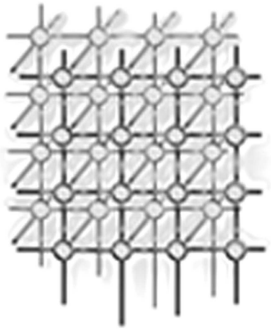
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **00**:1–6 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Is Morton layout competitive for large two-dimensional arrays, yet?



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

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

SUMMARY

Two-dimensional arrays are generally arranged in memory in row-major order or column-major order. Traversing a row-major array in column-major order, or vice-versa, leads to poor spatial locality. With large arrays the performance loss can be a factor of 10 or more. This paper explores the Morton storage layout, which has substantial spatial locality whether traversed in row-major or column-major order.

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. We show that Morton layout can lead to better performance than the worse of the two canonical layouts; however, the performance of Morton layout compared to the better choice of canonical layout is often disappointing. We further study one simple improvement of the basic Morton scheme: we show that choosing the correct alignment for the *base address* of an array in Morton layout can sometimes significantly improve the competitiveness of this layout.

KEY WORDS: *Compilers, memory organisation, tiling, space-filling curves, Morton order, spatial locality*

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 [3, 19]). 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

Received 19^{th} *April 2003*
Revised 24^{th} *March 2004*

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. It is of course possible that the compiler might help by adjusting storage layouts or by interchanging loops; however, in the examples which we studied in this paper, we have not seen evidence of the compiler performing either of these transformations. Providing a clear performance programming model is an important aspect of research into compilation; the severe performance differences that can be observed depending on the traversal order of large two-dimensional arrays represent a failure to provide such a model. Even if production compilers did eliminate some loops accessing large arrays with poor stride by suitable transformations, the behaviour of the compiler with respect to this optimisation would have to become part of the performance model presented to the application programmer. In this paper, we have carried out an extensive study of whether Morton order, when used as the default layout for large two-dimensional arrays, can deliver predictable performance, and we quantify the performance penalty that such a choice would incur when compared to an optimal choice of lexicographic layout.

1.1. Contributions of this paper

- We show that using lookup tables to calculate Morton layout addresses is remarkably effective. It compares well with the dilated arithmetic scheme proposed by Wise *et al.* [19], and offers useful flexibility (Section 4.2).
- 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 6).
- For each processor and each kernel, we calculate the slowdown, over a range of problem sizes, of using Morton layout compared with the better of the two lexicographic layouts (Section 6.2).
- We show that the effectiveness of Morton layout can often be significantly improved if the base address of the array is page-aligned (Section 5).

The paper is organised as follows: Section 2 explains the relationship with our previous paper on this topic, and puts this paper in the context of earlier research in the area. Section 3 introduces the Morton storage layout. Section 4 discusses the address calculation strategies for Morton layout. Section 5 discusses the impact of the alignment of the base address of a Morton array and Section 6 reports the detailed results of our experimental investigation into the effectiveness of Morton layout. Section 7 concludes the paper with future directions for research in this area.

2. Related work

In our earlier paper [18], 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) reports [17] (see Table III in Section 6).
- We present an extensive and systematic study using random problem sizes. 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 are possible.
- We analyse the importance of aligning base address of a Morton array to a cache line or page boundary.

Compiler techniques. Locality can be enhanced by restructuring loops to traverse the data in an appropriate order [13, 20]. Tiling can suffer disappointing performance due to associativity conflicts, which, in turn, can be avoided by copying the data accessed by the tile into contiguous memory [12]. Copying can be avoided by building the array in this layout. More generally, storage layout can be selected to match execution order [11]. While loop restructuring is limited by what the compiler can infer about the dependence structure of the loops, adjusting the storage layout is always valid. However, each array is generally traversed by more than one loop, which may impose layout constraint conflicts which can be resolved only with foreknowledge of program behaviour. Anderson, Amarasinghe and Lam [2] describe a compilation system for automatically parallelising sequential code for shared memory multiprocessors which makes use of data layout transformations in order to reduce the adverse effects of false sharing of cache lines between processors. Cierniak and Li [4] present a unified algorithm for applying both iteration space and data layout transformations, which is shown to perform better than applying the two transformations separately. For a compiler to change the storage layout of an array in order to improve locality, it has to determine the impact of that transformation through the entire program. Cierniak and Li [5] discuss compiler algorithms for deciding whether changing the layout of an array is valid. Procedure cloning [6] can be used to dis-ambiguate array accesses, making it easier to change the layout of individual arrays. Locally reshaped arrays are arrays that are accessed inside a procedure in a different layout from the calling context. O'Boyle and Knijnenburg [14] describe a static approach to calculating the effect of global data transformations such as partitioning for parallelisation on locally reshaped arrays, as well as loop transformations that can be used to undo locally adverse effects, such as poor stride, of such global data transformations.

Blocked and recursively-blocked array layout. Wise *et al.* [19] advocate Morton layout for multidimensional arrays, and present a prototype compiler that implements the dilated arithmetic address calculation scheme which we evaluate in Section 4. They found it hard to overcome the overheads of Morton address calculation, and achieve convincing results only with recursive formulations of the loop nests.

Chatterjee *et al.* [3] study Morton layout and a blocked "4D" layout (explained in Section 3.3). They focus on tiled implementations, for which they find that the 4D layout achieves higher performance than 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 (or "shackled") 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 for unsophisticated programmers, without relying on very powerful compiler technology.

3. Background

In this section, we briefly review various array mappings and their resulting spatial locality.

3.1. 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 the row-major (for example in C and Pascal) and column-major (as used by Fortran) mappings expressed by

$$\mathcal{S}_{rm}^{(M,N)}(i,j) = N \times i + j \qquad \text{and} \qquad \mathcal{S}_{cm}^{(M,N)}(i,j) = i + M \times j \tag{1}$$

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").

3.2. Opaque array storage: array descriptors

In more modern languages, such as Fortran 90 (and notable earlier designs — Algol 68 and APL), arrays are represented by a descriptor which provides run-time information on how the address calculation should be done [9]. This is needed to support multidimensional array slicing — where the array descriptor hides the actual array representation, and allows the implementor freedom to select storage layout at will.

Using a descriptor allows a single fragment of source code to operate on arrays whose layout varies from call to call — a form of "shape" polymorphism [10]. This raises performance problems since the storage layout is not known at compile-time — the stride of successive memory accesses depends on how a function is called. For optimal performance, different variants of each function need to be generated for each combination of array operand layouts. There may be many distinct combinations requiring distinct code variants. The variants can be selected by run-time dispatch. More aggressively, the appropriate procedure "clone" can be called according to call site context [6].

3.3. Blocked array storage

How can we reduce the number of code variants needed to achieve high performance? 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 subarrays, 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 [3]) as:

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

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. Notice that the same cache hit rate results

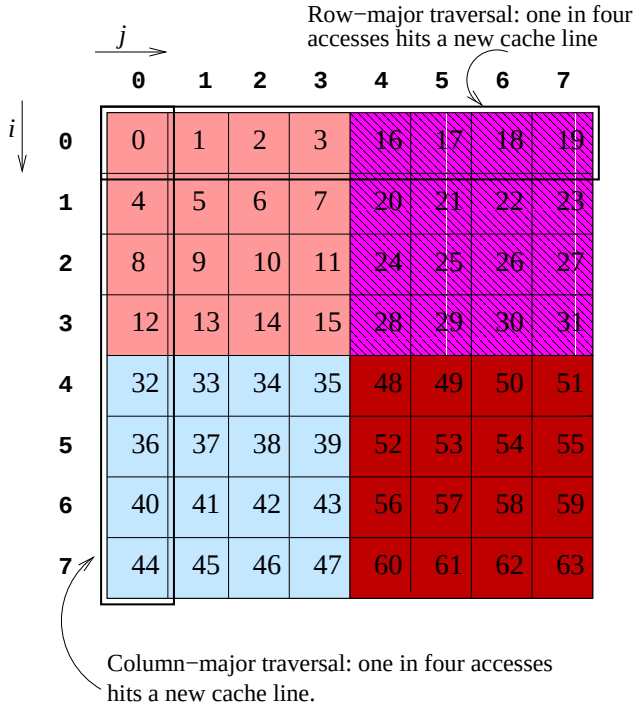


Figure 1. **Blocked row-major ("4D") layout $\mathcal{S}^{(\mathbf{8},\mathbf{8})}_{\mathbf{brm}}(\mathbf{i},\mathbf{j})$ with block-size parameters $\mathbf{P} = \mathbf{Q} = \mathbf{4}$.** The diagram illustrates that with 16-word cache lines, illustrated by different shadings, the cache hit rate is 75% whether the array is traversed in row-major or column-major order.

with column-major traversal, i.e. when the loop structure is "`do i...do j`" rather than the "`do j...do i`" loop of row-major traversal.

3.4. Recursive blocking

The impact of virtual memory pages on blocked array storage. Modern computer systems rely on a TLB to cache address translations: a typical 64-entry data TLB with 8 KB pages has an effective span of $64 \times 8 = 512$ KB. Unfortunately, as illustrated in Figure 2, 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: Each 8 KB page (1024 doubles) would hold a 16×16 array of 2×2-element subarrays.

Memory hierarchies. 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 blocksizes are not whole multiples of the next smaller blocksize.

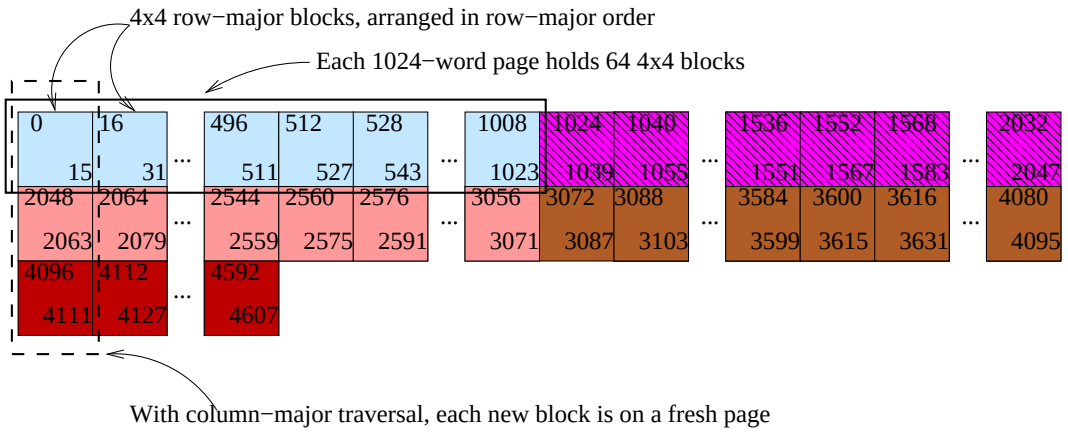


Figure 2. **Blocked row-major layout for large array.** If a large blocked row-major array is traversed in column-major order, only one subarray per page is usable. The diagram shows an array with rows of 2048 doubles, using the blocked row-major layout with 4×4 blocks. Each 8 KB page holds 1024 doubles, 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 $1 - 1/4 = 75\%$).

3.5. Bit-interleaving

Assume that for an $M \times N$ array, $M = 2^m$, $N = 2^n$. Write the array indices i and j as

$$\mathcal{B}(i) = i_{m-1}i_{m-2}\ldots i_2 i_1 i_0 \qquad \text{and} \qquad \mathcal{B}(j) = j_{n-1}j_{n-2}\ldots j_2 j_1 j_0 \tag{2}$$

respectively. From this point onwards, we restrict our analysis to square arrays (where $M = N$; we address non-square arrays in Section 4.2). Now the lexicographic mappings can be expressed as bit-concatenation (written "$\|$"):

$$\begin{aligned}\mathcal{S}_{rm}^{(M,N)}(i,j) &= N \times i + j = \mathcal{B}(i)\|\mathcal{B}(j) \\ &= i_{n-1}i_{n-2}\ldots i_2 i_1 i_0 j_{n-1}j_{n-2}\ldots j_2 j_1 j_0 \qquad (3)\\ \mathcal{S}_{cm}^{(M,N)}(i,j) &= i + M \times j = \mathcal{B}(j)\|\mathcal{B}(i) \\ &= j_{n-1}j_{n-2}\ldots j_2 j_1 j_0 i_{n-1}i_{n-2}\ldots i_2 i_1 i_0 \qquad (4)\end{aligned}$$

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

$$\begin{aligned}\mathcal{S}_{brm}^{(M,N)}(i,j) = &\quad (P \times Q) \times \mathcal{S}_{cm}^{(M/P,N/Q)}(i,j) + \mathcal{S}_{rm}^{(P,Q)}(i\%P, j\%Q) \\ = &\quad \mathcal{B}(i)_{(n-1)\ldots p}\|\mathcal{B}(j)_{(m-1)\ldots q}\|\mathcal{B}(i)_{(p-1)\ldots 0}\|\mathcal{B}(j)_{(q-1)\ldots 0} \qquad (5)\end{aligned}$$

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

$$\mathcal{S}_{mz}^{(N,M)}(i,j) = i_{n-1}j_{n-1}i_{n-2}j_{n-2}\ldots i_2 j_2 i_1 j_1 i_0 j_0 \tag{6}$$

This mapping is called the Morton Z-order, and is illustrated in Figure 3.

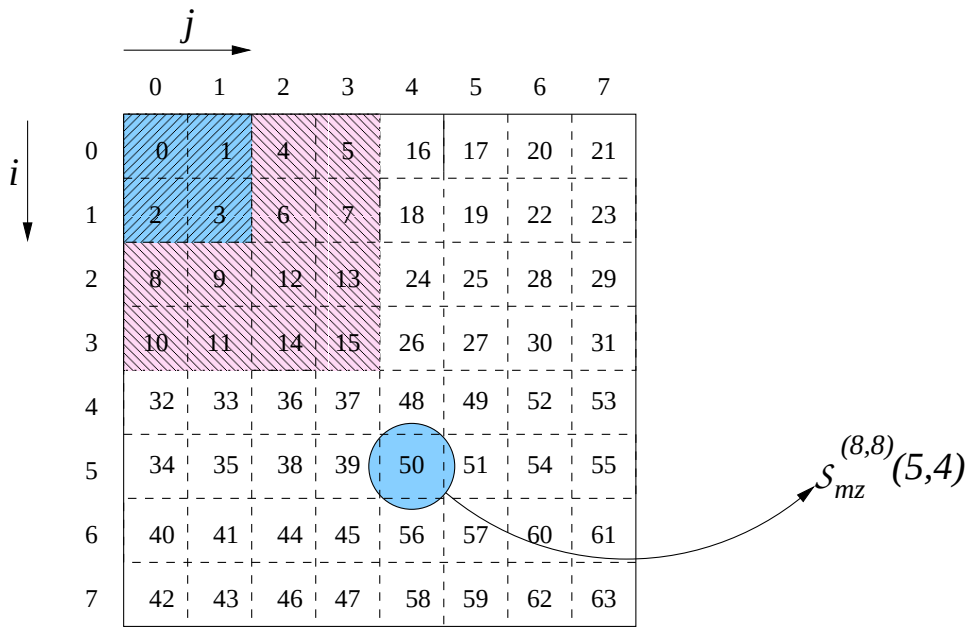


Figure 3. **Morton storage layout for an 8×8 array.** Location of element $A[5,4]$ is calculated by interleaving "dilated" representations of 5 and 4 bitwise: $\mathcal{D}_1(5) = 100010_2$, $\mathcal{D}_0(4) = 010000_2$. $\mathcal{S}_{mz}(5,4) = \mathcal{D}_1(5) \mid \mathcal{D}_0(4) = 110010_2 = 50_{10}$. A 4-word cache block holds a 2×2 subarray; a 16-word cache block holds a 4×4 subarray. Row-order traversal of the array uses 2 words of each 4-word cache block on each sweep of its inner loop, and 4 words of each 16-word block. Column-order traversal achieves the same hit rate.

3.6. 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:

- 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 3.
- 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 8 KB pages (1024 words, $k = 5$), the hit rate is 96.875%. In Table I, 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 5, we show that this desirable property of Morton layout is conditional on choosing a suitable alignment for the base address of the

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

Table I. **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.

array. In our conclusion in Section 7, we also point out that non-square cache blocks and pages lead to a more complicated picture.

4. Morton-order address calculation

4.1. Dilated arithmetic

Bit-interleaving is too complex to execute at every loop iteration. Wise *et al.* [19] 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

$$\mathcal{B}(\mathcal{D}_0(i)) = 0i_{n-1}0i_{n-2}0\ldots 0i_20i_10i_0 \qquad \text{and} \qquad \mathcal{B}(\mathcal{D}_1(i)) = i_{n-1}0i_{n-2}0\ldots i_20i_10i_00 \tag{7}$$

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:

$$\mathcal{D}_0(i+1) = ((\mathcal{D}_0(i) \mid \mathrm{Ones}_0) + 1) \ \& \ \mathrm{Ones}_1 \tag{8}$$

$$\mathcal{D}_1(i+1) = ((\mathcal{D}_1(i) \mid \mathrm{Ones}_1) + 1) \ \& \ \mathrm{Ones}_0 \tag{9}$$

where

$$\mathcal{B}(\mathrm{Ones}_0) = 10101\ldots01010$$

$$\mathcal{B}(\mathrm{Ones}_1) = 01010\ldots10101 \quad .$$

This is illustrated in Figure 4, which shows the ikj variant of matrix multiply. Note that the strength-reduction in equations 8 and 9 can only be used when an array is accessed in unit stride. Most current processors do not offer instruction-level support for dilated arithmetic; if Morton layout can be seen to be competitive in terms of spatial locality for application programs, this would be an interesting architectural feature to investigate.

4.2. Morton-order address calculation using a lookup table

The dilated arithmetic approach works when the array is accessed using an induction variable which can be incremented using dilated addition. We found that a much simpler scheme often works nearly as well: we simply pre-compute a table for the two mappings $\mathcal{D}_0(i)$ and $\mathcal{D}_1(i)$. We illustrate this for the

```
#define ONES_0 0xaaaaaaaa
#define ONES_1 0x55555555
#define INC_0(vx) (((vx + ONES_0) + 1) & ONES_1)
#define INC_1(vx) (((vx + ONES_1) + 1) & ONES_0)

void mm_ikj_da(double A[SZ*SZ], double B[SZ*SZ],
                                double C[SZ*SZ])
{
  int i_1, j_0, k_0;
  double r;
  int SZ_0 = Dilate(SZ);
  int SZ_1 = SZ_0 << 1;
  for (i_1 = 0; i_1 < SZ_1; i_1 = INC_1(i_1))
    for (k_0 = 0; k_0 < SZ_0; k_0 = INC_0(k_0)){
      unsigned int k_1 = k_0 << 1;
      r = A[i_1 + k_0];
      for (j_0 = 0; j_0 < SZ_0; j_0 = INC_0(j_0))
        C[i_1 + j_0] += r * B[k_1 + j_0];
    }
}
```

Figure 4. **Morton-order matrix-multiply implementation using dilated arithmetic for the address calculation.** Variables `i_1` and `k_0` are dilated representations of the loop control counter $\mathcal{D}_i(\mathtt{i})$ and $\mathcal{D}_0(\mathtt{k})$. Counter `j` is represented by `j_0`$= \mathcal{D}_0(\mathtt{j})$. The function `Dilate` converts a normal integer into a dilated integer.

```
void mm_ikj_tb(double A[SZ*SZ], double B[SZ*SZ],
               double C[SZ*SZ],
               unsigned int MortonTab0[],
               unsigned int MortonTab1[])
{
  int i, j, k;
  double r;
  for (i = 0; i < SZ; i++)
    for (k = 0; k < SZ; k++){
      r = A[MortonTab1[i] + MortonTab0[k]];
      for (j = 0; j < SZ; j++)
        C[MortonTab1[i] + MortonTab0[j]]
          += r * B[MortonTab1[k] + MortonTab0[j]];
    }
}
```

Figure 5. **Morton-order matrix-multiply implementation using table lookup for the address calculation.** `MortonTab0` is initialised with the values taken by function $\mathcal{D}_0$, `MortonTab1` is initialised with the values taken by function $\mathcal{D}_1$. The compiler detects that `MortonTab0[i]` and `MortonTab0[k]` are loop invariant, leaving just one table lookup in the inner loop.

ikj matrix multiply variant in Figure 5. Note that for programs with regular stride, the table accesses are very likely cache hits, as their range is small and the tables themselves are accessed in unit stride. One small but important detail: we use addition instead of logical "or". This may improve instruction selection. It also allows the same loop to work on lexicographic layout using suitable tables. If the array is non-square, $2^n \times 2^m$, $n < m$, we construct the table so that the j index is dilated only up to bit n.

Figure 6 shows the performance of these two variants on a variety of computer systems. This shows that the dilated arithmetic implementation is almost always faster; however, the difference is usually less than 20%. In the remainder of the paper, we use the table lookup scheme exclusively. We comment on this decision further in Section 7. With compiler support, many applications could benefit from the dilated arithmetic approach, leading in many cases to more positive conclusions.

5. Effect of Memory Alignment in Morton Layouts

With lexicographic layout, it is often important to pad the row (respectively column) length to avoid associativity conflicts [15]. With Morton layout, it turns out to be important to consider padding the base address of the array.

In our previous discussion of the cache hit rate resulting from Morton order arrays, 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 means that the base address of the Morton array needs to be 32-byte aligned. As we have illustrated previously in Section 3.6, such an allocation is unbiased towards any particular order of traversal. However, in Figure 7 we show that if the allocated array is offset from this "perfect" alignment, Morton layout may no longer be an unbiased compromise storage layout. Furthermore, the actual average hit rates over the entire array can be significantly worse compared with perfect alignment of the base address. In Figure 8, we consider the case where the size of a cache line does not match a square tile of array elements. This is the case, for example with 64 byte cache lines and arrays of double word floating point numbers. As shown in the figure, this means that the symmetry property of Morton order is lost. It still appears, however, that perfect alignment of the base address of the Morton array, 64-byte alignment in this case, leads to the best hit rates in *both* traversal orders. A similar effect is replicated on each level of the memory hierarchy.

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. The results, which are included in Figures 9–18 and discussed in more detail in the next section, broadly confirm our theoretical conclusion.

6. 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-

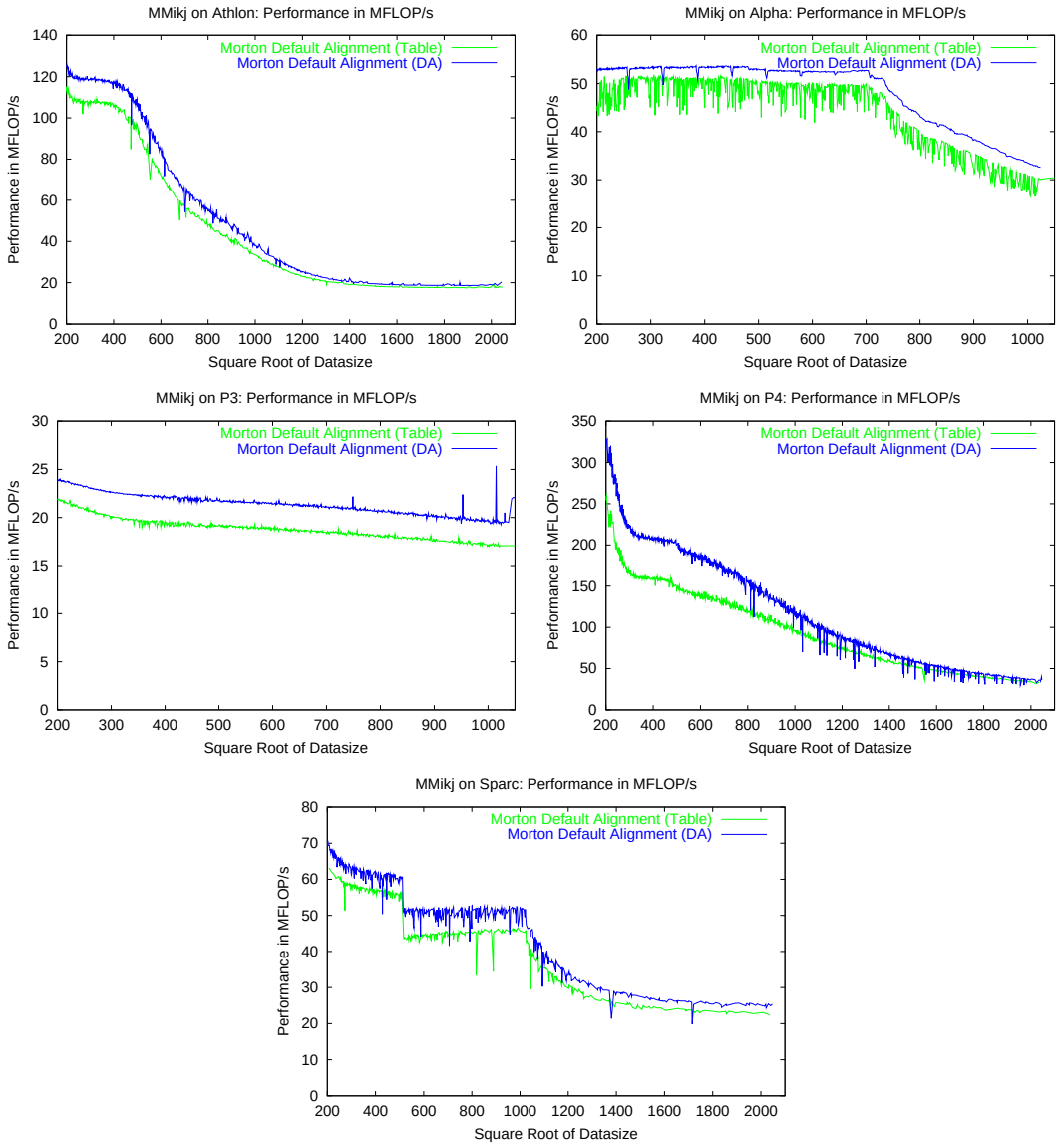


Figure 6. **Matrix multiply (ikj) performance in MFLOPs of dilated arithmetic Morton address calculation (see Figure 4) compared against the table-based Morton address calculation (see Figure 5).** The graphs show performance in MFLOPs achieved by the implementations at each problem size on each system. Details of the systems are given in Table III. On nearly all systems , the dilated-arithmetic implementation performs relatively better than the table implementation. However the difference is usually less than 20%.

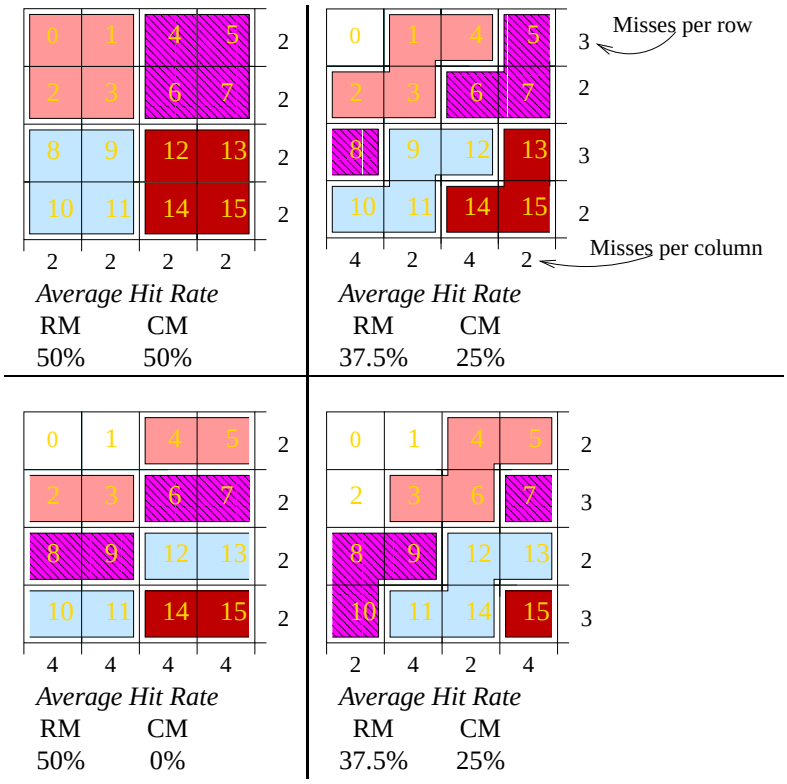


Figure 7. **Alignment of Morton order arrays I:** Figures illustrate how the cache performance can change when the alignment is varied (in the order of perfectly aligned at 32-byte boundary and offset from there by 8, 16 and 24 bytes) for a part of a larger Morton array. The numbers next to each row and below each column indicate the number of misses encountered when traversing a row (column) of the block 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 colum-major (CM) traversal. As can be seen by the illustrations, when an array is imperfectly aligned, in addition to losing the symmetry of the Morton layout, spatial locality also worsened.

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 II. **Numerical kernels used in our experimental evaluation.**

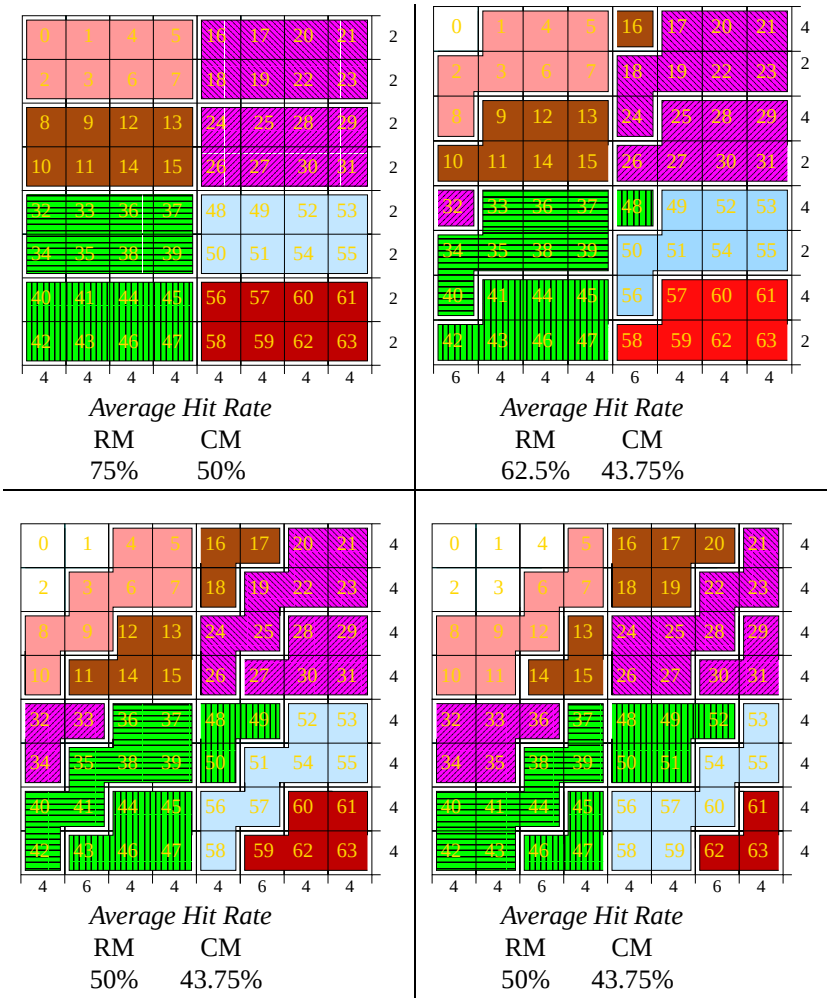


Figure 8. **Alignment of Morton order arrays II:** For a non-square cache block, such as 64 bytes or 8 double words, this figure illustrates how cache performance varies with the alignment of the base address of the array, in the order of perfectly aligned at 64 bytes, and offset by 8, 24 and 40 bytes. Although, there are 7 possible misalignments, we show only some interesting examples. Numbers next to (below) each row (column) show the number of misses encountered when traversing a row (column) of the block 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 colum-major (CM) traversal. When the array is imperfectly aligned, in addition to losing the symmetry of the Morton Layout we get worse spatial locality.

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

Table III. **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 [17].

dimensional arrays and carried out experiments on five different architectures. The benchmarking kernels used are shown in Table II and the platforms in Table III.

Problem sizes. As mentioned in Section 2, our previous paper [18] reported performance results for power-of-two problem sizes. For this paper, we decided to carry out an exhaustive study, collecting performance data, where possible, for all problem sizes between 200×200 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.

Experimental methodology. Most of the architectures we used for experiments were multi-user platforms. In the case of the x86 architectures (Pentium III, Pentium 4 and Athlon), we used clusters of identical teaching machines. The absence of a fully-controlled environment, and our desire to collect data for a full range of problem sizes (which implies running experiments for a very long time in total), led us to design carefully an experimental methodology aimed at minimising the impact of external interferences in our results.

- During off-peak hours, we ran a script for collecting measurements on each available platform. In order to minimise the impact of any transient effects on particular ranges of experiments, the scripts are programmed to repeatedly make a random choice of benchmark kernel (from the list in Table II), array layout (i.e. row-major, column-major or Morton), alignment of the base address of the array (from a list of all significant sizes in the memory hierarchy, i.e. cache line lengths and page size) and problem size.
- Once a kernel, layout, alignment and problem size are chosen, the kernel is run once and the time recorded in a shared file structure using suitable locking.
- Over time, we accumulated a total of more than 23 million measurements. In our evaluation, we proceeded as follows: For each tuple of platform, experiment (kernel), layout, alignment and problem size, we gather all timing results obtained. Notice that due to the random choice of parameters, the number of samples for each point varies. We first use the Dixon Test [7] to eliminate up to one outlier. Following that, we calculate various statistical parameters, such as mean, standard deviation, median and 90% confidence intervals.
- The performance numbers we report in the following figures and tables are all based on the *median* of measurements taken. The reason for this is that the median is less liable to interference from outliers than the mean [16]. Although we do not show these in the paper, we have calculated and plotted 90% confidence intervals for all data we report.

6.1. Performance results

Table IV shows the baseline performance achieved by each machine using standard row-major layout. Figures 9–18 show our results in detail. We make some comments on each graph directly in the figures. 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. As an overview, we record *wins* for

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

	Adi		Cholk		Jacobi2D		MMijk		MMikj	
	min	*max*	*min*	*max*	*min*	*max*	*min*	*max*	*min*	*max*
Alpha	25.1	84.5	4.8	40.6	26.8	167.1	5.8	139.5	52.7	177.0
Athlon	43.8	210.4	8.8	308.5	150.6	1078.6	10.1	655.2	117.4	884.2
P3	21.8	46.6	3.9	42.1	51.7	141.5	14.8	134.1	45.9	153.8
P4	46.1	134.1	4.8	266.1	126.6	1337.3	17.9	766.0	281.4	939.1
Sparc	14.6	54.5	3.5	78.4	33.2	139.2	2.9	131.9	32.0	145.1

Table IV. **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 9–18).

- Jacobi2D: Alpha, Sparc over column-major but not over row-major.
- MMikj: Alpha, Sparc over column-major but not over row-major.
- MMijk: Alpha, Sparc over both row-major and column-major.

Notice that for kernels with high spatial locality, such as MMikj and Jacobi2D, which run close to the machine's peak performance, bandwidth to L1 cache for table access may be a major factor. Our results also confirm that for row-major layout, padding the length of the rows of an array can significantly improve performance. The amount of padding required is small, but needs to be chosen very carefully. In Section 6.2, which follows the detailed performance graphs, we evaluate the competitiveness of Morton layout in overview by calculating the slowdown of Morton order over the best lexicographic layout.

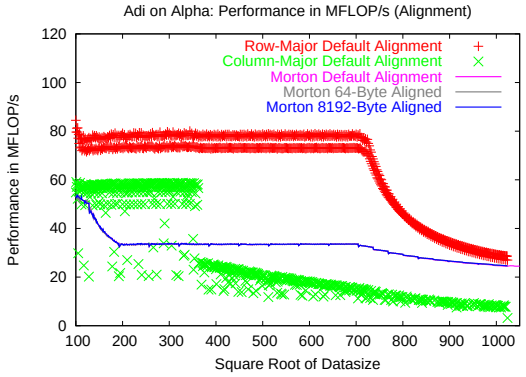
6.2. Competitive efficiency of Morton layout

If we know how an array is going to be accessed, and if there are no conflicting access patterns to that array, we could choose optimally between the two lexicographic layouts for that array. If we have only partial or no information about the access patterns, we would like to choose a layout which is competitive with the optimum layout. If we do not know how arrays in a program might be aliased, we probably need to choose *one* default layout for all arrays in that program, and we want this choice to compare well to an optimal decision.

One way to evaluate the use of the Morton layout as such a default-choice is by analogy with *competitive online algorithms*. Suppose we have an optimal array layout scheme OPT. Following [8], a memory layout scheme ALG is c-competitive (for a constant "efficiency" factor c) if there exists a constant α such that for all utilisation scenarios (or access sequences) σ,

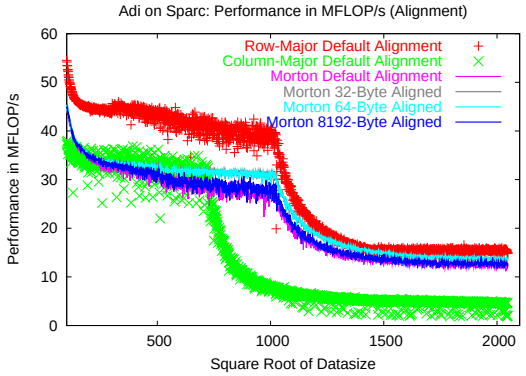
$$\mathrm{COST}_{\mathrm{ALG}}(\sigma) \leq c \cdot \mathrm{COST}_{\mathrm{OPT}}(\sigma) + \alpha$$

It is not generally possible to find an optimum layout efficiently — specifically if there are conflicting access patterns for arrays. Therefore, in order to quantify our hypothesis that Morton layout



Adi on Alpha: Win over Column-Major

- Notice upper limit is 1024×1024.
- The fall-off in RM performance occurs at 725×725 when the total datasize exceeds L2 cache size (4MB, direct mapped). This assumes a working set of 725×725 doubles.
- RM below about 725×725 has a bimodal distribution.
- Notice the sharp drop in CM performance at around 360×360.
- Alignment does not change Morton performance, the three lines coincide.



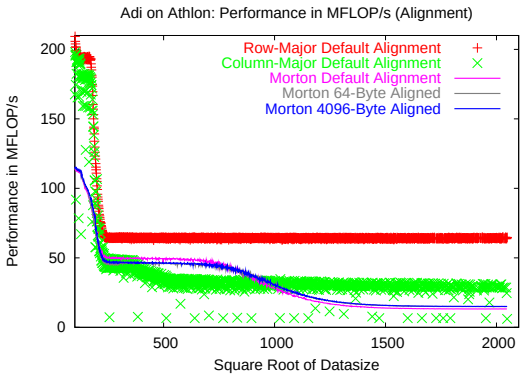
Adi on Sparc: Win over Column-Major

- The fall-off in RM performance occurs at 1024×1024 when the total datasize exceeds the L2 cache size (8MB, direct mapped). This assumes a working set of 1024×1024 doubles.
- Notice the drop in CM performance which occurs after 720×720.
- All Morton versions have high variation between problem sizes (confi dence intervals for the measurements are also larger than on other machines).

Figure 9. **ADI performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).

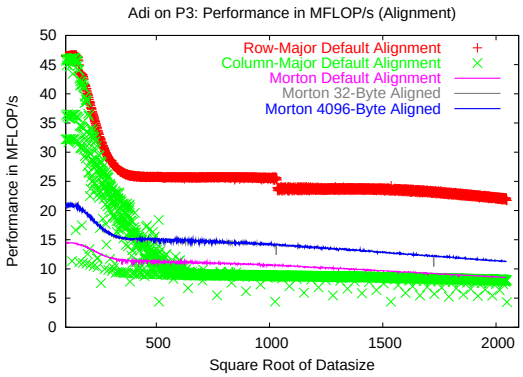
implemented using table lookup is a competitive default choice, we compare against the better of the two lexicographic layouts. The constant α is designed so that a constant overhead cost could be taken into account (such as the cost of filling lookup tables). However, in our experiments, we have simply measured the time of the numerical kernels, and we therefore set $\alpha = 0$ in the above equation.

In Figures 19–23, we show the range of slowdown factors c over the best lexicographic layout. For each problem size, we select from our performance data the better of the two lexicographic layouts and then calculate the slowdown of Morton over this "optimal" time. We show the data as box-and-whisker plots, indicating maximum minimum, median values and the range in which 50% of all data points lie. From these plots, we offer the following tentative conclusions about the competitiveness of Morton layout.



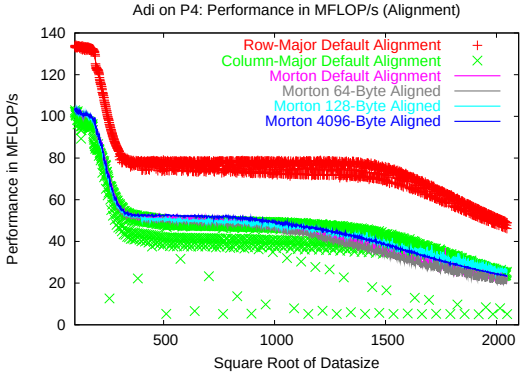
Adi on Athlon: Conditional Win over Column-Major

- There is a cross-over between default Morton and page-aligned Morton at around 900×900.
- For large datasizes, page-aligned is the best Morton version.
- Notice some very bad performance drops on CM for individual problem sizes.



Adi on Pentium III: Win over Column-Major

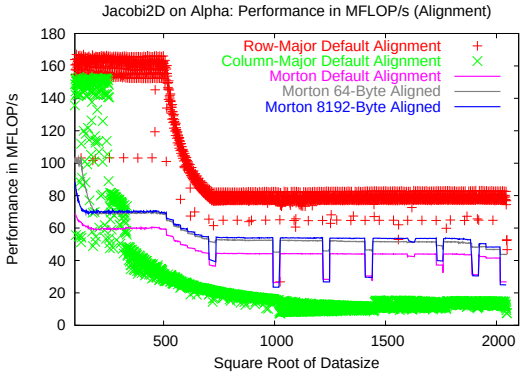
- Morton with default alignment virtually coincides with CM.
- Morton aligned to either L2 cache line length (32 bytes) or page size leads to a clear improvement.
- Only becomes a win with alignment.



Adi on Pentium 4: Lose

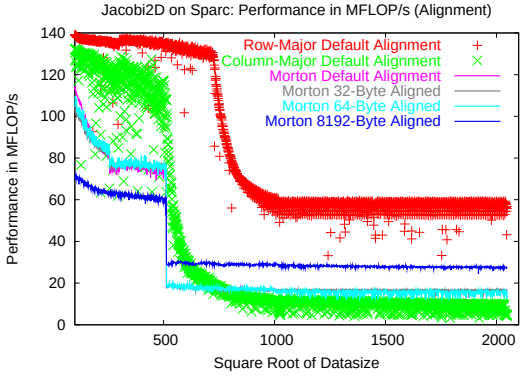
- Morton generally performs no better than CM.
- Notice, however, some really bad drops in CM performance for some datasizes, which Morton does not suffer.
- For large problem sizes, L2 aligned is slightly faster than page-aligned Morton. However, page-aligned has lower variance.

Figure 10. **ADI performance in MFLOPs on Athlon, P3 and P4.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).



Jacobi2D on Alpha: Win over Column-Major

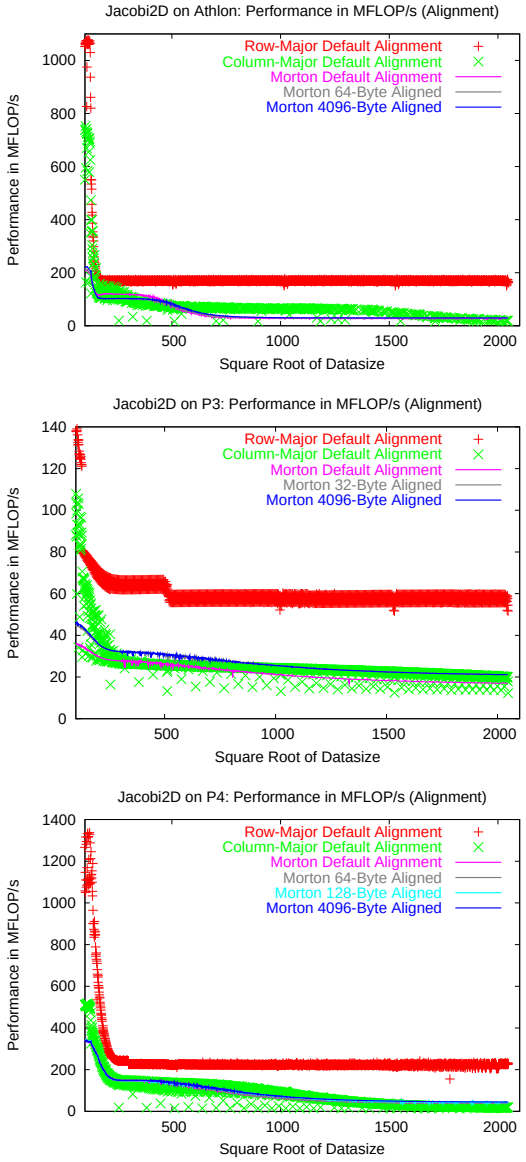
- Alignment to page or L2 cache line length improves Morton performance.
- RM performance drops off after 512×512. Assuming a working set of two arrays of 512×512 doubles, this is the point where the working set exceeds L2 cache size (4MB, direct mapped). RM performance levels off after 725×725, which appears to be when one single 725×725 array of doubles exceeds the L2 cache size.



Jacobi2D on Sparc: Win over Column-Major

- There is a sharp drop in Morton performance occurs at 512×512.
- There is a cross-over between page-aligned Morton and all other Morton versions at 512×512.
- Morton only becomes a win when arrays are page-aligned.
- RM performance drops off after 725×725. Assuming a working set of two arrays of 725×725 doubles, this is the point where the working set exceeds L2 cache size (8MB, direct mapped).

Figure 11. **Jacobi2D performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).



Jacobi2D on Athlon: Lose

- Notice there is a cross-over between the default and L2-/page-aligned Morton versions at around 450×450.
- For large problem sizes, page-aligned Morton is very slightly better than default-aligned.

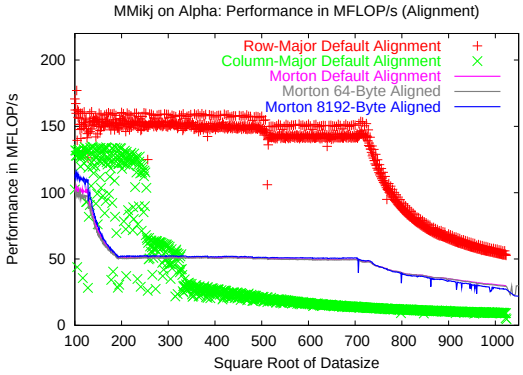
Jacobi2D on Pentium III: No Win

- L2- and page-aligned Morton is slightly better than default-aligned.

Jacobi2D on Pentium 4: Conditional Win over Column-Major

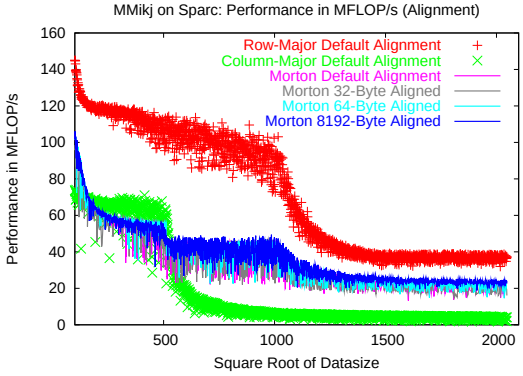
- Page-aligned and L2-aligned Morton are slightly better than default-aligned.
- Morton becomes a win for very large problem sizes.

Figure 12. **Jacobi2D performance in MFLOPs on Athlon, P3 and P4.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).



MMikj on Alpha: Win over Column-Major

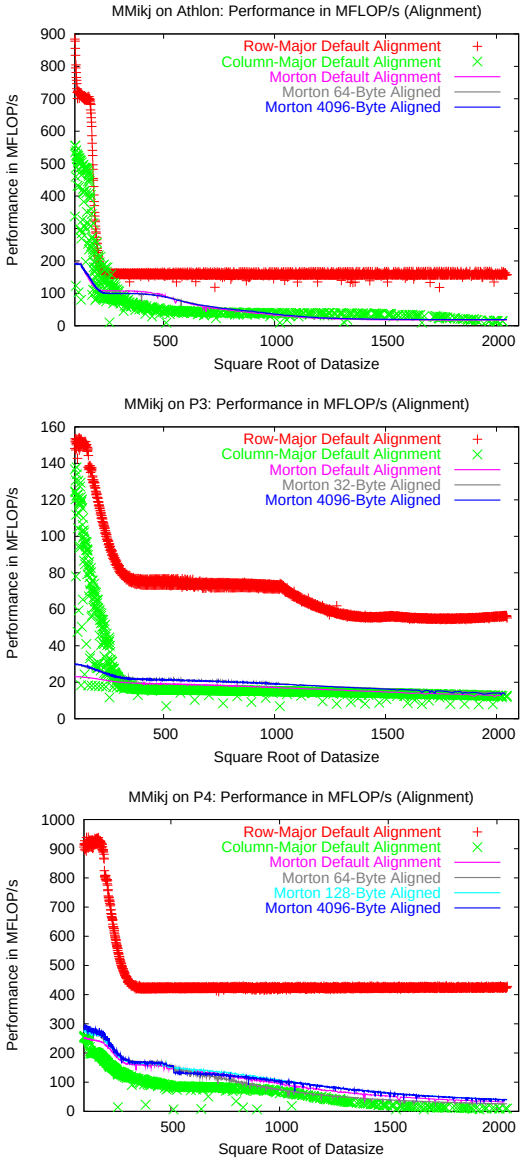
- Notice upper limit is 1024×1024.
- Alignment makes little difference to the overall Morton performance.
- The drop in RM (and Morton) performance occurs at 725×725. This corresponds to the datasize where one array of 725×725 doubles exceeds the L2 cache size (4 MB, direct mapped).



MMikj on Sparc: Win over column-Major

- Page-aligned Morton is slightly faster than the other versions.
- The drop in RM (and Morton) performance occurs at 1024×1024. This corresponds to the datasize where one array of 1024×1024 doubles exceeds the L2 cache size (8MB, direct mapped).

Figure 13. **MMikj performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).



MMikj on Athlon: Lose

- Notice there is a cross-over between the default and L2-/page-aligned Morton versions at around 500×500.
- For large problem sizes, page-aligned Morton is very slightly better than default-aligned.

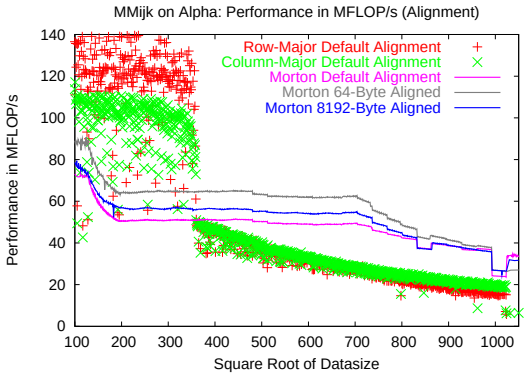
MMikj on Pentium III: No Win

- For large problem sizes, L2- and page-aligned Morton is very slightly better than default-aligned.

MMikj on Pentium 4: Slight Win

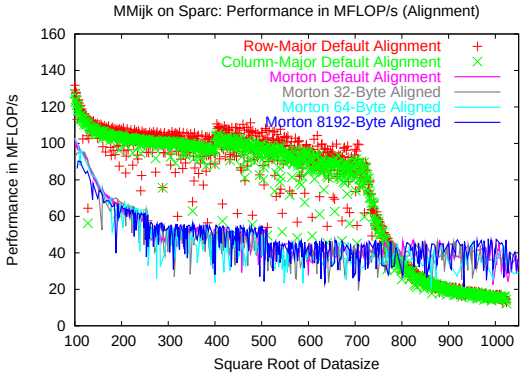
- L2 (128 byte) and page-aligned Morton is better than Morton with default alignment.
- L1 (64 byte) alignment is slightly worse than default-aligned.

Figure 14. **MMikj performance in MFLOPs on Athlon, P3 and P4.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).



MMijk on Alpha: Win over both Row-Major and Column-Major

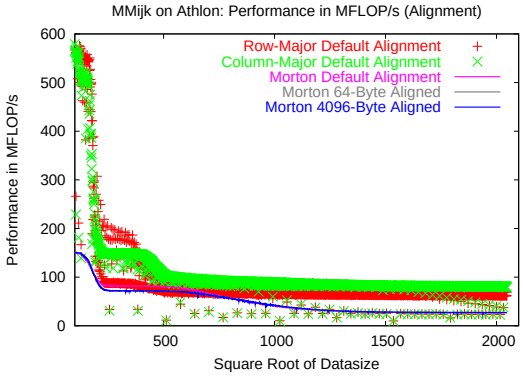
- Notice upper limit is 1024×1024.
- Notice the sharp drop in RM and CM performance around 360×360.
- Page-aligned Morton is faster than default, but L2-aligned is faster than page-aligned.
- For Morton on problem sizes $832(= 26 * 32) - 864(= 27 * 32)$ and $992(= 31 * 32) - 1024(= 32 * 32)$ we see a noticeable drop in performance, presumably due to some self- or inter-array interference effect in the direct-mapped L2 cache.



MMijk on Sparc: Potential Win over both Row-Major and Column-Major

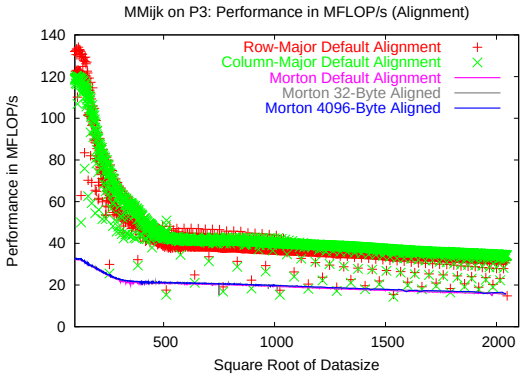
- Notice upper limit is 1024×1024.
- Notice the sharp drop in RM and CM performance around 720×720.
- Notice for large problem sizes Morton is faster than either lexicographic layout.

Figure 15. **MMijk performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).



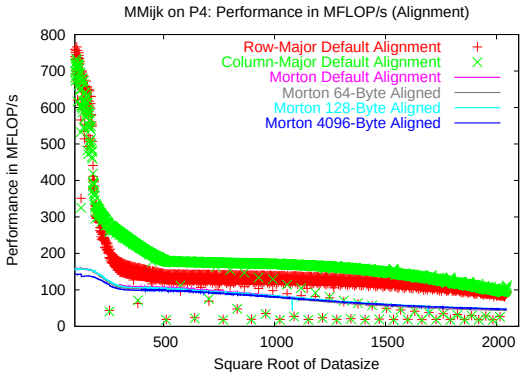
MMijk on Athlon: Lose

- Default Morton is faster than L2- or page-aligned for smaller problem sizes.
- RM data is trimodal.



MMijk on Pentium III: Lose

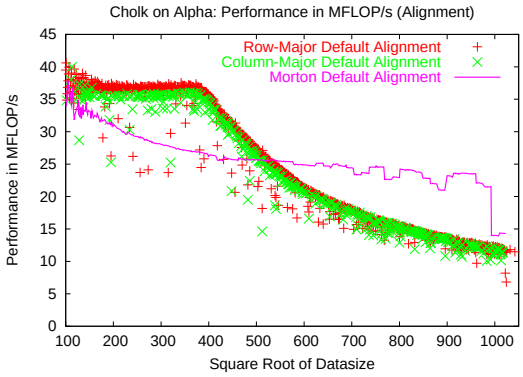
- The RM and CM performance across the range of data sizes is very close.



MMijk on Pentium 4: Lose

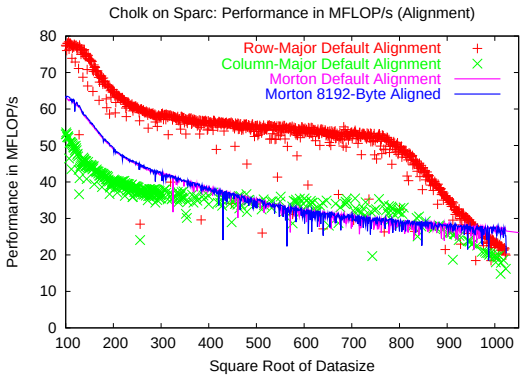
- L2 (128 byte) and page-aligned Morton is slightly better than Morton with default alignment.
- L1 (64 byte) alignment is slightly worse than default-aligned.
- Notice that for some individual problem sizes, both RM and CM drop drastically below Morton.

Figure 16. **MMijk performance in MFLOPs on Athlon, P3 and P4.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).



Cholk on Alpha: Win

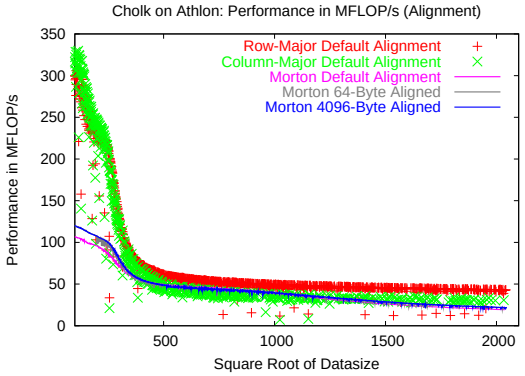
- Notice upper limit is 1024×1024.
- For data sizes larger than around 550×550, Morton is faster than either column-major or row-major.



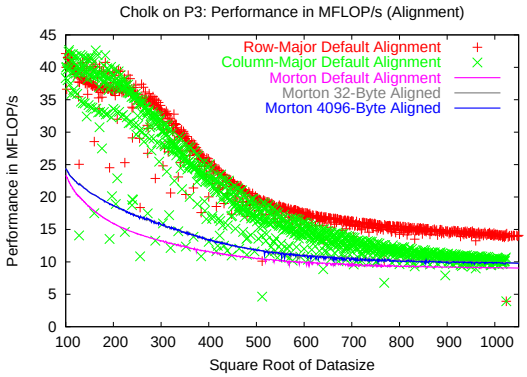
Cholk on Sparc: Potential Win

- Notice upper limit is 1024×1024.
- For data sizes larger than around 950×950, Morton is faster than either column-major or row-major.

Figure 17. **Cholesky k-variant performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).

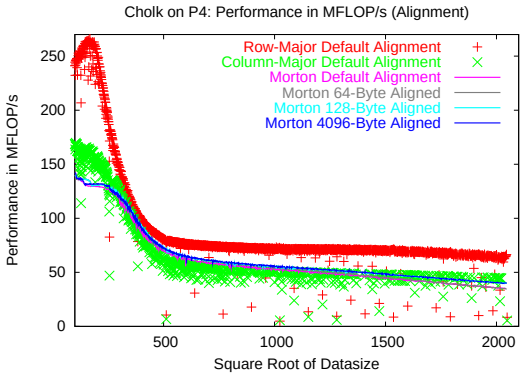


Chol-k on Athlon: Lose



Chol-k on Pentium III: Lose

- Notice upper limit is 1024×1024.



Chol-k on Pentium 4: Marginal Lose

Figure 18. **Cholesky k-variant performance in MFLOPs on Athlon, P3 and P4.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and signifi cant sizes in the memory hierarchy (cache line lengths and page size).

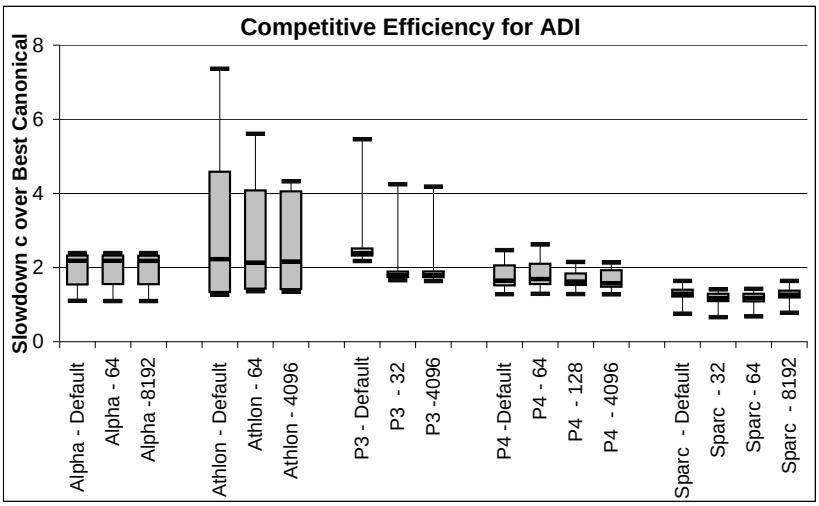


Figure 19. **Range of slowdown factors c over the best lexicographic layout for Adi.** For each machine / layout pair, we show the range of all c factors as a box-and-whisker plot: The gray box indicates the area where 50% of the datapoints lie while the "whiskers" indicate the maximum and minimum points. The line in the middle of the grey box indicates the median. Most median slowdown factors for Adi are around or below 2. The very high maximum factors encountered on Athlon and P3 occur for small datasizes.

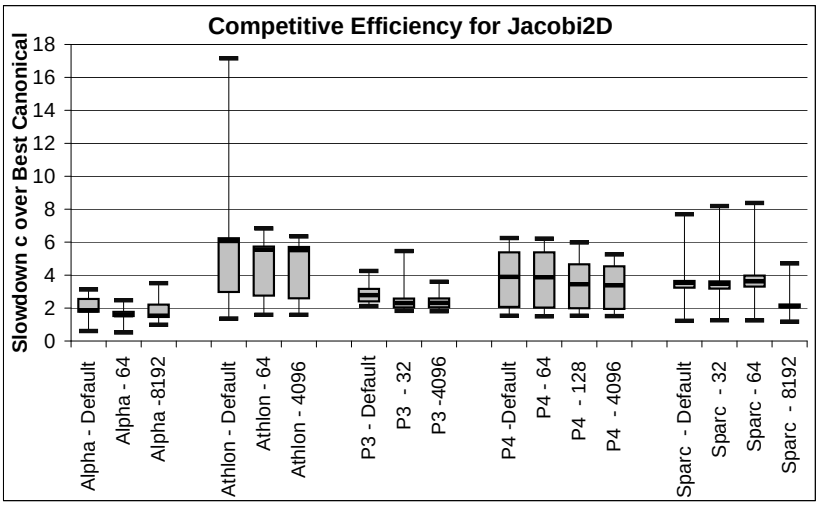


Figure 20. **Range of slowdown factors c over the best lexicographic layout for Jacobi2D.** For each machine / layout pair, we show the range of all c factors as a box-and-whisker plot: The gray box indicates the area where 50% of the datapoints lie while the "whiskers" indicate the maximum and minimum points. The line in the middle of the grey box indicates the median. Using page-aligned arrays, the median slowdown factor on Alpha, P3 and Sparc is close to 2. The high maximum on Athlon for default-alignment occurs for very small datasizes; note however, that alignment improves this considerably.

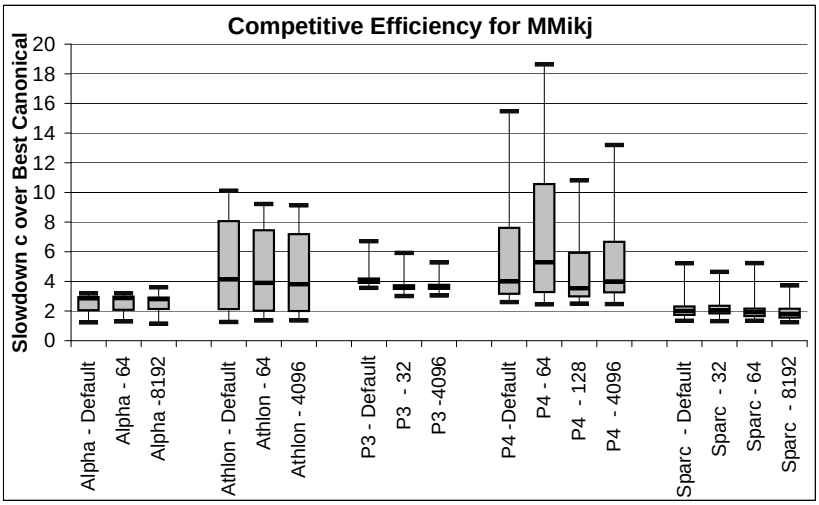


Figure 21. **Range of slowdown factors c over the best lexicographic layout for MMikj.** For each machine / layout pair, we show the range of all c factors as a box-and-whisker plot: The gray box indicates the area where 50% of the datapoints lie while the "whiskers" indicate the maximum and minimum points. The line in the middle of the grey box indicates the median. On i386 architectures, Morton suffers a disappointingly large slowdown. In particular on the Pentium 4, the stride-1 traversal of arrays in the MMikj algorithm is handled very well by the memory system. In contrast, Morton almost certainly does not benefi t from features such as the P4's hardware prefetching mechanism, which is based on detecting regular access strides.

Impact of Alignment of Morton Arrays. For all experiments, except for MMijk and Cholesky on Athlon, our theoretical conclusions from Section 5 are supported by our experimental data: Padding the base address of Morton order arrays to a significant size in the memory hierarchy, such as cache line size or page size can significantly improve performance.

Considering spatial locality alone, we would expect alignment to the largest significant size, i.e. page size, to have the greatest benefit. This is supported in most, but not all cases by our experimental data, and we assume that where this is not the case (such as MMijk on Alpha), this is due to interference effects.

In some cases, specifically Adi on Athlon, Jacobi on Sparc and MMikj on P4, the improvement with page-alignment is quite significant.

Morton layout can outperform both lexicographic layouts. Notice that there are cases where the minimum, and occasionally (MMijk on Alpha) also the mean slowdown is less than 1, meaning that Morton layout performs better than the best lexicographic layout.

Worst slowdown. Although we have shown that proper alignment can reduce the maximum slowdown factors found, and therefore increase the competitiveness of the basic Morton scheme, the maximum slowdown factors are still very high — the worst probably around 11 for MMikj on P4. In our conclusion, we discuss some further ways in which we hope to improve these figures.

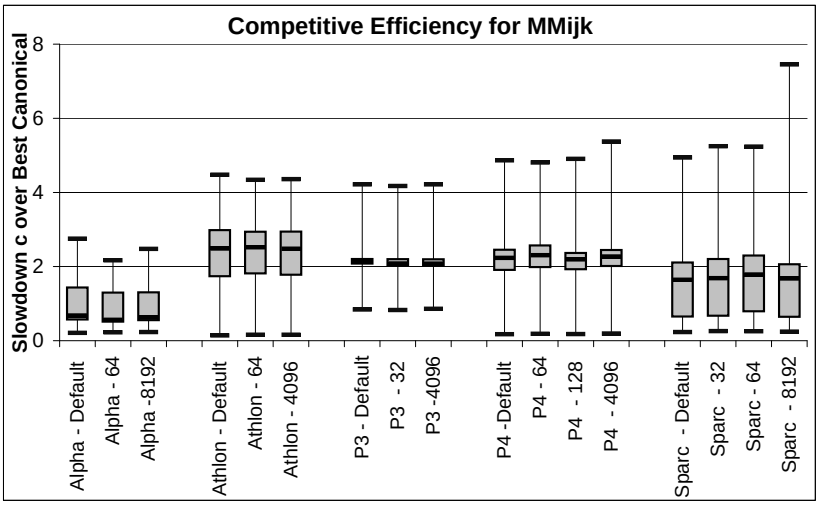


Figure 22. **Range of slowdown factors c over the best lexicographic layout for MMijk.** For each machine / layout pair, we show the range of all c factors as a box-and-whisker plot: The gray box indicates the area where 50% of the datapoints lie while the "whiskers" indicate the maximum and minimum points. The line in the middle of the grey box indicates the median. The MMijk loop suffers from severe performance drops for specifi c datasizes on most architectures for both row-major and column-major layout. Morton layout does not incur this problem, and the minimum slowdown on all architectures here is less than 1 —i.e. on all architectures there are problem sizes where Morton out-performs the better of the two lexicographic layouts. On Alpha, Morton out-performs both lexicographic layouts for all datasizes larger than 360×360, hence the median slowdown of less than 1.

7. Conclusions and directions for further research

Using a small suite of dense kernels working on two-dimensional arrays, we have studied the impact of poor array layout. 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. A number of interesting issues remain:

- **Non-square cache blocks and pages**
  In our brief analysis of spatial locality using Morton layout (Section 3.6, Figure 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. As we showed in Figure 8, row-major and column-major traversal of Morton layout then lead to differing spatial locality. A more subtle non-linear layout might address this.
- **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

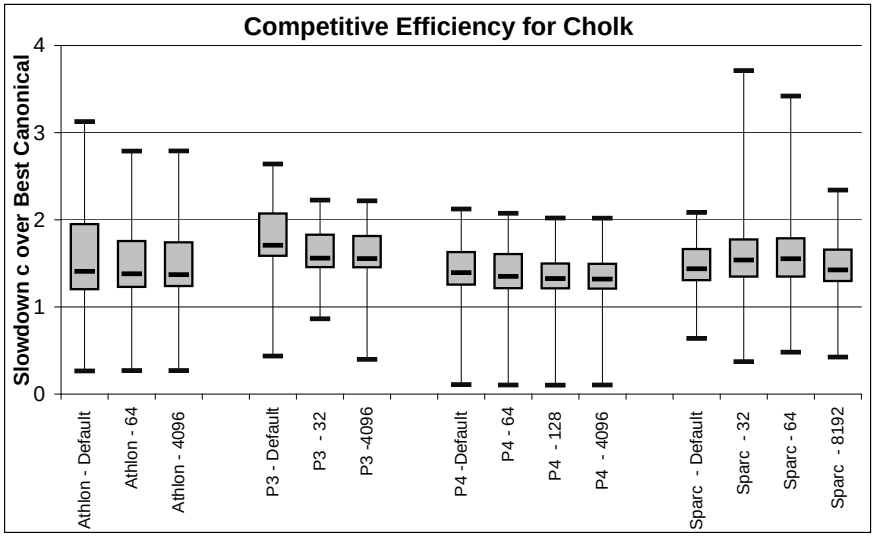


Figure 23. **Range of slowdown factors c over the best lexicographic layout for Cholk.** For each machine / layout pair, we show the range of all c factors as a box-and-whisker plot: The gray box indicates the area where 50% of the datapoints lie while the "whiskers" indicate the maximum and minimum points. The line in the middle of the grey box indicates the median. Median slowdown for all architectures is less than factor 2.

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.* [15]). 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.
- **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.
- **Performance analysis using performance counters**
  We plan to use performance counter instrumentation in order to better understand and analyse why we see the performance patterns that we have reported in this paper.

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). We thank the anonymous reviewers for their constructive suggestions on improving the paper.

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. Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In *PPoPP '95: Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 30(8) of *ACM SIGPLAN Notices*, pages 166–178, August 1995.
3. Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *ICS '99: International Conference on Supercomputing*, pages 444–453, June 20–25, 1999.
4. Michal Cierniak and Wei Li. Unifying data and control transformations for distributed share d-memory machines. In *PLDI '95: ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.
5. Michal Cierniak and Wei Li. Validity of interprocedural data remapping. Technical Report 642, University of Rochester, Computer Science Department, November 1996.
6. Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. In *IEEE International Conference on Computer Languages*, pages 96–105, March 1992.
7. W. J. Dixon. Ratios involving extreme values. *The Annals of Mathematical Statistics*, 22(1):68–78, March 1951.
8. Amos Fiat, Richard Karp, Mike Luby, Lyle McGeoch, Daniel Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, December 1991.
9. Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. In *POPL '78: Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 1–8, January 1978.
10. C. Barry Jay. Shape in computing. *ACM Computing Surveys*, 28(2):355–357, 1996.
11. Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, N. Shenoy, and Prithviraj Banerjee. Enhancing spatial locality via data layout optimizations. In *Proceedings of Euro-Par'98*, number 1470 in LNCS, pages 422–434, September 1998.
12. Monica S. Lam, Edward E. Rothenberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS '91: Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.
13. 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.
14. Michael F. P. O'Boyle and Peter M. W. Knijnenburg. Integrating loop and data transformantions for global optimisation. *Journal of Parallel and Distributed Computing*, 62:563–590, 2002.
15. Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *PLDI '98: ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 38–49, June 1998.
16. Lothar Sachs. *Statistische Methoden*. Springer Verlag, 5^{th} edition, 1982.
17. http://www.specbench.org/.
18. Jeyarajan Thiyagalingam and Paul H. J. Kelly. Is Morton layout competitive for large two-dimensional arrays? In *Euro-Par 2002: 8^{th} International Euro-Par Conference*, number 2400 in LNCS, pages 280–288, August 2002.
19. David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for Morton-order matrices. In *PPoPP 01: Principles and Practice of Parallel Programming*, volume 36(7) of *ACM SIGPLAN Notices*, July 2001.
20. Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26(6) of *ACM SIGPLAN Notices*, pages 30–44, June 1991.