of the program matches the data layout. For example, L_{4D} works very well for BMXM which has a nested loop control structure, while L_{MO} actually increases execution time. In contrast, the recursive control structure of RECMXM matches the recursive nature of L_{MO}, producing commensurate improvements in execution time. Gustavson [26] and and Chatterjee *et al.* [12] discuss this issue in greater detail.

3.7 Cache and TLB simulations

To gain further insight into the memory system behavior of nonlinear layouts, we simulated various cache configurations: L1 caches of 16KB, 1-way and 2-way associative, with 32-byte blocks; and L2 caches of 128KB and 512KB, 1-way associative, with 64-byte blocks. Our results add further support to our claim that nonlinear layouts reduce cache misses over canonical layouts. However, the reduction is not uniform across all portions of the memory hierarchy for all benchmarks. For a 16KB direct-mapped L1 cache, and matrix size of 512, L_{CM} incurs miss ratios of 5.8% and 33% for RECMXM and CHOL respectively, while RECMXM achieves a 4.4% miss ratio using L_{MO}, and CHOL achieves 3.2% with L_{4D}. In contrast, the nonlinear layouts increase the L1 miss ratio for both STDHAAR and NONHAAR, from 18% using the canonical layout to 27% for the nonlinear layouts. This is due to degenerate interference misses between tiles. Increasing the associativity significantly reduces this disparity. We are also investigating techniques to eliminate this effect through instruction scheduling or by using alternative nonlinear layouts.

Although the L1 miss ratio can increase in some cases, we note that the L2 miss ratio decreases dramatically for STDHAAR and NONHAAR when using nonlinear layouts. Specifically, NONHAAR drops from 14% for the canonical layout to 4.5% for L_{MO} for the 512KB L2. Similar reductions occur for the 128KB cache and STDHAAR. For the other benchmarks L2 global miss ratios are generally lower for the nonlinear layouts. Associativity and capacity generally reduce the miss ratio for all layouts.

The TLB benefits of nonlinear layouts are most evident for STDHAAR and NONHAAR. Simulations of a 64-entry fully-associative TLB reveal that nonlinear layout reduces the TLB miss ratio to 0.01% compared to 2.5% for the canonical layout. We note that TLB performance could be improved by using superpages [57] to map the entire array with a single TLB entry.

3.8 Summary

The experimental data support our claim that nonlinear layouts provide significant performance benefits for dense matrix codes.

By making the layout of a matrix a programmable attribute, we enable the global optimization of this attribute. This would further reduce the number of conversions between canonical and non-standard layouts. This is essentially the approach to data parallelism taken in High Performance Fortran [34], and techniques for automatic data mapping developed in that context (*e.g.*, [11, 25, 31, 32]) can equally well be applied to a uniprocessor environment to improve memory hierarchy performance.

Such layout functions can initially coexist with default layouts by encapsulating them within user-defined record types or classes, without having to change the definition of the language. The use of the BLAS routines in our codes establishes this point.

4 Related work

We categorize related work into two categories: previous application of nonlinear array layout in scientific libraries, and work in the compiler community related to tiling for parallelism and cache optimizations.

Scientific libraries The PHiPAC project [7] aims at producing highly tuned code for specific BLAS 3 [16] 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.

Frens and Wise [18] provide an implementation of recursive matrix multiplication. We adopted their idea of computation restructuring by recursion unfolding. They appear to carry the recursion down to the level of single array elements, which causes a dramatic loss of performance.

Gustavson [26] discusses the role of recursive control strategies in automatic variable blocking of dense linear algebra codes, and shows dramatic performance gains compared to implementations of the same routines in IBM's Engineering and Scientfic Subroutine Library (ESSL).

Stals and Rüde [53] investigate algorithmic restructuring techniques for improving the cache behavior of iterative methods. They do not investigate nonlinear data reorganization.

The goal of out-of-core algorithms [23] is related to ours. However, the problems differ in two fundamental ways: 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.

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 [5, 28, 29, 45, 51, 61]. They have also been applied for bandwidth reduction in information theory [6], for graphics applications [24, 39], and for database applications [30]. Most of these applications have far coarser granularity than our test codes. We have shown that the overheads of these layouts can be reduced enough to make them useful for fine-grained computations.

Tiling and related work The compiler literature contains much work on iteration space tiling. Some authors aim at gaining parallelism [63], while others target improving cache performance [9, 62]. Kodukula *et al.* [33] present a data-centric approach to loop tiling called "shackling" that handles imperfect loop nests and can be composed to tile for multiple levels of the memory hierarchy. Carter *et al.* [10] discuss hierarchical tiling schemes for a hierarchical shared memory model. Porterfield's dissertation [46] discusses program transformations and software pre-fetching techniques to improve the cache behavior of scientific codes.

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 nonlinear data layouts can be viewed as an early binding version of this recommendation, where the copying is done possibly as early as compile time. Coleman and McKinley [15] choose tiles based on cache parameters to reduce self- and cross-interference misses, usually resulting in non-square tiles. They claim that their method outperforms the copy optimization recommended by Lam *et al.* [36]. We have compared our approach with theirs in §3.5. Rivera and Tseng [47, 48] discuss intra- and inter-array padding as a means of reducing conflict misses. Ghosh *et al.* [21, 22] present an analytical model for estimating cache misses for perfect loop nests.

A substantial body of work in the parallel computing literature deals with layout optimization of arrays. Representative work includes that of Mace [41] for vector machines; of various authors investigating automatic array alignment and distribution for distributed memory machines [11, 25, 31, 32]; and of Cierniak and Li [14] for DSM environments. The last paper also recognizes the importance of joint control and data optimization.

5 Conclusions and future work

The canonical array layout functions provided by current high-level programming languages can cause performance problems in hierarchical memory systems. Loop restructuring techniques can only partially correct these performance problems. Nonlinear layout functions provide a complementary solution. We have examined two nonlinear layout functions for multidimensional arrays that promise improved performance at low cost. The experimental and simulation results are promising, and we conclude that the L_{4D} and L_{MO} layouts deliver high performance, improve robustness of performance, and work well with multi-level memory hierarchies.

Our experimental implementations were based on C macros and functions, with no special compiler support. The observed performance is nonetheless quite respectable. It is our position that the ability to directly manipulate array layout has ramifications all the way up to algorithm design [50, 60], and is not something that compilers alone should manipulate. Replacing one layout by another is simple and easily mechanizable, but determining matching control-flow changes is significantly more complicated than loop tiling. Further research is needed to determine whether such changes can be automated, and how best such layouts can be packaged to enable users to easily tune their codes for the memory hierarchy.

We are currently working on developing an analytical model of the behavior of such layouts, similar to the models for canonical layouts [19, 21, 22, 58]. We are also working on increasing the size of our benchmark suite and covering more of the architectural design space through timing and simulation.

References

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

[2] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178, Santa Barbara, CA, July 1995.

[3] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick. Empirical evaluation of the CRAY-T3D: A compiler perspective. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.

[4] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Norwell, MA, 1993. ISBN 0-7923-9318-X.

[5] 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.

[6] 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.

[7] 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.

[8] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, CA, Oct. 1998.

[9] 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.

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

[11] 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.

[12] 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*, Saint-Malo, France, June 1999. To appear.

[13] T. M. Chilimbi, J. R. Larus, and M. D. Hill. Improving pointer-based codes through cache-conscious data placement. Technical Report CS-TR-98-1365, University of Wisconsin—Madison, Mar. 1998.

[14] 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.

[15] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, CA, June 1995.

[16] 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.

[17] K. Esseghir. Improving data locality for caches. Master's thesis, Department of Computer Science, Rice University, Houston, TX, Sept. 1993.

[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] C. Fricker, O. Temam, and W. Jalby. Influence of cross-interference on blocked loops: A case study with matrix-vector multiply. *ACM Trans. Prog. Lang. Syst.*, 17(4):561–575, July 1995.

[20] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall Series in Computational Mathematics. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981. ISBN 0-13-165274-5.

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

[22] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, Oct. 1998.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[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] 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.

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

[32] 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.

[33] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, NV, June 1997.

[34] 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.

[35] R. E. Ladner, J. D. Fix, and A. LaMarca. Cache performance analysis of algorithms. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Baltimore, MD, Jan. 1999. To appear.

[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] A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *The ACM Journal of Experimental Algorithmics*, 1:Article 4, 1996. Also available as Technical Report UW CSE TR# 96-02-03.

[38] A. G. LaMarca. *Caches and Algorithms*. PhD thesis, Department of Computer Science, University of Washington, Seattle, WA, 1996.

[39] 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.

[40] A. R. Lebeck and D. A. Wood. Active memory: A new abstraction for memory system simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(1):42–77, Jan. 1997.

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

[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] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997. ISBN 1-55860-320-4.

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

[45] 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.

[46] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, Houston, TX, May 1989. Available as technical report CRPC-TR89009.

[47] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 1998.

[48] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 353–360, Melbourne, Australia, July 1998.

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

[50] S. Sen, S. Chatterjee, and A. R. Lebeck. Towards a theory of cache-efficient algorithms. In preparation, Apr. 1999.

[51] 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.

[52] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[53] 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.

[54] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996. ISBN 1-55860-375-1.

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

[56] M. R. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 204–213, June 1998.

[57] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, Oct. 1994.

[58] D. Thiebaut and H. Stone. Footprints in the cache. *ACM Trans. Comput. Syst.*, 5(4):305–329, Nov. 1987.

[59] 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.

[60] U. Vishkin. Can parallel algorithms enhance serial implementation? *Commun. ACM*, 39(9):88–91, Sept. 1996.

[61] 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.

[62] 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.

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