The Opie Compiler from Row-major Source to Morton-ordered Matrices[*]

Steven T. Gabriel[†]
Computer Science Dept.
Indiana University
Bloomington, IN 47405–7104, USA
stgabrie@cs.indiana.edu

David S. Wise[‡]
Computer Science Dept.
Indiana University
Bloomington, IN 47405–7104, USA
dswise@cs.indiana.edu

ABSTRACT

The Opie Project aims to develop a compiler to transform C codes written for row-major matrix representation into equivalent codes for Morton-order matrix representation, and to apply its techniques to other languages. Accepting a possible reduction in performance we seek to compile a library of usable code to support future development of new algorithms better suited to Morton-ordered matrices.

This paper reports the formalism behind the Opie compiler for C, its status: now compiling several standard Level-2 and Level-3 linear algebra operations, and a demonstration of a breakthrough reflected in a huge reduction of L1, L2, TLB misses. Overall perforamnce improves on the Intel Xeon architecture.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, preprocessors.*; E.1 [**Data Structures**]: Arrays.; E.2 [**Data Storage Representations**]: contiguous representations.; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed and parallel languages; applicative (functional) languages.*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical algorithms and problems—*computations on matrices.*

[*]Supported, in part, by the National Science Foundation under a grant numbered EIA–0202048.
©Association for Computing Machinery (2005). This is the authors' 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 *Proc. 3rd Workshop on Memory Performance Issues (WMPI-2004).*
[†]Supported, in part, by the National Science Foundation under a grant numbered CCR–0073491.
[‡]Supported, in part, by the National Science Foundation under grants numbered CCR–0073491 and ACI–0219884.

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.
WMPI '04, Munich Germany
Copyright 2004 ACM 1–59593–040–X ...$5.00.

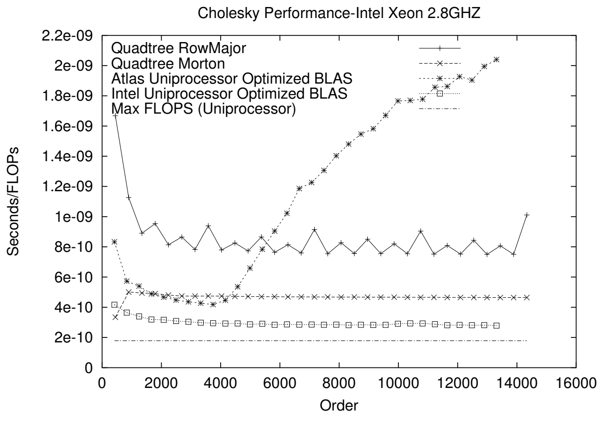

Figure 1: Performance of block-recursive Cholesky on Morton-order representation.

General Terms

Design, Languages, Performance

Keywords

Cache, Paging, Quadtrees, Scientific Computing

1. INTRODUCTION

Traditionally matrices are laid out in memory as rows or columns forming a row- or column-major ordering respectively. This corresponds to a natural understanding of a matrix however it does not do a good job of taking into account properties of the memory hierarchy in modern computers. For reasons related to memory usage, most modern matrix algorithms do not operate over rows or columns but rather over matrix subblocks. With row- or column-major matrices any given matrix subblock from a large matrix will contain several disjoint portions of rows or columns and will be likely to contain data residing in several pages of memory. This property of traditional matrix layout schemas provides inherent difficulties with locality that cause contention for, and inefficient use of, the computer's memory hierarchy. A solution for this is to rearrange the matrix layout such that subblocks are actually contained in contiguous memory. This paper discusses Morton-ordering, a recursive ordering of matrix elements that accomplishes this goal.

As a sample of the impact of the inherent locality of Morton-ordering, Figure 1 exhibits the performance from this representation . This plot shows the leading coefficient from the asymptoticly cubic time of Cholesky factorizations of dense SIMD matrices whose orders ranging from 500 to 14,000. These times are derived from algorithms written entirely in C and running without hyperthreading on a single 2.8GHz Intel Xeon. The algorithm uses Morton-ordering and a block-recursive style of programming that utilizes the recursive layout of Morton-ordering that will be discussed in this paper. Flat lines indicate algorithms that scale well; Atlas's `dpotrf` does not [23]. Intel's assembly-coded BLAS `dpotf` is fast, indeed [16, 17], but within reach of our pure-C code that uses Morton-order for locality. The figure shows big improvements from using block recursion and Morton ordering together; improvements from Morton order alone appear later in this paper. The block-recursive algorithm, as well as its supporting paradigm, is reported in another paper [25].

Morton ordering of matrices is a single representation that simultaneously supports three different indexing schemes for both dense and sparse matrices. (Figure 3). Tersely introduced here, MORTON ORDER is used to lay out the elements of a matrix in any address space. AHNENTAFEL INDICES are used as control variables in block-recursive algorithms, while conventional CARTESIAN INDEXING is yet available for local access within base blocks, or globally from classic codes.

The block-recursive programming style mentioned above follows the recursive layout of the Morton-ordered code to full effect. However it is possible to use the Cartesian indexing scheme to translate code written with indexing for a row- and column-major matrix into code that will operate over an isomorphic Morton-ordered matrix. With this it is possible circumvent some of the difficulty of switching to a new mindset in writing matrix codes for a Morton-order style of memory layout.

This paper discusses the OPIE compiler which uses these translation methods to convert code written for row- and column- major matrices into code that executes on Morton-order matrices. The OPIE compiler has three goals: first to make existing program libraries available for Morton-ordered matrices, second to erode social resistance to change by presenting users' code already running on Morton-ordered codes, and third to use an automated method for translating codes as a platform from which to demonstrate the success of this technique in yielding performance gains.[1]

This paper is organized into six sections. This introduction is followed by one on basic definitions of Morton order and the algebra of dilated integers. The third section discusses reinterpretation of matrix indices, which is used by the OPIE compiler, described in the fourth. The fifth presents performance results and the last reviews related literature and foresees future work.

2. DEFINITIONS

DEFINITION 1. *The base of a matrix has* Morton-order *index* 0. *A submatrix (block) at Morton-order index i is either an element (scalar), or it is composed of* 4 *submatrices, with indices* $4i+0, 4i+1, 4i+2, 4i+3$. [24].

[1]The Opie project, whose goal is to transform source code, was named by Greg Alexander after Optimus Prime. O.P. was, of course, the most powerful Transformer^{TM} the world has ever known.

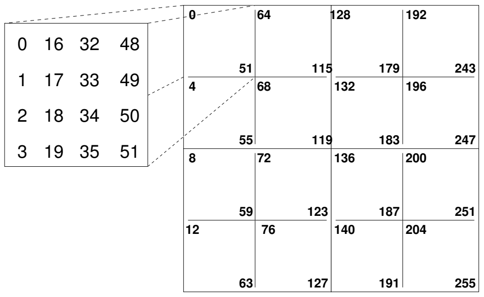

Figure 2: Column-major indexing of a 16×16 matrix.

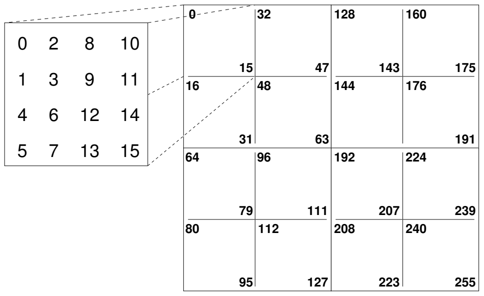

Figure 3: Morton-order indexing of a 16×16 matrix.

In И order in Figure 4 the four submatrices are oriented northwest, southwest, northeast, and southeast, respectively.

DEFINITION 2. *A complete matrix has* Ahnentafel *index* 3. *A submatrix (block) at Ahnentafel index i is either a scalar, or it is composed of* 4 *submatrices, with indices* $4i+0, 4i+1, 4i+2, 4i+3$ [24, 8].

Conversion between these types is free at run time: subtract $(4^\ell-1)/3$ at level ℓ. Both definitions share the important properties, illustrated in Figure 3, that all indices increase monotonically across rows or columns, and that the nested blocks of size 4^p at address $k4^p$ are accessed by 4^p consecutive indices for all p. These blocks, therefore, are the ones used in decomposing matrix problems because their elements share optimum locality with one another.

THEOREM 1. *The Morton indices on the elements of a matrix, or Ahnentafel indices on blocks of any single size, increase monotonically to the east and south.*
So bounds checking is easy with either indexing.

NOTATION 1. *When subscripted, variables* i, j *denote bits in a binary representation of an integer.*

NOTATION 2. *A quaternary digit,* q, *in Definitions 1, 2 selects a northwest/southwest/northeast/southeast quadrant.*

NOTATION 3. *Let* w *number the bits of a* `short` *integer.*

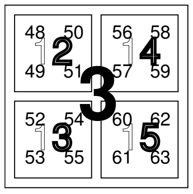

Figure 4: Ahnentafel indexing of the order-4 quadtree.

THEOREM 2. [18] *The Morton index* $x = \sum_{k=0}^{w-1} q_k 4^k = \sum_{k=0}^{w-1} i_k 4^k + 2\sum_{k=0}^{w-1} j_k 4^k$ *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 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 [18]. Masking a Morton index with `0x55555555` or `0xaaaaaaaa` extracts the bits of the row and column cartesian indices, introduced next as *dilated integers*.

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

These basic properties of Morton ordering have been independently reintroduced over the years [22, 18, 19, 13]. Samet gives an excellent history [20]. The elegant, additive algebra of dilated integers is surprisingly old [22, 21, 24]. Cartesian indices are represented as dilated integers (with information stored only in every other bit). Their algebra is really a classic type [24, 26]. The JAVA or C++ programmer should envision it as a class, whose methods each reduce to a couple of processor cycles. The C programmer will implement it as a library of `#define`d macros.

DEFINITION 3. *The* even-dilated representation *of* $x = \sum_{k=0}^{w-1} i_k 2^k$ *is* $\sum_{k=0}^{w-1} i_k 2^{2k}$, *denoted* $\overrightarrow{x}$. *The* odd-dilated representation *of* $y = \sum_{k=0}^{w-1} j_k 2^k$ *is* $\sum_{k=0}^{w-1} j_k 2^{2k+1}$, *denoted* $\overleftarrow{y}$. [24].

Theorem 3 gives evidence of the fast, additive methods available in this class. Let the infix operators $\overset{\rightarrow}{+}$, $\overset{\rightarrow}{-}$, and $\overset{\rightarrow}{\cdot}$, for example, be the addition, subtraction, and multiplication methods of the class of even-delated integers.

THEOREM 3. *Register-local algorithms for subtraction, addition, constant-addition, and the rare multiplication of dilated integers* [21, 24]*:*

$$(\overrightarrow{x} - \overrightarrow{z})\,\&\,\overrightarrow{b} = \overrightarrow{x} \overset{\rightarrow}{-} \overrightarrow{z}; \quad \overleftarrow{y} \overset{\leftarrow}{-} \overleftarrow{z} = (\overleftarrow{y} - \overleftarrow{z})\,\&\,\overleftarrow{b};$$

$$(\overrightarrow{x} + \overrightarrow{z} + \overleftarrow{b})\,\&\,\overrightarrow{b} = \overrightarrow{x} \overset{\rightarrow}{+} \overrightarrow{z}; \quad \overleftarrow{y} \overset{\leftarrow}{+} \overleftarrow{z} = (\overleftarrow{y} + \overleftarrow{z} + \overrightarrow{b})\,\&\,\overleftarrow{b};$$

$$(\overrightarrow{x} - \overrightarrow{(-c)})\,\&\,\overrightarrow{b} = \overrightarrow{x} \overset{\rightarrow}{+} \overrightarrow{c}; \quad \overleftarrow{y} \overset{\leftarrow}{+} \overleftarrow{c} = (\overleftarrow{y} - \overleftarrow{(-c)})\,\&\,\overleftarrow{b};$$

$$\overrightarrow{x} \overset{\rightarrow}{\cdot} \overrightarrow{y} = (\sum_{k=0}^{w-1} i_k 4^k) \overset{\rightarrow}{\cdot} \overrightarrow{y} = \overrightarrow{\textstyle\sum_{k=0}^{w-1}} i_k(\overrightarrow{y} \ll (2k)).$$

Multiplication is almost never necessary because dilated integers shadow cartesian indices whose products occur mostly inside loops, where strength reduction compiles them to additions. [Note added in proof: Unfortunately, the Opie transformation occurs in the front end, before the C compiler can reduce them.]

The largest index into the vector containing a zero-based Morton $m \times n$ matrix is $s = \overrightarrow{m-1} + \overleftarrow{n-1}$. The matrix occupies $s + 1$ consecutive locations, not all of which are within bounds and, when touched, migrate from slow, cheap memory to fast, dear cache. Figure 5 distinguishes between address space and active space in a sample of the worst cases.

It is quite important for bounds checking that Morton/Ahnentafel indexing is monotonic across any row or column; contrast [2]. The value of s suffices as a coarse, improper bound on the entire matrix, but its even (respectively, odd) bits finely bound those in the dilation of any cartesian row (column) index. Moreover, adding two high-order bits to s yields similarly fine bounds on Ahnentafel indices.

3. REINTERPRETING MATRIX INDICES

Although Morton indices seem to increment wildly through a matrix, the path they trace is not at all crazy. Each index contains its cartesian indices just beneath its surface, a fact that allows OPIE's transform to work. It allows the direct translation of row- or column-major codes into codes that instead use dilated integers. This section describes a useful isomorphism that has been used in our compiler for automatic translation of such codes. This isomorphism is presented in terms of row-major representations although it generalizes to column-major. (The role of odd and even dilations is simply reversed.)

Typically a matrix is dereferenced by an vector-index value that is the result of several other computations. Individual integers used to dereference into a matrix often play identifiable roles. Some refer to a specific row or to a column of the matrix, or to the stride of the matrix. Others may refer to a combination of row and column indices, such as a block reference that is the sum of a row index and the product of a column index with the stride. When dilated integers are used to index into a Morton-ordered matrix, these roles are more strongly reflected. Even-dilated integers always refer to a row and odd-dilated integers always refer to a column. In order to convert from cartesian indices we distinguish integer values that are involved exclusively in row indexing from those that are involved in column indexing from those that play both roles. These are mapped, respectively, into even-, and odd-, and even-dilated integer values. Definition 3 yields the simple casts between the two dilations.

Four relevant types of integers are treated as if they were types in C or C++

- `int`: An integer value in the traditional sense. With respect to any matrix it may contain a row or a column index or some combination of these.
- `oddDilatedInt`: An odd-dilated integer semantically is a column index to a matrix.
- `evenDilatedInt`: An even-dilated integer semantically is a row index to a Morton-ordered matrix.
- `mixedDilatedInt`: This might also be called a Morton index. It contains information from both row and column indices. Similarly to the `int` type, it may contain both row and column data although in this case, the two are easily separated (by masking off the odd or even bits). It is also the type used to index an element in a Morton-ordered matrix.

Next, define methods for casting between these types, using traditional integer operations. For `e`, `o`, and `m` of `evenDilatedInt`, `oddDilatedInt` and `mixedDilatedInt` types respectively

```
  (oddDilatedInt) e  =  e << 1;
(mixedDilatedInt) e  =  e;
 (evenDilatedInt) o  =  o >> 1;
(mixedDilatedInt) o  =  o;
  (oddDilatedInt) m  =  m & 0xaaaaaaaa;
 (evenDilatedInt) m  =  m & 0x55555555.
```

The last two casts involve some loss of information. The addition of an `evenDilatedInt` and an `oddDilatedInt` has type `mixedDilatedInt`. Compare Theorem 2; this is how row and column indices are combined to form a Morton index. More computation is involved for casting between the `int` type and the the even- and odd-dilated integer types. Through a lookup table or in some other manner, functions `oddDilate`, `evenDilate`, `oddUndilate` and `evenUndilate` can be provided such that for `r`, an `int`:

$$\begin{aligned}
\texttt{(int) e} &= \texttt{evenUndilate(e)};\\
\texttt{(int) o} &= \texttt{oddUndilate(o)};\\
\texttt{(evenDilatedInt) r} &= \texttt{evenDilate(r)};\\
\texttt{(oddDilatedInt) r} &= \texttt{oddDilate(r)}.
\end{aligned}$$

Conversion between the `int` and `mixedDilatedInt` types is trickier. The `mixedDilatedInt` is intrinsically a combination of two indices. To convert to an isomorphic `int` index more information about the matrix that the value indexes is needed: the stride for that matrix. If that is available then casting between these two types can be done

$$\begin{aligned}
\texttt{(mixedDilatedInt)} \quad & \texttt{r}\\
&= \texttt{oddDilate(r\%stride)}\\
&+ \texttt{evenDilate(r} \div \texttt{stride)};\\
\texttt{(int) m} &= \texttt{evenUndilate((evenDilatedInt) m)}\\
&+ \texttt{oddUndilate((oddDilatedInt) m)}.
\end{aligned}$$

This last cast is expensive in real time. For each portion of a code where a matrix `a` with stride `aStride` is dereferenced by an `int` index this `int` might be cast in place into a `mixedDilatedInt`. This translation is inefficient, however, as it involves significant computation to dilate an integer. It might still be used where there are no alternatives, or where it can be amortized against many cycles to follow, such as at a function call.

Our compiler instead attempts to identify row and column indexing in the original code, and then add additional variables and arithmetic to construct corresponding dilated integers. These replace the computation needed for in-place dilation with cheap integer arithmetic from Theorem 3.

It is necessary to construct an algorithm that, given stride information, attempts to dissect an integer expression into row- and column-indexing portions. This is done for general expressions relative to a declared stride using the following recursive algorithm. The result of the algorithm is a pair of expressions for row and column indexing as the respective elements. The algorithm recurs over the parse tree of the integer expression and at the lowest level of recursion it generates the mapping:

$$\begin{aligned}
\mathit{var}\ v &\mapsto \langle v, 0\rangle \text{ when } v \text{ is not stride};\\
\mathit{var}\ v &\mapsto \langle 0, 1\rangle \text{ when } v \text{ is equal to stride};\\
\mathit{constant}\ c &\mapsto \langle c, 0\rangle.
\end{aligned}$$

At higher levels it distributes over several arithmetic operators:

$$\begin{aligned}
\langle x_1, x_2\rangle + \langle y_1, y_2\rangle &\mapsto \langle x_1 + y_1, x_2 + y_2\rangle\\
\langle x_1, x_2\rangle \& \langle y_1, y_2\rangle &\mapsto \langle x_1 \& y_1, x_2 \& y_2\rangle\\
\langle x_1, x_2\rangle | \langle y_1, y_2\rangle &\mapsto \langle x_1 | y_1, x_2 | y_2\rangle\\
\langle x_1, 0\rangle * \langle 0, 1\rangle &\mapsto \langle x_1, 0\rangle\\
\langle 0, 1\rangle * \langle x_1, 0\rangle &\mapsto \langle x_1, 0\rangle\\
\langle x_1, x_2\rangle * \langle y_1, y_2\rangle &\mapsto \langle x_1 * y_1,\\
&\qquad x_2 * y_2 * \mathit{stride} + x_1 * y_2 + x_2 * y_1\rangle
\end{aligned}$$

Any expression that was not recognized by any of the rules above is handled by the inelegant cast:
$\mathit{expression}\ e \mapsto \langle e\%\mathit{stride}, e/\mathit{stride}\rangle$

This corresponds to our casting rule for casting an `int` to a `mixedDilatedInt`.

The expressions within these pairs are still of type `int` and all relevant algebraic properties (commutativity, identity, annihilation, *etc.*) still apply. For an initial expression *expr* and its pair $\langle \mathit{expr}_1, \mathit{expr}_2\rangle$ resulting from the algorithm above, we have $\mathit{expr} = \mathit{expr}_1 + \mathit{expr}_2 * \mathit{stride}$, verifying that no information was lost; it is only moved around. This algorithm also incorrectly handles variables that contain both column and row indexing. The problem can be caught in most instances with an analysis of all all side-effects, noticing any that modify a variable by a multiple of stride.

4. COMPILING TO MORTON ORDER

This section describes the Opie compiler, which has been implemented using the ideas in Section 3 to convert existing row-major or column-major codes into Morton order codes. The compiler is implemented in Scheme using the Edison Design Group's C/C++ parsing engine as a front end [1]. It is a C to C compiler, generating code that will then be compiled by the machine-specific C compiler.

4.1 Shadowing Matrix Indices

The compiler operates by identifying row and column indexes in a traditional code and shadowing these with `evenDilatedInt` and `oddDilatedInt` values. Our C-to-C transformer operates only on syntax and still depends on the ordinary C optimizer as a back end.

One possible optimization would be to displace the original integer arithmetic leaving only shadows behind. The original design anticipated such a reduction but experience with early prototypes exposed a severe problem: conventional optimizers cannot optimize loops that use dilated integers for control, because they are yet unenlightened about the simple algebra of dilated integers. Without knowledge of what the `+` and `&` operators are doing, the optimizer could not alter any control that depended on dilated integers. Therefore the underlying integers are retained for iteration counts and bounds checking, and are augmented with shadows for dereferencing arrays.

All side-effects on the original values are also shadowed in the arithmetic of dilated integers. As an example of this process, consider this matrix multiplication:

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

The variables `i`, `j` and `k` are involved in matrix dereferences. With prior identification of variables `cStride, aStride, bStride` as strides, it can be deduced that `k` is being used as a row index for referencing `c` and similarly that `i` is being used as a column index. Similar deductions are performed for the indexing of `a` and `b`. As a result, `i`, `j`, and `k` are all shadowed and dilated assignments and increments are inserted to match the assignments and increments on the original variables. The resulting code in this case is:

```
int i,j,k;
evenDilatedInt i_even;
oddDilatedInt j_odd, k_odd;
for(    i=0, i_even=0; i<m; i++, evenInc(i_even))
  for(  j=0,  j_odd=0; j<n; j++,  oddInc(j_odd))
    for(k=0,  k_odd=0; k<p; k++,  oddInc(k_odd))
      c[i_even+k_odd] +=        a[i_even+j_odd]
            * b[((evenDilatedInt) j_odd)+k_odd];
```

The compiled code that works when the original code works, as long as `a`, `b` and `c` aree pointers to proper Morton matrices.

Information matching strides to matrices is given by the programmer in the form of a pragma

```
#pragma mortonMatrix(c,cStride).
```

When all variables are shadowed, we apply the algorithm from Section 3 to each matrix dereference and to the right hand side of each assignment between shadowed variables. With this information, all matrix dereferences can be converted to the addition of dilated integers, and all side-effects can be shadowed so that they also occur for all shadowing dilated integers (this process is covered in the next subsection).

The effect is reduction of the multiplication of a cartesian column index by its stride to the accumulation of a shadowing odd-dilated index, just as any C compiler uses strength reduction within a loop to reduce that multiplication to repeated addition [3].

The characterization of traditional matrix indexing and Morton indexing here and the resulting compiler are more robust than prior efforts. In addition the compiler has been expanded to cover several additional ways of expressing matrix indexing, most of which stem from possible consequences of assignment which is covered in the next section.

4.2 Handling Assignments

When an assignment is encountered with the left-hand side an integer that is to be shadowed, the entire assignment must be shadowed. The right-hand side must be translated into an even or odd value. The algorithm for translating matrix indices is reused here. However it is also important that this shadowing of the assignment may in turn lead to shadowing of new variables. For example, consider a snippet of code:

```
i = n;
A[i+j*stride] = 5;
```

If `A` is a matrix being converted to Morton order, then it will be necessary to shadow the three variables: `i, j, n`. Due to the fact that `i`'s value depends upon it, `n` must be shadowed. This may spark a whole chain of further shadowing.

Thus far, this discussion has been primarily concerned with variables that are used in traditional indexing. That is, each integer index is used specifically as a row or column index and will be separate from a stride (and indexing a column will always require multiplying by stride). Indices used in such a manner will be called "simple". It is not uncommon, however, for a programmer to mix notions of indices together and to create variables that refer not to a row or a column but rather a specific location in a matrix. Most instances of this programming style will be expressed in terms of an assignment. For example, consider the following assignment:

```
i = i + stride;
```

It is possible to use the existing framework presented here to capture this notion and understand that `i` should be seen now as an explicitly `even` value. Handling such a variable will require adding a new base case to our translation algorithm:

$$\mathit{var}\ v \mapsto \langle 0, v\rangle \text{ when } v \text{ is explicitly even}$$

Furthermore, the right-hand side of an assignment may contain values whose roles are partially even and partially odd. An example of this would be the following algorithm for setting a matrix diagonal to all 1:

```
i = 0;
while(j<n) {
  A[i] = 1;
  i += stride + 1;
  j++;
}
```

This situation is handled by treating `i` as a mixed index. One simple method for implementing this is to create an even and odd shadow of `i` and to then shadow assignments to these twice. Using this methodology, the preceding algorithm could be compiled to:

```
i_odd = 0;
i_even = 0;
while(j<n) {
  A[i_even+i_odd] = 1;
  i_even = evenAdd(i_even,1);
  i_odd  = oddAdd(i_odd,1);
  j++;
}
```

The current version of the compiler partially supports usage of such mixed indices. In practice it is very difficult to support such programming techniques without obtaining more information from the programmer. For example, a variable may be used in an explicitly even manner for part of a function and may then be reused later as a simple index. One way to allow stronger guarantees about the compiler's correctness for such usage patterns would be to require the programmer explicitly to type all integers that are involved in matrix dereference calculations (as simple or mixed, odd or even).

4.3 Linking Morton Functions

Shadowing of variables leads to the shadowing of function parameters. This requires sending the correct value from the caller to the callee. If the compiler knew that all incoming nonlocals and arguments were simple variables then this is not so troublesome: a copy of each variable can be made immediately upon entering the function. With the possibility of mixed variables, however, this conversion becomes uncertain. The more general solution for this, available through the compiler, is that parameter lists for functions may be extended to require shadowed copies of variables. Information about the function signatures is stored during compilation

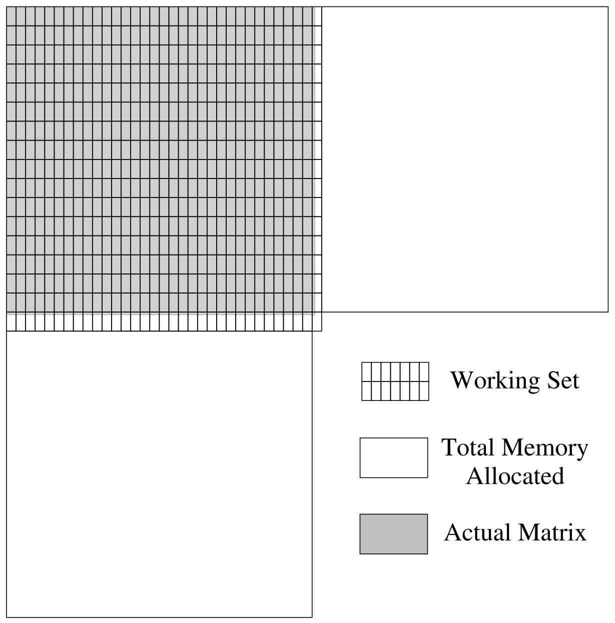

Figure 5: A 1025×1025 Morton-ordered matrix of `doubles` in 16kB pages, each 64×32. The address space spans 1537 pages, of which only 561 are used.

and can be used to "link" files at a later point, updating function calls from each call site to match the new signature. With stronger indexing typing, it would be possible to remove the need for this and use the typing information to generate a conversion of the original variable at entry into the function.

5. PERFORMANCE RESULTS

The Opie compiler was tested with several "standard" Level-2 and Level-3 linear-algebra operations, as defined by Golub and Van Loan [15]. A test bed has been built to launch these operations generating correctness tests and timing `double`-precision data. The compiler was used to generate a new testing apparatus and new versions of these algorithms over Morton-order matrices.

The only change to usual testing apparatus is that memory allocation was modified to account for larger address-space required for a Morton-order matrix. While Morton-order matrices require more address space, most of the allocated pages will never be used and will exist only in virtual-memory tables, never as data i cache. The increase in the working in actual working set of the matrix is only linear in the matrix order as Figure 5 demonstrates

The resulting codes were tested on two platforms, a 2.8GHz Intel Xeon, with 2GB of memory and 512KB cache, and a 195MHz SGI Octane, an R10000 with 32KB each of L1 Icache and Dcache, 1MB L2 cache, and only 128MB RAM. The latter was used n its memory hierarchy, as well as its performance-tracking counters (in hardware) to get data on caching and TLB performance. All files were compiled using the native C compilers with the `--O3` optimization flag.

All times here are presented in units of seconds-per-FLOP, with a flat plot indicating good scaling. Morton order yields flat plots. An immediate observation is that all resources (time and cache-misses) for Morton order tend to be much smoother than that for row-major source, and even that for manufacturers' BLAS routines. We have seen this pattern repeatedly, suggesting better predictability here of both conventional, looping codes and later for new, recursive codes on this data structure.

Figures 6–9 show results using performance counters on the Octane for an implementation of blocked multiplication. Using Morton order for a row/column algorithm, a dramatic decrease in TLB misses—as well as in Level-1 and Level-2 cache misses—already anticipates the locality from algorithms that better use block-recursion [26, 14]. These results further corroborate the paradigm [25], even though the time shows a significant slowdown. The performance gains in using the memory hierarchy are not sufficient to overcome the effects of additional integer operations and the implicit handicap from a C optimizer designed for row/column traversal.

Our tests on the Xeon do show significant performance gains in Figures 10–13. For these implementations we use algorithms straight out of Golub and Van Loan [15]. `gaxpy` is an implementation of their Algorithm 1.1.4. Outer product is an implmentation of the one of the first algorithm in their Section 1.1.9. Our first multiplication algorithm is an implementation of Algorithm 1.1.5 and the blocked version is Equation 1.3.3. Gaussian is an implementation of Algorithm 3.2.1. The compiler has been successfully used for other versions of these algorithms as well as several other Level-2 and Level-3 linear-algebra algorithms.

While the performance gains on the Xeon are impressive, it is important that the overall performance still pales beside hand-coded and manufacturer's versions of these codes. In order to match and surpass these codes it will be necessary to take these ideas further and to implement hand-tuned versions of each algorithm.

6. CONCLUSIONS

6.1 Related Work

There are many contemporary efforts to improve performance of scientific codes running on hierarchical and shared memories [9]. Most retain the problems of row- and column-major ordering: as matrices grow, locality crumbles at one level or another in the hierarchy.

Two groups have experimented with decompositions of matrices that are close to the quadtrees implicit in Morton order. Chatterjee *et al.* adopted a hybrid representation tailored to fit base cases to BLAS3 `dgemm` as a primitive operation, in order to take advantage of BLAS3 codes on row-major base cases [6, 7], but using our recursive codes higher in the tree [13, 26]. Using Morton order there, they obtained good times but their cost to change between representations was high. (In spite of his implications to the contrary [6], we had always advocated the homogeneous representation that is also our target here.)

Second, Gustavson's work addresses the halving of quadtrees, but he does not enforce powers of two [10]. Rather, he cleaves matrices into roughly equal quarters, obtaining balanced subproblems but, again, not the advantages of dilated integers and bounds checking. More recently he introduced *recursive packed storage* that still uses column-major base blocks, presumably for their hand-written codes [4, 11].

The ATLAS project is also aimed at compiling optimal block sizes into classic programming style [23], so that blocks nicely fill L1 cache, but it addresses only a single level of caching. L2 cache or page reuse is not included except through disjoint analysis. Related work in the PHiPAC project does address multiple levels of blocking, but only the register file and L1 cache are targeted [5]. 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. Pingali experimented with translation of iterative codes to blockwise traversal, but the matrix representation is unchanged and he ordered the blocks in a manner that does not suit simple bounds control [2].

6.2 Past and Future Work

Our purpose is *not* to recompile all codes for high-performance use. Indeed, our transformation may make some codes run slower and will not beat hand-tuned codes. It is to ease the path for programmers to migrate from iterative row-major programs to block-recursive quadtree algorithms, just as they migrated their FORTRAN codes to C [12]. We want all their libraries to run under Morton order, so that they can reprogram incrementally, and so that our demonstrated gains from superlative locality can be compared and appreciated. And we want to realize future performance possibilities, like hyperthreading for Figure 1.

The OPIE compiler opens a path to Morton-ordered matrices from C on row-major and, by transitivity, from FORTRAN on column-major The compiler also demonstrates significant merit for using Morton-ording. Using an automated algorithm for translating from one representation to another, and no hand-coding, we do see significant performance gains in the form of improved timings on the Xeon and better cache and TLB usage on the Octane.

The OPIE compiler is available for download from `http://www.cs.indiana.edu/~dswise/Opie/distribution.html`.

Two themes continue: first to extend the OPIE compiler to work on more arbitrary styles of C code and to extend these techniques to other source languages. The second theme is the ARCEE project to develop the paradigm of programming using quadtree recursion, to take advantage of all the locality and parallelism that this representation allows.

7. ACKNOWLEDGEMENT

We thank the Edison Design Group for allowing use of its front end for C [1]. Figure 1 was built using tools tools by Craig Citro and Mike Rainey. The second author thanks the folks at CS&E Department at the University of Washington for their warm hospitality.

8. REFERENCES

[1] ADAMCZYK, S., SPICER, J., AND VANDEVOORDE, D. Edison Design Group: Compiler front ends for the OEM market, 2002. Visited Nov. 2002. http://www.edg.com/

[2] AHMED, N., AND PINGALI, K. Automatic generation of block-recursive codes. 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. 368–378. http://springerlink.metapress.com/link.asp?id=1lv21jg7k16mj591

[3] 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.

[4] ANDERSEN, B. S., WAŚNIEWSKI, J., AND GUSTAVSON, F. G. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Trans. Math. Softw. 27*, 2 (June 2001), 214–244. http://doi.acm.org/10.1145/383738.383741

[5] BILMES, J., ASANOVIĆ, K., CHIN, C.-W., AND DEMMEL, J. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proc. 1997 Int. Conf. Supercomputing.* ACM Press, New York, July 1997, pp. 340–347. http://doi.acm.org/10.1145/263580.263662

[6] CHATTERJEE, S., JAIN, V. V., LEBECK, A. R., MUNDHRA, S., AND THOTTETHODI, M. Nonlinear array layouts for hierarchical memory systems. In *Proc. 13th Int. Conf. Supercomputing.* ACM Press, New York, June 1999, pp. 444–453. http://doi.acm.org/10.1145/305138.305231

[7] 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://www.computer.org/tpds/td2002/l1105abs.htm

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

[9] CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, K. E., SANTOS, E., SUBRAMONIAN, R., AND VON EICKEN, T. LogP: a practical model of parallel computation. *Commun. ACM 39*, 11 (Nov. 1996), 78–85. http://doi.acm.org/10.1145/240455.240477

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

[11] ELMROTH, E., GUSTAVSON, F., JONSSON, I., AND KGSTRÖM, B. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev. 46*, 1 (Mar. 2004), 3–45. http://epubs.siam.org/sam-bin/dbq/article/42869

[12] FELDMAN, S. I., GAY, D. M., MAIMONE, M. W., AND SCHRYER, N. L. Availability of `f2c`—a FORTRAN to C converter. *SIGPLAN Fortran Forum 10*, 2 (July 1991), 14–15. http://doi.acm.org/10.1145/122006.122007

[13] Frens, J. D., and Wise, D. S. 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 (July 1997), 206–216.
`http://doi.acm.org/10.1145/263764.263789`

[14] Frens, J. D., and Wise, D. S. QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism. *Proc. 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program, SIGPLAN Not. 38*, 10 (Oct. 2003), 144–154.
`http://doi.acm.org/10.1145/781498.781525`

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

[16] Goto, K., and van de Geijn, R. On reducing TLB misses in matrix multiplication. FLAME Working Note 9, Univ. of Texas, Austin, Nov. 2002.
`http://www.cs.utexas.edu/users/flame/pubs/GOTO.ps.gz`

[17] Intel Corp. *Intel Math Kernel Library.* Santa Clara, CA, 2003. `http://www.intel.com/software/products/mkl/`

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

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

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

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

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

[23] Whaley, R. C., and Dongarra, J. J. Automatically tuned linear algebra software. In *Proc. Supercomputing '98*. IEEE Computer Soc. Press, Washington, DC, 1998, pp. 1–27.
`http://portal.acm.org/citation.cfm?id=509096`

[24] 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`

[25] Wise, D. S., Citro, C., and Rainey, M. Parallel programming with Morton-ordered matrices. In preparation, 2003.

[26] Wise, D. S., Frens, J. D., Gu, Y., and Alexander, G. A. Language support for Morton-order matrices. *Proc. 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not. 36*, 7 (July 2001), 24–33.
`http://doi.acm.org/10.1145/379539.379559`

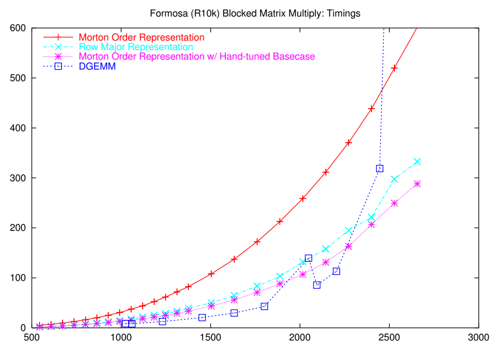

Figure 6: Performance of compiled, blocked matrix multiplication on SGI Octane.

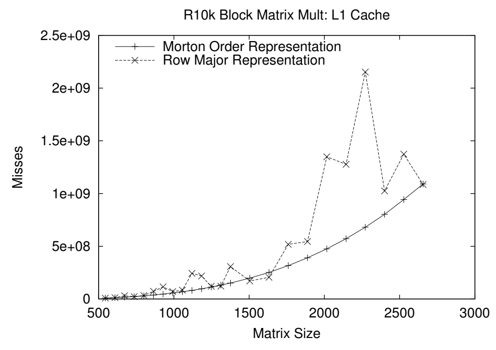

Figure 7: L1-cache misses from compiled, blocked matrix multiplication on SGI Octane.

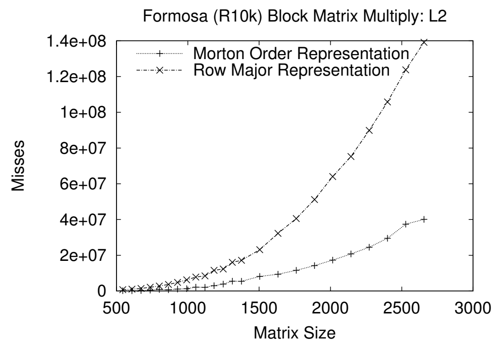

Figure 8: L2-cache misses from compiled, blocked matrix multiplication on SGI Octane.

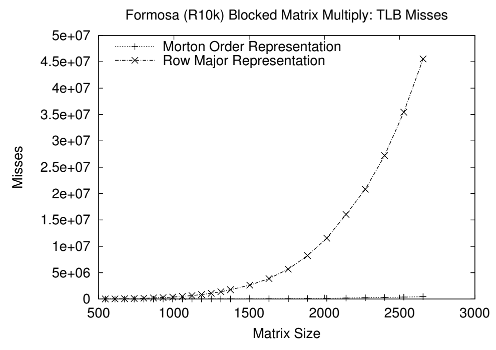

Figure 9: TLB misses from compiled, blocked matrix multiplication on SGI Octane.

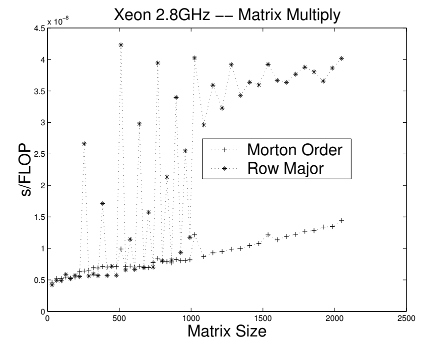

Figure 10: Performance of compiled, unblocked matrix multiplication on Intel Xeon.

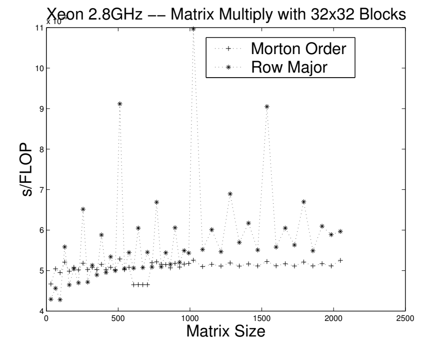

Figure 11: Performance of compiled, blocked matrix multiplication on Intel Xeon.

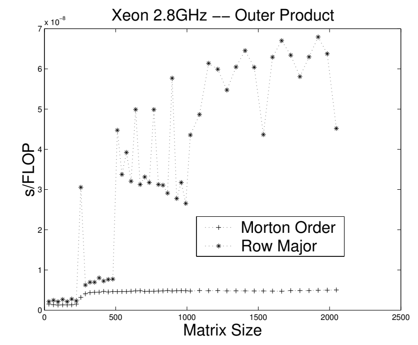

Figure 12: Performance of vector-matrix outer product on Intel Xeon.

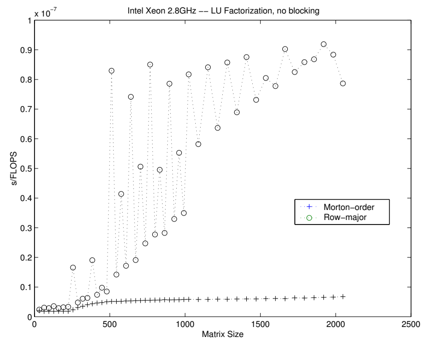

Figure 13: Performance of compiled, non-blocked LU decomposition on Intel Xeon.