Analyzing Block Locality in Morton-Order and Morton-Hybrid Matrices *

K. Patrick Lorton†
Current address: Schrodinger
120 West 45th Street
New York, NY 10036-4041, USA
klorton@cs.indiana.edu

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

ABSTRACT

As the architectures of computers change, introducing more caches onto multicore chips, even more locality becomes necessary. With the bandwidth between caches and RAM now even more valuable, additional locality from new matrix representations will be important to keep multiple processors busy. The default storage representations of both C and FORTRAN, row- and column-major respectively, have fundamental deficiencies with many matrix computations. By switching the storage representation from cartesian to block indices, one is able to take better advantage of cache locality at all levels from L1 to paging. This paper only changes storage representation from row-major to Morton-hybrid, and applies it to matrix multiplication. Its purpose is to show that, even with only traditional iterative algorithms, simply changing storage representation offers significant speedups.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: MultiProcessors—*Single-program, multiple-data-stream processors (SPMD)*; D.1.m [**Programming Techniques**]: Miscellaneous; E.1 [**Data Structures**]: Arrays; 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

Cholesky factorization, Morton order, quadtrees.

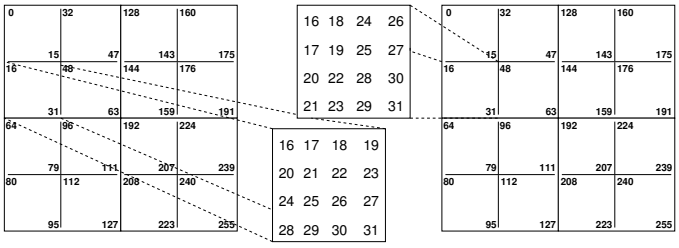

Figure 1: Morton-hybrid (left) and Morton-order (right) indexing of a 16×16 matrix.

1. INTRODUCTION

The goal of this paper is to explore the performance of the memory hierarchy when using Morton-order [25] and Morton-hybrid matrix representations. By isolating improvements attained from blockwise indexing, they improve the performance of linear-algebra applications with better use of the memory hierarchy. Improvements of blockwise indexing can be attained simply by using the OPIE compiler to convert row-major order code to Morton-order code [10]. The codes offered here are iterative, though the structure also lends itself to recursion [26].

There has been a great amount of work done in this field [6, 17, 24, 3, 20, 7]. Our contribution is to focus alone on altering storage representation without altering algorithms, and empirically to present the benefits solely due to that representation.

The closest work to ours is by Park and Prasanna, who explore another blocked representation that they call BDL, and whose only dense plots are timing results [20, 22]. They

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

†Supported, in part, by NSF under REU grant ACI-0334592.
‡Supported, in part, by NSF under grants CCR–0073491 and ACI–0219884.

MEDEA'06 2006 September 16, Seattle, Washington.

```
for      (iBlock=0; iBlock<stride; iBlock+=INNER_BLOCK_SIZE)
   for   (jBlock=0; jBlock<stride; jBlock+=INNER_BLOCK_SIZE)
     for (kBlock=0; kBlock<stride; kBlock+=INNER_BLOCK_SIZE)
            for     (i=iBlock; i<iBlock+INNER_BLOCK_SIZE; i++)
              for   (j=jBlock; j<jBlock+INNER_BLOCK_SIZE; j++)
                for (k=kBlock; k<kBlock+INNER_BLOCK_SIZE; k++)
                  c[i*stride+j] += a[i*stride+k]*b[k*stride+j];
```

Figure 2: Row-major 6-loop matrix multiplication

```
for(i = 0; i < stride; i++)
  for(j = 0; j < stride; j++)
    for(k = 0; k < stride; k++)
      c[i*stride+j] += a[i*stride+k] * b[k*stride+j];
```

Figure 3: Row-major 3-loop matrix multiplication

```
for     (iBlock=0; iBlock<stride; iBlock+=INNER_BLOCK_SIZE)
  for   (jBlock=0; jBlock<stride; jBlock+=INNER_BLOCK_SIZE)
    for (kBlock=0; kBlock<stride; kBlock+=INNER_BLOCK_SIZE)
    {
      ijBlock=evenDilate(iBlock)+oddDilate(jBlock);
      ikBlock=evenDilate(iBlock)+oddDilate(kBlock);
      kjBlock=evenDilate(kBlock)+oddDilate(jBlock);
      dgemm('n', 'n', INNER_BLOCK_SIZE, INNER_BLOCK_SIZE,
       INNER_BLOCK_SIZE, 1,
       &a[ikBlock], INNER_BLOCK_SIZE,
       &b[kjBlock], INNER_BLOCK_SIZE, 1,
       &c[ijBlock], INNER_BLOCK_SIZE);
    }
```

Figure 4: Morton-hybrid 3-loop multiplication

also present analyses and some simulation results on cache misses. In contrast, we offer results from dense tests with times and L1, L2, and Translation Lookaside Buffer (TLB) misses, as well as page faults, all in a format that can be compared by eyeball. That comparison shows, for instance, that L1 performance is not the whole story—that a severe reduction in page faults or TLB misses can upset the race [2].

The remainder of this paper is in five sections: A section defining the underlying representations, a section which describes the approach, a section which applies our technique to traditional matrix multiplication, a section which uses BLAS libraries with our approach, and finally a conclusion that points to future work.

2. DEFINITIONS

DEFINITION 1. *The base of a matrix has* Morton-order *index 0. A submatrix (block) at* Morton-order *index i is either an element (scalar), or it is composed of four submatrices, with indices 4i+0,4i+1,4i+2,4i+3.* [25].

Figure 1 illustrates the Morton indexing of a matrix in $\mathcal{U}$ order [11, 21, 19, 23]. Related to Morton indexing is *Morton-hybrid indexing* that indexes blocks in Morton order but elements within them in row- or column-major. Chatterjee used this order with limited success in 1999 [4]. Either allows cartesian indexing using a class of masked integers [1].

DEFINITION 2. *An* inner block *of order $n = 2^p$ is a square block at Morton-order m, whose elements are indexed in row-major order. The $\langle i, j \rangle^{th}$ element of that block has* Morton-hybrid *index $m4^p + i2^p + j$.*

Arithmetic on cartesian indices of both Morton-ordered and Morton-hybrid matrices is directly available in two minor cycles of any processor [1, 25], but it is not used in source code. With it we avoid all temptation to translate between representations to which others have acquiesced.

3. APPROACH

The main thrust here is to study the underlying behavior of the basic algorithms relative to locality in memory. To analyze the advantages of using block-indices, we explore traditional algorithms on different representations. This approach measures the speed of different representations without the extreme optimizations that technologies, such as hand-coded BLAS, offer [5, 13, 18]. While every representation will run significantly slower than BLAS in this approach, it will be used in a comparison to identify advantages that block indices offer. In order to analyze the efficiency of each storage representation we offer five primary measurements, L1 cache misses, L2 cache misses, page faults, TLB misses, and uniprocessor time. These measurements are collected using the monitoring tool PAPI [15].

All plots are normalized by dividing the resource measurement by the number of FLOPs in the algorithm. There are $2n^3 - n^2$ flops in an $n \times n$ matrix multiplication, and the normalization plots the leading coefficient in the idealized cubic equation for each resource; ideally it should be a constant [16]. The first four measurements are used primarily to analyze the locality and speed benefits of each representation.

For the 6-loop algorithm, a textbook algorithm written in C is used, as shown in Figure 2 [12]. The well known blocked algorithm for matrix multiplication uses three loops to traverse block products and three inner loops to traverse scalar products. It is nearly the same as the usual result of automatic loop tiling [27, 20], with the size of a tile being 32 or 128, as discussed later.

In order to convert this algorithm to Morton-order we use the OPIE Transformer [9, 10]. No knowledge of Morton-order is required to use it, and any speed-up noticed should be considered an automated optimization to row-major code.

For the second part of testing the traditional matrix-multiplication algorithm is altered to use the BLAS library as the inner loop. The purpose of this experiment is to test whether the new block representations are able to compete with row-major order under heavy optimizations. In order to use these libraries, the innermost block must be in row-major order. Unlike the computer-generated Morton code, this hybrid code was written by hand with knowledge of Morton-order. The algorithm applied to this Morton-hybrid is similar to the traditional 6-loop algorithm. Three loops iterate through the outer-Morton, while the inner three loops are replaced with a call to BLAS, as seen in Figure 4.

Testing for optimal inner block size must be methodical with manufacturers' BLAS. Empirical results have shown that inner-block size is largely dependent on the memory architecture and, in addition, the BLAS being used. That effect will be seen later in Figures 22 and 23. This is one distinct disadvantage of Morton-hybrid. While Morton-order has the advantage of being largely cache-oblivious [8], and dependent only on the architecture, Morton-hybrid requires significant cache-testing to achieve optimal results on both the architecture and the BLAS implementation.

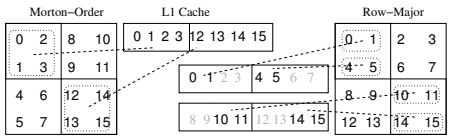

Figure 5: Morton-Order and Row-Major in Cache. A cache line holds four elements which here is either a block or a row. In the latter case, column traversals generate cache misses.

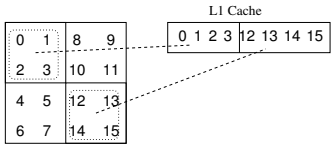

Figure 6: Morton-Hybrid in Cache. With four element cache lines the hybrid representation still allows compilers to optimize row-major inner blocks.

4. TRADITIONAL MATRIX MULTIPLICATION

4.1 3-Loop Matrix Multiplication

Almost all iterative matrix-multiplication algorithms are based on the 3-loop algorithm in Figure 3. As the stride increases, it is hampered by poor cache locality at every level of the memory hierarchy. In order to improve locality of this algorithm, the operations were rearranged into the 6-loop algorithm.

4.2 6-Loop Matrix Multiplication

The 6-loop matrix-multiplication algorithm was written to multiply a block at a time; each block uses the 3-loop algorithm as shown in Figure 2. This blocking helps at only one level of the memory hierarchy. Even with this approach as matrices grow there will eventually be an L2-cache miss on every stride down a column. Even sooner there will be L1-cache misses striding down the column. This effect is illustrated on a micro-scale in Figure 5.

It is is a very small illustration of what occurs with a hypothetical cache size of 8 integers, a cache line of 4 integers, and a matrix size of 16 integers. When traversing columns in Morton order, one must fetch only twice from memory to fill cache with new data for 16 flops, while in row-major order there must be four fetches, loading 16 integers. While this is a very primitive model of cache misses, it illustrates the problems experienced by block-algorithms with non-blocked indices. One of the limiting factors in preventing most $O(n^3)$ linear-algebra algorithms from achieving maximum speed is cache-misses and page-faults. We have seen that reducing cache-misses and page-faults yields significant speed improvement.

4.3 Morton-Hybrid Matrix Multiplication

In order to take advantage of the library and compiler optimizations on current processors, such as BLAS's, the innermost block must look like column- or row-major form. In comparison with Figure 5, Figure 6 illustrates Morton-order on the outside with a row-major inner block. By setting the inner block to row-major, we can then use all of the optimizations of BLAS for these processors, and still benefit from the low cache misses that Morton-order allows.

5. RESULTS

The figures that follow are positioned using the page-formatting language LaTeX so that they can easily be compared by eye. The figure numbering does not necessarily follow the text.

For instance, Figures 8–11 are grouped together so that the reader can see times, TLB misses, L1 misses, and L2 misses patterned to compare which has most effects on the first. Similarly, Figures 12–21 are positioned on a single page to show Opteron performance on the left and Itanium's on the right, with times in the middle and expensive page faults above. Below the timing plots are the cache misses: L1, L2, and TLB—as above—which are to be tracked together to see their impacts on the timings.

At the end are Figures 22 and 23 which explore the choice of tile/block sizes for this representation.

5.1 Hardware Description

The 64-bit machines selected for these tests are a 1.6Ghz AMD Opteron with 64KB L1 cache, 1MB L2 cache, and 1GB RAM; and a 1.5Ghz Intel Itanium with 16KB L1 cache, 256KB L2 cache, 3MB L3 cache, and 4GB RAM. Some matrix orders exceed 2^{13}, where 64-bit addressing becomes necessary.

5.2 Pure C 6-loop Multiplications

Figure 8 plots the normalized time for the pure C code for 6-loop multiplication against that available by transforming the same code onto Morton-order representation. It shows that the latter code performs twice as fast as the former, and even faster in most cases. There are more test points immediately adjacent to powers of two in order to expose the weakness of C striding that is elided with Morton order. Figure 9 exposes the TLB misses for those runs which are far fewer with Morton order, and particularly bad at 4096 and 8192 for C's row-major representation.[1] Figure 10 shows favorable and uniform L1 misses for Morton order, and particularly poor L1 misses for C, spiking at its worse times. Similarly, L2 cache misses do well in Morton order. Figure 11 shows favorable L2 misses for Morton order, and poor L2 misses for C, spiking similarly. We conclude that TLB misses account for the doubling between the times for pure C and for Opie/Morton, and that cache misses create the timing spikes on the former.

5.3 Benefits and Inefficiencies of BLAS

Intel has done an excellent job at optimizing L1, L2 and TLB misses; even with the gains seen by using this Morton-hybrid, its MKL BLAS was typically faster by approximately 4–8% until swapping. The most significant advantage that can be seen for Morton-hybrid occurs when the problems grow to a size that requires paging. While BLAS has excellent optimizations to lower L1, L2 and TLB misses,

[1]Yes, strides ought not to be powers of two but, as we avoid striding, we also avoid that old constraint.

it is severely hindered by page faults. The Morton-hybrid matrices were able to keep page faults fairly low, but straight BLAS incurred severe penalties once the algorithm exceeds RAM.

5.4 BLAS on row-major versus Hybrid on Morton-hybrid

Figures 14 and 15 compare the times purely for manufacturers' BLAS over row-major matrices running on the Opteron and Itanium with the times for our hybrid code running on our Morton-hybrid representation. Recall that our algorithms use three outer loops to isolate a block, and use the manufacturer's BLAS in the inner loop. Pure BLAS performs better than the hybrid codes on both blocks of order 32 and 128, *until* the problem size overruns RAM. When swapping begins, BLAS runs into trouble. Those problems are expanded below. We pretransposed the inner blocks of the second matrix to avoid repeated transposing, with surprising results of no speed-up. We expect that BLAS must be able to combine prefetching with the transpose to eliminate this penalty in the first place.

Figures 16 and 17 present the L1 cache misses for these problems. On the Opteron, the order-32 hybrid wins the L1 misses; on the Itanium, MKL BLAS wins, again until it begins to page. In the latter case, the order-32 L1 swaps are strikingly uniform, without the impact of paging. Similarly, Figures 18 and 19 show the L2 performance of these runs. On the Opteron, ACML BLAS wins, but on the Itanium the order-32 hybrid soundly defeats it. Even the order-128 hybrid beats BLAS's L2 performance, suggesting it has striding problems on the Itanium.

The dramatic plots are Figures 12 and 13. They show the impact of thrashing that dominates the time plots visited above. Of course, all tests must do some paging at those sizes, but BLAS suffers worse on both machines. The hybrid representation experiences a step in page faults on the Opteron, but hardly any on the Itanium because of its 4GB memory. In spite of that capacity, BLAS hits the wall.

Figure 20 exhibits the TLB misses on the Opteron, where hybrid 128 won, and BLAS came in second. The analogous plot for the Itanium is Figure 21 and shows 128 winning, as well. This last set of plots, using BLAS for inner-blocks on the Itanium, was very irregular, with jumps not well shown here; they fill the frame. Instead, these plots show median results from five runs at each point for each case. In fact, the results for BLAS and 128-blocks are smooth, with the latter deviating only rarely (one of five), but the medians of the 32-blocks still remain ragged. Ultimately, the TLB misses for 128-blocks are stable and, at worst, 2 orders of magnitude less that BLAS's; 3.5 orders less on small problems. The times correlate well.

Finally, Figures 22 and 23 present timing experiments done early on to select the block sizes for the hybrid tests. We tested inner block sizes of orders 16, 32, 64, 128, and 256 and timed them to determine which ones to race against BLAS. Ironically, 32 was the net winner on the Opteron, with 16 doing poorly on both. Beyond that, the results are difficult to explain. On the Opteron 32, 64, and 128 were close. On the Itanium 128 and 256 did well, suggesting that larger is better, but 16 and 64 did worst. Still on the Itanium, 32 is not terribly bad but 128 seems best.

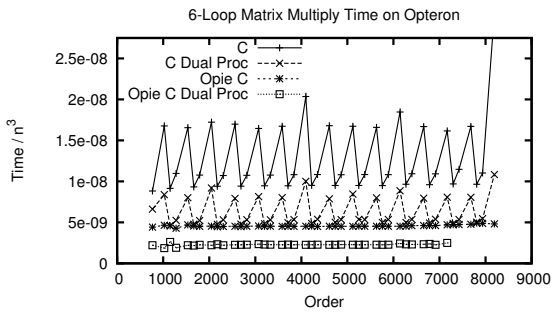

Figure 7: Scaling of times of two algorithms on shared-memory dual Opterons.

5.5 TLB Results

One of the more interesting results was the impact of TLB misses on time. While plots of L1 and L2 cache misses show spikes in the traditional 6-loop algorithm, TLB misses adhere strongly to cubic growth (Their plot is flat) and are extremely low for Morton and Morton-hybrid representations. Although the exact correlation between TLB misses and time has not been revealed, the plots suggest that our performance can be attributed to TLB misses (Figure 9), as we perform favorably (Figure 8) even when there are no significant L1 or L2 differences (Figures 10 and 11).

5.6 Simple Scaling with Parallelism

Figure 7 exhibits scaling readily available from the algorithms, described above. It shows two things.

First, all these algorithms scale from one to two processors, here single and dual shared-memory Opterons, and elsewhere to distributed machines [26]. The data structure ensures memory locality on any single machine, and simplifies block transfers among many because blocks have contiguous addresses.

Second, it shows that the uniprocessor times for both pure C and for Opie-transformed C source scale very well to multiple processors. Remarkably, the Opie-transformed C code is as fast as (and faster than) the dual-processor compiled C. The ragged performance of raw C code, scaled perfectly in its parallel versions, contrasts poorly with kpie's flat times.

6. CONCLUSION

The test problem on 64-bit processors is that old chestnut: square, dense matrix multiplication. Remarkable improvements over the manufacturer's BLAS3 `dgemm` are demonstrated in TLB, L1/L2 cache misses, page faults, and *time* on the AMD Opteron *just* by changing the data structure. These scale to the shared-memory tests, as well. The most striking improvement is in page faults, the slowest level of the memory hierarchy.

Intel's Itanium BLAS performs more favorably than AMD's BLAS up to and after paging, but our representation nearly met its times and locality by using its `dgemm` for small, inner blocks. Page faults also are the major win on the Itanium—after exceeding its 4GB RAM.

There are many results on block recursive matrices [6, 17, 24, 3, 20, 7] In contrast to earlier reports, these larger experiments with Morton-hybrid indexing on 64-bit processors show that excellent memory-bandwidth is available simply

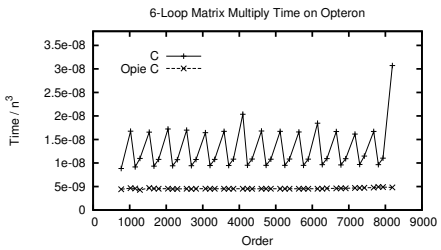

Figure 8: Times on Opteron for 6-Loop matrix multiplication.

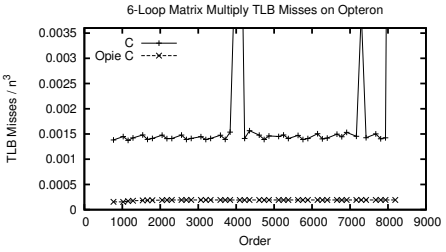

Figure 9: TLB misses on Opteron for 6-Loop matrix multiplication.

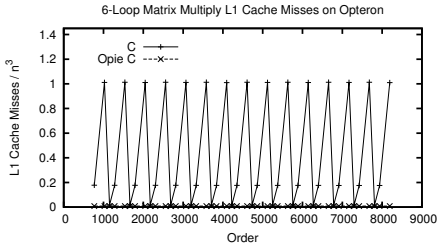

Figure 10: L1 Cache Misses on Opteron for 6-Loop matrix multiplication.

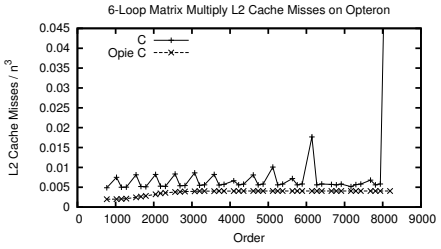

Figure 11: L2 Cache misses on Opteron for 6-Loop matrix multiplication.

by changing the representation of matrices. Our results, particularly on TLB misses, should be distinguished from those of Goto and van de Geijn who attain excellent results with low-level programming [13, 14]. Aside from tiling the algorithm, all we did was to change the data structure.

Future work is under way to use the block-recursive data structures, themselves, to deliver even more locality through block-recursive algorithms. No recursion is used here; these are conventional loops on the newer representations. Parallel performance on distributed memories is also being studied, where interprocess communication introduces new overheads. A serious overhead of distributed memory is the impact of the interfacing software, like OpenMPI, on already cached data.

Our overall conclusion, however, is that Morton-order delivers better performance via its inherent block locality. Even if the source code has been written with loops for row-major C representation, we still realize better performance simply by fitting inner blocks to Morton's or by transforming row-major code with Opie.

7. 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] Adams, M. D., and Wise, D. S. 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*. ACM Press, New York, Oct. 2006, pp. 41–50. http://doi.acm.org/10.1145/1178597.1178604

[3] Bader, M., and Zenger, C. Cache oblivious matrix multiplication using an element ordering based on the Peano curve. In *Parallel Processing and Applied Mathematics* (Berlin, 2006), vol. 3911 of *Lecture Notes in Comput. Sci.*, Springer, pp. 1042–1049. http://dx.doi.org/10.1007/11752578_126

[4] 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

[5] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw. 16*, 1 (Mar. 1990), 1–17. http://doi.acm.org/10.1145/77626.79170

[6] Fox, G. C. A graphical approach to load balancing and sparse matrix-vector multiplication. In *Numerical Algorithms for Modern Parallel Architectures*, M. Schultz, Ed., vol. 13 of *IMA Vol. in Math. & Appl.* Springer, New York, 1988, pp. 37–61.

[7] Fraguela, B. B., Guo, J., Bikshandi, G., Garzarán, M. J., Almási, G., Moreira, J., and Padua, D. The hierarchically tiled arrays programming approach. In *LCR '04: Proc. 7th Wkshp. Languages, Compilers, and Run-Time Support for Scalable Systems*, vol. 81 of *ACM Int. Conf. Proc. Series*. ACM Press, New York, 2004, pp. 1–12. http://doi.acm.org/10.1145/1066650.1066657

[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] Gabriel, S. T., Chenoweth, B., Lorton, K. P., Carlson, M., and Wise, D. S. *The Opie Compiler Distribution*. Indiana University, Bloomington, IN, Apr. 2005. http://www.cs.indiana.edu/~dswise/Opie/distribution.html

[10] Gabriel, S. T., and Wise, D. S. The Opie compiler from row-major source to Morton-ordered matrices. In *Proc. 3rd Wkshp. on Memory Performance Issues*, J. Carter and L. Zhang, Eds. ACM Press, New York, 2004, pp. 136–144. http://doi.acm.org/10.1145/1054943.1054962

[11] Gargantini, I. An effective way to represent quadtrees. *Commun. ACM 25*, 12 (Dec. 1982), 905–910. http://doi.acm.org/10.1145/358728.358741

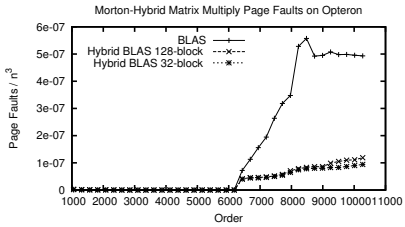

Figure 12: Page Faults on Opteron for hybrid matrix multiplication.

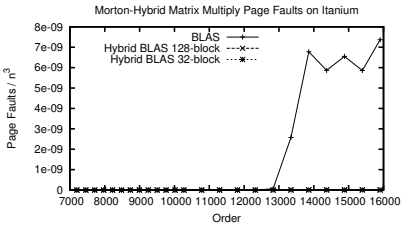

Figure 13: Page Faults on Itanium for hybrid matrix multiplication.

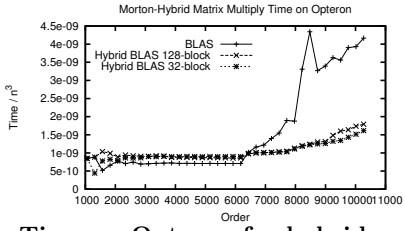

Figure 14: Time on Opteron for hybrid matrix multiplication.

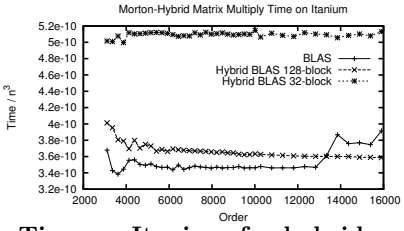

Figure 15: Time on Itanium for hybrid matrix multiplication.

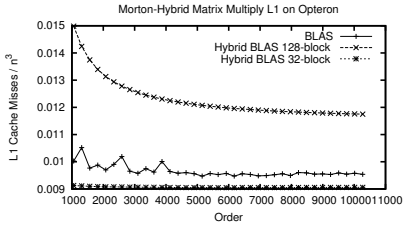

Figure 16: L1 cache misses on Opteron for hybrid matrix multiplication.

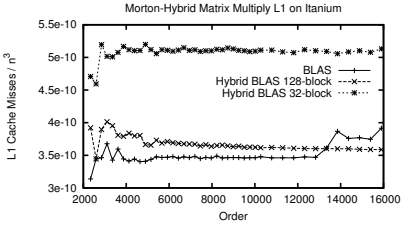

Figure 17: L1 cache misses on Itanium for hybrid matrix multiplication.

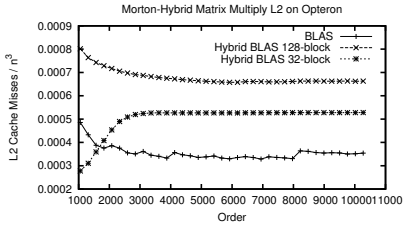

Figure 18: L2 cache misses on Opteron for hybrid matrix multiplication.

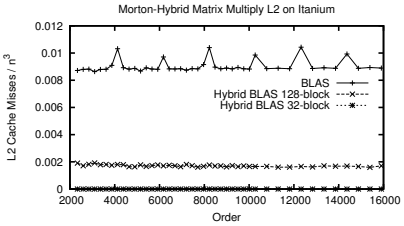

Figure 19: L2 cache misses on Itanium for hybrid matrix multiplication.

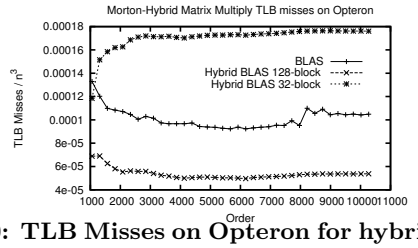

Figure 20: TLB Misses on Opteron for hybrid matrix multiplication.

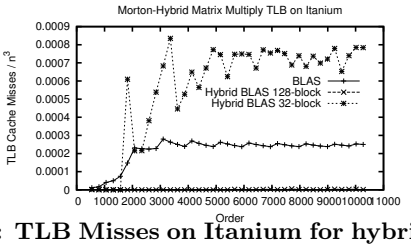

Figure 21: TLB Misses on Itanium for hybrid matrix multiplication.

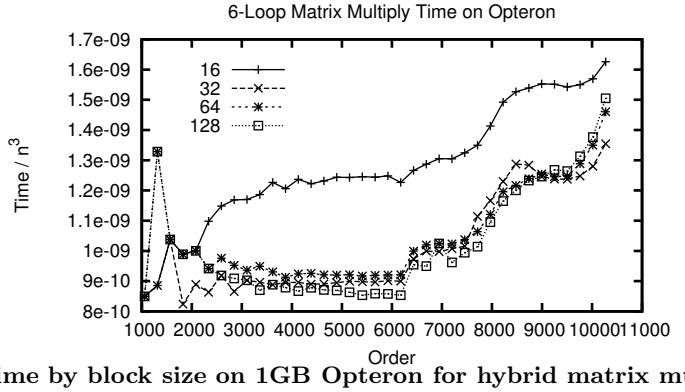

Figure 22: Time by block size on 1GB Opteron for hybrid matrix multiplication.

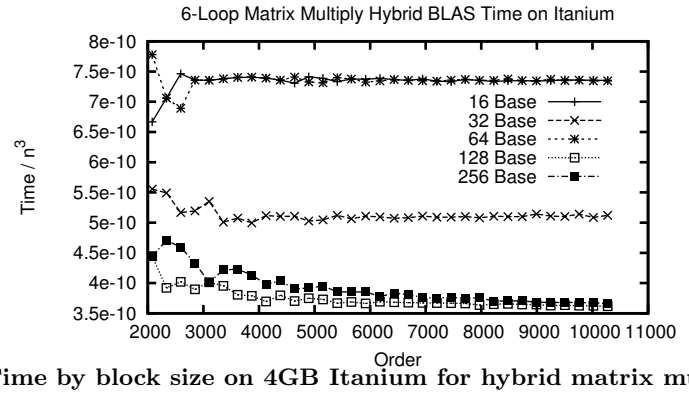

Figure 23: Time by block size on 4GB Itanium for hybrid matrix multiplication.

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

[13] 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

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

[15] Innovative Computing Laboratory, Univ. of Tennessee. *Performance Application Programming Interface (PAPI)*. Knoxville, TN, Dec. 2005. http://icl.cs.utk.edu/papi/

[16] 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

[17] Li, K. Scalable parallel matrix multiplication on distributed memory parallel computers. In *14th Int. Parallel and Distributed Processing Symp. (IPDPS'00)*. IEEE Computer Soc. Press, Washington, DC, May 2000, pp. 307–314. http://dx.doi.org/10.1109/IPDPS.2000.846000

[18] 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

[19] Morton, G. M. A computer oriented geodetic data base and a new technique in file sequencing. Tech. rep., IBM Ltd., Ottawa, Ontario, Mar. 1966. % pagebreak[4]

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

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

[22] Sang Park, J., Penner, M., and Prasanna, V. K. Optimizing graph algorithms for improved cache performance. *IEEE Trans. Parallel Distrib. Syst. 15*, 9 (Sept. 2004), 769–782. http://dx.doi.org/10.1109/TPDS.2004.44

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

[24] 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

[25] 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/content/0pc0e9gfk4x9j5fa

[26] 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

[27] Wolfe, M. More iteration space tiling. In *Proc. Supercomputing '89*. ACM Press, New York, NY, USA, Nov. 1989, pp. 655–664.