Is Morton layout competitive for large two-dimensional arrays?

Jeyarajan Thiyagalingam and Paul H J Kelly

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

Abstract. Two-dimensional arrays are generally arranged in memory in row-major order or column-major order. 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 — up to a factor of 20. With knowledge of how the array will be used, it is often possible to choose between the two layouts in order to maximise spatial locality. In this paper we study the Morton storage layout, which has substantial spatial locality whether traversed in row-major or column-major order. We present results from a suite of simple application kernels which show that, on the AMD Athlon and Pentium III, for arrays larger than 256×256, Morton array layout, even implemented with a lookup table with no compiler support, is always within 61% of both row-major and column-major — and is sometimes faster.

1 Introduction

Every student learns that multidimensional arrays are stored in "lexicographic" order: row-major (for Pascal etc) or column-major (for Fortran). Modern processors rely heavily on caches and spatial locality, and this works well when the access pattern matches the storage layout. However, accessing a row-major array in column-major order leads to dismal performance (and vice-versa). The Morton layout for arrays (for background and history see [13, 3]) offers a compromise, with some spatial locality whether traversed in row-major or column-major order — although in neither case is spatial locality as high as the best case for row-major or column-major. A further disadvantage is the cost of calculating addresses. So, should language implementors consider using Morton layout for all multidimensional arrays? This paper explores this question, and provides some qualified answers.

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

analyse and optimise the code, but we recognise that very often, neither is the case.

The idea is this: if we know how the array is going to be used, we could choose optimally between the two lexicographic layouts. If we don't know how the array will be used, we can guess. If we guess right, we can expect good performance. If wrong, we may suffer very badly.

One way to evaluate the use of Morton layout to avoid such worst-case behaviour is by analogy with competitive on-line algorithms. Suppose we have an optimal array layout scheme OPT. Following [9, 11], 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 σ,

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

In this paper we evaluate experimentally whether the Morton layout is c-competitive with respect to a scheme OPT, in which the faster of the two lexicographic layouts is chosen. The key issue is whether the competitive efficiency c is low enough in practice.

We use a small suite of simple application kernels to test this hypothesis, and evaluate the competitive efficiency c for various computer systems. We also evaluate the slowdown which occurs with these applications when the wrong layout is chosen.

2 Related work

Compiler techniques Locality can be enhanced by restructuring loops to traverse the data in an appropriate order [14, 12]. 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 [10]. Copying can be avoided by building the array in this layout. More generally, storage layout can be selected to match execution order [8]. 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.

Blocked and recursively-blocked array layout Wise *et al.* [13] 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 below). 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 (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

Here we briefly review various array mappings and the 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 row-major (for example in Pascal) and Column-major (as used by Fortran) mappings expressed by

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

respectively. We refer to row-major and column-major as lexicographic layouts, i.e. the sort order of the two indices (another term is "canonical"). Historically, array layout has been mandated in the language specification.

3.2 Opaque array storage: array descriptors

In more modern languages, such as Fortran 90 (and notable earlier designs — Algol 68, APL), arrays are represented by a descriptor which provides run-time information on how the address calculation should be done [5]. 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 [7]. This raises performance problems. The storage layout is not known at compile-time — the stride of successive memory accesses depends on how the 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 [4].

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 $N \times M$ array into small, $P \times Q$ row-major subarrays, arranged as a $N/P \times M/Q$ row-major array. We define the blocked row-major mapping function (this is the 4D layout discussed in [3]) as:

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

For example, consider 16-word cache blocks and $P = Q = 4$. Each block holds a $P \times Q = 16$-word subarray. 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 cache hit rates calculated above apply whether the array is accessed in row-major or column-major order (i.e. whether the loop is "`do j...do i`" as shown, or "`do i...do j`").

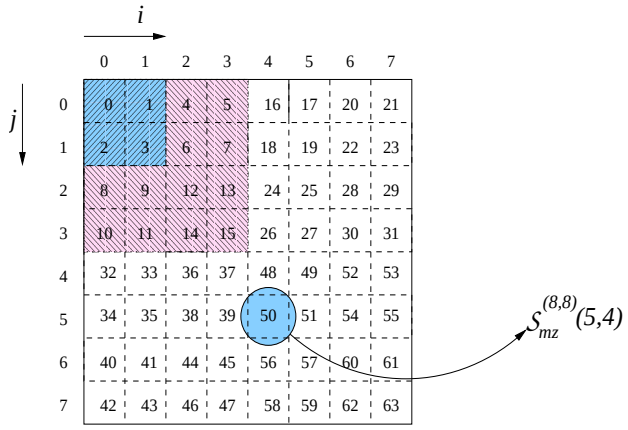

Fig. 1. Morton storage layout for 8×8 array. Location of element $A[4,5]$ is calculated by interleaving "dilated" representations of 4 and 5 bitwise: $\mathcal{D}_0(4) = 010000_2$, $\mathcal{D}_1(5) = 100010_2$. $\mathcal{S}_{mz}(5,4) = \mathcal{D}_0(5) \mid \mathcal{D}_1(4) = 110010_2 = 50_{10}$. A 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.

3.4 Larger cache blocks and virtual memory pages

With larger cache blocks, we can get a higher hit rate. Although many current processors have the same cache block size at all levels of the cache, there are

exceptions (e.g. SunFire 6800 has a blocksize of 32 bytes at level 1, and 64 bytes at level 2). Virtual memory pages are also a major consideration — a typical 64-entry data TLB with 8KByte pages has an effective span of $64 \times 8 = 512KB$.

3.5 Recursive blocking

Unfortunately, if the blocked row-major array is traversed in row-major order, only one subarray per page is usable. Thus, we find that the blocked row-major layout is still biased towards column-major traversal. We can overcome this by applying the blocking again, recursively. Thus, each 8KByte page (1024 doubles) would hold a 16×16 array of 2×2-element subarrays.

In general, modern systems have a deep memory hierarchy, with block size, capacity and access time increasing geometrically with depth [1]. Blocking should be applied for each level. However, we must now consider the complexity of calculating array locations.

```
#define ONES_1 0x55555555
#define ONES_0 0xaaaaaaaa
#define INC_1(vx) (((vx + ONES_0) + 1) & ONES_1)
#define INC_0(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_0, j_1, k_0;
  double r;
  int SZ_0 = Dilate(SZ);
  int SZ_1 = SZ_0 << 1;
  for (i_0 = 0; i_0 < SZ_0; i_0 = INC_0(i_0))
    for (k_0 = 0; k_0 < SZ_0; k_0 = INC_0(k_0)){
      unsigned int k_1 = k_0 << 1;
      r = A[i_0 + k_1];
      for (j_1 = 0; j_1 < SZ_1; j_1 = INC_1(j_1))
        C[i_0 + j_1] += r * B[k_0 + j_1];
    }
}
```

Fig. 2. Morton-order matrix-multiply implementation using dilated arithmetic for the address calculation. Variables `i_0` and `k_0` are dilated representations of the loop control counter $\mathcal{D}_0(\mathtt{i})$ and $\mathcal{D}_0(\mathtt{k})$. Counter `j` is represented by `j_1`$= \mathcal{D}_1(\mathtt{j})$. The function `Dilate` converts a normal integer into a dilated integer.

3.6 Bit-interleaving

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

$$\mathcal{B}(i) = i_{n-1}i_{n-2}\ldots i_3 i_2 i_1 i_0 \qquad \text{and} \qquad \mathcal{B}(j) = j_{n-1}j_{n-2}\ldots j_3 j_2 j_1 j_0$$

respectively. Now the lexicographic mappings can be expressed as bit-concatenation (written "||"):

$$\begin{aligned}
\mathcal{S}_{rm}^{(N,M)}(i,j) &= N \times i + j = \mathcal{B}(i)\|\mathcal{B}(j) \\
&= i_{n-1}i_{n-2}\ldots i_3 i_2 i_1 i_0 j_{n-1}j_{n-2}\ldots j_3 j_2 j_1 j_0 \\
\mathcal{S}_{cm}^{(N,M)}(i,j) &= i + M \times j = \mathcal{B}(j)\|\mathcal{B}(i) \\
&= j_{n-1}j_{n-2}\ldots j_3 j_2 j_1 j_0 i_{n-1}i_{n-2}\ldots i_3 i_2 i_1 i_0
\end{aligned}$$

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

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

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

$$\mathcal{S}_{mz}^{(N,M)}(i,j) = i_{n-1}j_{n-1}i_{n-2}j_{n-2}\ldots i_3 j_3 i_2 j_2 i_1 j_1 i_0 j_0$$

This mapping is called the Morton Z-order, and is illustrated in Fig. 1.

3.7 Cache performance with Morton-order layout

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. In fact, this applies given any cache hierarchy with even power-of-two block size at each level. This is illustrated in Fig. 1. The problem of calculating the actual cache performance with Morton layout is somewhat involved; an interesting analysis for matrix multiply is presented in [6].

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

Fig. 3. Morton-order matrix-multiply implementation using table lookup for the address calculation. The compiler detects that `MortonTabEven[i]` and `MortonTabEven[k]` are loop invariant, leaving just one table lookup in the inner loop.

4 Morton-order address calculation

4.1 Dilated arithmetic

Bit-interleaving is too complex to execute at every loop iteration. Wise *et al.* [13] 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

$$\mathcal{B}(\mathcal{D}_0(i)) = 0i_{n-1}0i_{n-2}0\ldots 0i_2 0i_1 0i_0 \quad \text{and} \quad \mathcal{B}(\mathcal{D}_1(i)) = i_{n-1}0i_{n-2}0\ldots i_2 0i_1 0i_0 0$$

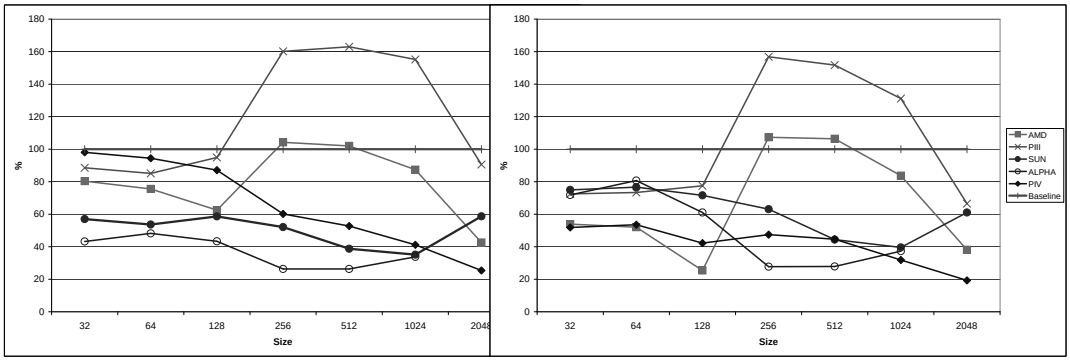

Fig. 4. Matrix multiply (ikj) performance (in MFLOPs) of *(left)* dilated arithmetic Morton address calculation (see Fig. 2) versus *(right)* table-based Morton address calculation (see Fig. 3). The graphs show MFLOPs normalised to the performance achieved by the standard row-major ikj implementation at each problem size on each system. Details of the systems are given in Table 1. At worst, the table lookup scheme is 46% slower than the dilated-arithmetic scheme on P4. For problem sizes larger than 256 the worst figure is 24% on PIII. On the SunFire 6800 the lookup table implementation is always faster. Larger numbers are better.

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

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

where "&" denotes bitwise-and, and

$$\mathcal{B}(\mathrm{Ones}_0) = 01010\ldots10101 \qquad \text{and} \qquad \mathcal{B}(\mathrm{Ones}_1) = 10101\ldots01010$$

This is illustrated in Fig. 2, which shows the ikj variant of matrix multiply.

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 ikj matrix multiply variant in Fig. 3. Note that the table accesses are very likely cache hits, as their range is small and they have 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.

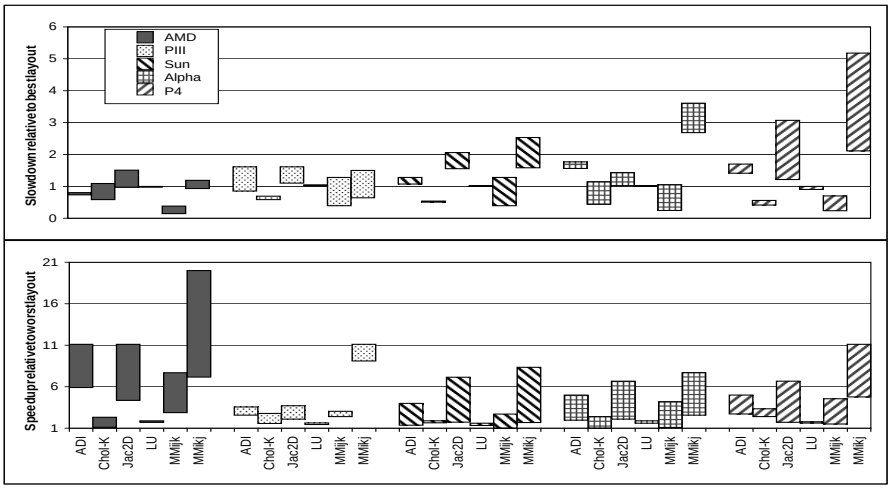

Fig. 5. Performance of table-lookup-based implementation of Morton layout for various common dense kernels. In the upper graph we show how much slower Morton layout can be compared with row-major layout (which for our benchmarks is usually fastest). In each case we show the maximum and minimum slowdown over a range of problem sizes from 256×256 to 2048×2048. In the lower graph, we show how much faster Morton layout can be compared with column-major layout. In each case we show the maximum and minimum speedup over the same range of problem sizes.

Fig. 4 shows the performance of these two variants on a variety of computer systems. In the remainder of the paper, we use the table lookup scheme exclusively. With compiler support, many applications could benefit from the dilated arithmetic approach, leading in many cases to more positive conclusions.

5 Experimental results

We have argued that Morton layout is a good compromise between row-major and column-major. The notion of c-competitiveness provides a way to quantify this claim. The competitive efficiency c is the maximum slowdown we should suffer relative to making the best layout choice.

To test this experimentally, we have collected a suite of simple implementations of standard numerical kernels operating on two-dimensional arrays:

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)
LU	LU decomposition with pivoting (based on Numerical Recipes)
Jacobi2D	Two-dimensional four-point stencil smoother
ADI	Alternating-direction implicit kernel, ij,ij order
Cholesky	k variant (usually poor due to large stride)

Alpha	Alpha 21264 (EV6) 500MHz,
Compaq	L1 D-cache: 2-way, 64KB, 64B cache block
AlphaServer ES40	L2 cache: direct mapped, 4MB. Compiler: Compaq C V6.1-020 "-fast"
Sun	UltraSparc III (v9) 750MHz
SunFire 6800	L1 D-cache: 4-way, 64KB, 32B cache block
	L2 cache: direct-mapped, 8MB.
	Compiler: Sun Workshop 6 "-xO5" (update 1 C 5.2 Patch 109513-07)
PIII	Intel Pentium III Coppermine, 1GHz
	L1 D-cache: 4-way, 16KB, 32B cache block
	L2 cache: 8-way 256KB, sectored 32B cache block
	512MB SDRAM. Compiler "gcc-2.95 -O3"
P4	Pentium 4, 1.3 GHz
	L1 D-cache: 8-way, 8KB, sectored 64B cache block
	L2 cache: 8-way, 256KB, sectored 64B cache block
	256MB RDRAM. Compiler "gcc-2.95 -O3"
AMD	AMD Athlon Thunderbird, 1.4GHZ
	L1 D-Cache: 2-way, 64KB, 64B cache block
	L2 cache: 8-way, 256KB, 64B cache block
	512MB DDR RAM. Compiler "gcc-2.95 -O3"

Table 1. Cache and CPU configurations used in the experiments.

In each case we run the code on square arrays of various sizes, repeating the calculation if necessary to ensure adequate timing resolution. The system configurations are detailed in Table 1. Table 2 shows the baseline performance achieved by each machine using standard row-major layout. Results using Morton layout are summarised in Fig. 5 and shown in more detail in Figures 6 and 7.

	ADI		Chol-K		Jacobi2D		LU		MMijk		MMikj	
	min	*max*	*min*	*max*	*min*	*max*	*min*	*max*	*min*	*max*	*min*	*max*
AMD	33.81	34.72	11.05	47.61	195.84	199.25	16.76	83.02	10.05	32.18	90.27	92.72
PIII	21.17	23.71	16.05	26.99	122.21	128.90	32.44	69.32	27.44	37.19	58.90	59.20
SunFire	37.64	40.35	16.12	21.62	140.69	411.78	44.48	77.08	16.16	69.90	125.57	137.24
Alpha	49.77	63.47	12.02	41.90	120.23	245.53	30.22	112.28	14.41	95.34	148.78	254.13
P4	65.04	67.56	23.05	43.15	410.16	419.32	41.72	73.98	32.35	34.98	293.51	297.92

Table 2. Performance of various kernels on different systems. For each kernel, for each machine, we show performance range in MFLOPs for row-major array layout, for array sizes ranging from 256×256 to 1024×1024.

Our results show that Morton layout is not effective for arrays smaller than 256×256. We therefore confine our attention to larger problem sizes. On the AMD Athlon PC and Pentium III, we find that Morton layout is often faster than both row-major and column-major, and is never more than 61% slower. Furthermore, the costs of poor layout choice on these machines are particularly acute - in extreme cases a factor of 20. We have only studied up to 2048×2048 (32MB), and further investigation is needed for very large problems.

On the other machines, the picture is less clear. Kernels with high spatial locality, such as MMikj and Jacobi2D, run close to the machine's peak performance; so bandwidth to L1 cache for table access is probably a major factor.

6 Conclusions and directions for further research

The main contributions of this paper are:

- Using a small suite of dense kernels working on two-dimensional arrays, we studied the impact of poor array layout/array traversal order. If an array's layout does not match the traversal order, performance is poor, with a slowdown of more than 20 (Matrix multiply, ikj variant, on the AMD Athlon).
- On the AMD Athlon and Pentium III, for arrays larger than 256×256, we found that Morton array layout, even implemented with a lookup table with no compiler support, is always within 61% of both row-major and column-major. In fact, it is sometimes faster.
- On other machines, the benefits can also be very large — but further work is needed to avoid serious slowdown for some high-performance kernels.
- Using a lookup-table for address calculation allows flexible selection of fine-grain non-linear array layout, while still offering attractive performance compared with lexicographic layouts, on untiled loops.

The advantage of Morton layout on existing codes is unlikely to be large as users normally do avoid worst-case performance. However, simplifying the performance model should allow programmers to focus on building functionally-robust software. Furthermore, if a loop can be tiled (or shackled, or executed in a recursive form with high temporal reuse) the overheads of our lookup table scheme are excessive. Layout then has to be selected in combination with loop restructuring.

- The next step is building Morton layout into a compiler, or perhaps a self-optimizing BLAS library [2] (which would allow run-time layout selection).
- It should be possible to achieve better results using competitive redistribution - i.e. instrument memory accesses and copy the array into a more appropriate distribution if indicated.
- We should evaluate adding hardware support for non-linear layouts.
- We have not used non-square arrays in this paper, but the approach handles them reasonably effectively (see Section 4.2), at the cost of padding each dimension to the next power of two.
- In our brief analysis of spatial locality using Morton layout (Section 3.7, Fig. 1), we assumed that cache blocks and VM pages are a *square* (even) power of two. This depends on the array's element size, and is often not the case. Then, row-major and column-major traversal of Morton layout lead to differing spatial locality. A more subtle non-linear layout could address this.
- It seems less likely that Morton layout can offer a competitive compromise for arrays with more than two dimensions.

Acknowledgements. This work was partly supported by mi2g Software, and a Universities UK Overseas Research Scholarship. We also thank Imperial College Parallel Computing Centre (ICPC) for access to their equipment.

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. Olav Beckmann and Paul H. J. Kelly. Efficient interprocedural data placement optimisation in a parallel library. In *LCR98: Languages, Compilers and Run-time Systems for Scalable Computers*, number 1511 in LNCS, pages 123–138. Springer-Verlag, May 1998.
3. Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing*, pages 444–453, 1999.
4. K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, 1992.
5. Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. In *Conference record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 1–8. ACM Press, January 1978.
6. P. J. Hanlon, D. Chung, S. Chatterjee, D. Genius, A. R. Lebeck, , and E. Parker. The combinatorics of cache misses during matrix multiplication. 2000. to appear in the Journal of Computer Sciences and Systems.
7. C. Barry Jay. Shape in computing. *ACM Computing Surveys*, 28(2):355–357, 1996.
8. Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, N. Shenoy, and Prithviraj Banerjee. Enhancing spatial locality via data layout optimizations. In *European Conference on Parallel Processing*, pages 422–434, 1998.
9. A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1), 1988.
10. Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGPLAN Notices*, 26(4):63–74, 1991.
11. Mark Manasse, Lyle McGeoch, and Daniel Sleator. Competitive algorithms for on-line problems. In *Proceedings of the 1988 Twentieth Annual ACM Symposium on Theory of Computing*, pages 322–333. ACM Press New York, NY, USA, 1988.
12. 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.
13. David S. Wise, Jeremy D. Frens, Yuhong Gu, , and Gregory A. Alexander. Language support for Morton-order matrices. In *Proc. 2001 ACM Symp. on Principles and Practice of Parallel Programming, SIGPLAN Not. 36, 7*, 2001.
14. Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

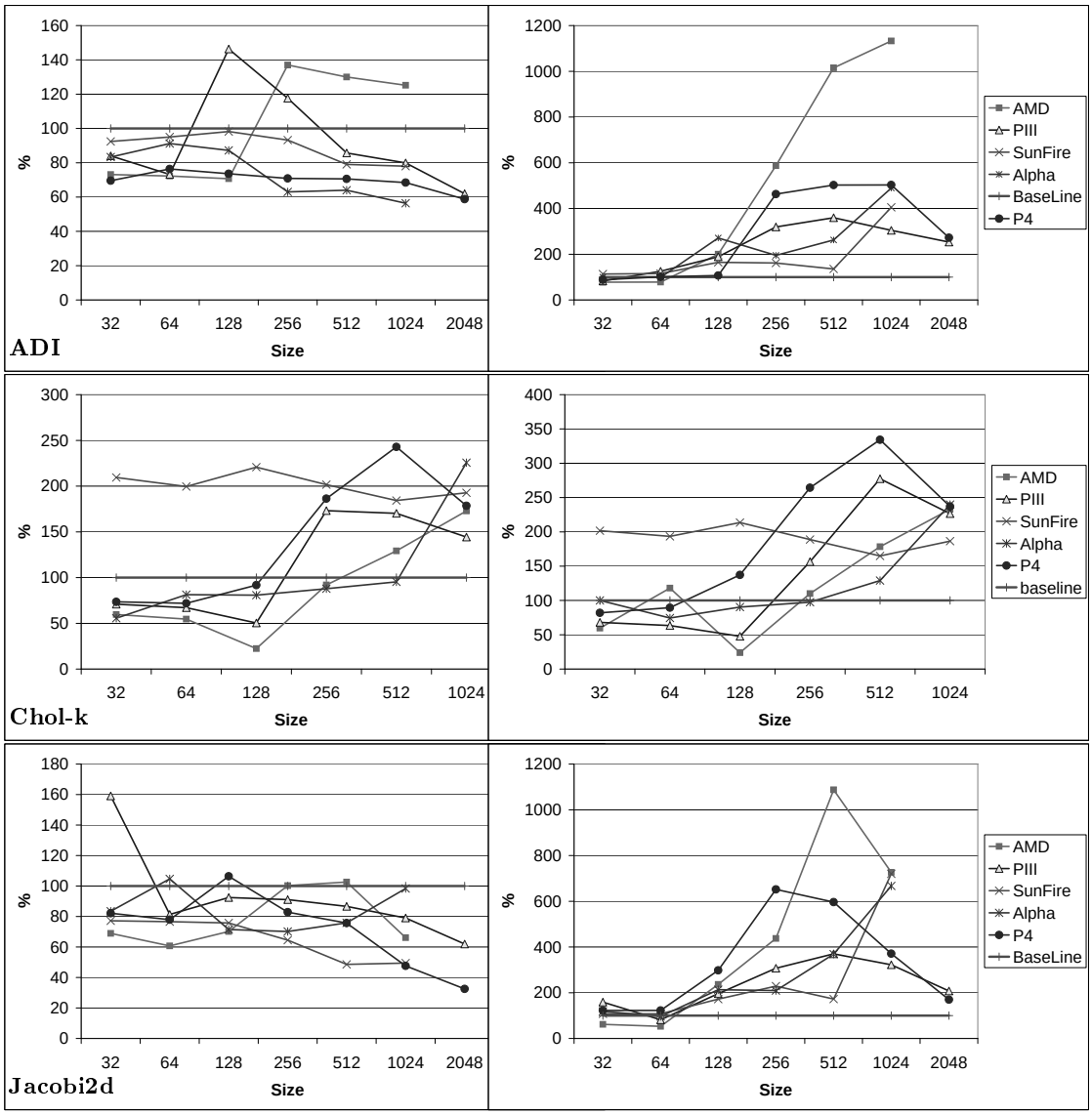

Fig. 6. Various common dense kernels: The left-hand graphs show performance of the Morton layout version relative to the performance (at each problem size) with all arrays in row-major layout. The right-hand graphs show performance of the Morton layout version relative to the (usually much lower) performance with all arrays in column-major. Note the scales of each graph are different. (included for refereeing only; see Section 5)

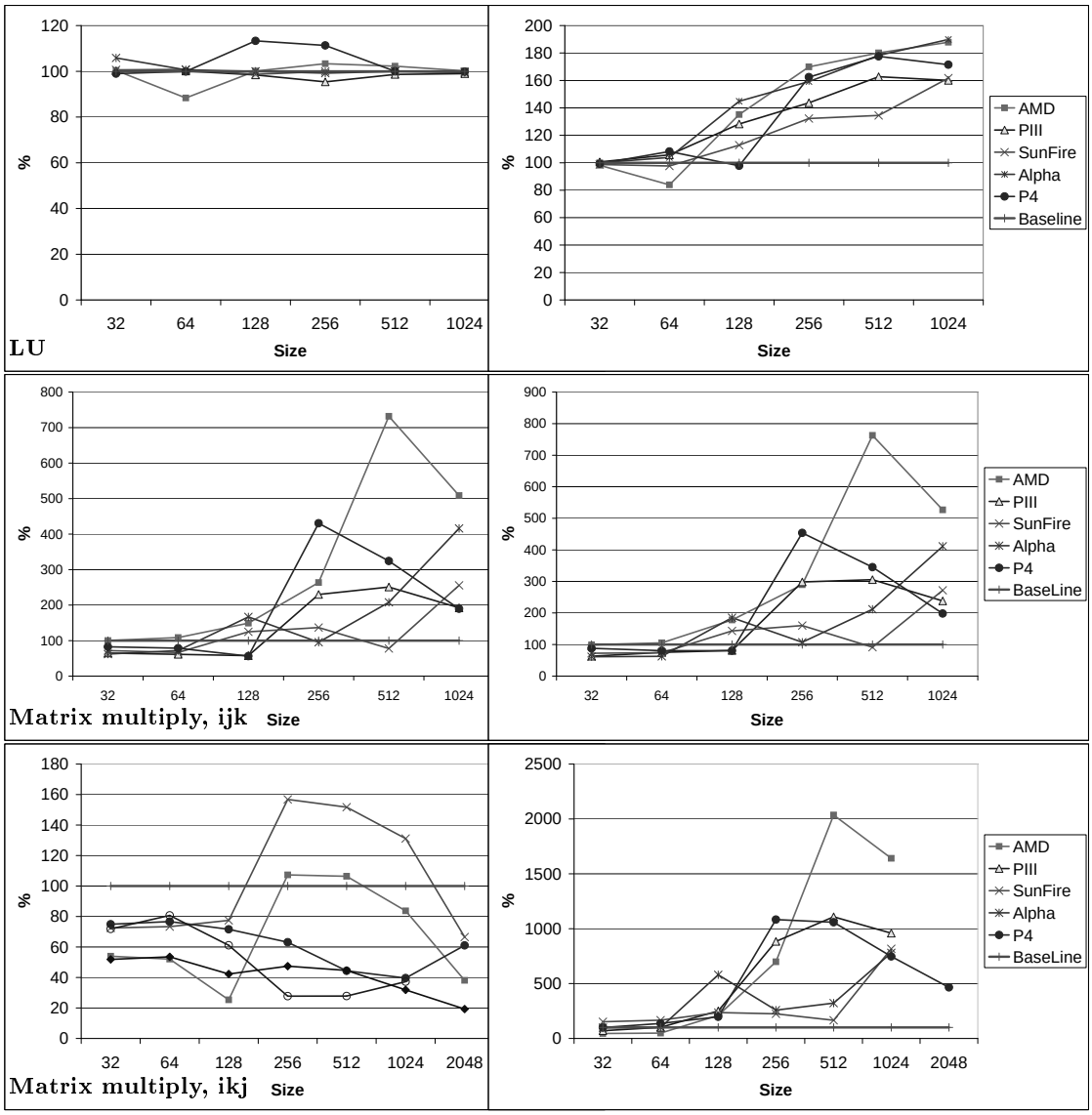

Fig. 7. Various common dense kernels, continued: As in Fig. 6, the left-hand graphs show performance of the Morton layout version relative to the performance (at each problem size) with all arrays in row-major layout. The right-hand graphs show performance of the Morton layout version relative to the (usually much lower) performance with all arrays in column-major. Note the scales of each graph are different. (included for refereeing only; see Section 5)