Morton-order Matrices Deserve Compilers' Support*
Technical Report 533

David S. Wise† Indiana University

Jeremy D. Frens‡ Northwestern College

November 29, 1999

Abstract

A proof of concept is offered for the uniform representation of matrices serially in Morton-order (or Z-order) representation, as well as their divide-and-conquer processing as quaternary trees. Generally, d dimensional arrays are accessed as 2^d-ary trees. This data structure is important because, at once, it relaxes serious problems of locality and latency, while the tree helps schedule multi-processing. It enables algorithms that avoid cache misses and page faults at all levels in hierarchical memory, independently of a specific runtime environment.

This paper gathers the properties of Morton order and its mappings to other indexings, and outlines for compiler support of it. Statistics on matrix multiplication, a critical example, show how the new ordering and block algorithms achieve high flop rates and, indirectly, parallelism without low-level tuning.

Perhaps because of the early success of column-major representation with strength reduction, quadtree representation has been reinvented and redeveloped in areas far from the center that is Programming Languages. As target architectures move to multiprocessing, super-scalar pipes, and hierarchical memories, compilers must support quadtrees better, so that more programmers invent algorithms that use them to exploit the hardware.

CCS Categories and subject descriptors: E.1 [Data Structures]: Arrays; D.3.2 [Programming Languages]: Language Classifications—concurrent, distributed and parallel languages; applicative (functional) languages; D.4.2 [Operating Systems]: Storage management—storage hierarchies; E.2 [Data Storage Representations]: contiguous representations; F.2.1 [Analysis of Algorithms and Problem Complexity]: Numerical algorithms and problems— computations on matrices.
General Terms: Design, Performance.
Additional Key Words and Phrases: caching, paging, quadtree matrices, matrix multiplication.

1 Introduction

Maybe we've not been representing them efficiently for some time. Matrix problems have been fodder for higher-level languages from the beginning [4], and the classical representations of matrices appear in the textbooks. Advocates of column-major representation have tilted with adherents to row-major representation. Both alternatives use the same space, and both conventions yet survive; Crout still is mentioned together with Dolittle [25][26, p. 104].

But maybe both are archaic perspectives on matrix structure, which might best be represented using a third convention. Architecture has developed quite a bit since we had to pack scalars into small memory. With hierarchical—rather than flat—memory, only the faster memory is precious; with distributed processing instructions on local registers are far faster than those touching remote memory. And, of course, multiprocessing on many cheap processors demands less crosstalk than code for single threading on uniprocessors with unshared memory. Address space, itself, has grown so big that chunks of it are extremely cheap, when mapped to virtual memory and never touched. But fast cache, local to each processor, remains dear, and, perhaps, row-major and column-major are exactly wrong for it.

*Supported in part by the National Science Foundation under a grant numbered CDA93–03189.

†Supported, in part, by the National Science Foundation under a grant numbered CCR-9711269. Author's address: Computer Science Dept, 215 Lindley Hall, Bloomington, IN 47405–7104, USA. dswise@cs.indiana.edu

‡Work, in part, performed at Indiana University and supported by the U.S. Department of Education under a grant numbered P200A50237. Author's address: Computer Science Dept., Northwestern College, Orange City, IA 51041, USA. jdfrens@nwciowa.edu

This paper advocates Morton-order (sometimes called Z-order) storage for matrices. Consistent with the conventional sequential storage of vectors, it also provides for the usual cartesian indexing (row, column indices) into matrices. It extends to higher dimensional matrices. The main attraction of Morton order, however, is that it offers a memory locality to the quadtree-matrix algorithms whose use is yet growing. That is, we can provide cartesian indexing for the "dusty decks" while we write new divide-and-conquer and recursive-descent codes for block algorithms on the same structures. Thus, parallel processing is accessible at a very high level in decomposing a problem. Best of all, the content of any block is addressed sequentially, and their sizes vary naturally (they undulate) to fit the chunks of memory transferred between levels of the memory hierarchy.

Compilers have long supported implicit ameliorations on row-major or column-major ordering, such as strength reduction on operators and the introduction of blocking (or tiling) for locality. In order for Morton order to penetrate the stronghold of column-major libraries, compiler writers need to deliver its beautiful features, supporting it as a (perhaps *the*) standard representation for arrays. Many programmers may not need to know that their array is Morton-ordered at run time, but in order to take advantage of quadtree structure they need access (perhaps syntax) to use Morton or Ahnentafel indexing directly.

1.1 Historical Perspective

The forces that elevate the importance of Morton order come from several sources—none from traditional programming languages (PL). The primary forces are architectures of hierarchical memory. Memory may have been flat (albeit small) in the 1960's, but by 1970 paging arrived; now programs deal with register files, primary caches for both instructions and data, secondary cache, RAM, paging disks, a LAN, and the Internet.

Processors are not central in either control or cost. Multiprocessors abound, and any processor can now be superscalar, with several pipes. Multiprocessing bollixes up the classic fetch/execute tempo of processing, because memory conflicts will impede processing that is only threatened by unfavorable patterns of memory access. Cache coherency is another manifestation of this same problem.

Today's real cost is communication: moving bits around. Once the data is close to a processor, it pays to compute on it as much as possible. In terms of algorithms, the divide-and-conquer style, often associated with functional programming, becomes very attractive for decomposing computation into a few, big, independent chunks. These become fodder for schedules and data decomposition that preclude processor and memory conflicts.

Against this simple cure is legacy code following FORTRAN style, which dominates both the program libraries and the preparation of generations of programmers. This is especially valued among scientific programmers who still dominate the supercomputing marketplace because they have big problems and big codes that justify the big computers. When placing this proposal before those users, the PL community should recall a premise stated by Backus for the first popular language:

> It was our belief that if FORTRAN, during its first few months, were to translate any reasonable "scientific" program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger. . . To this day I believe that our emphasis on object-program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed. . . . [T]he next revolution in programming will take place only when *both* of the following requirements have been met: (a): a new kind of programming language,[1] far more powerful than those of today, has been developed and (b): a technique has been found for executing its programs at not much greater cost than of today's programs. [4, p. 29]

A justification for this requirement is provided by his description of what later came to be called *strength reduction* discussed below [4, p. 34][1].

In contrast to this wisdom is the fact that even FORTRAN compilers do not now deliver the relative performance that Backus envisioned. Supercomputing depends on libraries like BLAS3 [16], which are often hand-coded by the manufacturer to deliver the performance that sells its big iron. Current scientific codes use BLAS3 functions like assembly code—as if the target of compilers is not the hardware, but a model BLAS3 machine. In turn, the standard source code for which hardware is tailored is BLAS3—not the recursive blocking anticipated here.

Yet, underneath BLAS3 there remains FORTRAN's column-major representation, whose displacement is the target of this paper. Although BLAS3 performance is used later as an ideal benchmark, there is little hope of reproducing that performance through compilers that will not generate code that fine. Perhaps they should.

[1] His contemporary Turing Lecture suggests a functional style [5].

1.2 Onward

A major result of this paper is a demonstration of BLAS3 performance from a quadtree algorithm coded in C. That is, we show BLAS3 performance using Morton-ordered matrices and a quadtree algorithm that takes advantage of it. It is little more complex than the classic dot-product code, and relatively easy even, if one sees past its length to appreciate its symmetry. a problem here is that the latter code enjoys traditional compiler support (such as strength reduction, automatic blocking, and loop unrolling), but we had to simulate equivalent compilation (especially recursion unfolding.) Performance is within 10% of BLAS3 DGEMM on the SGI R8000 with a huge main memory and negligible caching for floating-point numbers; that is, without paging or caching. When run on an SGI R10000 with small memory and with both paging and caching, the BLAS3 code is left in the dust: 24 times slower.

This paper collects, consolidates, and clarifies the relationships among cartesian indexing, Morton indexing, Ahnentafel indexing, level-order indexing, and of course row-major, column-major and Morton orders. Also reviewed and extended are the remarkably efficient implementations of (additive) group operations on dilated integers that provide the common strength reductions on cartesian indexing to Morton-order matrices [41]. [15].

The new results are BLAS3 performance from Morton order and divide-and-conquer recursion, which lend themselves nicely to parallelism although they are not explored here [21, 12], as well as a review of transformations available to compilers to support this representation. Together, they suggest new formulations for old algorithms and, perhaps even, new ones. They offer high locality to solutions for many matrix problems, regardless of the memory parameters of the target machine.

This insensitivity has been described as *cache oblivious* because sizes of blocks to suit cache need never be specified, especially at compile time [23]. This perspective contrasts with blocking that is dependent on page-size [35, 18, 37] or cache-size [3, 10, 11, 32, 29, 47]. Indeed, much promising work on dense matrix computation using quadtrees and Morton order has appeared recently [12, 13, 21, 23, 28, 33, 43]. It follows earlier work on linked quadtrees that was aimed at sparse problems [6, 27]. This all follows its impact on graphics, and both geographic and spatial databases [40]. The beautiful features of Morton ordering are being rediscovered and redispersed because they have not yet been absorbed deeply into PL practice.

This paper has five sections. The second defines and explores the perspective of arrays as trees, reflected in Morton order, level order, and Ahnentafel and cartesian indexing. The third presents arithmetic on dilated integers that allows Morton order to support row and column traversal, of both elements and blocks. The next section presents timing results on matrix multiplication, exploring improvements that compilers should support. The fifth section concludes.

2 Basic Definitions

Morton presented his ordering in 1966 to index frames in a geodetic data base [36, 39]. He defines the indexing of the "units" in a two-dimensional array much as in Figure 1, and he points out the truncated indices available for enveloping blocks (subtrees), similar to Figure 2. Finally, he points out the conversion to and from cartesian indexing available through bit interleaving.

This interleaving is formalized below, but a quick description of it will orient the reader. Consider the hexidecimal constant `evenBits=0x55555555` from C; it has all the even bits set and all odd bits cleared. So, `oddBits=evenBits<<1` has the value `0xaaaaaaaa`, its complement. Now imagine a cartesian row index i with its significant bits "dilated" so that they occupy those set in `oddBits`. Similarly, a cartesian column index j is dilated so its significant bits occupy those set in `evenBits`. If the matrix A is stored as a vector, `a`, in Morton order, then $a_{i,j}$ can be accessed as C's `a[i+j]` *regardless* of the size of the matrix. (Try this for $i = 4$ and $j = 8$ in Figure 1; the Morton index is `0x020 + 0x040` or 96_{10}.) C programmers can write `a[i][j]` for this reference, but only after already fixing row length (or its stride) at compile time; or they write `a[i*rowStride +j]` when the stride is determined at run time.

The definitions that follow will narrow to two-dimensional arrays: matrices. The early ones are general, for higher-dimensions; a d-dimensional array is represented as a 2^d-ary tree.

2.1 Arrays

Definition 1 *In the following* $\mathrm{m} = 2^d$ *is the degree of the tree appropriate to the dimension* d.

If the maximal order in any dimension is n then the tree has maximal level $\lceil \lg n \rceil$.

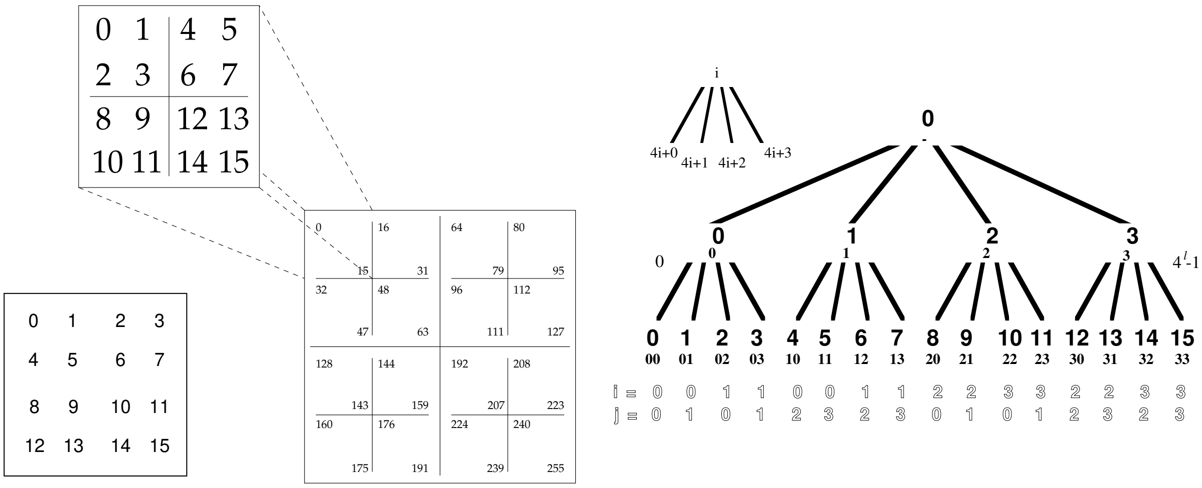

Figure 1: Row-major indexing of a 4×4 matrix, analogous Morton indexing (embedded in a larger one), and Morton indexing of the order-4 quadtree.

Definition 2 *A complete array has* level-order index 0. *A subarray (block) at level-order index i is either a scalar, or it is composed of m subarrays, with indices $mi+1, mi+2, \ldots, mi+m$.*

Definition 3 *The root of an array has* Morton-order *index* 0. *A subarray (block) at Morton-order index i is either a unit (scalar), or it is composed of m subarrays, with indices $mi+0, mi+1, \ldots, mi+(m-1)$ at the next level.*

So Morton indexing is zero-based at every level. The difference between the level-order index of a block at level l in an array and its Morton-order index is $(m^l - 1)/(m-1)$.

Theorem 1 *The difference between the level-order index of a block at level l in an array and its Morton-order index is $(m^l - 1)/(m-1)$.*

The difference is the number of nonterminal nodes above level l. Since each level is indexed by its zero-based scheme, it is necessary to know the level, as well as the Morton index, to identify a node. Figure 3 illustrates for a 4×4 matrix.

We introduce Ahnentafel indices because we find them immensely useful to identify blocks at all levels [48]. Algorithms that use recursive-descent (divide-and-conquer) to descend to a block of arbitrary size, or to return the index of a selected block, need only this single index to identify any subtree. But treat them only as identification numbers because there are gaps in the sequence between levels. Conversions among Ahnentafel indices, cartesian indices, Morton order, and level order are easy.

Ahnentafel indices come to us from genealogists who invented them for encoding one's pedigree as a binary, family tree. This generalization to m-ary trees is new.

Definition 4 [14] *A complete array has* Ahnentafel *index $m-1$. A subarray (block) at Ahnentafel index a is either a scalar, or it is composed of m subarrays, with indices $ma+0, ma+1, \ldots, ma+(m-1)$.*

Theorem 2 *The nodes at level l have Ahnentafel indices from $(m-1)m^l$ to $m^{l+1}-1$. The gap in indices between level $l-1$ and level l is $m^l(m-2)+1$.*

Theorem 3 *The level of a node with Ahentafel index a is $l = \lfloor \log_m a \rfloor = \lfloor \log_m \frac{a}{m-1} \rfloor$. The difference between the Ahnentafel index and the level-order index is $(m^{l+1}(m-2)+1)/(m-1)$. The difference between the Ahnentafel index and the Morton-order index is $(m-1)m^l$.*

The bothersome gaps between levels in Ahnentafel indexing do not arise in binary trees; Figure 4 illustrates. The strong similarity between level-order and Ahnentafel indexing in this common case perhaps explains why the latter is often overlooked. For instance, Knuth's level-order indexing, based at one, is off-by-one relative to our level-order indexing [31, p. 401], but on binary trees it coincides with Ahnentafel indexing.

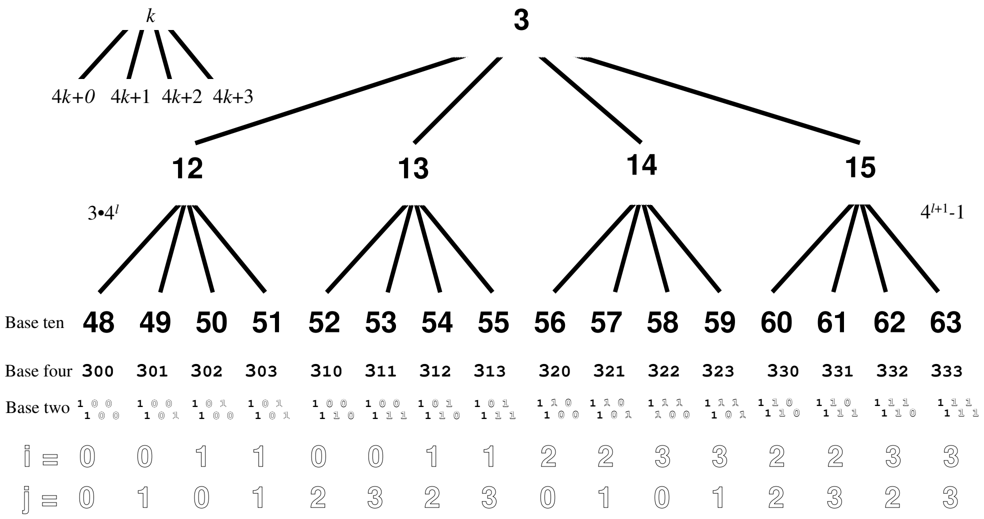

Figure 2: Ahnentafel indexing of the order-4 quadtree.

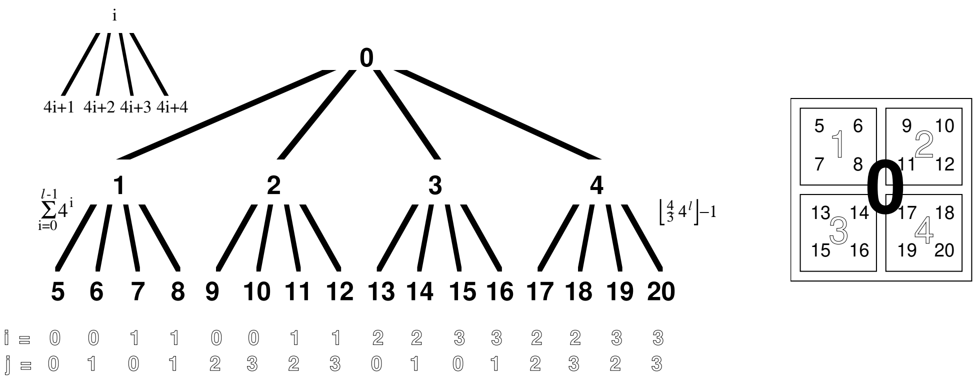

Figure 3: Level-order indexing of the order-4 quadtree and the associated submatrices.

2.2 Matrices

Hereafter, we assume $d = 2$ for matrices; so $m = 4$. In all the figures, the cartesian indices of the leaves appear in outlined font below the tree.

Corollary 1 *The difference between level-order index of a matrix block at level l and its Morton-order index is* $(4^l - 1)/3$.

Corollary 2 *The gap in Ahnentafel indices between level* $l - 1$ *and* l *is* $2^{2l+1} + 1$.

Corollary 3 *The difference between the Ahnentafel index of a block or element at level l of a matrix and its level-order index is* $(2^{2l+3} + 1)/3$. *The difference between its Ahnentafel and its Morton-order index is* $3 \cdot 4^l$.

Definition 5 *Let* w *be the number of bits in a (short) word. Each* $\mathrm{q_k}$ *is a modulo-4 digit (or* quat*). Each* q_k *is alternatively expressed as* $q_k = 2i_k + j_k$ *where* $\mathrm{i_k}$ *and* $\mathrm{j_k}$ *are bits.*

Cartesian indices will have w bits; Morton indices (and, later, dilated integers) have $2w$ bits.

Theorem 4 [36] *The Morton index*

$$\sum_{k=0}^{w-1} q_k 4^k = 2\sum_{k=0}^{w-1} i_k 4^k + \sum_{k=0}^{w-1} j_k 4^k.$$

corresponds to the cartesian indices: row $\sum_{k=0}^{w-1} i_k 2^k$ *and column* $\sum_{k=0}^{w-1} j_k 2^k$

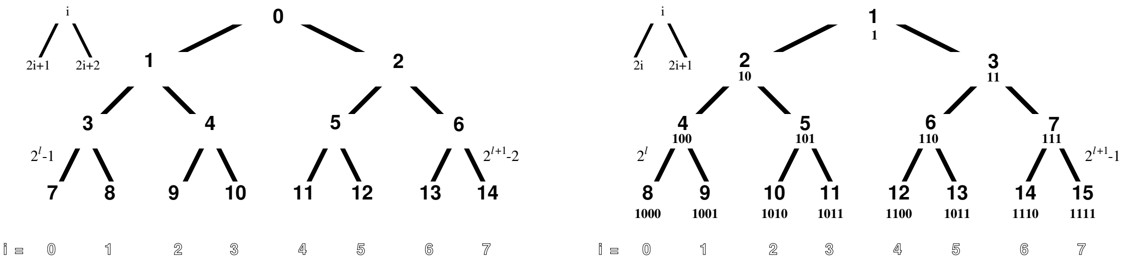

Figure 4: Level-order vs. Ahnentafel indexing of a binary tree.

The quats, read in order of descending subscripts, select a path from the root to the node, as in Figure 2.

Corollary 4 *Let $l = \lfloor \log_4 a \rfloor$. The Ahnentafel index, $a = \sum_{k=0}^{l} q_k r^k$ corresponds to the Morton index $\sum_{k=0}^{l-1} q_k r^k$.*

Of course, the set of bits, $\{i_k\}$, are the odd-numbered bits in the Morton index, and the $\{j_k\}$ are just the even-numbered bits; and also, without Corollary 3's two high-order **1** bits, also in an Ahnentafel index. This is Morton's bit interleaving of cartesian indices.

Ad hoc code to convert from cartesian indices to a Morton index by shuffling bits, or the inverse conversion that deals out the bits, can be expensive in processor cycles. If register-local, it seems either to be logarithmic in the magnitude of the indices (looping to shift/mask bit-by-bit) or to require table look-up (say, byte-by-byte). Fortunately, as the next section shows, most conversions can be elided.

Definition 6 *The integer $\overrightarrow{b} = \sum_{k=0}^{w-1} 4^k$ as a bit pattern has **1**'s only in even bits.*

In C code, $\overrightarrow{b}$ is called `evenBits`[2]. Also useful is $\overleftarrow{b} = 2\overrightarrow{b}$: the complementary mask, `oddBbits`.[3] By masking a Morton index with $\overrightarrow{b}$ and $2\overrightarrow{b}$, one extracts the bits of the column and row cartesian indices. Morton describes how to use these to obtain indices of neighbors. Easy identification of neighbors makes the indexing attractive for graphics in two dimensions and for spatial data bases in three.

It is remarkable how often these basic properties of Morton ordering have been reintroduced in different contexts [8, 9, 21, 30, 38, 46]. Samet gives an excellent history [40].[4]

When first introduced, it might seem that Morton indexing is poor for large arrays that are not square or whose size is not near a power of two, because it seems to waste space. With the valid elements justified to the north and west, the "wasted" space lands in the south and east. It can be viewed as padding, perhaps all zeroes. However, the perceived waste is merely *address* space. In hierarchical memory only its margin will ever be swapped into cache. That is, the "wastage" exists only within the logical/physical addressing of swapping disk; little valuable, fast memory is lost.

With these orders defined and the various index translations among them understood, we can summarize the programmer's view of Morton order:

- The elements of a matrix (an array) are mapped onto memory in Morton order.
- Row and column traversals can still be handled. See the next section.
- Restrict blocking to submatrices implicit in the quadtree (those with Ahnentafel indices.)
- If data is associated also with nonterminal blocks, store it in level order.
- Use Ahnentafel indices or level indices to control recursive-descent algorithms [48].
- With the data justified to the northwest, bounds-checking is only necessary in perimeter recursions toward the south and east. None is needed in recursions only to the northwest.

[2]It becomes a very important constant that cannot be loaded as an immediate value, because sign extension fails. But `((unsigned int)-1)/3)` uses two immediates to generate this pattern, regardless of word length.

[3] For three-dimensional matrices, one would use patterns of every third bit: `0x49249249` and shifted left one and two bits, as well as complements, like `0xb6db6db6`.

[4]Similar decompositions have been applied to mesh refinement which is a different problem [19, 46, *e.g.*]. That decomposition generates the vector space for a much larger (but sparse) matrix.

An early context for implementation of this design might be in HASKELL, which provides higher dimension arrays as aggregates (also, array comprehensions) but does not specify an internal representation. That is, no code depends on a particular ordering. ML and SCHEME are also available targets, because their specifications only demand arrays of one dimension. As we have seen, the various orders coincide there, so current implementations comply. They can be extended with array packages for higher dimensions that use Morton-ordering.

Perhaps not accidentally, these three languages do a good job with recursion in preference to iteration. That fact delivers implicit encouragement toward recursive-descent algorithms, even though iterative, row- or column-traversing, code can still be supported, as discussed in the next section. Morton order might open a wider market for scientific programming in these languages.

3 Cartesian indexing and Morton ordering

The following techniques for cartesian indexing of Morton-order arrays seem to be hardly known, a fact that is unfortunate because they make the structure useful for blocking matrices even if only used with cartesian indexing. In particular, this section newly shows how to compile row and column traversals (of blocks at any level of the tree) with the reductions in operator strength associated with optimizing compilers.

3.1 Dilated integers

The algebra of dilated integers is surprisingly old. Tocher outlined them in 1954 and under similar constraints to those again motivating us: non-flat memory with access time dependent on locality, and nearby information more rapidly accessible [44, p. 53–55]. But how the size of the memory has changed! Tocher needed fast access into a $32 \times 32 \times 32$ boolean array stored on a drum (4Kb!).

Schrack shows how to effect efficient cartesian indexing from row i and column j indices into Morton-order matrices [41]. The trick is to represent i and j as dilated integers, with information stored only in every other bit.

Definition 7 *The* even-dilated representation *of* $j = \sum_{k=0}^{w-1} j_k 2^k$ *is* $\sum_{k=0}^{w-1} j_k 4^k$*, denoted* $\overrightarrow{j}$*.*

Definition 8 *The* odd-dilated representation *of* $i = \sum_{k=0}^{w-1} i_k 2^k$ *is* $2\overrightarrow{i}$ *and is denoted* $\overleftarrow{i}$*.*

Theorem 5 $\lceil \lg \overrightarrow{j} \rceil + 1 = 2\lceil \lg j \rceil = \lceil \lg \overleftarrow{j} \rceil$.

The arrows suggest the justification of the meaningful bits within a dilated representation.

Theorem 6 *A matrix of m rows and n columns should be allocated a sequential block of* $\overleftarrow{m-1} + \overrightarrow{n-1} + 1$ *scalar addresses.*

This value is, of course, the Morton index of the southeast-most element of the matrix, plus one for (the northwest-most) one whose Morton index is 0. Not all of that sequence need be active; undefined data at the idle addresses will remain resident in the lowest level of the memory hierarchy. Only data in the active addresses will ever migrate to cache.

Theorem 7 [41] *If "$\equiv$" is read as boolean equality, and "$=$" as integer equality, then for unsigned integers*

$$(\overrightarrow{i} = \overrightarrow{j}) \equiv (i = j) \equiv (\overleftarrow{i} = \overleftarrow{j});$$

$$(\overrightarrow{i} < \overrightarrow{j}) \equiv (i < j) \equiv (\overleftarrow{i} < \overleftarrow{j}).$$

So comparison of dilated integers is effected by the same processor commands as those for ordinary integers.

Definition 9 *The infix operator $\wedge$ indicates bitwise conjunction; $\vee$ denotes bitwise disjunction.*

Instead of representing i and j internally, represent them as $\overleftarrow{i}$ and $\overrightarrow{j}$.

Theorem 8 *The Morton index for the $\langle i, j \rangle^{\mathrm{th}}$ element of a matrix is* $\overleftarrow{i} \vee \overrightarrow{j}$*, equivalently* $\overleftarrow{i} + \overrightarrow{j}$*.*

Often addition is chosen to associate with an adjacent constant addition at compile time. In other contexts, Theorem 12 for instance, disjunction is necessary.

Addition and subtraction of dilated integers can be performed with a couple of minor instructions.

Definition 10 *Addition ($\overset{\rightharpoonup}{+}$, $\overset{\leftharpoonup}{+}$) and subtraction ($\overset{\rightharpoonup}{-}$, $\overset{\leftharpoonup}{-}$) of dilated integers:*

$$\overrightarrow{j} \overset{\rightharpoonup}{-} \overrightarrow{n} = \overrightarrow{j-n}; \qquad \overleftarrow{i} \overset{\leftharpoonup}{-} \overleftarrow{n} = \overleftarrow{i-n}.$$

$$\overrightarrow{j} \overset{\rightharpoonup}{+} \overrightarrow{n} = \overrightarrow{j+n}; \qquad \overleftarrow{i} \overset{\leftharpoonup}{+} \overleftarrow{n} = \overleftarrow{i+n}.$$

Theorem 9 [41] $\qquad \overrightarrow{\jmath} \overset{\rightarrow}{-} \overrightarrow{n} = (\overrightarrow{\jmath} - \overrightarrow{n}) \wedge \overrightarrow{b}\,; \qquad \overleftarrow{\imath} \overset{\leftarrow}{-} \overleftarrow{n} = (\overleftarrow{\imath} - \overleftarrow{n}) \wedge \overleftarrow{b}\,.$

Theorem 10 [41] $\qquad \overrightarrow{\jmath} \overset{\rightarrow}{+} \overrightarrow{n} = (\overrightarrow{\jmath} + \overleftarrow{b} + \overrightarrow{n}) \wedge \overrightarrow{b}\,; \qquad \overleftarrow{\imath} \overset{\leftarrow}{+} \overleftarrow{n} = (\overleftarrow{\imath} + \overrightarrow{b} + \overleftarrow{n}) \wedge \overleftarrow{b}\,.$

The representations of twos-complement negative integers can also be dilated, and these computations remain valid. So, when one of the addends is a constant, a cycle can be saved by using the precomputed dilation of its negation[5]:

Corollary 5

$$\overrightarrow{\imath} \overset{\rightarrow}{+} \overrightarrow{n} = \overrightarrow{\imath} \overset{\rightarrow}{-} \overrightarrow{(-n)}; \qquad \overleftarrow{\imath} \overset{\leftarrow}{+} \overleftarrow{n} = \overleftarrow{\imath} \overset{\leftarrow}{-} \overleftarrow{(-n)}.$$

Theorem 11 *In twos-complement arithmetic* $\quad \overrightarrow{b} = \overrightarrow{(-1)}; \qquad \overleftarrow{b} = \overleftarrow{-1}.$

The previous two facts suggest that the C loop

```
for (int i=0; i<n; i++){...}
```

be compiled to `int nn=`$\overleftarrow{n}$ and

```
for (int ii=0, ii<nn, i=(ii-oddBits)&oddBits ){...}
```

Theorem 12 *When additions are cascaded, the final conjunction can be postponed and done once at the end.*

$$\overrightarrow{\jmath} \;\overset{\rightarrow}{+}\; \overrightarrow{k_0} \;\overset{\rightarrow}{+}\; \overrightarrow{k_1} \;\overset{\rightarrow}{+} \cdots \overset{\rightarrow}{+}\; \overrightarrow{k_n} = \langle \ldots [(\, [(\overrightarrow{\jmath} + \overleftarrow{b} + \overrightarrow{k_0}) \vee \overleftarrow{b}\,] + \overrightarrow{k_1}) \vee \overleftarrow{b}\,] + \cdots \overrightarrow{k_n} \rangle \wedge \overrightarrow{b}\,;$$

$$\overleftarrow{\imath} \;\overset{\leftarrow}{+}\; \overleftarrow{k_0} \;\overset{\leftarrow}{+}\; \overleftarrow{k_1} \;\overset{\leftarrow}{+} \cdots \overset{\leftarrow}{+}\; \overleftarrow{k_n} = \langle \ldots [(\, [(\overleftarrow{\imath} + \overrightarrow{b} + \overleftarrow{k_0}) \vee \overrightarrow{b}\,] + \overleftarrow{k_1}) \vee \overrightarrow{b}\,] + \cdots \overleftarrow{k_n} \rangle \wedge \overleftarrow{b}\,.$$

So, if i and j are not represented literally as integers, but translated by the compiler to their images, $\overleftarrow{\imath}$ and $\overrightarrow{\jmath}$, the resulting object code can be just a simple translation of what the programmer expects. Code like the following source might be demanded from the programmer, but it would be better introduced via a transformation by the helpful compiler:

```
#define evenBits ((unsigned int) -1)/3)
#define oddBits  (evenBits <<1)
#define  oddIncrement(i)  (i= ((i -  oddBits) &  oddBits))
#define evenIncrement(j)  (j= ((j - evenBits) & evenBits))
...
   for     (i = 0; i< rowCountOdd ;  oddIncrement(i))
     for   (j = 0; j< colCountEven; evenIncrement(j))
       for (k = 0; k< pCountEven  ; evenIncrement(k))
          c[i + j] += a[i + k] * b[j + 2*k];
```

The index computations of $\overrightarrow{k}$ and $\overleftarrow{k}$ in the innermost loop above reduce to three RISC instructions, plus two to sum the matrix-element addresses. Although fast constant time, this is still half-again what is available from column-major representation. With integer/floating processors tuned to column-major this difference may once have mattered. It no longer does, now that register operations are so fast relative to memory access. It becomes important, tough, for compilers to provide this arithmetic as loop control.

For three and higher dimensions, it seems best to implement arithmetic only for even-dilated integers, and to translate other representations from it. Otherwise too many constants [6] will occupy too many registers; for matrices there are only two and addition needs both.

Dilated integers simplify input and output of Morton-ordered matrices in human-readable raster order [9]. Such convenience is not computationally significant because I/O delays dominate that indexing, but it may be politically important just to make this matrix representation accessible to the programmers who need to experiment with it.

3.2 Space and Bounds

Theorem 6 tells how much address space an $m \times n$ matrix occupies, usually more than the mn positions containing data. As already mentioned, if the difference, $\overleftarrow{m-1} + \overrightarrow{n-1} + 1 - mn$, is large then most of it will never move into faster memory. Moreover, the larger the difference, the more remote (and cheaper) will be the bulk of the addresses allocated. The experiments in Section 4 demonstrate how allocated-but-untouched space does not interfere with processing.

[5] Mind the sign bits. Even-dilated addition has an arithmetic-overflow exception that is harder to trap.

[6] See Foonote 3.

Rows:Columns	Best Case	Worst Case
1:1	100%	33%
1:2	100%	33%
1:3	60%	33%
1:4	66%	22%

Table 1: Portion of allocated address space occupied by rectangular matrices. Unoccupied space never enters cache.

The size of that excess space in a rectangular matrix depends not just on the number of elements, but also on the ratio of the numbers of rows and columns. With Z ordering it is more favorable to orient the matrix so that there are more columns than rows—lest the southwest one fill as the northeast one sits empty—and the symmetry of the indexing machinery helps encourage that. Alternatively, if there are more rows than columns, as in underconstrained matrix problems, exchange the roles of $\overleftarrow{\imath}$ and $\overrightarrow{\jmath}$ or replace Z with И ordering in all the definitions.

It is easy to arrive at Table 1 of address space vs. matrix size. The best cases arise with 2^p rows; the worst occur with $2^p + 1$ rows, as the extra row and columns spill into new blocks that must be allocated but sit mostly unused. Remarkably, however, full space use is possible even with a 1:2 aspect ratio.

In block algorithms, cache hits make it better to iterate through Morton-order sequentially. For instance, if `b` is the initial index of a block from matrix A of size `n=`4^p that is to be zeroed, it is better to initialize it with a single, localized loop:

```
for (int i=0; i<n; i++) A[b+i]=0;
```

than to use column-major traversal.

Bounds checking on each row and column is elegant. First of all, Theorem 7 provides a fast check of either a Morton or an Ahnentafel index, `a`, against predilated row and column bounds:

```
if ( (a &oddBits)<rowCountOdd && (a &evenBits)<colCountEven ) ...
```

For Ahnentafel indices especially, the compiler can do even more. It can precompute two vectors of bounds on a row or column index at each level of the quadtree. The first, `rowBound[]`, is a bound on the perimeter of the matrix, to preclude access to southern and eastern padding. The second, `rowDense[]`, contains bounds on interior Ahnentafel indices that are north and west of this perimeter which, once passed, obviates the need for further bounds checking at any of its subtress/subblocks. Allocate two vectors of size h to contain row bounds on the Ahnentafel indices at each level, where h is the height of the tree. For both, the h^{th} entry is $\overleftarrow{r}$. Thereafter,

```
rowBound[level-1] = rowBound[level]>>2;
```

$$\texttt{rowDense[level-1] = (rowDense[level]}\overleftarrow{-}\overleftarrow{\texttt{1}}\texttt{) >>2;}$$

Column bounds are to be handled similarly, but with even dilations; we usually test row and column limits simultaneously.

One might not be overly precise on Ahnentafel bounds at the perimeter. Because extra space is often allocated to the south and east anyway, it has been observed to be cheaper to round up the actual bounds on the matrix to, say, the next multiple of eight and then to treat the margins as "active" padding. For a recursive block algorithm, the alternative is to detect blocks of size four, two, and one where there will be too few operations; with the smallest block at order eight, operations on it can be dispatched as unconditional, straight-line, superscalar code that is faster without the further bounds checking. That is, we choose to treat padding in small marginal blocks as sentinels initialized so they can participate in the gross algorithm without affecting its net result. Typically this is zeroes to the south or east, and the identity matrix to the southeast:

```
for (int j=0; j<n; j= (j-evenBits)&evenbits ) A[3*j]=1;
```

Finally, symmetric matrices and matrix transpose also are easy with Morton or Ahnentafel indices. If `m` is either kind of index, then the index of its reflected element or (untransposed) block is quickly computed by exchanging its even and odd bits in a dosido:

```
( (m &evenBits) <<1) + ( (m &oddBits) >>1)
```

3.3 Compiler ameliorations

From the beginning of FORTRAN, programmers have expected code improvements from compiling their high-level programs. The compiler needs knowledge of the algebra of dilated integers for Ahnentafel, Morton, and cartesian indexing in order to compile good object code.

3.3.1 Unfold and reroll

The most important of these arises more from the foreseeable use of Ahnentafel indices, which follows so simply from Morton order. Both these indexing schemes fix recursive block sizing to powers of two, index any block consecutively, and both suggest recursive, divide-and-conquer algorithms.

So, at the leaves of the recursion tree the programmer should expect a compiler "optimization" similar to that provided to iteration over cartesian indices. Specifically, he should expect *unfolding* of a level or three of recursions at the leaves in order to provide the straight-line code to fill an instruction pipe or instruction cache, and to allow superscalar processing [7]. The analog of loop unrolling, this is especially effective because many architectures are far less efficient on tight recursions than they are on tight loops; so, recursive code can benefit tremendously from simple unfolding there. Unsurprisingly C compilers, tuned to iteration, rarely provide this transformation.

But long runs of straight-line code are not sufficient alone. Since unfolding is exponential in the degree of recursion, those runs quickly explode. (For the examples in the next section it rises with powers of eight.) Our experiments showed that, to achieve a long run from a high degree, it is desirable to unfold more than one level (exponential explosion) and then to reroll the code back into a shorter loop (linear compression). That is, recursion unfolding is likely to be followed by an immediate rerolling into a loop. We have observed that the low-level optimizer in even the best compilers seems to be blind to long, linear runs of code, but attracted to shorter, encapsulated loops.

Rerolling into a shorter loop, especially, requires the algebra of dilated integers as presented in Section 3.1. That is, the innermost recursions must be rearranged into loops on Morton indices, already manifest as row/column sums of dilated integers.

3.3.2 Strength reduction

When Morton-order arrays are used with cartesian indices, the ideal compiler will augment the cartesian indices with their dilated representation, and later try to replace them entirely with that representation. Part of this translation maps the additive operators on cartesian integers to the Section 3.1 operations.

As this translation is made, however, various induction-variable optimizations could be threatened. The most important of these is *strength reduction*, by which slower operations within a loop are transmogrified to faster ones. Allen *et al.* catalog ten opportunities for strength reduction accelerating a loop [1]. Of the ten, *seven* depend only on addition, subtraction, and branching as provided in Section 3.1. The other three occur more rarely (exponentiation, trig functions, and continuously differentiable functions). Even as integers are translated into dilated notation, therefore, the reductions that programmers expect from compilers remain accessible if they introduce *additive* operators.

For instance, we are now building a source-to-source translator for C to translate all matrix declarations, for instance `double a[4][8]`, into Morton-order representations, like `double aa[96]`. All index expressions on integers will be transliterated into shadowing expressions on dilated integers; let `jj` be this even-dilated shadow for an integer variable `j`, and let `ii` become $\overleftarrow{\mathtt{i}}$, the odd dilation of `i`. References like `a[i][j]` will be translated into `aa[ii+jj]`. In the context of purely simple increments on `ii` and `jj`, the underlying `i` and `j` become useless.

Explicit, row-major indexing like `a[i*rowStride +j]` cannot be transliterated homomorphically, but a reference like `a[`$\overleftarrow{\mathtt{i * rowStride}}$` +jj]` will first be addressed via strength reduction to eliminate the multiplication. When this reference is wrapped in a loop incrementing `i`, then the usual strength reduction translates the multiplication into `product+=rowStride` with each iteration. Then, with odd-dilated shadows for `product` and `rowStride`, the addition from Theorem 10 or Corollary 5 suffices to complete the transliteration.

The fact that row and column traversals are tractable *and* supported by strength reduction means that many legacy codes using cartesian indices can successfully be recompiled to the Morton-order representation. A significant obstacle, however, is represented by overlays that should have long ago been set aside by virtual memory.

4 A motivating example

Perhaps trite as a theoretical example, matrix multiplication is perfect for the purposes of studying the representation of arrays and for experimenting with programming transformations on it. First of all, matrix multiplication is an algorithm representative of the marketplace of scientific computation. Indeed it is the critical step in many numeric algorithms, and others that don't use it explicitly do use coincident resources. Its asymptotic bounds on time and space, and its pattern of memory use are typical.

Expressed either iteratively or recursively, furthermore, it is a simple algorithm; experiments can be controlled. It has the critical characteristic of computation-intensive algorithms: that time is superlinear in space. So, effects from access to slower memory will always show up in timings of problems sufficiently large to use it. Finally, it typically uses parallel processing, even though parallelism is not tested here; this matrix representation and the algorithms tested are suited to it [21].

We had been exploring the clean expression of block-oriented divide-and-conquer algorithms over quadtrees. This effort began as exercises in applied functional programming, but developed a life of its own as we developed such algorithms for LU decomposition, including undulant pivoting and exact arithmetic and more recently for QR factorization [48, 20]. Others have implemented LU and Cholesky factorization and other classic algorithms over the quadtree representation [45, 27, 22].

Since we seek style and consistent representation before raw performance, translating internal blocks to and from column-major representation just to get impressive BLAS3 performance from DGEMM is not a choice [12, 13]. The idea is to explore a uniform representation, and to discover the support it needs, as well as the algorithms it suggests.

Morton-order representation also fits Strassen's algorithm perfectly [42, 26]. We advocate a homogeneous matrix representation for all these algorithms. Morton-order indexing works well in our experiments; we have attained BLAS3 performance with a homogeneous matrix representation.

4.1 Experimental results

Our underlying effort here is a source-to-source synthesis from the matrix-multiplication code in Figure 5 which was originally motivated by performance under parallel processing and hierarchical memory. It has the property that one of the three blocks in play at any level of the quadtree remains in cache between two successive calls. The exercise in hand-compilation uses Morton order and only two compiling techniques: unfolding of base cases (with rerolling), and strength reduction on the computation and bounding of indices. Machine parameters, like cache size, was used only late in the game, and an optimizing compiler could use these far better.

The Figure 5 algorithm is implemented using Ahnentafel indices to control the recursion. The `offset` is set to the Corollary 3 difference between Ahnentafel and Morton indices, and presubtracted from all matrix references. Bounds checking follows Section 3.2. The arguments to `up_mult` and `down_mult` are actually passed in local registers in order to avoid stack use; since the Ahnentafel computations can be inverted simply by a shift, they need not be stacked. Moreover, the base case is unfolded from 1×1 to 8×8 in order to avoid excessive overhead from function call and to take advantage of superscalar processing (like loop unrolling). Balanced scheduling of parallel processes is suggested by extra braces in the figure; statements separated only by semicolons remain sequential because of shared write addresses [21].

4.2 Experiments

Timings were run on three machines:

- Sun 400 MHz UltraSPARC-II v9 processor including sparcv9 floating-point and 1Gb main memory (shared), 16Kb on-chip instruction cache, 16Kb on-chip data cache, and 4Mb secondary cache. Compiler CC with optimization `-fast`.
- SGI 195MHz R10000 ip30 processor with R10010 floating-point chip and 128 Mb main memory, 32 Kb instruction cache, 32Kb data cache. Secondary unified instruction/data cache of 1 Mb. Compiler CC with optimization `-Ofast -64`.

```
#define nw(i) (i*4+0)
#define ne(i) (i*4+1)
#define sw(i) (i*4+2)
#define se(i) (i*4+3)
#define pop(i) (i/4)

int offset;
register Scalar* A_matrix, B_matrix, C_matrix;

void multiply (Matrix a, Matrix b, Matrix c) {
  offset   = a.offset;
  A_matrix = a.matrix;
  B_matrix = b.matrix;
  C_matrix = c.matrix;
  up_mult (3, 3, 3);
}

static void dn_mult (register Index i_C, register Index i_A, register Index i_B) {
    /* All assertions about cache refer to extreme corners of    */
    /* the named quadrant.                                       */
  if (outOfBounds(i_A) || outOfBounds(i_B)) {}
  else if (i_A >= offset)
    C_matrix[i_C-offset] += A_matrix[i_A-offset] * B_matrix[i_B-offset];
  else { /* Precondition: one extreme block of C_ne,A_nw, or B_ne in cache. */
  {{dn_mult (ne (i_C), nw (i_A), ne (i_B));     /* Leaving C_ne_nw in cache. */
    up_mult (ne (i_C), ne (i_A), se (i_B));}    /* Leaving B_se_ne in cache. */
   {dn_mult (se (i_C), se (i_A), se (i_B));     /* Leaving C_se_nw in cache. */
    up_mult (se (i_C), sw (i_A), ne (i_B));}}   /* Leaving A_sw_nw in cache. */
  {{up_mult (sw (i_C), sw (i_A), nw (i_B));     /* Leaving C_sw_nw in cache. */
    dn_mult (sw (i_C), se (i_A), sw (i_B));}    /* Leaving B_sw_ne in cache. */
   {up_mult (nw (i_C), ne (i_A), sw (i_B));     /* Leaving C_nw_nw in cache. */
    dn_mult (nw (i_C), nw (i_A), nw (i_B));}}
          /* Postcondition: extreme blocks of C_nw, A_nw, B_nw in cache.     */
  }
}

static void up_mult (register Index i_C, register Index i_A, register Index i_B) {
  if (outOfBounds(i_A) || outOfBounds(i_B)) {}
  else if (i_A >= offset)
    C_matrix[i_C -offset] += A_matrix[i_A -offset] * B_matrix[i_B -offset];
  else {  /* Precondition one extreme block of C_nw,A_nw, or B_nw in cache.  */
  {{up_mult (nw (i_C), nw (i_A), nw (i_B));     /* Leaving C_nw_ne in cache. */
    dn_mult (nw (i_C), ne (i_A), sw (i_B));}    /* Leaving B_sw_nw in cache. */
   {up_mult (sw (i_C), se (i_A), sw (i_B));     /* Leaving C_sw_ne in cache. */
    dn_mult (sw (i_C), sw (i_A), nw (i_B));}}   /* Leaving A_sw_nw in cache. */
  {{dn_mult (se (i_C), sw (i_A), ne (i_B));     /* Leaving C_se_nw in cache. */
    up_mult (se (i_C), se (i_A), se (i_B));}    /* Leaving B_se_ne in cache. */
   {dn_mult (ne (i_C), ne (i_A), se (i_B));     /* Leaving C_ne_nw in cache. */
    up_mult (ne (i_C), nw (i_A), ne (i_B));}}
          /* Postcondition:  extreme blocks of C_ne, A_nw, B_ne in cache.    */
  }
}
```

Figure 5: Two-miss algorithm for quadtree matrix multiply [21].

- SGI 75MHz R8000 ip21 processor with R8010 floating-point chip and 2 Gb main memory 8-way interleaved, 16 Kb instruction cache, 16Kb data cache **but** not for floats. Secondary unified instruction/data cache of 4 Mb. Compiler cc with optimization `-Ofast -64`.

The matrices are square, of the order given in the first column. We have run extensive experiments on the Morton-order representation and the statistics grow smoothly. There are no hiccoughs due to striding, for instance, because the blocking is inherently sequential in memory. These matrix sizes apear in our earlier paper on the parallel algorithm [21].

The original three algorithms are

BLAS3: The manufacturer's DGEMM library routine. That on the Sun and the SGI R100000 seem to be sensitive to matrices with orders that are powers of two; probably striding trouble with the column-major representation.

INPROD: The usual three-nested-loop inner-product matrix multiplication on column-major matrices, similar to that in Section 3.1. The stride is the number of rows and, so, there is an obvious sensitivity to orders that are powers of two, where memory addresses clash.

INPROD4: This is the first test on Morton-ordered matrices. The algorithm is essentially that in Section 3.1, but it has been blocked to a 4×4 "element." That is, the loops increment in steps of $\overleftarrow{4}$ or $\overrightarrow{4}$, and the body of the inner loop is an in-line 4×4 matrix multiply. Compilers on ordinary `for` loops provide such blocking and unrolling; with the strange loop control we had to synthesize ours by hand.

QUADTREE8: Figure 5 with the base case unfolded thrice (8×8) and then rerolled to a loop iterated four times.

The last algorithm tests several features that a good compiler would inject into a recursive algorithm on Ahnentafel indices. That is, the unfolded base case of 512 multiply-adds was rolled into a loop on 16 dot-products of vectors of size 8. This transformation was done by hand, again, and is only one of many possibilities for handling this problem. It was difficult to shape the strange index patterns into something that a loop-oriented C compiler would optimize. This one worked.

Both bounds vectors in Section 3.2 were used, and so the code of Figure 5 was duplicated: for the general case of perimeter blocks and for the case where both factors are dense, in the center of the array. No further bounds testing occurred in the latter case.

Blocks at the perimeter were forced to be 8×8 even if this overlapped into padding; the padding had been preset to zero, and so would not corrupt the aggregate product. If this algorithm were applied *within* another program, it would only be applied to blocks in the Morton/Ahnentafel indexing and, so, this wash into "unallocated" space would never occur except at the global perimeter. If blocking were regularized—as it is with Morton indexing—the strategy of requiring this minimal block size seems reasonable for any representation; bounds checking at less than this granularity seems to be wasteful in all blocked algorithms.

4.3 Results

These results are typical times expressed to three decimal places, but accurate only to two. In all three tables, the order is immediately followed by the MINimum time to do the necessary floating-point multiply-add operations, computed only from the architecture and clock speed; it ignores memory access. An immediate observation is that even the BLAS3 code on the UltraSPARC is running at less than half the specified flop rate. If the manufacturer's BLAS does not take advantage of its superscalar processor, we cannot expect its C compiler to do any better.

The columns to the right present the number of voluntary swaps reported by UNIX's `getrusage`. This measure is presented in order to report thrashing of the paging algorithm. We could have presented the number of major page faults instead, but they track this swap count, which is quietly zero without any paging.

Both the BLAS3 and the INPROD codes on the UltraSPARC and the R10000 exhibit a sensitivity to matrix striding (Tables 2 and 3). This is not visible with BLAS3 on the R8000,[7] although INPROD still shows it (Table 4). The Morton-order representation does not exhibit this problem because its blocking does not "stride" through the array.

[7] BLAS3 times in Table 4 are old [21] because there remains a problem linking to a revised library. So swapping information is not available, though surely zero in its monster memory.

Order	min time	BLAS3	INPROD col-major	INPROD4 Morton order	QUADTREE8 to 8×8 basis	BLAS3 Volun. swp.	INPROD Volun. swp.	INPROD4 Volun. swp.	QUADTREE8 Volun. swp.
1023	2.68	7.97	276.1	27.6	19.06	0	0	0	0
1024	2.69	7.57	392	27.5	19.1	0	0	0	0
1025	2.69	7.80	269.8	28.2	19.94	0	0	0	0
1116	3.48	10.0	259.7	37.3	24.78	0	0	0	0
1280	5.25	15.17	747.5	57.5	37.1	0	0	0	0
1536	9.05	26.17	1313	69.1	64.1	0	0	0	0
1792	14.4	41.9	1817	172	101.9	0	0	0	0
1854	16.0	47.2	1211	194.3	113.2	0	0	0	0
2047	21.5	90.7	1960	264.3	151.4	0	1	0	0
2048	21.5	99.8	3243	263.5	151.4	0	0	0	0
2049	21.5	100.8	1760	265.6	154.7	0	0	0	0
2108	23.0	69.3	1804	290.9	164.9	0	2	0	3
2340	32.1	94.1	2469	397	227.7	0	0	0	0
3050	71.0	212.3	5660	889	500	0	0	0	0
4095	172	1324	14,480	2255	1203	2719	0	36	0
5000	313	996	28,530	4085	2240	5310	64,890	55,347	66,118

Table 2: Uniprocessor running times (seconds) and voluntary swaps for 4 algorithms on SUN ULTRASPARC-II

Order	min time	BLAS3	INPROD col-major	INPROD4 Morton order	QUADTREE8 to 8×8 basis	BLAS3 Volun. swp.	INPROD Volun. swp.	INPROD4 Volun. swp.	QUADTREE8 Volun. swp.
1023	5.49	8.68	24.53	24.83	8.72	2	0	0	0
1024	5.51	8.47	61.2	24.84	8.71	0	0	0	0
1025	5.52	10.23	29.85	30.15	14.18	0	518	582	582
1116	7.13	9.26	14.73	37.88	15.69	0	619	637	635
1280	10.8	14.51	31.45	55.32	24.11	0	812	784	787
1536	18.6	24.84	124.8	95.82	37.3	8	1175	1142	1063
1792	29.5	45.2	126.6	152.2	57.4	12	1610	1613	1327
1854	32.7	51.4	137.5	170.5	65.1	44	1722	1740	1681
2047	44.0	189.9	673	227.1	91.9	4160	2207	2086	2472
2048	44.1	151.0	1039	225.9	88.0	4150	2205	2087	2361
2049	44.2	154.9	670	273.7	142.3	4600	7371	4480	7610
2108	48.1	125.7	307.2	286.7	140.2	5750	7814	4660	7650
2340	65.7	209.8	1279	387	188.4	11,140	47,100	5710	11,260
3050	145.5	7634	26,350	862	382	927,000	2,481,000	15,750	21,500
4095	352	20,690	66,864	34,800	879	2,180,000	5,990,000	4,230,000	46,500

Table 3: Uniprocessor running times (seconds) and voluntary swaps for 4 algorithms on SGI R10000

Order	min time	BLAS3	INPROD col-major	INPROD4 Morton order	QUADTREE8 to 8 × 8 basis	BLAS3 Volun. swp.	INPROD Volun. swp.	INPROD4 Volun. swp.	QUADTREE8 Volun. swp.
1023	7.14	9.94	22.51	22.38	10.8	*	0	0	0
1024	7.15	9.94	43.72	22.36	10.81	*	0	0	0
1025	7.18	9.97	22.83	22.67	11.48	*	0	0	0
1116	9.27	12.74	36.0	29.38	14.29	*	0	0	0
1280	14.0	19.43	53.37	44.9	21.23	*	0	0	0
1536	24.2	33.6	96.4	77.7	36.5	*	0	0	0
1792	38.4	53.3	167.4	124.5	58.0	*	0	0	0
1854	42.5	58.2	197.2	138.1	64.7	*	0	0	0
2047	57.2	78.7	325	186.0	86.6	*	0	0	0
2048	57.3	79.5	731	186.7	86.7	*	0	0	0
2049	57.4	79.0	327	188.1	89.3	*	0	0	0
2108	62.4	86.5	308	209.4	95.7	*	0	0	0
2340	85.4	116.9	563	315.1	131.2	*	0	0	0
3050	189.2	257.6	1120	819	289	*	0	0	0
4095	458	626	3250	2161	694	*	0	0	0

Table 4: Uniprocessor running times (seconds) and voluntary swaps for 4 algorithms on SGI R8000

Any blocking algorithm uses a block that is, itself, sequential in address space, so the cache addressing is almost certain to be clean; rarely a block will conflict with another operand's but never with itself. This observation is already an endorsement of Morton order.

Surprisingly, the INPROD4 code using Morton order was also quite fast, around thrice BLAS3 speeds across all three machines. Based on the observation that BLAS3 runs close to machine capacity, we observe that this much performance for so little coding effort becomes another endorsement.

The QUADTREE8 algorithm competes very well with DGEMM. On the SGI R8000 it runs within 10% of BLAS3, which is quite an improvement over the 600% difference of [21]; no parallel performance is offered there, but there is no reason not to see a similar speedup. The improvement can be attributed to the unfolding, rerolling, register-passing procedure calls, and to better compiler optimization.

The most surprising result is the relative performance of QUADTREE8 on the R10000. Noticing its small-machine context, we observe a 20-fold improvement at order 3050, away from a power-of-two. The wretched performance of BLAS3 (and column-major INPROD) can be attributed to the caching of the R10000, and paging behavior of the small main memory. The R10000 does retain floats in primary cache, so Morton order will immediately look better than on the R8000; this improvement shows for small orders. For larger and larger matrices we can see the 128Mb memory constraint forcing paging on the column-major representations, and less severly on Morton order. With memory swapping at two levels, the blocking for column-major cannot keep up. Morton order is not sensitive to a single block size, however; whatever is the size of a cache or a page, it has a block that fits nicely. As a result, it is paging 100 times less for the larger problems.

Overall, the algorithms using Morton order keep up with, and eventually beat BLAS3 in this DGEMM race, based on relatively unspecific C code. They contrast with those tuned by hand or by parametrization of architecture [47].

5 Conclusion

This essay appeals for compiler support for Morton ordering, a different and promising representation of arrays. It has already been shown to be convenient and effective for expressing parallel algorithms [21, 23] but, as demonstrated here, it suffers from the quality of compiler support that was thought essential even for the first higher-level programming language for scientific programming.

Morton-order indices, used for internal representation, is easily converted to/from Ahnentafel indexing, used for recursion control, and conversions are readily available between them and cartesian indexing, as used by classic codes, as well as level-order indexing, used to decorate internal nodes of the quadtree structure. Each has its own use, but the translations among them are readily available. It is particularly convenient that, as late as run time, indexing proceeds without provision for column-size or striding; good cache use and paging arises from the "right" algorithms and blocking by subtrees.

It is always possible to convert from column-major to Morton order in time proportional to the size of the matrix, so why bother changing conventions? First, any large conversion is memory intensive, and so especially wasteful. One can also anticipate successive conversions as several algorithms are applied over different representations. Second, we need Morton order used extensively to encourage recursive/quadtree, localized algorithms even though cartesian indexing is also available there. Third, newer languages, like HASKELL [2, 17], have aggregate operations on matrices that can be cast into any representation; Morton order is ideal for them and this common representation would drop barriers between them and other languages. Finally, programmers need more support for block-local decompositions and divide-and-conquer on modern architectures. Morton order leads them to these low-level efficiencies, but they need the minimal compiler support outlined here.

The INPROD4 results show how programs that only use loops can benefit from run-time locality that would result from the compiler's transformation of his matrix into Morton order. It would need to convert his `do`-loop indices into dilated representation, and to use a block tiling and a schedule that fits the underlying quadtree. Of course, the usual compiler tricks, like strength reduction must remain.

PL occupies high ground in Computing Research; it provides the tools with which all other work is done. If it is to fulfill its obligation to help different fields communicate with one another, then it has to listen to what they are already saying. Morton order has long been in use in several disparate fields, and its interest is yet rising in the oldest ones. So it is surprising that it has not been adopted by PL as a supported tool. We ask that this happen soon.

6 Acknowledgements

Special thanks to Hanan Samet who provided the earlier references.

References

[1] F. E. Allen, J. Cocke, & K. Kennedy. Reduction of operator strength. In S. W. Muchnick & N. D. Jones (eds.) *Program Flow Analysis: Theory and Applications* Englewood Cliffs, NJ: Prentice-Hall (1981), 79–101.

[2] S. Anderson & P. Hudak. Compilation of HASKELL array comprehensions for scientific computing. *Proc. ACM SIGPLAN '90 Conf. on Program. Language Design and Implementation, SIGPLAN Not.* **25**, 6 (June 1990), 137–149.

[3] J. M. Anderson, S. P. Amarainghe, & M. S. Lam. Data and computation transformations for multiprocessors. *Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.* **30**, 8 (August 1995), 166–178.

[4] J. Backus. The history of FORTRAN I, II, and III. In R. L. Wexelblat (ed.), *History of Program. Languages*, New York, Academic Press (1981), 25–45. Also preprinted in *SIGPLAN Not.* **13**, 8 (August 1978), 165–180.

[5] J. Backus. Can programming be liberated from the von Neumann style? A functional style and the algebra of its programs. *Commun. ACM* **21**, 8 (August 1978), 613–641.

[6] P. Beckman. *Parallel LU Decomposition for Sparse Matrices Using Quadtrees on a Shared-Heap Multiprocessor.* Ph.D. dissertation, Indiana University, Bloomington (1993).

[7] R. M. Burstall & J. Darlington. A transformation system for developing recursive programs. *J.ACM* **24**, 1 (January 1977), 44–67.

[8] F. W. Burton & J. G. Kollias. Comment on 'The explicit quad tree as a structure for computer graphics.' *Comput. J.* **26**, 2 (May 1983), 188.

[9] F. W. Burton, V. J. Kollias, J. G. Kollias. Real-time raster-to-quadtree and quadtree-to-raster conversion algorithms with modest storage requirements. *Angew. Informatik* **4** (1986), 170–174.

[10] S. Carr & R. B. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Trans. Math. Softw.* **23**, 3 (September 1997).

[11] M. Cierniak & W. Li. Unifying data and control transformations for distributed shared-memory machines. *Proc. ACM SIGPLAN '95 Conf. on Program. Lang. Design and Implementation, SIGPLAN Not.* **30**, 6 (June 1995), 205–217.

[12] S. Chatterjee, A. R. Lebeck, P. K. Patnala, & M. Thottenthodi. Recursive array layouts and fast parallel matrix multiplication. *Proc. 11th ACM Symp. Parallel Algorithms and Architectures*, 222–231. `http://www.acm.org/pubs/citations/proceedings/spaa/305619/p222-chatterjee/`

[13] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, & M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. *Intl. Conf. on Supercomputing, 1999* (to appear). `http://info.acm.org/pubs/contents/proceedings/supercomputing/`

[14] H. G. Cragon A historical note on binary tree. *SIGARCH Comput. Archit. News* **18**, 4 (Dec 1990), 3.

[15] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, & T. von Eicken. LogP: a practical model of parallel computation. *Commun. ACM* **39**, 11 (November 1996), 78–85.

[16] J. Dongarra, J. DuCroz, S. Hammarling, & R. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.* **14**, 1 (March 1988), 1–17.

[17] J. Fasel & P. Hudak. A gentle introduction to HASKELL. *SIGPLAN Not.* **27**, 5 (May 1992), T-48–T-53.

[18] P. C. Fischer & R. L. Probert. Storage reorganization techniques for matrix computation in a paging environment. *Commun. ACM* **22**, 7 (July 1979), 405–415.

[19] G. C. Fox. A graphical approach to load balancing and sparse matrix-vector multiplication. In M. Schultz (ed.) *Numerical Algorithms for Modern Parallel Architectures, IMA Vol. in Math. & Appl.* **13**. New York: Springer (1988) 37–61.

[20] J. Frens. *Matrix Factorization Using a Block-Recursive Structure and Block-Recursive Algorithms.* PhD dissertation, Indiana University, Bloomington (in progress).

[21] J. Frens & D. S. Wise. Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code. *Proc. 1997 ACM Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.* **32**, 7, (July 1997) 206–216.

[22] M. Frigo, C. E. Leiserson, & K. H. Randall. The implementation of the CILK-5 multithreaded language. *Proc. ACM SIGPLAN '98 Conf. on Program. Language Design and Implementation, SIGPLAN Not.* **33**, 6 (May 1998), 212–223.

[23] M. Frigo, C. E. Leiserson, H. Prokop, & S. Ramachandran. Cache-oblivious algorithms, Extended abstract. Lab for Computer Science. M.I.T. (May 1999). `http://supertech.lcs.mit.edu/cilk/papers/abstracts/FrigoLePr99.html`

[24] Irene Gargantini. An effective way to represent quadtrees. *Commun. ACM* **12**, 12 (December 1982), 905–910.

[25] G. H. Golub & C. F. Van Loan. *Matrix Computations,* 1st ed., Baltimore: The Johns Hopkins University Press (1989), 86.

[26] G. H. Golub & C. F. Van Loan. *Matrix Computations* 3rd ed., Baltimore: The Johns Hopkins University Press (1996).

[27] P. W. Grant, J. A. Sharp, M. F. Webster, & X. Zhang. Experiences of parallising finite-element problems in a functional style. *Softw. Prac. Exper.* **25**, 9 (September 1995), 947–974.

[28] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Develop.* **41**, 6 (November 1997), 737–755.

[29] N. J. Higham. Exploiting fast matrix multiplication within the Level 3 BLAS. *ACM Trans. Math. Softw.* **16**, 4 (December 1990), 352–368.

[30] Y. C. Hu, S. L. Johnsson & S.H. Teng. High performance Fortran for highly irregular problems. *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.* **32**, 7 (July 1977), 13–24.

[31] D. E. Knuth. *The Art of Computer Programming* **I,** *Fundamental Algorithms* (3rd ed.), Reading, MA: Addison-Wesley, (1997).

[32] I. Kodukula, N. Ahmed, & K. Pingali. Data-centric Multi-level Blocking. *Proc. 1997 ACM Conf. on Program. Language Design and Implementation, SIGPLAN Not.* **32**, 7, (May 1997) 346–357.

[33] Dinh Lê. *Block, Systolic, and Recursive Block Decompostion Schemes for Randomized Gaussian Elimination*. PhD dissertation, Univ. of California at Los Angeles (1996).

[34] M. Lee & H. Samet. Navigating through triangle meshes implemented as linear quadtrees. Submitted for publication. `http://www.cfar.umd.edu./ftp/TRs/CVL-Reports-1998/TR3900-lee.ps.gz`.

[35] A. C. McKellar & E. G. Coffman, Jr. Organizing matrices and matrix operations for paged-memory systems *Commun. ACM* **12**, 3 (March 1969), 153–165.

[36] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Ottawa, Ontario: IBM Ltd. (March 1, 1966).

[37] G. Newman Organizing arrays for paged memory systems. *Commun. ACM* **38**, 7 (July 1995), 93–103 + 108–110.

[38] J. A. Orenstein & T. H. Merrett. A class of data structures for associative searching. *Proc. 3rd ACM SIGACT–SIGMOD Symp. on Principles of Database Systems* (1984), 181–190.

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

[40] H. Samet. *The Design and Analysis of Spatial Data Structures* Reading, MA: Addison-Wesley, (1990), §2.7.

[41] G. Schrack. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Underst.* **55**, 3 (May 1992), 221-230.

[42] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.* **13** (1969), 354–356.

[43] M. Thottethodi, S. Chatterjee, & A. R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency *Proc. Supercomputing '98*. Los Alamitos, CA: IEEE Computer Society (1998). /cdrom/sc98/sc98/techpape/sc98full/thotteth/index.htm

[44] K. D. Tocher. The application of automatic computers to sampling experiments. *J. Roy. Statist. Soc. Ser. B* **16**, 1 (1954), 39–61.

[45] S. Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.* **18**, 4 (October 1997), 1065–1081.

[46] M. S. Warren. & J. K. Salmon. A parallel hashed oct-tree N-body problem. *Proc. Supercomputing '93.* Los Alamitos, CA: IEEE Computer Society Press (1993), 12–21.

[47] R. C. Whaley & J. J. Dongarra. Automatically tuned linear algebra software. *Proc. Supercomputing '98*. Los Alamitos, CA: IEEE Computer Society (1998).

[48] D. S. Wise. Undulant block elimination and integer-preserving matrix inversion. *Sci. Comput. Program.* **33**, 1 (January 1999), 29–85.
`http://www.cs.indiana.edu/ftp/techreports/TR418.html`