Seven at One Stroke: Results from a Cache-Oblivious Paradigm for Scalable Matrix Algorithms*

Michael D. Adams
Computer Science Dept.
Indiana University
Bloomington, IN 47405–7104, USA
adamsmd@cs.indiana.edu

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

ABSTRACT

A blossoming paradigm for block-recursive matrix algorithms is presented that, at once, attains excellent performance measured by

- time,
- TLB misses,
- L1 misses,
- L2 misses,
- paging to disk,
- scaling on distributed processors, and
- portability to multiple platforms.

It provides a philosophy and tools that allow the programmer to deal with the memory hierarchy invisibly, from L1 and L2 to TLB, paging, and interprocessor communication. Used together, they provide a cache-oblivious *style* of programming.

Plots are presented to support these claims on an implementation of Cholesky factorization crafted directly from the paradigm in C with a few intrinsic calls. The results in this paper focus on low-level performance, including the new Morton-hybrid representation to take advantage of hardware and compiler optimizations. In particular, this code beats Intel's Matrix Kernel Library and matches AMD's Core Math Library, losing a bit on L1 misses while winning decisively on TLB-misses.

*Supported, in part, by the National Science Foundation under grants numbered ACI–0219884, EIA–0202048, and CCF-0541364.

MSPC'06 October 22, 2006, San Jose, CA, USA

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed and parallel languages*; D.1.3 [**Programming Techniques**]: Concurrent programming—*Distributed programming*; B.3.2 [**Memory Structures**]: Design Styles—*cache memories, primary memory, virtual memory*; E.2 [**Data Storage Representations**]: contiguous representations; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical algorithms and problems—*computations on matrices*.

General Terms

Design, Languages, Performance

Keywords

Cache misses, TLB, Paging, Cholesky factorization, quadtrees, Morton-hybrid, parallel processing

1. INTRODUCTION

Just as Grimm's brave little tailor used simple diligence to proceed from killing seven flies at one stroke to slaying a giant, we present a simple paradigm that at once yields seven palpable improvements to Cholesky factorization, which will apply as well to the broad class of matrix-programming problems [10].

This paper continues the presentation of a paradigm that uses quadtree representation and nested-block recursions to develop effective and scalable parallel algorithms for problems from matrix algebra [24, 25]. Here we use a few simple tools to outrun Intel's Matrix Kernel Library (MKL) on its Xeon and AMD's Core Math Library (ACML) on its Opteron, as well as tracking MKL on Intel's Pentium 4. These high-performance results derive directly from our block-recursive paradigm and its implict cache locality.

The tools form a paradigm of cache-oblivious programming for matrix problems that uses block-recursion instead of global iteration, a new Morton-hybrid order instead of row-major representation, base-case iteration and vectorization, and normalization of the plots that makes performance much easier to interpret. Detailed statistics are reported that demonstrate performance of the algorithm in detail, but the real story is the development of the algorithm from the paradigm. All code

is written in C, with minimal intrinsic calls, and tested on a cluster using OpenMPI over Infiniband.

Important lessons along the way are that L2 cache use and TLB turn out to be more important than a few L1 misses, and that cache-oblivious programming yields excellent performance without specialized coding [13].

With the advent of multi-core processors (Chip Multiprocessors, CMPs), it becomes even more important to avoid cache and TLB misses because those processors will be sharing on-chip caches and competing for bandwidth to off-chip RAM.

The remainder of this paper is in six parts. The next section explains our choice of problem, Cholesky factorization and Section 3 briefly presents some definitions. It is followed by Section 4 describing the block-recursive algorithm and Section 5 explains the style of our graphs. Section 6 presents results measured in time, cache misses, paging, parallelism, and portability. The final section offers conclusions and suggests future work.

2. WHY CHOLESKY?

The example problem is dense Cholesky factorization. Because it exhibits very tight dependencies, it is especially awkward for distributed parallel processing and cache reuse. In the literature one finds plenty of parallel algorithms for sparse or banded Cholesky factorization [17], and even for shared-memory parallelism [16], but comparatively little for both dense and distributed parallelism [18, 14, 5].

Part of that paucity is due to the the prevailing paradigm of iterative programming (`do` loops) constraining the way algorithms are conceived. That style and the captivating simplicity of row-major representation leads the programmer naturally to row-wise dot products across row pairs in the matrix. That convenience, combined with hardware support for dot product as a single operation, seduces one into treating whole rows as a unit.

The problem is that—as soon as rows are sufficiently wide—a few row traversals, say west-to-east, evict their western elements from the cache before arriving at their eastern ends. Then the cache needs to be reloaded at a low level during the next traversal. With distributed memory—really just another level in the memory hierarchy—the situation becomes even worse because the results are soon needed by other processors.

This paper shows how we use the block-recursive, cache-oblivious style to localize the base cases for efficient processing to support the communication and sharing of blocks over them. That is, the divide-and-conquer paradigm leads to a much improved decomposition of Cholesky factorization that enhances memory locality within the processors, and simplifies block communication among them. Processor scheduling naturally follows a binary decomposition, dividing the computation into stripes across large square blocks, rather than into full row traversals across the matrix.

3. DEFINITIONS

The following definitions are presented tersely but cited to sources. On first reading one may just follow the patterns in the figures.

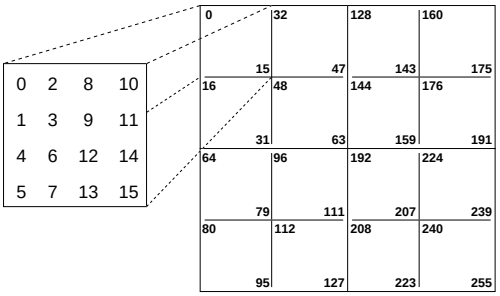

Figure 1: Morton indexing of a 16×16 matrix. Submatrices are addressed consecutively.

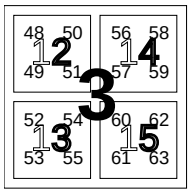

Figure 2: Ahnentafel indexing of a 4×4 matrix.

We only use 2-dimensional arrays, matrices, so the trees are quaternary. The following definitions impose indexings onto those trees.

Definition 1. [23] *The root of a matrix has* Morton-order *index* 0. *A subarray (block) at Morton-order index i is either a scalar, or it is composed of* 4 *subarrays, with indices $4i+0, 4i+1, 4i+2, 4i+3$ at the next level (respectively called* northwest*,* southwest*,* northeast*, and* southeast*).*

Figure 1 illustrates the Morton indices for a 16×16 matrix. Closely related to it is Ahnentafel indexing, which prepends two high-order 1 bits before each Morton index so that every subarray, at every level of the recursion, has a unique index. Ahnentafel indices are used to control recursion.

Definition 2. [23] *A complete matrix has* Ahnentafel *index* 3. *A submatrix (block) at Ahnentafel index a is either a scalar, or it is composed of m submatrices with indices $4a+0, 4a+1, 4a+2, 4a+3$.*

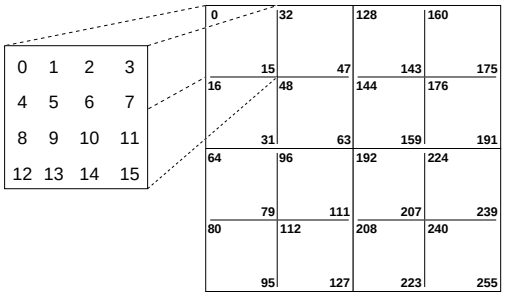

Figure 3: Morton-hybrid indexing of a 16×16 matrix with 4×4 blocks in row-major order.

Illustrated in Figure 2, Ahnentafel indices were invented by genealogists who used them to index ancestors in a binary tree [4].

DEFINITION 3. *A* Morton-hybrid *(or* hybrid*) matrix with base blocks of order* 2^r*, has its elements indexed within those blocks in row-major order. That is, a base block with Morton index* b *has its elements indexed in raster order by integers of the form* $b2^{2r}+j_1 2^r+j_2$ *where* $\{0 \le j_1, j_2 < 2^r\}$ *are a set of row-major indices within that block.*

Morton-hybrid indexing is a blend. Morton indexing of square blocks is used down to a predefined level; within those blocks, conventional row-major ordering is used. Figure 3 illustrates for a 16×16 matrix with 4×4 row-major blocks. It provides the locality implicit in Morton ordering among blocks, so that the recursive algorithms experience excellent cache reuse. Importantly, it still allows optimizing compilers to generate efficient code within the base-cases of the recursion. Typically a base block is 32×32, yielding 2^{16} flops in a rank-32 update. Even with perfect cache locality the processor still must execute them quickly because performance of these base cases is amplified across the whole solution.

In the algorithms below, the rows in a matrix are abstracted to stripes that span these row-major blocks.

DEFINITION 4. [7] *A* stripe *is a set of adjacent rows in a matrix. A* colonnade *is a set of adjacent columns.*

The stripes of interest are composed of equally sized blocks that align horizontally to one another.

4. THE ALGORITHM

4.a The recursive case

One can find iterative code in many textbooks for Cholesky factorization. However, any one of these naive implementations is likely to perform badly for all but the smallest matrices due to cache misses. During each iteration a rank-k update is performed over all remaining elements. Any cache that is smaller than the updated submatrix will then experience an excessively large number of misses as elements are evicted from cache only to be reloaded on the very next iteration.

As in previous work [24], we cast Cholesky factorization as a recursion initially over problems of size $2^s \times 2^s$. This approach results in the algorithm in Figure 4 from which the parallel version is later developed. In all cases the first argument specifies the Ahnentafel index of the submatrix to be modified.

The `doCholesky` function is the root of all recursive calls and replaces the lower-triangular matrix Q with $\hat{Q}$ such that $Q = \hat{Q}\hat{Q}^T$. Likewise `triSolve` takes the southwestern submatrix S and replaces it with $\hat{S}$ such that $S = \hat{S}N^T$, where N is the triangular Cholesky factor occupying the northwestern block. The function `schur` replaces an eastern E with $E - WN^T$, where W is a block of the same size to its west and N^T would be the block of the same size to the north of W whose transpose N lies north of W. Finally `triSchur` is a special case of `schur` where E is on Ehe main diagonal (and so lower triangular), and so $W = N^T$, halving the computational effort. The mathematics is explained in more detail in an earlier paper [24]

This recursive algorithm can be viewed as the iterative, blocked algorithm in its extreme 2×2 case on nested blocks, where there are only two iterations per loop. This style ensures that any cache reuse available to a blocked algorithm is realized at all levels by our algorithm. The difference is that arbitrarily blocked algorithms require tuning of block sizes. Ours requires no such tuning.

The advantage of the recursive paradigm is compounded with multi-level caching because an iterative, blocked algorithm needs separate tuning at each level of the memory hierarchy (L1, L2, TLB, paging, *etc.*). This comes for free with a recursive algorithm.

4.b The base case

The base case of the recurrence is selected so that the problem is small enough to fit in cache. Thus the base case code can be written under a flat memory model, ignoring cache effects. After 50 years of iteration, processor architecture favors iteration, once locality of reference can be assured, as it is here.

Reasonable performance for the base case has been achieved using the pure Morton-order representation with the classic iterative algorithms [24]. Lack of compiler support for Morton-order matrices, however, limited the performance that was available from source code.

Switching to the Morton-hybrid representation, with row-major base blocks, remedies both locality and and low-level efficiency, conveniently separating these concerns. While the recursive, outer control obliviously handles cache issues, the iterative inner control enables conventional compiler optimizations like loop unrolling, as well as hardware support such as vectorization and pipelining [2]. The outer, Morton order serializes the addressing of all outer blocks in adjacent memory.

This matrix representation has resulted in excellent performance that compares to and outperforms vendor-supplied routines. The key difference is that the vendor routines must be optimized for specific processor/cache configurations. Our compiled code is independent of those concerns, except for selecting the order of the base case and, therefore, is portable across different cache architectures.

In fact, our tests on the Pentium 4 with 2MB of L2 cache use exactly the same binary as the Xeon with only 512KB of L2. The code does not depend on the size of that cache.

4.c The parallel case

Once the algorithm has been couched in terms of recursion, parallelism is quickly derived. Looking first at `triSolve`, the recursive calls that write to the northern quadrants are completely independent of the ones for the southern quadrants. Thus the available processors can be dispatched in two groups, one north and one south, subdividing as the algorithm proceeds down the recursive calls until the uniprocessor case has been reached.

```
void doCholesky(int Q){
  if (doBase(Q)) {
    doCholeskyBase(Q);
  } else {
    doCholesky(              nw(Q));
    triSolve  (       sw(Q), nw(Q));
    triSchur  (se(Q), sw(Q)       );
    doCholesky(se(Q)              );
  }
}
```

```
void triSolve(int S, int N) {
  if (doBase(S)) {
    triSolveBase(S, N);
  } else {
    triSolve(         nw(S),     nw(N));
    schur   (ne(S),   nw(S),   sw(N)  );
    triSolve(ne(S),     se(N)         );

    triSolve(         sw(S),     nw(N));
    schur   (se(S),   sw(S),   sw(N)  );
    triSolve(se(S),     se(N)         );
  }
}
```

```
void schur(int E, int W, int N) {
  if (doBase(E)) {
    schurBase(E, W, N);
  } else {
    schur(ne(E),   nw(W),       sw(N));
    schur(ne(E),     ne(W),   se(N)  );
    schur(  nw(E),   ne(W),     ne(N));
    schur(  nw(E), nw(W),     nw(N)  );

    schur(sw(E),     sw(W),   nw(N)  );
    schur(sw(E),   se(W),       ne(N));
    schur(  se(E), se(W),     se(N)  );
    schur(  se(E),   sw(W),     sw(N));
  }
}

void triSchur(int E, int W)
{
  if (doBase(W)) {
    triSchurBase(E, W);
  } else {
    schur   (sw(E),   sw(W),   nw(W)  );
    schur   (sw(E),     se(W),   ne(W));
    triSchur(  se(E),   se(W)         );
    triSchur(  se(E), sw(W)           );
    triSchur(nw(E),     ne(W)         );
    triSchur(nw(E),   nw(W)           );
  }
}
```

Figure 4: Code for the four functions. Quadrants identified by compass points.

Consider next the many ways to make the Schur complement parallel. It is tempting to dispatch it into four processor subgroups because each quadrant of the result is independent of the others. However considering its calling context in `triSolve`, one notices that it is better to dispatch in a north/south fashion so that the processors that are allocated to each subblock will be the same in both functions. Then no communication at all will be required within `triSolve`. The result of each computation will already be local to the processor that needs it. Only after the top-most `triSolve` call from `doCholesky` is communication required because those results must be shared among the processors.

Finally, examination of `triSchur` reveals a marvelous serendipity. By not parallelizing the calls from `triSchur` and instead allowing the existing parallelism in the `schur` complement to take its course, communication can be completely avoided in `triSchur`. Since the `schur` complement doesn't perform any communication, the results from the `triSchur` will accumulate on the processors where they were calculated. The top level `triSchur`, however, is called from `doCholesky` which will initiate a recursive call to `doCholesky` on the southeast quadrant. That will, in turn, initiate a call to `triSolve` that has the exact same processor allocation as the original `triSchur`. Thus, the data doesn't need to be communicated because it already resides on the processor where it is to be used later.

Using this paradigm the communication has been reduced to one synchronization among processors after the `triSolve` call in `doCholesky` which in total amounts to $n^2/2$ data, which is the size of the entire problem. This processor schedule is also balanced at each step; exactly half of the work is given to each half of the processor group. In the shared memory context of a CMP this would lead to perfect scaling as the data communication would be essentially free and the balanced schedule ensures little wait time at the barrier that would be required after the top level `triSolve`.

On a distributed-processor cluster the scaling cannot be perfect because of communication overhead, but our paradigm still limits the amount of necessary communication. The algorithm attains the scaling shown in Figures 21 and 22. They will be discussed in more detail in the section following the next, after the plotting and testing contexts have been presented.

5. HOW TO PLOT PERFORMANCE

Except for one plot set here as an example, all our results are plotted in units of RESOURCE/FLOP, where FLOPs is the number of floating point operations in that instance of the algorithm and RESOURCE refers to the particular resource being measured, typically cycles or cache misses.

This ratio is underused in the performance community, even though it is common in the experimental algorithms and analysis communities [11, p. 28]. If the resource were time, then a more common plot is FLOPs/SECOND, the reciprocal of what we use. However, many more measures will be plotted here in the

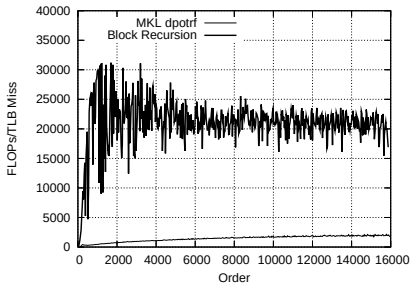

Figure 5: A poor choice for Normalization: FLOPs/TLB-miss on Xeon.

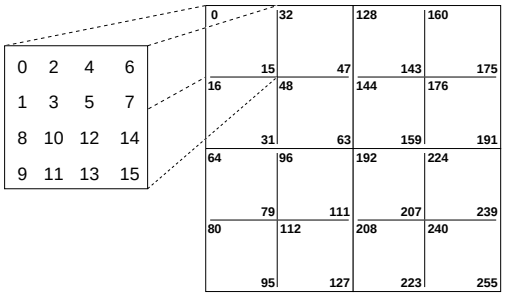

Figure 6: Morton-hybrid indexing of a 16×16 matrix with 4×4 blocks in sharktooth order.

same way, so a good plot must contrast the effects of various resources against their aggregate impact, reflected in running time.

Also, we measure time relative to the processor's clock, timing in cycles rather than seconds, because it is so common for the same processor to be marketed with different clock speeds. Measuring time in cycles also enhances comparisons across similar architectures with slightly different clock rates, like AMD's and Intel's.

All these measures grow as a function of the problem size, proportional to a cubic function for the family of algorithms addressed here. It is wrong to plot them unscaled, like the times in Figure 7, because measures from small experiments (on the left) will be smeared by the polynomial that gives impressive results for large ones (on the right). The pretty curve is nice but meaningless, so some normalization is necessary. Contrast this with our normalization in Figure 8, which exhibits an important difference on the left, with coincident asymptotes on the right. Times for the small tests are different!

Still we do not present our times in FLOPs/cycle (an impressive measure for Marketing because up is good), but rather its reciprocal, cycles/FLOP. Two important ordinates lie below that reciprocal, and the ratio of its distances to them tells a story. The first is zero, which is impossible for any timing. The second is the maximum flop rate of the machine, which is typically 1 times the clock speed or often $\frac{1}{2}$ for `double`s on SSE2 processors like the Xeon. Our time plots lie above both, and their ratios illustrate how much slower are our timings than a perfect maximum. That ratio would scale if the impact of parallelism were perfect.

Other resources must be plotted in the same way for comparison, and here this ratio shines. Consider TLB misses, later plotted as Figure 11. There is no lower bound for TLB misses and, in fact, we generate measures relatively close to zero. The Intel's MKL has about ten times the misses, and both plots, show small irregularities that might be simple operating-system overhead. If we chose FLOPs/cycle, then we should plot TLB/cycle, as in Figure 5; it is horrible for two reasons. While it does show MKL as ten times worse than our performance, one's eye is drawn away from that ratio to the terribly irregular jumps in our performance. Those jumps are entirely irrelevant, however, because they are just the same operating-system irregularities from Figure 11 artificially amplified by the tenfold reciprocal. The relatively insignificant TLB misses translate here to values closer to infinity, which are hard to appreciate on any plot.

In either case, the plot will be truncated at some maximal ordinate, too conveniently the maximum measured value. In the case of FLOPs/cycle that omits the relevant upper bound—the maximum FLOPs/cycle of the machine. In the case of TLB/cycle, the bound is infinity and will never be included. In the cases of cycles/FLOP and cycles/TLB it is easy to plot both a hard bound and the zero that displays important ratios to your eyes.

6. RESULTS

The codes were tested on an Aspen Systems cluster of eight, dual-processor 2.8GHz Intel Xeons, each with 2GB of memory, 8KB L1, and 512KB L2 cache. All the uniprocessing code was compiled using the native ICC compiler with `-O3 -ip` optimization for the Xeon. Both LAM-MPI 7.1.2b20 and OpenMPI 1.0.1 were used for communication. All data points start at order 32 and proceed in increments of 32.

Other machines were a Dell dual-processor 3.2GHz Intel Pentium 4, with 1GB of memory, 16KB L1, and 2MB L2 cache; and an ACT cluster of dual-processor 2GHz AMD Opterons with 3GB of memory, 64KB L1 cache, and 1MB L2 cache. It uses the Portland Group's PGCC 6.0 64-bit compiler also `-O3 -fastsse -Mipa=safe` but without prefetching.

6.a Uniprocessor

Figure 8 shows the block-recursive algorithm's performance versus Intel's Matrix Kernel Library (MKL). Since SSE2 instructions allow two floating point double operations to be executed at the same time, ideal performance would be at 0.5 cycles per FLOP. However SSE2 operations are destructive so the cost of register loads makes $\frac{3}{4}$ cycles per FLOP a more realistic ideal, allowing for one register load per multiply-add. Not only does our recursion come very close to that ideal but we outperform MKL for both small and large matrices. Our code reaches its asymptotes already at order 1000, while MKL only reaches its above 6000. The source of this performance is cache and especially TLB performance.

Figures 9 and 10 show our algorithm's L1 and L2

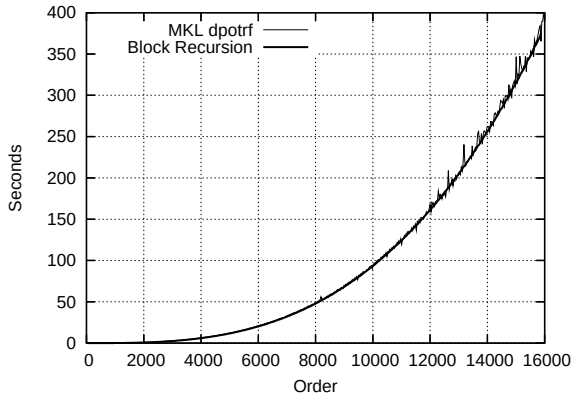

Figure 7: Unnormalized, uniprocessor time on Xeon.

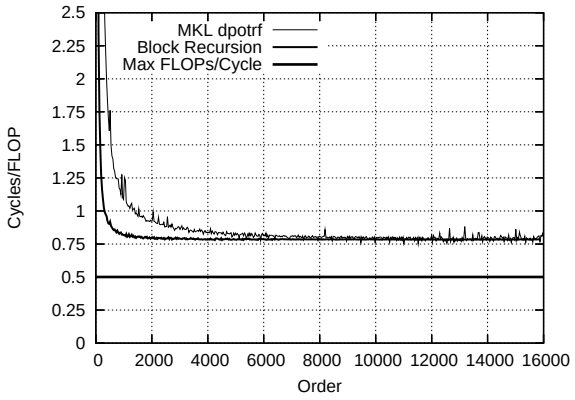

Figure 8: Normalized, uniprocessor time on Xeon.

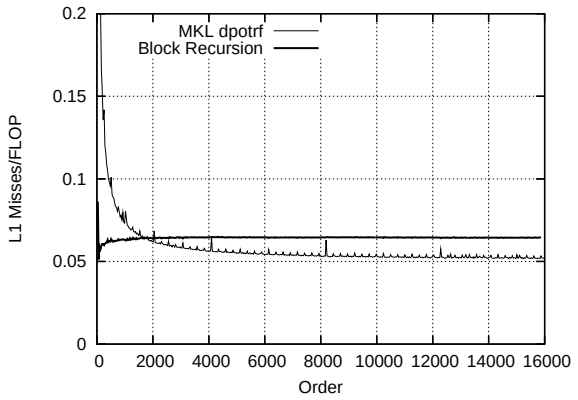

Figure 9: Uniprocessor L1 misses on Xeon.

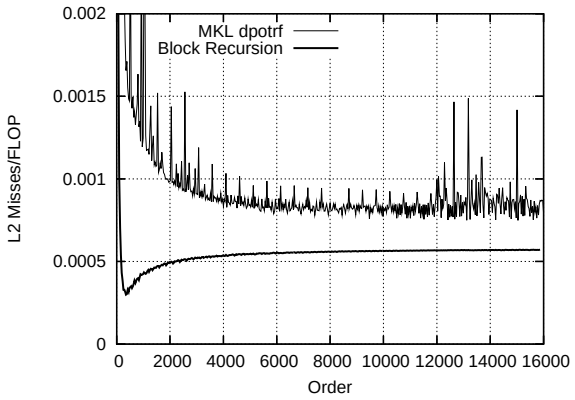

Figure 10: Uniprocessor L2 misses on Xeon.

cache misses, which compare favorably to those of MKL. The recursive algorithm has more L1 misses than MKL but fewer L2 misses.

The most striking results by far, however, are the TLB-miss rates in Figure 11. The block-recursive algorithm has an order of magnitude fewer misses than MKL. Not only are the overall rates 10 times better, but the recursive algorithm quickly reaches its asymptote at matrices of order 500, while MKL has not reached its even at order 16,000. (Our TLB-miss rate is so low that the visible misses should be dominated by the background from the operating-system interventions.) This result has a huge impact on running times because of the high costs for TLB misses, relative to L1 and L2 misses.

A qualitative difference between MKL's and block-recursion's cache-miss rates is their general shapes, for both L1 and L2. An ideal implementation would have no misses per FLOP for a tiny problem and misses would grow slowly as it overflows each cache, respectively. This behavior is indeed reflected in the plots of our block-recursive algorithm, since it starts at infinity and initially drops like a hyperbola but turns (where cache fills) and approaches its asymptote from below, as we would expect. MKL also starts with a high ratio, but it falls gently, approaching its asymptote from above, a fact that surprises us.

When addressing problems large enough to require paging, we consider memory to be just another level of cache. In Figure 12 we tested this scenario by running with the same machine configuration as before but with only 1GB of memory to force paging to disk. The costs of swapping can easily be seen above order 11,000. While there is a definite performance hit, it is moderate compared to the brick wall traditionally associated with thrashing.

Because the block-recursive code is cache-oblivious, it is portable to other platforms without adjustments for different cache sizes. Figures 14 and 13 show timing results for the Opteron and Pentium 4, respectively. Figures 15, 16, and 19 for the Opteron are similar to Figures 9, 10, and 11 for the Xeon, validating the patterns there. It is remarkable, however, that Figure 14's ACML plot requires large matrices before it reaches its asymptote, whereas the recursive algorithm descends quickly beating ACML for problems up to order 7000. Similarly, the recursive algorithm in Figures 8 and 12 descends quickly, although their MKL plots converge sooner. Just like the swapping, this fast descent to asymptote was achieved without any specialized pro-

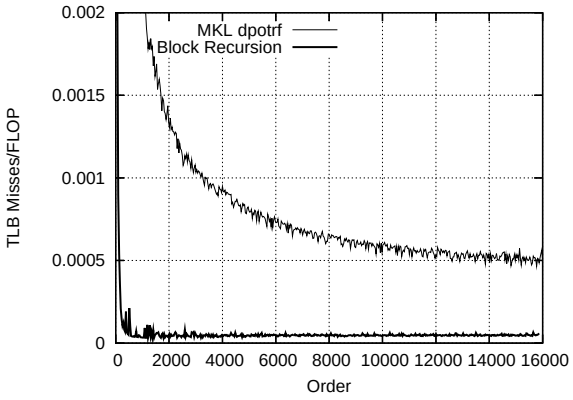

Figure 11: Uniprocessor TLB misses on Xeon.

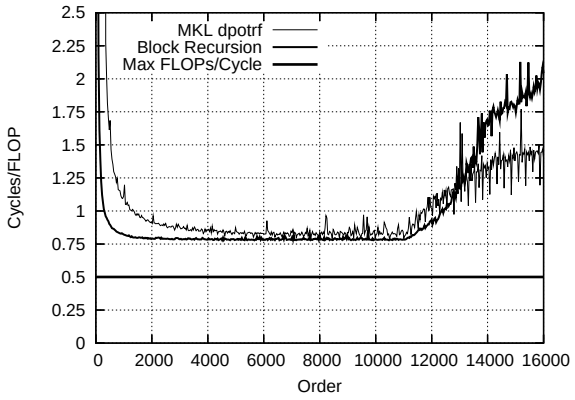

Figure 12: Uniprocessor time with 1GB memory on Xeon.

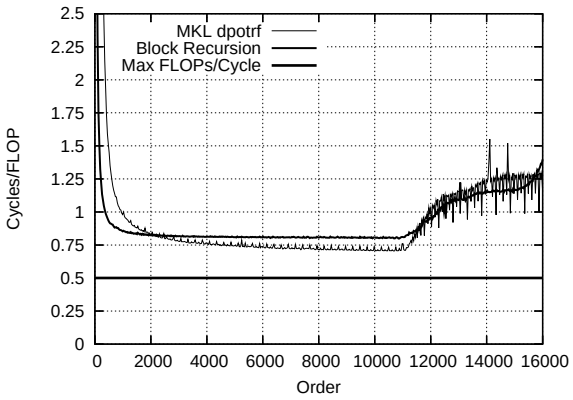

Figure 13: Uniprocessor time on Pentium 4.

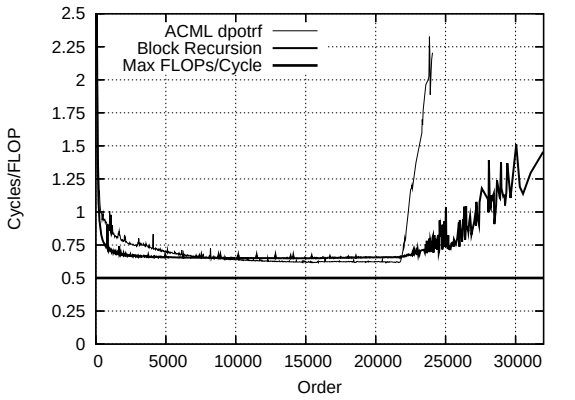

Figure 14: Uniprocessor time on Opteron.

gramming (*i.e.* prefetching), beyond the recursive style.

The Opteron uses the same source as used on the Xeon, except that the base case for `schur` was rewritten to allow for 8 more SSE2 registers, 64-bit issues, and, of course, `pragma`tic differences between compilers. The Pentium 4 tests use the exact same binary as the Xeon's, even with its different cache sizes. In these cases the recursive algorithm matches ACML in asymptote but is beaten on the Pentium 4; on both machines block recursion wins after the order grows large enough to require paging. Even though that order is much larger on the Opteron (64-bit addressing and 3GB RAM), it thrashes worse than the Pentium.

A minor change to the Morton-hybrid representation, with related C/intrinsic coding of the base case of `schur`, yields the times and L2 misses plotted in Figures 17 and 18. That representation is described here tersely in terms of *masked integers* for its cartesian indices [1]: mask `0x555557B1` defines the class of row indices and mask `0xAAAAA83E` defines the class of column indices; it is illustrated in Figure 6. That low-level change enables code improvements in the base case and reduces L2 misses that, together, make the recursive algorithm faster than ACML's `dpotrf` everywhere. This representation and its code was designed to take advantage of the Opteron's hardware.

6.b Parallelism

Our parallel tests were run on the Xeon cluster described above. They yeld the results in Figure 21 for 1, 2, 4, and 8 nodes. In order to see how well they scale it is helpful to plot them on a logarithmic scale, as shown in Figure 22. For an algorithm that scales perfectly (*i.e.* twice as fast with twice as many processors) all the plots would spread out equally distant from one another. Any deviation is scaling loss due to overhead such as interprocess communication, process synchronization, or unbalanced schedules. From these plots we conclude that our algorithm scales well. Communication overhead is present but is small. We project that in the shared memory environment of a chip multi-processor even this cost would vanish since our algorithm is implicitly balanced and has regular communication patterns.

The spikes in the plot occur at orders $256n + 32$. We are investigating the cause of these spikes, but two facts are of note here. First, we have not seen them when running on TCP in place of InfiniBand. Second, they seem to be related to an unusual stepping behavior in the L2 cache misses with parallel InfiniBand in Figure 20.

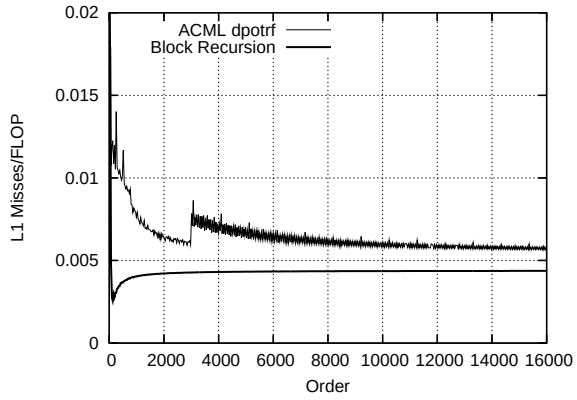

Figure 15: Uniprocessor L1 misses Opteron.

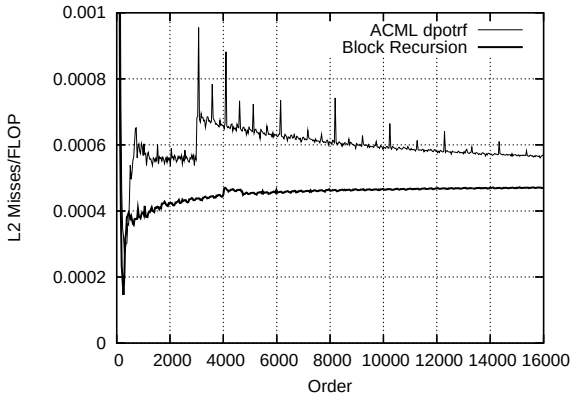

Figure 16: Uniprocessor L2 misses on Opteron.

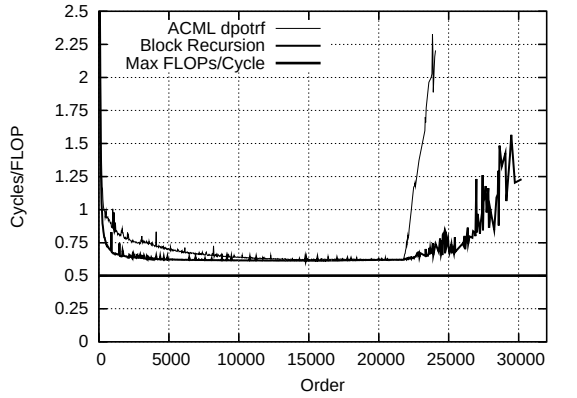

Figure 17: Uniprocessor time on Opteron, but with Figure 6's Morton-hybrid ordering.

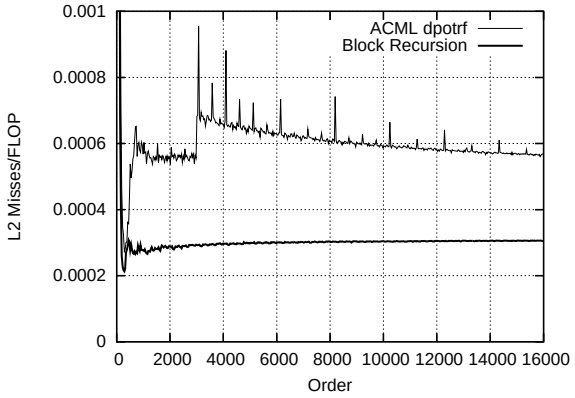

Figure 18: Uniprocessor L2 misses on Opteron, but with Figure 6's Morton-hybrid ordering.

7. CONCLUSION AND FUTURE

7.a Using the Paradigm

The main results presented here stem from careful use of the divide-and-conquer paradigm on matrices. Both the algorithm and the representation of the dense matrices presented here are blockwise recursive. As a result, locality is manifest at all levels of the memory hierarchy from interprocess communication, to paging, to L2 and L1 caches, and especially in the translation look-aside buffer (TLB).

We demonstrate time results as good as, or better than, the hand-coded manufacturers' LAPACK libraries. This is remarkable because we used almost nothing beyond C programming. The exception is some intrinsic code necessary to drive SSE2 to capacity—a pattern for superscalar loads that might have been generated by an optimizing compiler. The important feature of our code—that data migrates efficiently up and down the memory hierarchy—is realized not by register manipulations, but rather by patterns of use that are implicit in the block-recursion style. The recursive paradigm yields cache-oblivious code that uses local memory soon: that is, locally in time [8].

As a result, our code outperforms Intel's MKL on problems up to order 8000, and ties it thereafter until paging interferes. On the Pentium, ours pages better but on the Xeon, Intel's MKL pages better. In contrast, AMD's ACML thrashes badly.

It is remarkable that all these are essentially the same C code, with some differences at the lowest level if only because some are 64-bit machines and some are 32-bit. While we use Morton order to obtain our outstanding TLB misses, we have other results that indicate that the recursion still does well even on row-major representation [12].

7.b Related Work

There have been many results related to our paradigm, although no one yet seems to have demonstrated performance competing directly with manufacturers' libraries. Chatterjee *et al.* used a very similar algorithm and data structure, but accepted the overhead of translating matrices to it as part of his algorithm [3]. Elmroth *et al.* describe a similar structure but offer few results on it [5]. Valasam and Skjellum start with a similar structure but tune it specifically to a particular machine [21]. Thiyagalingam, Beckmann, and Kelly address Morton order directly, but use it without the block recursion

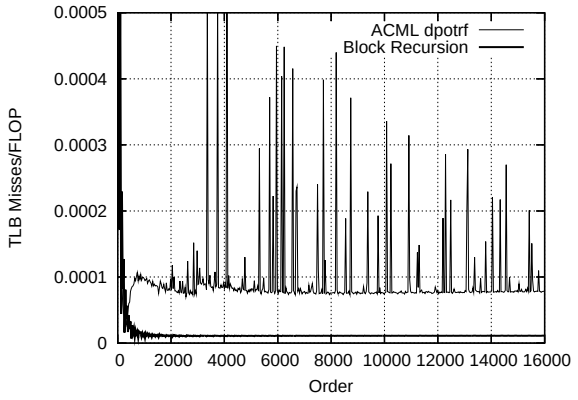

Figure 19: Uniprocessor TLB misses on Opteron.

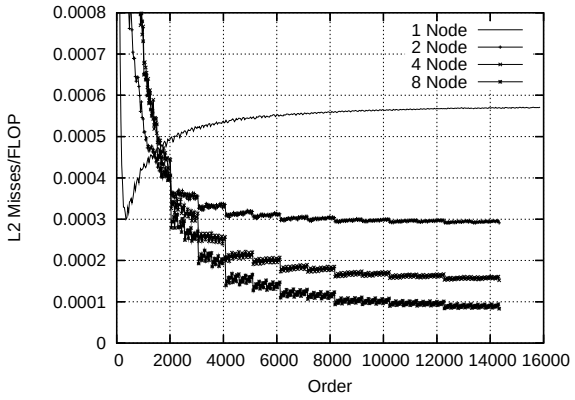

Figure 20: Unscaled, multiprocessor L2 on Xeon cluster.

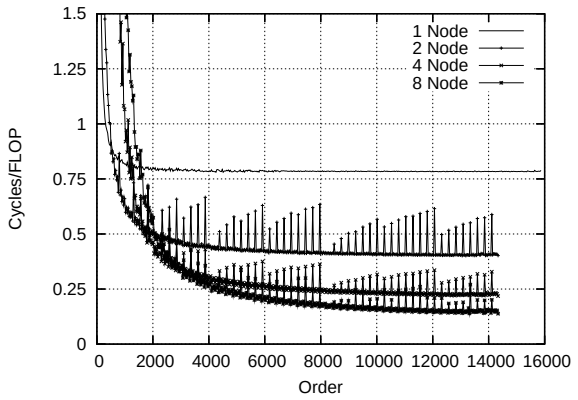

Figure 21: Multiprocessor time on Xeon cluster.

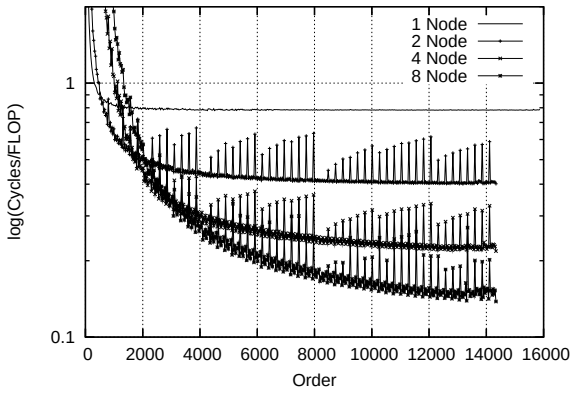

Figure 22: Multiprocessor time on Xeon cluster, logarithmically scaled.

presented here, and obtain inferior results [20].

At the other end of the scale, Goto obtains excellent performance by hand-crafting code to deal with the memory hierarchy [9, 13], but our paradigm solves that problem entirely from C. Other approaches of note are ATLAS [22] and Fisher and Probert, who settled on square blocking thirty years ago [6].

7.c Future Work

Much remains to be done. The scientific breakeven reported here is being generalized to other algorithms: direct methods, like *LU* decomposition as well as indirect methods. The paradigm carries over quite easily.

Tests on other machines have been attempted. Recent release of OpenMPI [15] and different tools for interprocess communication (*e.g.* Myrinet, TCP) should be better tested. An effort is underway to incorporate the paradigm as a package of programming tools installed in a revision of the C++ Matrix Template Library [19].

The arrival of CMPs, creating a new level in the memory hierarchy, presents a new challenge for memory-access patterns. In aggregate they have inherently constrained bandwidth to RAM. As distributed processors, however, they also have a dual limitation on bandwidth among nodes. So we need a way to program collaborating processes that can share fetched data on the CMP, and also effectively communicate results among the nodes. This paradigm offers leverage on the problem of local reference, as already demonstrated in these seven ways.

Like Grimms' brave little tailor, the nested-block data configuration, control, and communication produces seven compelling consequences for *both* kinds of parallelism at a single stroke.

8. REFERENCES

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

[2] Carr, S., and Kennedy, K. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst. 16*, 6 (Nov. 1994), 1768–1810. http://doi.acm.org/10.1145/197320.197366

[3] Chatterjee, S., Lebeck, A. R., Patnala, P. K., and Thottenthodi, M. Recursive array layouts and fast parallel matrix multiplication.

IEEE Trans. Parallel Distrib. Syst. 13, 11 (Nov. 2002), 1105–1123. http://dx.doi.org/10.1109/TPDS.2002.1058095

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

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

[6] Fischer, P. C., and Probert, R. L. Storage reorganization techniques for matrix computation in a paging environment. *Commun. ACM 22*, 7 (July 1979), 405–415. http://doi.acm.org/10.1145/359131.359134

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

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

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

[10] Gebrüder Grimm. Das tapfere schneiderlein. The Brave Little Tailor. http://gutenberg.spiegel.de/grimm/maerchen/tapfere.htm http://disneyshorts.toonzone.net/years/1938/bravelittletailor.html

[11] Johnson, D. S. A theoretician's guide to the experimental analysis of algorithms. In *Data Structures, Near Neighbor Searches, and Methodology: 5th & 6th DIMACS Implementation Challenges*, M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, Eds., vol. 59 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.* Amer. Math. Soc., Providence, 2002, pp. 215–250. http://www.research.att.com/~dsj/papers.html

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

[13] Markoff, J. Writing the fastest code, by hand, for fun: A human computer keeps speeding up chips. *The New York Times CLV*, 53,412 (2005 Nov. 28), C1, C6. http://www.nytimes.com/2005/11/28/technology/28super.html

[14] Naik, V. K., and Patrick, M. L. Data traffic reduction schemes for Cholesky factorization on asynchronous multiprocessor systems. In *ICS '89: Proc. 3rd Int. Conf. Supercomputing* (New York, 1989), ACM Press, pp. 283–294.

[15] Open MPI. `www.open-mpi.org`, Jan. 2006.

[16] OpenMP Architecture Review Board. `www.openmp.org`. Arcadia, CA, June 2005.

[17] Rothberg, E., and Gupta, A. An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Sci. Comput. 15*, 6 (Nov. 1994), 1413–1439. http://locus.siam.org/fulltext/SISC/volume-15/0915085.pdf

[18] Santos, E. E., and Chu, P.-Y. P. Efficient parallel algorithms for dense Cholesky factorization. In *ParNum '99: Proc. 4th Int. ACPC Conf.* (Berlin, 1999), P. Zinterhof, M. Vajtersic, and A. Uhl, Eds., vol. 1557 of *Lecture Notes in Comput. Sci.*, Springer, pp. 600–602. http://www.springerlink.com/link.asp?id=lceletru3wwv8q4b

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

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

[21] Valsalam, V., and Skjellum, A. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concur. Comp. Prac. Exper. 14*, 10 (2002), 805–839. http://dx.doi.org/10.1002/cpe.630

[22] Whaley, R. C., and Dongarra, J. Automatically tuned linear algebra software. In *SC'98: Proc. 1998 IEEE/ACM Conf. Supercomputing*. IEEE Computer Society Press, Los Alamitos, CA, USA, Nov. 1998, pp. 1–27. http://dx.doi.org/10.1109/SC.1998.10004

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

[24] Wise, D. S., Citro, C. L., Hursey, J. J., Liu, F., and Rainey, M. A. A paradigm for parallel matrix algorithms: Scalable Cholesky. In *Euro-Par 2005 – Parallel Processing*, J. C. Cunha and P. D. Medeiros, Eds., no. 3648 in Lecture Notes in Comput. Sci. Springer, Berlin, Aug. 2005, pp. 687–698. http://dx.doi.org/10.1007/11549468_76

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