Representation-transparent Matrix Algorithms with Scalable Performance[*]

Peter Gottschling[†]
Indiana University
Bloomington, IN 47405, USA
pgottsch@cs.indiana.edu

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

Michael D. Adams [§]
Indiana University
Bloomington, IN 47405, USA
adamsmd@cs.indiana.edu

ABSTRACT

Positive results from new object-oriented tools for scientific programming are reported. Using template classes, abstractions of matrix representations are available that subsume conventional row-major, column-major, either Z- or И-Morton-order, as well as block-wise combinations of these. Moreover, the design of the Matrix Template Library (MTL) has been independently extended to provide recursators, to support block-recursive algorithms, supplementing MTL's iterators. Data types modeling both concepts enable the programmer to implement both iterative and recursive algorithms (or even both) on all of the aforementioned matrix representations at once for a wide family of important scientific operations.

We illustrate the unrestricted applicability of our matrix-recursator on matrix multiplication. The same generic block-recursive function, unaltered, is instantiated on different triplets of matrix types. Within a base block, either a library multiplication or a user-provided, platform-specific code provides excellent performance. We achieve 77% of peak-performance using hand-tuned base cases without explicit prefetching. This excellent performance becomes available over a wide family of matrix representations from a single program. The techniques generalize to other applications in linear algebra.

*Supported, in part, by NSF Grant EIA-0202048.

†Supported, in part, by NSF Grant ACI–0219884 and the Lilly Endowment.

‡Supported, in part, by NSF under grants numbered CCR–0073491 and ACI–0219884.

§Supported, in part, by NSF Grant CCR–0073491.

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, Morton-order, dilated Integers, doppled integers

1. INTRODUCTION

The arrival of multicore chips (a.k.a. chip multiprocessors or CMPs) presents a new challenge for high-level programming, just as the stratification of memory did years ago when caching was introduced to provide smaller, faster memories to keep pace with ever faster processors. Caches became a challenge first to programmers to take explicit advantage of the new performance, and then to tool designers to deliver that performance implicitly to high-level programs. This cache race has now been suspended since the next round of Moore's-law improvements will be delivered via CMPs. Programmers and tool designers now must find ways to use them well.

1.1 A Programmer's Toolkit

FORTRAN's original model of flat memory—clocked the same as the processor—has been extended over the years to vector processing, to demand paging, to distributed multiprocessing, to shared memories, to hierarchical caches, and now to the multicores. Some of these extensions prompted changes of programming style (*e.g.* libraries); others affected the tools, altering the programming languages themselves (*e.g.* `forall` loop control).

We focus on the important class of solutions to problems from linear algebra, and present an alternative approach to present and future problems of using these new architectures. In particular, we develop a style that will identify independent, localized threads in a balanced schedule. We have been experimenting with alternative representations for dense matrices, as well as with block-wise recursion recently with some superlative results [2]. Here we offer language tools to support those representations and that recursive style—independently of one another—even though they do exhibit a synergy that encourages their use together.

Conventional practice for matrix problems already tiles matrices for locality of reference, and layers control structures to fit the tiles. A standard tool from generic programming is the *iterator* to abstract traditional loopwise control. The analogous tool introduced here is the *recursator* fitted to the quadtree recursions that we have found so successful.

Independently of the recursator, we have also implemented tools that provide a whole family of dense-matrix representations, with different memory layouts. They range from conventional row- and column-major orderings to Morton-order and various kinds of hybrid representations. One interpretation of "hybrid" is the familiar tiling, to extract locality with small submatrices. We deliver a whole family of these representations simply by selecting a mask that describes the representation of its cartesian indices. Their use allows a convenient and efficient hybrid representation where (for example) outer blocks are Morton-ordered for locality in blockwise recursion and base-case blocks are column-major to suit hardware designed. This family includes the representation called Morton-hybrid or Block Data Layout (by PARK *et al.*) whose conversions CHATTERJEE *et al.* had found so expensive [1, 6, 17].

These matrices have been implemented in a library of C++ templates that allows the programmer to take advantage of either recursion or iteration (using recursators or iterators), with any of these representations specified by the masks. Moreover, it allows several matrix arguments for a single code to be represented heterogeneously, so that many different representations can be tested just by changing a mask for the indices of each. Remarkably, a program written with recursion can be compared with one written with iteration on any of these representations just by changing those masks.

This package offers two independent contributions: the recursator to augment the iterator, and the free specification of dense-matrix representation with masks for its Cartesian indices. It also shows how they may be used independently or consistently to achieve excellent performance.

The remainder of this paper is in five parts. This introduction ends with a review of the memory constraints on matrix algorithms. Section 2 introduces block recursion for representing matrices in several ways and introduces our tools to facilitate their declarations. Section 3 uses those structures to develop the recursator, the analog of MTL's matrix iterator. Both are applied to matrix multiplication in the next section and to Cholesky decomposition in Section 5. The last section visits conclusions and ongoing work.

1.2 Architectural challenge

The goal of tiling, whether it is accomplished by the nested-block recursion (recursators) or by deeper nestings of iterations (iterators over tiles), is to provide chunks of local computation within running programs. The need for this locality is easiest to understand in algorithms whose running time is superlinear in the space for its data. In that case, many slow RAM cycles can be avoided if much processing happens while data resides in faster cache. The idea is that a tile should reside in cache long enough to absorb many more FLOPs than its size, amortizing whatever RAM probes were expended moving it to/from cache.

Matrix addition is an example of an algorithm that does not illustrate benefits of tiling, because its time is linear in

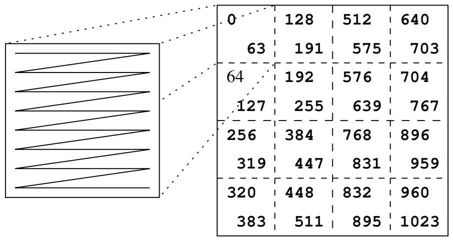


Figure 1: Memory layout of hybrid Morton-order with row-major base case

the space of its operands and result. Matrix multiplication, however, performs $\Theta(s^{1.5})$ flops on space s, or maybe $\Theta(s^{1.4})$ when Strassen's algorithm is used[1] . We can improve the running time, even choosing an algorithm that needs more flops, if we can deliver those flops from cache at uniformly higher hertz rates than from RAM at much lower hertz.

So, when a matrix multiplication is tiled by hand, we seek to fit tiles into the faster caches with the tile size depending on the run-time environment. That still leaves the problem of fitting L1, L2, L3, and even TLB cache with nested iterations (on tiles within tiles) with more nested iterations. In contrast, we have found that the block recursion fits all these caches—eventually—and so no multilevel or prefetching fitting is necessary; the recursive codes come out to be cache-oblivious [10]. Nevertheless, in both cases conventional hardware suggests that the base blocks use something close to conventional row-major or column-major, in order to take best advantage of the hardware's superscalar instructions and pipelined processing. In either case, the size of these base blocks should be tuned for the specific target machine but, as a rule of thumb, 32×32 works well.

Another oft-overlooked step is to align the matrices on cache and page boundaries. This needs to be done only once at start-up; thereafter it guarantees that tiles or base-case blocks occupy the minimum number of cache lines. Since the sizes of those lines tend to be powers of two, we also favor those powers as block orders for ever larger tiles/blocks. We have even observed the best base case order overfilling L1 cache, while running fastest by reducing L2 and TLB misses: truly cache-oblivious [2].

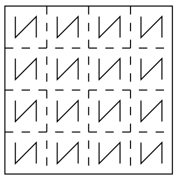
2. RECURSIVE MEMORY LAYOUT

2.1 Types of Dense Matrices

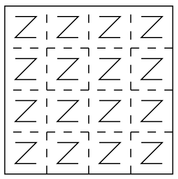
Everyone is familiar with representations of dense matrices as column-major (in FORTRAN) and row-major (in C/C++). Most know of the advantages of Morton order for representing arbitrarily large blocks in contiguous memory [15], and some appreciate how its indexing, monotonically increasing horizontally and vertically, provides bounds checking with cartesian indices represented as dilated integers [19]. We prefer to use it in И-order for matrices that tend to be taller than they are wide; computer graphics uses Z-order for the dual reason.

Here we will be dealing with generalizations of all these in a template library that makes all these operations transpar-

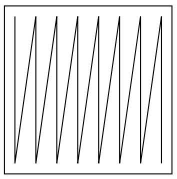
[1]Big-$\mathcal{O}$ notation can mislead one here because it hides the very constants that distinguish RAM speed from cache speed. We want less of the former and more of the latter.



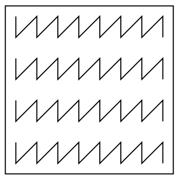
(a) И-order



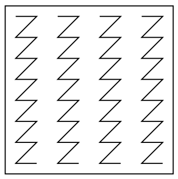
(b) Z-order



(c) Column-major



(d) Row-major shark tooth



(e) Column-major shark tooth

Figure 2: Other 8×8 base cases

ent over some new, hybrid representations. An example is the Morton-hybrid representation in Figure 1, that has also been called BDL [6, 17]. Figure 2 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 101010101010_2, represents an И-order and 010101111000_2 characterizes a hybrid Z-order matrix with 8×8 row-major blocks.

2.3 Shark-tooth

To compute matrix operations efficiently using superscalar 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 [2]. 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 2(d) and 2(e).

2.4 Declaration of Recursive Matrix Types

Bitmasks are used in the soon-to-be-released version [14] of the matrix template library (MTL) [20]. 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 block and И-order in the recursive part.

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 [23].

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 unifying tables derived from equal submasks.

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 tables for A's row indices are used for B's column indices and vice versa. Examples for matrices with complementary bit-masks are И-order vs. Z-order or hybrid И-order with 32×32 row-major base vs. hybrid Z-order with 32×32 column-major base.

Many algorithms access elements of a matrix with increasing/decreasing row/column indices within one column/row, respectively. In such cases, it is more efficient to compute a new Morton index by incrementing/decrementing dilated row/column indices with the addition method in the index's class, instead of computing it from the natural row and column `int` representation.

3. RECURSIVE COMPUTATIONS

The general idea of recursive computation is that a calculation is performed on a large aggregate of values:

1. If the aggregate is minimal in size, by computing the result directly.
2. Otherwise, by cleaving the aggregate set into subsets that ideally are disjoint;
3.
   - By performing computations on each subset (recursively), and then...
   - By combining the partial results (as necessary) into the aggregate result.

Recursive descent stops when the subset is empty or small enough to be computed directly. A prototypical example for such divide-and-conquer recursion is HOARE's QUICKSORT.

The cleaving provides a natural division for many problems of large computation beyond locality in the caches of a uniprocessor. The same subproblems may be used to distribute a massive computation among either local or remote processors. The submatrices become quanta of communication among distributed processors, the subproblems can be used to create a balanced schedule over the computational resource, and in cases where only the aggregate memory of the collection of processors is sufficient to hold the data, the recursion can organize the communication of data from remote memories to the processor responsible for a dependent subcomputation. And now, with the arrival of the CMPs, that same organization can be used to locate sibling processes, that can usefully share L3 cache, on the same chip to be performed collaterally by the sibling processors there. That is, we intend that the cleaving of Step 2 be used to divide the porcess into local, large, balanced, and independent subprocesses.

An advantage of recursion is that many algorithms that are complicated to express iteratively are much simpler to program recursively. The recursive algorithm can exhibit lower time complexities than obvious iterative algorithms; an example is quicksort, as opposed to 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; taken to the extreme of scalars, recursion will lose much efficiency to stack manipulation. Therefore, high-performance recursions do not descend to the 1×1 cases, but rather switch to iterations on blocks small enough to be local, but large enough to take advantage of hardware support like pipelines. Spieß showed that these base-case iterations ran faster even though they had higher asymptotic complexity than their recursive competition [21]; asymptotes are irrelevant down around order-32.

Our goal is to improve 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 avoiding too many moves between RAM and L1 cache.

The complexity of simple iterative code may seem simpler than the corresponding recursion, but that view may depend on training. Certainly, as one reaches for cache-local codes and 3 nested loops become 6 or even 9, the unchanging structure of nested-block recursion becomes more attractive. Such high-performance iterative codes need not only higher development effort but also more tuning moving among different platforms.

3.1 The Matrix Recursator

The Recursator is a concept to support recursion in similar manner as the Iterator concept supports iteration in STL [16, 22] and MTL. Currently we focus on using recursators for matrix operations but the generalization of the Recursator concept is subject to future research, e.g., to enable generic functions to work at the same time on trees and subdividable 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<double, hybrid_mask> hybrid_matrix;

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

The matrix to which a recursator refers (which can be a sub$\cdots$submatrix of the user's matrix) is accessible with the member function get_value(). The Boolean member function is_empty() tests whether the referred matrix is empty, for instance to return immediately from a function. Using the four submatrices is realized with the quadrant functions north_west(), north_east(), south_west(), and south_east().

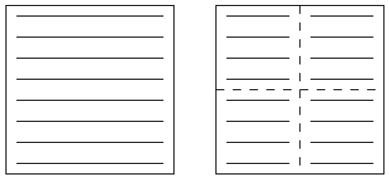


Figure 3: Recursive handling of regular dense matrix

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.

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. The recursator could even be 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-weigh this, we refrain from decomposing sparse matrices represented that way. Figure 3 illustrates how a row-major dense matrix (left) can be decomposed into four dense submatrices (right).

4. MATRIX MULTIPLICATION

The most intensively studied operation in scientific computing is matrix multiplication, especially the product of dense matrices. One of the reasons is that it is, performance-wise, the most important building block of LINPACK [8] which in turn is used to rank the most powerful computers in the world (Top 500 [7]). The existence of highly efficient implementations of LINPACK for cache-based architectures, as well as for vector processors, allows for relatively fair comparison of most high-performance computers.

Our intention in studying matrix multiplication is not to build a faster implementation but to study the run-time behavior of our generic, recursive approach. The recursive techniques are intended to improve the cache locality in a transparent manner. Optimizations for vector processors require completely different techniques and are not addressed in this paper.

As mentioned before, the recursion can occur in two forms, firstly, in terms of the matrices' memory layout and, secondly, in the manner how a complex computation is decomposed into smaller operations that could run in parallel. Both approaches can be used independently or in combination.

4.1 Iterative Computations on Recursive Data

Dense matrix multiplication is implemented in its simplest form as three nested loops. It is a popular experiment to observe the performance change by switching the order of loops. Although this leads to accelerations for some sizes of matrices, it is unavoidable with homogeneous row- or column-major order that memory will eventually be accessed in large strides, increasing the likelihood of accessing slow RAM very often. Cache-reuse can be improved by computing block-wise on contiguously stored sub-matrices. In fact, most high-performance implementations use blocking realized, in effect, by 6 or even 9 nested loops to improve locality. The drawback of this approach is that the size of the blocks must be adjusted to every size of L1, of L2, and eventually of L3 cache, as well as of the translation look-aside buffer (TLB). Another possibility to improve locality are recursive matrix types as presented in Section 2.1.

THIYAGALINGAM *et al.* investigated five different SC kernels, one of them is matrix multiplication, using Morton-ordered matrices with iterative implementations [23]. There are several reasons for the reported low performance results:

1. All Morton-indices were build each time from scratch. In many algorithms, repeated redilation of `int` indices can be avoided by incrementing or decrementing extant dilated integers with a method in the (dilated) index class, of few processor cycles [19].

2. The alleged acceleration of index dilation with look-up tables uses one large table occupying a large footprint in dear cache, or forcing it to be refreshed often (especially with one-way interleaved cache.) These memory waits can be reduced by creating multiple small look-up tables as described in Section 2.4 (at loop initialization where dilation cannot be easily replaced by increments) [18].

3. Recursion was limited to the memory layout and the iterative algorithms used were not appropriate to the data structure. In the following, we show more suitable algorithms for Morton-order and other dense representations.

Although iterative computations on blocked matrix representations can be significantly improved, locality would still be constrained by repeated cache overruns from full row and column traversals, and the over-all performance on large matrices would suffer. For these reasons, we avoid purely iterative algorithms and study various recursive and block-recursive algorithms in the following sections. (Other results, predating efficient base codes and these templates are also available [9, 13, 25].)

4.2 Recursive Matrix Multiplication

Earlier results showed that not only pure iteration but also pure recursion performs rather slowly. Pure recursion, descending down to 1×1 [5] or 2×2 blocks [9], is not wrong but the overhead of excessive function calls at leaves of the recursion tree displaces the hardware support that is available there on modern architectures, such as superscalar operations and pipelining.

Block-recursive multiplication combines the advantages of the iterative paradigm—low-overhead loops and regular often contiguous memory access—with benefits of recursive functions—cache-oblivious hierarchical blocking of large matrices. The generic function used internally for block-recursion is:

```
template <typename BaseCase, typename BaseCaseTest,
          typename MatrixA, typename MatrixB,
          typename MatrixC>
void inline
recursive_mult_add(MatrixA const& a, MatrixB const& b,
                   MatrixC& c, BaseCaseTest 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);
}
```

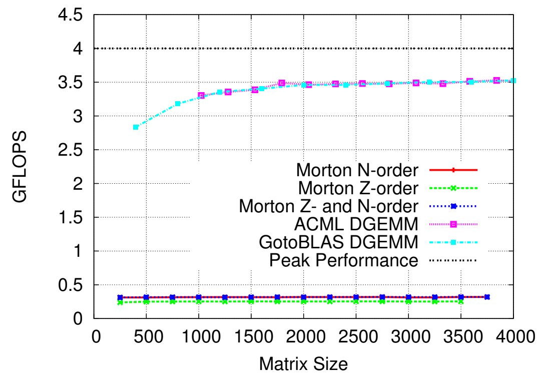


Figure 5: Morton-order matrix product on Opteron

This function creates recursators for each matrix and calls the recursive functions with them; see Figure 4. The function `equalize_depth` assures that the depth of recursion is consistent for all three recursators independently, whether the matrices are square or rectangular.

Combined with block-recursive structures, like tiled or Morton-order matrices, the block recursion localizes the addressing within aligned submatrices. So, cache hits are relatively more likely and false sharing within cache lines is prevented.

The function `rec_mult_add` in Figure 4 performs the entire block-recursive multiplication generically using recursators. Each recursator is first checked: whether the referenced matrix is empty. For example, when the matrix has dimensions 64×0, then no further action is necessary.

There follows a test whether this is a base case, where the computation can be done directly and locally. The `test` itself is parametric. It can be based either on data in the recursator or on information in the referenced matrix. We provide several templated classes for testing so that the size of the base case will have been chosen by the user. Some classes include the base case size in the type information, enabling optimizations using matrix-type conversions depending on the recursive memory layout and the base-case size; see Section 4.3.

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`. How this operation is realized is defined by the 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. (Current compilers are not able to verify whether the performed operation is semantically correct, only whether the signature of the functor's **operator()** matches the base case types. Research is ongoing to check semantic properties [12].) If the base case has not yet been reached, the block-recursion is performed on recursators selecting sub-matrices, oriented by the compass.

Figure 5 shows the performance of multiplying of Morton-order matrices in И-order, in Z-order, and with the combination of both (for technical reasons we used the optically very similar N instead of И to label the plots). The com-

```
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 (rec_a.is_empty() || rec_b.is_empty() || rec_c.is_empty())
        return;
    if (test(rec_a)) {
        bc(rec_a.get_value(), rec_b.get_value(), rec_c.get_value());
        return;
    }
    RecursatorC c_north_west= rec_c.north_west(), c_north_east= rec_c.north_east(),
                c_south_west= rec_c.south_west(), c_south_east= rec_c.south_east();
    rec_mult_add(rec_a.north_west(), rec_b.north_west(), c_north_west, bc, test);
    rec_mult_add(rec_a.north_west(), rec_b.north_east(), c_north_east, bc, test);
    rec_mult_add(rec_a.south_west(), rec_b.north_east(), c_south_east, bc, test);
    rec_mult_add(rec_a.south_west(), rec_b.north_west(), c_south_west, bc, test);
    rec_mult_add(rec_a.south_east(), rec_b.south_west(), c_south_west, bc, test);
    rec_mult_add(rec_a.south_east(), rec_b.south_east(), c_south_east, bc, test);
    rec_mult_add(rec_a.north_east(), rec_b.south_east(), c_north_east, bc, test);
    rec_mult_add(rec_a.north_east(), rec_b.south_west(), c_north_west, bc, test);
}
```

Figure 4: Listing of recursive multiplication

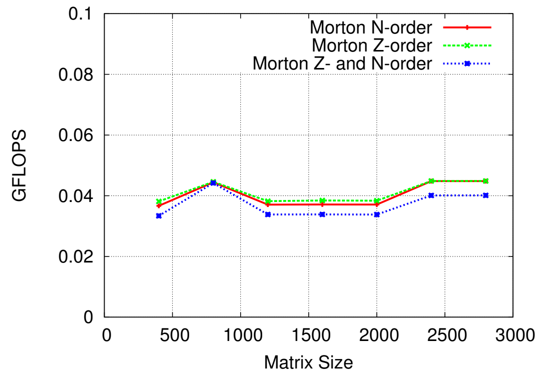


Figure 6: Morton-order matrix product on PowerPC

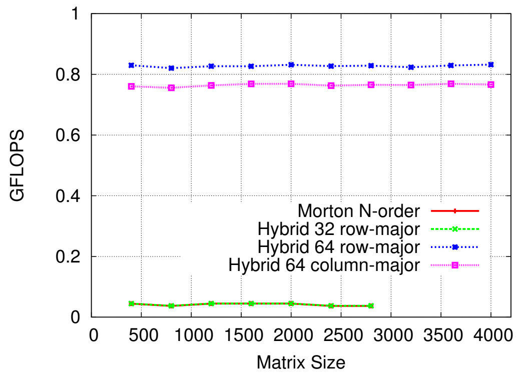


Figure 7: Casting base case

putations are computed on a 2 GHz AMD Opteron with 4 GB RAM. The program was compiled using Intel's icc with the flag −O3. Why the computation on Z-order is slightly slower than on И-order or than the mixed multiplication is not known. For comparison, we added the performance of the highly optimized dgemm routine from the manufacturer's library, ACML [3], and from GotoBLAS [11] with the same excellent performance. Figure 6 shows the same computations performed on a 2.5 GHz PowerPC with 512 MB L2 and 3.5 GB RAM using gcc version 4.0.1 with flags −O3 −fast −ffast−math. Despite the aggressive optimization flags, the gcc executable was considerably less efficient on the PowerPC than icc executable on the slower Opteron.

However, the measurements do not confirm the slow results of Thiyagalingam *et al.* [23]. The compute time is almost completely consumed in the base blocks which is a generic multiplication on matrices of dimensions 32×32 or smaller. Therefore, these performance results show how their iterative algorithms could perform for small matrices if the Morton-indices would have been computed incrementally instead of by dilation. The recursive descent requires only a negligible amount of execution time (≤ 1 % in this example). However, it can be seen clearly from the flat plots that the recursive algorithm delivers consistent performance as the problem grows to overflow caches and TLBs, and elsewhere RAM [2].

4.3 Type-wise Acceleration of Block-recursion

Morton-ordered matrices can be efficiently traversed row-wise and column-wise by using integral arithmetic on dilated integers. Our matrix types provide data structures to traverse all rows and columns of a matrix and to iterate over all elements within one row or column.

In order to reach maximal performance for the iterative computations in the base case blocks, we convert recursive matrix formats to regular dense matrices if the memory layout of the base block is equivalent to a row-major or column-major matrix. This is achieved by examining the least-significant bits of the matrix's mask. How many bits are considered depends on the base-case size that must therefore be available at compile-time in the type of the base case test. If the base case size is for instance 32×32 then the last $10 = 2 \cdot \log_2 32$ bits of the mask represent the memory layout of the base case matrix. The function

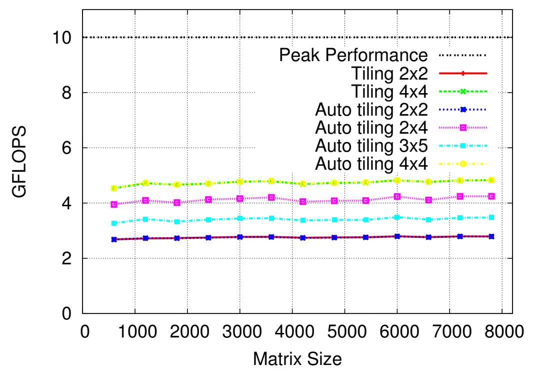


Figure 8: Optimizing the performance of the base case multiplication

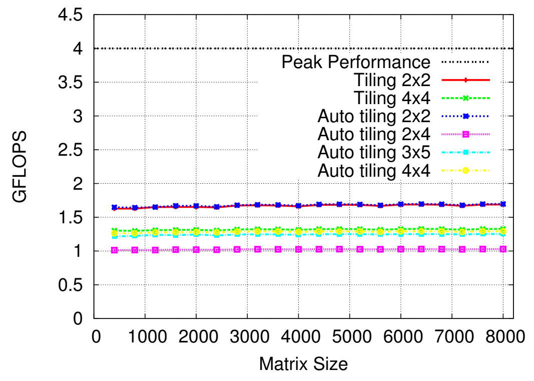


Figure 9: Performing the multiplications from Figure 8 on 2 GHz Opteron compiled with icc

```
base_case_cast<BaseCaseTest>(matrix);
```

converts hybrid Morton-order matrices into regular row-major if the mask's last 10 bits are 1111100000 or column-major matrix if the mask ends on 0000011111. Otherwise the original matrix is returned.

Figure 7 shows the performance for multiplying different types of matrices with a potential matrix cast at the 64×64 block level. Pure Morton-order matrices and hybrid matrices with 32×32 block size are not cast. Matrices with a 64×64 block—where the memory layout is either row-major or column-major within—are treated as regular dense matrices at block level providing much faster traversal.

Multiplying regular dense matrices allows for different kinks of optimization. The well-known technique of tiling is applied here in a new way. The tiling is implemented only once with tile sizes customized by template parameters. In several experiments this new meta-programming technique was demonstrated to be as performant as hand-written code and will be subject of future publication(s).

Figure 8 compares the performance for different tiling sizes and shows that 4×4 is the best choice for the PowerPC. A hybrid Morton-order matrix with 64×64 row-major block is multiplied with a corresponding matrix with column-major block. The plot also illustrates that the meta-functions with 2×2 and 4×4 tiles perform exactly as fast as their traditional counter-parts.

The same multiplications as in Figure 8 are performed on the 2 GHz Opteron compiled with icc version 9.0, shown in Figure 9. The perfect scaling of the performance can be observed.

4.4 Recursive Multiplication on Non-recursive Representations

In all previous examples, the factor matrices used block-recursive memory layout. Nevertheless, benchmarks demonstrated that the recursive approach also improves the locality for conventional dense matrices. This provides good scalability Figure 10[2], although we observed that hybrid Morton matrices perform more stably. The same optimizations as in Section 4.3 were used. Although the submatrices used in base blocks are not stored in contiguous memory and the strides between two rows or columns increase with matrix size, the effective locality of block-recursive algorithms still provides good scalability.

[2]At size 3200 the plots seem to dip because it is the only even multiple of 128 tested. Such striding is a weakness of row-major.

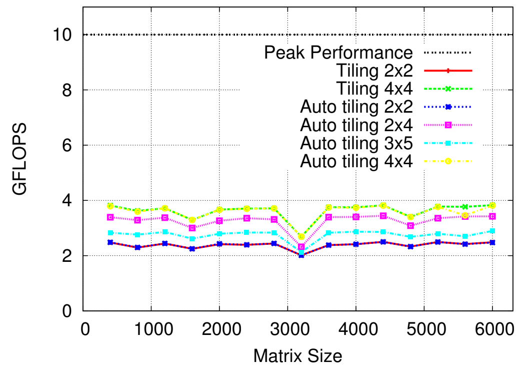


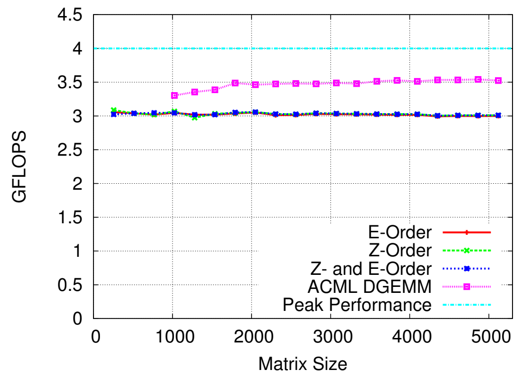
Figure 10: Recursive multiplication of dense matrix with optimization

4.5 High-Performance Tuning

A hand-tuned multiplication used only in the 32×32 base case sufficed for performance competitive with the manufacturers' MKL or ACML libraries [2]. Taking that code into our block-recursive superstructure provides the scalability in Figure 11.

Here, the base case was written as assembly-level intrinsics using SSE2 operations to attain superscalar performance: two `double` operations in one cycle. For better alignment of SSE2 commands, the matrix is laid out in shark-tooth order, changing only the base-case block of the storage mapping. Whether the structure around that block is И- or Z-order is unimportant, as shown in Figure 11. These data are plotted twice, once in units of FLOP/second, and once in units of cycles/FLOP which is favored by ADAMS and WISE because it is independent of clock speed, and because it presents performances as a multiple of peak performance, which can be compared directly to other resource measures, like TLB misses [2]. It would plot the leading coefficient for an $\mathcal{O}(n^3)$ function.

This base case code is tailored for a specific platform and memory layout of the matrix's base block. Applying this code correctly is only possible when all of the following con-



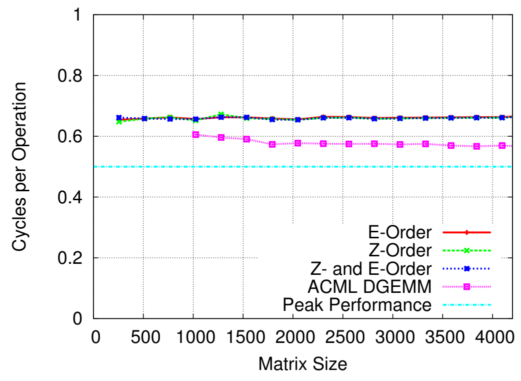


Figure 11: Performance with hand-tuned base cases

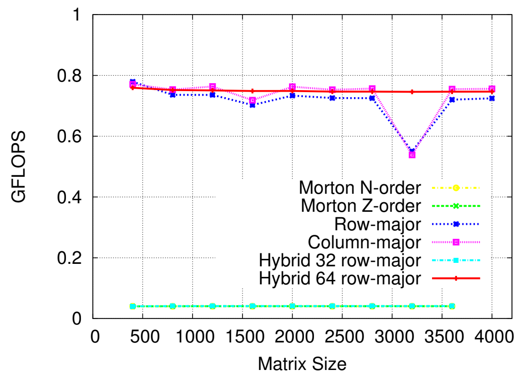


Figure 12: Performance of Cholesky with simple implementation on 2.5 GHz PowerPC

ditions hold:

- The program is directed to use Opteron tuning (specified by a macro);
- It is compiled with gcc (detected by predefined compiler macros);
- The matrix's value type is **double** (verified by template specialization); and
- The matrices' base cases provide the correct memory layout (validated by a meta-function on the bitmask in case the matrix is in generalized Morton-order, which in turn is verified by template specialization).

In case that not all of the conditions above apply, the base case operation is dispatched to another functor at compile-time. Therefore, our library enables users to run block-recursive computations with platform-dependently tuned base case code if available and with generic base case functions otherwise. The dispatching is realized at compile-time transparent to the user.

5. CHOLESKY DECOMPOSITION

In order to solve linear systems $Ax = b$, the matrix A is typically factorized into a product of a lower triangular matrix L and a upper triangular matrix U. The linear system $LUx = b$ is then easily solved by $y = L^{-1}x$ and $x = U^{-1}y$. Especially for multiple linear systems with the same matrix A and different vectors b this approach is extremely beneficial since the factorization or direct solution are $\mathcal{O}(n^3)$ operations and the subsequent substitutions only $\mathcal{O}(n^2)$.

A symmetric positive-definite matrix A is representable as a product of a lower triangular matrix L and its transposed $A = LL^T$. Accordingly, a Hermitian positive-definite matrix A can be decomposed into a product of a lower triangular matrix L with strictly positive diagonal entries and its conjugate transposed L^*, i.e. $A = LL^*$. Cholesky factorization is only half as expensive as LU decomposition.

We use the recursive computation from Adams and Wise [2]. The matrix A is transformed according to

$$A = \begin{bmatrix} B & C^* \\ C & D \end{bmatrix} \Longrightarrow \begin{bmatrix} \hat{B} & C^* \\ C & D \end{bmatrix} \Longrightarrow \begin{bmatrix} \hat{B} & C^* \\ \hat{C} & D \end{bmatrix} \Longrightarrow \begin{bmatrix} \hat{B} & C^* \\ \hat{C} & \hat{D} \end{bmatrix} \Longrightarrow \begin{bmatrix} \hat{B} & C^* \\ \hat{C} & \tilde{D} \end{bmatrix} \quad (1)$$

such that

$$B = \hat{B}\hat{B}^*, C = \hat{C}\hat{B}^*, \hat{D} = D - \hat{C}\hat{B}^*, \hat{D} = \tilde{D}\tilde{D}^* \quad (2)$$

where $mtrB$ and $\tilde{D}$ are lower triangular matrices. Each of these computations is itself recursively defined on sub-matrices. As in the previous section, we apply a block-recursive algorithm where calculations on base blocks improperly smaller than 64×64 are iterative.

Figure 12 depicts our first measurement block-recursive Cholesky factorization for real matrices on the 2.5 GHz PowerPC. It uses a very simple implementation on the block-level. To illustrate the simplicity, the following listing provides the entire implementation of the first two transformations:

```
template < typename Matrix >
void cholesky_base (Matrix & matrix) {
    for (int k = 0; k < matrix.num_cols(); k++)
        matrix[k][k] = sqrt (matrix[k][k]);
        for (int i = k + 1; i < matrix.num_rows(); i++) {
            matrix[i][k] /= matrix[k][k];
            typename Matrix::value_type d = matrix[i][k];
            for (int j = k + 1; j <= i; j++)
                matrix[i][j] -= d * matrix[j][k];
        }
}

template < typename MatrixSW, typename MatrixNW >
void tri_solve_base(MatrixSW & SW, const MatrixNW & NW) {
    for (int k = 0; k < NW.num_rows (); k++)
        for (int i = 0; i < SW.num_rows (); i++) {
            SW[i][k] /= NW[k][k];
            typename MatrixSW::value_type d = SW[i][k];
```

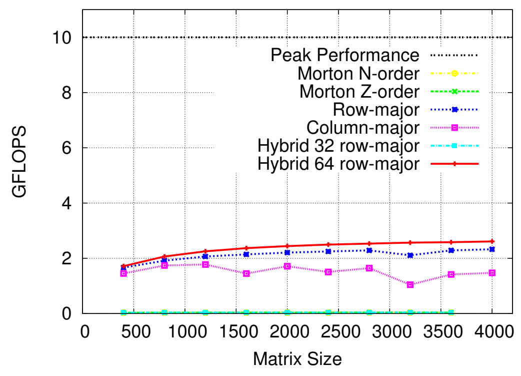


Figure 13: Cholesky decompositon with optimized Schur update (2×2 tiling)

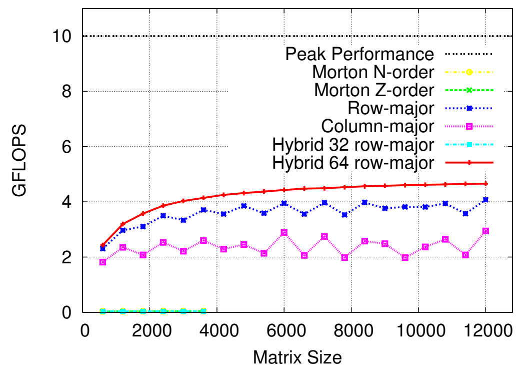


Figure 14: Cholesky decompositon with optimized Schur update (4×4 tiling)

```
                for (int j = k + 1; j < SW.num_cols (); j++)
                    SW[i][j] −= d ∗ NW[j][k];
            }
    }
```

Especially for Morton-order matrices, accessing matrix elements directly is rather expensive. To decrease the amount of address calculation, we implemented the iterative computation in terms of iterators. The performance is only slightly better and we omit the results here.

A stronger impact can be observed from casting hybrid Morton-order matrices to regular dense matrices if their block size matches the base case size. Then the performance is similar to traditional matrices while hybrid matrices with non-matching block size behave similarly to pure Morton-order matrices.

Replacing the operation $D- = \hat{C}\hat{B}^T$ on block level with a fast matrix multiplication from Section 4 leads to tremendous accelerations, especially for large matrices. In Figure 13, the unrolled matrix product with tiling of 2×2 blocks is used. Accordingly, Figure 14 depicts the performance with 4×4 tiling. It is observable that the block-recursive memory layout enables faster computations than the corresponding regular matrix. Large matrices can be decomposed with more than 4.6 GFOPS on the PowerPC. The plots also show clearly that the block-recursive memory layout guarantees more stable performance.

6. CONCLUSIONS

The major point of this paper is the ease of programming many specialized algorithms using our new extensions to the Matrix Template Library (MTL). One only needs to program a single recursive algorithm, with the convenience of recursators, to generate many runnable codes over matrices represented in many orders, including Morton-ordered, Morton-hybrid, and block-tiled variants on row- and column-major. The last are useful for hardware support, including superscalar instructions and processor pipelines. Analogously, one only need one generic program using iterators to specify a tiled iteration, independently of how a matrix is represented internally.

There are many other approaches to programming style for large matrix problems, notably ATLAS, GotoBLAS, and FLAME [24, 11, 4]. Others have drilled deeper into parallel performance than we have been able to, so far. But no one else presents such a unified approach to programming a single program over so many different matrix representations. Other techniques, moreover, require choreographed prefetching that ours avoids.

For the interesting case of dense matrices, a specific representation is determined by a typing template, based primarily on a bit mask over the integer that is the displacement from the base of a matrix to an element. It separates those bits that index the row from the rest, that index the column. It is at once a simple, powerful description that provides the matrix orderings mentioned just above, as well as hybrids of them and novel base cases like shark-tooth to suit SSE2 architectures. From these masks, alone, specific operations for row- and column-increment, quadrant indexing in quadtrees, casting conversions, and the address resolution into base-case blocks all are determined at compile time for inlined runtime performance.

Using these tools together provides a mix-and-match for operations. We have demonstrated matrix multiplication using a single recursive code on different representations, including cases where the three arguments to the multiplication are represented in three entirely different ways. The same recursators work regardless of the matrices' orders, as determined by the bit mask in their declarations. Our iterators work similarly—on any of these orders, as well as some sparse representations.

Demonstrations on the familiar problem of matrix multiplication show decent performance from pure, generically generated code, contrasting with those of THIYAGALINGAM *et al.* [23], and excellent performance when machine-specific, tuned code is provided. It is necessary *only* for the 32×32 base case of recursion (and no others)—and still driven by the abstract recursator.

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

This is not to say that one can ignore blockwise-local storage. This work is also motivated as a generic-programming approach to the CMPs. In order to use the multi-core chips, we anticipate a parallel control that allows for collaborative computations on (read-only) data by neighboring processors, even sharing L3 cache. And collections of those CMPs may still be communicating blocks of partial results within a multimemory/multiprocessor. Blockwise-local storage will help that sharing and facilitate that communication.

The recursator becomes leverage on decomposing large problems into independent, collaborating subprocesses, simply via the concept of collateral argument evaluation, borrowed from functional programming. And so these templated tools 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.

7. REFERENCES

[1] 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

[2] 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

[3] Advanced Micro Devices, Inc., Sunnyvale, CA. *AMD Core Math Library (ACML)*, 2006. http://developer.amd.com/acml.jsp

[4] P. Bientinesi, B. Gunter, and R. A. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 2007. Accepted upon revisions. http://www.cs.utexas.edu/users/pauldj/pubs/TOMS_SPD.pdf

[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 and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. Technical report, Univ. of Texas, Austin. Submittted for publication. Visited Sept. 2006. http://www.cs.utexas.edu/users/flame/pubs/GOTO_TOMS.pdf

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

[13] K. P. Lorton and D. S. Wise. Analyzing block locality in Morton-order and Morton-hybrid matrices. In P. Foglia, C. A. Prete, S. B. Bartolini, and R. Giorgi, editors, *Proc. 7th MEDEA Wkshp. MEmory performance: DEaling with Applications, systems and architecture*, pages 5–12. ACM Press, New York, Sept. 2006. http://doi.acm.org/10.1145/1166133.1166134

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

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

[16] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 2nd edition, 2001.

[17] 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

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

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

[20] 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://springerlink.metapress.com/link.asp?id=95b3nt4qngm2kj8d

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

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

[23] 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

[24] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, Jan. 2001. http://dx.doi.org/10.1016/S0167-8191(00)00087-9

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