

An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays

Jeyarajan Thiyyagalingam, Olav Beckmann, Paul H. J. Kelly

Department of Computing, Imperial College
180 Queen's Gate, London SW7 2AZ, U.K.
{phjk, ob3, jeyan}@doc.ic.ac.uk

Abstract. Morton layout is a compromise storage layout between the programming language mandated layouts row-major and column-major, providing substantial locality of reference when traversed in either direction. This paper explores the performance of Morton, row-major and column-major layouts in detail on some representative architectures. Using a small suite of dense kernels working on two-dimensional arrays, we have carried out an extensive study of the impact of poor array layout and of whether Morton layout can offer an attractive compromise. Whether Morton layout is better than traversing a column-major array in row-major order (or vice versa) depends on problem size and architecture. Morton layout generally leads to much more consistent performance and only a small improvement in its performance could make it an attractive alternative.

1 Introduction

Two-dimensional arrays are generally arranged in memory in row-major order (for C, Pascal etc) or column-major order (for Fortran). Modern processors rely heavily on caches and prefetching, which work well when the access pattern matches the storage layout. Sophisticated programmers, or occasionally sophisticated compilers, match the loop structure to the language's storage layout in order to maximise spatial locality. Unsophisticated programmers do not, and the performance loss is often dramatic — a factor of 10 or more. In this paper we study the Morton storage layout (for background and history see [2, 11]).

Morton layout is a compromise between row-major and column-major, with some spatial locality whether traversed in row-major or column-major order — but in neither case is spatial locality as high as the best case for row-major or column-major. Further, the way that array elements are stored requires fairly complicated address calculation. So, should language implementors still consider providing support for Morton layout for multidimensional arrays? In this paper, we explore and analyse this question and provide some qualified answers.

Perhaps controversially, we confine our attention to “naively” written codes, where a mismatch between access order and layout is reasonably likely. We also assume that the compiler does not help, neither by adjusting storage layout, nor by loop nest restructuring such as loop interchange or tiling¹. Naturally, we fervently hope that users will

¹ In the examples which we studied, we have not seen evidence of the compiler either interchanging loops or changing storage layout in order to improve the stride of memory access.

be expert and that compilers will successfully analyse and optimise the code, but we recognise that very often, neither is the case. In this paper, we evaluate the hypothesis that Morton layout, implemented using lookup tables, is a useful compromise between row-major and column-major layout. We present extensive experimental results using five simple numerical kernels, running on five different processors (Section 4).

2 Related work

In our earlier paper [10], we argued that Morton layout is an effective compromise storage layout, with the evidence of experimental data on various architectures for various kernels, on power-of-two problem sizes. Our later work on selected non-power-of-two sizes (presented at the CPC workshop in January 2003) gave similar results. This paper improves on our earlier work:

- We use the best available compilers for each of the five processors, using the compiler flags chosen by the vendors for their SPEC CFP2000 (base) benchmark reports [9] (see Table 3 in Section 4).
- We present an extensive and systematic study using all problem sizes in the range 100×100 to 2048. This shows a number of interesting effects, and Morton layout appears less attractive. However, as we discuss at the end of the paper, further improvements to the performance of Morton layout might be possible.

In [10], we included a discussion about related work in the area of compiler techniques [4–6, 12], blocked and recursively-blocked array layouts [2, 3, 11].

3 Background

Lexicographic array storage. For an $M \times N$ two dimensional array A , a mapping $S(i, j)$ is needed, which gives the memory offset at which array element $A_{i,j}$ will be stored. Conventional solutions are row-major (for example in C and Pascal) and column-major (as used by Fortran) mappings expressed by

$$S_{rm}^{(M,N)}(i, j) = N \times i + j \quad \text{and} \quad S_{cm}^{(M,N)}(i, j) = i + M \times j$$

respectively. We refer to row-major and column-major as lexicographic, i.e. elements are arranged by the sort order of the two indices (another term is “canonical”).

Blocked array storage. Traversing a row-major array in column-major order, or vice-versa, leads to poor performance due to poor spatial locality. An attractive strategy is to choose a storage layout which offers a compromise between row-major and column-major. For example, we could break the $M \times N$ array into small, $P \times Q$ row-major sub-arrays, arranged as a $M/P \times N/Q$ row-major array. We define the blocked row-major mapping function (this is the 4D layout discussed in [2]) as:

$$S_{brm}^{(M,N)}(i, j) = (P \times Q) \times S_{rm}^{(M/P, N/Q)}(i/P, j/P) + S_{rm}^{(P,Q)}(i \% P, j \% Q)$$

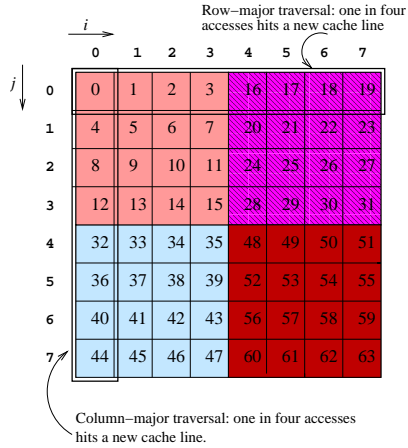


Fig. 1. Blocked row-major (“4D”) layout $S_{brm}^{(8,8)}(i, j)$ with block-size $P = Q = 4$. The diagram illustrates that with 16-word cache lines, illustrated by different shadings, the cache hit rate is 75% whether the array is traversed in row-major or column-major order.

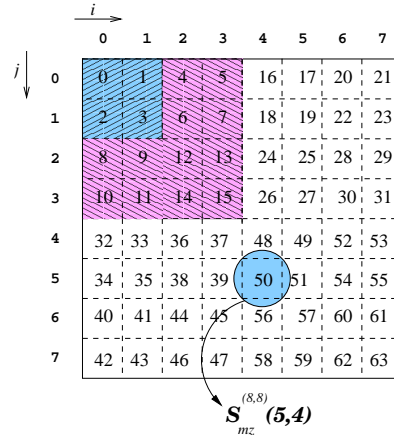


Fig. 2. Morton storage layout for an 8×8 array. Location of element $A[5, 4]$ is calculated by interleaving “diluted” representations of 5 and 4 bitwise: $\mathcal{D}_0(5) = 100010_2$, $\mathcal{D}_1(4) = 010000_2$. $S_{mz}(5, 4) = \mathcal{D}_0(5) \mid \mathcal{D}_1(4) = 110010_2 = 50_{10}$.

For example, consider 16-word cache blocks and $P = Q = 4$, as illustrated in Figure 1. Each block holds a $P \times Q = 16$ -word subarray. In row-major traversal, the four iterations $(0, 0)$, $(0, 1)$, $(0, 2)$ and $(0, 3)$ access locations on the same block. The remaining 12 locations on this block are not accessed until later iterations of the outer loop. Thus, for a large array, the expected cache hit rate is 75%, since each block has to be loaded four times to satisfy 16 accesses. The same cache hit rate results with column-major traversal, i.e. when the loop structure is “do $i \dots$ do j ” rather than the “do $j \dots$ do i ” loop of row-major traversal.

Recursive blocking. Modern computer systems rely on a TLB to cache address translations: a typical 64-entry data TLB with 8KByte pages has an effective span of $64 \times 8 = 512KB$. Unfortunately, as illustrated in Figure 3, if a blocked row-major array is traversed in column-major order, only one subarray per page is usable. Thus, we find that the blocked row-major layout is still biased towards row-major traversal. We can overcome this by applying the blocking again, recursively. Thus, each 8KByte page (1024 doubles) would hold a 16×16 array of 2×2 -element subarrays.

Modern systems often have a deep memory hierarchy, with block size, capacity and access time increasing geometrically with depth [1]. Blocking should therefore be applied for each level. Note, however, that this becomes very awkward if larger block sizes are not whole multiples of the next smaller block size.

Morton-order layout is an unbiased compromise between row-major and column-major. The key property which motivates our study of Morton layout is the following:

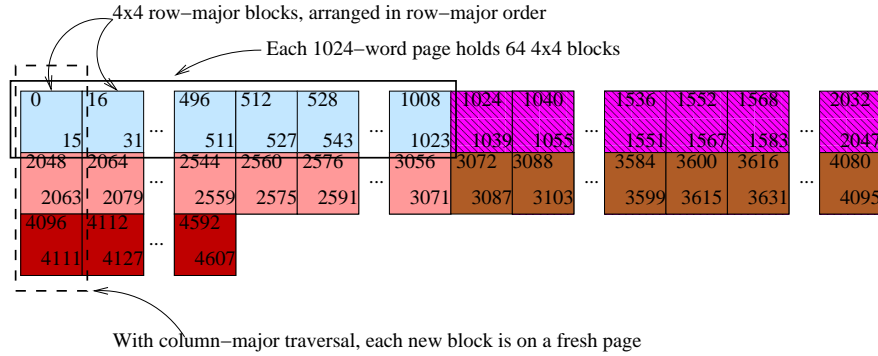


Fig. 3. Blocked row-major layout for large array. If a large blocked row-major array is traversed in column-major order, only one subarray per page is usable. The diagram shows an array with rows of 2048 doubles, using the blocked row-major layout with 4×4 blocks. Each 8KByte page holds 1024 doubles, in 64 blocks. When traversed in row-major order, one fresh page is accessed every 256 accesses (a hit rate of $1 - 1/256 = 99.6\%$), but when traversed in column-major order, a fresh page is accessed every 4 accesses (a hit rate of $1 - 1/4 = 75\%$).

	Row-major layout	Morton layout	Column-major layout
32B cache line	75%	50%	0%
128B cache line	93.75%	75%	0%
8KB page	99.9%	96.875%	0%

Table 1. Theoretical hit rates for row-major traversal of a large array of double words on different levels of memory hierarchy. Possible conflict misses or additional hits due to temporal locality are ignored. This illustrates the compromise nature of Morton layout.

- Given a cache with any even power-of-two block size, with an array mapped according to the Morton order mapping S_{mz} , the cache hit rate of a row-major traversal is the same as the cache-hit rate of a column-major traversal.
- This applies given any cache hierarchy with even power-of-two block size at each level. This is illustrated in Figure 2.
- The cache hit rate for a cache with block size 2^{2k} is $1 - (1/2^k)$.

For cache blocks of 32 bytes (4 double words, $k = 1$) this gives a hit rate of 50%. For cache blocks of 128 bytes (16 double words, $k = 2$) the hit rate is 75% as illustrated earlier. For 8KB pages (1024 words, $k = 5$), the hit rate is 96.875%. In Table 1, we contrast these hit rates with the corresponding theoretical hit rates that would result from row-major and column-major layout. Notice that traversing the same array in column-major order would result in a swap of the row-major and column-major columns, but leave the hit rates for Morton layout unchanged.

Morton-order address calculation. The offset $S_{mz}^{(M,N)}(i, j)$ of an element in Morton layout can be calculated by bitwise interleaving of the binary digits representing i and j . This can be calculated incrementally using "dilated arithmetic", but in our earlier pa-

MMijk	Matrix multiply, ijk loop nest order (usually poor due to large stride)
MMikj	Matrix multiply, ikj loop nest order (usually best due to unit stride)
Jacobi2D	Two-dimensional four-point stencil smoother
ADI	Alternating-direction implicit kernel, ij,ij order
Cholesky	K-variant (usually poor due to large stride)

Table 2. Numerical kernels used in our experimental evaluation.

System	Processor	Operating System	L1/L2/Memory Parameters	Compiler and Flags Used
Alpha Compaq AlphaServer ES40	Alpha 21264 (EV6) 500MHz	OSF1 V5.0	L1 D-cache: 2-way, 64KB, 64B cache line L2 cache: direct mapped, 4MB Page size: 8KB Main Memory: 4GB RAM	Compaq C Compiler V6.1-020 -arch ev6 -fast -O4
Sun SunFire 6800	UltraSparcIII(v9) 750MHz	SunOS 5.8	L1 D-cache: 4-way, 64KB, 32B cache line L2 cache: direct-mapped, 8MB Page size: 8KB Main Memory: 24GB	Sun Workshop 6 -fast -xcrossfile -xalias_level=std
PIII	PentiumIII Coppermine 450MHz	Linux 2.4.20	L1 D-cache: 4-way, 16KB, 32B cache line L2 cache: 4-way 512KB, sectored 32B cache line Page size: 4KB Main Memory: 256MB SDRAM	Intel C/C++ Compiler v7.00 -xK -mp -ipo -O3 -static
P4	Pentium 4 2.0 GHz	Linux 2.4.20	L1 D-cache: 4-way, 8KB, sectored 64B cache line L2 cache: 8-way, 512KB, sectored 128B cache line Page size: 4KB Main Memory: 512MB DDR-RAM	Intel C/C++ Compiler v7.00 -xW -mp -ipo -O3 -static
AMD	AMD Athlon XP 2100+ 1.8GHz	Linux 2.4.20	L1 D-Cache: 2-way, 64KB, 64B cache line L2 cache: 16-way, 256KB, 64B cache line Page size: 4KB Main Memory: 512MB DDR-RAM	Intel C/C++ Compiler v7.00 -xK -mp -ipo -static

Table 3. Cache and CPU configurations used in the experiments. Compilers and compiler flags match those used by the vendors in their SPEC CFP2000 (base) benchmark reports [9].

per [10], we found a simple table lookup scheme works remarkably well. We use two tables, D_0 and D_1 , which map i and j to their dilated representations (that is, bitwise interleaved with zeroes, where $\mathcal{B}(D_0(i)) = 0i_{n-1} \dots 0i_0$ where $\mathcal{B}(x)$ is the binary representation of x and $D_1(i) = D_0(i) \ll 1$). The Morton offset $S_{mz}^{(M,N)}(i, j) = D_0(i) + D_1(j)$. The tables are small and are traversed with unit stride. In this paper, we exclusively use the table lookup scheme.

4 Experimental setup and experimental results

Benchmark kernels and architectures. To test our hypothesis that Morton layout, implemented using lookup tables, is a useful compromise between row-major and column-major layout experimentally, we have collected a suite of simple implementations of standard numerical kernels operating on two-dimensional arrays and carried out experiments on five different architectures. The benchmarking kernels used are shown in Table 2 and the platforms in Table 3.

Problem sizes. As mentioned in Section 2, our previous paper [10] reported performance results for power-of-two problem sizes. For this paper, we decided to carry out

	Adi		Cholk		Jacobi2D		MMijk		MMikj	
	<i>min</i>	<i>max</i>	<i>min</i>	<i>max</i>	<i>min</i>	<i>max</i>	<i>min</i>	<i>max</i>	<i>min</i>	<i>max</i>
Alpha	27.0	84.5			24.0	167.1	6.0	139.5	42.7	177.0
Athlon	43.8	210.4	8.8	308.5	150.6	1078.6	9.5	262.5	118.2	884.2
P3	13.7	46.6	4.1	42.2	38.7	122.3	15.5	92.3	45.5	173.5
P4	46.2	134.1	4.8	266.1	159.6	1337.3	12.6	147.3	281.4	939.1
Sparc	11.3	54.4	3.5	78.4	33.2	139.2	4.8	131.9	22.7	142.8

Table 4. Baseline performance of various kernels on different systems. For each kernel, for each machine, we show the performance range in MFLOPs for row-major array layout over all problem sizes covered in our experiments (as shown in Figures 4–7).

an exhaustive study, collecting performance data, where possible, for all problem sizes between 100×100 and 2048×2048 . In some cases, the running-time of the benchmarks was such that we were not able yet to collect data up to 2048×2048 . In those cases, we report data up to 1024×1024 ; however, we are continuing to collect measurements. In all cases, we used square arrays.

Performance results. The performance numbers we report in this paper are all based on the *median* of measurements taken. See [7] for more details on experimental methodology/framework. Table 4 shows the baseline performance achieved by each machine using standard row-major layout. In figures 4–7 we show our interesting/important results in detail. The full range of results and annotations can be found in [7].

On nearly all systems, the results clearly show the impact of L2 cache and TLB span on overall performance. Frequently, when either the whole working set or some part thereof exceeds the capacity of a particular level of memory hierarchy, a substantial drop in performance can be observed. For example, a sudden drop in the performance of *MMijk* with the column-major layout on *Alpha*, near the problem size 350 coincides with the working set exceeding the size of the TLB span. (Alpha has 128-entry Data TLB, each entry pointing to an 8KB page: This matches the size of a 362×362 array of doubles). Similar observations can be made for other levels of the memory hierarchy as well. Further, row-major and column-major layouts show wide variations in performance with small changes in problem sizes whereas the performance of Morton layout remains very consistent. Although padding the length of the rows/columns of an array can significantly improve performance, the amount of padding required needs to be chosen very carefully.

For each experiment/architecture pair, we state whether Morton layout is a useful compromise between row-major and column-major in this setting by annotating the figures with *win*, *lose*, etc. As an overview, we record *wins* for

- Adi: Alpha, P3 and Sparc over column-major but not over row-major.
- Jacobi2D: Alpha, Sparc over column-major but not over row-major.
- MMijk: Alpha, Sparc over column-major but not over row-major.

- MMijk: Alpha, Sparc over both row-major and column-major.

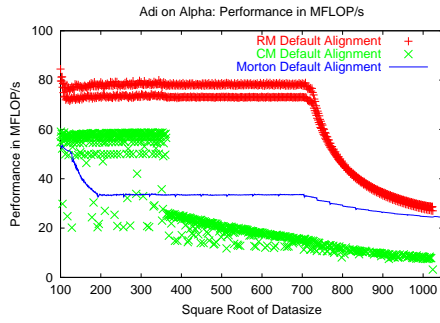
This suggests that Morton layout performs well on machines with large L2 caches.

5 Conclusions and directions for further research

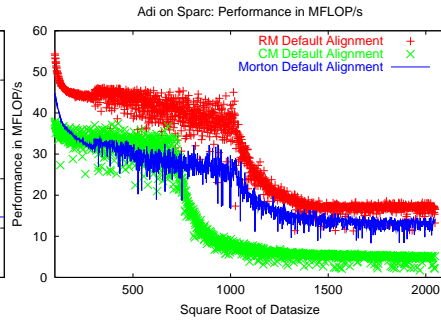
Using a small suite of dense kernels working on two-dimensional arrays, we have carried out an extensive study of row-major, column-major and Morton layouts and the impact of poor array layout on performance, covering non-power-of-two problem sizes within a substantial range. On some machines, we found that Morton array layout, even implemented with a lookup table with no compiler support, is remarkably competitive to both row-major and column-major layouts. We also found that using a lookup-table for address calculation allows flexible selection of fine-grain non-linear array layout, while offering attractive performance on some architectures compared with lexicographic layouts on untiled loops. Although the overall performance of the basic Morton scheme, as described in this paper, is only attractive for some architectures and kernels, a small improvement in its performance could make it a promising alternative to lexicographic layouts. A number of interesting issues remain:

- **Non-square cache blocks and pages.** In our brief analysis of spatial locality using Morton layout (Section 3), we assumed that cache blocks and virtual memory pages are a *square* (even) power of two. This depends on the array's element size, and is often not the case. In these cases, Morton layout can lead to differing spatial locality, when traversed in row-major or column-major order. A more subtle non-linear layout might address this.
- **Base address alignment.** In our implementation, we have only considered the default alignment, as returned by `malloc()`, of the base address of Morton arrays. For large arrays, we have found this to be page-aligned plus 8 bytes on several systems. Our studies show that alignment of the base address of Morton arrays can make a significant difference in performance (especially when the alignment is of cache line length or of page size). Further study and experiments are needed in this aspect.
- **Unrolling.** The results presented here are based on code which uses the lookup table for every address calculation. By strip-mining the innermost loop (which is always valid) by a small square power-of-two factor such as 4, it is possible to replace some lookup table accesses with constant offsets from the base of a 2×2 block. This should give higher performance for the Morton layout, at the loss of some of the addressing flexibility which the lookup table scheme allows.
- **Associativity conflicts within and between Morton arrays.** Associativity conflicts have been studied extensively for lexicographic layouts (*e.g.* [8]). Our results show evidence that associativity conflicts also impact performance with Morton layout, and further study of the effect is needed.
- **Cache contention between arrays and lookup tables.** The lookup table scheme relies for its performance on the tables, which are accessed with unit stride, occupying first-level cache. However, array accesses can displace lookup table entries. We believe this effect may explain some features of our performance graphs and plan to investigate further.

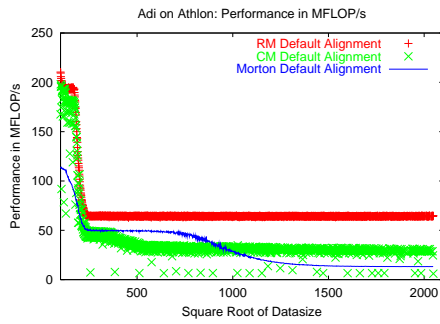
Win over CM for problem sizes larger than about 362×362



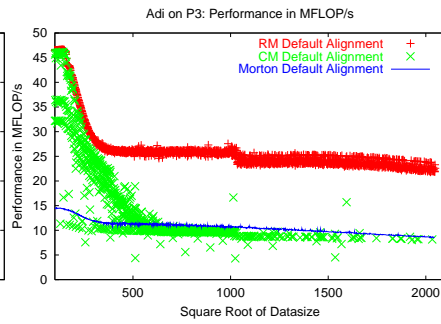
Win over CM for problem sizes larger than about 700×700



Lose



Marginal win over CM



Lose

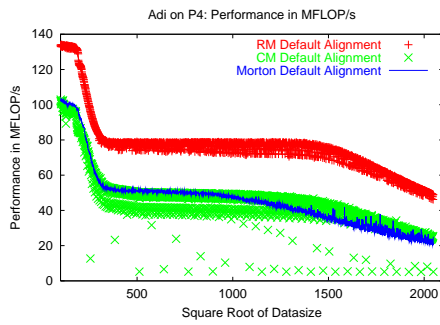


Fig. 4. ADI performance in MFLOPs on different platforms. We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables.

*Win over CM for problem sizes
larger than about 330×330*

Marginal win over CM

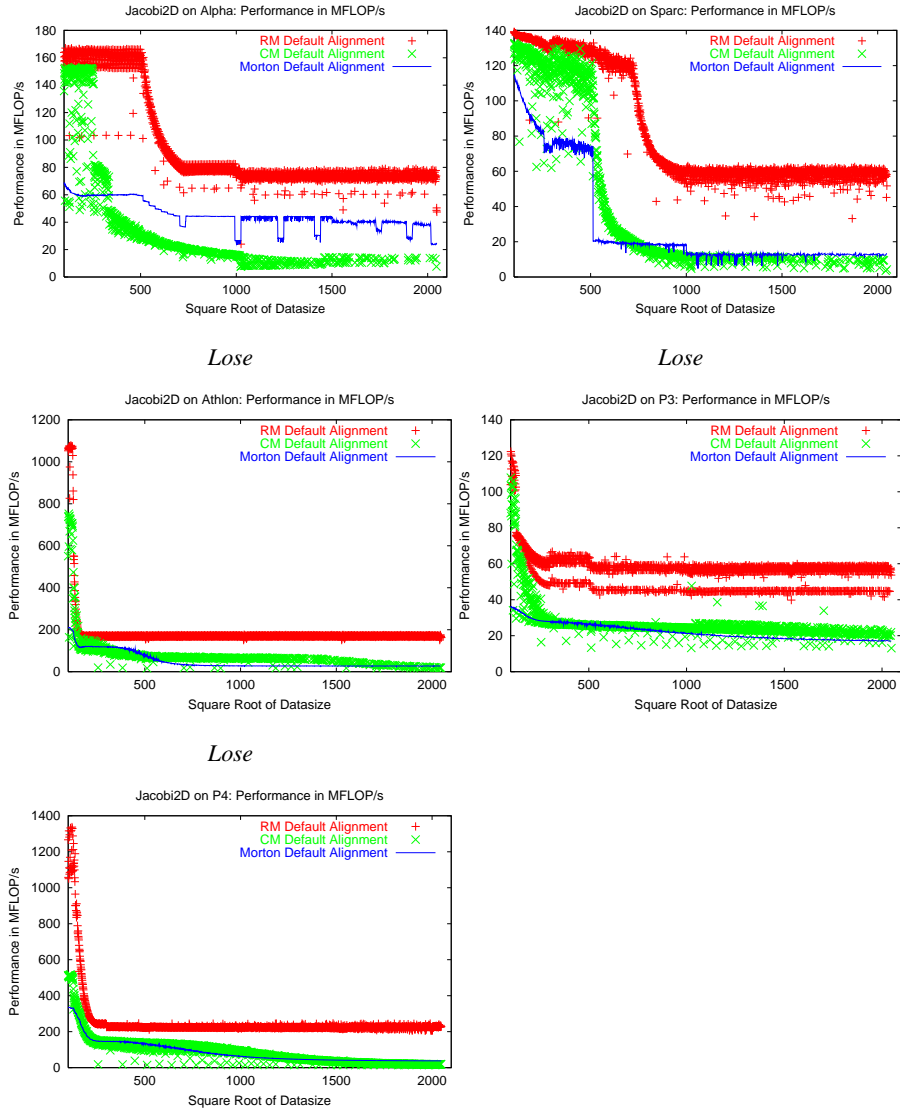
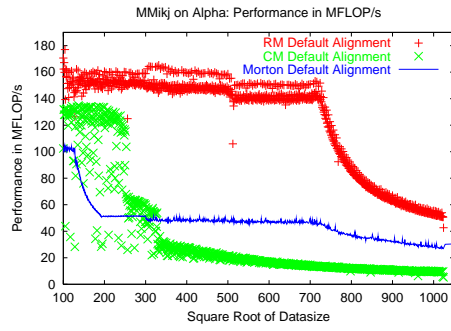
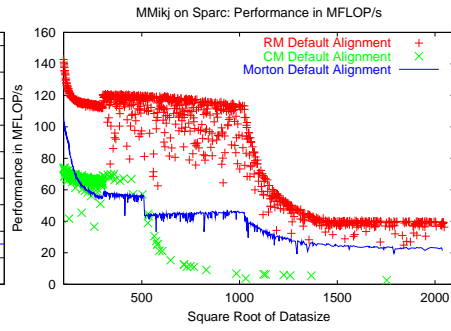


Fig. 5. Jacobi2D performance in MFLOPs on different platforms. We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables.

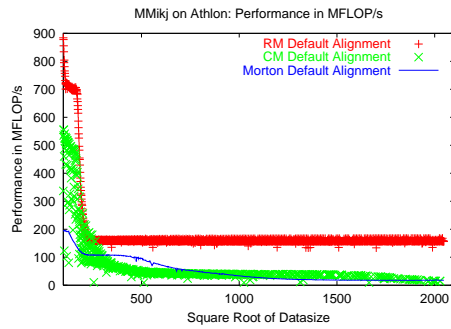
Win over CM for problem sizes larger than about 330×330



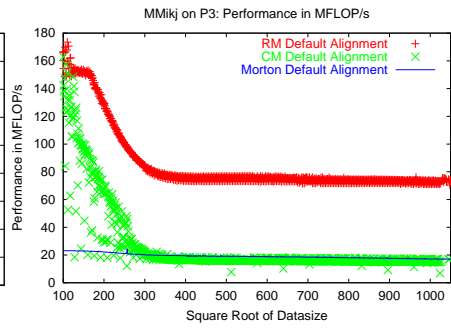
Win over CM for problem sizes larger than about 500×500



Lose



Lose



Marginal Win

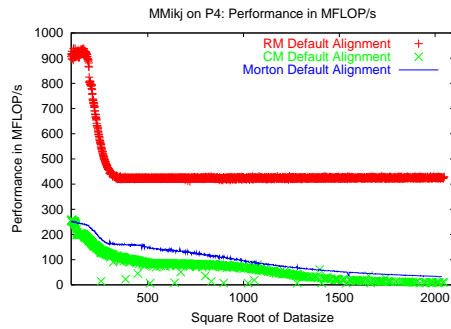


Fig. 6. MMikj performance in MFLOPs on different platforms. We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables.

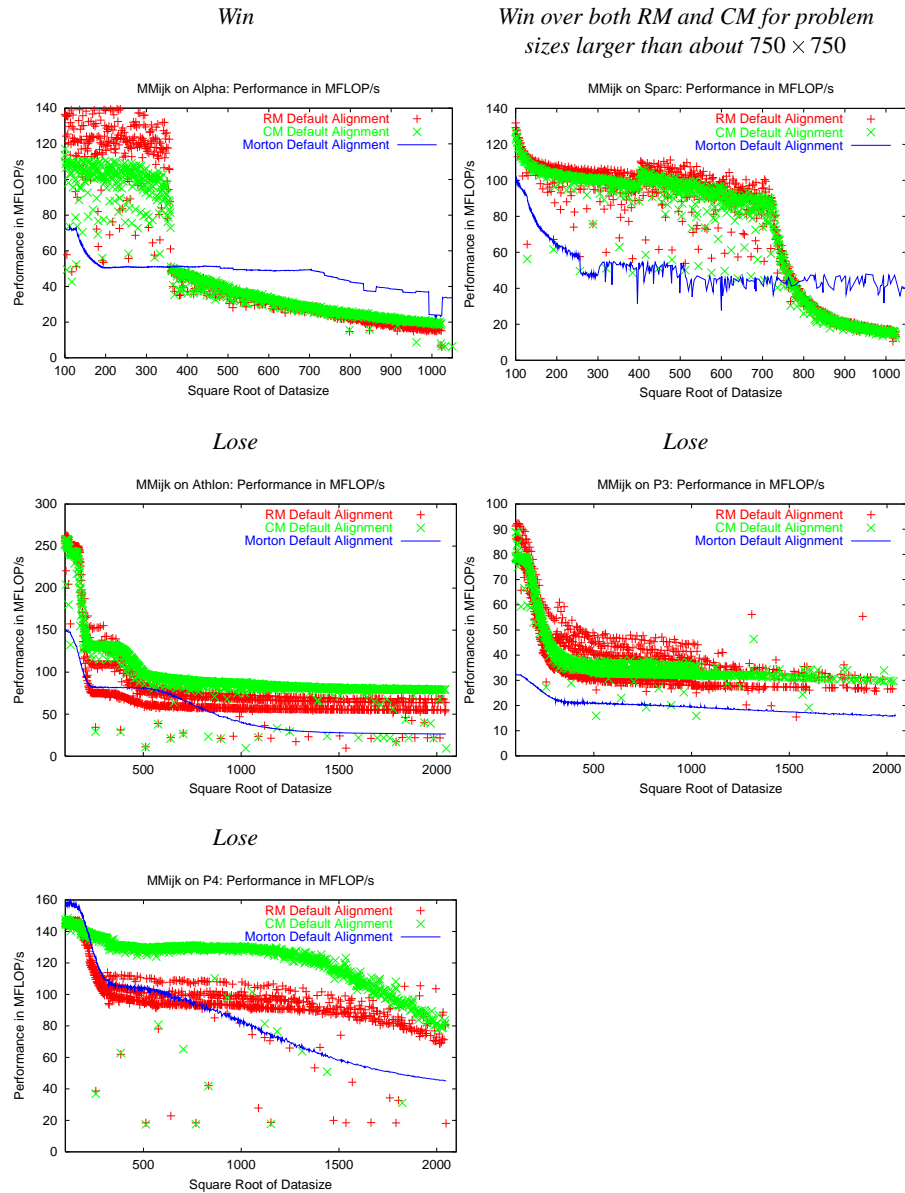


Fig. 7. MMijk performance in MFLOPs on different platforms. We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables.

- **Prefetching.** Most modern processors have both autonomous prefetching of uniform address streams, and explicit prefetching instructions. With lexicographic layout, fixed-stride accesses are common and autonomous prefetch mechanisms should work well. With Morton layout, the access pattern is known in advance but is not uniform. To sustain memory access bandwidth we need to issue prefetch instructions carefully.

It seems unlikely that Morton layout can offer a competitive compromise for three-dimensional arrays, since a given lexicographic traversal would use only 2^k words of each 2^{3k} -word cache block.

Acknowledgements. This work was partly supported by mi2g Software, a Universities UK Overseas Research Scholarship and by the United Kingdom EPSRC-funded OSCAR project (GR/R21486). We also thank Imperial College Parallel Computing Centre (ICPC) for access to their equipment. We are very grateful for helpful discussions with Susanna Pelagatti and Scott Baden, whose visits were also funded by the EPSRC (GR/N63154 and GR/N35571).

References

1. Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, 1994.
2. Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing*, pages 444–453, 1999.
3. P. Drakenberg, F. Lundevall, and B. Lisper. An Efficient Semi-Hierarchical Array Layout. *Proc. Workshop on Interaction between Compilers and Computer Architectures (Kluwer)*, 2001.
4. Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, N. Shenoy, and Prithviraj Banerjee. Enhancing spatial locality via data layout optimizations. In *European Conference on Parallel Processing*, pages 422–434, 1998.
5. Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGPLAN Notices*, 26(4):63–74, 1991.
6. Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
7. <http://www.doc.ic.ac.uk/~jeyan/MortonResults.html>.
8. Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49, 1998.
9. <http://www.specbench.org/>.
10. J. Thiyaalingam and P. H. J. Kelly. Is Morton Layout Competitive for Large Two-Dimensional Arrays? *Lecture Notes in Computer Science*, 2400:280–288, 2002.
11. David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for Morton-order matrices. In *Proc. 2001 ACM Symp. on Principles and Practice of Parallel Programming, SIGPLAN Not. 36, 7*, July 2001.
12. Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.