Calvin University
Calvin Digital Commons

University Faculty Publications | University Faculty Scholarship

1-1-2001

Language support for Morton-order matrices

David S. Wise
Indiana University Bloomington

Gregory A. Alexander
Indiana University Bloomington

Jeremy D. Frens
Calvin University

Follow this and additional works at: https://digitalcommons.calvin.edu/calvin_facultypubs

 Part of the Computer Sciences Commons

Recommended Citation

Wise, David S.; Alexander, Gregory A.; and Frens, Jeremy D., "Language support for Morton-order matrices" (2001). *University Faculty Publications*. 558.
https://digitalcommons.calvin.edu/calvin_facultypubs/558

This Article is brought to you for free and open access by the University Faculty Scholarship at Calvin Digital Commons. It has been accepted for inclusion in University Faculty Publications by an authorized administrator of Calvin Digital Commons. For more information, please contact dbm9@calvin.edu.

Language Support for Morton-order Matrices*

David S. Wise† ‡
Gregory A. Alexander
Computer Science Dept.
Indiana University
Bloomington, IN 47405, USA
dswise@cs.indiana.edu

Jeremy D. Frens§
Dept. of Computer Science
Calvin College
Grand Rapids, MI 49546, USA
jdfrens@calvin.edu

Yuhong Gu¶
Oracle Corporation
One Oracle Drive
Nashua, NH 03062, USA
yuhong.gu@oracle.com

ABSTRACT

The uniform representation of 2-dimensional arrays serially in Morton order (or И order) supports both their iterative scan with cartesian indices and their divide-and-conquer manipulation as quaternary trees. This data structure is important because it relaxes serious problems of locality and latency, and the tree helps to schedule multi-processing. Results here show how it facilitates algorithms that avoid cache misses and page faults at all levels in hierarchical memory, independently of a specific runtime environment.

We have built a rudimentary C-to-C translator that implements matrices in Morton-order from source that presumes a row-major implementation. Early performance from LAPACK's reference implementation of `dgesv` (linear solver), and all its supporting routines (including `dgemm` matrix-multiplication) form a successful research demonstration. Its performance predicts improvements from new algebra in back-end optimizers.

We also present results from a more stylish `dgemm` algorithm that takes better advantage of this representation. With only routine back-end optimizations inserted by hand (unfolding the base case and passing arguments in registers), we achieve machine performance exceeding that of the manufacturer-crafted `dgemm` running at 67% of peak flops. And the same code performs similarly on several machines.

Together, these results show how existing codes and future block-recursive algorithms can work well together on this matrix representation. Locality is key to future performance, and the new representation has a remarkable impact.

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, Languages, Performance

Keywords

Paging, quadtrees

1. INTRODUCTION

The original motivations underlying traditional row- and column-major representations of matrices are stale. In today's world of hierarchical memory and both homogeneous and heterogeneous parallelism, the seamless decomposition of data and locality of memory access are far more important than the dense use of address space. This work argues, therefore, for a different representation of matrices, in Morton or И order which addresses these needs. It also supports locally sparse matrices, blocks of varying (undulant) sizes, and compiler techniques for both iterative programming style (`for` loops over cartesian indices) and divide-and-conquer decomposition (recursion over Ahnentafel indices; see below.)

The new style also suggests a new view of familiar algorithms that decomposes blocks recursively—regardless of the sizes of packets, pages, cache lines, or register files—down to a size that suits any particular machine. Compilers also must change to handle the new indexing schemes to mimic, for instance, the strength reduction that the first FORTRAN compiler provided on row indexing [3]. Programmers, designers of languages/compilers, and analysis of algorithms once ignored the relative locality and latency among memory addresses because all were equally slow and power-hungry [24]; now all must strive to reuse data already in cache.

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

†Supported, in part, by the National Science Foundation under grants numbered CCR–9711269 and CCR–0073491.

‡Supported, in part, by the National Science Foundation under a grant numbered CCR–9711269.

§Work performed, in part, at Indiana University and supported, in part, by the U.S. Department of Education under a grant numbered P200A50237.

¶Work performed at Indiana University and supported by the National Science Foundation under a grant numbered CCR–9711269.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPOPP'01, June 18-20, 2001, Snowbird, Utah, USA.
Copyright 2001 ACM 1-58113-346-4/01/0006 ..$5.00

One way to take advantage of the latter-day architectures is to build over a library of hand-coded basic routines as an artificial base language. This is the approach of LAPACK and BLAS [12], but this tack contravenes a primary goal of programming-language (PL) research. PL history asks for a high-level language, which can express an algorithm in machine-independent style that the compiler casts to efficient code to suit whatever the run-time environments may be. And now we must think in terms of *multiple* environments for one program because running code is distributed to different machines. This work is offered as a likely foundation to elevate PL practice to that former standard.

The following results explore a different representation of arrays—especially matrices—that offers efficiency to many extant source languages and styles and a way to translate them onto architectures of today and tomorrow. Our results validate that promise. More importantly, they support a stronger style of programming

This paper revisits data structures (specifically, Morton-order arrays and dilated integers), programming style and algorithms (recursive descent and divide-and-conquer blocking), and architectural constraints (locality of memory access) applied to large matrix problems. We have observed excellent patterns of memory access which are critical to shared, hierarchical memories [25, 14, 27, 10]. Leiserson calls this kind of behavior "cache-oblivious" because the programmer can relax her awareness of the behavior between various levels of the memory hierarchy: L1, L2 cache, main RAM, swapping disk, and even the Internet [17]. Perhaps a better term is "cache-conscious" because the resulting algorithms reflect an abstract inclination toward immediate reuse, once data is local to an active process.

Section 2 of this paper reviews definitions relating to Morton-order indexing and the algebra of dilated integers. Section 3 describes a prototype compiler built to compile C code to this representation and an iterative style to take advantage of it. The fourth section describes the testbeds used for our experiments, and then presents times for generic LAPACK `dgesv` compiled to Morton-order representation with our compiler; little performance is lost compared to directly compiled C. Section 5 presents more extensive timing results specifically on matrix multiplication, showing how future compilers will improve recursive code to achieve the locality to compete even with manufacturer's hand-coded BLAS3 codes [12]. Section 6 reviews related work and offers conclusions.

2. DEFINITIONS

All these definitions become easier with Figures 2, 3, 4 [35, 36]. In the following m $= 2^d$ is the degree of the tree appropriate to dimension d.

DEFINITION 1. *The base 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* $mi + 0, mi + 1, \ldots, mi + (m - 1)$ [35].

By convention, with a matrix in Ⅳ order its four submatrices are oriented northwest, southwest, northeast, and southeast, respectively.

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 level-order indices* $mi + 1, mi + 2, \ldots, mi + m$ [35].

DEFINITION 3. *A complete array has* Ahnentafel *index* $m - 1$. *A subarray (block) at Ahnentafel index i is either a scalar, or it is composed of m subarrays, with indices* $mi + 0, mi + 1, \ldots, mi + (m - 1)$ [35].

All the definitions share the property, illustrated at the right of Figure 1, that the nested blocks of a matrix of size 4^p are accessed by 4^p consecutive indices for all p. Their elements, therefore, have high locality to one another, inversely with p. Figure 5 shows why Definitions 2 and 3 for binary trees [9] are often blurred [22, p. 401]. Henceforth, assume $m = 4$ for matrices.

The conversions among the three indexing schemes defined above depend only on constants that are identified in the figures down the left spine of the tree.

NOTATION 1. *Let* w *be the number of bits in a* `short`.

NOTATION 2. *Each* $\mathrm{q_k}$ *is a modulo-4 digit or* quat. *Alternatively, each* q_k *can be expressed as* $q_k = i_k + 2j_k$ *where* $\mathrm{i_k}$ *and* $\mathrm{j_k}$ *are* bits.

In Figures 2 and 4 the cartesian indices of the leaves appear in outlined font below the tree. Morton and Ahnentafel indices (and dilated integers, below) have $2w$ bits; cartesian indices have w bits. The Morton index $\sum_{k=0}^{w-1} q_k 4^k = \sum_{k=0}^{w-1} i_k 4^k + 2\sum_{k=0}^{w-1} j_k 4^k$ corresponds to the cartesian indices: row $\sum_{k=0}^{w-1} i_k 2^k$ and column $\sum_{k=0}^{w-1} j_k 2^k$. The quats, read in order of descending subscripts, select a path from the root to the node, as in Figure 4.

The set of bits $\{i_k\}$ are the even-numbered bits in the Morton index, and the $\{j_k\}$ are the odd-numbered bits. This is Morton's bit interleaving of cartesian indices [26]. Conversion algorithms between Morton (similarly, level-order or Ahnentafel) indexing and cartesian indices were known from its beginning; code to convert from cartesian indices to a Morton index by shuffling bits, or the inverse conversion that deals out the bits, can be slow: logarithmic in w or requiring table look-up.

Masking a Morton index with `0x55555555` or `0xaaaaaaaa` extracts the bits of the row and column cartesian indices, introduced next as *dilated integers*. In C code `0x55555555` is an important constant available as `((unsigned int)-1)/3)`.

NOTATION 3. *The integer* $\overrightarrow{\mathrm{b}} = \sum_{k=0}^{w-1} 4^k$ *is called* `evenBits` *in C, and is* `0x55555555`. *Similarly,* $\overleftarrow{\mathrm{b}} = 2\overrightarrow{b}$ *is called* `oddBits`, `0xaaaaaaaa`.

It is remarkable how often these basic properties of Morton ordering have been reintroduced in different contexts [29, 32, 26, 6, 18, 28, 33, 16]. Samet gives an excellent history [30].

The additive algebra of dilated integers, itself, is surprisingly old [32, 31, 35]. The trick is to represent all cartesian indices as dilated integers (with information stored only in every other bit), and to use only register operations on them—like ordinary integers.

DEFINITION 4. *The* even-dilated representation *of* $i = \sum_{k=0}^{w-1} i_k 2^k$ *is* $\sum_{k=0}^{w-1} i_k 4^k$, *denoted* $\overrightarrow{\imath}$.
The odd-dilated representation *of* $j = \sum_{k=0}^{w-1} j_k 2^k$ *is* $2\overrightarrow{\jmath}$ *and is denoted* $\overleftarrow{\jmath}$ [35].

The arrows suggest the justification of the meaningful bits in either dilated representation. For example, the right arrow suggests rightmost Bit 0 and its even kin.

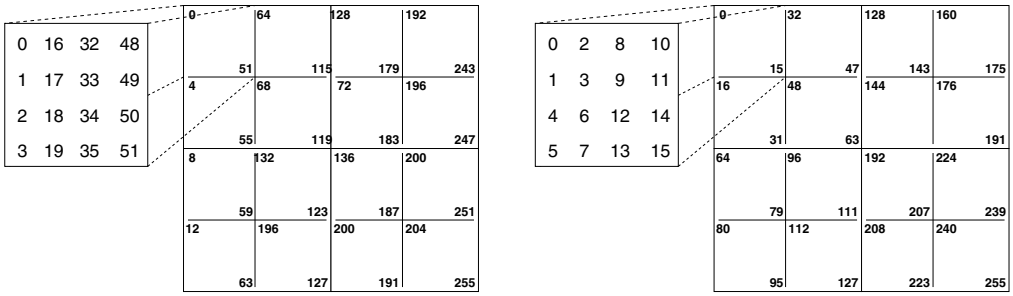

Figure 1: Column-major indexing of a 16×16 matrix, and analogous Morton indexing.

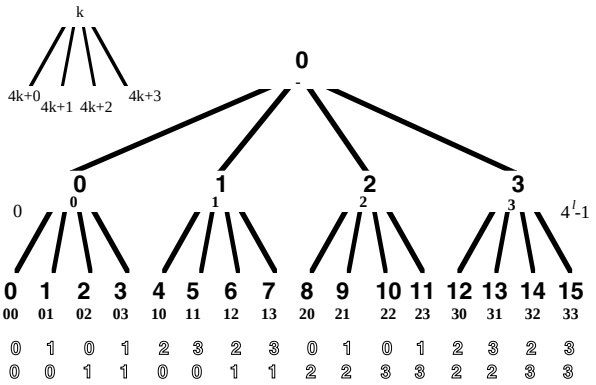

Figure 2: Morton indexing of the order-4 quadtree [35].

THEOREM 1. *A matrix of m rows and n columns is allocated as a sequential block of $\overrightarrow{m-1} + \overleftarrow{n-1} + 1$ scalar addresses* [35].

One might wonder that this can be almost thrice the *address* space as needed for a square, column-major matrix, but not all that address space will be active. Undefined data at idle addresses remains resident only in cheaper, lower levels of the memory hierarchy. Only data near active addresses ever migrates into precious cache.

THEOREM 2. *With "$\equiv$" read as semantic equivalence and with "==" denoting equality on integer representations, then for unsigned integers* [31]

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

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

THEOREM 3. *The Morton index for the $\langle i, j \rangle^{\text{th}}$ element of a matrix is $\overrightarrow{i} \vee \overleftarrow{j}$, or $\overrightarrow{i} + \overleftarrow{j}$* [35].

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

DEFINITION 5. *Addition ($\overrightarrow{+}, \overleftarrow{+}$) and subtraction ($\overrightarrow{-}, \overleftarrow{-}$) of dilated integers* [35]*:*

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

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

THEOREM 4. *Register-local implementations of subtraction, addition, constant addition, and constant shifts of twos-complement dilated integers* [35]*:*

$$\overrightarrow{i} \overrightarrow{-} \overrightarrow{n} = (\overrightarrow{i} - \overrightarrow{n}) \wedge \overrightarrow{b}; \qquad [31]$$

$$\overleftarrow{j} \overleftarrow{-} \overleftarrow{n} = (\overleftarrow{j} - \overleftarrow{n}) \wedge \overleftarrow{b}; \qquad [31]$$

$$\overrightarrow{i} \overrightarrow{+} \overrightarrow{n} = (\overrightarrow{i} + \overleftarrow{b} + \overrightarrow{n}) \wedge \overrightarrow{b}; \qquad [31]$$

$$\overleftarrow{j} \overleftarrow{+} \overleftarrow{n} = (\overleftarrow{j} + \overrightarrow{b} + \overleftarrow{n}) \wedge \overleftarrow{b}; \qquad [31]$$

$$\overrightarrow{i} \overrightarrow{+} \overrightarrow{c} = \overrightarrow{i} \overrightarrow{-} \overrightarrow{(-c)}; \qquad \overleftarrow{j} \overleftarrow{+} \overleftarrow{c} = \overleftarrow{j} \overleftarrow{-} \overleftarrow{(-c)};$$

$$\overrightarrow{b} = \overrightarrow{-1}; \qquad \overleftarrow{b} = \overleftarrow{-1};$$

$$\overrightarrow{i \ll k} = \overrightarrow{i} \ll (2k); \qquad \overleftarrow{j \ll k} = \overleftarrow{j} \ll (2k);$$

$$\overrightarrow{i \gg k} = \overrightarrow{i} \gg (2k); \qquad \overleftarrow{j \gg k} = \overleftarrow{j} \gg (2k).$$

Taken together, the algorithms in this theorem become macros in C and C++, and for a class of dilated integers either methods in JAVA or operators in HASKELL.

3. THE COMPILER

Our compiler is motivated by the suggestion from Theorems 4 and 2 that the C loop

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

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

```
for (int ii=0; ii<nn; ii=(ii-evenBits)&evenBits ){...}
```

It is built using the front end of LCC [15], followed by the transformations described here, implemented in `Scheme`, and

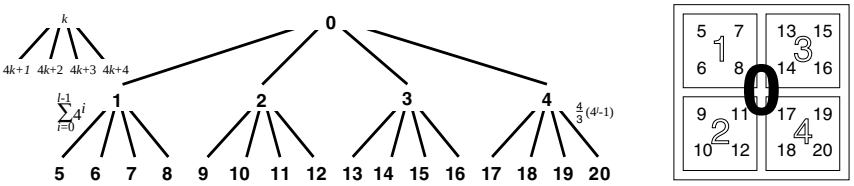

Figure 3: Level-order indexing of the order-4 quadtree and its submatrices [35].

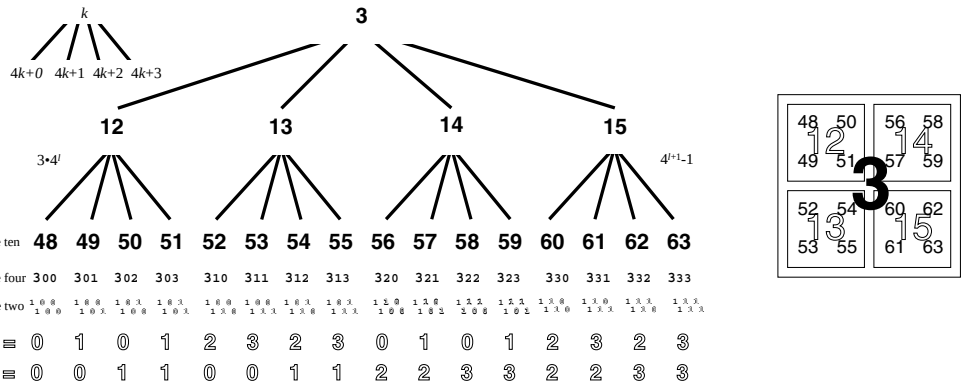

Figure 4: Ahnentafel indexing of the order-4 quadtree.

---

the output is C code that is fed back through the manufacturer's C compiler, targeted to the respective hosts.

The first step in the transformation is to identify all variables that are two-dimensional matrices. Since these often appear in C modules only as parameters to functions, we introduce a comment-like pragma whose only purpose is to identify a parameter that is a reference to a `double` as two-dimensional and to identify the `int` parameter that is its stride in address computations. The pragma appears as a flagged comment; for example:

```
/*[] double : c[rows][cols];
     double : a[rows][p];
     double : b[p][cols]; */
void matrixMultiply (double* c, double* a, double* b,
                     int rows, int cols, int p) {...}
```

Such a pragma would not be necessary in a strongly typed language, and is not even necessary in C when the matrix is declared using square brackets within the same module.

Then the integer variables that index into each matrix are identified, and they are each shadowed with a new dilated integer. If an index is only used as a column index, then its dilation is odd; otherwise it is even-dilated. If used in both roles, then doubling gives the odd-dilation as needed.

Strength reduction is applied to multiplications that occur within loops (incrementing one factor) [2], so that each becomes a subtraction via Theorem 4. If none of the operations on indices that remain are multiplicative (or inaccessible functions), then the translation will be successful. Each operation on the original integer is coupled, also, with the analogous operation on its dilated shadow. Because such computations are already interleaved with memory access, the extra, processor-local cycles are cheap.

The rich opportunities for bounds checking with Morton indexes are described elsewhere [35]. We do not yet convert comparisons on the underlying integers to comparisons on their dilated shadows, as suggested in Theorem 2 because of parsing difficulties with conditional expressions. With a new parser that allows this translation, also, we hope to excise many source-code `int`s from their reamining roles in flow control. Then we could remove them entirely, leaving only the shadow/dilated integers in their place.

So, after i and j are translated by the compiler to their images, $\overrightarrow{i}$ and $\overleftarrow{j}$, the resulting object code can be just the simple translation that the programmer expects. Code like the following C source will result via our helpful compiler from `for` loops on ordinary `int`s.

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

The next step is to perform the usual loop unrolling to fill an instruction pipe with straight-line code. Without knowledge of the algebra on dilated integers, no C complier now can take this step from the code above, as ours will. We have, however, simulated this step on blocked algorithms with hand expansion of the innermost loops. With that we show this algorithm can run faster on Morton-order matrices, compared to the source C codes on row-major matrices (*cf.* InProd8 in Section 5).

The experience with the prototype compiler teaches us lessons both on how to compile and on how to program with this representation.

- For better locality, blocked algorithms should use square blocks of size 4^p at cartesian indices that are multiples of 2^p.

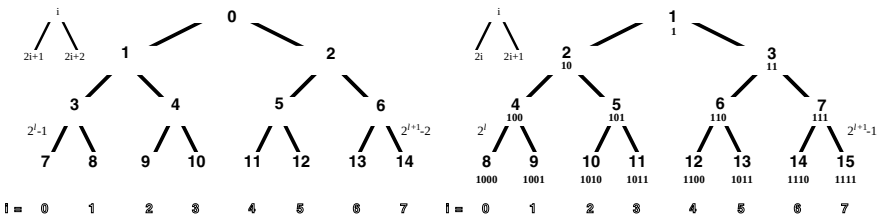

Figure 5: Level-order vs. Ahnentafel indexing of a binary tree.

- Column indices should be compiled to their odd-dilated representation. Row indices should be even-dilated.
- Similarly, row and column bounds should be precomputed as appropriately dilated constants.
- In complicated codes, it is possible to represent one value as both dilated and undilated. When the increments are embedded in a barrage of memory accesses, it requires another register but not more time.

4. EXPERIMENTS: CARTESIAN CODE

Results from three contemporary machines are presented here; in all cases we use the manufacturer's C compiler with full optimization.

- Sun Enterprise 450 with 400 MHz UltraSPARC-II, 16kB on-chip instruction cache, 16kB on-chip data cache, and 4MB secondary cache, 1GB RAM (shared). Sun WS 5.0 compiler.
  The Sun codes did not perform as well as SGI's, relative only to the respective processors' peak flops.
- SGI Octane, 195MHz R10000 ip30 processor with 128 MB main memory, 32 kB instruction cache, 32kB data cache. Secondary unified instruction/data cache of 1 MB. Compiler MIPSPro 7.3.1.1m with optimization `-Ofast -64 -OPT:alias=restrict`.
  This machine has effective L1 and L2 caching, and RAM is also so small that even paging affects its performance on larger problems.
- SGI PowerChallenge, 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 MIPSPro 7.3.1.1m with optimization `-Ofast -64 -OPT:alias=restrict`.
  With no L1 cache for floats, only L2 matters. Moreover, RAM is so huge that paging never happened.

The first example is running output from our compiler from the LAPACK's reference C codes for `dgesv` and its supporting routines: `dgemm`, `dger`, `dgetf2`, `dgetrf`, `dgetrs`, `dlaswp`, `dscal`, `dswap`, `dtrsm`, `idamax`, `ilaenv`, `lsame`, and `xerbla`. It illustrates two things: first, it is evidence of early success from the prototype compiler. The only amendments to the published source code are the pragmas to identify the strides, illustrated above. Second, it illustrates performance from the style described in Section 3 for iterative code on Morton-order matrices.

Figure 6 shows the performance of the translation that implements cartesian indexing on Morton-indexed matrices using the macros from Section 2. On the left, one can see that the code compiled from simple source performs quite closely to the manufacturer's version, but that the transliteration to Morton order performs badly because the C compiler does not know the algebra of dilated integers. But (on the right) when the loops are unrolled by hand to 8×8 blocking, the performance on Morton order improves markedly, demonstrating that smooth performance *is* available there, after the compiler gets some help unrolling those loops. Likewise, hand unrolling of clean, column-major loops degrades performance of a tuned compiler.

With both plots so close (in the right graph), it is fair to expect them both to improve if the compiler were handling the blocking and unrolling, instead of being constrained by hand-written source. So, we look forward to even better performance from our compiler when it can unroll loops to take advantage of locality, as illustrated here.

5. A MORE APPROPRIATE EXAMPLE

The second example is matrix multiplication, represented here by the code from Section 3 and by that in Figure 11 [16]. That figure also illustrates how a block recursion can be used to identify non-interfering, balanced parallel processes; extra braces partition the eight recursions into two or four sequences that can be dispatched together. And it illustrates a convenient style that exposes cache reuse, as the comments there indicate.

There are significant differences among these codes. The C code in the previous section was written for row-major matrices (originally column-major in FORTRAN). and compiled to Morton-order. The examples in this section are all hand-written C, two for Morton-order representation in different programming styles, Yet, both these codes can be loaded into a single program because the matrix representation is common.

We present the results of a source-to-source synthesis using Morton order and only two compiling techniques: unfolding of base cases (with rerolling) [5], and strength reduction on the computation and bounding of indices [2]. Machine-specific parameters, like cache size, were never used, because effects from the C optimizer were more important.

The Figure-11 algorithm is implemented using Ahnentafel indices to control the recursion. The `offset` is set to the difference between Ahnentafel and Morton indices and presubtracted from all matrix references. Bounds checking on Ahnentafel indices uses a precomputed bounds table, logarithmic in the size of the matrix [35, §3.2]. The arguments to `up_mult` and `dn_mult` are passed in local registers in order to avoid stack use. Moreover, the base case is unfolded from a 1×1 block to an 16×16 block in order to avoid excessive overhead from function calls and to take advantage of superscalar processing.

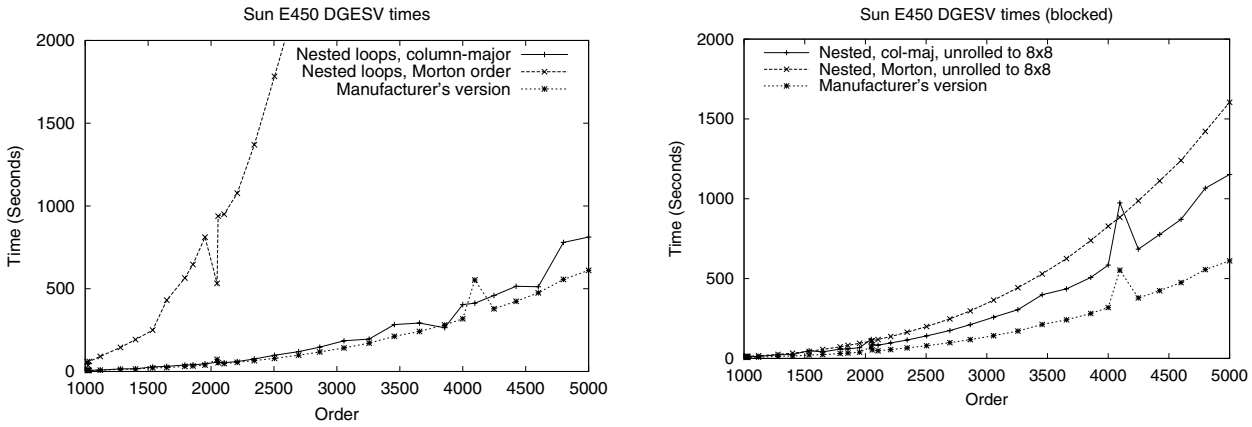

Figure 6: Uniprocessor running times for `dgesv` compiled to Morton-order matrices.

The matrices are square, of the order given on the x-axis. We have run extensive experiments on the Morton-order representation and the statistics grow smoothly. Morton order has no hiccups due to striding, for instance, because the blocking is inherently sequential in memory.

The four algorithms are

INPROD — The conventional three-nested-loop inner-product matrix multiplication on column-major matrices, similar to that in Section 3. The stride is an odd number greater than the number of rows.

INPROD16 — This is the first test on Morton-order matrices. The algorithm is essentially that in Section 3, but it has been blocked to an 16×16 "element." That is, the loops increment in steps of $\overrightarrow{16}$ or $\overleftarrow{16}$, and the body of the inner loop is an 16×16 matrix multipication, expressed as $16 \times 2 \times 128$ in-line multiply-adds. Compilers on ordinary `for` loops would provide unrolling; with the dilated loop controls we synthesized ours by hand, including folding two nested loops, incrementing $\overrightarrow{\texttt{i}}$ evenly and $\overleftarrow{\texttt{j}}$ oddly, into a single loop simply incrementing ($\overrightarrow{\texttt{i}}+\overleftarrow{\texttt{j}}$) and so getting the optimizer's attention.

QUADTREE16 — Figure 11 with the base case unfolded four times (16×16) and then rerolled just like INPROD16's.

BLAS3 — The manufacturer's `dgemm` library routine. SGI's achieve close to maximal flops.

Figures 7 to 9 plot the running times of the four algorithms, as well as the ratios of each to the cube of the order of the matrices. (That is, the constants of proportionality of the $O(n^3)$ algorithms are also plotted.) As a demonstration of the impact of the Morton-order structure alone, *the codes are identical across all three machines.*

We have also tracked the count of major page faults from UNIX's `getrusage` and, on the R10000, TLB misses. They do not affect the large-RAM machines, but paging is dramatic on the SGI R10000, where the timings, even of BLAS3, suddenly exhibit a huge leap as disk thrashing takes over, thrashing from which the QUADTREE16 algorithm is remarkably immune, and which even the InProd16 avoids for a while.

Both the BLAS3 and the INPROD code on the R10000 exhibit some sensitivity to matrix striding (Figure 8). This is not visible with BLAS3 on the R8000. The Morton-order representation does not exhibit this problem because its blocking does not "stride" through the array. Any quadtree 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.

Surprisingly, the INPROD16 code using Morton order was very fast; close to QUADTREE16 on the E450 but its performance, even for BLAS3, is far from the machine's capacity. On the SGI machines, however, the timings do approach capacity, illustrated by the dotted line; QUADTREE16 does *very* well, indeed. Based on the observation that BLAS3 was hand tuned to run at machine capacity, we observe that this much performance for so little of our effort is already a strong endorsement of Morton order. That is, we can expect other algorithms coded in C to achieve this performance with Morton order and similar style.

The QUADTREE16 algorithm beats BLAS3. On the SGI R8000 it runs 9% under BLAS3, quite an improvement over the 600% excess of [16]. The improvement can be attributed to the unfolding, to the rerolling, and finally to optimization by the compiler.

The most surprising result is the relative performance of both INPROD16 and QUADTREE16 on the R10000, which is cache-rich. Noticing its small-machine context, we observed a 32-fold improvement from the latter at order 3050, away from a power-of-two. The wretched performance of BLAS3 (and column-major INPROD) on the R10000 is attributed to the memory hierarchy: first caching and then paging of the small main memory. Overall, the algorithms using Morton order beat BLAS3 in this `dgemm` race.

Finally, the parallel-dispatch code on the PowerChallenge is illustrated in Figure 10, showing little degrading in the behavior of either BLAS or QUADTREE8 [16]. But this is old code [36], only showing how it tracks BLAS3.

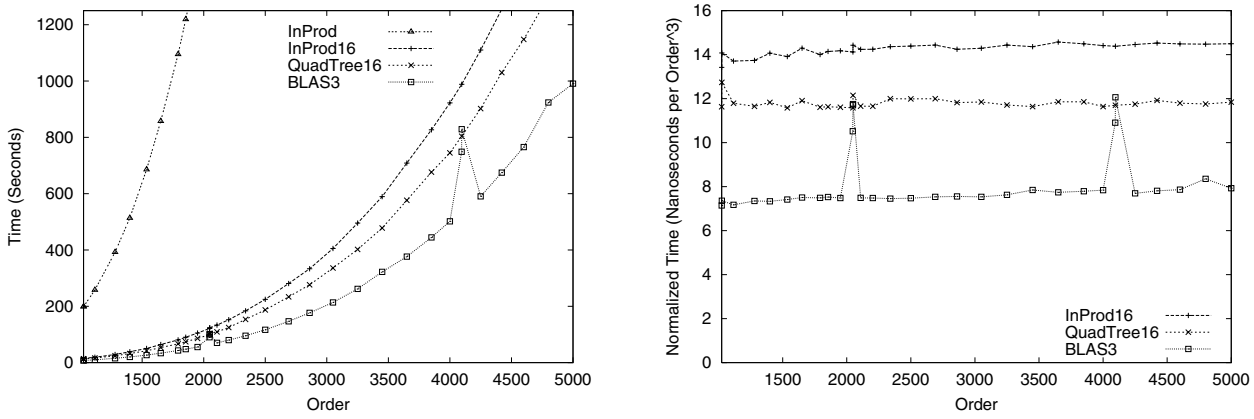

Figure 7: Uniprocessor running times (seconds) and ratios for four algorithms on Sun E450.

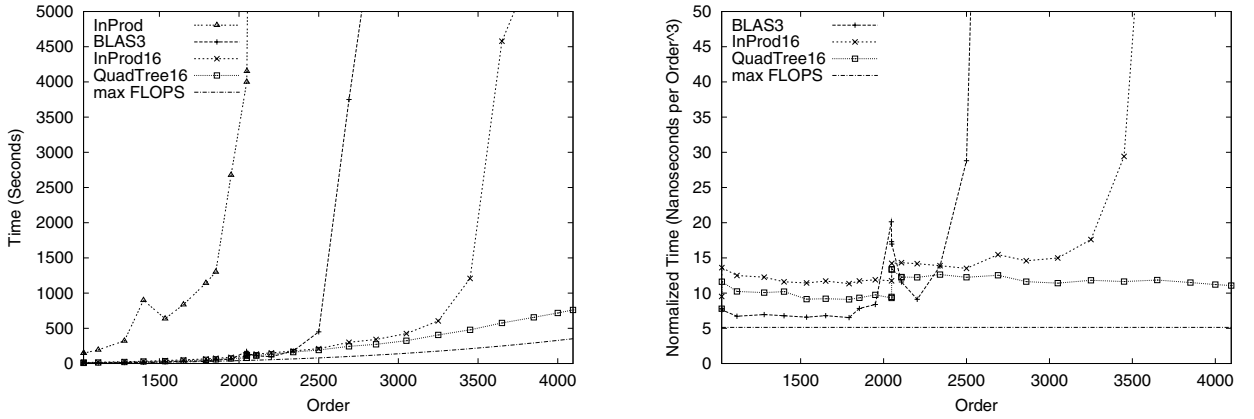

Figure 8: Uniprocessor running times (seconds) and ratios for four algorithms on SGI R10000.

6. RELATED WORK & CONCLUSIONS

Much recent work by others lies at the perimeter of this project. But they do not place Morton ordering at the center of algorithm development. Chatterjee has experimented with the Figure-11 algorithm, but he adopted a hybrid representation tailored to BLAS3 `dgemm` that uses Morton-order at higher levels in the tree, and column-major for the base-case blocks [8, 7]. This yields a complicated structure for anything other than this algorithm, or incurs an observed overhead to move data to and from it. His results do not seem useful in the context of real applications.

Also close to our approach to `dgemm` is the work on sparse matrices of Im and Yelick [21]. It lands somewhere between a compiler and a library, where the blocking is selected according to the register capacity of the target machine. They appear to deal only with locality at one level of the hierarchy, although it is inappropriate to go much further for sparse matrices. Related work in the PHiPAC project does address multiple levels of blocking, but only the register file and L1 cache are targeted [4]. That work generalizes to multiple levels, but it does not presume the square, power-of-two restrictions of Morton order and, thus, does not enjoy the advantages of bounds checking and dilated indices.

Gustavson's work does address the halving of quadtrees, but he does not enforce powers of two [20, 13]. Rather, he cleaves matrices into roughly equal quarters, obtaining balanced subproblems but, again, not the advantages of dilated integers and bounds checking.

Ahmed and Pingali explicitly address cache reuse with data shackles [23] but they are not constrained beyond a single level. Moreover, the constraints are related more to control rather than to representation. A space-filling Hilbert curve is used [1] but its indexing is not monotonic in rows or columns, and so is not useful for control, specifically for bounds checking. It seems to be used not for representation, but rather for control of an iterative traversal, which would compare to whatever order results from linearizing, say, a pattern of recursive traversal from Figure 11.

The ATLAS project is aimed at compiling optimal block sizes into classic programming style [34], so that blocks nicely fill cache, but it only addresses a single level of caching. L2 cache or page reuse is not addressed.

Ding and Kennedy address the problem of bandwidth through the memory by measuring bandwidth at many levels in the hierarchy and then introducing loop transformations to ameliorate the performance [11]. No performance is offered. Related work addresses the automatic introduction of block recursive code [37], which might help schedule processes, but none of this work changes the underlying representation of matrices.

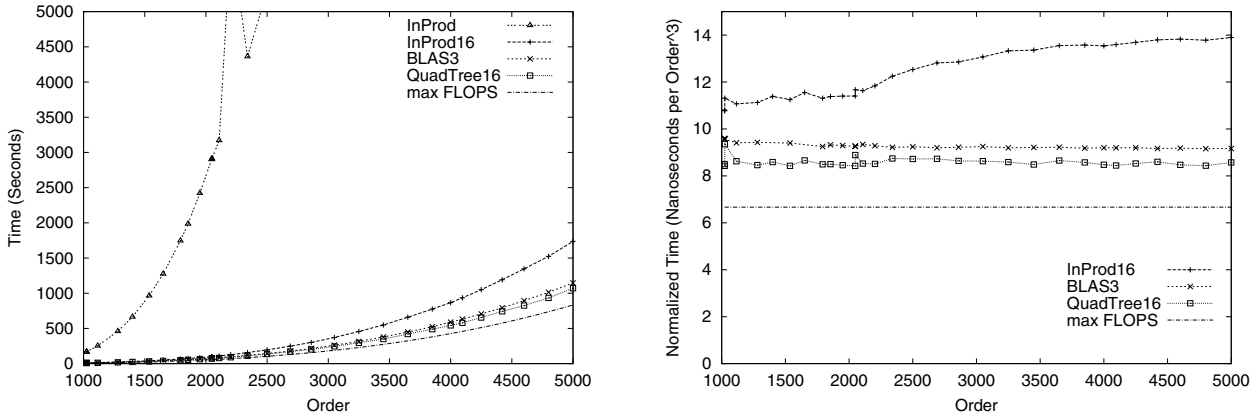

Figure 9: Uniprocessor running times (seconds) and ratios for four algorithms on SGI R8000.

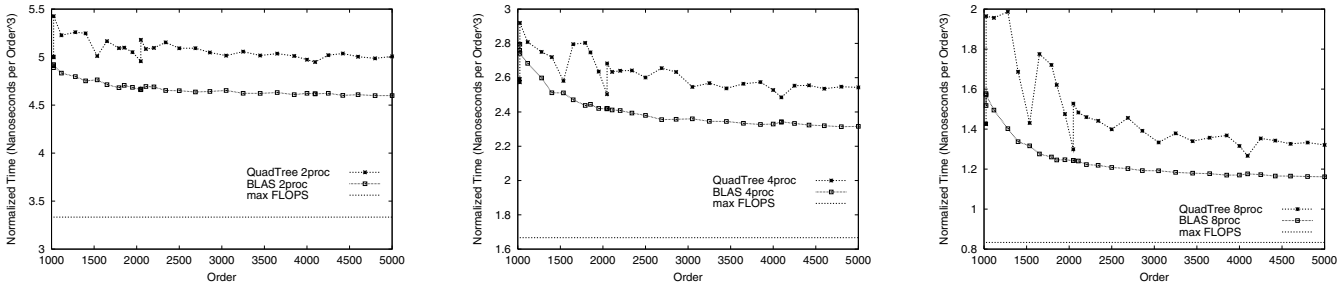

Figure 10: Multiprocessor ratios for four algorithms on 2, 4, 8 processors on SGI R8000.

Finally, there is significantly more work on scientific computation available in the HASKELL community. An example is Grant's rendering of Cholesky factorization [19]. Those codes, all free of synchronizing side-effects, ought to be revisited now with Morton-order matrices represening the array comprehensions. Without it, HASKELL performance has been far too slow.

Changes in architectures expose unnecessary assumptions in classic compiling techniques, which will change only after both of the following are realized: some alternative is offered which promises better performance for future languages and architectures, *and* a migration path is offered from the current style into the alternative that will sustain performance until the present programming style is displaced and the promises can be realized via new compilers. Acknowledging both these requirements, we here present a proof-of-concept and are working toward a research demonstration for the representation of matrices in Morton order.

Several hurdles must be cleared before programmers will use Morton-order matrices with the same facility as row- or column-major representations. The first is to build compilers that dilate cartesian indices to yield Morton-order indexing, so that existing codes can be used alongside future block-recursive algorithms; we describe a prototype. It must be improved to unroll loops over dilated indices.

The long-term goal is to offer the programmer a comfortable context for Ahnentafel indexing: syntax and compilers that optimize it well. A major problem is to generate inline, superscalar code from the bases of a recursion; future compilers must combine unfolding and instruction scheduling on Morton indices as effectively as present ones unroll and schedule loops on cartesian indices.

These results are part of an effort to attain locality in running code compiled from high-level algorithms, to deliver that performance to multi-threaded and multi-processing run-time environments, to develop a new and artful programming style, and finally to discover new and pretty algorithms with it.

7. REFERENCES

[1] N. Ahmed and K. Pingali. Automatic generation of block-recursive codes. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *EURO-PAR 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 368–378, Heidelberg, 2000. Springer.
http://link.springer.de/link/service/series/0558/bibs/1900/19000368.htm

[2] F. E. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. W. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 3.2, pages 79–101. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] J. Backus. The history of FORTRAN i, ii, and iii. In R. L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, New York, 1981. Also preprinted in *SIGPLAN Not.*, 13(8):166–180, Aug. 1978.

[4] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a protable, high-performance, ansi c coding methodology. In *Proc. 1977 Intl. Conf. on Supercomputing*, pages 340–347, New York, July 1997. ACM Press.
http://www.acm.org/pubs/citations/proceedings/supercomputing/263580/p340-bilmes/

[5] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J.ACM*, 24(1):44–67, Jan. 1977.
http://www.acm.org/pubs/citations/journals/jacm/1977-24-1/p44-burstall/

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

[7] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proc. 1999 Intl. Conf. on Supercomputing*, pages 444–453, New York, June 1999. ACM Press.
http://www.acm.org/pubs/citations/proceedings/supercomputing/305138/p444-chatterjee/

[8] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottenthodi. Recursive array layouts and fast parallel matrix multiplication. In *Proc. 11th ACM Symp. Parallel Algorithms and Architectures*, pages 222–231, New York, June 1999. ACM Press.
http://www.acm.org/pubs/citations/proceedings/spaa/305619/p222-chatterjee/

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

[10] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: a practical model of parallel computation. *Commun. ACM*, 39(11):78–85, Nov. 1996.
http://www.acm.org/pubs/citations/journals/cacm/1996-39-11/p78-culler/

[11] C. Ding and K. Kennedy. The memory bandwidth bottleneck and its amelioration by a compiler. In *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, Los Alamitos, CA, 2000. IEEE Computer Society Press.
http://www.computer.org/proceedings/ipdps/0574/05740181abs.htm

[12] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, Mar. 1988. http://www.acm.org/pubs/citations/journals/toms/1988-14-1/p1-dongarra/

[13] E. Elmroth and F. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Develop.*, 44(4):605–624, July 2000.
http://www.research.ibm.com/journal/rd/444/elmroth.html

[14] P. C. Fischer and R. L. Probert. Storage reorganization techniques for matrix computation in a paging environment. *Commun. ACM*, 22(7):405–415, July 1979.
http://www.acm.org/pubs/citations/journals/cacm/1979-22-7/p405-fischer/

[15] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler : Design and Implementation.* Benjamin/Cummings, Redwood City, CA, 1995.

[16] J. D. Frens and D. S. Wise. Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code. *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.*, 32(7):206–216, July 1997.
http://www.acm.org/pubs/citations/proceedings/ppopp/263764/p206-frens/

[17] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Ann. Symp. Foundations of Computer Science*, session 6B. IEEE Computer Society, Los Alamitos, CA, 1999. http://www.computer.org/proceedings/focs/0409/04090285abs.htm

[18] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, Dec. 1982.
http://www.acm.org/pubs/citations/journals/cacm/1982-25-12/p905-gargantini/

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

[20] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Develop.*, 41(6):737–755, Nov. 1997.
http://www.research.ibm.com/journal/rd/416/gustavson.html

[21] E.-J. Im and K. Yelick. Optimizing sparse matrix vector multimplication on SMPs. In *9th SIAM Conf. on Parallel Processing for Scientific Computing*, volume 98 of *Proc. in Applied Mathematics*, Mar. 1999.
http://www.siam.org/catalog/mcc07/pr98.htm

[22] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, third edition, 1997.

[23] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. *Proc. ACM SIGPLAN '97 Conf. on Program. Language Design and Implementation, SIGPLAN Not.*, 32(7):346–357, May 1997.
http://www.acm.org/pubs/citations/proceedings/pldi/258915/p346-kodukula/

[24] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power-aware page allocation. *Proc. 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Not.*, 35(11):105–116, Nov. 2000.
http://www.acm.org/pubs/citations/proceedings/asplos/356988/p105-lebeck/

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

[26] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Ontario, Mar. 1, 1966.

[27] G. Newman. Organizing arrays for paged memory systems. *Commun. ACM*, 38(7):93–103 + 108–110, July 1995.
http://www.acm.org/pubs/citations/journals/cacm/1995-38-7/p93-newman/

[28] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. 3rd ACM SIGACT–SIGMOD Symp. on Principles of Database Systems*, pages 181–190, New York, 1984. ACM Press.

[29] G. Peano. Sur une courbe, qui remplit toute une aire plaine. *Math. Ann.*, 36:157–160, 1890.

[30] H. Samet. *The Design and Analysis of Spatial Data Structures*, section 2.7. Addison-Wesley, Reading, MA, 1990.

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

[32] K. D. Tocher. The application of automatic computers to sampling experiments. *J. Roy. Statist. Soc. Ser. B*, 16(1):39–61, 1954. See pp. 53–55.

[33] M. S. Warren. and J. K. Salmon. A parallel hashed oct-tree N-body problem. In *Proc. Supercomputing '93*, pages 12–21, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[34] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. Supercomputing '98*, Los Alamitos, CA, 1998. IEEE Computer Society.
http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Whaley814/INDEX.HTM

[35] D. S. Wise. Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 – Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 774–883, Heidelberg, 2000. Springer.
http://link.springer.de/link/service/series/0558/bibs/1900/19000774.htm

[36] D. S. Wise and J. D. Frens. Morton-order matrices deserve compilers' support. Technical Report 533, Computer Science Dept., Indiana University, Nov. 1999.
http://www.cs.indiana.edu/ftp/techreports/TR533.html

[37] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. *Proc. ACM SIGPLAN '00 Conf. on Program. Language Design and Implementation, SIGPLAN Not.*, 35(5):169–181, May 2000.
http://www.acm.org/pubs/citations/proceedings/pldi/301618a/p169-yi/

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

int offset;
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)
{
  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];

    /* Both assertions about cache refer to extreme corners of   */
    /* the named quadrant.                                       */
  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 11: The two-miss algorithm for quadtree matrix multiplication [16].