Embedding Quadtree Matrices in a Lazy Functional Language [∗]

MICHAEL RAINEY [†]
Computer Science Dept.
Indiana University
Bloomington, IN 47405-7104, USA
(*e-mail:* `mrainey@cs.uchicago.edu`)

DAVID S. WISE [‡]
Computer Science Dept.
Indiana University
Bloomington, IN 47405-7104, USA
(*e-mail:* `dswise@cs.indiana.edu`)

Abstract

A technique for supporting quadtree matrices in a lazy functional langauge is presented that ameliorates tensions between performance and purity. In large matrix computations the facility to update in-place is a requisite for high performance. Thus, techniques for applying safe side effects to quadtree matrices are presented. Performance enhancements come via intrinsic properties of quadtree matrices, e.g., locality of reference. The expressiveness of quadtree matrix algorithms is enhanced by the tools of the higher-order, typed language, Haskell. Matrix-matrix multiply and Cholesky factorization are provided with time comparisons to C. In contrast to our earlier work, the protocol is convenient even for Fortran programmers and its performance scales nicely and compares well with Fortran enough to demonstrate that lazy languages can compete.

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

Additional Key Words and Phrases: HASKELL, quadtrees, functional programming

∗ Copyright©2005 by the Trustees of Indiana University; all rights reserved. Supported, in part, by the National Science Foundation under grants numbered EIA-0202048 and CCR-0073491. The US Government has a worldwide license to this copyright, excepting sales.
† Current address: University of Chicago, 1100 E 58th Street, Chicago, IL 60637, USA. Supported, in part, by the National Science Foundation under an REU grant numbered NSF CCR03-34593.
‡ Supported, in part, by the National Science Foundation under a grant numbered NSF ACI-0219884.

1 Introduction

A general problem is posed to implementers of lazy functional languages: How can arrays be efficiently and cleanly implemented without compromising the purity of the language? This is, indeed, a difficult question because functional languages seem better suited to incremental lists than arrays. It is, nevertheless, essential to performance for programming applications, e.g., in scientific computing, to have in-place array structures. The utility of a language comes into question without efficient array support. There are many proposed solutions for arrays in lazy functional languages, discussed in Section 5. Only a few are used in practice. Furthermore, there seems to be no single accepted way to support arrays.

This paper is concerned with embedding *quadtree matrices* into lazy functional languages. An inherited benefit of using them is the block recursive decomposition they support fits naturally into divide-and-conquer functional style.

The underlying structure of quadtree matrices is an array. So, providing this structure in a lazy functional language remains a tricky problem. The methods and techniques for quadtree matrices are made specific here; they are only intended for two-dimensional arrays.

There are three goals to make quadtree matrices into a valuable functional programming tool:

- The indexing schemes for quadtree matrices can be seamlessly utilized with Haskell's type system. **Structured libraries** are thus provided to exploit the benefits of this higher-order, typed language.
- Matrix algorithms should be **efficient**. In-place updates are necessary to be competitive, but functional purity must be enforced. The tools presented here address performance and style using other research on quadtree matrices. Quadtree matrices are known for their excellent patterns of memory access at all levels of the hierarchy (Frens & Wise, 1997). Thus, they fulfill an important goal for programming-language research: Quadtree matrices address efficiency in a high-level, machine-independent style. For reasons of style and performance, functional languages deserve support for quadtree matrices.
- This method for providing **safe side effects** should not be specific to any lazy functional language. Haskell is used for this implementation because of the Glasgow Haskell Compiler's rich feature set. Other languages will work as well.

The rest of the paper is structured as follows. To make matrix algorithms efficient, a method for safely side-effecting must be available, and Section 2 describes a method for them. Section 3 gives a review of definitions for quadtree matrices and their various indexing schemes. These are used for building the indexing and matrix tools. Section 4 presents the quadtree matrices for Haskell, and the block recursive style of programming them. Section 5 gives a history of previous solutions to functional arrays, and comparisons to the solution provided in Section 2. Finally, Section 6 draws conclusions and suggests future research.

2 Safe Side Effects

A contribution of this paper is to provide safe side effects for quadtree matrices in a freewheeling functional style. The underlying array to be side-effected should be named and duplicated safely, as well.[1] This is not a straightforward task, however, due to Haskell's lazy semantics. The major hurdle is guaranteeing the array is *single-threaded*. C and Fortran are implicitly single-threaded, and so millions of programmers take single threading for granted. But in a lazy language it is crucial to define this property clearly.

Definition 1
(Anderson & Hudak, 1990) An array is *single-threaded* if, at each in-place update, no reference to the previous version of the array is available.

Single threading an array implies that it is uniquely referenced along a thread. It imposes a far weaker requirement than a monad.

2.1 Operational description

The approach proposed here allows safe mutable arrays. Compile-time analyses of single-threadedness remains future work. Indeed, some measure must ensure side effects are single-threaded; this property is too important to leave to the programmer. Therefore, a lightweight, runtime solution enforces it. The workhorse can be thought of as a vague analog of a `cons` box, dubbed an *array box* (Dybvig, 1996). The array box is an *opaque* data type. That is, the array box's internals are inaccessible from user code. These internals contain a reference to the array, and possibly a small stub of information about the array, for instance, bounds. *To guarantee single-threadedness, the array box is poisoned after each in-place update.* A fresh array box is then generated for the next access. This process is described later in more detail.

There are three functions permitted to access the array box's internals, namely $setMtx$, $getMtx$, and $safeGetMtx$. To preserve single-threadedness, it is important to concentrate on $setMtx$ because it performs the in-place update. It is expressed as follows: Given an array x mutate the location of ix with value v.

```
let x' = setMtx x ix v in
   ...
```

An overly safe, conventional implementation for this update, namely *pass-by-value* (Hudak, 1986), dictates that the original array x is copied to a new location x'. Thus, pass-by-value has n^2 worst case time on an $n \times n$ matrix. But if x is known to be single-threaded, *pass-by-reference* (Hudak, 1986) suffices. With pass-by-reference, only the pointer to the old array needs to be copied. This is preferred because it has constant time access. Therefore, $setMtx$ only generates an array box for future accesses. What remains to be shown is *how* $setMtx$ can safely pass-by-reference.

[1] "Side-effect" is hereby a transitive verb.

```
class Mtx a b where
    setMtx     :: (MatIx d) =>
                  Matrix a b ->   -- a matrix A
                  d ->            -- an index
                  b ->            -- a value
                  Matrix a b      -- A'
    getMtx     :: (MatIx d) =>
                  Matrix a b ->   -- a matrix A
                  d ->            -- an index
                  b               -- a value
    safeGetMtx :: (MatIx d) =>
                  Matrix a b      -- a matrix A
                  d ->            -- an index
                  (b,Matrix a b)
```

Fig. 1. Functions that access the array.

Four operations are performed when *setMtx* is executed. First, the incoming reference is stored temporarily. The following step is the key to enforcing single-threadedness: A poisoning `nil` is written to the incoming array box. It prevents access to obsolete array boxes like x. So, the physical array can be mutated only once. Finally, the temporary reference is boxed and returned as a new array box. In a nutshell, *setMtx*'s job is to poison an obsolete path and to mutate the array. Figure 2 shows a possible single-threaded evaluation sequence on an array to make this more tangible. Notice *setMtx* always ends operations on the array box, and destroys its internal reference. Also, these in-place updates need not be restricted to 1×1 leaves; 32×32 blocks can be updated at a stroke.

To make this process more concrete, consider the following example where an array is is mutated with multiple threads. The function *dupSetMtx* generates a pair of in-place updates on *mtx*. This is a perfectly legal definition by itself. But an attempt to access both side effects of the pair would violate Definition 1. The violation happens because the array has been split across separate threads. Evaluating one of the threads side-effects the array, thus destroying *mtx*'s reference. When both threads are evaluated, as in *illegalOp*, the latter thread fails because a `nil` was baited in the reference. The result of *illegalOp* is a single-threadedness error message. It should be apparent now that in-place updates must always be single-threaded; an attempt to access a poisoned copy of an array always signals an error at run time.

```
-- Duplicate in-place updates to mtx at
-- location ix.
dupSetMtx = let mtx = allocMtx n in
                 (setMtx mtx ix v, setMtx mtx ix v)

-- Only one of the threads is evaluated.
-- There is no single-threadedness violation in
-- legalOp.
```

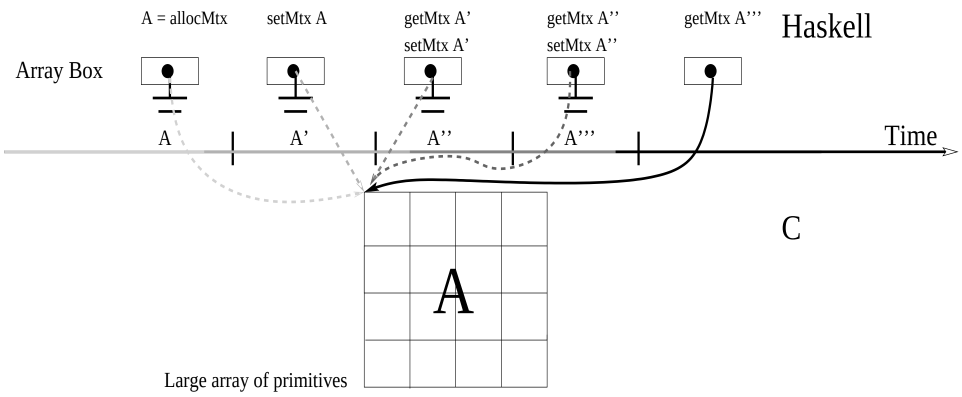

Fig. 2. An example evaluation of array operations.

```
legalOp = let (mtx1,mtx2) = dupSetMtx in
             (getMtx mtx1 ix)

-- This code forces evaluation of mtx1 and mtx2,
-- thus violating the single threadedness of
-- the array.
illegalOp = let (mtx1,mtx2) = dupSetMtx in
               (getMtx mtx1 ix) + (getMtx mtx2 ix)
```

There are a few obvious weaknesses of this approach. First, single-threadedness errors are only signaled at run-time. Debugging an algorithm might be exacerbated by the lack of compile-time warnings.

Another disadvantage of this method is the need to generate a new handle (identifier) after each *setMtx*. Since most block recursive matrix algorithms require only eight or less recursive calls per environment, these weaknesses should not overburden the programmer (Frens & Wise, 1997; Wise, 1999; Wise *et al.*, 2001; Wise, 1987; Wise, 2000).

2.2 Performance Issues

The next potential problem is performance. Chakravarty and Keller identify several factors that can make Haskell arrays slow (Chakravarty & Keller, 2003). One big performance penalty comes from storing array elements lazily. Lazy arrays require heap allocated boxes for elements, which are likely to have poor locality of reference. Another source of inefficiency is nesting, for example, when blocks are independently heap allocated. Quadtree arrays are therefore flat stores of primitives. Consequently, quadtree arrays are strict. Chakravarty and Keller call this a *parallel array* because all elements of the array are computed at the first time an element is needed. These guidelines provide a framework for fast arrays. Yet, there are further hurdles to jump before arrays can truly be fast.

2.2.1 Base cases

For resource intensive matrix algorithms, such as matrix-matrix multiplication and various matrix factorizations, optimized base case blocks are crucial for performance. There are several techniques that could optimize the base case. Unfolding of the base cases(Burstall & Darlington, 1977) can be automated with something like Template Haskell (Lynagh, 2003).

This paper demonstrates a manual technique for block recursive matrix-matrix multiply: unfolding the base case (with rerolling). The implementation of this algorithm is described in Section 4.3. Recursions are stopped above 32×32 blocks. Machine-generated C code (Figure 17) iteratively multiplies within the base block. The following section contains time comparisons based on this technique.

The tuned base case block used in time tests is 32×32. Two considerations influence the block size:

- A major motivation for using Morton-ordered matrices is cache reuse. Introducing a new array box for each write would spoil any chance for efficient cache reuse.[2] A 16×16 base case is decent for most modern cache architectures (Wise *et al.*, 2001).
- In addition to improving cache reuse, the array box allocation cost is amortized by using larger base cases. Say, for example, some code writes to each element of an $n \times n$ array. Recurring down to the 1×1 base case requires n^2 array box allocations. But recurring down to 32×32 blocks requires $n^2/1024$ array box allocations.

2.2.2 Experimental Results

Results are presented from an Intel Pentium IV with 20kB on-chip cache, 256KB L2 cache, and 256MB RAM, and also from an Intel Xeon with 12kB on-chip cache, 512KB L2 cache, and 2GB RAM. Tests are compiled with the Glasgow Haskell Compiler and the GNU C Compiler. Haskell code is compiled with optimization `-O2 -fnumbers-strict -fvia-c`; C is compiled with `-O2 -mcpu=pentium4`.

Three possible matrix multiply bases cases (Figures 14, 15, and 17) are plugged into the recursive algorithm in Figure 16. Time comparisons are presented in Figure 3. The bottom graph is a normalization of time by dividing it by its $2n^3$ flops for an order-n matrix multiply (Johnson, 2002). Doing this extracts the leading coefficient from this $O(n^3)$ algorithm. These plots indicate the substantial impact bases cases have on performance. The expanded base case outperforms standard triple-loop algorithms because of its in-line code. Thus, subsequent Haskell times use the expanded base case to 32×32 blocks. This block size is useful for eliminating array-box allocations, and exhibits decent cache reuse.

Time tests show Haskell to be only 33% slower than C for matrix-matrix multiply

[2] Recurring down to a 1×1 base case has never been recommended for high performance (Frens & Wise, 1997).

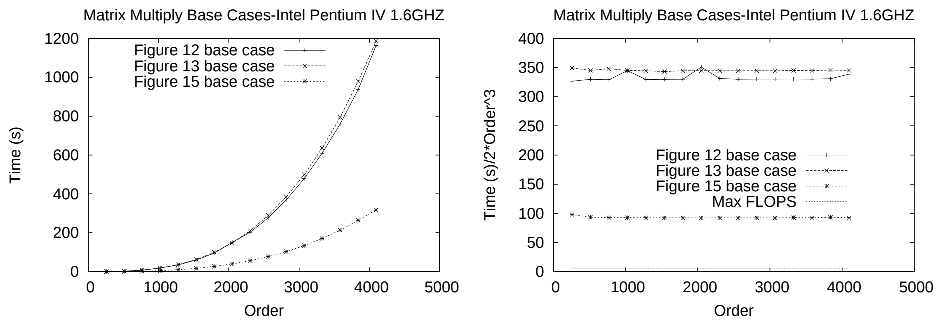

Fig. 3. Comparison of Haskell matrix multiplication base cases on a 1.6GHz Intel Pentium IV.

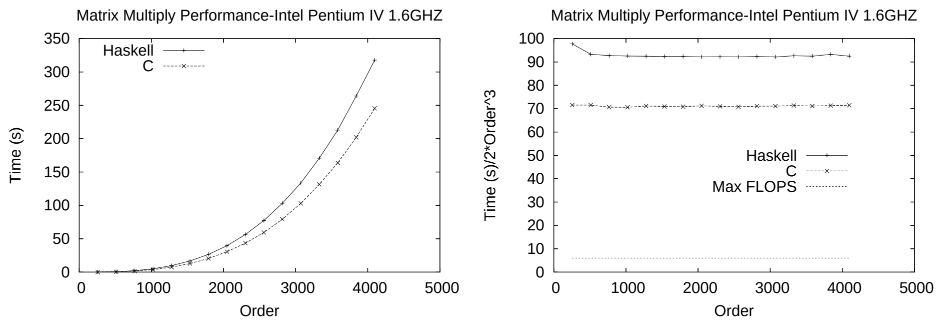

Fig. 4. Comparison of Haskell and C matrix multiplication on a 1.6GHz Intel Pentium IV.

on the Pentium. Figure 4 plots running times from the algorithm in Figure 16 and its C equivalent. The results show that, despite the array box overhead, Haskell, with the tuned base case, is not far behind C code. The C code is already known to scale nicely (Frens & Wise, 1997). The ratios in the bottom graph show the Haskell version can scale without too much overhead penalty; it remains constant, although slightly slower than C. Figure 6, however, shows Haskell has a larger penalty than 33% on the Xeon.

Cholesky factorization is a more substantial example for Haskell quadtree matrix libraries. Elmroth, Gustavson, Jonsson, and Kågstroöm use this algorithm to demonstrate block recursion (Elmroth *et al.*, 2004). Thus, Cholesky factorization is used here as a non-trivial comparison of Haskell and C. The technique for coding Cholesky is the same as multiplication. Timing results are provided in Figures 5 and 7. Similar to matrix-matrix multiply, the Haskell code is roughly 33% slower C on the Pentium. Haskell Cholesky is also close to 33% slower on the Xeon.

2.3 Uniprocessing vs. Multi-processing

This technique for enforcing single-threadedness is monad-neutral. That is, the programmer can use monads to code-in linearity for her algorithms. Or, alteratively,

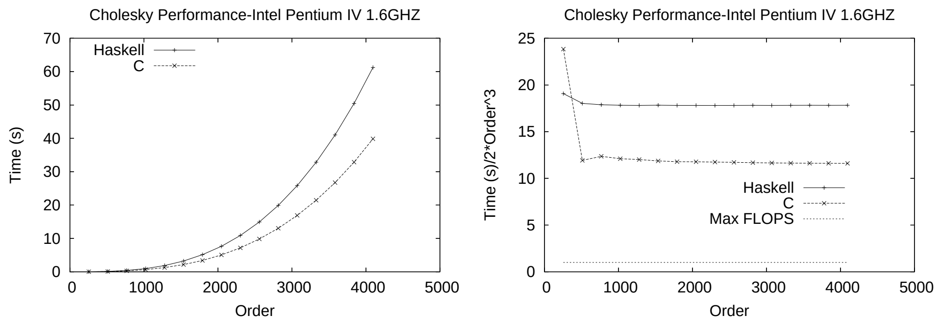

Fig. 5. Comparison of Haskell and C Cholesky Factorization on a 1.6GHz Intel Pentium IV.

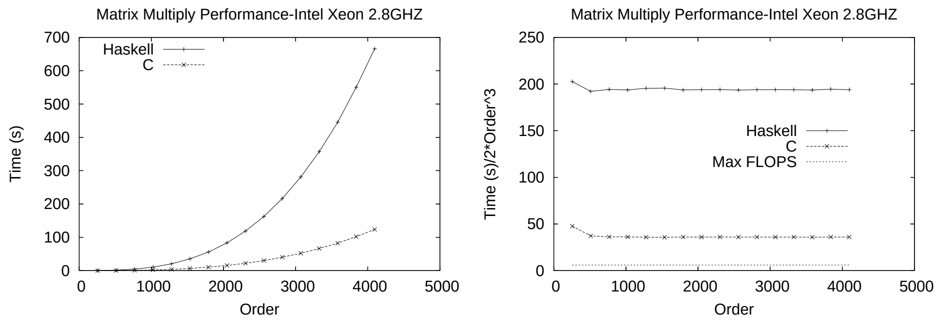

Fig. 6. Comparison of Haskell and C matrix multiplication on a 2.8GHz Intel Xeon.

the programmer can be responsible for linearity without using monads. In both cases, the scheme for single-threadedness fits. Data dependencies enforce linearity in the latter case. Furthermore, splitting data dependencies allows for parallelism. Thus, this non-monadic approach emphasizes opportunities for parallel dispatch. Previous work investigates parallelism over quadtree matrices (Frens & Wise, 1997).

For convenience, both monadic and non-monadic libraries are provided for quadtree arrays. Marketing quadtree array libraries to Fortran and C programmers is another motive for providing the non-monadic version. Fortran programmers, for example, might be more likely to use Haskell for scientific computing without the burden of honoring the restrictions of a the monad.

3 Definitions

The following implements a 2-dimensional array, a matrix.

Definition 2
(Wise, 2000) The base of a matrix has Morton-order index 0. A sub-matrix (block) at Morton-order index i is either an element (scalar), or it is composed of 4 sub-matrices indexed $4i + 0, 4i + 1, 4i + 2, 4i + 3$.

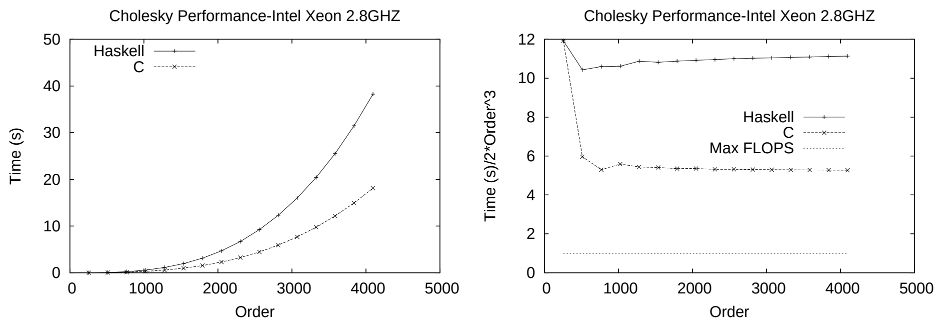

Fig. 7. Comparison of Haskell and C Cholesky Factorization on a 2.8GHz Intel Xeon.

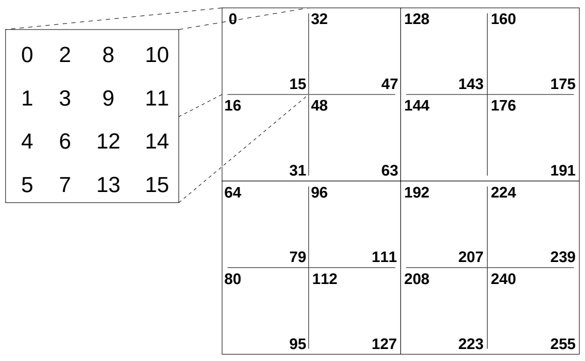

Fig. 8. Morton indexing of a 16 × 16 matrix.

These sub-matrices are identified as northwest, southwest, northeast, and southeast respectively. Figure 8 shows the Morton-order indexing of a 16 × 16 matrix.

Definition 3
(Wise, 2000) A complete matrix has Ahnentafel index 3. A sub-matrix (block) at Ahnentafel index i is either a scalar, or it is composed of 4 submatrices, with indices $4i + 0, 4i + 1, 4i + 2, 4i + 3$.

Definition 4
(Wise, 2000) A complete matrix has Level-order index 0. A sub-matrix (block) at Level-order index i is either a scalar, or it is composed of 4 submatrices, with indices $4i + 1, 4i + 2, 4i + 3, 4i + 4$.

Operators for Morton-order, Ahnentafel, and Level-order are later defined by the $QdIx$ class in Section 4.1.

Theorem 1
(Wise, 2000) The difference between Ahnentafel index of a matrix block at level l and its Morton-order index is $3 \cdot 4^l$

Theorem 2
(Wise, 2000) The difference between Level-order index of a matrix block at level l and its Morton-order index is $(4^l - 1)/3$

Conversions between Morton-order, Ahnentafel, and Level-order indexing schemes are therefore cheap, only a few processor cycles.

Theorem 3
(Wise, 2000) The Morton-indices on the elements of a matrix, or Ahnentafel or Level-order indices on blocks of any single size, increase monotonically to the east and south.

Thus, bounds checking is available for these indexing schemes; it is made exact with dilated integers, introduced below. Also, because Level-order, Morton-order, and Ahnentafel indices increase monotonically to the east and south, the universal index class *MatIx a* in Section 4.1 is consistent with *Ord a*.

The algebra of dilated integers is presented tersely here. In Haskell the dilated Int is an instance of *Num*. All dilated `Ints` are stored internally as "even dilated" by default.

Notation 1
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`.

Definition 5
(Wise, 2000) 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 $\overleftarrow{\jmath}$.

The arrows suggest the justification of the meaningful bits in either dilated representation.

Theorem 4
(Wise, 2000) A matrix of m rows and n columns is allocated as a sequential block of $\overrightarrow{m-1} + \overleftarrow{n-1} + 1$ scalar addresses.

Theorem 5
(Schrack, 1992) With "$\equiv$" read as semantic equivalence and with "==" denoting equality on integer representations, then for unsigned integers

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

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

Theorem 6
(Wise, 2000) The Morton index for the $\langle i, j\rangle^{\mathrm{th}}$ element of a matrix is $\overrightarrow{\imath} \vee \overleftarrow{\jmath}, or\, \overrightarrow{\imath} + \overleftarrow{\jmath}$

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

Definition 6
(Wise, 2000) Addition $(\overset{\rightarrow}{+}, \overset{\leftarrow}{+})$ and subtraction $(\overset{\rightarrow}{-}, \overset{\leftarrow}{-})$ of dilated integers:

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

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

Theorem 7
(Wise, 2000) Register-local implementations of subtraction, addition, constant addition, and constant shifts of twos-complement dilated integers:

$$\overrightarrow{i}\overset{\rightarrow}{-}\overrightarrow{n} = (\overrightarrow{i} - \overrightarrow{n})\&\overrightarrow{b}; \qquad \textit{(Schrack, 1992)}$$

$$\overleftarrow{j}\overset{\leftarrow}{-}\overleftarrow{n} = (\overleftarrow{j} - \overleftarrow{n})\&\overleftarrow{b}; \qquad \textit{(Schrack, 1992)}$$

$$\overrightarrow{i}\overset{\rightarrow}{+}\overrightarrow{n} = (\overrightarrow{i} + \overleftarrow{b} + \overrightarrow{n})\&\overrightarrow{b}; \qquad \textit{(Schrack, 1992)}$$

$$\overleftarrow{j}\overset{\leftarrow}{+}\overleftarrow{n} = (\overleftarrow{j} + \overrightarrow{b} + \overleftarrow{n})\&\overleftarrow{b}; \qquad \textit{(Schrack, 1992)}$$

$$\overrightarrow{i}\overset{\rightarrow}{+}\overrightarrow{c} = \overrightarrow{i}\overset{\rightarrow}{-}\overrightarrow{(-c)}; \qquad \overleftarrow{j}\overset{\leftarrow}{+}\overleftarrow{c} = \overleftarrow{j}\overset{\leftarrow}{-}\overleftarrow{(-c)}; \qquad \textit{(Wise, 2000)}$$

$$\overrightarrow{b} = \overrightarrow{-1}; \qquad \overleftarrow{b} = \overleftarrow{-1}; \qquad \textit{(Wise, 2000)}$$

$$\overrightarrow{i\texttt{<<}k} = \overrightarrow{i}\texttt{<<}(2k); \qquad \overleftarrow{j\texttt{<<}k} = \overleftarrow{j}\texttt{<<}(2k); \qquad \textit{(Wise, 2000)}$$

$$\overrightarrow{i\texttt{>>>}k} = \overrightarrow{i}\texttt{>>>}(2k); \qquad \overleftarrow{j\texttt{>>>}k} = \overleftarrow{j}\texttt{>>>}(2k). \qquad \textit{(Wise, 2000)}$$

4 Quadtree Matrix Libraries

The previous section provides mutable arrays. Now it is possible to build a practical framework for quadtree matrices on top of it. Types and functions for indexing matrices are provided. Then the underlying matrix representation is examined. Finally, a style for programming quadtree matrices is presented with examples.

4.1 Universal Indexing

Another contribution of this paper is to provide a method for using various indexing schemes interchangeably. Haskell's polymorphic type system simplifies this by seamlessly applying defined coercions (Hall *et al.*, 1996). The class structure to support these coercions (Figure 9) is defined as follows: $MatIx$ is the superclass for all indices. It defines the essential coercions $toMorton$ and $toRowMajor$, which take an index closer to a hardware address. Higher level indices, such as $AhnenIx$ and $CartesianIx$, determine the style of recursions and bounds checking.

Indexing operations and coercions must be fast. Thus, index types are built from the Glasgow Haskell Compiler's 32-bit $Word$. Because $Word$ is unsigned, bounds checking is supported by Theorem 3. All coercions are done in unit time as they are invoked frequently.

Finally, the $QdIx$ class (Figure 10) is defined for indices that support quadtree decomposition. Quadtree decomposition is particularly relevant to Haskell because

```
class (Ord a) => MatIx a where
   toMorton   :: a ->
                 Word ->  -- Row major stride
                 MortonIx
   toRowMajor :: a ->
                 Word  -> -- Row major stride
                 RowMajorIx
   toWord     :: a -> Word

newtype MortonIx    = MkM Word
newtype RowMajorIx  = MkRM Word
newtype AhnenIx     = MkAhnen Word
newtype LevelIx     = MkLevel Word
newtype CartesianIx = MkCart (Word,Word)
```

Fig. 9. Indexing class.

```
class (MatIx a) => QdIx a where
    root        :: a
    nw,ne,sw,se :: a -> a

instance QdIx MortonIx where
   ...

instance QdIx AhnenIx where
  ...

instance QdIx LevelIx where
  ...
```

Fig. 10. Class for indices that support quadtree decomposition.

it leads to block recursion, a style arguably more in the spirit of functional programming (Wise, 1987). The block recursive style is described in Section 4.3.

There are several good reasons to use each of the *QdIx* block indices. For instance, an Ahnentafel index is useful for recursive control (Wise, 2000). Because it avoids constant conversions and has good locality of reference, *MortonIx* should be typically used only in a base case block.

4.2 Matrix Representation

Morton-ordered and row-major matrix representations are provided in the Quadtree Matrix libraries.[3] Both can accept an index of type *MatIx* because any instance of this class is coercible to Morton-order or row-major. As seen in Figure 1, matrix operations take any kind of index, which need not be the same type of index as the matrix's internal representation.

[3] Column major would also be possible.

```
data Matrix a b =
          MkMMtx {   -- Morton-order
                   mb    :: ArrayBox,
                   order :: Word         } |
          MkRMMtx {  -- row-major
                   mb    :: ArrayBox,
                   m     :: Word,
                   n     :: Word         }
```

Fig. 11. The Matrix data constructor.

```
instance Mtx MortonIx Double where
   setMtx (MkMMtx mb order) ix v =
     if mb == null
        then error "Single-threadedness violation"
        else
          -- Create the fresh ‘‘array box’’ mb’
          -- and plant a null in mb.
          let mix = (toMorton ix order) in
            case writePrimMtx mb mix v of
             () -> MkMMtx mb’ order
   getMtx (MkMMtx mb order) ix =
     if mb == null
        then error "Single-threadedness violation"
        else readPrimMtx mb (toMorton ix order)
   safeGetMtx mtx@(MkMMtx mb order) ix =
     if mb == null
        then error "Single-threadedness violation"
        else
          -- Create the fresh ‘‘array box’’
          -- and plant a null in mb.
            let mix = (toMorton ix order) in
               (readPrimMtx mb mix, mtx)
```

Fig. 12. Pseudocode for $setMtx$, $getMtx$, and $safeGetMtx$.

The $Matrix$ data constructor, shown in Figure 11, determines the two properties of the matrix: the matrix representation, e.g., Morton order or row major, and the type of stored values. The minimal fields, as in $MkMMtx$ and $MkRMMtx$, consist of the array box and matrix's order or dimensions. For efficiency, the number of the matrix's data fields should be minimized, as the fields are copied at each in-place update.

The internals of the $Matrix$ data constructor are held private to the module. Open access to the $ArrayBox$ would allow duplicating the pointer, thereby undermining the enforcement of single-threadedness.

There are a few situations where the ability to sequence read operations is necessary. Most of these can be avoided in practice, but $safeGetMtx$ is provided for sequencing reads anyway. The in-place transpose of two matrix elements (Figure 13) demonstrates the need for $safeGetMtx$. The reason is that the first read must

```
let mtx' =
       -- xi = transpose of index ix
   let xi = combine (getCol ix) (getRow ix)
       -- read value at ix
       (tmp1,m1)   = safeGetMtx m0 ix
       -- tread value at xi
       tmp2       = getMtx m1 xi
       -- set value at ix
       m2       = setMtx m1 ix tmp2   in
       -- set value at xi
                   setMtx m2 xi tmp1
```

Fig. 13. In-place transpose of two elements.

be evaluated before the first write. In the transpose, the matrix m_1 must be used to reflect the state of the matrix when the read takes place. Luckily, situations like this, where an in-place swap is necessary, are rare.

4.3 Block Recursion

This section introduces matrix-matrix multiplication as a working example of the Haskell quadtree matrix. Figures 14, 15, and 16 show very different possible styles for solving this problem. The first two use C-style dot products 1.1.12. Row major is represented by Figure 14; Morton order is represented by 15. The block-recursive algorithm in Figure 16 is a simplified version of one from Frens and Wise (Frens & Wise, 1997). A more cache efficient ordering of recursions, called dovetailing, is left out for brevity (Frens & Wise, 1997).

There are a few important things to note about the recursive code. The result of $a * b$ is accumulated in c. Parameters to *matrixMult* follow the order of an assignment statement. The first argument is c, the matrix to be side-effected. Subsequent arguments are read-only. Also, this code uses Ahnentafel indices to control its eight recursions. Fast translation from Ahnentafel to Morton-order indices happens transparently at the base block. As Section 2.2 mentions, the base case is optimized using a simple transformation that exploits super-scalar processing and retains block recursive style.

The technique of unfolding (with rerolling) the base case is described briefly. Start by examining the base case in the example code. Recursions stop at 1×1 blocks to call *setMtx*. A modification to the bounds checking allows for recursions to stop above 32×32 blocks instead. Now the *setMtx* call is replaced with iterative code from Figure 15. A further optimization to the iterative code is possible because the base block size is known. The code in Figure 15 is unrolled to a 32×32 in-line code of Figure 17. Now the processor can pipeline this unrolled base case. In practice the in-line base case is generated by a macro.

```
for(int i = 0; i < n; i++) {
   for(int j = 0; j < m; j++) {
      for(int k = 0; k < p; k++) {
         c[i * cStride + j] += a[i * aStride + k]
                             * b[k * bStride + j];
      }
   }
}
```

Fig. 14. Inner-product row-major Matrix Multiplication (in C) 1.1.12.

```
for(int i = 0; i < evenDilate(n); evenInc(i)) {
   for(int j = 0; j < oddDilate(m); oddInc(j)) {
      for(int k = 0; k < oddDilate(p); oddInc(k)) {
         c[i + j] += a[i + k] * b[oddToEven(k) + j];
      }
   }
}
```

Fig. 15. Inner-product Morton-order Matrix Multiplication (in C) 1.1.12.

5 History of Functional Arrays

Lazy functional languages require special measures for clean, safe side effects. Many solutions have appeared with varying degrees of success. One early method uses a compile-time path analysis to determine single-threadedness of updates (Bloss, 1989). Hudak offers a reference counting scheme (Hudak, 1986). Single-threadedness is guaranteed if the reference count for the aggregate is always one. Another proposed run-time method is *trailers*. Instead of copying a whole array at each update, trailers just copy the cell being updated. All of these schemes are too costly for practical use (Anderson & Hudak, 1990). More efficient approaches are now used in Haskell and Clean.

Monads are the common means by which Haskell programmers destructively update data structures. The method for guaranteeing single-threadedness via monads has long been known (Peyton Jones & Wadler, 1993). But the ability to destructively update *named references* is not as straightforward, and not permitted with the aforementioned method. The monadic way of dealing with references involves *state*, a finite mapping from references to values (Launchbury & Peyton Jones, 1994). When state is implemented naively, Chen and Hudak show a trivial example of how it can be duplicated, thereby precluding the enforcement of single-threadedness (Chen & Hudak, 1997). Launchbury and Peyton Jones solve this problem using *parametric polymorphism* to encapsulate the state (Launchbury & Peyton Jones, 1994). One benefit is support for named references. But their solution requires something beyond the Hindley-Milner type system; either rank-2 polymorphic types must be supported, or a special type judgment must be hard-coded. The popular Glasgow Haskell Compiler uses the former technique for in-place updates.

```
-- To be read as: Store the result of A * B in C.
matrixMult :: (Mtx a b) => Matrix a b ->  -- C
                        a b ->            -- A
                        a b ->            -- B
                        a b ->            -- C'
matrixMult c a b = mm c (root::AhnenIx)
                        (root::AhnenIx)
                        (root::AhnenIx) 0
  where leafBound = mkLeafBound (getOrder c)
        mm    c0 cIx aIx bIx lvl
             -- out of bounds
           | (outOfBnds aIx) || (outOfBnds bIx) = c0
             -- base case
           | cIx >= leafBound =
               setMtx c0 cIx ((getMtx a aIx) *
                              (getMtx b bIx) +
                              (getMtx c0 cIx))
         -- block recursion
           | otherwise =
               let c1 = mm c0 (nw cIx) (nw aIx) (nw bIx)
                   c2 = mm c1 (nw cIx) (ne aIx) (sw bIx)
                   c3 = mm c2 (ne cIx) (nw aIx) (ne bIx)
                   c4 = mm c3 (ne cIx) (ne aIx) (se bIx)
                   c5 = mm c4 (sw cIx) (sw aIx) (nw bIx)
                   c6 = mm c5 (sw cIx) (se aIx) (sw bIx)
                   c7 = mm c6 (se cIx) (sw aIx) (ne bIx)
                   c8 = mm c7 (se cIx) (se aIx) (se bIx) in
                        c8
```

Fig. 16. Block decomposition Morton-order Matrix Multiplication (in Haskell).

Haskell's close cousin, Clean, offers a similar approach to in-place updates (Barendsen & Smetsers, 1993). Clean has a special "unique type" for references. The type system ensures that each side effect is given the sole reference to a value. It is interesting to contrast this paper's technique with Clean and Haskell. Clean shifts the burden of analysis on the type system. The expense is that the programmer must learn about unique types, something a C or Fortran programmer might consider distracting. Haskell has a similar problem. The programmer must be familiar with monads to use in-place updates. Even though monads support in-place updates, the ability to name arrays only comes with something like the technique described in (Launchbury & Peyton Jones, 1994). The array box technique does not depend on monads or the type system. It will work for any lazy functional language. All that is needed is a way to disable the incoming array box.

6 Conclusions

Since efficient use of the memory hierarchy increasingly dominates performance, sound languages, like Haskell, must rise to the challenge of elegantly expressing memory-efficient algorithms. This fact is a strong motive for embedding quadtree

```
void baseCaseDoubleMult(
                 double* C, double* A, double* B,
                 int cIx,   int aIx,   int bIx,
                 int bcSize) {
 int ij,k;
 for     (ij = 0; ij<(32*32); ij+=16)
  {

  double*  restrict cBlock = C + cIx *(32*32) +ij;

  for (k=0; k<(32*32); k+=128)
  {

   double* restrict aBlock = A + aIx *(32*32)
                               + (ij & 0xa0) + k/2;
   double* restrict bBlock = B + bIx *(32*32)
                               + (ij & 0x50) + k;


   cBlock[ 0]
    += aBlock[ 0] * bBlock[ 0] + aBlock[ 1] * bBlock[ 2]
     + aBlock[ 4] * bBlock[ 8] + aBlock[ 5] * bBlock[10]
     + aBlock[16] * bBlock[32] + aBlock[17] * bBlock[34]
     + aBlock[20] * bBlock[40] + aBlock[21] * bBlock[42];

   cBlock[ 1]
    += aBlock[ 0] * bBlock[ 1] + aBlock[ 1] * bBlock[ 3]
     + aBlock[ 4] * bBlock[ 9] + aBlock[ 5] * bBlock[11]
     + aBlock[16] * bBlock[33] + aBlock[17] * bBlock[35]
     + aBlock[20] * bBlock[41] + aBlock[21] * bBlock[43];

   cBlock[ 2]
    += aBlock[ 2] * bBlock[ 0] + aBlock[ 3] * bBlock[ 2]
     + aBlock[ 6] * bBlock[ 8] + aBlock[ 7] * bBlock[10]
     + aBlock[18] * bBlock[32] + aBlock[19] * bBlock[34]
     + aBlock[22] * bBlock[40] + aBlock[23] * bBlock[42];

   cBlock[ 3]
    += aBlock[ 2] * bBlock[ 1] + aBlock[ 3] * bBlock[ 3]
     + aBlock[ 6] * bBlock[ 9] + aBlock[ 7] * bBlock[11]
     + aBlock[18] * bBlock[33] + aBlock[19] * bBlock[35]
     + aBlock[22] * bBlock[41] + aBlock[23] * bBlock[43];

      ...

   cBlock[15]
    += aBlock[10] * bBlock[ 5] + aBlock[11] * bBlock[ 7]
     + aBlock[14] * bBlock[13] + aBlock[15] * bBlock[15]
     + aBlock[26] * bBlock[37] + aBlock[27] * bBlock[39]
     + aBlock[30] * bBlock[45] + aBlock[31] * bBlock[47];

  }
 }
}
```

Fig. 17. An in-line, 32×32 base case block for matrix multiplication (Frens & Wise, 1997).

matrices in any functional language. They provide machine-independent performance and allow for block-recursive decomposition (Wise *et al.*, 2001). But laziness makes quadtree matrices inefficient. The second problem that is tackled in this paper is that of safe side effects. Performance hinges on being able to change values in place without system overhead.

A general theme of this paper is to build a framework for making Haskell accessible and competitive with scientific Fortran programming. The attack is therefore two-pronged; allow for performance via side effects, and utilize a programming style that exploits the benefits of functional languages.

There is much work to be done to improve support for quadtree matrices in lazy languages. A stylistic improvement might be gained by adding some syntactic sugar for quadtree decomposition. As can be seen in Figure 16, there is a good deal of structure to be exploited. The question is how to do exploit the structure *well*.

Haskell has long offered elegant "array comprehensions". There has been work to optimize their performance (Anderson & Hudak, 1990; Chakravarty & Keller, 2001). Another improvement would be to add compiler support for quadtree matrices. A method for automatically unfolding recursions and rerolling a fast base case would accelerate performance while keeping code clean. How feasible this may be remains to be investigated.

References

Anderson, Steven, & Hudak, Paul. (1990). Compilation of Haskell array comprehensions for scientific computing. *Proc. ACM SIGPLAN '90 Conf. on Program. Language Design and Implementation, SIGPLAN Not.*, **25**(6), 137–149. http://doi.acm.org/10.1145/93548.93561

Barendsen, Erik, & Smetsers, Sjaak. (1993). Conventional and uniqueness typing in graph rewrite systems. *Pages 41–51 of:* Shyamasundar, Rudrapatna (ed), *Foundations of Software Technology and Theoretical Computer Science 13*. Lecture Notes in Comput. Sci., vol. 761. Berlin: Springer.

Bloss, Adrienne. (1989). Update analysis and the efficient implementation of functional aggregates. *Pages 26–38 of: Proc. 4th Int. Conf. on Functional Programming Languages and Computer Architecture*. New York: ACM Press. http://doi.acm.org/10.1145/99370.99373

Burstall, R. M., & Darlington, John. (1977). A transformation system for developing recursive programs. *J. ACM*, **24**(1), 44–67. http://doi.acm.org/10.1145/321992.321996

Chakravarty, Manuel M. T., & Keller, Gabriele. (2001). Functional array fusion. *Proc. 6th ACM Int. Conf. on Functional Programming, SIGPLAN Not.*, **36**(10), 205–216. http://doi.acm.org/10.1145/507635.507661

Chakravarty, Manuel M. T., & Keller, Gabriele. (2003). An approach to fast arrays in Haskell. *Pages 27–58 of:* Jeuring, Johan, & Peyton Jones, Simon (eds), *Advanced Functional Programming*. Lecture Notes in Comput. Sci., vol. 2638. Heidelberg: Springer. http://www.springerlink.com/link.asp?id=93amy8yr5m49kd3u

Chen, Chih-Ping, & Hudak, Paul. (1997). Rolling your own mutable ADT—a connection between linear types and monads. *Pages 54–66 of: Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. New York: ACM Press. http://doi.acm.org/10.1145/263699.263708

Dybvig, R. Kent. (1996). *The Scheme Programming Language: ANSI Scheme*. Second edn. Upper Saddle River, NJ: Prentice Hall. http://www.scheme.com/tspl2d/

Elmroth, Erik, Gustavson, Fred, Jonsson, Isak, & Kågstroöm, Bo. (2004). Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev.*, **46**(1), 3–45. http://epubs.siam.org/sam-bin/dbq/article/42869

Frens, Jeremy D., & Wise, David S. (1997). 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. http://doi.acm.org/10.1145/263764.263789

Hall, Cordelia V., Hammond, Kevin, Peyton Jones, Simon L., & Wadler, Philip L. (1996). Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, **18**(2), 109–138. http://doi.acm.org/10.1145/227699.227700

Hudak, Paul. (1986). A semantic model of reference counting and its abstraction (detailed summary). *Pages 351–363 of: Proc. 1986 ACM Conf. on LISP and Functional Programming.* New York: ACM Press. http://doi.acm.org/10.1145/319838.319876

Johnson, David S. (2002). A theoretician's guide to the experimental analysis of algorithms. *Pages 215–250 of:* Goldwasser, M. H., Johnson, D. S., & McGeoch, C. C. (eds), *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges.* DIMACS Ser. Discrete Math. Theoret. Comput. Sci., vol. 59. Providence: American Mathematical Society. http://www.research.att.com/~dsj/papers.html

Launchbury, John, & Peyton Jones, Simon L. (1994). Lazy functional state threads. *Proc. ACM SIGPLAN '94 Conf. on Program. Language Design and Implementation, SIGPLAN Not.*, **29**(6), 24–35. http://doi.acm.org/10.1145/178243.178246

Lynagh, Ian. 2003 (May). *Template Haskell: A Report From The Field.* http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/#reportfromfield

Peyton Jones, Simon L., & Wadler, Philip. (1993). Imperative functional programming. *Pages 71–84 of: Proc. 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages.* New York: ACM Press. http://doi.acm.org/10.1145/158511.158524

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

Wise, David. (1987). Matrix Algebra and Applicative Programming. *Pages 134–153 of:* Kahn, Gilles (ed), *Programming Languages and Computer Architecture.* Lecture Notes in Comput. Sci., vol. 274. Berlin: Springer. http://www.cs.indiana.edu/~dswise/FPCA87.html

Wise, David S. (1999). Undulant block elimination and integer-preserving matrix inversion. *Sci. Comput. Program.*, **33**(1), 29–85. http://dx.doi.org/10.1016/S0167-6423(98)00005-7

Wise, David S. (2000). Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. *Pages 774–883 of:* Bode, Arndt, Ludwig, Thomas, Karl, Wolfgang, & Wismüller, Roland (eds), *Euro-Par 2000 – Parallel Processing.* Lecture Notes in Comput. Sci., vol. 1900. Heidelberg: Springer. http://www.springerlink.com/link.asp?id=0pc0e9gfk4x9j5fa

Wise, David S., Frens, Jeremy D., Gu, Yuhong, & Alexander, Gregory A. (2001). Language support for Morton-order matrices. *Proc. 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.*, **36**(7), 24–33. http://doi.acm.org/10.1145/379539.379559