Improving the Performance of Morton Layout by Array Alignment and Loop Unrolling

Reducing the Price of Naivety

Jeyarajan Thiyagalingam, Olav Beckmann, and Paul H. J. Kelly

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

Abstract. Hierarchically-blocked non-linear storage layouts, such as the Morton ordering, have been proposed as a compromise between row-major and column-major for two-dimensional arrays. Morton layout offers some spatial locality whether traversed row-wise or column-wise. The goal of this paper is to make this an attractive compromise, offering close to the performance of row-major traversal of row-major layout, while avoiding the pathological behaviour of column-major traversal. We explore how spatial locality of Morton layout depends on the alignment of the array's base address, and how unrolling has to be aligned to reduce address calculation overhead. We conclude with extensive experimental results using five common processors and a small suite of benchmark kernels.

1 Introduction

Programming languages that offer support for multi-dimensional arrays generally use one of two linear mappings to translate from multi-dimensional array indices to locations in the machine's linear address space: row-major or column-major. Traversing an array in the same order as it is laid out in memory leads to excellent spatial locality; however, traversing a row-major array in column-major order or vice-versa, can lead to an order-of-magnitude worse performance. Morton order is a hierarchical, non-linear mapping from array indices to memory locations which has been proposed by several authors as a possible means of overcoming some of the performance problems associated with lexicographic layouts [2, 4, 9, 11]. The key advantages of Morton layout are that the spatial locality of memory references when iterating over a Morton order array is not biased towards either the row-major or the column major traversal order and that the resulting performance tends to be much smoother across problem-sizes than with lexicographic arrays [2]. Storage layout transformations, such as using Morton layout, are always valid. These techniques complement other methods for improving locality of reference in scientific codes, such as tiling, which rely on accurate dependence and aliasing information to determine their validity for a particular loop nest.

Previous Work. In our investigation of Morton layout, we have thus far confined our attention to non-tiled codes. We have carried out an exhaustive investigation of the effect of poor memory layout and the feasibility of using Morton layout as a compromise between row-major and column-major [7]. Our main conclusions thus far were

- *It is crucial to consider a full range of problem sizes.*
  The fact that lexicographic layouts can suffer from severe interference problems for certain problem sizes means that it is important to consider a full range of randomly generated problem sizes when evaluating the effectiveness of Morton layout [7].
- *Morton address calculation: table lookup is a simple and effective solution.*
  Production compilers currently do not support non-linear address calculations for multi-dimensional arrays. Wise *et al.* [11] investigate the effectiveness of the "dilated arithmetic" approach for performing the address calculation. We have found that a simple table lookup scheme works remarkably well [7].
- *Effectiveness of Morton layout.*
  We found that Morton layout can be an attractive compromise on machines with large L2 caches, but the overall performance has thus far still been disappointing. However, we also observed that only a relatively small improvement in the performance of codes using Morton layout would be sufficient to make Morton storage layout an attractive compromise between row-major and column-major.

Contributions of this Paper. We make two contributions which can improve the effectiveness of the basic Morton scheme and which are both always valid transformations.

- *Aligning the Base Address of Morton Arrays (Section 2).*
  A feature of lexicographic layouts is that the exact size of an array can influence the pattern of cache interference misses, resulting in severe performance degradation for some datasizes. This can be overcome by carefully padding the size of lexicographic arrays. In this paper, we show that for Morton layout arrays, the alignment of the base address of the array can have a significant impact on spatial locality when traversing the array. We show that aligning the base address of Morton arrays to page boundaries can result in significant performance improvements.
- *Unrolling Loops over Morton Arrays (Section 3).*
  Most compilers unroll regular loops over lexicographic arrays. Unfortunately, current compilers cannot unroll loops over Morton arrays effectively due to the nature of address calculations: unlike with lexicographic layouts, there is no general straight-forward (linear) way of expressing the relationship between array locations `A[i][j]` and `A[i][j+1]` which a compiler could exploit. We show that, provided loops are unrolled in a particular way, it is possible to express these relationships by simple integer increments, and we demonstrate that using this technique can significantly improve the performance of Morton layout.

1.1 Background: Morton Storage Layout

Lexicographic array storage. For an $M \times N$ two-dimensional array A, a mapping $\mathcal{S}(i,j)$ 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

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

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").

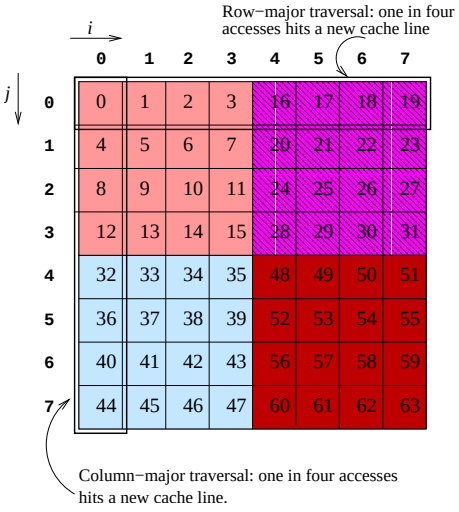


Fig. 1. Blocked row-major ("4D") layout with block-size $P = Q = 4$. The diagram illustrates that with 16-word cache lines, illustrated by different shadings, the cache hit rate is 75% whether the array is traversed in row-major or column-major order.

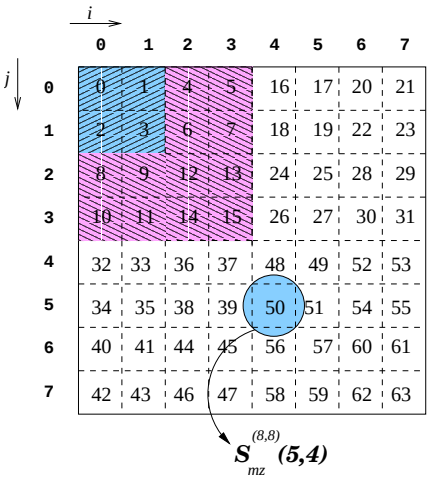


Fig. 2. Morton storage layout for an $\mathbf{8} \times 8$ array. Location of element $A[5,4]$ is calculated by interleaving "dilated" representations of 5 and 4 bitwise: $\mathcal{D}_0(5) = 100010_2$, $\mathcal{D}_1(4) = 010000_2$. $\mathcal{S}_{mz}(5,4) = \mathcal{D}_0(5) \mid \mathcal{D}_1(4) = 110010_2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

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:

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

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 rate results with column-major traversal. Most systems 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 blocksizes are not whole multiples of the next smaller blocksize.

Bit-interleaving and Morton layout. Assume for the time being that, for an $M \times N$ array, $M = 2^m$, $N = 2^n$. Write the array indices i and j as

$$\mathcal{B}(i) = i_{m-1} i_{m-2} \ldots i_1 i_0 \qquad \text{and} \qquad \mathcal{B}(j) = j_{n-1} j_{n-2} \ldots j_1 j_0$$

	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.

respectively. From this point, we restrict our analysis to square arrays (where $M = N$). Now the lexicographic mappings can be expressed as bit-concatenation (written "$\|$"):

$$\mathcal{S}_{rm}^{(M,N)}(i,j) = N \times i + j = \mathcal{B}(i)\|\mathcal{B}(j) = i_{n-1}i_{n-2}\ldots i_1 i_0 j_{n-1}j_{n-2}\ldots j_1 j_0$$
$$\mathcal{S}_{cm}^{(M,N)}(i,j) = i + M \times j = \mathcal{B}(j)\|\mathcal{B}(i) = j_{n-1}j_{n-2}\ldots j_1 j_0 i_{n-1}i_{n-2}\ldots i_1 i_0$$

If $P = 2^p$ and $Q = 2^q$, the blocked row-major mapping is

$$\begin{aligned}\mathcal{S}_{brm}^{(M,N)}(i,j) &= (P \times Q) \times \mathcal{S}_{cm}^{(M/P,N/Q)}(i,j) + \mathcal{S}_{rm}^{(P,Q)}(i\%P, j\%Q) \\ &= \mathcal{B}(i)_{(n-1)\ldots p}\|\mathcal{B}(j)_{(m-1)\ldots q}\|\mathcal{B}(i)_{(p-1)\ldots 0}\|\mathcal{B}(j)_{(q-1)\ldots 0}\end{aligned}$$

Now, choose $P = Q = 2$, and apply blocking recursively:

$$\mathcal{S}_{mz}^{(N,M)}(i,j) = i_{n-1}j_{n-1}i_{n-2}j_{n-2}\ldots i_1 j_1 i_0 j_0$$

This mapping is called the Morton Z-order, and is illustrated in Figure 2.

Morton layout can be an unbiased compromise between row-major and column-major. The key property which motivates our study of Morton layout is the following: Given a cache with any even power-of-two block size, with an array mapped according to the Morton order mapping $\mathcal{S}_{mz}$, 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 $1-(1/2^k)$.

Examples. For cache blocks of 32 bytes (4 double words, $k = 1$) this gives a hit rate of 50%. For cache blocks of 128 bytes ($k = 2$) the hit rate is 75% as illustrated earlier. For 8kB pages, 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. In Section 2, we show that this desirable property of Morton layout is conditional on choosing a suitable alignment for the base address of the array.

Morton-order address calculation using dilated arithmetic or table lookup. Bit-interleaving is too complex to execute at every loop iteration. Wise *et al.* [11] explore an

intriguing alternative: represent each loop control variable i as a "dilated" integer, where the i's bits are interleaved with zeroes. Define $\mathcal{D}_0$ and $\mathcal{D}_1$ such that

$$\mathcal{B}(\mathcal{D}_0(i)) = 0i_{n-1}0i_{n-2}\ldots 0i_1 0i_0 \qquad \text{and} \qquad \mathcal{B}(\mathcal{D}_1(i)) = i_{n-1}0i_{n-2}0\ldots i_1 0i_0 0$$

Now we can express the Morton address mapping as $\mathcal{S}_{mz}^{(N,M)}(i,j) = \mathcal{D}_1(i) \mid \mathcal{D}_0(j)$, where "|" denotes bitwise-or. At each loop iteration we increment the loop control variable; this is fairly straightforward. Let "&" denote bitwise-and. Then:

$$\begin{aligned}
\mathcal{D}_0(i+1) &= ((\mathcal{D}_0(i) \mid \text{Ones}_0) + 1) \ \& \ \text{Ones}_1 \\
\mathcal{D}_1(i+1) &= ((\mathcal{D}_1(i) \mid \text{Ones}_1) + 1) \ \& \ \text{Ones}_0 \qquad \text{where} \\
\mathcal{B}(\text{Ones}_0) &= 10101\ldots01010 \quad \text{and} \quad \mathcal{B}(\text{Ones}_1) = 01010\ldots10101 \quad .
\end{aligned}$$

This approach works when the array is accessed using an induction variable which can be incremented using dilated addition. We found that a simpler scheme often works nearly as well: we simply pre-compute a table for the two mappings $\mathcal{D}_0(i)$ and $\mathcal{D}_1(i)$. Table accesses are likely cache hits, as their range is small and they have unit stride.

2 Alignment of the Base Address of Morton Arrays

With lexicographic layout, it is often important to pad the row or column length of an array to avoid associativity conflicts [5]. With Morton layout, it turns out to be important to pad the base address of the array. In our discussion of the cache hit rate resulting from Morton order arrays in the previous Section, we have implicitly assumed that the base address of the array will be mapped to the start of a cache line. For a 32 byte, i.e. 2×2 double word cache line, this would mean that the base address of the Morton array is 32-byte aligned. As we have illustrated previously in Section 1.1, such an allocation is unbiased towards any particular order of traversal. However, in Figure 3 we show that if the allocated array is offset from this "perfect" alignment, Morton layout may no longer be an unbiased compromise storage layout: The average miss-rate of traversing the array, both in row- and in column-major order, is always worse when the alignment of the base address is offset from the alignment of a 4-word cache line. Further, when the array is mis-aligned, we lose the symmetry property of Morton order being an unbiased compromise between row- and column-major storage layout.

Systematic study across different levels of memory hierarchy. In order to investigate this effect further, we systematically calculated the resulting miss-rates for both row- and column-major traversal of Morton arrays, over a range of possible levels of memory hierarchy, and for each level, different miss-alignments of the base address of Morton arrays. The range of block sizes in memory hierarchy we covered was from 2^2 double words, corresponding to a 32-byte cache line to 2^{10} double words, corresponding to an 8kB page. Architectural considerations imply that block sizes in the memory hierarchy such as cache lines or pages have a power-of-two size. For each 2^n block size, we calculated, over all possible alignments of the base address of a Morton array with respect to this block size, respectively the best, worst and average resulting miss-rates for both

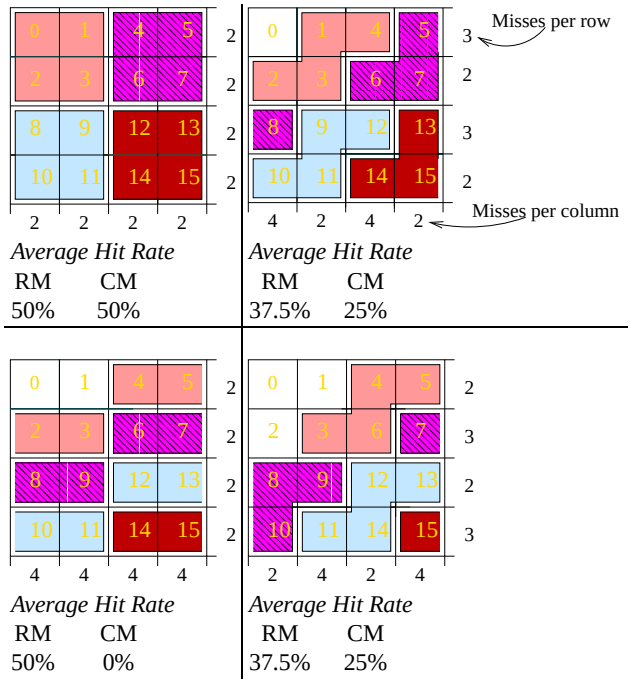


Fig. 3. Alignment of Morton-order Arrays. This figure shows the impact of mis-aligning the base address of a 4×4 Morton array from the alignment of a 4-word cache line. The numbers next to each row and below each column indicate the number of misses encountered when traversing a row (column) of the array in row-major (column-major) order, considering only spatial locality. Underneath each diagram, we show the average theoretical hit rate for the entire Morton array for both row-major (RM) and column-major (CM) traversal.

row-major and column-major traversal of the array. The standard C library `malloc()` function returns addresses which are double-word aligned. We therefore conducted our study at the resolution of double words. The results of our calculation are summarised in Figure 4. Based on those results, we offer the following conclusions.

1. The *average* miss-rate is the performance that might be expected when no special steps are taken to align the base address of a Morton array[1]. We note that the miss rates resulting from such alignments is *always suboptimal*.
2. The best average hit rates for both row- and column-major traversal are always achieved by aligning the base address of Morton array to the largest significant block size of memory hierarchy (e.g. page size).
3. The difference between the best and the worst miss-rates can be very significant, up to a factor of 2 for both row-major and column-major traversal.
4. We observe that the symmetry property which we mentioned in Section 1.1 is in fact *only* available when using the best alignment and for even power-of-two block

[1] In reality, some operating systems do not return randomly aligned addresses (modulo double word size). Linux, for example, seems to consistently return "page plus 8 bytes" addresses for large arrays. However, these are consistently sub-optimal for Morton arrays.

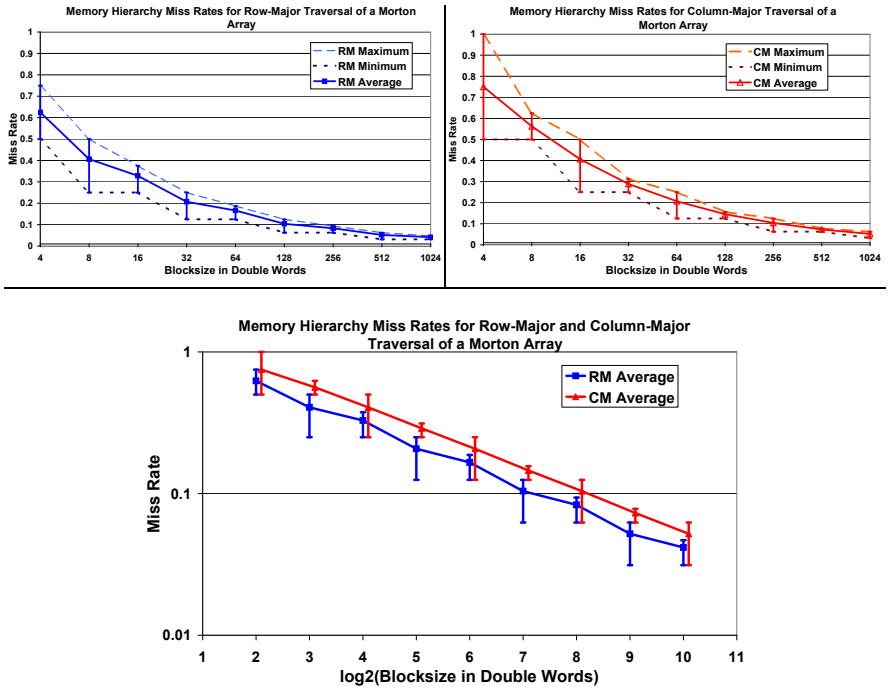


Fig. 4. Miss-rates for row-major and column-major traversal of Morton arrays. We show the best, worst and average miss-rates for different units of memory hierarchy (referred to as blocksizes), across all possible alignments of the base address of the Morton array. The top two graphs, use a linear y-axis, whilst the graph underneath uses a logarithmic y-axis to illustrate that the pattern of miss-rates is in fact highly structured across all levels of the memory hierarchy.

sizes in the memory hierarchy. For odd power-of-two block sizes (such as $2^3 = 8$ double words, corresponding to a 64-byte cache line), we find that the Z-Morton layout, which we use, is still significantly biased towards row-major traversal. An alternative recursive layout [3] may have better properties in this respect.

5. The absolute miss-rates we observe drop exponentially through increasing levels of the memory hierarchy (see the graphs in Figure 4). However, if we assume that not only the block size but also the access time of different levels of memory hierarchy increase exponentially [1], the penalty of miss-alignment of Morton arrays does not degrade significantly for larger block sizes. From a theoretical point of view, we therefore recommend aligning the base address of all Morton arrays to the largest significant block size in the memory hierarchy, i.e. page size.

In real machines, there are conflicting performance issues apart from maximising spatial locality, such as aliasing of addresses that are identical modulo some power-of-two, and some of these could negate the benefits of increased spatial locality resulting from making the base address of Morton arrays page-aligned.

Experimental Evaluation of Varying the Alignment of the Base Address of Morton Arrays. In our experimental evaluation, we have studied the impact on actual performance

of the alignment of the base address of Morton arrays. For each architecture and each benchmark, we have measured the performance of Morton layout both when using the system's default alignment (i.e. addresses as returned by `malloc()`) and when aligning arrays to each significant size of memory hierarchy. Our experimental methodology is described in Section 3.1. Detailed performance figures showing the impact of varying the alignment of the base address of Morton arrays over all significant levels of memory hierarchy are contained in an accompanying technical report [8]. Our theoretical assertion that aligning with the largest significant block size in the memory hierarchy, i.e. page size, should always be best is supported in most, but not all cases, and we assume that where this is not the case[2], this is due to interference effects. Figures 5–8 of this paper include performance results for Morton storage layout with default- and page-alignment of the array's base address.

3 Unrolling Loops over Morton Arrays

Linear array layouts have the following property. Let $\mathcal{L}(\binom{i}{j})$ be the address calculation function which returns the offset from the array base address at which the element identified by index vector $\binom{i}{j}$ is stored. Then, for any offset-vector $\binom{k}{l}$, we have

$$\mathcal{L}\left(\binom{i}{j}+\binom{k}{l}\right) = \mathcal{L}\binom{i}{j} + \mathcal{L}\binom{k}{l} \quad . \tag{1}$$

As an example, for a row-major array A, $A(i, j+k)$ is stored at location $A(i, j)+k$. Compilers can exploit this transformation when unrolling loops over arrays with linear array layouts by strength-reducing the address calculation for all except the first loop iteration in the unrolled loop body to simple addition of a constant.

As stated in Section 1.1, the Morton address mapping is $\mathcal{S}_{mz}(i, j) = \mathcal{D}_1(i) \mid \mathcal{D}_0(j)$, where "|" denotes bitwise-or, which can be implemented as addition. Given offset k,

$$\mathcal{S}_{mz}(i, j+k) = \mathcal{D}_1(i) \mid \mathcal{D}_0(j+k) = \mathcal{D}_1(i) + \mathcal{D}_0(j+k) \quad .$$

The problem is that there is no general way of simplifying $\mathcal{D}_0(j+k)$ for all j and all k.

Proposition 1 (Strength-reduction of Morton address calculation). *Let u be some power-of-two number such that $u = 2^n$. Assume that $j \bmod u = 0$ and that $k < u$. Then,*

$$\mathcal{D}_0(j+k) = \mathcal{D}_0(j) + \mathcal{D}_0(k) \quad . \tag{2}$$

This follows from the following observations: If $j \bmod u = 0$ then the n least significant bits of j are zero; if $k < u$ then all except the n least significant bits of k are zero. Therefore, the dilated addition $\mathcal{D}_0(j+k)$ can be performed separately on the n least significant bits of j.

As an example, assume that $j \bmod 4 = 0$. Then, the following strength-reductions of Morton order address calculation are valid:

$$\begin{aligned}
\mathcal{S}_{mz}(i, j+1) &= \mathcal{D}_1(i) + \mathcal{D}_0(j) + 1 \\
\mathcal{S}_{mz}(i, j+2) &= \mathcal{D}_1(i) + \mathcal{D}_0(j) + 4 \\
\mathcal{S}_{mz}(i, j+3) &= \mathcal{D}_1(i) + \mathcal{D}_0(j) + 5 \quad .
\end{aligned}$$

[2] For MMijk on Alpha, L2-aligned is better than page-aligned.

	Adi		Cholk		Jacobi2D		MMijk		MMikj	
	Alternating-direction implicit kernel, ij-ij order		Cholesky k-variant		Two-dimensional four-point stencil smoother		Matrix multiply, ijk loop nest order		Matrix multiply, ikj loop nest order	
	min	*max*	*min*	*max*	*min*	*max*	*min*	*max*	*min*	*max*
Alpha	27.0	84.5	6.8	41.1	24.2	167.1	6.0	139.5	37.6	177.0
Athlon	43.8	210.4	8.8	308.5	150.6	1078.6	9.5	262.5	117.4	884.2
P3	13.7	46.6	3.9	42.1	38.7	122.3	15.5	91.8	43.9	153.8
P4	46.2	134.1	4.8	266.1	159.6	1337.3	12.6	147.3	281.4	939.1
Sparc	11.4	54.3	3.5	78.4	33.2	138.6	5.0	131.9	20.5	142.8

Table 2. Numerical kernels used in our evaluation, together with their baseline performance on the different platforms used. 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.

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 -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 -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 -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 [6].

An analogous result holds for the *i* index. Therefore, by carefully choosing the *alignment* of the starting loop iteration variable with respect to the array indices used in the loop body and by choosing a power-of-two unrolling factor, loops over Morton order arrays can benefit from strength-reduction in unrolled loops. In our implementation, this means that memory references for the Morton tables are replaced by simple addition of constants. Existing production compilers cannot find this transformation automatically. We therefore implemented this unrolling scheme by hand in order to quantify the possible benefit. We report very promising initial performance results in Section 3.1.

3.1 Experimental Evaluation

Benchmark kernels and architectures. To test our hypothesis that Morton layout is a useful compromise between row-major and column-major layout experimentally, we have collected a suite of simple implementations of standard numerical kernels operat-

ing on two-dimensional arrays and carried out experiments on five different architectures. The kernels used are shown in Table 2 and the platforms in Table 3.

Performance Results. Figures 5–8 show our results in detail, and we make some comments directly in the figures. We have carried out extensive measurements over a full range of problem sizes: the data underlying the graphs in Figures 5–8 consist of more than 25 million individual measurements. For each experiment / architecture pair, we give a broad characterisation of whether Morton layout is a useful compromise between row-major and column-major in this setting by annotating the figures with *win*, *lose*, etc.

Impact of Unrolling. By inspecting the assembly code, we established that at least the `icc` compiler on x86 architectures does automatically unroll our benchmark kernels for row-major layout. In Figures 5–8, we show that manually unrolling the loops over Morton arrays by a factor of four, using the technique described in Section 3, can result in a significant performance improvement of the Morton code: On several architectures, the unrolled Morton codes are for part of the spectrum of problem sizes very close to, or even better than, the performance of the best canonical code. We plan to explore this promising result further by investigating larger unrolling factors.

4 Related Work and Conclusions

Related Work. Chatterjee *et al.* [2] study Morton layout and a blocked "4D" layout. They focus on tiled implementations, for which they find that the 4D layout achieves higher performance than the Morton layout because the address calculation problem is easier, while much or all the spatial locality is still exploited. Their work has similar goals to ours, but all their benchmark applications are tiled for temporal locality; they show impressive performance, with the further advantage that performance is less sensitive to small changes in tile size and problem size, which can result in cache associativity conflicts with conventional layouts. In contrast, the goal of our work is to evaluate whether Morton layout can simplify the performance programming model presented by compilers for languages with multi-dimensional arrays. In [10] Wise *et al.* argue for compiler-support for Morton order matrices. They use the recursive implementation of the Morton layout, with the base case being manually unfolded and re-rolled, to compare against the BLAS-3 kernels. They justify that such a technique is valid as optimising compilers should provide similar support for small loops. However, they find it hard to overcome the cost of addressing without recursion.

Conclusions. We believe that work on nonlinear storage layouts, such as Morton order, is applicable in a number of different areas.

- Simplifying the performance-programming model offered to application programmers is one important objective of language design and compiler research. We believe that the work presented in this paper can reduce the price of the attractive properties offered by Morton layout over canonical layouts.
- Storage layout transformations are always valid and can be applied even in codes where tiling is not valid or hard to apply. Store layout transformation can thus be additional and complementary to iteration space transformations.

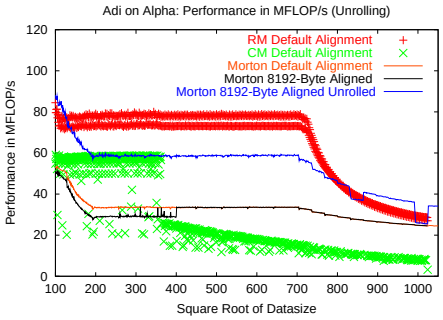
Future Work. We have reason to believe that unrolling loops over Morton arrays by factors larger than four is likely to yield greater benefits than we have measured thus far. We are also planning to investigate the performance of Morton layout in tiled codes and software-directed pre-fetching for loops over Morton arrays. We believe that the techniques we have presented in this paper facilitate an implementation of Morton layout for two-dimensional arrays that is beginning to fulfil its theoretical promise.

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 grateful to David Padua and J. Ramanujam for suggesting that we investigate unrolling during discussions at the CPC 2003 workshop in Amsterdam.

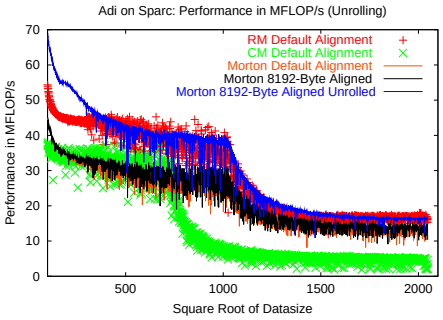
References

1. B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, Aug./Sept. 1994.
2. S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *ICS '99: Proceedings of the 1999 International Conference on Supercomputing*, pages 444–453, June 20–25, 1999.
3. S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *SPAA '99: Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, New York, June 1999.
4. P. Drakenberg, F. Lundevall, and B. Lisper. An efficient semi-hierarchical array layout. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*, Monterrey, Mexico, Jan. 2001. Kluwer. Available via `www.mrtc.mdh.se`.
5. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *PLDI '98: Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, 17–19 June 1998.
6. `www.specbench.org`.
7. J. Thiyagalingam, O. Beckmann, and P. H. J. Kelly. An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays. In S. A. Jarvis, editor, *Performance Engineering:* 19^{th} *Annual UK Performance Engineering Workshop*, pages 340–351. University of Warwick, UK, July 2003.
8. J. Thiyagalingam, O. Beckmann, and P. H. J. Kelly. Improving the performance of basic morton layout by array alignment and loop unrolling — towards a better compromise storage layout. Technical report, Department of Computing, Imperial College London, Sept. 2003. Available via `www.doc.ic.ac.uk/~jeyan/`.
9. V. Valsalam and A. Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–839, Aug. 2002.
10. D. S. Wise and J. D. Frens. Morton-order Matrices Deserve Compilers' Support. *Technical Report*, TR533, Nov. 1999.
11. D. S. Wise, J. D. Frens, Y. Gu, and G. A. Alexander. Language support for Morton-order matrices. *ACM SIGPLAN Notices*, 36(7):24–33, July 2001. Proceedings of PPoPP 2001.

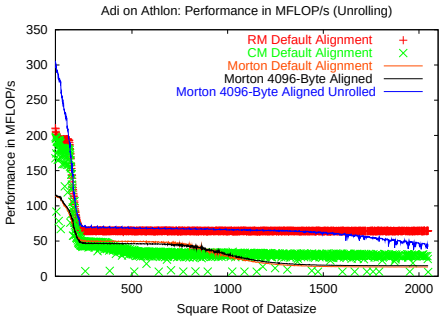
Win over CM



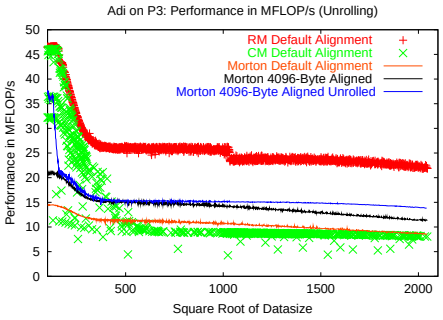
Win over CM



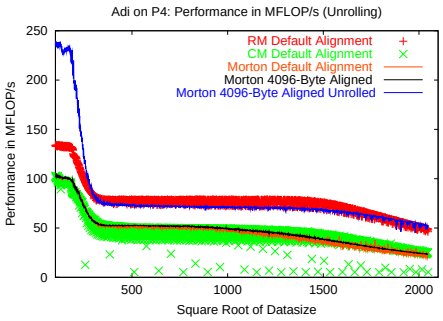
Marginal Win



Win over CM with Alignment



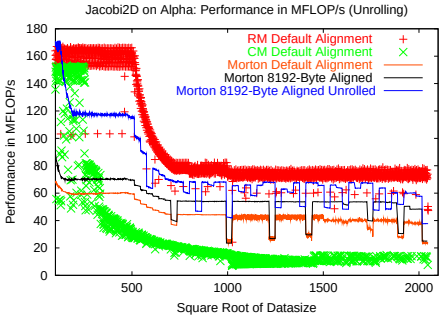
Mariginal Win



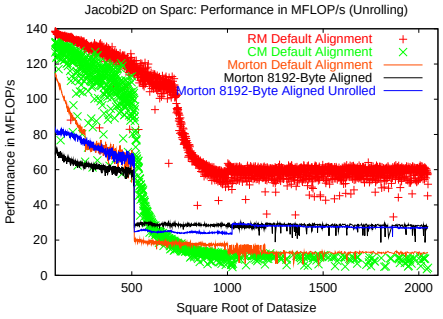
- Notice for Alpha, the upper limit is 1024×1024.
- For Alpha (Sun), the fall-off in RM performance occurs at 725×725 (1024×1024) when the total datasize exceeds L2 cache size of 4MB (8MB), direct mapped. This assumes a working set of 725×725 (1024×1024) doubles.
- Alignment significantly improves performance of the default Morton array on P3. On other platforms, alignment also yields slight improvements.

Fig. 5. ADI performance in MFLOPs on different platforms. We compare row-major, column-major, Morton with default alignment of the base address of the array, Morton with page-aligned base address and unrolled-Morton with page-aligned base address and factor 4 loop unrolling.

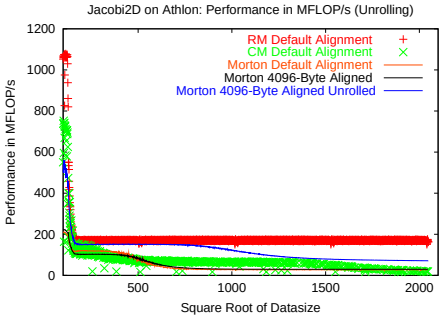
Win over CM



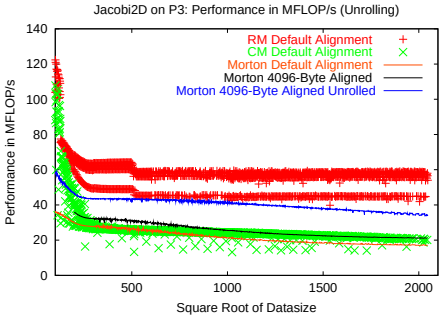
Win over CM for problem sizes larger than about 600×600



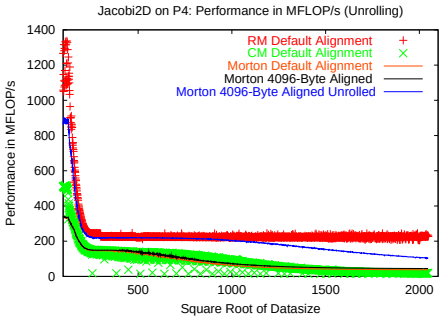
Win over CM



Win over CM



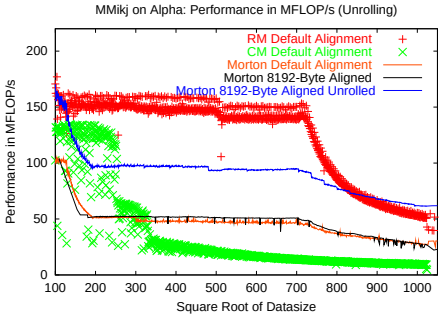
Marginal Win/Win over CM



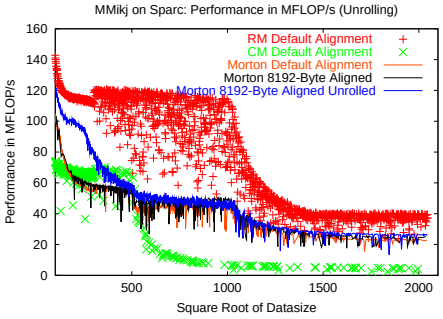
- L2-aligned version on Alpha, and page-aligned on other platforms, improves basic Morton performance.
- Unrolling improves performance of the best aligned Morton implementation, in particular on x86 where the unrolled Morton performance is within reach of the best canonical.

Fig. 6. Jacobi2D performance in MFLOPs on different platforms. We compare row-major, column-major, Morton with default alignment of the base address of the array, Morton with page-aligned base address and Morton with page-aligned base address and factor 4 loop unrolling.

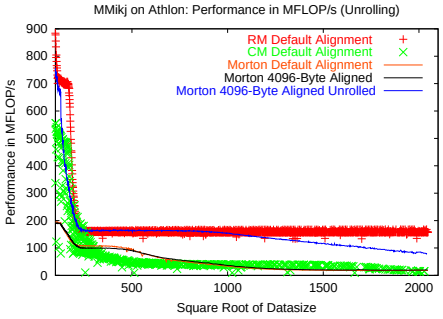
Win over CM for problem sizes larger than about 330×330



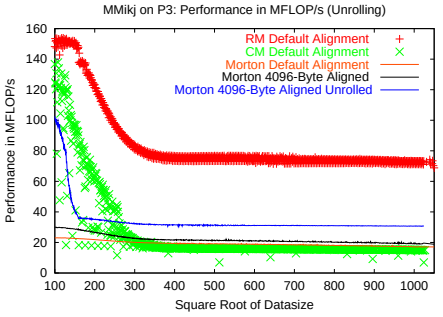
Win over CM for problem sizes larger than about 500×500



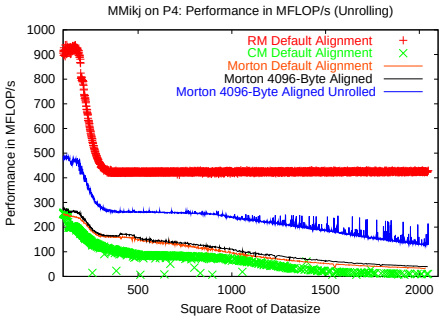
Marginal Win/Win over CM



Marginal Win/Win over CM



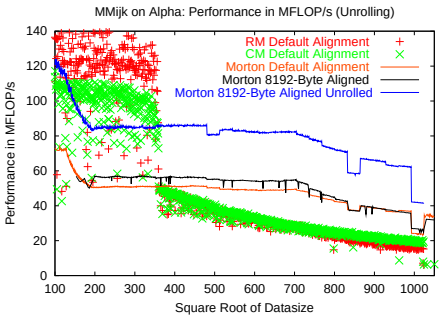
Win over CM



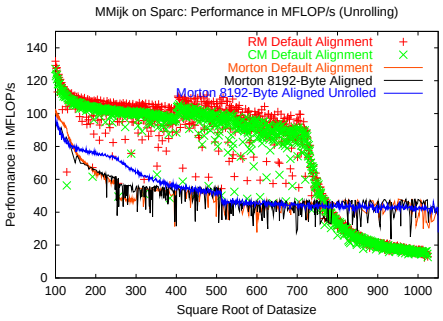
- For Alpha, notice that upper limit is 1024×1024.
- Alignment causes a small improvement in Morton performance on all platforms except Athlon.

Fig. 7. MMikj performance in MFLOPs on different platforms. We compare row-major, column-major, Morton with default alignment of the base address of the array, Morton with page-aligned base address and Morton with page-aligned base address and factor 4 loop unrolling.

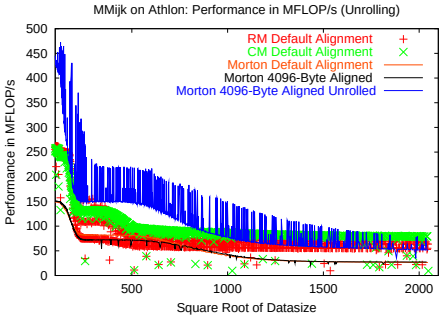
Win over both RM and CM for problem sizes larger than about 360×360



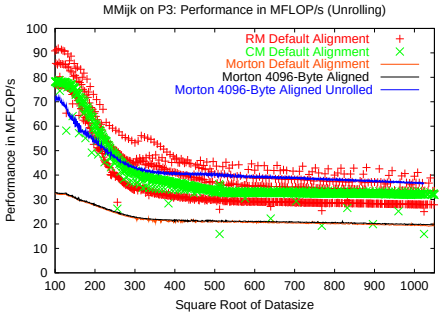
Win over both RM and CM for problem sizes larger than about 750×750



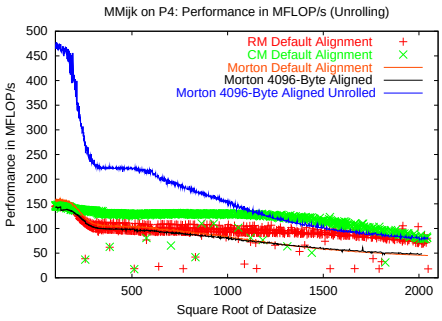
Win over both RM and CM for problem sizes smaller than about 1200×1200



Marginal Win/Win over CM



Win over both RM and CM for problem sizes smaller than about 1200×1200



- For Alpha, notice that the upper limit is 1024×1024.
- Notice the sharp drop in RM and CM performance on Alpha (around 360×360) and on Sparc (around 700×700) platforms .
- Page-aligned Morton is marginally faster than default on most platforms.

Fig. 8. MMijk performance in MFLOPs on different platforms. We compare row-major, column-major, Morton with default alignment of the base address of the array, Morton with page-aligned base address and Morton with page-aligned base address and factor 4 loop unrolling.