Fast Additions on Masked Integers*

Michael D. Adams and David S. Wise
Computer Science Dept., Indiana University
Bloomington, IN 47405–7104, USA.

Abstract: Suppose the bits of a computer word are partitioned into d disjoint sets, each of which is used to represent one of a d-tuple of cartesian indices into d-dimensional space. Then, regardless of the partition, simple group operations and comparisons can be implemented for each index on a conventional processor in a sequence of two or three register operations.

These indexings allow any blocked algorithm from linear algebra to use some non-standard matrix orderings that increase locality and enhance their performance. The underlying implementations were designed for alternating bit postitions to index Morton-ordered matrices, but they apply, as well, to any bit partitioning. A hybrid ordering of the elements of a matrix becomes possible, therefore, with row-/column-major ordering within cache-sized blocks and Morton ordering of those blocks, themselves. So, one can enjoy the temporal locality of nested blocks, as well as compiler optimizations on row- or column-major ordering in base blocks.

CCS Categories:
E.1 [Data Structures]: Arrays; **D.1.0**[Programming Techniques]: General; **F.2.1**[Numerical Algorithms and Problems]: Computations on matrices; **E.2**[Data Storage Representations]: contiguous representations.
General Terms: Algorithms, Performance
Keywords: dilated integers, Morton order, quadtrees, compilers, index arithmetic

1 Introduction

Speed of access through the memory hierarchy (from registers through caches, to RAM, swapping disk, and communication with remote memories) constrains the performance of high-performance computing. Locality within such memory can be enhanced simply by partitioning the bits in a matrix index into a set X that indexes the columns, and Y that indexes the rows. This paper gives fast algorithms independent of X and Y to add, to subtract, and to compare such row/column indices. A free choice of partitioning allows the physical representation of the matrix to be easily rearranged, introducing new locality into old blocked algorithms that enhances their performance [6].

Let $m = \sum_{k=0}^{w-1} m_k 2^k$ be interpreted as a constant mask of a w-bit computer word; a k^{th} bit whose $m_k = 0$ is excluded, and a k^{th} bit whose $m_k = 1$ is included in the mask. The class of integers represented only by the included bits forms a group, whose additive methods and comparisons can be implemented within two or three cycles on any conventional processor. With excluded bits represented as 0 in the unsigned `value` of a represented object, the following C++ template shows the ideas behind these efficient methods. Assume that `T` is typed as `unsigned int` initially:

[*] Work supported, in part, by the National Science Foundation under grants numbered ACI-0219884 and EIA-0202048.
Copyright ©2006 Association for Computing Machinery, except for the the figures, still copyrighted by the authors. This is an author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *SIGPLAN Not.* **41**, 5 (May 2006), 39–45. `http://doi.acm.org/10.1145/1149982.1149987`

```
template<typename T, T mask>
class MaskedInteger{
private:
    T value;  // stored as normalized integer at  mask’s  1 bits.
public:
    // Constructor and Getter
    MaskedInteger(T val) : value(val & mask) {}
    T getVal() { return value; }

    // Simple additive operators.
    MaskedInteger operator-(MaskedInteger d) {
      return MaskedInteger((value           - d.value) & mask);
    }
    MaskedInteger operator+(MaskedInteger d) {
      return MaskedInteger((value + (~mask) + d.value) & mask);
    }
    MaskedInteger plus1()  {return MaskedInteger((value - mask) & mask);}
    MaskedInteger minus1() {return MaskedInteger((value -    1) & mask);}

    // Comparisons.  For example:
    bool operator<(MaskedInteger d) {
      return (value < d.value);     }

    ...
};
```

Type `T` can be any *unsigned* integer type: `char`, `short`, `int`, `long`, or `long long`.

Intuitively, subtraction of two values is direct because the borrows are propagated across excluded bits, and the final masking with m clears any that remain. Similarly, addition of two values is direct after adding the bitwise complement of m, setting the excluded bits in the addend so that carry bits are propagated, and again masked off. Increments and decrements use subtraction to avoid the extra cycle setting excluded bits. The former treats `mask` as the -1 of the signed class.

If the binary representation of m has b bits set, that is if $b = |\{i \mid m_i = 1\}|$, then an unsigned integer $i = \sum_{k=0}^{b-1} i_k 2^k$ can be represented and added in the bits selected by the constant mask m, allowing the other the $w - b$ bits to represent other values. Useful applications of these algorithms come from matrix representations, where i is a row index and the low-order of the remaining $w - b$ bits represent j, a column index. The indexing generalizes to higher dimensions by partitioning those remaining bits to represent more than one other index.

2 Applications

Focusing now on two-dimensions, we see that the mask `m` and its complement `~m` partition the w bits into two sets called Y and X, selecting the bits representing cartesian coordinates in the vertical (row index) and horizontal (column) orientations. For instance, consider a byte ($w = 8$) and its bits indexed high-to-low by the integers $7 \ldots 0$; then we might have $Y = \{5, 1, 0\}$ and $X = \{7, 6, 4, 3, 2\}$. We illustrate such patterns as $Y =$ `..*...**`$_2$ and $X =$ `**.***..`$_2$, where asterisks represent meaningful bits (one bits in the `mask`) and the periods are padding (zero bits in the `mask`.) Then a row index $i = 5$ could be represented as `..1...01`$_2$ and a column index $j = 17$ as `10.001..`$_2$, and both of them summed in one bite as `10100101`$_2$.

Schrack defines three kinds of padding at the dots, only one of which is used here [12]. If the padding within the `value` of a MaskedInteger is forced to zero bits (Dots become 0), then the representation is *normalized*; 5 is normalized in $Y =$ `00*000**`$_2$ to `00100001`$_2$. If the padding/dots were forced to one bits, then the representation is *antinormalized*; 5 is antinormalized in $X =$ `11*111**`$_2$ to `11111101`$_2$. *Unnormalized* representations have padding undefined, like the periods above. Everything below is normalized.

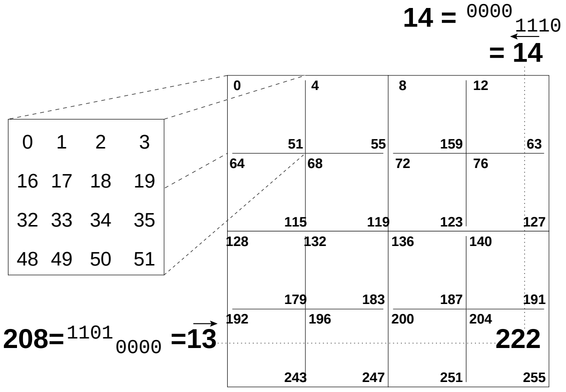

Figure 1: Row-major indexing of a 16×16 matrix. The integers in the boxes indicate row-major indices in base ten. The mask for row indices here is Y = `****....` and for column indices is X = `....****`.

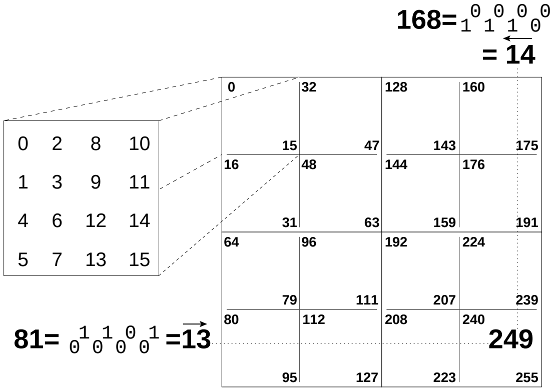

Figure 2: Morton-order indexing of a 16×16 matrix. The integers in the boxes again indicate Morton indices, base ten. The mask for row indices is Y = `.*.*.*.*` and for column indices is X = `*.*.*.*.`—its complement.

2.1 Row- and Column-Major Matrices

Row- and column-major orderings pack the elements of a matrix consecutively in memory using a raster order. Row major, as in C, indexes across rows consecutively, with each row following its predecessor. If there are s columns, then the $\langle i, j\rangle^{\text{th}}$ element has index $si + j$ in the matrix. The value of s is called the `stride.` Column major, in FORTRAN, indexes consecutively down columns, yielding matrix address $sj+i$ with s set to the number of rows.

Restricting the stride of a row- or column-major representation to be a power of two yields such a bit partitioning for the resulting matrix index. The multiplication by s becomes a shift of $\lg(s)$ bits. Row-major has Y as high-order bits and X as the low-order. (Conversely, column-major has X as high-order bits and Y as low-order.) For a $n \times 16$ matrix in row-major order, these complementary bit patterns would be implemented with the 4-byte C constants `0x0000000f` and `0xfffffff0`.

The example where $n = 16$, a 16×16 matrix, appears in Figure 1. It shows the indexed mapping to the $\langle 13, 14\rangle^{\text{th}}$ element, representing 13 in bits preshifted to the left. (The overline arrows in the figures suggest the implicit shifts of cartesian indices into their Y and X bit positions.) The anticipated products, that would shift i to high-order bits are exactly those long associated with strength reduction [1]. That is, representing those products as integers held in high-order bits is exactly the same as the sums from the additions that have always been introduced by optimizing compilers [2].

Fixed interleaving of hardware cache-replacement strategies are known to make power-of-2 strides especially poor when blocks exceed L1 cache capacity. For blocks that fit entirely within L1 cache, however, they have the attraction of very fast, repeated register transfer.

2.2 Morton-order Matrices

Morton ordering for matrices has the advantage of representing nested blocks in adjacent memory at all levels of the memory hierarchy [9, 13]. At run time this suffices for a single translation look-aside buffer (TLB) entry to span any block. Recursive programming on that nesting leads naturally to cache-oblivious algorithms [5]. Some of its desired locality is still available, however, simply by looping on cartesian indices into an array that is represented in Morton-order [6]. They would be represented as *dilated integers.*

Schrack introduces algorithms for direct arithmetic on dilated integers [12]. For matrices, X is a set of small, consecutive odd numbers and Y is a set of small consecutive even numbers (or vice versa). These masks are useful for both Morton and Ahnentafel indexing into matrices, whose even and odd bits identify, respectively the row and column cartesian indices [13]. This is the so-called bit interleaving to implement

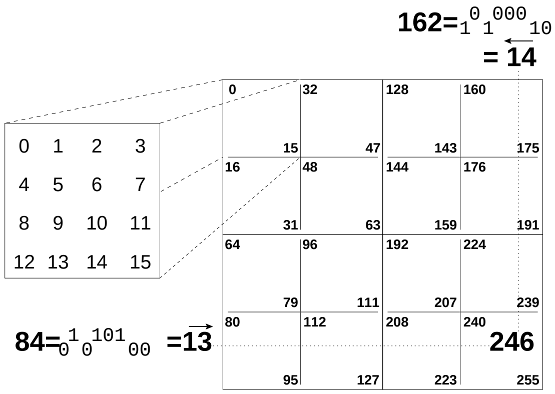

Figure 3: View this Morton-hybrid indexing of a 16×16 matrix as a 4×4 Morton-ordered matrix of blocks, each of which is 4×4 in row-major order. As before, the integers inside indicate Morton-hybrid indices in base ten.

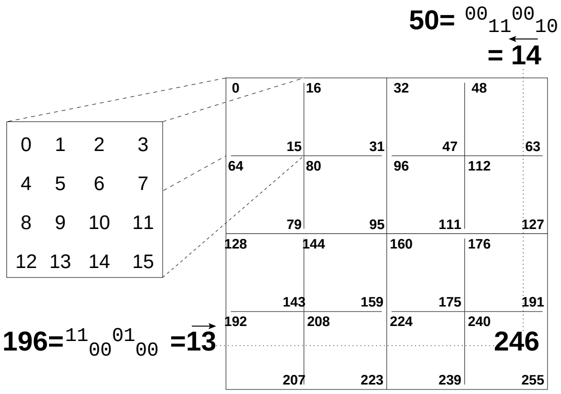

Figure 4: View this Major-major indexing of a 16×16 matrix as a 4×4 row-major matrix of blocks, each of which is 4×4 row-major. Integers inside again indicate Major-major indices in base ten.

cartesian indices.

More formally, if the row index is $i = \sum_{k=0}^{w/2-1} i_k 2^k$ and the column index is $j = \sum_{k=0}^{w/2-1} j_k 2^k$, then the corresponding address of $A_{i,j}$ is displaced by $\sum_{k=0}^{w/2-1}(i_k 4^k + 2j_k 4^k)$ from the base address A (in И order). Masking yields $2\sum_{k=0}^{w/2-1} j_k 4^k$ as an odd-dilated integer, and $\sum_{k=0}^{w/2-1} i_k 4^k$ as an even-dilated integer because they occupy the odd and even numbered bits of the word. For $w = 32$ these complementary bit patterns are generated by the C masks `0xaaaaaaaa` and `0x55555555`, respectively.[1] (Z order reverses even and odd.)

Figure 2 illustrates a 16×16 matrix in Morton order. Such matrices need not be square or have orders that are powers of two, although these features make this initial example easier. As this block-recursive И order extends across the plane, the dilation of integer 13 indexes the entire fourteenth row, and doubling the dilation of integer 14 indexes the whole fifteenth column.

2.3 Morton-hybrid Matrix Indexing

A hybrid of these two indexings represents matrices whose, say, 16×16 inner blocks are represented locally in row-major order, but all those blocks are represented, themselves, in Morton order. This *Morton hybrid* enables hardware and compilers to use existing optimizations derived for row- or column-major representations to deliver excellent performance in base blocks. Locality is yet available above L1 cache by using recursion on Morton-ordered blocks.

Chatterjee *et al.* created this hybrid ordering and used manufacturers' optimized BLAS3 libraries to seek high performance *and* locality [3, 4]. They did not, however, use the address arithmetic described here and also wasted computation converting their structure to and from a pure raster order.

For $w = 32$, one can represent $Y =$ `.*.*.*.*.*.*.*.*.*.*.*.*****....`$_2$, and
$X =$ `*.*.*.*.*.*.*.*.*.*.*.*.....****`$_2$ for И-ordered 16×16 blocks, each in row-major order. These complementary bit indexings are identified with the 32-bit C masks: `0xaaaaaa0f` and `0x555555f0`. While the order of the base block is not to be confused with w, nevertheless, in practice 32 and 16 have proven to be good choices for that order.

The resulting integer representations will be called *doppled integers* because the dilated-then-compressed bit patterns suggest the compression of waveforms from the Doppler effect on the sound of, say, a passing train.

Figure 3 illustrates a 16×16 matrix in Morton-hybrid order: as a 4×4 Morton-ordered matrix of blocks, each of which is 4×4 row-major. The mask for row indices is $Y =$ `.*.***..` and $X =$ `*.*...**`.

[1] Available in any unsigned twos-complement integer type, `T`, as `((T) -1)/3`.

for column indices. These determine the conversions of 13 and 14 to doppled integers indexing row and column, as shown.

2.4 Major-major Matrix Indexing

Another hybrid blends row-major and row-major (or column-major and column-major) in the same way. It can as easily be column-major/column-major or mixed as row-major/column-major, and vice versa. To cover the confusion of all four alternatives it is collectively called *major major* here [8].

The first alternative represents matrices whose $2^p \times 2^p$ inner blocks are represented locally in row-major order, and the array of those blocks is represented, as well, in row-major. It allows hardware and optimizing compilers to use vectorization in cache. It suits the common iterative practice of wrapping block-oriented iterations around others that only handle local blocks, with the advantage that those inner blocks are each stored at consecutive memory addresses—with a single TLB entry. Blocked versions of classic algorithms that implicitly favor row-major or column-major representation (*e.g.* Crout and Doolittle [7, p. 104]) will experience even more locality from these blocked representations.

For $w = 32$ and a global stride of 4096, one can represent $Y = \texttt{****************........****....}_2$, and $X = \texttt{................********....****}_2$ for row-major-ordered 16×16 blocks, each in row-major order. These complementary bit indexings are identified with the 32-bit C masks: `0xffff00f0` and `0x0000ff0f`, respectively.

Figure 4 illustrates a 16×16 matrix in major-major order: as a 4×4 row-major matrix of blocks, each of which is 4×4 row-major. Its mask for row indices is $Y = \texttt{**..**..}$ and $X = \texttt{..**..**}$. for column indices. These determine the conversions of 13 and 14 to blocked integers indexing row and column, as shown.

2.5 Declarations

So, after filling out the methods for MaskedInteger in Section 1, the following declarations *alone* suffice for declaring the classes for row and column indices for the three examples above.

```
typedef MaskedInteger<unsigned int, 0xfffffff0> Row16RowMajor;
typedef MaskedInteger<unsigned int, 0x0000000f> Col16RowMajor;

typedef MaskedInteger<unsigned int, 0x55555555> RowMorton2D; //even dilated
typedef MaskedInteger<unsigned int, 0xaaaaaaaa> ColMorton2D; // odd dilated

typedef MaskedInteger<unsigned int, 0x555555f0> Row16MortonHybrid;
typedef MaskedInteger<unsigned int, 0xaaaaaa0f> Col16MortonHybrid;

typedef MaskedInteger<unsigned int, 0xffff00f0> Row16MajorMajor;
typedef MaskedInteger<unsigned int, 0x0000ff0f> Col16MajorMajor;
```

For two dimensions, the paired masks are complementary; no additional masks are necessary at run time.

Three-dimensional Morton indexing is as easily available as three instantiations:

```
typedef MaskedInteger<unsigned short, 0x9249> RowMorton3D;
typedef MaskedInteger<unsigned short, 0x2492> ColMorton3D;
typedef MaskedInteger<unsigned short, 0x4924> RodMorton3D;
```

Full implementation of the last three uses six different masks, which are no longer pairwise complementary.

3 Beyond Indexing: Other Considerations

3.1 Small Observations

This note addresses arithmetic on integers represented within bits masked from a computer word. The envisioned applications are representations of cartesian indices for matrices, some of which may store dense matrices non-densely in address space. That fact does not imply, however, that valuable memory is wasted. The gaps in the address space correspond to memory that never migrates into valuable cache. For really large matrices one may perceive it as pages on swapping disk that are never even allocated.

The usual bounds checking on a cartesian tuple/integer that indexes into such a matrix may be precisely done by masking off each dimension from that matrix index and comparing it with a pre-cast bound using a MaskedInteger comparator. Alternatively, a fast-and-coarse integer comparison can be made between that aggregate index and the bitwise-or of all the cartesian indices' improper upper bounds.

Unsigned integers are used here in order to avoid interference from the highest-order of the w bits acting as a sign bit in the class of integers whose mask includes it. If these integers are signed, then methods for its comparators need to respect that sign bit.

3.2 Example Code

The following segment of code for ijk matrix multiplication illustrates the use of these codes for *any* of the matrix indexings described above. That is, one can substitute any pair of `MaskedInteger` representations specified by complementary masks for `Row2D` and `Col2D` in the following:

```
unsigned int i,j,k;
Row2D  ii,kk;
Col2D jjj,kkk;
for     (i=0, ii=0       ; i<n; i++, ii++       )
  for   (j=0,       jjj=0; j<m; j++,       jjj++)
    for (k=0; kk=0, kkk=0; k<p; k++, kk++, kkk++)
      C[ii.getVal() + jjj.getVal()]
        += A[ii.getVal()+kkk.getVal()] * B[kk.getVal()+jjj.getVal()];
```

Pure integers are retained for loop control and integer addition is used to assemble the matrices' indices —because conventional optimizers understand their algebra (for unrolling and code motion).

Elementary casting-conversion algorithms between the underlying unsigned-integer types `T` and MaskedIntegers are readily available [11]. Casts to and from type `T` are reserved to effect those bit-compressing and bit-spreading conversions; table lookup is recommended. But they can be simple: when an integer `bo` is a power of two, like an inner `b`lock's `o`rder, then its dilation `(Row16Morton2D)bo` is its own square.

Conversions directly between related MaskedIntegers can also be simple. For instance, conversions among similar dilated-integer classes require only simple shifting. An example is the following cross construction of `ColMorton2D` that can displace `kkk` from the code above:

```
ColMorton2D(RowMorton2D kk) : value( (kk.getVal()) <<1 ) {}
```

Thus, the code above can be unwound to six nested loops without changing the blocked structure where, for either MortonHybrid and MajorMajor, conventional optimizations are easily and *locally* applied to the inner three loops, inside each block. Or they can be replaced by calls to row-major library functions there. [The spill conditional and its three inner-block loops are omitted in the following:]

```
unsigned int       I,J,K, i,j,k;
const int blockOrder=16;
Row16MortonHybrid  II,     KK;
Col16MortonHybrid     JJJ,    KKK;
for     (I=0,  II=0       ; I<n; I+=blockOrder, II += (Row16MortonHybrid)blockOrder )
  for   (J=0,        JJJ=0; J<m; J+=blockOrder, JJJ+= (Col16MortonHybrid)blockOrder )
    for (K=0;  KK=0, KKK=0; K<p; K+=blockOrder, KK += (Row16MortonHybrid)blockOrder,
                                                KKK+= (Col16MortonHybrid)blockOrder ) {
      double* c = C + II.getVal() + JJJ.getVal();
      double* a = A + II.getVal() + KKK.getVal();
      double* b = B + KK.getVal() + JJJ.getVal();
      for     (i=0; i<blockOrder; i++)
        for   (j=0; j<blockOrder; j++)
          for (k=0; k<blockOrder; k++)
            c[i + j*blockOrder] += a[i + k*blockOrder] * b[k + j*blockOrder];
    }
```

4 Conclusions

A whole family of integer representations is introduced with one set of operations that are sufficient for several important array representations. In particular Morton-ordered and Morton-hybrid arrays have already demonstrated more local addressing within the memory hierarchy. As that hierarchy grows, and as multi-core chips (chip multiprocessors) are used, there will be even greater demand for local access in aggregate structures. Corresponding to these matrix representations are the families of dilated integers and doppled integers, that allow immediate cartesian indexing into these structures. That is, alternative matrix representations are available without reprogramming, simply by transforming the representation and existing programs to use these new representations for indices [6].

Since these array structures and indexing algorithms generalize nicely, they gently introduce valuable locality into old codes. Unfortunately their current lack of software and hardware support create temporary disadvantages for them. Some C++ compilers impose an abstraction overhead on these templates at run time. The standard unroll and jam optimizations don't work on masked integers because conventional optimizers do not yet recognized that they form a simple abelian group. Similarly, their group is ordered and bounds checking works fine, but compliers don't yet know that. Finally, vectorizing hardware could accomodate them, as well. In all cases the programmer can help, in the last case with block-local declarations and loops equivalent to what the complier expects from row- or column-major arrays.

Remarkably, all these types are available from the same algorithms via a single template. Regardless of the mask defining these masked integers, however, only one algorithm is necessary for each operator across all these representations; only the constants vary [10, ¶1].

References

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

[2] Backus, J. The history of Fortran I, II, and III. In *History of Programming Languages*, R. L. Wexelblat, Ed. Academic Press, New York, 1981, pp. 25–45. Also preprinted in *SIGPLAN Not.*, 13(8):166–180, Aug. 1978. http://doi.acm.org/10.1145/800025.808380

[3] Chatterjee, S., Lebeck, A. R., Patnala, P. K., and Thottenthodi, M. Recursive array layouts and fast parallel matrix multiplication. *IEEE Trans. Parallel Distrib. Syst. 13*, 11 (Nov. 2002), 1105–1123. http://dx.doi.org/10.1109/TPDS.2002.1058095

[4] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw. 16*, 1 (Mar. 1990), 1–17. http://doi.acm.org/10.1145/77626.79170

[5] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. Cache–oblivious algorithms. In *Proc. 40th Ann. Symp. Foundations of Computer Science*. IEEE Computer Soc. Press, Washington, DC, Oct. 1999, pp. 285–298. http://dx.doi.org/10.1109/SFFCS.1999.814600

[6] Gabriel, S. T., and Wise, D. S. The Opie compiler from row-major source to Morton-ordered matrices. In *Proc. 3rd Wkshp. on Memory Performance Issues*, J. Carter and L. Zhang, Eds. ACM Press, New York, 2004, pp. 136–144. http://doi.acm.org/10.1145/1054943.1054962

[7] Golub, G. H., and Van Loan, C. F. *Matrix Computations*, third ed. The Johns Hopkins Univ. Press, Baltimore, 1996.

[8] Heller, J. *Catch-22*. Simon and Schuster, New York, 1961.

[9] Morton, G. M. A computer oriented geodetic data base and a new technique in file sequencing. Tech. rep., IBM Ltd., Ottawa, Ontario, Mar. 1966.

[10] Perlis, A. J. Special feature: Epigrams on programming. *SIGPLAN Not. 17*, 9 (1982), 7–13. http://doi.acm.org/10.1145/947955.1083808

[11] Raman, R., and Wise, D. S. Converting to and from dilated integers. Submitted for publication, Jan. 2006. http://www.cs.indiana.edu/˜dswise/Arcee/castingDilated-comb.pdf

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

[13] Wise, D. S. Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In *Euro-Par 2000 – Parallel Processing*, A. Bode, T. Ludwig, W. Karl, and R. Wismüller, Eds., vol. 1900 of *Lecture Notes in Comput. Sci.* Springer, Heidelberg, 2000, pp. 774–883. http://www.springerlink.com/link.asp?id=0pc0e9gfk4x9j5fa