Generic Support of Algorithmic and Structural Recursion for Scientific Computing *

Peter Gottschling
Institut für Wissenschaftliches Rechnen
Technische Universität Dresden
Peter.Gottschling@tu-dresden.de

David S. Wise †
Adwait Joshi
Computer Science Department
Indiana University
{dswise, joshia}@cs.indiana.edu

Abstract

Recursive algorithms, like quick-sort, and recursive data structures, like trees, play a central role in programming. In the context of scientific computing, recursive algorithms and memory layouts are studied to provide excellent cache and TLB locality independently of the platform. We show how, for the first time, generic programming (GP) and OO allow us to abstract a multitude of dense-matrix memory layouts: from conventional row-major and column-major layouts over Z- and U-Morton orders to block-wise combinations of them. All are provided by a single class that is based on our new matrix abstraction.

The algorithmic recursion is supported in generic fashion by classes modeling the new recursator, an analog of the STL iterator. Although this concept supports recursion in general, we focus again on matrix operations. Results are presented for matrix multiplication, on both conventional and tiled representations using both homogeneous and heterogeneous matrix representations. Reaching about 60% peak performance in portable C++ code establishes competitive performance in the absence of explicit prefetching and other platform-specific tuning. Comparisons with the manufacturers' libraries show superior locality. These new techniques are embedded in the the Matrix Template Library, Version 4 (MTL4).

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented programming; E.1 [*Data Structures*]: Arrays; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—object-oriented programming; D.2.2 [*Software Engineering*]: Design Tools and Techniques—software libraries; E.2 [*Data Storage Representations*]: Contiguous Representations

General Terms Design, Languages, Performance

Keywords Matrix Template Library, Generic Programming, Morton-order, Dilated integers, Doppled integers

1. Introduction

When higher-level programming first burst into applied science about 1960, computers were sold as bundles. The major item was the hardware with central processor and memory carefully designed to match one another. Each was tuned to properties of the path between them. Programming-language support was initially incidental but later matured to be a selling point.

Roll the clock forward half a century. Software is an industry unto itself and machine architecture is considerably different. Not only are there many processors and hierarchical/distributed memories, but also the path between memories and processors gets muddled. The arrival of chip multiprocessors (CMPs or multi-cores) means that on-chip processors share a path to RAM, and may be sharing local L3 cache. If the programming tools are to keep four or sixty-four CMPs busy, then they had best be collaborating on data in their shared cache—even if it be read-only. The alternative is that 95% of them be waiting on that shared path to RAM.

This over-simplified horror story is not an argument against CMPs, so much as it is a warning about underpinnings of programming languages that are 50-years young. Old assumptions about the path between processor and memory must be reconsidered if new constraints from the CMPs are not to bury us—or them.

1.1 A way forward

This paper presents high-level programming tools for dealing with abstract representations for matrices, and for presenting abstract methods over those representations. These

* Supported, in part, by NSF under a grant numbered EIA-0202048.

† Supported, in part, by NSF under grants numbered ACI–0219884 and CCF-0541364.

POOSC July, 2008, Paphos, Cyprus.

are delivered as C++ templates for generated object-oriented programs. Importantly, the programmer need not be aware of the matrix representation while writing the code and, especially as we shall see, she can choose to ignore specifics of the memory hierarchy while developing her program.

It is forward looking, in the sense that we do not here present results from its parallel performance on CMPs. Those results are available—as suggested later—and related ones have already been presented elsewhere [3, 29]. We present the versatile object-oriented tools here, and demonstrate their flexibility for the programmer.

1.2 Iterators and recursators

The Matrix Template Library (MTL) is a package of C++ templates over classes that delivers object-oriented iterative programming over many abstract types of matrices [23]. With compile-time specification of the element types (*e.g.* **int**, **float**, complex), the matrix type (*e.g.* sparse, dense, banded), and representation (*e.g.* row-major, column-major), the package delivers methods for indexing, initialization, and indexing conventions (*e.g.* transpose). Most importantly, our favorite `for`-loop is enhanced by *iterators* that increment and decrement cartesian indices without the programmer's concern for element structure or striding. These methods make iterative code for tiled algorithms considerably simpler and more expressive.

In support of the ongoing development of block-recursive algorithms over matrices, we also describe and demonstrate the *recursator* which plays the analogous role for decomposing matrices into four quadrants: northwest, southwest, northeast and southeast. These algorithms have already been demonstrated to use the memory hierarchy particularly effectively, running even faster than manufacturers' BLAS codes *without any prefetching*. Other matrix codes require the programmer to anticipate cache loading by fitting tiles to cache size (that varies across hosts) and by issuing explicit prefetch instructions in anticipation of future use of data. These block recursive algorithms avoid both requirements because they are cache oblivious [10]. Cache reuse is effected simply by careful ordering of calls to a recursive function so that sequential calls share an argument which—on the second call—should still reside in cache ready for reuse. The magic of exponential growth as the program descends a quadtree (of data) and an octree (of calls to a function $\in \Theta(n^3)$ time for order n) amplifies this reuse *in situ*.

In contrast to tiled algorithms, an algorithm using recursators at one level is effective at efficient use of all levels of the memory hierarchy: L1, L2, especially TLBs, and paging—should it become necessary. However, recursion should be abandoned somewhere above 1×1 blocks in order to balance virtues of hardware tuned for iteration against its expense for procedure calls [6, 24];a good choice to abandon recursion is at a 64×64 base block.

1.3 Matrix representation

Related to the ideas of tiled matrices and blockwise recursion are several matrix representations that use clever orderings of memory addresses. These have the effect of storing local blocks in adjacent and contiguous segments of memory addresses, aligned so that only a few lines in cache, or a page on disk suffices to hold an entire block while the algorithm uses and reuses its contents. Examples of these orderings are Morton order [16], block-data layout (BDL) or major-major—which is a row-major ordering of row-major blocks [2, 18], and Morton-hybrid—similarly a Morton ordering of row-major blocks [6, 29].

If we counter-traditionally require strides and orders of inner blocks to be a power of 2, then any of these can be represented abstractly and simply by a bit mask that identifies which bits of an offset identify its row index and, by complementation, which identify its column index. This mask is that associated with *masked integers*, a concept that unifies cartesian indexing into all the representations above. As explained below, this mask allows both iterators and recursators to be templated, in anticipation of one of these representations being chosen at compile time.

1.4 Algorithms independent of matrix representation

Notably, that mask provides both cartesian and block indexing on both row-major and Morton order representations, so that either matrix representation can be used with recursators and iterators—or even both. Moreover, since the indexing methods are associated with the matrix objects, it allows algorithms to operate on matrix objects of dissimilar representation. For example, either multiplication algorithm below, iterative or recursive, works just fine on two factors that are a Morton-ordered matrix of **float** values, and a row-major matrix of **double** values.

1.5 Onward!

The remainder of this paper is structured in five parts. Section 2 on representations of dense matrices in memory is followed by a section introducing the new recursator and explaining its use for blockwise recursion on matrix problems. Then Section 4 illustrates the use of recursators and iteration on the simple problem of matrix multiplication. Performance results are plotted for various algorithms, various representations, and differing base-cases. To show a more general application, Section 5 visits Cholesky factorization similarly including experimental results. The last section offers conclusions and a look forward into parallelism, explaining how block recursion lends itself to localized and balanced parallel threads.

2. Recursive memory layout

Good use of hierarchical memory demands that programs reuse data already stored close to the processor. Some meet this requirement with off-line prefetches that anticipate fu-

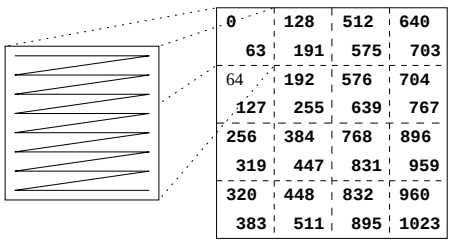


Figure 1. Memory layout of hybrid Morton-order with row-major base case

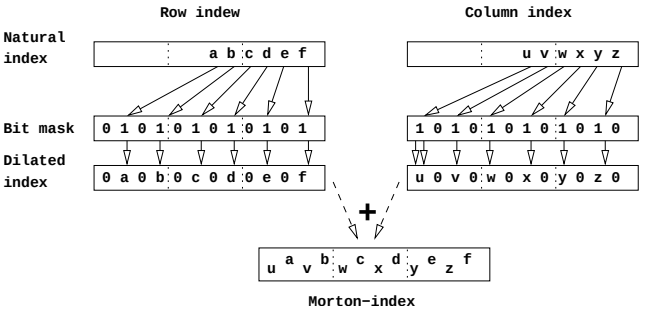


Figure 2. Bit mask for И-Morton order

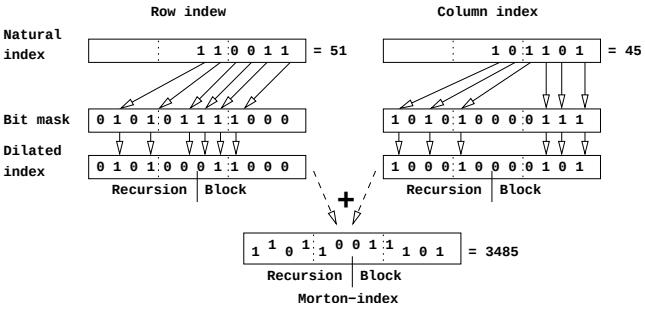


Figure 3. Bit mask for hybrid Morton order with row-major 8×8 block

ture "use;" that is eschewed here. Aside from implementing an algorithm with the abstract recursator/iterator control structures, this section develops an abstraction of dense matrix representations that delivers block locality at run time. Of course, implementation of the control structures must be sensitive to this class of representations in memory, in order to use them well.

One way to perceive the desired representations is that they are fancy tilings of matrices at different levels of the memory hierarchy. While that is consistent with historical approaches to locality [14, 28], it is important to appreciate the facility that these abstractions allow. While traditional tiling was closely tied to the iterative structure of an algorithm, the dense structures described here stand independently of an algorithm. Importantly, the algorithm can be developed and then details of the structure can be specified in tuning runtime performance. Alternatively, it is easy to develop one algorithm as a function on three matrices, and then to select three entirely different representations of those arguments—as the application requires.

2.1 Types of dense matrices

Scientific programmers are familiar with representations of dense matrices as column-major (in FORTRAN) and row-major (in C/C++). Most know of the advantages of Morton order for representing arbitrarily large blocks in contiguous memory [16], and some appreciate how its indexing, monotonically increasing horizontally and vertically, provides bounds checking with cartesian indices represented as dilated integers [21]. We prefer to use it in И-order for matrices that tend to be taller than they are wide; computer graphics uses the transposed Z-order for the dual reason.

Here we will be dealing with generalizations of all these in a template library that make these operations transparent over some new, hybrid representations. An example is the Morton-hybrid representation in Figure 1 [6, 2]. Figure 4 illustrates others as the base blocks of recursions and iterations discussed below.

2.2 Representation with bit-masks

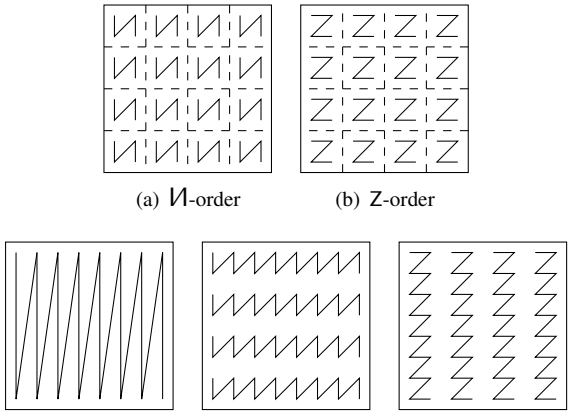
Matrix layout is specified with bit-masks. In our masks, a 1 indicates that the bit position is part of the row index; dually, a 0 indicates a bit in the column index. For example, the mask 010101010101_2, represents an И-order and 010101111000_2 characterizes a hybrid И-order matrix with 8×8 row-major blocks.

Figure 2 illustrates the mapping from a natural integer to a *dilated integer* that corresponds to indices for И-order [21, 28]. On the left-hand side, the dilation of the row index is depicted. The natural index represented by the bit chain $abcdef_2$ giving the binary number $a \cdot 2^5 + b \cdot 2^4 + c \cdot 2^3 + d \cdot 2^2 + e \cdot 2^1 + f \cdot 2^0$ is dilated with regard to the bit-mask 010101010101_2 yielding the even-dilated integer $0a0b0c0d0e0f_2 = a \cdot 4^5 + b \cdot 4^4 + c \cdot 4^3 + d \cdot 4^2 + e \cdot 4^1 + f \cdot 4^0$.

The column-index mask of И-order is 101010101010_2, and the binary value $uvwxyz_2$ on the right is mapped to the odd-dilated index $u0v0w0x0y0z0_2$. The sum or disjunction of the dilated row and column indices yields the offset of the matrix element in a contiguous memory, $uavbwcxdyezf_2$ [28]. This Morton-index is therefore only a interleaving of the indices in binary representation where the bit-mask determines in which pattern the indices are interleaved.

Traditional dense-matrix representations can be specified with bit masks when the minor dimension is a power of two. A column-major matrix with 2^b rows is characterized by a bit-mask with leading 0s and ending with b 1s. Conversely, leading 1s and b trailing 0s represent a row-major matrix with 2^b columns.

Modern processors have design features supporting row- and column-major representations with hardware support for vector processing; as a result, random masks can yield rep-



(a) И-order (b) Z-order

(c) Column-major (d) Row-major shark tooth (e) Column-major shark tooth

Figure 4. Other 8×8 base cases

resentations that cannot use these features. If the patterns of the previous paragraph are mimicked, however, in the $2b$ low-order bits of a mask then a pattern of $2^b \times 2^b$ base blocks in row/column-major results. Iterative computations can perform better with the hardware support. As an example, Figure 3 depicts the dilation of row and column indices of a *hybrid* matrix with И-order on the outer blocks, as indicated by leading bits of 010101_2, and row-major within the 8×8 inner blocks, represented by the last six bits 111000_2. The dilation of row and column indices is algorithmically identical to that with purely recursive memory layouts. To access $A_{51,45}$ in this matrix, for example, the zero-based row index $51 = 110011_2$ is dilated with the row mask 010101111000_2 yielding 010100011000_2 and the dilation of the column index $45 = 101101_2$ with the complementary mask 101010000111_2 results in 100010000101_2. Their sum, 3485, is the offset of matrix element $A_{51,45}$ from the aligned address of $A_{0,0}$ in memory.

This mapping $f\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ from the index pair to the offset is a bijective function. The inverse, f^{-1}, can be computed by separating the offset via bit-masking into dilated row and column indices and subsequently compressing them into their natural representation [20].

2.3 Shark-tooth

To compute matrix operations efficiently using super-scalar hardware support, we have found some advantage in storing matrices as row or column vectors of a short, fixed size to fit the 128-bit MMX registers [3]. These hardware-dependent sizes are the number of scalar operations computable in a single cycle by the SSE2 commands. Current sizes are 2, 4, or 8. We call this representation *shark-tooth* because of its ragged pattern illustrated in 4(d) and 4(e).

Another way to look at the shark-tooth representation is that memory blocks of size $2^t, 0 < t < b$ switch between row-major and column-major orientation in the matrix representation. Since these memory blocks are aligned we can calculate the shark-tooth bit-mask from the corresponding row-major or column-major bit-mask by flipping the last t bits. The cases $t = 0$ and $t = b$ are admissible but trivial: $t = 0$ is identical with straight row-major or column-major and $t = b$ switches the whole base case block between the two orientations.

For example, Figure 4(d) shows an 8×8 base case with shark teeth in row-major orientation. The tooth length is $2 = 2^1; 1 = t$ so that the last bit is inverted with regard to the straight row-major representation. The last 6 bits of the mask are therefore 110001_2 for every recursive matrix type with this base case. Correspondingly, the column-major shark-tooth base case in Figure 4(e) is represented by bit-masks ending in 001110_2.

2.4 Declaration of recursive matrix types

Bitmasks are used in the newly released version [15] of the matrix template library (MTL) [22]. Using our tools, one declares recursive matrices with the template class

```
morton_dense<Element, Mask>
```

where the type Element defines the type of the matrix elements and Mask characterizes the memory layout with the above-described bit-masks. For instance, the mask 0xaaaaaaaa defines a Z-order matrix and 0x555557e0 represents a hybrid Morton-order matrix with a 32×32 row-major within a block and И-ordering of those blocks. More convenient than writing the bit-masks in hexadecimal form is the generation with the meta-function generate_mask (cf. [1] for meta-programming). A regular function would not work because the bit-mask is part of the type definition and therefore already needed at compile-time.

The meta-function has four parameters:

1. Order of the recursive part as **bool**: **true** for И-order and **false** for Z-order;
2. Base block size as logarithm-base-2, e.g., the value 5 represents a base block of 32×32;
3. Base case order: this is a type parameter that can be either row_major or col_major; and
4. Shark tooth size as logarithm, e.g., 0 signifies no shark teeth and 2 represents teeth of length 4.

If the base block size is zero, the last two parameters are irrelevant. For instance, matrices of **double**s with the above-mentioned memory layouts are defined by:

```
const unsigned long morton_Z_mask=
    generate_mask<false, 0, row_major, 0>::value;
const unsigned long hybrid_mask=
    generate_mask<true, 5, row_major, 0>::value;
morton_dense<double, morton_Z_mask> a(400, 300);
morton_dense<double, hybrid_mask> b(222, 333);
```

Elements of recursive matrices can be accessed in different manners: by indices, with iterators, or with specialized address calculations for specific memory layouts [2]. The latter can provide significantly faster execution but requires different implementations for different memory layouts, and is subject to future specializations.

All recursive matrix types provide element access via bracket operator. For better code reuse and because the **operator**[] must be defined in the class [26] it is publicly derived from a CRTP class [4]. Internally, two **operator**[] calls are mapped to one **operator**() call. This base class is applicable to all matrices that provide **operator**() and is also used for compressed sparse matrices and regular dense matrices. The syntactic transformation between the different operators is realized by inline functions and generated code can be eliminated by optimization so that a[i][j] performs as fast as a(i, j).

The **operator**() dilates the row and column indices with regard to the matrix's row and column bit-masks as described in Section 2.2 in order to compute the address of the matrix element. The dilation is implemented internally with look-up tables. Generating one look-up table per bit-mask can result in very large tables because the look-up table must contain at least as many elements as the size of the largest possible index. Therefore, the look-up table that is intended to accelerate memory access can cause a critical performance obstacle by consuming too much cache space, as THIYAGALINGAM *et al.* experienced [27].

Instead of one large table, we use several tables of size 256 to look up each byte of the index and sum the results. With index types of typically four bytes, the look-up tables for one bit-mask usually require 4 kB cache space if completely loaded. It is possible that the memory requirement can be further reduced by re-using tables with equal submasks [20].

The look-up tables for a specific bit-mask are only computed once at program initialization. Two matrices with the same bit-mask use the same tables, even if the type of the elements are different. Furthermore, if A has the complementary bitmask of B then the same tables as for A's row indices are used for B's column indices and vice versa. Examples for matrices with complementary bit-masks are $\mathcal{U}$-order vs. Z-order or hybrid $\mathcal{U}$-order with 32×32 row-major base vs. hybrid Z-order with 32×32 column-major base.

Many algorithms access elements of a matrix by increasing or decreasing a row/column index within one column/row, respectively. In such cases, it is faster to compute a new Morton index by incrementing or decrementing within the class of dilated row/column indices, instead of recasting it from natural row/column representation.

3. Recursive computations

The recursive, divide-and-conquer paradigm delivers local memory use within large aggregates by following this pattern of programming:

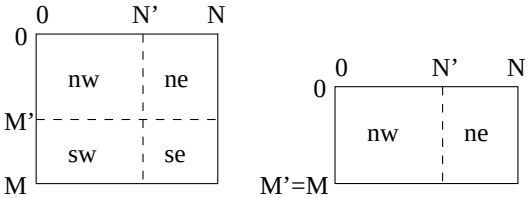
1. If the aggregate is too small, by computing the result directly, probably with iteration rather than recursion [24].
2. Otherwise, by cleaving the aggregate set into subsets that ideally are disjoint;
3. - By performing computations on each subset (recursively), and then. . .
   - By assembling the partial results (as necessary) into the aggregate result.

The recursive calculation completes when the subset is empty or small enough to be computed directly. HOARE's QUICKSORT is the prototypical example for this divide-and-conquer recursion.

One effect of such natural division is the locality among the caches with in a memory hierarchy. The same subproblems may be used to distribute a massive computation among either local or remote processors. Submatrices become the quanta of communication among distributed computers, and the subproblems can be used to create a balanced schedule over that computational resource. In cases where only the distributed memory of local computers sufficies to hold the data, the recursion can organize the communication of data from remote memories to the computer responsible for a dependent computation. Now, with the arrival of the CMPs, that same recursion can be used to locate sibling processes that share L3 cache on the same chip to be performed collaterally by the among the processors there.

A tremendous benefit of recursion is that many algorithms that are complicated to express iteratively are much simpler to program recursively. Often the recursive algorithm exhibits lower time complexities than comparable iterative algorithms; an example is quicksort versus bubble sort. Since computation grows exponentially as one descends levels in the tree of recursive calls, however, it is important to stop slightly short of the scalar case; if taken to the extreme of scalars, much efficiency will be lost to avoidable manipulation of the stack. Therefore, high-performance recursions do not descend to trivial cases, but rather they switch to iteration on blocks sufficiently large to take advantage of hardware support like pipelines and superscalar operations.

We aim for improving the locality of memory accesses for matrix operations without any prefetching. Abstract asymptotic complexity is unchanged in the cases we study, but the effort of programming and the constants of proportionality—obfuscated in big-$\mathcal{O}$ asymptotic results—are considerably reduced. With those constants now depending so much on memory speed, much is to be gained by moving computation from RAM to L1 cache.



(a) Regular (b) Horizontal only

Figure 5. Splitting of matrices into quadrants

The code complexity of simple iterative programs may seem simpler than the corresponding recursion, but that view may depend on training. Certainly, as one reaches for cache-local codes and nesting of loops become 6 or even 9 deep, the unchanging structure of nested-block recursion becomes more attractive. Such high-performance iterative codes need higher development effort and more tuning when porting to a new platform.

3.1 Matrix splitting

Different ways to partition an $m \times n$ block into submatrices are possible. In all cases, the submatrices will be disjoint and cleaved from a single point; the decomposition is always characterized by two parameters: m' and n' (Figure 5). The matrices are decomposed into four submatrices as in Figure 5(a) but there are also cases of decomposition into two submatrices, e.g. Figure 5(b), although these may be interpreted as a special case of the four-way split.

More specifically, consider a matrix $A \in \mathbb{T}^{m\times n}$ with $\mathbb{T} = \mathbb{R}$, $\mathbb{T} = \mathbb{C}$, or any other semiring type, e.g., $\mathbb{T} = \mathbb{R}^{k\times k}, k \geq 1$. We use zero-based indexing and STL-like half-open intervals in order to simplify the definition of index intervals. Thus, the row (column) indices of A lie in the range $[0, m-1] = [0, m)$ ($[0, n-1] = [0, n)$ resp.). Ranges $[x, y)$ with $x \geq y$ are allowed and considered as empty ranges. A matrix with an empty range of row or column indices represents an empty matrix.

Using these definitions, the decomposition of a matrix into four submatrices, some possibly empty, can be specified by the parameters $0 \leq m' \leq m$ and $0 \leq n' \leq n$. Depicting a_{00} in top left corner and using the analogy of cartography, the submatrix with the lowest indices is associated with north-west (A_{nw}). The quadrants in this compass-oriented notations are:

$$\begin{aligned} A_{nw} &= A[0, m')[0, n') \\ A_{ne} &= A[0, m')[n', n) \\ A_{sw} &= A[m', m)[0, n') \\ A_{se} &= A[m', m)[n', n). \end{aligned}$$

Therefore, cases of four non-empty submatrices, Figure 5(a), result from splittings with $0 < m' < m$ and $0 < n' < n$. For $m' = m$, the matrices A_{sw} and A_{se} are empty, as in Figure 5(b). Similarly, A_{ne} and A_{se} are blank if $n' = n$. If both applies $A = A_{nw}$ and the other three matrices are empty. Decompositions with $0 = m' < m$ and $0 = n' < n$ are not used in our implementation.

To provide maximal flexibility for controlling the recursive descent, we have implemented several classes providing different matrix splitting. Objects of these classes return the value m' (n') in the member function row_split() (col_split() resp.). To avoid redundant computations, the values are computed once in the constructor. We implemented the following classes:

- half_splitter: The simple calculation $m' = \lceil m/2 \rceil$ and $n' = \lceil n/2 \rceil$ allows that all submatrices (and submatrices of submatrices) are approximately equally sized. On the other hand, matrix orders that are powers of 2, or at least multiples of 4 or 8, are favorable for optimization purposes, like loop unrolling and tiling.
- separate_dim_splitter: Choose $m' = 2^k$ such that $2^k < m \leq 2^{k+1}$ and $n' = 2^{k'}$ such that $2^{k'} < n \leq 2^{k'+1}$. This creates more submatrices with even sizes but also retains the rectangular shape of matrices. In some cases, almost square matrices are split into rectangular, e.g. 16×17 into 8×16, 8×1, 8×16, and 8×1.
- max_dim_splitter: The maximum $m'' = n'' = \max\{m', n'\}$ from the previous definition is used. This formula tends to create more square submatrices. The aforementioned matrix of size 16×17 is decomposed into one matrix of size 16×16, one matrix of size 16×1, and two empty matrices that can be ignored in recursive computations. If this splitting is used in recursive matrix products, the arguments can be decomposed inconsistently.

Unfortunately, all the splittings described above are error-prone and inefficient in the context of multiplying matrices in the context of block-recursive matrix multiplaction.

In order to provide optimally sized base cases in operations with multiple arguments (*i.e.* almost all blocks have size 32×32), we introduce a global outer bound. For the matrix product $C = AB$ with $A \in \mathbb{T}^{m\times p}$, $B \in \mathbb{T}^{p\times n}$, $C \in \mathbb{T}^{m\times n}$, we define a global variable k such that

$$2^{k-1} < \max\{m, p, n\} \leq 2^k. \tag{1}$$

Outer bounds are stored in matrix_recursators, introduced in the next section. Applications do not state the outer bounds, but they are inferred in the constructor of the recursator. Multi-argument functions balance the outer bounds of their recursator arguments. The corresponding splitting class using outer bounds is quite simple:

- outer_bound_splitter: For a given k split the matrix with $m' = \min\{m, 2^{k-1}\}$ and $n' = \min\{n, 2^{k-1}\}$.

Internally, the matrices are only virtually split using outer_bound_splitter, and submatrices are actually created only at the bottom of the recursive descent.

3.2 Matrix recursator

The Recursator is a concept to support recursion in similar manner as the Iterator concept supports iteration in STL [17, 25] and MTL [1]. Currently we focus on using recursators for matrix operations, but the generalization of the Recursator concept is a subject for future development, e.g., to enable generic functions to work simultaneously on vectors and matrices.

We provide the class matrix_recursator that is templated on the matrix type. For instance, defining a recursator rec for a hybrid row-major matrix B is implemented in the following way:

```
typedef morton_dense<int, hybrid_mask> hybrid_matrix;

hybrid_matrix B(222, 333);
matrix_recursator<hybrid_matrix> rec(B);
```

This class uses outer_bound_splitter to determine quadrants of matrices. Hard-wiring this decomposition type in the recursator allowed for several optimizations. On the other hand, the class type matrix_recursator_s is additionally parameterized with a decomposition type (default is max_dim_splitter<Matrix>).

The matrix to which a recursator refers (which can be a sub$\cdots$submatrix of the user's matrix) is accessible with **operator**$*$. The boolean function is_empty() tests whether the referred matrix is empty, allowing a function to use the algebra of zero (as multiplicative annihilator and additive identity) to return immediately. The four submatrices can be referred to with the quadrant functions north_west(), north_east(), south_west(), and south_east(). However, these function do not return the submatrices itself but recursators representing them at low cost. The size of the quadrants depend on the splitting method of the recursator.

The class matrix_recursator is optimized for outer-bound decomposition. At construction of a recursator the number of rows and columns are read from the matrix and the outer bound is computed as outer_bound $= 2^k$ with $k = \min\{\ell\colon 2^\ell \geq \max\{m, n\}\}$. Furthermore, the recursator stores the first row and column index. When a recursator is constructed from a matrix, both indices are set to 0. A quadrant function returns a recursator with halved outer bound and potentially increased first row and first column index, for instance:

```
self south_west() const
{
  self tmp(*this); // copy myself
  tmp.my_bound >>= 1; // divide by 2
  tmp.my_first_row += tmp.my_bound; // inc. 1st row
  return tmp;
}
```

[1] Latin suggests Recursor from *recurere*, but Cursor already exists in MTL. We root our word from later Latin *recursare*.

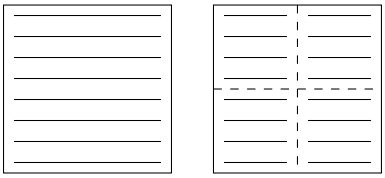


Figure 6. Recursive handling of regular dense matrix

The first column of the south-west quadrant is the same as in the containing matrix itself. The sample code also shows that the referenced matrix is not even read while creating a recursator representing a submatrix. The tests for emptiness (is_empty(), north_east_empty(), ...) are also implemented matrix-free. Only **operator**$*$ creates a submatrix. Thus, block-recursive algorithms can be implemented in a fashion where the complete recursive descent is executed without fetching from the matrix, and submatrices are only realized at the iterative base cases.

Due to their generic design, recursators are not limited to matrices with recursive memory layout. They can be applied to all matrices with a submatrix function. Theoretically, the recursator could be even used for sparse matrices if the type provides a submatrix function. Since the creation of compressed-row or compressed-column sparse sub-matrices is extremely inefficient in general and the benefits of recursive operations cannot out-weight this, we refrain from decomposing sparse matrices represented that way. Figure 6 illustrates how a row-major dense matrix (left) can be decomposed into four dense submatrices (right).

4. Matrix multiplication

Matrix multiplication is a very important example for performance measurement largely due to its role in LINPACK [8], which is used as a yardstick to rank the world's most powerful computers[7]. Ignoring the calling context, it is isomorphic to Schur complement or rank-k updates in many algorithms. The existence of highly efficient implementations of LINPACK for cache-based microprocessor architectures, as well as for vector processors, allows for fair comparisons of high-performance architectures.

4.1 Recursive algorithm

Although recursion provides excellent locality, run-time overhead from excessive function calls becomes a handicap where compilers do not fit them to hardware that—after fifty-some years of loops—has refined support for iteration. It has always been expensive to recur down to 1×1 [5] or 2×2 blocks [9, 24].

Block-recursive multiplication combines the advantages of the iterative paradigm (low-overhead loop and incremental, regular memory addresses) with the benefits of recursive functions (cache-oblivious blocking of large matrices

and easily identified parallelism). The generic function for C+= A∗B reads: [2]

```
template <typename BaseCase, typename BaseTest,
          typename MatrixA, typename MatrixB,
          typename MatrixC>
void inline
recursive_mult_add(MatrixA const& a, MatrixB const& b,
                   MatrixC& c, BaseTest const& test)
{
    using recursion::matrix_recursator;
    matrix_recursator<MatrixA> rec_a(a);
    matrix_recursator<MatrixB> rec_b(b);
    matrix_recursator<MatrixC> rec_c(c);
    equalize_depth(rec_a, rec_b, rec_c);

    rec_mult_add(rec_a, rec_b, rec_c, BaseCase(), test);
}
```

This function creates recursators for each matrix and calls the recursive function rec_mult_add with the recursators; see Figure 7. The function equalize_depth assures that the depth of recursion is consistent for all three recursators independently of whether or not the matrices are square.This balancing of the recursators guarantees that all three arguments reach base-case size at the same time.

The entire block-recursive multiplication is performed by function rec_mult_add in Figure 7 by means of recursators. Since the $m \times n$ matrices are embedded in virtual $2^k \times 2^k$ matrices, it is possible that sub-matrices of the virtual matrices are empty. This is checked in the first statement of each invocation and, in case that the incidence of one matrix on the addressed row- and column-range is empty, no operations are necessary.

Then follows a test of whether the computation reached the base case. If the test succeeds, the base case operation specified by bc is executed on the matrices referenced by the three recursators. This operation must perform the operation C+= A∗B on the sub-matrices when C+= A∗B or C= A∗B is being computed as the global problem. Accordingly C−= A∗B is computed with the same operation on the sub-matrices. How this operation is realized is defined by the base-case functor bc, which might be a generic functor from MTL, adapted from the manufacturer's library, or a user-defined code possibly optimized for a specific platform.[3] If the base case has not yet been reached, the 8-way block-recursion is performed on recursators selecting sub-matrices.

[2] The real version in MTL is implemented as a functor for better composition. It also contains dispatching to non-recursive functors when one of the matrix types does not support sub-divisibility. Another template argument parameterizes the element update to subsume `C+=A*B`, `C-=A*B`, and `C=A*B`. Nevertheless the recursive technique is the same and we omit other issues here for the sake of simplicity.

[3] Current compilers are not able to verify whether the performed operation is a semantically correct matrix multiplication, only whether the signatures match. Research is in progress to incorporate compilers into the semantic verification of generic and non-generic libraries [12].

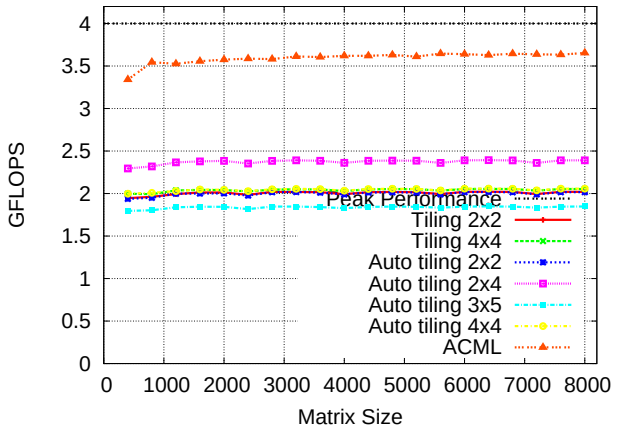


Figure 8. Performance of optimized matrix product with hybrid matrices

4.2 Performance results

In this section, we show performance results for different matrix formats and several combinations of matrix types. We also measure cache and TLB misses and compare the locality with the highly optimized vendor library. If not stated otherwise, the benchmarks are run on a 2 GHz AMD Opteron with 4 GB RAM, and the sources compiled with gcc 4.1 using the compiler flags:
−O3 −ffast−math −mcpu=opteron −mtune=opteron −mfpmath=sse −msse2.

4.2.1 Accelerating base case multiplication

One of MTL's most important research issues is to provide optimization in highly customizable fashion. Instead of implementing all possible variations of an algorithm in separated functions, parameterizable meta-functions are developed, which are transformed by the compiler into the desired functions. The bounds of the tile $m \times n$ are template parameters of the matrix product, and an $m \times n$ block in the inner loop is generated at compile-time. Several experiments have demonstrated that the generated code is as efficient as hand-written tiling or unrolling.

Different tests demonstrate that on certain platforms 2×4 tiling (e.g., on the before-mentioned Opteron as in Figure 8) can be the best choice, whereas on other platforms 4×4 (e.g., a PowerPC as in Figure 9) or 2×2 tiling is favorable; showing the necessity of such parameterization.

In contrast to non-generic libraries where only few combinations of argument types are supported, the free choice of types make an *a priori* parameterization impossible; it is important that the user can parameterize the operation in the function call. The details of these transformations are to be addressed in future publications.

Figure 8 compares the performance for different tiling sizes and shows that 2×4 is the best choice for Opteron within this context. Here, a hybrid Morton-order matrix with

```
template <typename RecursatorA, typename RecursatorB, typename RecursatorC,
          typename BaseCase, typename BaseCaseTest>
void rec_mult_add(RecursatorA const& rec_a, RecursatorB const& rec_b,
                  RecursatorC& rec_c, BaseCase const& bc,
                  BaseCaseTest const& test)
{
    if (is_empty(rec_a) || is_empty(rec_b) || is_empty(rec_c)) return;
    if (test(rec_a)) {
        bc(*rec_a, *rec_b, *rec_c); return; }

    RecursatorC c_north_west= north_west(rec_c), c_north_east= north_east(rec_c),
                c_south_west= south_west(rec_c), c_south_east= south_east(rec_c);
    rec_mult_add(north_west(rec_a), north_west(rec_b), c_north_west);
    rec_mult_add(north_west(rec_a), north_east(rec_b), c_north_east);
    rec_mult_add(south_west(rec_a), north_east(rec_b), c_south_east);
    rec_mult_add(south_west(rec_a), north_west(rec_b), c_south_west);
    rec_mult_add(south_east(rec_a), south_west(rec_b), c_south_west);
    rec_mult_add(south_east(rec_a), south_east(rec_b), c_south_east);
    rec_mult_add(north_east(rec_a), south_east(rec_b), c_north_east);
    rec_mult_add(north_east(rec_a), south_west(rec_b), c_north_west);
}
```

Figure 7. Listing of recursive multiplication

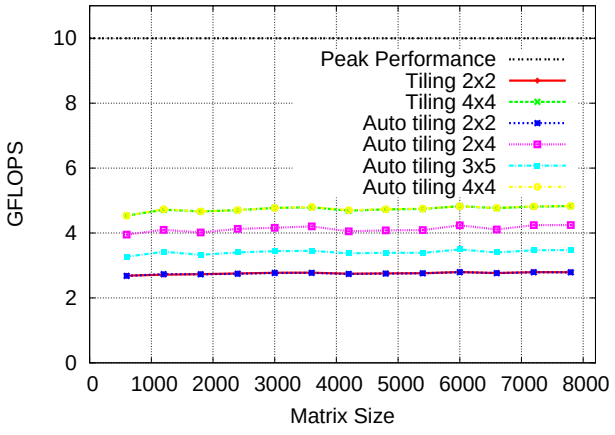


Figure 9. Performance of matrix product on PowerPC (same computations as in Figure 8)

64×64 row-major block is multiplied with a corresponding matrix with column-major block.

The same multiplications as in Figure 8 were performed on a 2.5 GHz PowerPC G5 with 512 MB L2 and 3.5 GB RAM using gcc version 4.0.1 with compiler flags $-$O3 $-$fast $-$ffast$-$math, shown in Figure 9.

Both plots also illustrate that the meta-functions with 2×2 and 4×4 tiles perform exactly as fast as their traditional counter-part. The perfect performance scaling of the product can also be observed in both curves.

4.2.2 Beyond BLAS

In the previous measurements we showed that computation can be performed as well by platform-tuned classical libraries, in particular by BLAS and its derivatives. Multiplying matrices of different scalar types is not supported by these packages. Figure 10 depicts the floating point performance of different argument combinations in the multiplication. A dense **float** matrix is multiplied with a **float**/**double** and stored in a dense matrix with **double** values. The first computation is correspondingly performed on block-recursive matrices (denoted with hybrid). Furthermore, a block-recursive **double** matrix is multiplied with a corresponding complex<**double**>. All computations revealed good performance.

Several combinations of dense (denoted by first letter 'd' in the plot) and hybrid ('h') matrices are examined in Figure 11. The second letters represent the types of matrix elements with 'd' for **double**, 'f' for **float**, and 'z' for complex<**double**>.

4.3 Locality

An important subtheme in all these results is the impact of locality. Algorithms without local reuse of extant cache contents incur a subtle overhead from moving data more often through the memory hierarchy. They will slow by a constant factor, reflecting the slower cycle times of repeating remote access, compared to another algorithm that avoids much of that access.

We have collected statistics on cache misses and even (for large problems) paging. L1, L2, and TLB misses are shown here in Figures 13 and 14. They are plotted in units

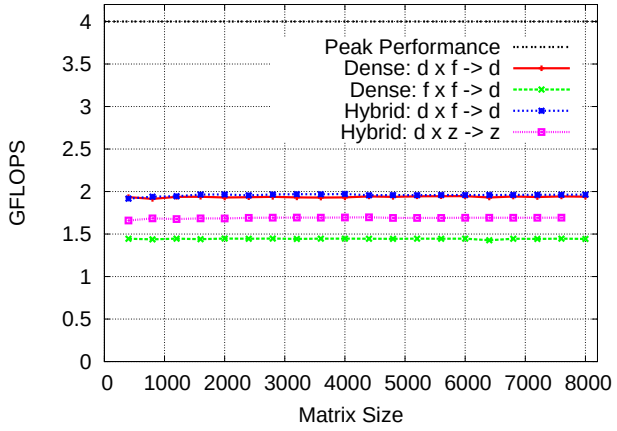


Figure 10. Multiplying matrices with different scalar types

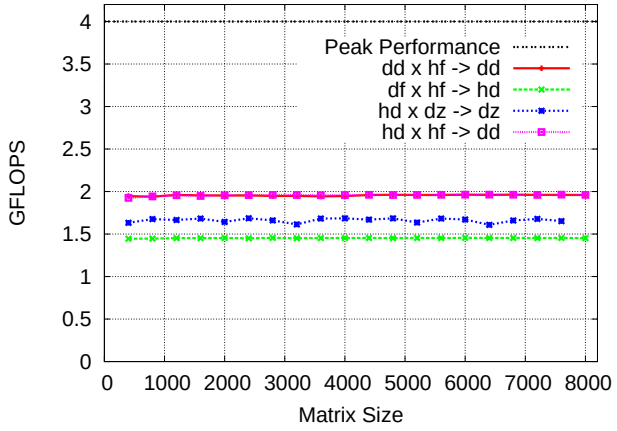


Figure 11. Multiplying matrices with different scalar types and memory layouts

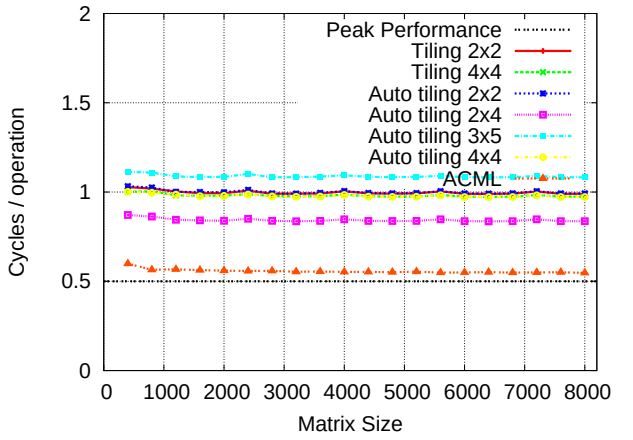


Figure 12. Performance of optimized matrix product with hybrid matrices, corresponds to Figure 8

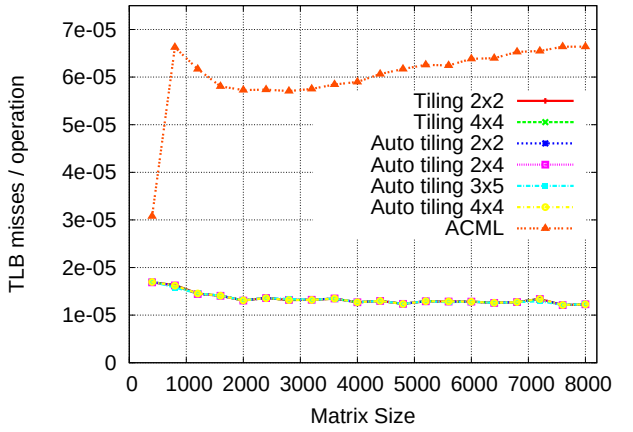


Figure 14. TLB misses

of "resource-per-floatingOperation" because, particularly in the latter figure, it is good to see expensive cache misses approaching zero. That is, fewer is better.

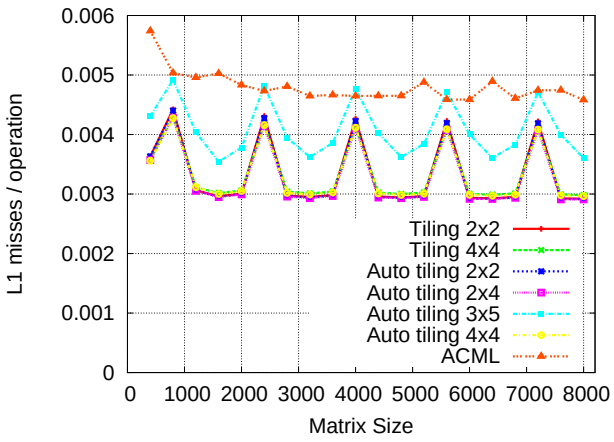
The analogous plot of time performance in Figure 8, which showed "floatOperations-per-timeUnit," becomes its reciprocal in Figure 12 except that we choose a processor's cycle as the timeUnit—abstracting away its clock rate. The data in these plots is identical but in the latter, once again, smaller is better for a nice comparison with Figures 13 and 14. Such plots of performance normalized to the necessary work (floating-point operations) is preferred in theoretical analyses and is explained more completely by ADAMS and WISE [3].

The results in Figures 13a and 13b show improvements in L1 and L2 cache misses available from the block-recursive algorithms without any prefetching. Figure 14, however, shows a dramatic *five-fold* improvement in expensive TLB misses (whose costs are interrupts rather than just memory cycles). [4] This improvement, in particular, is a harbinger of the future performance of these structures and algorithms in a CMP or distributed processing environment. If such improvement is available from abstract MTL style and structures, without any prefetching, then much more performance will be available from abstract programming on multiprocessors.

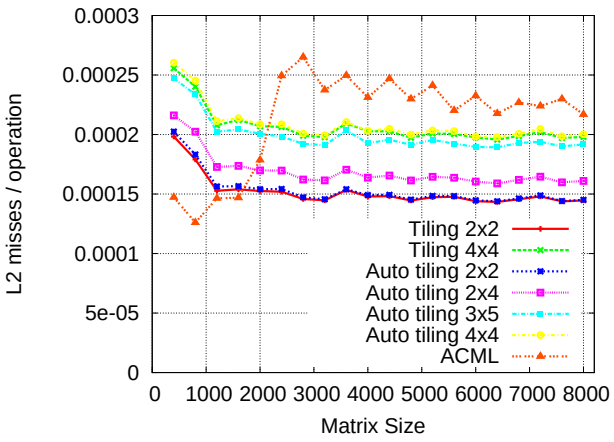
4.4 Multiplying non-recursive matrices recursively

In all previous examples, the multiplied matrices had a block-recursive memory layout. In fact, the computations combined two fashions of recursion: the memory layout of the matrices and the algorithmic approach. Performing iterative computations on recursive matrix structures have been

[4] In addition, we compared our locality with GotoBLAS for the Opteron [11]. For brevity, we omit the plots here and only state that our block-recursive revealed better locality regarding L1, L2, and TLB misses. However, the differences are much smaller than for ACML.



(a) L1 cache misses



(b) L2 cache misses

Figure 13. Cache misses

shown to be inefficient (e.g. [27]). In contrast, we will show that recursive matrix multiplication performs well on traditional row-major and column-major matrices.

Considering these types of matrices allows for comparability with other linear algebra packages. For the sake of variety, we finish benchmarking A∗B for the moment and consider products of transposed matrices. More precisely we investigate A∗trans(A), an operation that is favorable for row-major matrices because both arguments are traversed with stride one in the innermost loop (when implemented by dot product).

In order to provide an objective judgement we use the benchmarking library BTL from LAURENT PLAGNE [19]. Our matrix product is compared with the highly tuned assembly library GotoBLAS, with uBLAS from the Boost collection, Blitz++, MTL2, and with implementations by LAURENT PLAGNE in C, Fortran, and using the STL. The benchmarks are again performed on the 2GHz Opteron. Figure 15 shows that most implementations suffer under decreased locality. The Fortran version deteriorates especially rapidely with increasing matrix sizes, since the locality is further hindered by large strides in the inner dot product.

Conversely, the operation trans(A)∗A favors column-major matrices as can be seen in Figure 16. The Fortran implementation performs considerably better than most C and C++ codes for large matrices. The recursive algorithm in MTL4 is able to compensate for the large strides in both arguments. One can see nevertheless stronger performances loss when the matrix size is close to multiples of high powers of two. If the operation $A^T A$ is run-time dominant in the application, users are free to define A as column-major matrix. More stability can be reached by recursive matrix types yet.

Finally, we return to regular matrix products, i.e. AA. This operation induces large strides for row-major and

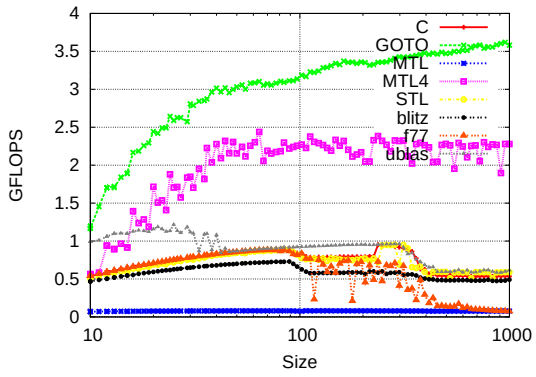


Figure 15. Performance of AA^T

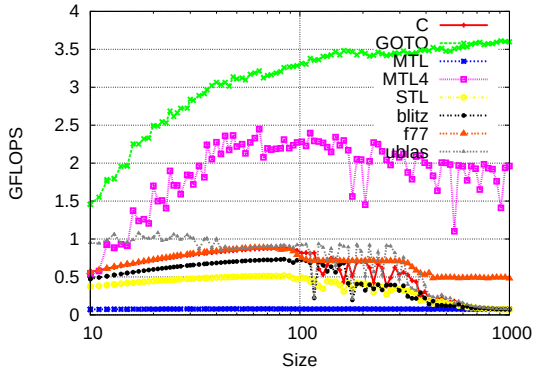


Figure 16. Performance of $A^T A$

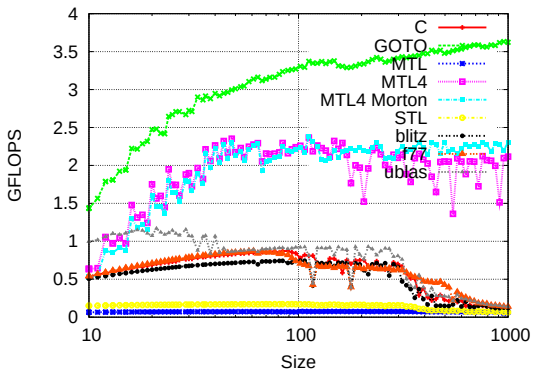


Figure 17. Performance of AA

column-major matrices in one argument or the other. As a result, the performance decrease is perceivable unless blocking techniques (as in GotoBLAS) or recursion is used (as in MTL4); see Figure 17. The plot also demonstrates the before-mentioned stabilizing impact of the recursive matrix types. The row-major matrix enables a rather higher average performance with occasional degradation. In contrast, the block-recursive matrix—where the block is row-major as well—yields stable high-performance.

Representative comparison between libraries are only possible when they base on the same technology, i.e. same language and compiler. Comparison of the raw performance of high-level language libraries with that of hand-tuned assembly codes encourages misjudgement of libraries' performance. Measuring floating-point performance of compiled software still shows how well the source-code implementation drives the potential capacity of a specific platform—or class of platforms—and how well the compiler can translate these sources into platform-tuned machine code[5]. Therefore, a compiled library that is slower than its hand-written counterpart is not necessarily programmed sub-optimally. Instead, this slowdown can be attributed to the fact that compilers are not yet as capable of choreographing the registers as experienced assembly programmers are.

In the next section we will explain, how MTL4 overcomes this performance gap elegantly without the need of modifying applications or sacrificing genericity.

4.5 Coping with Hand-tuned Performance in Generic Software

The generic design for maximal applicability is a paramount criterion for MTL4 and limiting functions to BLAS-supported argument combinations is beyond all question. Providing a specific interface, e.g., `gemm(alpha, beta, A, B, C)`, that only prodides BLAS operations would not be very convenient because type alterations would be required to modify an application (replacing the gemm calls with analogous expressions). More importantly, such BLAS usage cannot be incorporated into generic programs. Instead, MTL4 provides BLAS support in a very convenient form.

Applications are written naturally with mathematical operators, e.g., `A= B * C`, and the library selects based on the types the most efficient implementation. BLAS libraries (hopefully well-tuned) are prioritized over all C++ functions in case the type combination of the arguments permits. In case of dense-matrix multiplication[6] the requirements for using BLAS are that

- The matrix types must be all traditional dense matrices—whereby any combination of row-major and column-major is allowed;
- The matrix elements are either **float**, **double**, or their respective complex types; and
- All three matrices have the same type of elements.

As a consequence, applications are realized in generic fashion and acceleration via hand-tuned libraries is provided whenever feasible. In order to keep MTL4 as independent as possible, the presence of a BLAS library is only required when programs are compiled with the flag `-DMTL_HAS_BLAS` (thus, only then the transparent BLAS acceleration is provided). Resuming, MTL4 programs can perform with the same performance as any hand-tuned BLAS library *without any source code modification*.

5. Performance of Other Operations

5.1 Cholesky Decomposition

An efficient implementation of matrix multiplication enable the realization of CHOLESKY decomposition with comparable performance. We compute the CHOLESKY factoriztion of A recursively according to:

$$A = \begin{bmatrix} B & C^* \\ C & D \end{bmatrix} \Longrightarrow \begin{bmatrix} \hat{B} & C^* \\ C & D \end{bmatrix} \Longrightarrow \begin{bmatrix} \hat{B} & C^* \\ \hat{C} & D \end{bmatrix} \Longrightarrow \begin{bmatrix} \hat{B} & C^* \\ \hat{C} & \hat{D} \end{bmatrix} \Longrightarrow \begin{bmatrix} \hat{B} & C^* \\ \hat{C} & \tilde{D} \end{bmatrix} \quad (2)$$

such that

$$B = \hat{B}\hat{B}^*, C = \hat{C}\hat{B}^*, \hat{D} = D - \hat{C}\hat{B}^*, \tilde{D} = \hat{D}\tilde{D}^* \quad (3)$$

where $\hat{B}$ and $\tilde{D}$ are lower triangular matrices, cf. [3]. Each of these computations is itself recursively defined on sub-matrices. As in the previous section, we apply a block-recursive approach where all calculations on matrices smaller than or equal to 64×64 are processed iteratively.

[5] There is an exception to this statement. Some compilers have predefined, hand-tuned target code for certain source code patterns. Then, rather simplistic implementation might lead to excellent performance, while sophisticated source code might not match the corresponding pattern and result in lower performance.

[6] Currently, this dispatching is only available for dense matrix products. However, after the technology for it is developed, further utilizations will be straight-forward.

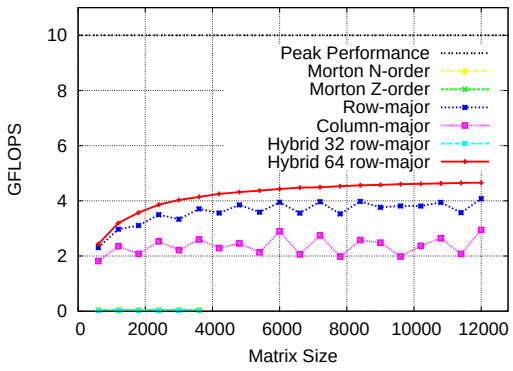


Figure 18. Cholesky decomposition with optimized Schur update (4×4 tiling)

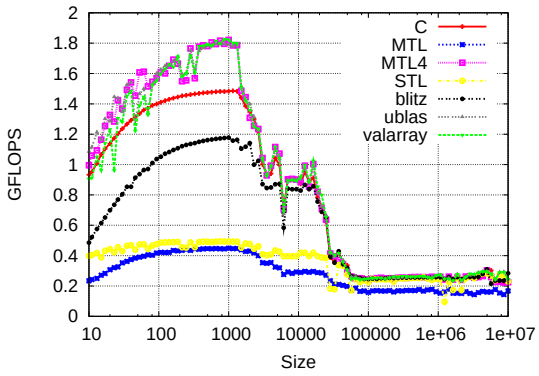


Figure 19. Binary vector expression

Large matrices can be decomposed with more than 4.6 GFLOPS on a PowerPC 2.5 GHz while the implementation is very simple and cache-oblivious. While BLAS only supports column-major matrices, our implementation also works on row-major matrices, as well as Morton-order matrices, block-recursive matrices and all formats presented in this paper, see Figure 18. More details are given in [13]

5.2 Binary Vector Expression

The performance for computing the vector expression

```
x= alpha * y + beta * z;
```

is compared in different libraries. The operation can be written directly in this form with MTL4. Internally the library implements such vector operations by means of expression templates. As a result, the entire computation is performed in one single loop. Figure 19 shows the performance on the 2GHz Opteron in comparison to other high-level language implementations

5.3 Dot Product

Figure 20 compares the performance for computing dot products of vectors. MTL4 uses meta-programming techniques to tune the computation of several operations. By default the loop is unrolled 8-fold. However, the user can choose an arbitrary block size in the loop. For instance, the expression dot<14>(v, w) performs the operation with 14-fold unrolling. Future versions will provide specialized default for block sizes that depend on the vector type and the platform.

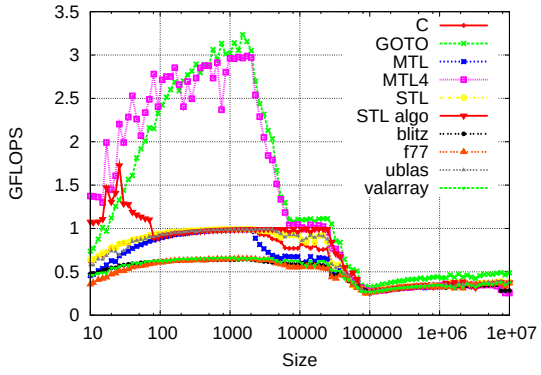


Figure 20. Performance of dot product

While not all operations in MTL4 are tuned yet to their full performance potential, it is evident from the presented results that the combination of meta-programming techniques with recursion provides the highest performance that modern compilers can deliver.

6. Conclusions

The new Matrix Template Library includes very general representations and tools for block-recursive programming for large matrix problems. This paper vertically explores their use on large, dense problems. That is, it explores a couple of problems deeply, showing how a single algorithm can be compiled onto various machines, representations, blockings/tilings, base cases, and element types. The tools are general enough to allow the same source code to execute well on heterogeneous matrix types, where arguments that are individually homogeneous (like all arrays) may have heterogeneous types among themselves.

The implementation is an impressive display of the power and convenience of object-oriented (OO) programming with C++ templates. Using the templating tools and type classes, it transforms abstract scalar types, memory layouts, algorithms, and control structures into cache-oblivious implementations that perform very well relative to their competition.

Here this competition is fairly mundane: scientific programming, but that class of problems occupies the high ground of big-iron computing. The future of OO and GP is illuminated by this high-level programming for performance that compares to that of algorithms and hardware that have been polished over half a century. Although this vertical exploration was done only on uniprocessors, their path into

parallelism has already been opened and is just as bright [3]. The demands of locality in the memory hierarchy and the appearance there of chip multiprocessors (*a.k.a.* CMPs, multicores) constitute a challenge for all programming paradigms; this work offers a way for GP to meet it.

We have here presented generic OO tools that provide compile-time specialization of dense matrix representations including Morton-ordered, Morton-hybrid, and block-tiled variants on row- and column-major. These last two are familiar and well supported by hardware, and so they remain attractive for base-case blocks in nested block recursions. Additionally, the OO tools include recursators, which are good for higher-level, cache-oblivious decomposition, and iterators which are especially useful down in the base blocks. Nevertheless, these control abstractions can be mixed and matched at any level—even among arguments heterogeneously—and still deliver correct results.

The performance delivered by these high-level tools is most respectable. Judicious use of recursators and a good choice of hybrid representation delivers matrix-multiplication times that are 67% of the performance of the manufacturer's BLAS3 library—*from* C++ *source code.* These results are quite an improvement on those of THIYAGALINGAM *et al.* [27] who used OO techniques less successfully on similar problems.

Other results show how performance varies with the size of the base cases or scalar types without changing any code. We have also presented results on L1, L2, and especially TLB misses (reduced fivefold) that demonstrate excellent locality without any prefetches. This cache-oblivious performance—polynomial computation of $s^{\frac{3}{2}}$ FLOPs on space s—points toward excellent parallel performance, which is so important for the new CMPs.

A side result is that the blockwise-recursive definition of the algorithm seems to be far more important for excellent performance than is a blockwise layout of a matrix in memory. We show good performance even without a block-oriented storage pattern.

So the future application of GP and OO techniques seems bright in the context of the new CMPs. Even for this remote—but demanding—set of matrix problems from scientific programming, object-oriented techniques offer leverage on the problems of locality and communication, as well as the on the problems of process decomposition and balanced scheduling that the recursator provides. They offer leverage on decomposing large problems into independent, collaborating subprocesses simply via the concept of collateral argument evaluation borrowed from functional programming. These generic tools, therefore, lead the way for object-oriented programming to tackle the challenges of using the parallel capacity of this next generation of processors. That is the challenge for the next MTL.

Acknowledgement: We thank Christopher Cole for his help rerunning data and final editing, and Laurent Plagne for his help with BTL and the libraries.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[2] M. D. Adams and D. S. Wise. Fast additions on masked integers. *SIGPLAN Not.*, 41(5):39–45, May 2006. http://doi.acm.org/10.1145/1149982.1149987

[3] M. D. Adams and D. S. Wise. Seven at one stroke: Results from a cache-oblivious paradigm for scalable matrix algorithms. In *MSPC '06: Proc. 2006 Wkshp. Memory System Performance and Correctness*, pages 41–50. ACM Press, New York, Oct. 2006. http://doi.acm.org/10.1145/1178597.1178604

[4] J. Barton and L. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.

[5] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottenthodi. Recursive array layouts and fast parallel matrix multiplication. In *Proc. 11th ACM Symp. Parallel Algorithms and Architectures*, pages 222–231. ACM Press, New York, June 1999. http://doi.acm.org/10.1145/305619.305645

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

[7] J. J. Dongarra, H. W. Meuer, E. Strohmaier, and H. Simon. Top 500 supercomputer sites. `http://www.top500.org`, 2006.

[8] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[9] J. D. Frens and D. S. Wise. Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code. *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.*, 32(7):206–216, July 1997. http://doi.acm.org/10.1145/263764.263789

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

[11] K. Goto. GotoBLAS. `http://www.tacc.utexas.edu/resources/software/#blas`.

[12] P. Gottschling. Fundamental algebraic concepts in concept-enabled C++. Technical Report 638, Computer Science Dept., Indiana University, Oct. 2006. http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR638

[13] P. Gottschling, D. S. Wise, and M. D. Adams. Representation-transparent matrix algorithms with scalable performance. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 116–125, New York,

NY, USA, 2007. ACM Press.

[14] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *Proc. 4th Int. Symp. Architectural Support for Programming Languages and Operating Systems, SIGPLAN Not.*, 26(4):63–74, Apr. 1991.
http://doi.acm.org/10.1145/106975.106981

[15] A. Lumsdaine, J. Siek, L.-Q. Lee, and P. Gottschling. The Matrix Template Library home page. `http://ww.osl.iu.edu/research/mtl`, 2006.

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

[17] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, second edition, 2001.

[18] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):640–654, July 2003.
http://dx.doi.org/10.1109/TPDS.2003.1214317

[19] L. Plagne and F. Hülsemann. BTL++: From performance assessment to optimal libraries. In *ICCS 2008, Kraków, Poland, Proceedings, Part III*, volume 5103 of *LNCS*, pages 203–212, June 23-25 2008.

[20] R. Raman and D. S. Wise. Converting to and from dilated integers. *IEEE Trans. Comput.*, 57(4):567–573, Apr. 2008.
http://dx.doi.org/10.1109/TC.2007.70814

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

[22] J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Comput. Sci.*, pages 59–70. Springer, Berlin, 1998.
http://dx.doi.org/10.1007/3-540-49372-7_6

[23] J. G. Siek and A. Lumsdaine. The matrix template library: generic components for high-performace scientific computing. *Computing in Science and Eng.*, 1(6):70–78, Nov. 1999.
http://dx.doi.org/10.1109/5992.805137

[24] J. Spieß. Untersuchungen des Zeitgewinns durch neue Algorithmen zur Matrix-Multiplication. *Computing*, 17:23–36, 1976.

[25] A. Stepanov. The Standard Template Library — how do you build an algorithm that is both generic and efficient? *Byte Magazine*, 20(10), Oct. 1995.

[26] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[27] J. Thiyagalingam, O. Beckmann, and P. H. J. Kelly. Is Morton layout competitive for large two-dimensional arrays, yet? *Concur. Comput. Prac. Exper.*, 18(11):1509–1539, Sept. 2006. http://dx.doi.org/10.1002/cpe.1018

[28] D. S. Wise. Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 – Parallel Processing*, volume 1900 of *Lecture Notes in Comput. Sci.*, pages 774–883. Springer, Heidelberg, 2000.
http://dx.doi.org/10.1007/3-540-44520-X_108

[29] D. S. Wise, C. L. Citro, J. J. Hursey, F. Liu, and M. A. Rainey. A paradigm for parallel matrix algorithms: Scalable Cholesky. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par 2005 – Parallel Processing*, number 3648 in Lecture Notes in Comput. Sci., pages 687–698. Springer, Berlin, Aug. 2005. http://dx.doi.org/10.1007/11549468_76