University of London
Imperial College London
Department of Computing

Alternative Array Storage Layouts for Regular Scientific Programs

Thiyagalingam Jeyarajan

June 2005

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Engineering of the University of London.

Abstract

This thesis concerns techniques for using hierarchical storage formats such as Morton layout as an alternative storage layout for regular scientific programs operating over dense two-dimensional arrays.

Programming languages with support for two-dimensional arrays use one of two linear mappings from two-dimensional array indices to locations in the machine's one-dimensional address space: row-major or column-major. Unfortunately, a wrong choice of one of these two layouts can lead to very poor spatial locality if an array is not traversed in the order in which it is stored. Although a simple loop interchange transformation may resolve the situation, such a transformation may not always be valid due to actual dependencies or dependencies assumed by conservative compiler analyses.

An attractive strategy is to store elements in such a way that both row-major and column-major traversals offer good spatial locality. Hierarchically-blocked non-linear storage layouts, such as Morton ordering have been proposed as a compromise between row-major and column-major layouts. Morton layout offers some spatial locality whether traversed row-wise or column-wise.

The contributions of the thesis are:

- An experimental exploration of performance issues of Morton layout using a suite of micro-benchmark kernels on several hardware platforms.
- We show that the performance of the basic Morton scheme can be improved by aligning the base address of Morton arrays to the largest significant size in the memory hierarchy, namely page size.
- We show that unrolling the loops with strength reduction reduces address calculation overhead associated with the usage of Morton layouts and significantly improves performance of basic Morton scheme.
- We discuss the design issues for implementing a prototype compiler to support Morton layout in large scientific programs including required transformations.

The optimisations we propose here enable Morton layout to be a promising alternative to conventional array layouts. Further, we support our claims through experimental results using selected benchmark kernels on several hardware platforms.

Acknowledgements

First and foremost, I am thankful and grateful to my supervisor, Paul Kelly. During the past three years I have acquired a great deal of technical knowledge, and a rigorous yet practical attitude towards research from Paul Kelly. I feel very honoured to have had the chance to work with Paul Kelly, and I am deeply indebted to him for his constant guidance and support throughout my Ph.D. studies at Imperial College London.

I am also grateful to Olav Beckmann whose help, stimulating suggestions and encouragement has helped me throughout my Ph.D studies. He deserves particular thanks for teaching me about fundamental aspects of experimental frameworks and performance measurement.

Many other people at Imperial College have also been very helpful throughout the time that I have been working on this thesis. My second supervisor Tony Field, my external MPhil assessor William Knottenbelt and Sarah Talbot who has read and commented on various pieces of my work, in particular during my MPhil transfer.

I also would like to thank my undergraduate supervisor Kirthi Walgama and Shanthini Walgama for their support throughout all these years.

I am very grateful to my parents for their love, support and encouragement throughout the time of my studies, and also to my brothers, especially to Anna, my sisters and to Prasshy.

During my PhD studies, many of my friends and their families, including Anpu, Aruna, Ellango, Kannan, Priya, Raj, Seher, Senthil and Sithran have helped me in various ways both financially and morally. Looking back, without their support and encouragement, I never would have been able to start or complete my Ph.D studies.

I am also grateful to the UK Universities for awarding me with the Overseas Research Studentship (UK Universities) Award, which provided the financial backup for my PhD studies at Imperial. I also would like to thank Mi2G and her Chairman D. K. Matai for their generous funding towards the beginning of my studies at Imperial College. Prof. Vladimir Getov of University of Westminster deserves particular mention here for providing me with a supporting research position which funded me towards the end of my Ph.D. studies.

Many thanks also to the Imperial College Parallel Computing Centre for permitting me to use their Alpha and Sun clusters and the Computing Support Group of the Department for allowing me to use their teaching clusters throughout my PhD.

The chain of my gratitude would be definitely incomplete if I would forget to thank the first cause of this chain — all those who educated and guided me.

With the help of
Lord Ganesh ;

To my parents:
Amma and Appa.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Two-dimensional arrays are generally arranged in memory in row-major order (for C, Pascal etc) or column-major order (for Fortran). These layouts are known as lexicographical or canonical layouts. Modern processors rely heavily on caches and prefetching, which work well when the order in which data is accessed matches the storage layout. Sophisticated programmers, or occasionally sophisticated compilers, match loop structures to the language's storage layout in order to maximise spatial locality. However, it is common that either the compiler cannot do this automatically due to conservative compiler analyses, or the transformation may not be valid due to dependencies.

If the access pattern conflicts with the storage layout and if the compiler or programmer is not sophisticated enough to perform the required transformations to resolve this, the resulting performance loss can be dramatic — a factor of 10 or more, as we show in Chapter 4.

An alternative strategy to loop nest transformation is to change the way that elements are laid out in memory — storage layout transformation. Compared to loop nest restructuring, the main advantage of storage layout transformation is that it preserves the semantics of the program in all cases and therefore it is always valid.

Space-filling curves [33, 66, 75] offer several alternative ways of storing array elements in main memory. Storage layouts based on these space-filling curves offer substantial spatial locality whether traversed in row-major or column-major order, subject to certain conditions (see Section 5.2). One such layout is Lebesgue's space-filling or Z-Morton curve and for reasons explored in Chapter 3, this thesis entirely focuses on the Z-Morton layout.

This thesis explores whether and when Morton layout can be an attractive alternative to canonical layouts.

1.1 Approach

Morton layout offers equal spatial locality whether traversed in row-major or column-major order, subject to certain conditions which we explain in Chapter 5 (see Section 5.2). Such a layout is an attractive option when the access pattern conflicts with the order in which array elements are stored. If Morton layout can be used as an alternative, the performance penalty for choosing a wrong layout or wrong traversal order is reduced.

It is our hypothesis that Morton layout is an attractive alternative to canonical layouts, that offers substantial spatial locality in any lexicographical traversal order and provides a clear performance

programming model. To test this hypothesis, this thesis takes the following approach:

1. We analyse theoretical characteristics of Morton layout.

2. We study the performance characteristics of the default Morton layout using a suite of micro-benchmarks.

3. We determine the optimisations that a compiler would need to perform to make Morton layout work.

We conclude that Morton layout is an attractive compromise to canonical layouts when combined with the optimisations we determine in our later chapters. Further, using Morton layout enables us to simplify the performance programming model for unsophisticated programmers, without relying on very powerful compiler technology.

1.2 Contributions

This thesis is based on four papers published in refereed conference proceedings and journals:

1. **Paper I**. “Is Morton layout competitive for large two-dimensional arrays?” [85].
   — In Proceedings of the Euro-Par 2002 Conference, Paderborn, Germany.

2. **Paper II**. “An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays” [81].
   — In Proceedings of the UK Performance Engineering Workshop 2003, University of Warwick, England.

3. **Paper III**. “Improving the Performance of Morton Layout by Array Alignment and Loop Unrolling: Reducing the Price of Naivety” [82]
   — In Proceedings of the Workshop on Languages and Compilers for Parallel Computing 2003, Texas, USA.

4. **Paper IV**. “Is Morton layout competitive for large two dimensional arrays, yet?” [83].
   — A journal paper due to appear in Concurrency and Computation: Practice and Experience, 2005.

The contributions of these papers and thus of this thesis, are as follows:

1. We perform an experimental analysis of the basic Morton scheme on different architectures using a suite of micro-benchmark kernels. We describe the performance characteristics of the basic Morton scheme, compared to canonical layouts. The architectures were chosen to have different and interesting cache characteristics and the benchmark kernels chosen were a small collection of scientific loops operating on dense, two-dimensional arrays.

2. We show that a simple table lookup scheme is remarkably effective for calculating the offset addresses of elements of Morton arrays.

3. We perform a careful analysis of the effects of alignment of the base address of Morton arrays. We show that aligning the base address of Morton arrays to the largest significant size in the memory hierarchy leads to better performance.

4. We analyse methods to unroll loops which operate over Morton arrays and show that unrolling Morton loops with appropriate alignment and strength reduction can lead to a significant improvement in performance.

5. From our experimental evaluation of simple optimisations we have suggested above, we establish the design requirements for a prototype compiler to support Morton layout in real scientific programs.

1.3 Structure

The remainder of this thesis is organised as follows:

1. Chapter 2 reviews locality-improving iteration space and data-layout transformations. The chapter restricts its attention to optimisations which improve locality of sequential code on uniprocessors.

2. Chapter 3 reviews data-layout transformations in detail. The chapter discusses multi-dimensional array layouts in detail including methods for calculating offset addresses of array elements.

3. Chapter 4 is based on Papers I , II and IV. This chapter analyses Morton layout in depth and evaluates its effectiveness as a compromise between row- and column-major layouts using a suite of micro-benchmarks.

4. Chapter 5 is based on Papers III and IV. The chapter evaluates the impact of alignment of the base address of Morton arrays on performance.

5. Chapter 6 is based on Paper III. The chapter describes a method for unrolling loops operating over Morton arrays with appropriate strength reduction. We show that the performance of Morton layout can be improved significantly.

6. Chapter 7 describes the design issues for implementing a prototype compiler to support Morton layout in real scientific programs. Following this, the chapter discusses future work and concludes the thesis.

1.4 Summary

This chapter has stated the approach, contributions and structure of the thesis. In the next two chapters, we review relevant research literature.

Chapter 2

Locality-Improving Transformations

In this chapter, we review existing work on locality-improving compiler transformations. We entirely focus on a uni-processor memory model for sequential applications.

2.1 Introduction

The performance gap between modern microprocessors and memory system architecture is one of the major factors which demands powerful compiler transformations. The aim of locality-improving compiler transformations is to bridge this gap. The main focus of such transformations are loops with array accesses [5]. In this thesis, our main interest is on kernels or programs which extensively use multi–dimensional arrays, which are, by default, stored lexicographically.

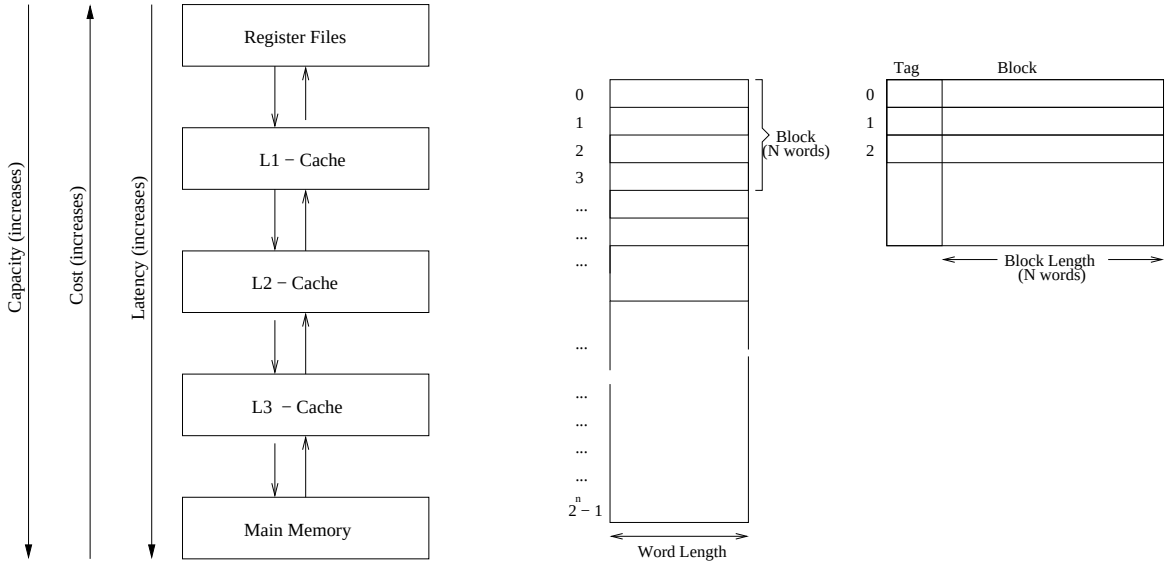
We refer to row-major and column-major as lexicographic orders, i.e. elements are arranged by the sort or linear order of the two indices (another term is "canonical"). When making a memory reference, for example referencing an array element, modern memory hierarchy designs fetch neighbouring elements into different levels of memory hierarchy. Any subsequent references to adjacent elements will be supplied by one of these levels. This improves spatial locality and the overall performance of a program. When array elements are accessed in the opposite direction of storage, the access is with non-unit stride and does not make use of data already available in the memory hierarchy and wastes memory bandwidth.

Locality optimisation techniques aim to improve temporal and spatial locality of accesses and they are often primarily concerned with loops. Transforming the iteration space of the surrounding loops and restructuring the underlying data layout of the arrays have been proven to be very effective techniques [21, 41, 62]. It is also possible to combine these two techniques to complement each other. This chapter introduces these techniques and pays particular attention to iteration space re-ordering techniques and combined transformations. We review important frameworks for transforming the iteration space, rather than individually covering each and every iteration space reordering technique. The next chapter covers the data layout transformations in detail.

2.2 Architectural Features — A Review

In this section, we briefly review the salient features of the memory hierarchy of modern computer systems.

Figure 2.1 illustrates the structure of the memory hierarchy and memory sub systems in a typical modern machine. As shown in Figure 2.1(a), the memory system is structured as a hierarchy of successively larger, slower and cheaper memory levels with the faster level being closer to the processor. These levels are connected through buses of varying bandwidth. Data is transferred from one level to another via these buses in fixed-sized blocks. The size of these blocks, the transfer time and the number of blocks that fit in a level are larger for levels farther from the processor. When a data item is required from the farthest memory level, it is transferred to the processor through these memory levels and retained for as long as possible. Subsequently, future requests for the same item may be satisfied by one of the memory levels closer to the processor. As a result, intermediate memory levels bridge the performance gap between the main memory and the processor by caching required data at appropriate levels.



(a) Structure of memory-hierarchy

(b) Cache/Main Memory Structure

Figure 2.1: **Structure of memory/memory hierarchy in modern architectures.** The figure shows a simplified structure of memory hierarchy and memory sub systems in modern architectures. Figure 2.1(a) shows the structure of the memory hierarchy present in modern systems; Figure 2.1(b) shows the structure of main/cache memory pair, in view of illustrating the placement of multiple words in a single cache block. In this figure, N words ($N = 2^k$ for some integer $k > 0$) are placed into a single cache line, starting from address zero. The address is often used as part of the mapping function to determine the cache line to occupy (see Figure 2.2 for more illustration on placement). Architectural and functional aspects of real systems are extremely complicated and their description can be found elsewhere [31]. (Diagram source: [31])

While individual registers hold a word, the structure of cache memory is slightly complicated. The cache memory is arranged as sequence of cache blocks and each of these blocks can hold multiple words — the block size. Since the capacity of cache memory is limited, a placement and a replacement policy are needed. The cache organisation describes the way that blocks are organised inside the cache and partly controls the placement — where a particular memory block goes inside the cache memory.

A simple organisation is a direct mapped cache. The mapping function uses part of the address

bits to determine the cache block that the memory block should belong to. In other words, part of the address bits are used to index the cache blocks and these bits are known as index bits. Because of this simple indexing scheme, each cache block is shared by multiple addresses in the main memory. This means that it is possible for a program to have access conflicts, where multiple locations compete for a placement inside the cache. In summary, each memory block has only one cache block inside the cache. There is no replacement policy in the direct mapped cache, as the candidate block for replacement is determined by the address and not by the usage.

The fully-associative cache permits a memory block to be placed anywhere in the cache and thus offers the best performance. However, the complexity in implementation limits the size of fully-associative caches.

A set-associative cache is a compromise between direct-mapped and a fully-associative cache, in terms of performance and cost. In an *n*-way set-associative cache, the set of cache blocks are divided into *n* ways and each set includes a cache block from each way. A memory block can be placed in one of the *n* cache blocks that make up a set. A segment of the address bits of a memory block is used to select the set. The actual placement within the set is then determined by the replacement policy. Figure 2.2 illustrates how a memory location is mapped into different types of caches with different associativities.

Upon a cache miss, the cache controller should find a block to be replaced by the newly fetched block. As mentioned above, in a direct mapped cache, there is no choice. However, in a set-associative cache, once the set is determined, there are many blocks to choose from for replacement. Often, the replacement strategy is based on usage: the least recently used item is evicted when required. This means that the controller chooses between constituent blocks of the indexed set.

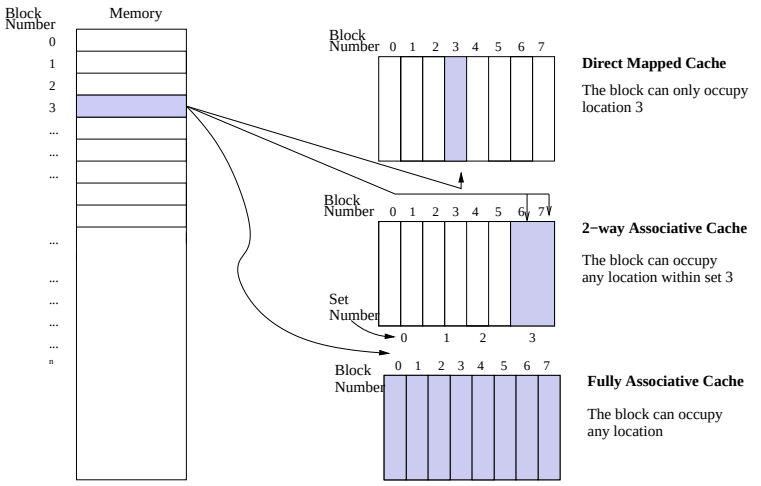
Use of cache memories is one of the most powerful ways of hiding and reducing the memory latency in memory intensive applications. Cache memories exploit the locality of data access. Locality of data accesses occurs in two different, but related ways: temporal and spatial locality. The property that the same data items tend to be used again in the near future can be defined as temporal locality and the property that data items which are adjacent to a particular data item tend to be accessed in the near future can be stated as spatial locality. As illustrated in Figure 2.1(b), when moving data items into cache, multiple words (which are adjacent in main memory address space) are moved together to amortise the cost of memory transfer. The spatial reuse is easily satisfied as cache memory contains adjacent memory items in the cache.

Real memory system hierarchy architecture is far more complicated than it is discussed here. Detailed analyses and treatment could be found in [31].

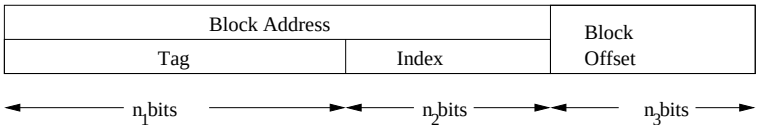
The key idea of iteration space transformations, discussed in the next section, is to restructure the loop nests in order to increase the locality of reference.

2.3 Iteration Space Transformations

The structure of a given loop nest determines the amount of data brought in between iterations. This in turn determines the likelihood of eviction of data items and therefore the locality of the program. The key idea behind iteration space transformations is to restructure the loop nests in order to increase the locality of reference. Iteration space transformations modify the iteration domain parameters and iteration schedule of each statement so that the required level of locality is achieved. For example,



(a) Associativity in Cache memories



(b) Addressing in cache memory

Figure 2.2: **Associativity and addressing in cache memory.** The figure illustrates the placement of a memory block in different associative caches and the general addressing scheme used by them. Figure 2.2(a) shows the placement of a memory block. The example assumes that memory block #3 has to be placed in a direct mapped, 2-way set associative and in a fully-associative caches, each with a total number of 8 blocks. In a direct mapped cache, the block has only one designated place — location 3 (**block number** *modulo* **number of blocks**). In a fully-associative cache, the block can be placed anywhere. In an *n-way* set associative cache, (2-*way*, here), the block can be placed anywhere within set 3 (**block number** *modulo* **number of sets**). The exact location within the set is determined by the replacement policy. Figure 2.2(b) illustrates how the virtual address bits are used for cache addressing. The first n_1 bits are used for tagging - for cache lookup, the next n_2 bits are used as index bits to select the set and the remaining n_3 bits are used as block offset bits, to address the desired data within a block. Fully-associative caches will not have the index bits. The exact values of n_1, n_2 and n_3 may vary by model for a given level of cache, but the principle remains the same. (Diagram source: [31])

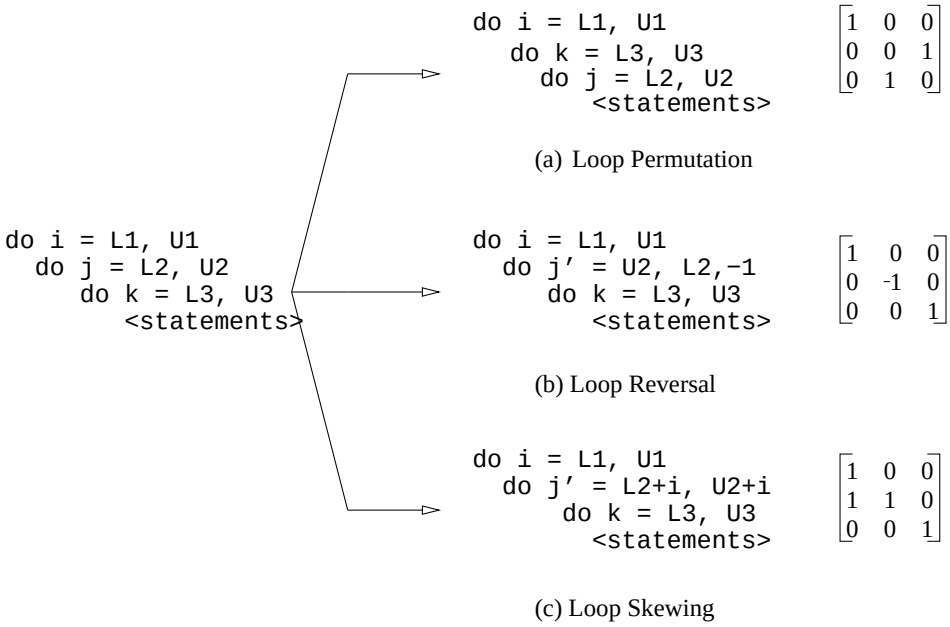


Figure 2.3: **Basic unimodular transformations and transformation matrices.** The diagram illustrates the basic unimodular transformations: loop interchanging, loop reversal and loop skewing with corresponding unimodular matrices.

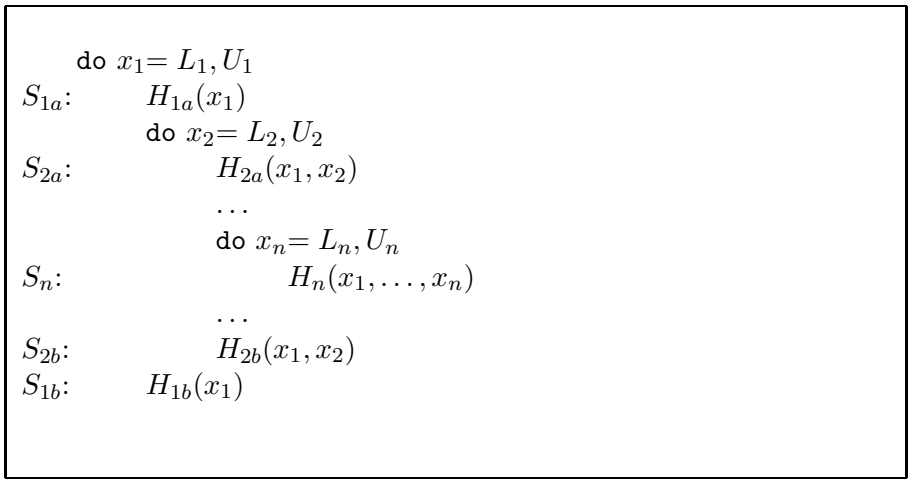
tiling, an iteration space transformation, may introduce new loops, may change the loop bounds, strides of innermost loops can be altered, and the exact sequence that points in the iteration space visited can be changed.

Other examples for iteration space transformation include loop interchange and loop reversal. A loop interchange transformation permutes a pair of loops in a loop nest and this can aid vectorisation and parallelisation or can transpose the access pattern of the enclosed arrays. Loop reversal, a valid transformation in the absence of loop carried dependencies, transforms a loop to run backward. This can enable further transformations, such as loop fusion. Loop tiling partitions the iteration space into uniform tiles of a given shape and size, such that computations are re-grouped. The re-grouping changes the execution order while improving the temporal locality. Computation on a single tile is completed before proceeding with another tile permitting maximal reuse of elements in a tile. In general, iteration space transformations change the order of execution, resulting in the order in which the data elements are accessed being changed, possibly improving locality of reference [15,21,45,46, 56,59,91].

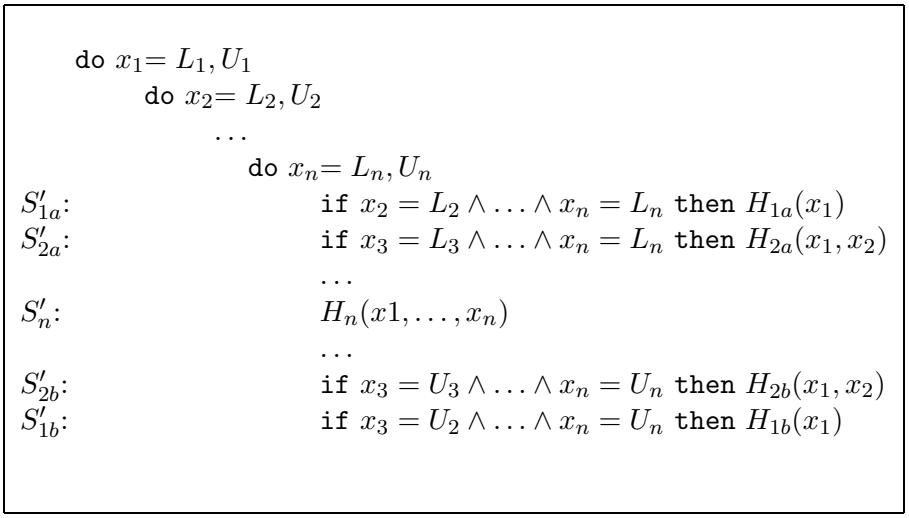
Unimodular transformations [6,91] are a unified framework that can describe any composition of the interchanging, skewing and reversal loop transformations. In this framework, a transformation is represented using a square integer matrix whose determinant is ± 1 (otherwise known as a unimodular matrix). Figure 2.3 illustrates the basic re-ordering transformations using this framework and corresponding matrices.

With the definition of unimodular matrices for basic transformations, any loop transformation can be expressed as a linear algebraic equation, consisting of a transformation matrix T and iteration vector I. The transformation matrix T restructures the original iteration space I to I' such that $I' = TI$, resulting in better locality.

Compiler infrastructures like Polaris [12] and SUIF [88] include unimodular transformations.



(a) Model of an imperfectly nested loop of depth *n*



(b) Loop nest in (a) is converted to a perfectly nested one using Lamport's Technique.

Figure 2.4: **Lamport's technique.** The figure illustrates Lamport's technique [52] being used to transform an imperfectly nested loop to a perfectly nested one. The idea is to move all imperfectly nested statements into the innermost loop and guard their execution using the IF statements.

These libraries accept as input a matrix T_x, representing the desired compound transformation, a pointer to the loop nest to be transformed, and the set of dependence vectors describing the dependencies between statements for the loop nest. They verify the legality of the transformation using the set of dependence vectors, and if the transformation is legal, proceed to transform both the dependence vectors and the code.

However, unimodular transformations are limited to perfectly nested loops and all statements in the loop body are changed in the same way. Imperfectly nested loops can be converted into perfectly nested ones using loop distribution [93] and Lamport's technique [52], where non-tightly nested statements are moved inside the innermost loop. In the absence of dependence cycles, loop distribution can be used to obtain several perfectly nested loops and unimodular transformations can be applied to them. If some of the statements in an imperfectly nested loop have dependency cycles, loop distribution may be illegal. In such situations, Lamport's technique may be used to move imperfectly

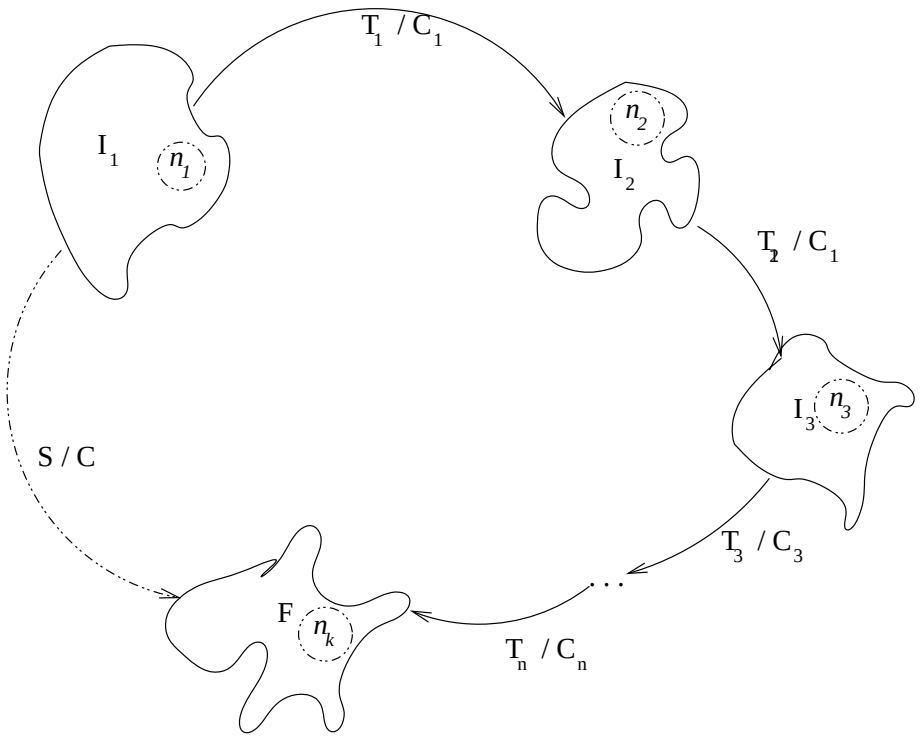


Figure 2.5: **Illustration of the Kelly and Pugh's unifying framework.** The figure shows how a desired transformation can be derived through composition of other transformations in Kelly and Pugh's Framework [44]. In this figure, the desired transformation is a schedule S which changes the iteration space I with dimension n_1 to an iteration space F with dimension n_k based on some optional condition C. T_i's are schedules with corresponding optional conditions C_i and $S = T_1 \circ T_2 \circ \ldots \circ T_n$ where the operator $\circ$ corresponds to composition.

nested statements to the innermost loop. Figure 2.4 illustrates how an imperfectly nested loop can be converted to a perfectly nested loop using these techniques.

Since all statements inside a loop nest are affected in the same way by the unimodular transformations, it is difficult to represent other forms of loop transformations such as loop fusion and loop distribution.

Kelly and Pugh propose a framework [44] to overcome this problem. They generalise a large set of reordering transformations into a unified polyhedral framework. Their framework transforms one iteration space $I = [i^1, \ldots, i^m]$ to another iteration space $F = [f^1, \ldots, f^n]$, based on an optional condition C. In their framework, a transformation is called a schedule and denoted by T. Multiple schedules can be composed to derive a desired schedule. In other words, the reordering transformations are based on schedules. A schedule T in [44] has the following general form:

$$T : [i^1, \ldots, i^m] \rightarrow [f^1, \ldots, f^n] | C \tag{2.1}$$

where $[i^1, \ldots, i^m]$ is the original iteration space, $[f^1, \ldots, f^n]$ is the new iteration space and C is an optional condition. The transformation takes place only if the condition C is satisfied. i^j's and f^j's are functions of iteration variables. Figure 2.5 shows the the basic idea behind the framework.

This framework relaxes most of the constraints in the original unimodular transformations framework. Most importantly, this framework permits:

- A separate schedule for each statement; this permits each statement inside a loop nest to be transformed independently of others.
- the dimensions of the mapped space and the original space to be different.
- non unimodular, but invertible schedules,
- functions of iteration variables (f^j's) to include division and modulo operations with the condition that the denominator is a known integer constant,
- schedules to be piecewise such that a single schedule is expressed as a union of other schedules.

These features enable a broader set of reordering transformations to be represented. The set of transformations represented by this framework includes any combination of the following:

- interchange
- tiling
- alignment
- reversal
- scaling
- coalescing
- skewing
- distribution
- statement reordering
- fusion
- interleaving
- index set splitting

Figure 2.6, adopted from [44] illustrates a tiling transformation for the LU decomposition.

Unimodular transformations are a special case, where:

- all statements are mapped with the same schedule,
- the dimensions of the mapped space and the original space are the same,
- the schedule is unimodular and invertible,
- f^j's are affine functions of the iteration variable,

The framework also provides a set of algorithms to manipulate (build, use and validate) schedules and to generate optimised codes for validated schedules. However, the framework does not provide support for analysing the suitability of a transformation for a given problem, and this is left to the user of the framework. Ideally, this framework would be a part of a larger system, for example an optimising compiler.

Although this framework demonstrates that most loop transformations can be modelled as geometric transformations of polyhedra, the framework is limited to iteration reordering only. Temam *et al.* [8] propose a similar polyhedral framework, covering a wide range of program transformations.

Kelly and Pugh's framework is sufficient to cover the basics of iteration space transformations. However, in reality, a sequence of transformations may be necessary to derive a desired transformation. For example, imperfectly nested loops can be converted into perfectly nested loops, using a sequence of transformations, including loop fusion and loop distribution. Then, iteration space transformations over this perfectly nested loop can be performed to improve performance. In [57], Li and Pingali extend the basic transformations to cover non-singular matrices by including loop scaling. Knijnenburg *et al.* [48] extend this framework by including support for loop alignment and statement-wise transformations.

Original code

```
    do 20 k = 1, n
      do 10 i = k+1, n
10      a(i,k) = a(i,k) / a(k,k)
        do 20 j = k+1, n
20        a(i,j) = a(i,j) - a(i,k) * a(k,j)
```

Schedule (for locality)

$T_{10} : [k, i \quad] \rightarrow [64((k-1) \ div \ 64) + 1, 64(i \ div \ 64), k, k, i]$
$T_{20} : [k, i, j] \rightarrow [64((k-1) \ div \ 64) + 1, 64(i \ div \ 64), j, k, i]$

Transformed code to satisfy schedules

```
    do 30 kB = 1, n-1, 64
      do 30 iB = kB-1, n, 64
        do 5 i = max (iB, k+1), min(iB+63,n)
5         a(i,kB) = a(i,kB) / a(kB,kB)
        do 20 t3 = kB+1, min(iB+62,n)
          do 10 k = kB, min(t3-1,kB+63)
            do 10 i = max(k+1,iB), min(iB+63,n)
10            a(i,t3) = a(i,t3) - a(i,k) * a(k,t3)
          do 20 i = max(iB,t3+1),min(iB+63,n)
20          if (t3<=kB+63) a(i,t3) = a(i,t3) / a(t3,t3)
        do 30 t3 = iB+63, n
          do 30 k = kB to min (iB+62,kB+63)
            do 30 i = max(k+1,iB), iB+63
30            a(i,t3) = a(i,t3) - a(i,k) * a(k,t3)
```

Sequence of elementary transformations required to achieve the results of schedules T_{10} and T_{20}

- strip mining
- index set splitting
- loop distribution
- imperfectly nested triangular loop interchange

Figure 2.6: **An example illustrating the application of Kelly and Pugh's framework.** The figure shows an example code (LU decomposition - non-pivoted version), the schedules T_{10} and T_{20} needed to tile the code (for statements at lines `10` and `20`), the resulting code (after applying the schedules) and the sequence of elementary transformations to achieve the same result as from schedules T_{10} and T_{20}. However, note that the tile size selection is not part of the framework.

When applying step by step transformations, an interesting problem is the effect of the order of these transformations. One way to deal with this problem is to rely on predictive heuristics [2, 5, 92]. Predictive heuristics try to determine *a priori* whether or not applying a particular optimisation will be beneficial. However, the complexity of this process limits the precision of predictive heuristics in practice. An alternative approach is to rely on iterative compilation [49], where a program is compiled multiple times with different optimisation configurations.

Some of these transformations require optimal selection of certain transformation parameters, which will not be part of the transformation itself. For example, tiling [14, 50, 51] requires selecting an optimal tile size. Choosing an inappropriate tile size may lead to disappointing performance. In the case of multi-level or hierarchical tiling [16], the size of each level in the memory hierarchy influences tile size selection. For example, a tile size optimised for first-level cache performance is likely to thrash the TLB (and vice versa) [60] and this can lead to a dismal performance [76]. There are different methods to solve this problem. Coleman and McKinley [22], Chame *et al.* [17] and

Knijnenburg *et al.* [49] describe different methods to automate the tile size selection when tiling loop nests. For example, the ATLAS project [87] automatically tunes several linear algebraic routines by empirically determining parameter values for various transformations.

In general, there are two classes of strategies used for optimisation: one strategy is empirical-driven optimisation, as in ATLAS [87], Fast Fourier Transform Compiler [26] and in iterative compilation frameworks [49]; and the other strategy is model-driven optimisation — a technique adopted by many compilers. Kamen Yotov *et al.* [96] compare these two optimisation strategies and they find that on some of the platforms for a restricted set of benchmarks, model-based optimisation strategies are more effective and are similar to empirical-driven optimisations.

In addition to the transformations covered by this framework, there are other transformations which can be very effective in performance optimisation. An interesting example is recursion. Ahmed and Pingali [1] describe a compiler technology for automatically converting loop nests with arrays to blocked-recursive versions. They consider perfectly nested loops and programs where array references are affine functions. With these considerations, partitioning the iteration space also results in partitioning the data space. The resulting code can be generated in blocked-recursive fashion. Yi *et al.* [95] also provide a similar compiler transformation for converting arbitrary loop nests into recursive code, to support multi-level memory hierarchies. They base their algorithm on ordinary dependence analysis and iteration space slicing [68].

2.3.1 Complementing Iteration Space Transformations

Iteration space transformations, while improving locality of reference may also lead to other performance problems. For example, cache memories have limited associativity; because of this, in a loop nest with array accesses, different blocks of data, either from the same array or from different arrays, can compete for the same cache line. In a loop nest, this can lead to repeated eviction of potentially useful data blocks. If this interference happens within the same array, it is known as self-interference and when it happens across different arrays it is known as cross-interference. While tiling improving locality, it may introduce self- or cross-interference. Some of these problems occurring during iteration space transformation can be overcome by complementing iteration space transformations with layout transformations. For instance, self-associativity conflicts due to tiling can be avoided by copying data accessed by a tile into contiguous memory locations [51]. If copying at runtime does not amortise the cost, it can be avoided by building the array in the desired layout or a selective copying could be done at compile time [80]. Alternatively, the array can be padded to avoid conflict misses as in [65, 71], often known as intra-array padding. Similarly, cross-interferences can be avoided by inter-array padding [71, 72].

Iteration space transformations can be very effective in improving locality of reference. Some transformations require optimal parameter selection, and some of them need to be complemented with layout techniques such as padding and copying in order to achieve better performance. However, when optimising loop nests, their dependence structure may prevent some of the transformations. In such cases, layout transformations can be used as an alternative technique or as a complementary technique to iteration space transformations. Layout transformations are briefly discussed in the next section and a detailed discussion is deferred for the next chapter.

2.4 Layout Transformations

Current programming languages support one of the two linear layouts: row-major and column-major layouts. Mapping function for the row-major (column-major) array layouts favour loops which accesses the array in row-major (column-major) order. When arrays are not accessed in their major order, neighbouring elements become distant in memory and this is known as the *dilation effect*. This means that accessing a row-major (column-major) array in column-major (row-major) order experiences the dilation effect. The dilation effect is one of the reasons for poor spatial locality when arrays are traversed in any order other than the array's major order. As discussed in the previous section, iteration space transformations may change the access pattern and may improve locality. However, iteration space transformations may not always be valid due to dependencies or due to complex control structures. An alternative strategy to this is to change the underlying array layouts. This technique is known as layout transformation or as array restructuring. Figure 2.7 illustrates an example layout restructuring which improves performance of a naively written code.

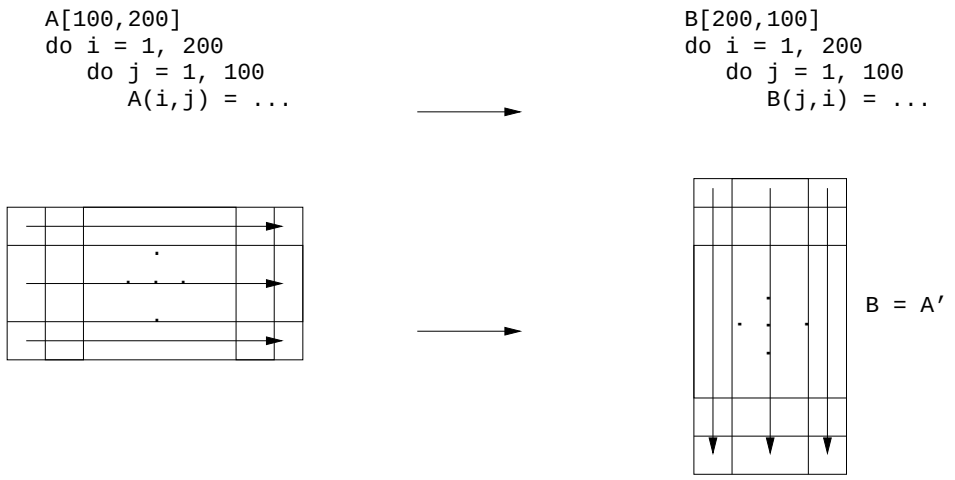


Figure 2.7: **Example layout restructuring.** The figure illustrates a simple layout restructuring optimisation. The code shown on the left traverses the column-major array A in row-major order. Instead of interchanging the loop, the underlying array layout is transposed with corresponding changes to the array indices. This results in improved spatial locality of the code.

The idea behind storage layout transformation is to choose or change a layout so that spatial locality is improved. When changing layouts, the mapping has to be one-to-one or invertible, so that it is unique. Array restructuring is attractive for several reasons: most importantly, the restructuring is not constrained by loop carried dependences or by the imperfectly nested loop structure and therefore layout restructuring is always valid. Further, given a loop nest with multiple array accesses, each array can be restructured independently of each other to maximise locality. However, loop restructuring affects only the loop nest to which the transformation has been applied whereas array restructuring affects all subsequent array accesses and can lead to differing performance in various parts of the code.

When switching between layouts, original array elements have to be mapped/de-mapped through corresponding mapping/de-mapping functions. Depending on the complexity of associated mapping functions and resulting spatial locality from target layouts, the choice for such layout switching can

be justified. Non-linear and blocked or recursive layouts discussed in Chapter 3 offer better spatial locality for blocked algorithms. Since blocking or tiling involves a non-linear mapping/de-mapping functions, these layout transformations are non-linear. Resulting mapping functions are fairly complicated compared to that of the linear layouts. Since compilers do not support layouts other than language-mandated layouts, the choice of layouts and possible optimisations are left to the programmer. We defer the discussion of storage layout optimisation, different storage layouts and their mapping functions until Chapter 3.

Layout transformation improves only spatial locality and does not affect temporal locality. This property leads to the idea of combining iteration space transformations with layout transformations such that both temporal and spatial locality can be improved. In the next section, we consider combined transformations — where both iteration space transformations and layout transformations are combined.

2.5 Combined Layout and Iteration Space Transformations

```
    do 10 j = 2, n
      do 10 i = 2, n
10      U[i,j] = V[j,i] + U[i-1,j] * V[j-1,i+1]

    do 20 j = 1, n
      do 20 i = 1, n
20      U[i,j-1] = V[i,j-1] + 1
```

Figure 2.8: **An example motivating combined transformations.** Due to conflicting accesses, both data and iteration space transformations need to be combined to improve the performance. The first loop nest could be optimised by transposing the layout of one of the arrays. The second loop nest has to be optimised taking the assumed layout into account. Although there is a choice of performing a layout transformation, performing an iteration space transformation in the second loop nest is optimal.

As illustrated in the previous section, combining iteration space and layout transformations enables us to exploit both temporal and spatial locality. Consider the example shown in Figure 2.8, assuming that the arrays are laid out in column-major order. In the first loop nest, array `U` is accessed in column-major order while the array `V` is accessed in row-major order. For the same array layout, these access patterns are conflicting. The dependence structure in the first loop prevents loop interchanging. However, in the first loop nest, the underlying data layout of one of the arrays (for example array V) can be transposed, as in Figure 2.7. This should improve locality of reference and therefore the performance. However, due to this transformation, array accesses in the second loop nest now become conflicting. The second loop nest has to be optimised taking the underlying layout of the array V into account. A layout transformation can be performed again in the second loop nest, transposing the array U. Alternatively, the loop nest can be interchanged, taking the array layout of V into account. However, when considering the overheads associated with dynamically switching between layouts and copying, performing an iteration space transformation is optimal.

This example demonstrates the need to combine both iteration space and data layout transformations. When all or some of the iteration space transformations are prevented by dependency structure of a loop nest, it is possible to use layout transformation techniques as an alternative to iteration space transformations or to create additional opportunities. Further, in the presence of multiple layouts in a loop nest, an iteration space transformation in favour of one layout may affect another one. Combining layout transformation with the iteration space transformation provides a means to overcome this problem.

Li and Cierniak [21] discuss techniques to unify control and data layout transformations. With the standard definition of access matrix, they use stride vectors instead of re-use vectors. Elements of a stride vector carries information about data re-use specific to loop nests. For example, if an element is 0, the corresponding loop nest has temporal locality and if an element is less than the size of the cache-line, then the loop carries spatial locality. Given a mapping vector m (for column-major layout $m = m_c = (1 \quad n)^T$ and for row-major layout $m = m_r = (n \quad 1)^T$ and access matrix A, the stride vector is defined as $v_x = A^T m_x$. For different layouts, different stride vectors could be obtained. Now combining this with the non-singular iteration space transformation requires a transformation matrix T with a mapping vector m such that a stride vector v is obtained satisfying the constraint:

$$T^T v = A^T m \tag{2.2}$$

The solution for the transformation matrix finds legal loop transformations and finding solutions for the mapping vector m means finding legal data layout transformations. An optimal solution for Equation(2.2) could be found by solving the equation heuristically with the legality constraints for mapping and transformations. Since the potential search space is very large, they limit their search space by introducing more constraints and assumptions. This is only valid for single array reference and the case of multiple array references can be addressed similarly. However the search space in the case of multiple array references is much larger than for the single array reference.

Kandemir *et al.* present a cache locality optimisation algorithm [43] which can optimise a loop nest for locality in the presence of multiple layouts for arrays within a loop nest. Their transformation techniques improve the spatial locality with respect to the innermost loop. They use a single linear algebraic framework to represent both layout and loop transformations. Then, for a given loop nest, they compute a non-singular loop transformation matrix to exploit the data locality in the innermost loop. To simplify the problem, they consider the case where arrays have different but non-dynamic layouts, meaning that array layouts do not change between loops.

2.6 Summary

In this chapter, we have reviewed closely-related compiler techniques for improving locality. Although iteration space transformations can improve locality, their application is often limited by the capability of the compiler in inferring dependencies. Data layout transformations can be used as a complementary technique or as an alternative to iteration space transformations. We discuss layout transformations in detail in the following chapter.

Chapter 3

Storage Layout Optimisation

In the previous chapter, we considered locality improving transformations, and iteration space transformations in particular. In this chapter, we introduce linear and non-linear layouts, layout transformations, mapping functions and their spatial locality. In particular, we pay special attention to recursive non-linear layouts such as Morton layout, which is the subject of this thesis.

3.1 Introduction

Iteration space reordering may not always be valid due to dependencies and/or due to the complexity of control structures. Further, iteration space transformation may need to be complemented by layout transformations techniques. Unlike iteration space re-ordering, data-layout restructuring is always valid as long as the mapping is *invertible*. The purpose of data-layout restructuring is to change the underlying data-layout such that spatial locality is improved. Layout transformations include transposition, stride re-ordering [42], intra- and inter-array padding, array merging [53], copying and recursive non-linear array layouts. Copying, intra- and inter-array padding and stride re-ordering are often used to complement iteration space transformation techniques.

When a layout transformation is performed, the resulting layout is chosen to match the dominant access pattern of the code. Applications may also demand different layouts at different phases of execution. In such cases, it may be difficult to decide on a single optimum layout. If the layout selection is static, performance may vary across different parts of the code. This problem can be overcome by selecting layouts dynamically: permitting the layouts to change at different parts of the code. However, when using dynamic layouts, the overhead of switching layouts is paid at runtime. Further, it may not be possible to determine an optimal layout at compile time for an array (for a given loop nest) as the information available about the access in a given region of code may be insufficient. Although this information will become available at runtime, the cost of determining an optimum layout and data copying may outweigh any performance benefit. On the other hand, if dynamic layouts are permitted, it is possible to restructure each array independently for maximum locality in a loop nest.

Locality and parallelism are also coupled together. Although our work is restricted to the uniprocessor memory model, it is possible to extend it to parallel machines and applications, where the memory model is different. In parallel applications, non-local data accesses resulting in communication can greatly affect application performance. Parallel data placement [9], array-alignment in parallel programs and computers [18, 94] and iteration space transformations such as tiling are used

to minimise remote data accesses. We again restrict our attention to the uni-processor memory model and sequential applications in the following sections and chapters.

Existing work on data or storage layout transformations is motivated by seeking maximum spatial locality for each array in a loop nest. In contrast, our motivation is to analyse and evaluate how non-linear layouts can be a compromise storage layout.

3.2 Linear layouts

Programming languages support one of the two lexicographical layouts: row- and column-major layouts. For a two-dimensional $M \times N$ array, let $\mathcal{S}_x^{(M,N)}(i,j)$ denote mapping function $\mathcal{S}$ for a layout x for an element at (i,j). The mapping functions for the lexicographical layouts, row-major (rm) and column-major(cm), are given by Equations (3.1) and (3.2).

$$\mathcal{S}_{rm}^{(M,N)}(i,j) = N \times i + j \tag{3.1}$$

$$\mathcal{S}_{cm}^{(M,N)}(i,j) = i + M \times j \tag{3.2}$$

These linear layouts have the dilation effect - meaning that elements in the opposite order of their major traversal order becomes distant in memory. That is, for row-major (column-major) array, elements along a column (row) becomes distant in memory — stride equal to the row-length (column-length). This dilation effect leads to poor spatial locality upon accessing elements in the wrong order.

In general, a k-dimensional array can be stored in $k!$ linear forms and each of them corresponds to a nested traversal of the axes in some pre-determined order. When the access pattern matches the fastest changing dimension of the layout, spatial locality is exploited. A linear storage layout transformation technique can opt for one of these possible layouts if that improves spatial locality.

The simplest linear storage layout transformation is a permutation of array dimensions [21, 40]. For a two-dimensional array, this would be a transpose — *row-major* array is transformed to *column-major* array and vice versa.

A linear layout can be restructured to another linear layout by means of a linear transformation of index vectors. Consider an m-dimensional array U being accessed in a loop nest of depth n. Assuming that all array indices are *affine* functions of loop variables, the index for the kth dimension ($k < m$) can be expressed as $v'_k = a_{k0}i_0 + a_{k1}i_1 + \ldots + a_{kn}i_m + o_k$, where o_k is a constant offset and a_{kj}'s are integer coefficients and i_j's are loop variables. It is possible to express an index vector for all indices by :

$$\mathbf{V}_U = A_U\mathbf{I} + \mathbf{o}_U \tag{3.3}$$

where A_U is an $m \times n$ *access* matrix, $\mathbf{I}$ is the iteration vector, $\mathbf{V}_U$ is the index vector of the element and $\mathbf{o}_U$ is an m dimensional constant offset vector, specific to this access. Each row of A_U and $\mathbf{o}_U$ correspond to one dimension of the array index vector. For example, a regular, nested loop of depth n can be represented by Equation 3.4.

$$\mathbf{V}_U = \begin{pmatrix} a_{00} & a_{01} & \ldots & a_{0n} \\ a_{10} & a_{11} & \ldots & a_{1n} \\ \cdot & \cdot & \cdot & \cdot \\ a_{m0} & a_{m1} & \ldots & a_{mn} \end{pmatrix} \begin{pmatrix} i_0 \\ i_1 \\ \cdot \\ i_m \end{pmatrix} + \begin{pmatrix} o_0 \\ o_1 \\ \cdot \\ o_m \end{pmatrix} \tag{3.4}$$

Leung and Zahorjan [54] discuss a technique whereby they transform the original array index space $\mathbf{V}_U$ to $\mathbf{V'}_U$ so that new accesses exhibit more spatial locality. They achieve this through an $m \times m$ uni-modular index transformation matrix T_U such that

$$\begin{aligned} \mathbf{V'}_U &= T_U \mathbf{V}_U \\ &= (T_U A_U)\mathbf{I} + (T\,\mathbf{o}_U) \end{aligned} \tag{3.5}$$

The mapping is *invertible*, meaning that it can de-mapped. Now the transformed index vector $\mathbf{V'}_U$ is used to index the restructured array. This transformation matrix is chosen for each array in a loop nest based on the access pattern that the array exhibits. So the selection of an optimum layout for an array is performed in two steps: determining the index transformation matrix T and then refining it for better locality using uni-modular transformations.

Kandemir *et al.* [41] suggest a technique to select a layout to match the execution order, at compile time. They use the concept of reuse vectors [56, 91]. The idea is to obtain these reuse vectors representing the direction in which the reuse occurs and then transform loop nests such that these reuse vectors have some desired properties. Linear-algebraic techniques are used to derive the locality information from these re-use vectors and used to guide the selection process of a layout for an array. With the definition of spatial locality, two different iterations I and J access elements residing in the same row (or column) when the condition $A_{U_s} I = A_{U_s} J$ is satisfied. Where A_{U_s} is the access matrix for U with the first row deleted [91]. Thus $A_{U_s}(J-I) = 0 \implies A_{U_s} r_{U_s} = 0$, where $r_{U_s} = J - I$ is a spatial reuse vector associated to an instance of array U. An important attribute of a spatial reuse vector is its height which is defined as the number of dimensions from the first non-zero entry to the last entry. By transformations, it is possible to introduce more leading zeros in the reuse vector and this is known as height reduction [56]. Kandemir *et al.* uses the spatial and temporal reuse vectors and their height information to form locality information. Then this locality information is used to reproduce the effects of iteration space transformation by using linear layout transformations, if possible. If there are multiple instances of the same array, then a spatial reuse matrix can be constructed instead of a vector. The same theory can be used to derive the temporal reuse vector(matrix), which is a special case of spatial reuse. Although the choice of layouts for spatial reuse is restricted to canonical layouts, this framework selects optimum array layout for each array individually.

Another domain, where data-layout transformations are used is in the synthesis of memory architectures, where efficient mapping of array elements in storage devices of embedded systems is crucial for optimal power/space consumption. Panda *et al.* [64] try to reduce the system power requirements of memory subsystems by minimising the transition count on the memory address bus when array elements are accessed. In order to achieve this, they exploit regularity and spatial locality in the memory accesses and determine the mapping of behavioural array references to physical memory locations. Hettiarachi *et al.* [32], map spatially and temporally local array elements to the same row, so that the transition count would be low for row- and column-address strobes for the memory subsystem. These power reduction optimisations are essentially data space transformations to improve locality of reference.

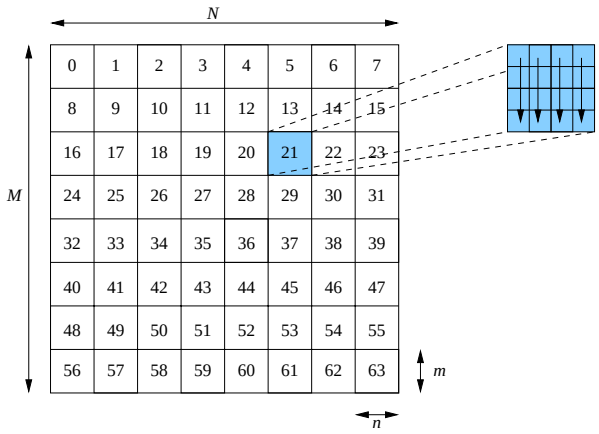


Figure 3.1: **Blocked row-major ("4D") layout**. The diagram illustrates the 4D layout where an array is arranged into tiles. Tiles and elements within each tile are arranged in one of the lexicographical orders, independent of each other. If the original array size is not a multiple of tile size, it can be padded or unequal tile sizes can be chosen.

3.3 Recursive and Non-linear Layouts

Conventional canonical layouts assume a flat, linear model of memory when mapping array elements to memory. However, modern machines have multiple levels of memory hierarchy with varying performance characteristics in terms of access time, capacity and cost (see Section 2.2). This mismatch between the actual and represented models accounts for poor spatial locality and poor performance in many applications. Non-linear and recursive layouts have hierarchical storage model. This matches well with the hierarchical memory model of modern machines.

A simple non-linear memory layout scheme is studied by Chatterjee *et al.* in [19], known as 4D layout. The idea is to divide the original array into lexicographically arranged tiles of $t_R \times t_C$ (t_R is the number of tiles across a column, and t_C is the number of tiles across a row) and then lay out elements inside these tiles in canonical order. Layout of tiles and of elements inside a tile can be independent of each other. Chatterjee *et al.* argue that such a freedom in the choice of layouts can help overcoming the dilation effect in canonical layouts in addition to closely mapping an array layout to the machine's memory model. Figure 3.1 shows this simple blocking scheme.

If a blocked row-major mapping function $\mathcal{S}_{brm}^{(M,N)}(i,j)$ is defined for an element at (i,j), such that:

$$\begin{aligned} \mathcal{S}_{brm}^{(m,n)}(i,j) &= (t_R \times t_C) \times \mathcal{S}_{rm}^{(M/t_R,N/t_C)}(i/t_R, j/t_R) \\ &\quad + \mathcal{S}_{rm}^{(t_R,t_C)}(i\%t_R, j\%t_C) \end{aligned} \tag{3.6}$$

elements could be mapped and de-mapped from two-dimensional space to the one-dimensional space and vice versa. We further illustrate this layout with $t_R = 4, t_C = 4$ with ordering of tiles and elements within tiles to be row-major in Figure 3.2. If we consider 16-word cache blocks, each block holds a $t_R \times t_C = 16$-word sub-array. In row-major traversal, the four iterations $(0,0)$, $(0,1)$, $(0,2)$ and $(0,3)$ access locations on the same block. The remaining 12 locations on this block are not accessed until later iterations of the outer loop. Thus, for a large array, the expected cache hit rate is 75%, since each block has to be loaded four times to satisfy 16 accesses. The same cache hit rate results with column-major traversal, i.e. when the loop structure is "`do i...do j`" rather than the "`do j...do`

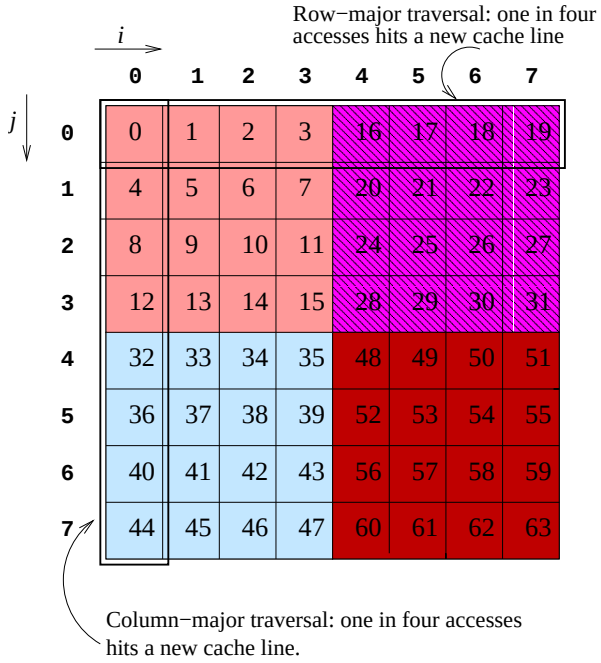


Figure 3.2: **Blocked row-major ("4D") layout**. $\mathcal{S}_{\mathbf{brm}}^{(8,8)}(i,j)$ with block-size parameters $t_R = t_C = 4$. The diagram illustrates that with 16-word cache lines, illustrated by different shadings, the cache hit rate is 75% whether the array is traversed in row-major or column-major order.

`i`" loop of row-major traversal.

Modern computer systems rely on a Translation Look-aside Buffer (TLB) to cache address translations: a typical 64-entry data TLB with 8KB pages has an effective span of $64 \times 8 = 512KB$. Unfortunately, as illustrated in Figure 3.3, if a blocked row-major array is traversed in column-major order, only one sub-array per page is usable. Thus, we find that the blocked row-major layout is still biased towards row-major traversal. We can overcome this by applying the blocking again, recursively. Thus, each 8KB page (1024 doubles) would hold a 16×16 array of 2×2-element sub-arrays. Further, modern systems often have a deep memory hierarchy, with block size, capacity and access time increasing geometrically with depth [4]. Therefore applying blocking recursively should also help matching an array layout with the memory model.

The tile sizes t_R and t_C can be made architecture-dependent and can be chosen empirically. The original array may need to be padded to have equal tile sizes, involving a space overhead. Alternatively, unequal tile sizes may be selected for the last row and column of the matrix. With the blocked 4D layout, the address calculation for an element requires two levels of computation: one for the tile containing that element and then the offset within that tile. This address calculation problem is easier in their work [19], because all their codes are tiled or shackled which greatly simplifies the address computation. Further, by tiling and shackling, the performance is less sensitive to small changes in tile size and problem size, which can result in cache associativity conflicts with conventional layouts.

Gustavson *et al.* [29] apply blocked layouts over blocked algorithms. Their data structure, which is known as *new data structure* (*NDS*) or as *blocked hybrid format (BHF)*, is very similar to the 4D layout described above in [19], except the fact that algorithms in the former are carefully designed to exclude the padded elements from computation. The new data structure is also called square blocked packed format for square arrays. Gustavson *et al.* [30] extend this to triangular arrays. With the symmetric

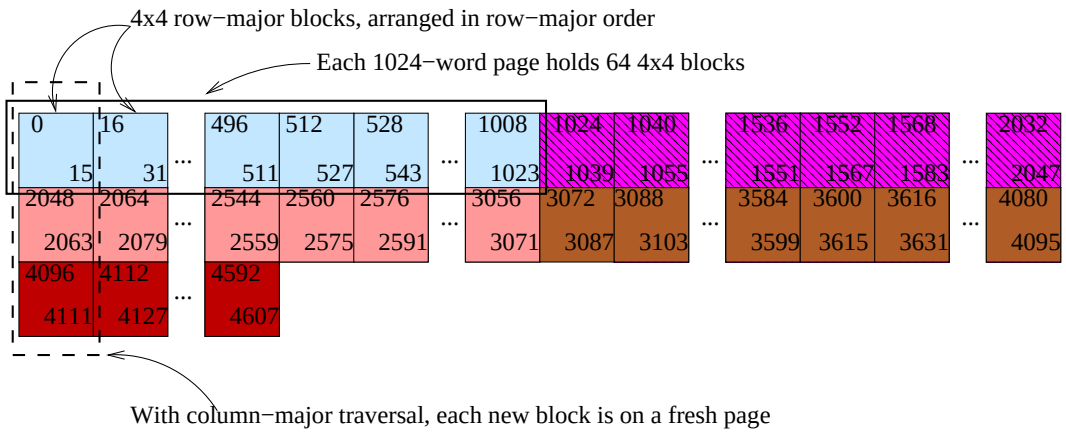


Figure 3.3: **Blocked row-major layout for large array.** If a large blocked row-major array is traversed in column-major order, only one sub-array per page is usable. The diagram shows an array with rows of 2048 doubles, using the blocked row-major layout with 4×4 blocks. Each 8KB page holds 1024 doubles, 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 $1 - 1/4 = 75\%$).

triangular matrices, the square blocked packed format becomes *square blocked lower packed* (SBLP) or *square blocked upper packed* (SBUP) formats, and the storage requirement is halved. They report promising results for selected linear algebraic algorithms after combining recursion with these layouts.

The use of Quad- or Oct-trees is known in parallel computing for improving the load balancing and locality [7, 34, 67]. These data structures are used in information theory for bandwidth reduction [11], in graphics and database applications [39]. Quad- and Oct-trees can be generalised through space-filling curves (or plane-filling curves) described in [33, 66]. Space-filling curves produce similar line segments with varying scales and with varying orientations (by rotating these line segments) over a given region. These space-filling curves are recursive in nature and can be described by a family of non-linear functions.

Multidimensional arrays can be thought of formed up of planes, and elements on each of these planes being laid out using a mapping function. Since space-filling curves can fill a plane without holes, it is possible that elements of an array to be laid along one of such plane-filling curves. This results in recursive layouts for arrays. Space-filling curve used to lay out the elements of an array is the required mapping function for the chosen layout.

Frens and Wise [90] advocated Morton layout for multidimensional arrays, based on these space-filling curves. The idea with Morton layout is to divide the original matrix into four different quadrants (NW, NE, SW, SE) and arrange these quadrants in the order of (NW, NE, SW, SE) quadrants in memory, then recursively apply the same technique for each tile until no further tiles are left. As in the blocked 4D layout, the address computation cost for Morton layout is high if elements are accessed at random. Frens and Wise found that it is hard to overcome the addressing costs. Again, as with the blocked 4D layout, the iteration space can either be tiled or recursively formulated to minimise address computation cost. If the original size of the matrix is not a power-of-two then the matrix has to be padded to the next integer power-of-two size.

The elements inside the smallest Morton block (2×2 tile) can be arranged in many different ways.

Figure 3.4 illustrates different Morton mappings. Each of these Morton layouts have different orientations and different addressing costs. We discuss four different space-filling curves in this chapter, all derived from Hilbert curve. Lebesgue's layout or Z-Morton layout, U-Morton and X-Morton layouts have single orientation, i.e. line segments are never rotated. The mnemonics represent the English alphabet that their ordering pattern resembles. G-Morton layout has two orientations - line segments rotated by 180 degrees.

Morton layout requires array size to be a power-of-two size in each dimension. If this is not the case, the array has to be padded to the next power-of-two, involving a space overhead as high as 75%. Valsalam *et al.* [86] studied an alternative approach, where they use Morton layout with an arbitrary size matrix. The idea behind their work is to identify non-square tile sizes, such that padding is avoided or minimised.

Drakenberg *et al.* [23] present a semi-hierarchical array layout (called HAT) to enable precise analysis of cache behaviour at compile time. The semi-hierarchical layout they propose is a hybrid of Morton and lexicographic layouts. In their semi-hierarchical layout they arrange large tiles in lexicographical layout and elements inside the tiles are arranged in Morton order. They choose each tile size to be a multiple of page size to improve TLB performance. Their compile time cache analysis is based on the cache miss equations framework proposed by Ghosh *et al.* [28].

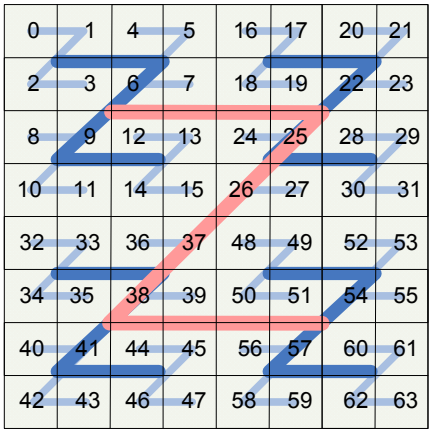
In all these recursive layouts, although padding adds to the space overhead, they do not affect the performance as they are never fetched to smaller levels of memory hierarchy. However, padding may not be acceptable for systems which do not have virtual memory. From the computational point of view, either the computation could be done over these padded area (after initialising them to zero) or loop boundaries can be adjusted to avoid any computation over this padded area. For example, Chatterjee and Gustavson *et al.* chose the latter method, while Frens and Wise chose the first one.

3.4 Mapping Functions for Recursive Array Layouts

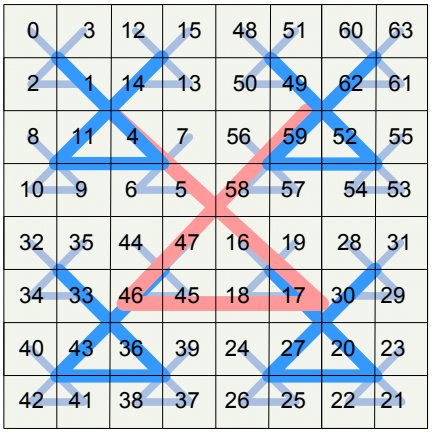
All recursive layout functions share a common operational interpretation inherited from Peano's and Hilbert's interpretation of space-filling curves [33, 66, 75]: divide the original matrix into four quadrants and lay out these sub-matrices or quadrants in memory using a layout function, say $\mathcal{S}_T^{M/P,N/Q}$. Then repeat this recursively using the same layout function, until no further division is possible.

We use the following notations and assumptions in our analyses:

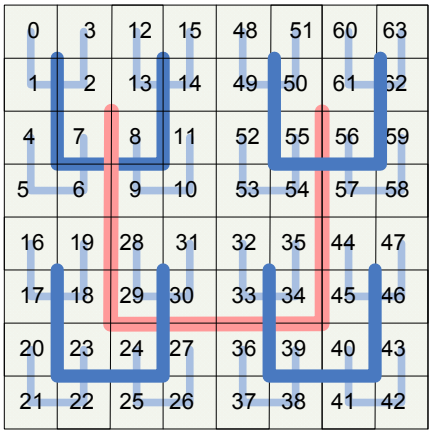
- We assume that all arrays are indexed from base zero.
- For an $M \times N$ array with a layout l, the offset of an element at (i, j) is given by $\mathcal{S}_l^{(M,N)}$,
- For a non-negative integer i, let $\mathcal{B}(i)$ be the binary representation, i.e. $\mathcal{B}(i) = i_{n-1}i_{n-2}\ldots i_0$.
- For a bit string s, let $\mathcal{B}^{-1}(s)$ be the non-negative integer i such that $\mathcal{B}(i) = s$,
- For a non-negative integer i, let $\mathcal{G}(i)$ be the Gray encoded representation [58],
- For a non-negative integer i, let $\mathcal{D}_0(i)$ be the left dilated (or odd dilated) representation, i.e. $\mathcal{B}(\mathcal{D}_0(i)) = 0i_{n-1}0i_{n-2}\ldots 0i_20i_10i_0$.
- For a non-negative integer i, let $\mathcal{D}_1(i)$ be the right dilated (or even dilated) representation, i.e. $\mathcal{B}(\mathcal{D}_1(i)) = i_{n-1}0i_{n-2}0\ldots i_20i_10i_00$.



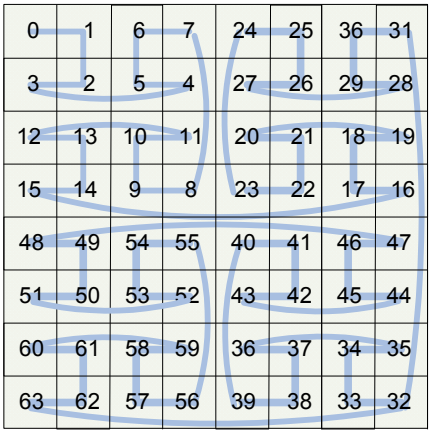
(a) Z-Morton



(b) X-Morton



(c) U-Morton



(d) G-Morton

Figure 3.4: **Morton layouts**. The diagram illustrates a family of Morton layouts for an 8×8 array. The array is divided into four different quadrants of $\{$NW, NE, SW and SE$\}$. Quadrants are stored as consecutive tiles, in the order of $\{$NW, NE, SW, and SE$\}$ for Z-Morton (Figure 3.4(a)), $\{$NW, SW, SE and NE$\}$ for U-Morton (Figure 3.4(c)) and $\{$NW, SE, SW and NE$\}$ for X-Morton (Figure 3.4(b)). For these three layouts, it could be noticed that the orientation of the placement does not vary from block to block. However, for G-Morton, the orientation alternates between the order of $\{$NW, NE, SE, and SW$\}$ and $\{$SE, SW, NW, and NE$\}$ for every two blocks. Thus G-Morton has the orientation of 2. This technique is applied recursively for each of the quadrant. The addressing function for these layouts is tabulated in Table 3.1.

- the exclusive OR operator is $\oplus$
- the bitwise OR operator is $|$
- the bitwise concatenation operator is $||$

Canonical Layouts

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

$$\begin{aligned} \mathcal{B}(i) &= i_{n-1} i_{n-2} \ldots i_3 i_2 i_1 i_0 \qquad \text{and} \\ \mathcal{B}(j) &= j_{m-1} j_{m-2} \cdots j_3 j_2 j_1 j_0 \end{aligned} \tag{3.7}$$

respectively. For simplicity, we restrict our analysis to square arrays (where $M = N$), but this constraint is relaxed in the next section. Now the lexicographic mappings can be expressed as bit-concatenation:

$$\begin{aligned} \mathcal{S}_{rm}^{(M,N)}(i,j) &= N \times i + j = \mathcal{B}(i) \| \mathcal{B}(j) \\ &= i_{n-1} i_{n-2} \ldots i_3 i_2 i_1 i_0 j_{n-1} j_{n-2} \cdots j_3 j_2 j_1 j_0 \qquad (3.8) \\ \mathcal{S}_{cm}^{(M,N)}(i,j) &= i + M \times j = \mathcal{B}(j) \| \mathcal{B}(i) \\ &= j_{n-1} j_{n-2} \cdots j_3 j_2 j_1 j_0 i_{n-1} i_{n-2} \ldots i_3 i_2 i_1 i_0 \qquad (3.9) \end{aligned}$$

Recursive Layouts

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

$$\begin{aligned} \mathcal{S}_{brm}^{(M,N)}(i,j) &= (P \times Q) \times \mathcal{S}_{cm}^{(M/P,N/Q)}(i,j) + \mathcal{S}_{rm}^{(P,Q)}(i\%P, j\%Q) \\ &= \mathcal{B}(i)_{(n-1)\ldots p} \| \mathcal{B}(j)_{(m-1)\ldots q} \| \mathcal{B}(i)_{(p-1)\ldots 0} \| \mathcal{B}(j)_{(q-1)\ldots 0} \end{aligned} \tag{3.10}$$

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

$$\begin{aligned} \mathcal{S}_{mz}^{(N,M)}(i,j) &= i_{n-1} j_{n-1} i_{n-2} j_{n-2} \ldots i_3 j_3 i_2 j_2 i_1 j_1 i_0 j_0 \qquad (3.11) \\ &= \mathcal{D}_1(i) | \mathcal{D}_0(j) \qquad (3.12) \end{aligned}$$

This mapping is called Z-Morton layout and the operation is known as *bit-interleaving*. This is illustrated in Figure 3.5.

Bit-interleaving is too complex to execute at every loop iteration. Wise *et al.* [89] explore an intriguing alternative: with the definitions of $\mathcal{D}_0$ and $\mathcal{D}_1$, and with the notion of loop control variable i being denoted as a dilated integer: increment of a loop control variable at each loop iteration is fairly straightforward. Let "&" denote bitwise-and. Then:

$$\begin{aligned} \mathcal{D}_0(i+1) &= ((\mathcal{D}_0(i) \mid \text{Ones}_0) + 1) \ \& \ \text{Ones}_1 \qquad (3.13) \\ \mathcal{D}_1(i+1) &= ((\mathcal{D}_1(i) \mid \text{Ones}_1) + 1) \ \& \ \text{Ones}_0 \qquad (3.14) \end{aligned}$$

where

$$\begin{aligned} \mathcal{B}(\text{Ones}_0) &= 10101 \ldots 01010 \qquad (3.15) \\ \mathcal{B}(\text{Ones}_1) &= 01010 \ldots 10101 \quad . \qquad (3.16) \end{aligned}$$

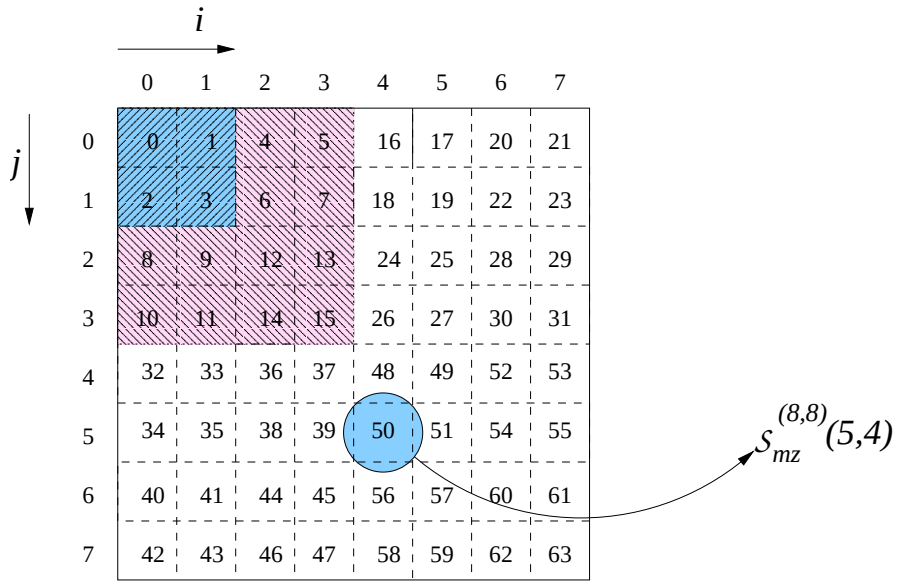


Figure 3.5: **Morton storage layout for an 8×8 array.** Location of element $A[5,4]$ is calculated by interleaving "dilated" representations of 5 and 4 bitwise: $\mathcal{D}_1(5) = 100010_2$, $\mathcal{D}_0(4) = 010000_2$. $\mathcal{S}_{mz}(5,4) = \mathcal{D}_1(5) \mid \mathcal{D}_0(4) = 110010_2 = 50_{10}$. A 4-word cache block holds a 2×2 sub-array; a 16-word cache block holds a 4×4 sub-array. 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.

Layout	*Layout function*	*Mapping function*	*Number of orientations*
Z-Morton	$\mathcal{S}_{mz}^{(N,M)}(i,j)$	$\mathcal{D}_1(i) \mid \mathcal{D}_0(j)$	1
U-Morton	$\mathcal{S}_{mu}^{(N,M)}(i,j)$	$\mathcal{D}_1(j) \mid \mathcal{D}_0(i \oplus j)$	1
X-Morton	$\mathcal{S}_{mx}^{(N,M)}(i,j)$	$\mathcal{D}_1(i \oplus j) \mid \mathcal{D}_0(j)$	1
G-Morton	$\mathcal{S}_{mg}^{(N,M)}(i,j)$	$\mathcal{G}^{-1}(\mathcal{D}_1(\mathcal{G}(i)) \mid \mathcal{D}_0(\mathcal{G}(j))$	2

Table 3.1: **Mapping functions and number of orientations of recursive layouts summarised from [11, 20].** The Table tabulates the mapping functions and their orientations for various recursive, non-linear layouts. The operators $\mathcal{D}_*(.)$ and $\mathcal{G}(.)$ follow the definitions stated at the beginning of Section 3.4.

We call this approach the dilated arithmetic scheme. However, implementing this dilated arithmetic scheme in all for-loops is a cumbersome process. Alternatively, we could pre-calculate the dilated values ($\mathcal{D}_O(i)$ and $\mathcal{D}_1(j)$) in lookup tables and we could use these lookup tables to find the corresponding offset address. Exact number of lookup tables dependents on the number of array dimensions and not on the number of loop control variables. We explore this further in the next chapter.

Mapping functions for the other layouts in the Morton family can be obtained similarly. Table 3.1 summarises the mapping functions and orientations for the layouts in the Morton family. H-Morton (or Hilbert Curve) has four orientations with line segments rotated by 90,180 and 270 degrees. The mapping function for the Hilbert layout is computationally more complex, which is informally described by Bially [11]. In summary, in the recursive layouts we discussed, we observe the following:

- The fact that only two of the four neighbours of element at (i, j) can be adjacent to each other in the mapped space, leads us to conclude that all layouts should experience the dilation effect. In recursive layouts, the dilation effect varies with the level of blocking involved and with the number of orientations.

- The instruction sets of current architectures do not include support for dilated arithmetic, Gray-coding or for bit dilation. Assuming that array indices are pre-dilated (or Gray-coded) and either table lookup or direct arithmetic is used to calculate the offset address, then the instruction count for each of layout function leads us to conclude that $\mathcal{S}_{mz}$ is computationally cheaper than other mapping functions.

- We have not included any recursive layouts with three orientations, though Chatterjee *et al.* [20] informally discuss this issue.

- All layouts in the Morton family show identical spatial locality whether traversed in row-major or column-major order (see Section 3.5).

3.5 Spatial Locality in Recursive Array Layouts

Consider a loop nest accessing a $2^m \times 2^n$ array A, arranged in any one of the radix-2 Morton layouts, in either row- or column-major order. Also assume that the cache capacity is C with an even power-of-two cache line size B ($= 2^{2b}$). The innermost loop traverses the array in either row- or column-major order with a unit stride. If $2^m, 2^n > C$, spatial re-use along the other direction of traversal can be neglected. With these assumptions, the Proposition 3.1 holds true.

Proposition 3.1 (Morton layout offers same spatial locality for both canonical traversal orders) In Morton layouts, given a cache with any even power-of-two block size, with an array mapped according to one of the Morton layouts ($\mathcal{S}_{mz}$, $\mathcal{S}_{mg}$, $\mathcal{S}_{mu}$ or $\mathcal{S}_{mx}$), 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. The proof is as follows:

Array A can be viewed as an array of $\frac{2^m}{2^b} \times \frac{2^m}{2^b}$ blocks of size 2^{2b}. That is, each block holds $2^b \times 2^b$ square sub-array of A. When elements of A are accessed, corresponding blocks are accessed in some order. Following a cache miss, the square sub-array containing of the requested element of array A (of size $2^b \times 2^b$) is fetched. For the row-major traversal, the first access will cause a cache miss, but since

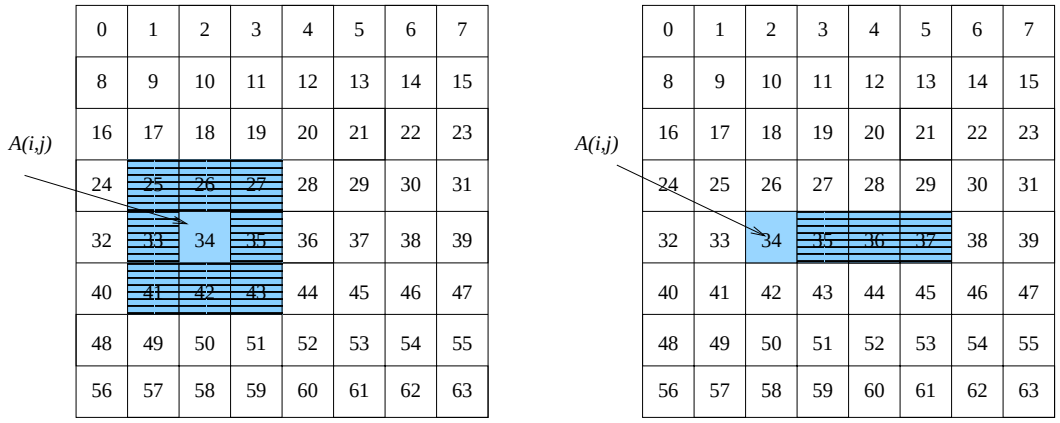


Figure 3.6: **Comparison of neighbourhood r-model spatial locality and spatial locality interpreted by modern machines.** The figure shows the major difference in the interpretation of spatial locality assumed by Avigail and Itai [63] from the standard interpretation. Their model assumes r-neighbourhood model, and all cells within the radius of r are considered to be demanded and fetched by cache whereas the standard model of spatial locality assumes that limited number of adjacent elements in the direction of layout axis are fetched in to a cache line. Figure (a) shows the case for $r = 1$ and the issue of alignment is ignored here.

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

Table 3.2: **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.

the square sub-array includes the rest of the 2^b-element row, the next $2^b - 1$ accesses will be cache hits The same is true for the column-major traversal as well. Thus, in either case, the cache miss rate is at most 1 in 2^b or $\frac{1}{2^b}$.
Thus, Morton array offers equal spatial locality whether traversed in row- or column-major order.

□

However, the presence of temporal re-use and spatial re-use along the other direction of traversal (row-to-row or column-to-column) will improve the miss rate (i.e. the miss rate is reduced).

Avigail and Itai [63] measure the locality of space filling curves and they conclude that space filling curves have better locality compared to row-major or column-major traversals. However, their cache model and the interpretation of spatial locality are much more restricted to the domain of image processing applications. In particular, their interpretation of spatial locality is bound to the data elements lying within a radius r from a considered data element. Figure 3.6 illustrates this difference in the interpretation of spatial locality.

3.6 Choice of a Layout for Experiments

When considering alternatives to canonical layouts, we have a choice between fully recursive layouts, where decomposition is applied up to the element level, and minimal blocking schemes, where the blocking is applied with specific block sizes, based on the underlying memory system. First, we justify our selection for a fully recursive layout over minimally blocked layout and then we justify the selection of a specific recursive layout, Z–Morton layout in particular, as the experimental layout to test the hypothesis that Morton layout is an attractive compromise and alternative to canonical layouts.

3.6.1 Criteria for Selection

We consider the following as some of the important criteria to select a layout:

- **Architecture Independence**: The chosen layout should work well on a wide range architectures with little or no tuning of the layout.
- **Addressing Cost**: The overall instruction count (in terms of simple instructions), which is a direct measure of relative addressing cost, should be as small as possible.
- **Spatial locality**: The dilation effect (see Section 3.4), and thus the number of orientations, has a direct connection to the spatial locality of a layout. As the number of orientations increases, this dilation effect is less pronounced and spatial locality is improved.
- **Symmetry**: The layout should offer the same spatial locality, for row-major and column-major traversals.
- **Space overhead**: Non-linear layouts may require additional padding. It can be argued that, given smaller system page-size, the space overhead introduced by padding has little impact on the overall performance of Morton layouts, as these pages are never loaded. However, if the system supports super-pages [61] and if the application uses this feature, then this argument is not valid.

3.6.2 Recursive Layouts

As pointed out in the earlier sections of this chapter, recursive layouts apply blocking at every power-of-two down to the element level. Depending on the array size, the number of levels to which the blocking is applied may vary and may even exceed the number of actual levels of memory hierarchy in the system, which rarely exceeds four (L1, L2, L3-level caches and system pages — we ignore register level blocking). Complete recursive blocking of these layouts matches well with the uniform memory hierarchy model suggested by Alpern *et al.* [4]. Extra levels of blocking have little impact on the overall performance and recursive layouts require no special tuning across architectures, which helps architecture-independent programming.

Characteristics, such as the number of orientations and address mapping functions of recursive layouts vary greatly. Part of the addressing cost can be recovered by control structures if the algorithms are tiled or recursively formulated. When this is not the case, the table-lookup scheme can be used. For Z-Morton, two lookup tables are sufficient. The Hilbert curve and G-Morton schemes require more

than two lookup tables. For X-Morton and U-Morton, the address calculation is partly complicated with the presence of the Exclusive-OR operation.

The Hilbert curve has the highest number of orientations — 4 and the layout suffers least from the dilation effect. G-Morton has two orientations and all other members of the Morton family have only single orientation.

Recursive layouts are only applicable to arrays with power-of-two sizes. When the size is not a power-of-two, the array has to be padded to the next power-of-two size. For small-sized system pages, padding has little impact on the overall performance as padded pages are never touched. However, in the presence of super-pages, padded space will be a significant overhead.

3.6.3 Minimally-Blocked Layouts

Minimally-blocked layouts have fewer levels of blocking — only to match the actual levels of memory hierarchy in the system. Once the number of levels for blocking is decided, block size for each level is carefully chosen to match the block size of the corresponding level in the memory hierarchy. Depending on the implementation, either tiles or elements of tiles can be ordered linearly. The 4D layout discussed in Section 3.3 is an example of minimally-blocked layout, where tiles are arranged in row-major order and elements inside these tiles are ordered in column-major order (or vice versa). Despite the fact that minimally-blocked layouts may offer good performance, there are disadvantages in using this layout scheme. Firstly, choosing the correct block size for each level is a search space problem, partly due to the fact that block size optimised for one level is likely to influence the performance of other levels [60, 76]. Secondly, the layout is a serious limitation for portability as the layout needs to be tuned based on the architectural parameters. In other words, the underlying architecture has a greater influence in the layout structure.

The address calculation cost, as in recursive layouts, can partly be recovered by means of control structures. The table-lookup scheme can also be utilised to simplify the implementation and unrolling, often with two different lookup tables.

Minimally-blocked layouts require the array sizes (of each dimension) to be a multiple of the chosen tile size (in the corresponding direction). Often, this involves padding but less than that of the recursive layouts.

3.6.4 Selection of a Layout

When considering these facts, we observe the following:

- Minimally-blocked layouts offer reduced space overhead over fully recursive layouts. However, in the absence of support for super-pages, the extra space overhead is expected to have little impact on performance.

- In many cases, both fully recursive and minimally-blocked schemes have similar addressing cost for non-tiled/non-recursive implementations. The exact cost of addressing may vary slightly, if recursive algorithms are used. However, all our kernels are non-blocked and are not recursive.

- For minimally-blocked layouts, choosing the correct block size is always restricted by the target architecture and involves further tuning. However, when optimal blocking factors are chosen, better performance can be expected. On the other hand, fully recursive layouts are inherently

blocked up to the element level and guaranteed to cover all levels of memory hierarchy. This, in contrast to minimally-blocked scheme, should cover a wide variety of architectures. By choosing a fully recursive layout, the implementation can be made architecture/platform independent. So fully recursive layouts have all the locality benefits of minimally-blocked layouts.

Based on these arguments, we consider that fully recursive layouts to be more promising in terms of architecture independence and in terms of offering equal spatial locality for row-major and column-major traversal orders. If, as we claim, extra blocking does not hurt us very much in terms of performance, fully recursive layouts are the best choice to test our hypothesis.

Among recursive layouts, the Hilbert layout has the overall minimum dilation effect. However, the addressing cost compared to the other layouts is very high, involving more than just the two lookup tables, required by Z-Morton. By the same token, G-Morton does not qualify. Among the remaining layouts, Z-Morton is an ideal choice considering the addressing cost and symmetry property within larger blocks. This has led us to select the Z-Morton layout to test our hypothesis, in this thesis.

3.7 Summary

In this chapter, we have discussed different recursive array layouts and layout transformations. Their orientations determine the address calculation complexity and the dilation effect. We also have justified the selection of Z-Morton layout to test the hypothesis outlined at the beginning of this thesis.

Theoretically, we have illustrated how a member in the Morton layout family can be a compromise and an attractive storage layout to canonical layouts. The hypothesis is entirely based on theoretically predicted/calculated values and they may vary on real machines. This hypothesis can only be verified with a systematic implementation and analysis of the results - which is explored in the chapters which follow.

Chapter 4

An Experimental Study of Basic Morton Layout on a suite of Micro-Benchmarks

In this chapter, we experimentally evaluate our hypothesis that Morton layout is a compromise storage layout between row-major and column-major layouts. We also evaluate two different methods of address calculation for Morton layouts. This chapter, which serves to establish a baseline against which the experiments in the later chapters are compared, is based on papers presented at Euro-Par 2002 [85], UKPEW 2003 [81] and on our journal paper [83].

4.1 Introduction

In the previous chapter, theoretically, we have illustrated that 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. The hypothesis is entirely based on theoretically predicted/calculated values and they may vary on real machines. In this chapter, we verify this hypothesis through systematic implementation of experiments and analysis of these experimental results.

Perhaps controversially, we confine our attention to "naively" written codes, where a mismatch between access order and layout is reasonably likely. In the final chapter of this thesis, we discuss how this constraint could be relaxed. 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 experts and that compilers will successfully analyse and optimise the code, but we recognise that very often, neither is the case.

4.2 Contributions of this Chapter

In this chapter:

- We present an extensive and systematic study of Morton layout using a substantial range of problem sizes. This shows a number of interesting effects, and Morton layout appears less attractive to the better performing canonical layout. However, in the later chapters, we discuss further improvements to the performance of Morton layout.

```
#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];
    }
}
```

Figure 4.1: **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 in to a dilated integer. The macros `INC_1(.)` and `INC_0(.)` increments the dilated variables $\mathcal{D}_1(.)$ and $\mathcal{D}_0(.)$ respectively.

- The dilated arithmetic approach for address calculation works when the array is accessed using an induction variable which can be incremented using dilated addition. Despite the performance, the dilated arithmetic approach is too complex to execute at every loop iteration. We found that a much simpler scheme often works nearly as well: pre-computing two different tables for the two mappings $\mathcal{D}_0(i)$ and $\mathcal{D}_1(i)$ and using lookup tables to calculate Morton layout addresses is remarkably effective. We show that it compares well with the dilated arithmetic scheme proposed by Wise *et al.* [89, 90], and offers useful flexibility.

- We evaluate the hypothesis that Morton layout, implemented using lookup tables, is a compromise between row-major and column-major layout. We present extensive experimental results using five simple numerical kernels, running on five different processors (Section 4.5).

In order to make sure that the performance results we obtain for different benchmarks on different hardware platforms are as good as can reasonably be evaluated if the micro-benchmark code were part of a larger program and yet representative of likely experience in practice, we need a common set of compilers and compiler flags. Compilers and compiler flags recommended by the vendors of hardware, for their SPEC CFP2000 (base) benchmark reports, are chosen to address exactly the same issue of fair comparison. We therefore use the same compilers and compiler flags recommended in the CFP2000(base). In addition to this, for each micro-benchmark, we manually verified that all compilers generate similar code, i.e. no loop interchange or vectorisation took place.

```
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]];
    }
}
```

Figure 4.2: **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.

4.3 Morton Order Address Calculation

A naive, but correct implementation strategy for calculating the offset address of an element at (i, j) in a Morton array A is to call a subroutine to compute that address. As discussed in Chapter 3, Section 3.4, this involves bit-dilation and bit-interleaving, which are expensive operations to execute at every loop iteration. As in linear layouts, it is possible to integrate the process of address computation into loop structures that surround the array accesses — the dilated arithmetic scheme. An alternative strategy is to pre-compute the values for $\mathcal{D}_1(i)$ and $\mathcal{D}_0(j)$ into two different look-up tables and perform lookups on demand.

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, (for example, $2^n \times 2^m$, $n < m$), we construct the lookup-tables with different sizes, so that corresponding indices are dilated only up to bit n and up to m. Such a scheme permits us to vary the radices used and to construct a completely non-uniform layout.

We have evaluated these two techniques using the Matrix-Multiply (ikj variant) on a number of different platforms. The code variants are shown in Figures 4.1 and 4.2 and resulting performance results are shown in Figure 4.3. Results show that the dilated arithmetic implementation is almost always faster; but, the difference is usually less than 20%. The exact performance of dilated-arithmetic scheme may vary depending on the benchmark kernel and on the architecture. In the remainder of this thesis, we use the table lookup scheme exclusively, for the following reasons:

- We expect the difference in performance can be regained by unrolling.
- The table lookup scheme offers considerable flexibility in choosing and mixing different layouts very easily.

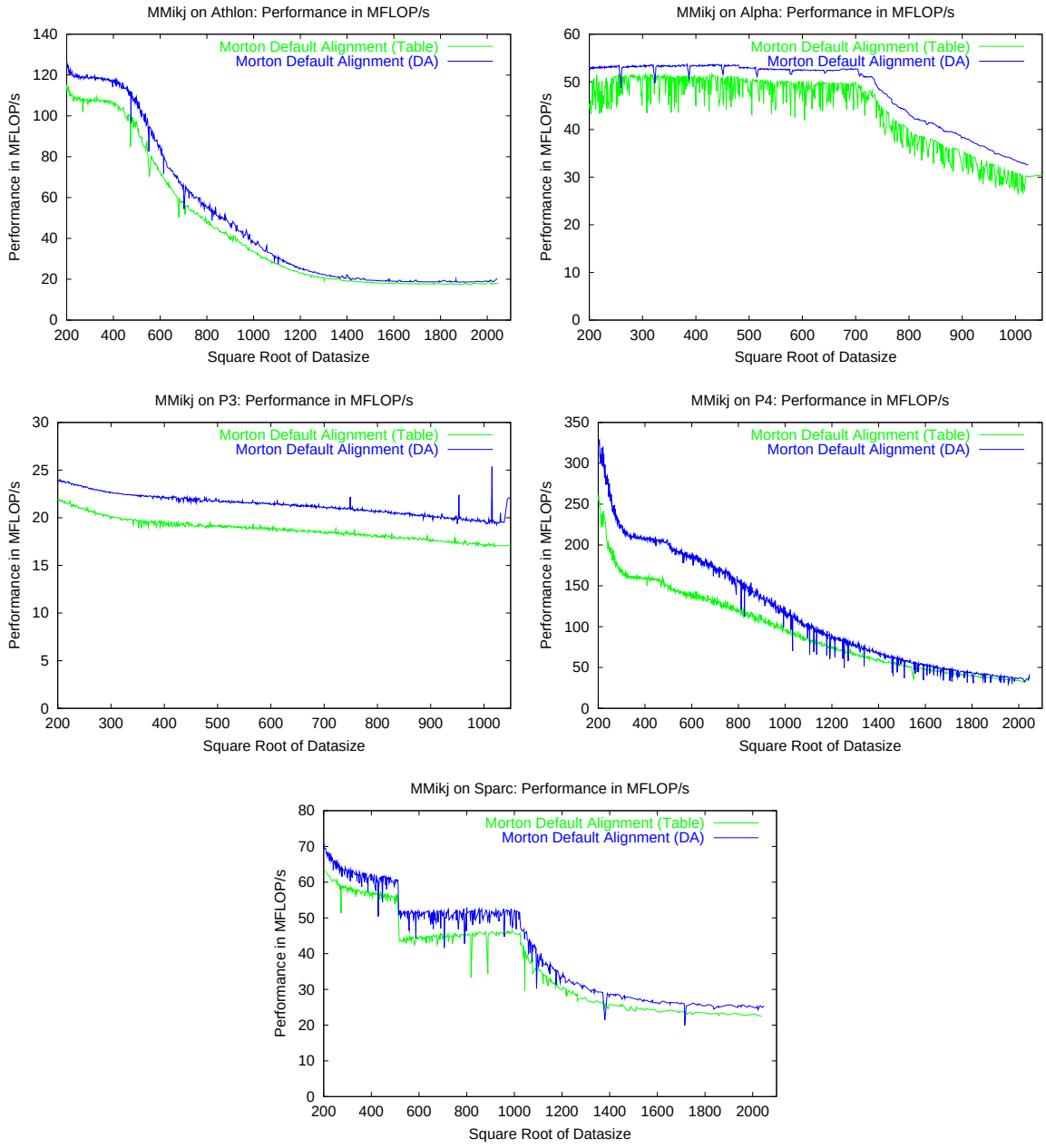


Figure 4.3: **Matrix multiply (ikj) performance (in MFLOPs) of dilated arithmetic Morton address calculation compared against the table-based Morton address calculation.** Graphs show performance achieved by the two different implementations (in MFLOPs) on different platforms, for problem sizes ranging from 200×200 to 2048×2048, wherever possible. Details of the systems are given in Table 4.1. On nearly all systems , the dilated-arithmetic implementation performs relatively better than the table implementation. However the difference is usually less than 20%. Further, our results in following chapters, show that this performance loss can be re-gained by unrolling the innermost loops of the kernels.

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

Table 4.1: **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 [78].

- Although our benchmarks are limited to simple array subscripts, in some applications where array subscripts are not induction variables, Dilated Arithmetic cannot be used.

However, with compiler support, many applications could benefit from the dilated arithmetic approach, possibly leading in many cases to more positive conclusions,

4.4 Experimental Setup

Benchmark kernels and architectures. To test our hypothesis that Morton layout, implemented using lookup tables, is a useful compromise between row-major and column-major layout experimentally, we have collected a suite of simple implementations of standard numerical kernels operating on two-dimensional arrays and carried out experiments on five different architectures. The benchmarking kernels used are shown in Figures 4.4 and 4.5 and the platforms in Table 4.1.

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

```
/* ADI Main Loop */
for (t=0;t<nIters;t++){
    for (i=1;i<sz;++i){
        for (j=0; j< sz; ++j)
          A[i][j] += A[i-1][j];
    for (i=0;i<sz;++i){
      for (j = 1; j< sz; ++j){
          A[i][j] += A[i][j-1];
}/*end of time loop*/
```

(a) Alternating Direction Implicit Kernel (ADI) – ii, ij order

```
/* MMijk Main Loop */
for( i = 0; i < sz; ++i ){
   for( j = 0; j < sz; ++j){
      for( k = 0; k < sz; ++k ){
        C[i][j] += A[i][k] * B[k][j];
      }
   }
}
```

(b) Matrix Multiply. ijk loop nest order

```
/* MMikj Main Loop */
for( i = 0; i < sz; ++i ){
   for( k = 0; k < sz; ++k ){
      r = A[i][k];
      for( j = 0; j < sz; ++j){
        C[i][j] += r * B[k][j];
      }
   }
}
```

(c) Matrix Multiply. ikj loop nest order

Figure 4.4: **Core loops of the MMijk, MMikj and ADI kernels used in our experimental framework.** The Figure shows the key loops of the MMijk, MMijk and ADI kernels. Though the kernels shown here use standard 'C' style notations, the actual implementations are slightly different. The ijk variant of the matrix multiply should perform very poorly due to large stride access for one of the arrays. The ikj variant should perform better than the ijk variant as it has a unit stride access for all of the arrays.

```
/* Jacobi2D Main Loop */
for (it=0; it<nIters; it++) {
    for (i=1; i<sz-1; ++i){
        for (j=1; j<sz-1; ++j){
            B[i][j]= 0.25 *(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1]);
        }
    }
    it++;
    if (it >= nIters) break;

    for (i=1; i<sz-1; ++i){
        for (j=1; j<sz-1; ++j)
            A[i][j]= 0.25 *(B[i-1][j]+B[i+1][j]+B[i][j-1]+B[i][j+1]);
        }
    }
}/* End of Time Loop */
```

(a) Jacobi2D two-dimensional four-point stencil smoother

```
for (k=0;k<sz;k++){
    A[k][k] = sqrt(A[k][k]);
    for (i=k;i<sz;i++){
        A[i][k] /= A[k]+[k];
        for (j = k; j < i; ++j){
            A[i][j]   -= A[i][k] * A[j][k];
        }
    }
}
```

(b) Cholesky, K-Variant

Figure 4.5: **Core loops of the Jacobi2D and Cholesky kernels used in our experimental framework.** The Figure shows the key loops of the Jacobi2D and Cholesky (k-variant) kernels. Though the kernels shown here use standard 'C' style notations, the actual implementations are slightly different.

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 of kernels described in Figures 4.4 and 4.5), 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. However for ease of completion of experiments, we preferred to specify the experiment to randomise the selection of experiment.

- 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.

- 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's Q-Test [74], a simple method to determine if the individual values that are farthest from the mean lie outside the confidence limits, 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 this thesis 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 [74]. Although we do not show these in this thesis, we have calculated and plotted 90% confidence intervals for all data we report.

4.5 Experimental Results

Table 4.2 shows the baseline performance achieved by each machine using standard row-major layout. In Figures 4.6–4.15 we show our interesting/important results in detail. For each experiment/architecture pair, we give a broad characterisation next to each graph.

As an overview, for the Morton layout, we notice that

- For the Adi kernel, on Alpha, Pentium III and Sparc systems, Morton layout performs well compared to column-major but not compared to row-major layouts. (Figures 4.6 and 4.7). However, on Athlon and Pentium 4, Morton layout performs as badly as the column-major layout, especially for problem sizes larger than around 1000×1000 (Figure 4.7).

	Adi		Cholk		Jacobi2D		MMijk		MMikj	
	min	*max*	*min*	*max*	*min*	*max*	*min*	*max*	*min*	*max*
Alpha	27.0	84.5	6.8	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	48.9	134.1	4.8	266.1	126.6	1337.3	17.9	766.0	281.4	939.1
Sparc	14.6	54.5	17.8	78.5	33.2	139.2	12.4	131.9	33.5	145.2

Table 4.2: **Baseline row-major performance of various kernels on different systems.** For each kernel, for each machine, we show the performance range in MFLOPs for row-major array layout over all problem sizes covered in our experiments (as shown in Figures 4.6–4.15).

- For the Jacobi2D kernel, on Alpha and Sparc systems, Morton layout performs better than column-major but not better than row-major layouts. (Figure 4.8). Morton layout performs as badly as or worse than the column-major layout on all x86 platforms (Figure 4.9).
- For the MMikj kernel, on Alpha and Sparc systems, Morton layout performs better than column-major but not better than row-major layouts (Figure 4.10). On Athlon and Pentium III, Morton layout is worse than or as badly as the column-major layout. On Pentium 4, Morton layout is slightly faster than the column-major layout ((Figure 4.11).
- For the MMijk kernel, Morton layout performs better than both the canonical layouts on Alpha, for problem sizes larger than around 350×350. On Sparc, Morton layout performs better than the worst canonical layout for problem sizes larger than around 750×750 (Figure 4.12). On all x86 platforms, Morton layout performs worse than the column-major layout (Figure 4.13).
- For the Cholesky-k kernel, on Alpha, Morton layout performs worse than both the canonical layouts for problem sizes up to around 512×512. However, for problem sizes larger than 512×512, Morton layout performs better than both the canonical layouts. On Sparc, Morton layout performs better than column-major but not better than row-major for problem sizes up to around 512×512. Following the cross-over point at around 512×512, Morton layout performs worse than both the canonical layouts (Figure 4.14). On all x86 platforms, Morton layout performs worse than both the canonical layouts (Figure 4.15).

These results show that Morton layout fails to be an attractive compromise to canonical layouts, on most of the x86 platforms. On Alpha and Sparc platforms, Morton layout performs better than the worst canonical layout. This appear to suggest that Morton layout performs well on machines with large L2 caches.

On nearly all systems, the results clearly show the impact of L2 cache and TLB span on overall performance. Frequently, when either the whole working set or some part thereof exceeds the capacity of a particular level of memory hierarchy, a substantial drop in performance can be observed. For example, a sudden drop in the performance of *MMijk* with the column-major layout on *Alpha* (Figure 4.12), near the problem size 350 coincides with the working set exceeding the size of the TLB

span (Alpha has 128–entry Data TLB, each entry pointing to an 8KB page: This matches the size of a 362×362 array of doubles). Similarly for Adi on Alpha (Figure 4.6) where the sudden drop occurs near the problem size 700×700, which is approximately 4MB. This corresponds to the L2 cache size of the Alpha. Similar observations can be made in Figures 4.6, 4.8 and 4.10 near problem sizes corresponding to their L2 cache size.

For kernels with high spatial locality, such as MMikj and Jacobi2D, where high MFLOPs are achieved, Morton layout might be slowed down by insufficient memory accesses compared to that for the canonical layouts.

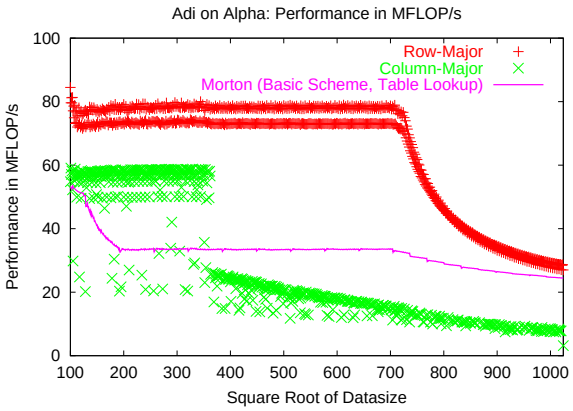
Further, row-major and column-major layouts show wide variations in performance with small changes in problem sizes whereas the performance of Morton layout remains very consistent. Although padding the length of the rows or columns of an array can significantly improve performance, the amount of padding required is small, but needs to be chosen very carefully.

These results are further improved in Chapters 5 and 6.

4.6 Conclusions

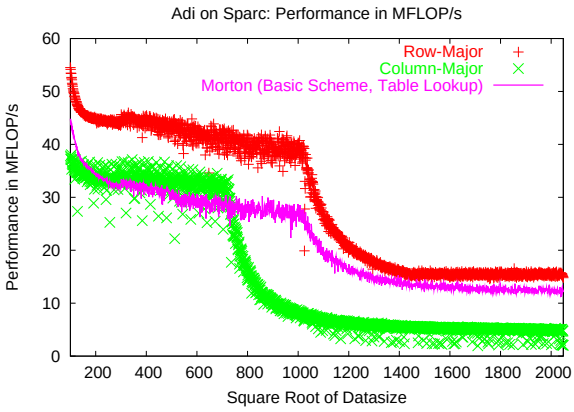
Using a small suite of dense kernels working on two-dimensional arrays, we have carried out an extensive study of row-major, column-major and Morton layouts, covering non-power-of-two problem sizes within a substantial range. On some machines, we found that Morton array layout, even implemented with a lookup table with no compiler support, to be potentially promising alternative 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. The overall performance of the basic Morton scheme, from our experimental results, suggests it is only attractive for some architectures and kernels. However, our analysis points to potential improvements of the Morton layout. We describe two such optimisation techniques to improve the performance of basic Morton layout in the next two chapters.

Adi on Alpha



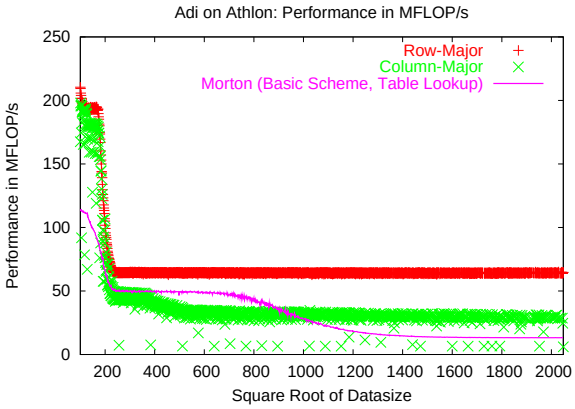
- Notice upper limit is 1024×1024.
- The fall-off in RM performance occurs at 725×725 when the total datasize exceeds L2 cache size (4MB, direct mapped). This assumes a working set of 725×725 doubles.
- RM below about 725×725 has a bimodal distribution.
- Notice the sharp drop in CM performance at around 360×360.

Adi on Sparc



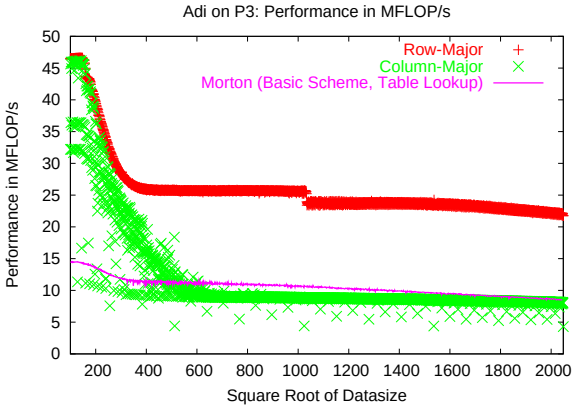
- The fall-off in RM performance occurs at 1024×1024 when the total datasize exceeds the L2 cache size (8MB, direct mapped). This assumes a working set of 1024×1024 doubles.
- Notice the drop in CM performance which occurs after 720×720.
- Morton version has high variation (confi dence intervals for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

Figure 4.6: **ADI performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. These results are further improved in Chapters 5 and 6, whose results are reported in Figures 5.4 and 6.7, respectively.



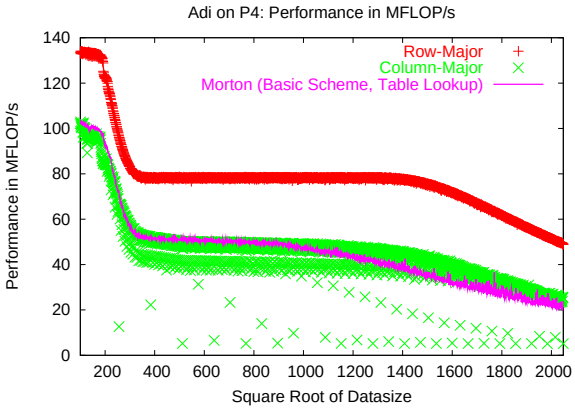
Adi on Athlon

- There is a cross-over between Morton layout and CM at around 900×900.
- Notice some very bad performance drops on CM for individual problem sizes.



Adi on Pentium III

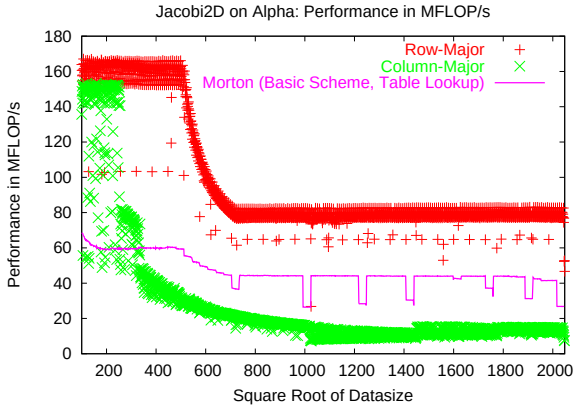
- Morton version virtually coincides with CM and performs as badly as the CM version.



Adi on Pentium 4

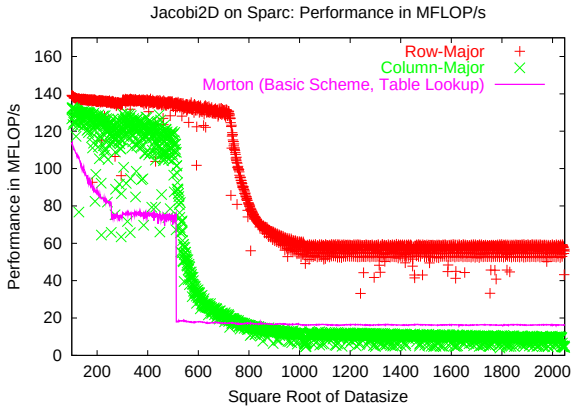
- Morton generally performs no better than CM.
- Notice, however, some really bad drops in CM performance for some datasizes.

Figure 4.7: **ADI performance in MFLOPs on Athlon, P3 and P4.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. These results are further improved in Chapters 5 and 6, whose results are reported in Figures 5.5, 5.6 and 6.7, respectively.



Jacobi2D on Alpha

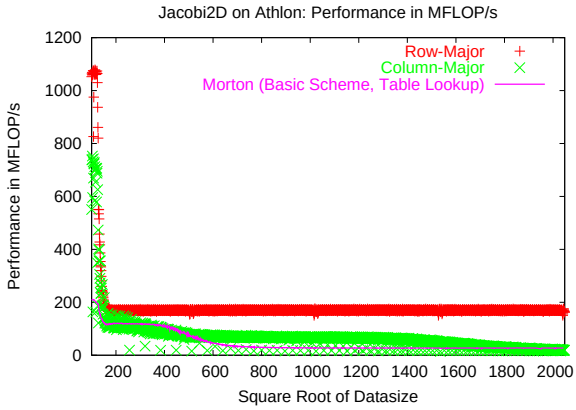
- RM performance drops off after 512×512. Assuming a working set of two arrays of 512×512 doubles, this is the point where the working set exceeds L2 cache size (4MB, direct mapped). RM performance levels off after 725×725, which appears to be when one single 725×725 array of doubles exceeds the L2 cache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



Jacobi2D on Sparc

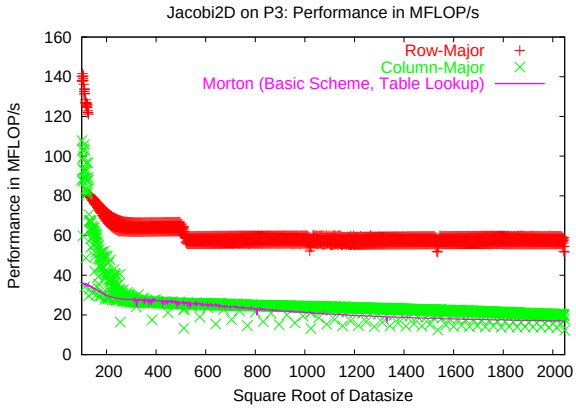
- There is a sharp drop in Morton, and to a slightly lesser extent in CM, performance at 512×512.
- RM performance drops off after 725×725. Assuming a working set of two arrays of 725×725 doubles, this is the point where the working set exceeds L2 cache size (8MB, direct mapped).

Figure 4.8: **Jacobi2D performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. These results are further improved in Chapters 5 and 6, whose results are reported in Figures 5.7 and 6.8, respectively.



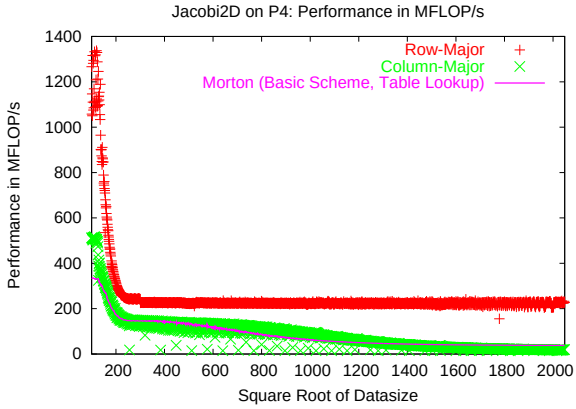
Jacobi2D on Athlon

- Notice there is a cross-over between the Morton layout and CM at around 512×512.
- Morton layout performs as badly as the CM for all problem sizes.



Jacobi2D on Pentium III

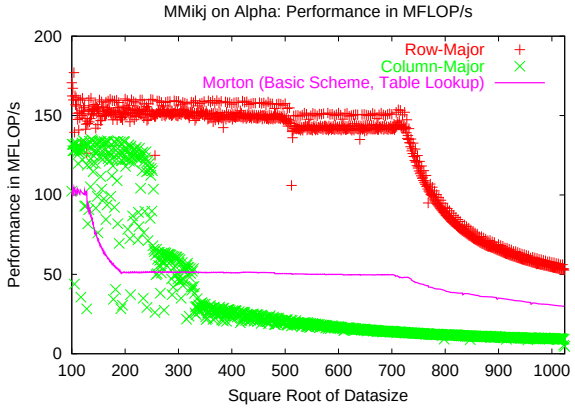
- Morton layout performs as badly as the CM for all problem sizes.



Jacobi2D on Pentium 4

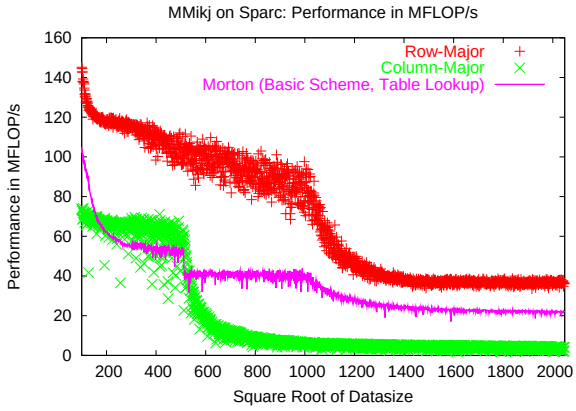
- Morton layout performs disappointingly throughout all problem sizes.

Figure 4.9: **Jacobi2D performance in MFLOPs on Athlon, P3 and P4.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. These results are further improved in Chapters 5 and 6, whose results are reported in Figures 5.8 and 6.8, respectively.



MMikj on Alpha

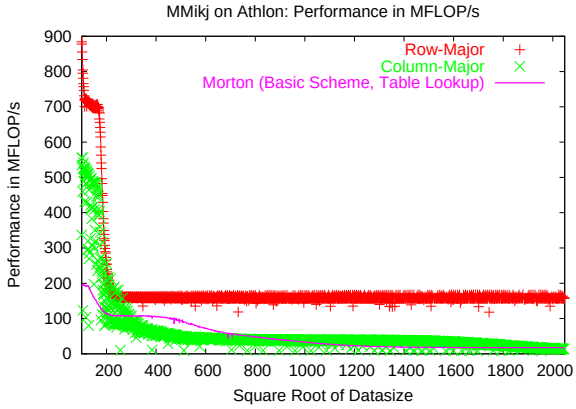
- Notice upper limit is 1024×1024.
- The drop in RM (and Morton) performance occurs at 725×725. This corresponds to the datasize where one array of 725×725 doubles exceeds the L2 cache size (4 MB, direct mapped).



MMikj on Sparc

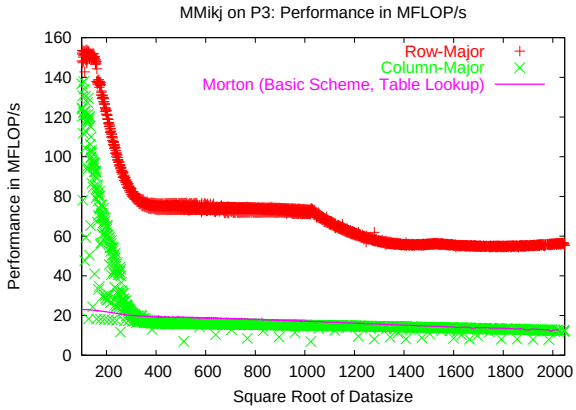
- The drop in RM (and Morton) performance occurs at 1024×1024. This corresponds to the datasize where one array of 1024×1024 doubles exceeds the L2 cache size (8MB, direct mapped).

Figure 4.10: **MMikj performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. These results are further improved in Chapters 5 and 6, whose results are reported in Figures 5.9 and 6.9, respectively.



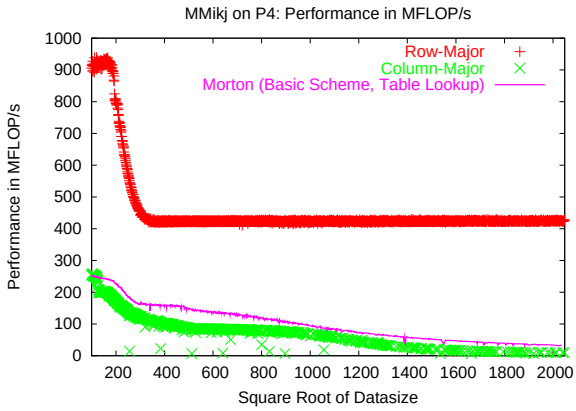
MMikj on Athlon

- Notice there is a cross-over between the Morton layout and CM at around 700×700.
- Morton layout is almost as badly as the CM for all problem sizes starting from 700×700.



MMikj on Pentium III

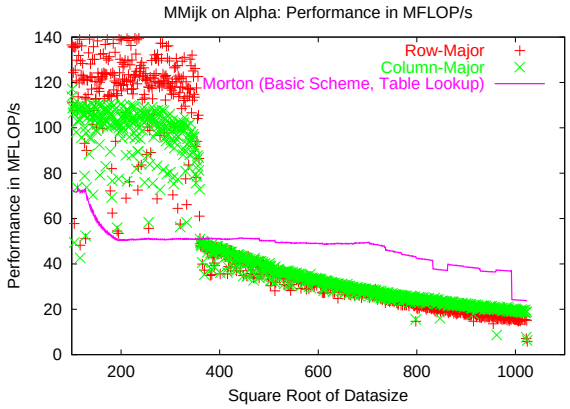
- Morton layout performs as badly as the CM layout for most of the problem sizes.



MMikj on Pentium 4

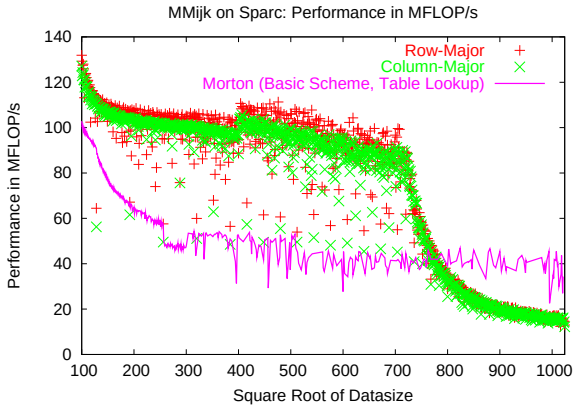
- Morton layout is only slightly faster than CM.

Figure 4.11: **MMikj performance in MFLOPs on Athlon, P3 and P4.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. These results are further improved in Chapters 5 and 6, whose results are reported in Figures 5.10 and 6.9, respectively.



MMijk on Alpha

- Notice upper limit is 1024×1024.
- Notice the sharp drop in RM and CM performance around 360×360.
- 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 interference effect.

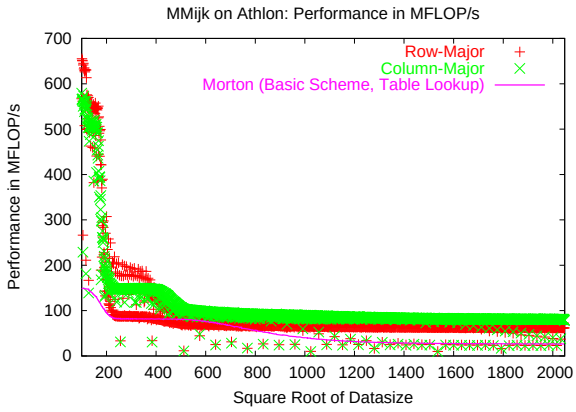


MMijk on Sparc

- 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.

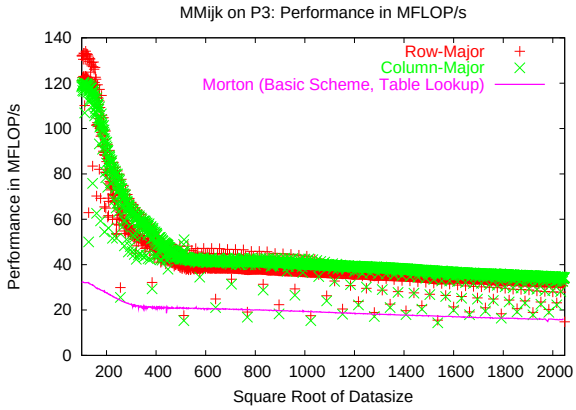
Figure 4.12: **MMijk performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. These results are further improved in Chapters 5 and 6, whose results are reported in Figures 5.11 and 6.10, respectively.

MMijk on Athlon



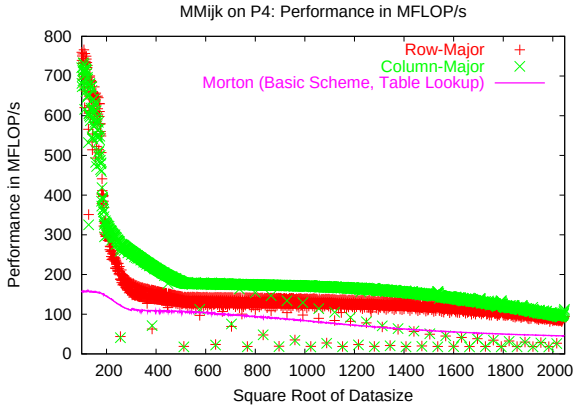
- RM data is trimodal.
- Lexicographical layouts perform better than Morton layout.

MMijk on Pentium III



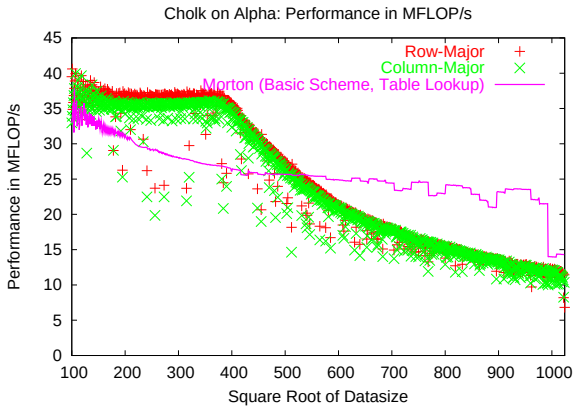
- Lexicographical layouts perform better than Morton layout.
- The mean RM and CM performance across the range of data sizes is very close. There are many problem sizes where RM is worse than CM; however, for particular problem sizes (or, carefully chosen row padding), RM can perform much better than CM.

MMijk on Pentium 4



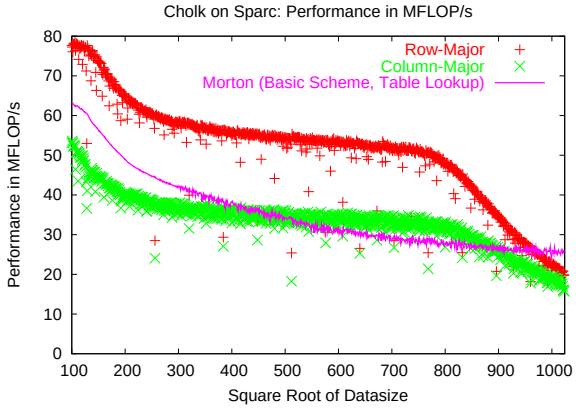
- Notice that for some individual problem sizes, both RM and CM drop drastically below Morton.
- In overall, lexicographical layouts perform better than Morton layout.

Figure 4.13: **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 significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapters 5 and 6, whose results are reported in Figures 5.12 and 6.10, respectively.



Cholesky-k on Alpha

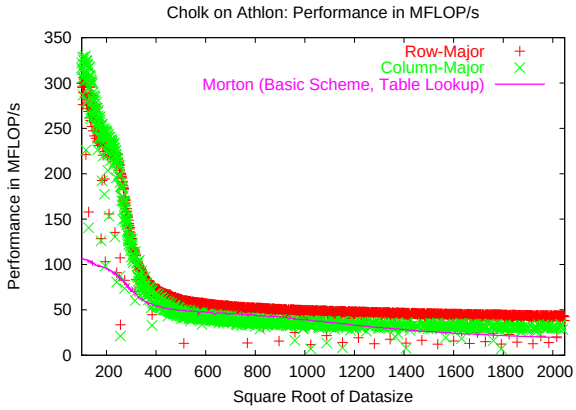
- Notice upper limit is 1024×1024.
- Notice the sharp drop in RM and CM performance around 360×360.
- Notice there is a cross-over between the Morton layout and canonical layouts at around 500×500 and Morton layout performs better than canonical layouts for problem sizes larger than the crossover point.



Cholesky-k on Sparc

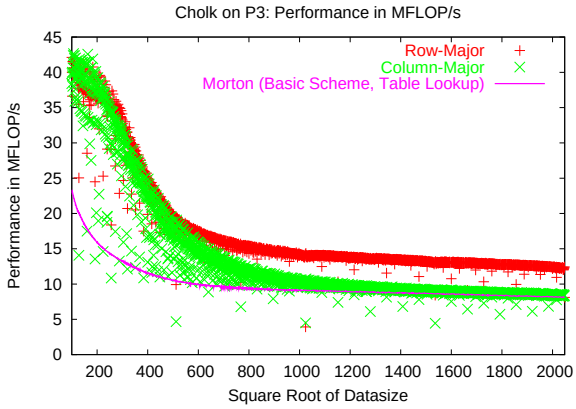
- Notice upper limit is 1024×1024.
- Notice the sharp drop in RM and CM performance around 750×750.
- Performance of the Morton layout varies throughout but seems to be promising for problem sizes larger than 1024×1024.

Figure 4.14: **Cholesky k-variant performance in MFLOPs on Alpha and Sparc.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables.



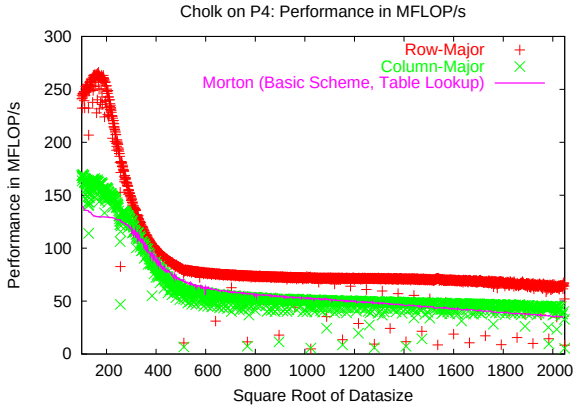
Cholesky-k on Athlon

- Morton layout performs as badly as the CM layout.
- Performance of canonical layouts collapses starting from problem sizes 200×200.
- Performance of Morton layout is worse than both the canonical layouts.



Cholesky-k on Pentium III

- Morton layout is worse than both the canonical layouts.
- The worst layout is at-most 30% slower than best canonical layout.



Cholesky-k on Pentium 4

- Morton layout performs as badly as the CM layout.
- Notice the sharp drop in RM and CM performance.

Figure 4.15: **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. These results are further improved in Chapters 5 and 6, whose results are reported in Figures 5.13 and 6.11, respectively.

Chapter 5

Significance of Memory Alignment in Morton Order Performance

In this chapter, we analyse the effect of alignment of base address of Morton array in overall performance and show that aligning Morton arrays at the page level leads to better performance. This chapter is based on our paper presented at Languages and Compilers for Parallel Computing Workshop(LCPC) 2003 [82], and on our journal paper [83].

5.1 Introduction

In the previous chapter we exhaustively evaluated Morton layout over number of different platforms using a suite of micro-benchmarks. Our results have shown that naive implementation of Morton layout is only attractive for some platforms and kernels and often we find their performance disappointing. In this chapter, we analyse the effect of alignment of base address of Morton array in overall performance and show that aligning Morton arrays at the page level leads to better performance.

5.2 Effect of Memory Alignment in Morton Layouts

With lexicographic layouts, it is often important to pad the row (respectively column) length to avoid associativity conflicts [71]. With Morton layout, it turns out to be important to consider padding the base address of the array, as will be explained below.

In Chapter 3, Section 3.5, we discussed the cache hit rate resulting from Morton order arrays implicitly assuming 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. Such an allocation is unbiased towards any particular order of traversal. However, in Figure 5.1 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 5.2, 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

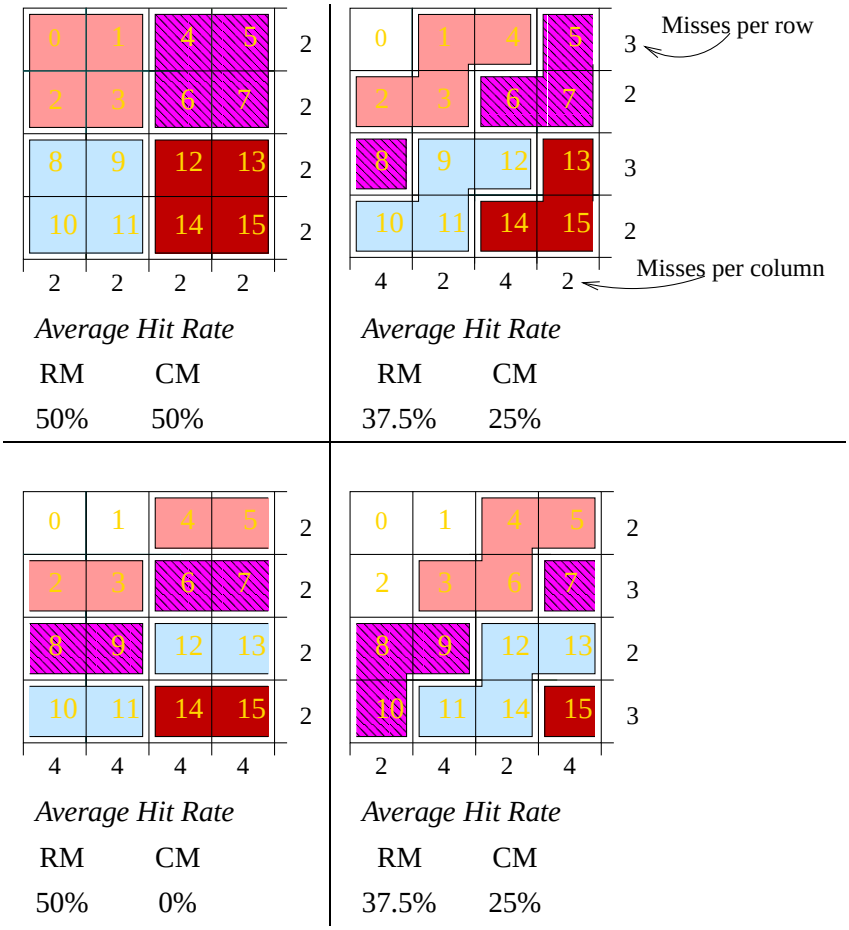


Figure 5.1: **Alignment of Morton order arrays I:** Figures illustrate how the cache performance can change when the alignment is varied (in the order of perfectly aligned at 32-byte boundary and offset from there by 8, 16 and 24 bytes) for a part of a larger Morton array. The numbers next to each row 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 column-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.

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 5.4–5.13 and discussed in more detail in the next section, broadly confirm the conclusion of our theoretical analysis.

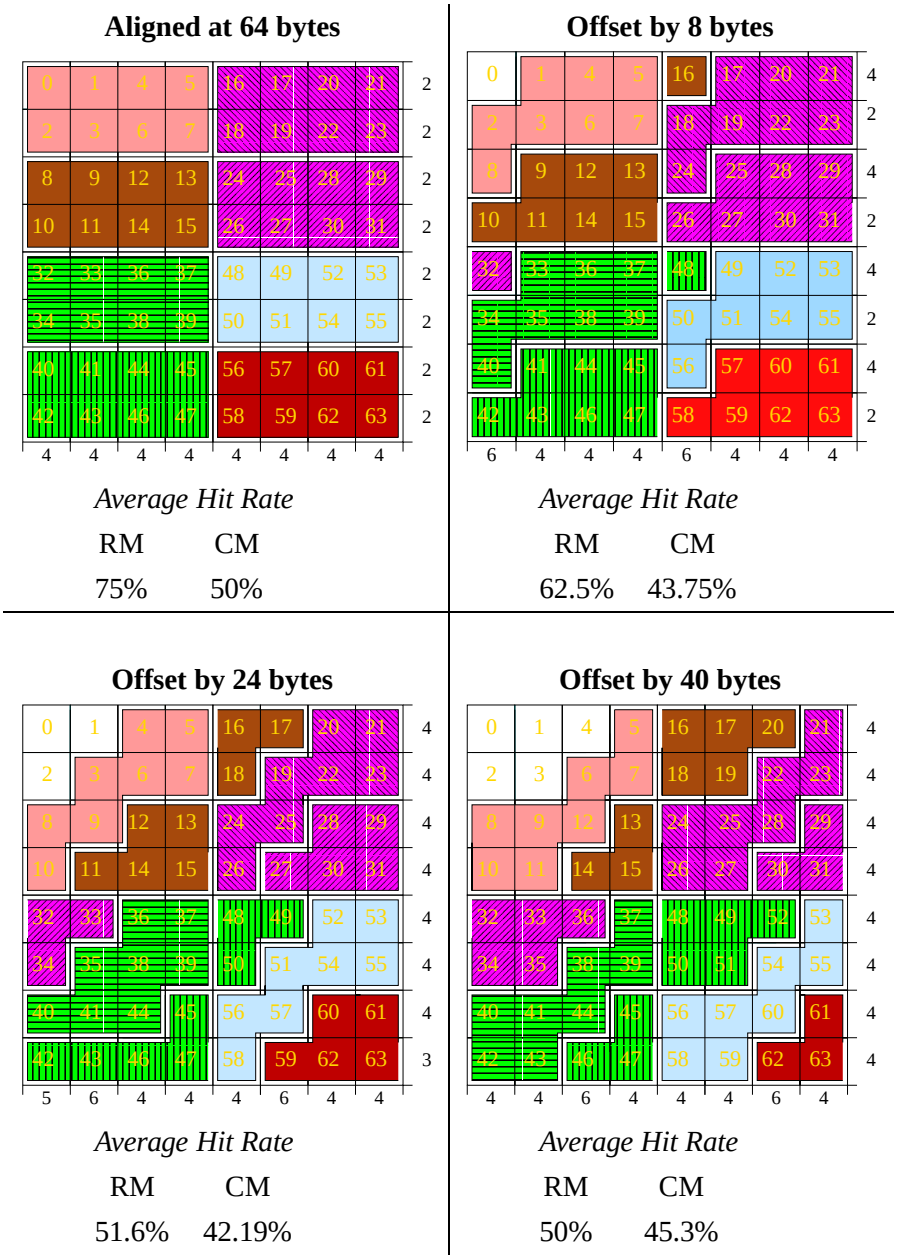


Figure 5.2: **Alignment of Morton order arrays II:** For a non-square cache block, such as 64 bytes or 8 double words, this figure illustrates how cache performance varies with the alignment of the base address of the array, in the order of perfectly aligned at 64 bytes (top-left), and offset by 8 (top-right), 24 (bottom-left) and 40 (bottom-right) 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 column-major (CM) traversal. As can be seen by the illustrations, when the array is imperfectly aligned, in addition to losing the symmetry of the Morton layout we get worse spatial locality.

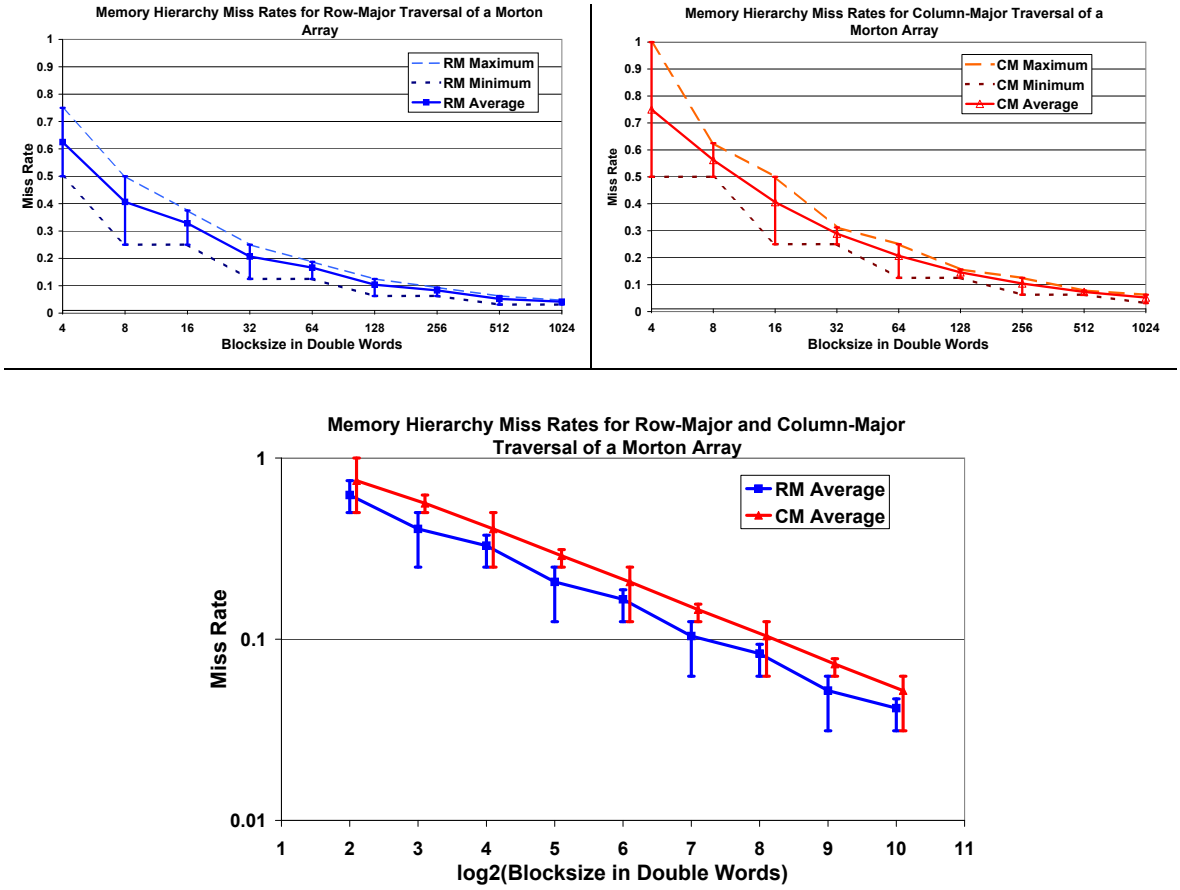


Figure 5.3: **Miss-rates for row-major and column-major traversal of Morton arrays**. We show the best, worst and average miss-rates for different units of memory hierarchy (referred to as block-sizes), across all possible alignments of the base address of the Morton array. The top two graphs use a linear y-axis, whilst the graph underneath shows the same data on logarithmic y-axis to illustrate that the pattern of miss-rates is in fact highly structured across all levels of the memory hierarchy. Notice that, for square block sizes, miss-rates for row- and column-major traversals coincide.

5.3 Effect of Alignment Across Different Levels of Memory Hierarchy

In order to investigate this effect further, we systematically calculated the resulting miss-rates for both row- and column-major traversal of Morton arrays, over a range of possible levels of memory hierarchy, and for each level, different mis-alignments of the base address of Morton arrays. The range of block sizes in memory hierarchy we covered was from 2^2 double words, corresponding to a 32-byte cache line to 2^{10} double words, corresponding to an 8KB page. Architectural considerations imply that block sizes in the memory hierarchy such as cache lines or pages have a power-of-two size. For each 2^n block size, we calculated, over all possible alignments of the base address of a Morton array with respect to this block size, respectively the best, worst and average resulting miss-rates for both row-major and column-major traversal of the array. The standard C library `malloc()` function returns addresses which are double-word aligned. We therefore conducted our study at the resolution of double words. The results of our calculation are summarised in Figure 5.3. Based on those results, we offer the following conclusions.

1. When no special steps are taken to align the base address of a Morton array, the *average* hit

rates resulting from traversing the same Morton array either in row- or column-major order are *always suboptimal*.

2. The best average hit rates for both row- and column-major traversal are always achieved by aligning the base address of Morton arrays to the largest significant block size of memory hierarchy (e.g. page size).

3. The difference between the best and the worst miss-rates can be very significant, up to a factor of 2 for both row-major and column-major traversal.

4. We observe that the symmetry property which we mentioned in Chapter 3, Section 3.5 is in fact *only* available when using the best alignment and for even power-of-two block sizes in the memory hierarchy. For odd power-of-two block sizes (such as $2^3 = 8$ double words, corresponding to a 64-byte cache line), we find that the Z-Morton layout is still significantly biased towards row-major traversal. An alternative recursive layout such as Hilbert layout [20, 33] may have better properties in this respect.

5. The absolute miss-rates we observe drop exponentially through increasing levels of the memory hierarchy (see the graphs in Figure 5.3). However, if we assume that not only the block size but also the access time of different levels of memory hierarchy increase exponentially [4], the penalty of mis-alignment of Morton arrays does not degrade significantly for larger block sizes. From a theoretical point of view, we therefore recommend aligning the base address of all Morton arrays to the largest significant block size in the memory hierarchy, i.e. page size.

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

5.4 Experimental Evaluation

In our experimental evaluation, we have studied the impact on actual performance of the alignment of the base address of Morton arrays. For each architecture and each benchmark, we have measured the performance of Morton layout both when using the system's default alignment (i.e. addresses as returned by `malloc()`) and when aligning arrays to each significant size of memory hierarchy. Our experimental methodology is same as described in Section 4.4. Our theoretical assertion that aligning with the largest significant block size in the memory hierarchy, i.e. page size, should always be best is supported in most, but not all cases, and we assume that where this is not the case, this is due to interference effects. Figures 5.4–5.13 include performance results for Morton storage layout with different alignments of the array's base address.

5.5 Performance Results

In Chapter 4, we evaluated the Z-Morton layout with the default alignment. In this chapter, we evaluate and present the performance results, when Morton order arrays are aligned at significant boundaries of the memory hierarchy.

Figures 5.4–5.13 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 aligning Morton arrays at different memory boundaries is useful or not. We summarise our observations here:

- On Alpha (Figures 5.4, 5.7, 5.9 and 5.11):
  - Alignment does not improve the performance of the default Morton layout in the Adi and MMikj kernels.
  - However, aligning Morton arrays at the L2 or page-level speeds up the default Morton performance in the Jacobi2D and MMijk kernels.
  - For the Jacobi2D kernel, aligning Morton arrays at the L2-level provides the best Morton performance while aligning to the page-level is less effective.
  - For the MMikj kernel, aligning at the page-level provides marginal benefit compared to aligning at the L2-level.
  - In summary, alignment improves the performance of most of the kernels. However, there is no clear indication whether aligning Morton arrays at the page-level is always the best.
- On Sparc (Figures 5.4, 5.7, 5.9 and 5.11):
  - Alignment improves the performance of the default Morton layout in the Adi, MMikj and MMijk kernels.
  - Aligning at the page-level improves the performance of the default Morton layout by at least 50%, in the Jacobi2D kernel.
  - For the ADI kernel, aligning at the L2-level provides the best performance. Aligning at the page-level is less effective.
  - For all the other kernels, aligning at the page-level provides the best performance.
  - In summary, alignment improves performance and aligning at the page-level is not wrong.
- On Athlon (Figures 5.5, 5.8, 5.10, 5.12, and 5.13):
  - Alignment makes very little difference to the overall Morton performance. The speedup gained by aligning Morton arrays are very small (regardless of the alignment boundaries).
  - Wherever there is a speedup, aligning beyond the L2-level is less effective.
  - In summary, alignment rarely improves the overall performance and aligning at the L2-level is sufficient.
- On Pentium III (Figures 5.5, 5.8, 5.10, 5.12, and 5.13):
  - Aligning Morton arrays at different memory boundaries provides a little, but considerable, speedup.
  - Page-aligning Morton arrays in the Adi kernel provides at least 50% speedup from the default version. Only by alignment, Morton layout becomes a compromise layout between row- and column-major layouts.

  - In all other kernels, the speedup gained by aligning Morton arrays is rather small.
  - Among all experimental results, the page-aligned version is the fastest.
  - In summary, alignment improves the overall performance. Aligning at the page-level is often produces the best possible performance.

- On Pentium 4 (Figures 5.6, 5.8, 5.10, 5.12 and 5.13):

  - In overall, alignment improves the default Morton performance only by a small factor.
  - In all cases, the page-aligned version is the fastest.
  - For the MMikj kernel, aligning at the L1 level, slows downs the performance of Morton layout.
  - In summary, aligning at the page-level always improve the default Morton performance.

These observations appear to suggest that aligning Morton arrays can improve the performance in majority of the cases. We further observe that, the maximum speedup that could be gained varies depending on the alignment, platform and the application. On some of the platforms for some of the kernels, aligning beyond the L2-level produced suboptimal performance results (Adi on Alpha). In some cases, aligning beyond the L2-level is less effective (Adi on Sparc, Adi on Athlon, Adi on Pentium III, Jacobi2D on Athlon, Jacobi2D on Pentium III, MMikj on Athlon, MMikj on Pentium III, Cholesky-k on Athlon and Cholesky-k on Pentium III). In all other cases, aligning Morton arrays at the page-level produces the best performance results. From these results, we can conclude that aligning Morton arrays should improve the performance of Morton layout and aligning at the page-level, in majority of the cases, is not always wrong. These results are further improved in Chapter 6.

5.6 Conclusions

This chapter has pointed out and illustrated the importance of alignment of base address of Morton order arrays and showed the performance benefits achieved when the arrays are aligned properly at the significant boundaries of memory-hierarchy of a system.

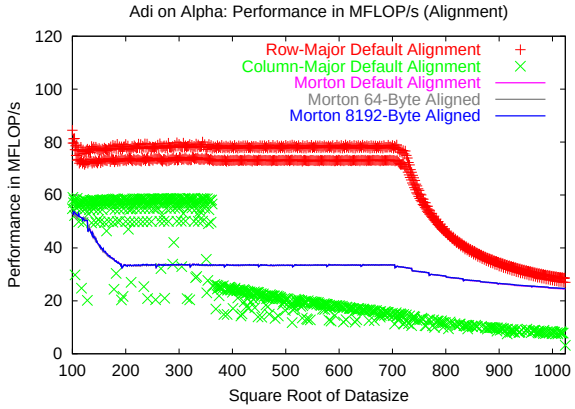
For most of the experiments, with the exceptions mentioned in the previous section, our theoretical conclusions from Section 5.2 are supported by our experimental data: padding the base address of Morton order arrays to a significant size in the memory hierarchy, such as cache line size or page size 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. This is to suggest that, aligning at page level is guaranteed to offer better performance than default alignment but the improvement over the L2-level alignment can be marginal depending on the platform.

Although we have shown that proper alignment can increase the performance of Morton layout, and therefore increase the competitiveness of the basic Morton scheme, the performance gained by proper alignment is still very low.

In the next chapter, we look at another simple optimisation which improves performance Morton layout — unrolling.

Adi on Alpha

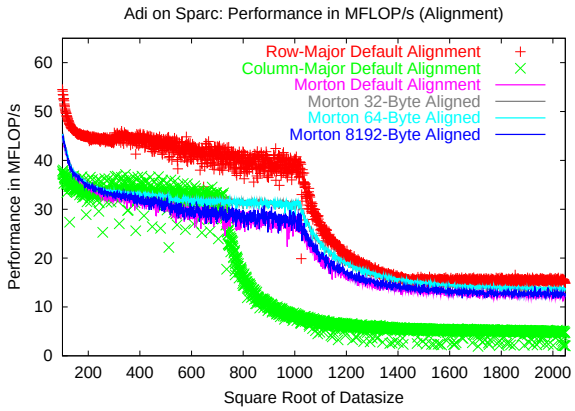


- Alignment does not make any difference to the Morton performance, all three lines coincide.

As in Chapter 4 (Figure 4.6):

- Notice upper limit is 1024×1024.
- The fall-off in RM performance occurs at 725×725 when the total data-size 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.

Adi on Sparc



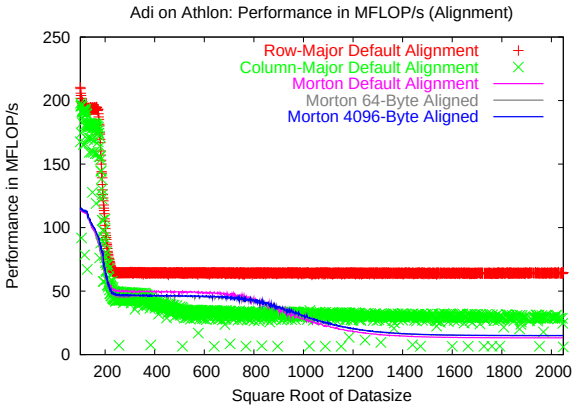
- Both the 64-Byte and 32-Byte aligned versions have equal performance characteristics.
- These two versions perform better than the default aligned and page aligned versions.
- Page aligned Morton version performs slightly better than the default Morton layout.

As in Chapter 4 (Figure 4.6):

- For smaller problem sizes (less than around 200×200) alignment makes very little difference.
- The fall-off in RM performance occurs at 1024×1024 when the total data-size 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 5.4: **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 significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.7.

Adi on Athlon

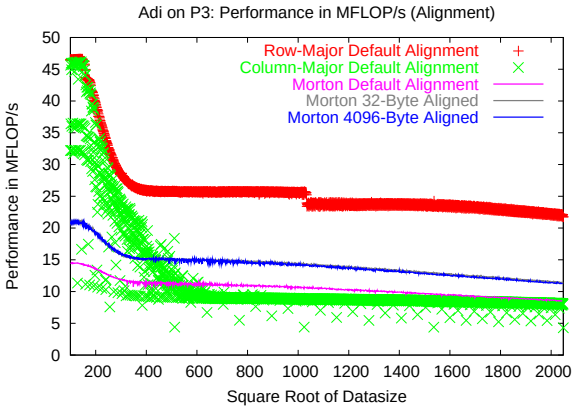


- The page-aligned and 64-byte aligned versions have the same performance characteristics.
- Within the problem size range of 200×200 and 900×900, default Morton performs better than any other aligned versions of Morton. Following the cross-over point at around 900×900, both page-aligned and 64-byte aligned versions perform better than the default Morton layout. That is, for large data-sizes, page-aligned is the best Morton version.
- From this cross-over point, CM version performs better than any Morton version.

As in Chapter 4 (Figure 4.7):

- Notice some very bad performance drops on CM for individual problem sizes.

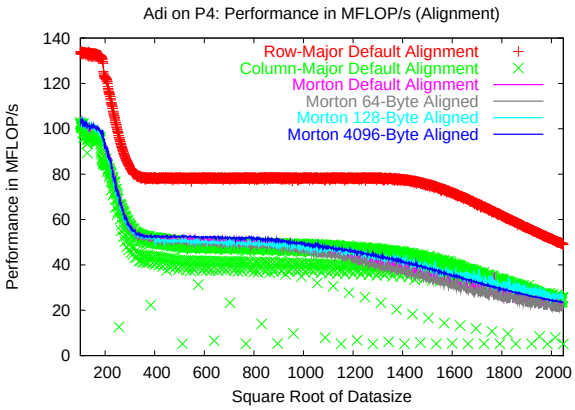
Adi on Pentium III



- Morton with default alignment virtually coincides with CM.
- Morton aligned to L2 cache line length (32 bytes) leads to a clear improvement. Alignment to page-size offers no further benefi t. (Alignment at page-size is superimposed over alignment at L2 cache line length)
- Morton layout performs better than column-major layout only with alignment.

Figure 5.5: **ADI performance in MFLOPs on Athlon and P3.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.7.

Adi on Pentium 4

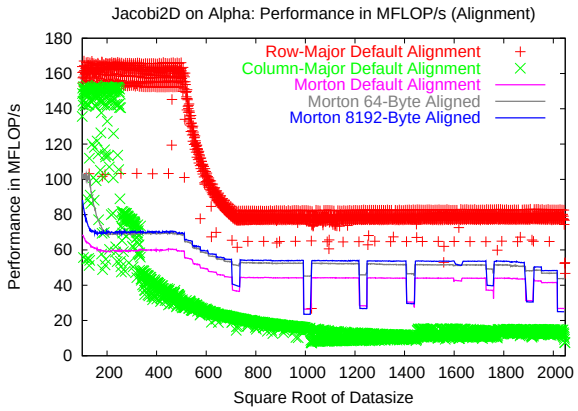


- Page-aligned Morton version has only a very little performance gain from L2 (128-Byte) aligned Morton version.
- L1-aligned (64-Byte) Morton version has the same performance characteristics as L2-aligned/page-aligned versions up to the problem size around 1024×1024. Starting from the problem size around 1024×1024, L1-aligned version performs worse than any of the Morton versions.

- For larger problem sizes, L1 aligned version performs worse than CM.
- For large problem sizes, L2 aligned is slightly faster than page-aligned Morton.
- All Morton versions except the page-aligned version, has considerably larger variation starting from problem size at around 1024×1024.
- All Morton versions generally performs no better than CM.
- Notice, that there are some really bad drops in CM performance.

Figure 5.6: **ADI performance in MFLOPs on P4.** We compare row-major (RM), column-major (CM) and Morton implemented using lookup tables. For Morton, performance is shown for default alignment and significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.7.

Jacobi2D on Alpha

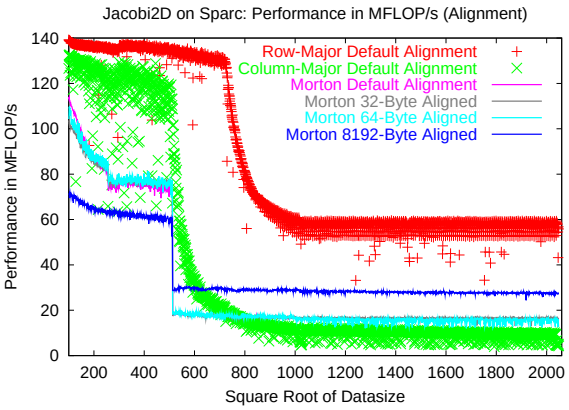


- Alignment to page or L2 cache line length improves Morton performance.
- Page-aligned version performs slightly better than L2 aligned version.

As in Chapter 4 (Figure 4.8):

- Notice upper limit is 1024×1024
- 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



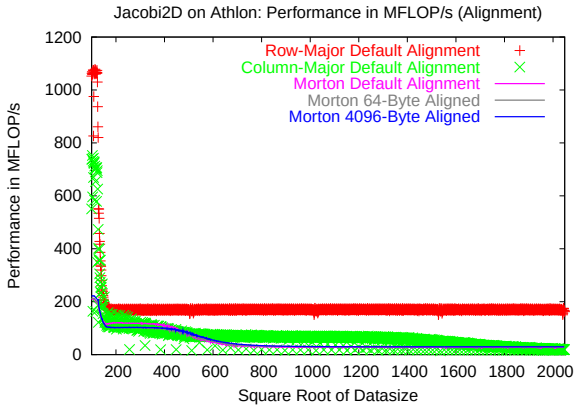
- Aligning to page-level improves Morton performance by around 50%.
- Page-aligned Morton performs better than CM starting from problem size around 600×600.
- There is a cross-over between page-aligned Morton and all other Morton versions at 512×512.
- For larger problem sizes, 32- and 64-Byte aligned versions perform as badly as the default Morton layout. However, default Morton layout has less variation.

As in Chapter 4 (Figure 4.8):

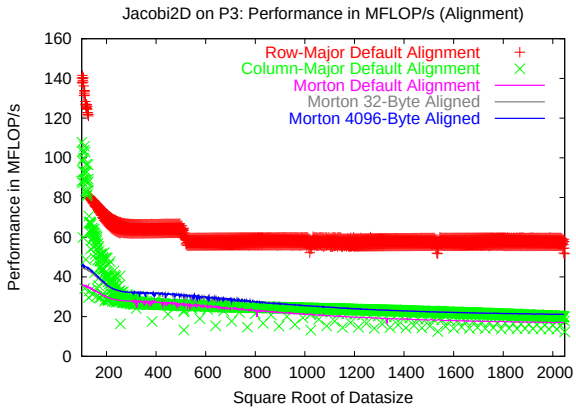
- There is a sharp drop in Morton, and to a slightly lesser extent CM, performance at 512×512.
- RM performance drops off after 725×725. Assuming a working set of two arrays of 725×725 doubles, this is the point where the working set exceeds L2 cache size (8MB, direct mapped).

Figure 5.7: **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 significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.8.

Jacobi2D on Athlon

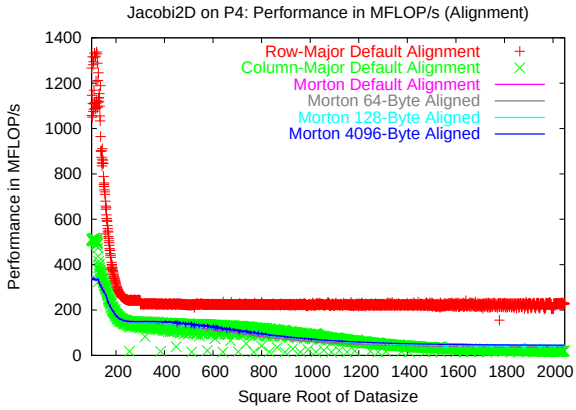


- Aligning to L2-/page-level slightly improves the performance of default Morton layout, especially for large problem sizes.
- Even the best Morton version (with alignment) does not perform any better than CM.
- Notice there is a cross-over between the default and L2-/page-aligned Morton versions at around 450×450.



Jacobi2D on Pentium III

- Aligning to L2- and page-level improves the performance of default Morton.
- Both the L2- and page-aligned versions improve the performance of default Morton almost by the same factor.
- However, the best Morton version (L2-/page-aligned) performs as badly as the CM version.

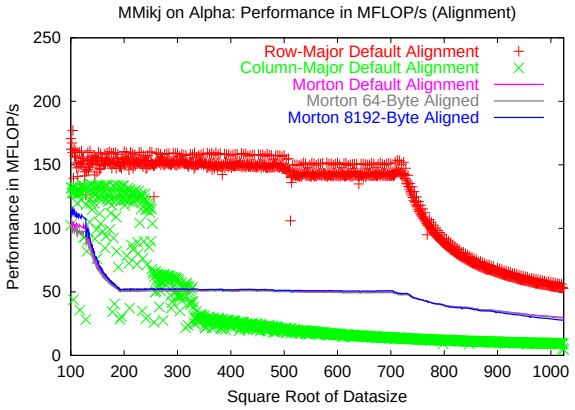


Jacobi2D on Pentium 4

- Page-aligned and L2-aligned Morton are slightly better than default-aligned.
- Page-aligned version performs better than L2-aligned; and L2-aligned version performs better than L1-aligned version.
- For large problem sizes, Morton layout performs marginally better than CM.

Figure 5.8: **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 significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.8.

MMikj on Alpha

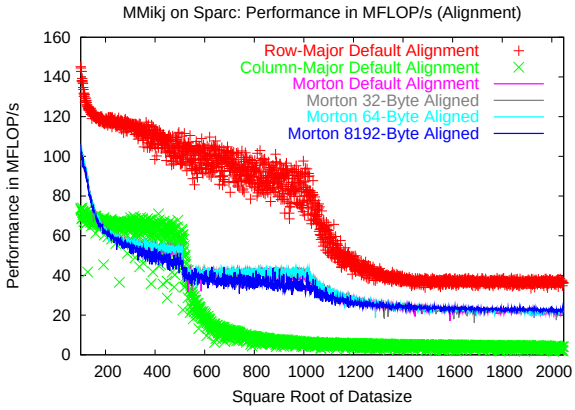


- Alignment makes little difference to the overall Morton performance.

As in Chapter 4 (Figure 4.10):

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

MMikj on Sparc



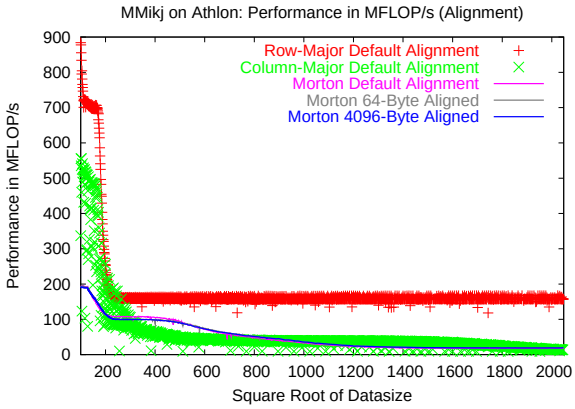
- L2-aligned (64-Byte) aligned version is slightly faster than other Morton versions for up to problem sizes around 1024×1024.
- For larger problem sizes, page-aligned Morton is slightly faster than the other versions.
- The overall improvement in the performance of Morton layout is considerably small.

As in Chapter 4 (Figure 4.10):

- The drop in RM (and Morton) performance occurs at 1024×1024. This corresponds to the data-size where one array of 1024×1024 doubles exceeds the L2 cache size (8MB, direct mapped).

Figure 5.9: **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 significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.9.

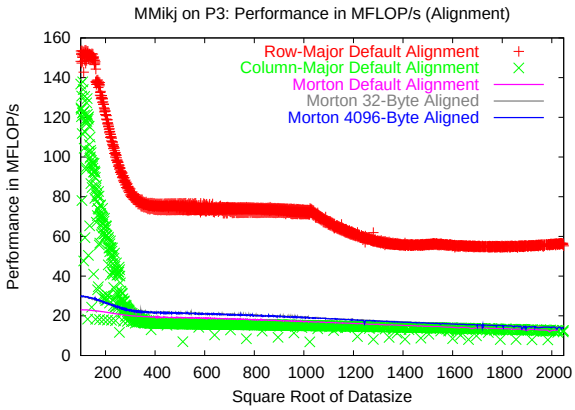
MMikj on Athlon



- Alignment makes little difference to the overall Morton performance.
- Aligning Morton arrays beyond L2 level does not improve performance.
- L2/Page-aligned Morton versions perform slightly better than default-aligned.

- Notice there is a cross-over between the default and L2-/page-aligned Morton versions at around 500×500.
- Morton layout performs as badly as the CM version.

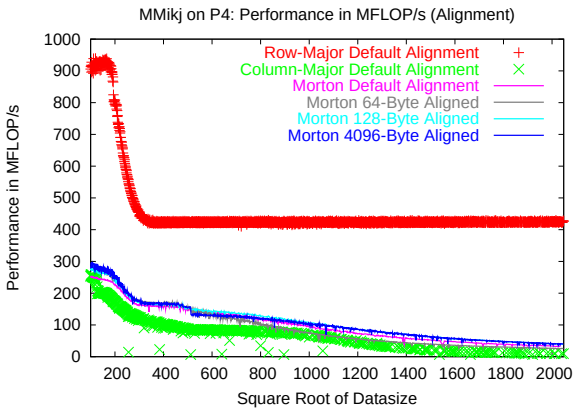
MMikj on Pentium III



- Aligning to L2- and page-level slightly improves the performance of default Morton layout, for all problem sizes.
- Aligning beyond L2 level does not improve performance of Morton layout (i.e. L2 and page-aligned versions virtually coincide).

- Morton layout, even with the best alignment, performs as badly as the CM version.

MMikj on Pentium 4

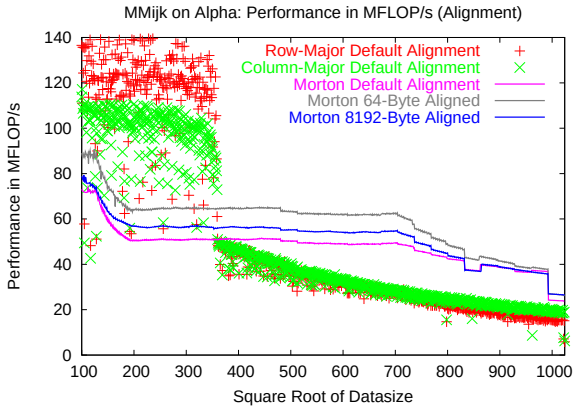


- L2 (128 byte) and page-aligned Morton are slightly better than default aligned version.
- L1 (64 byte) alignment is slightly worse than the default aligned version.
- Morton layout with L2/page alignment performs marginally better than the CM version.

- Notice that the variations that Morton layouts experience are not random and they are reproducible.

Figure 5.10: **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 significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.9.

MMijk on Alpha

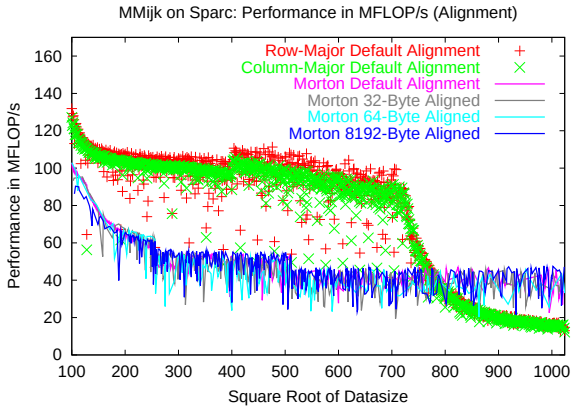


- Page-aligned Morton is faster than default, but L2-aligned is faster than page-aligned.
- The performance gain by different alignments diminishes as the problem size increases.
- Morton layout performs better than both the RM and CM versions.

As in Chapter 4 (Figure 4.12):

- Notice upper limit is 1024×1024.
- Notice the sharp drop in RM and CM performance around 360×360.
- 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 interference effect.

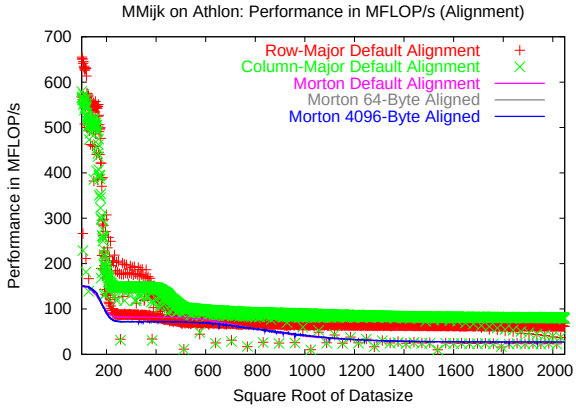
MMijk on Sparc



As in Chapter 4 (Figure 4.12):

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

Figure 5.11: **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 significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.10.

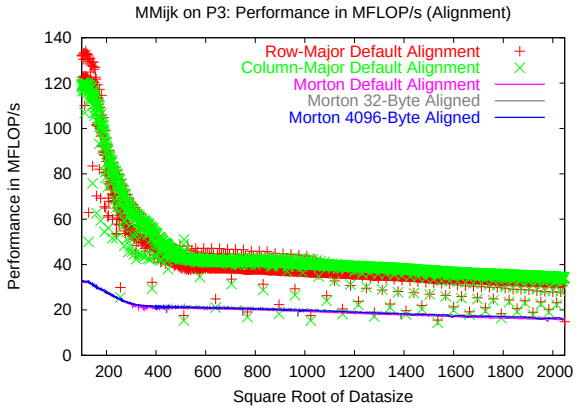


MMijk on Athlon

- Default Morton is faster than L2- or page-aligned for smaller problem sizes.
- Alignment makes very little difference in overall performance of Morton layout.

As in Chapter 4 (Figure 4.13):

- Performance of Morton layout is worse than that of canonical layouts, starting from problem sizes around 700×700.

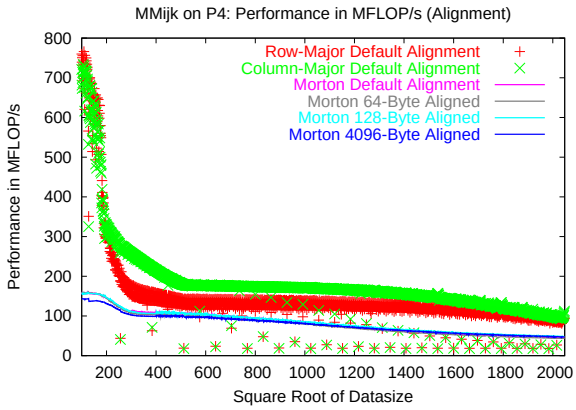


MMijk on Pentium III

- Speedup by aligning Morton arrays at L2-/Page-level is negligible.
- Aligning Morton layout any more than at L2 level does not improve the performance.

As in Chapter 4 (Figure 4.13):

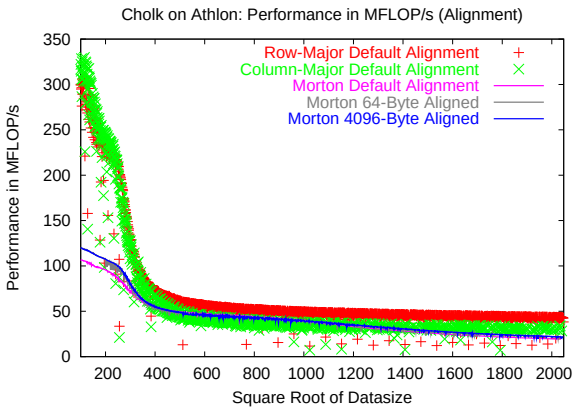
- The mean RM and CM performance across the range of data sizes is very close. There are many problem sizes where RM is worse than CM; however, for particular problem sizes (or, carefully chosen row padding), RM can perform much better than CM.
- Performance of Morton layout is worse than both the canonical layouts, throughout all problem sizes.



MMijk on Pentium 4

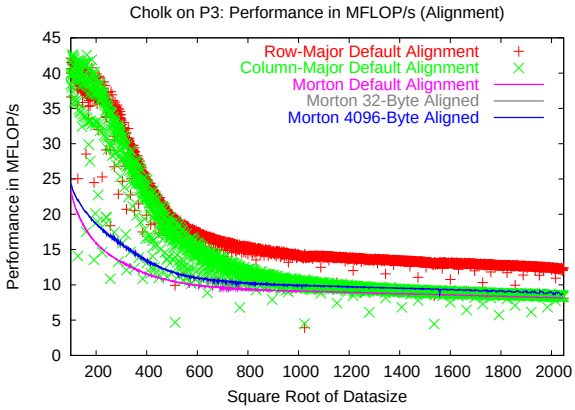
- L2 (128 byte) and page-aligned Morton is slightly better than default Morton.
- L1 (64 byte) alignment is slightly worse than default-aligned.
- Notice that for some individual problem sizes, both RM and CM drop drastically below Morton.

Figure 5.12: **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 significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.10.



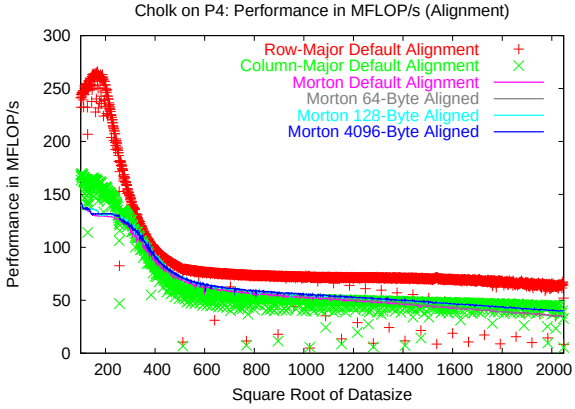
Cholesky-k on Athlon

- L2 and Page-aligned versions perform slightly better than default layout.
- Aligning beyond L2 level does not improve the performance.
- L2-/Page-aligned versions perform as badly as the worst layout throughout all problem sizes.
- Notice the sharp drop in performance of RM and CM implementations.



Cholesky-k on Pentium III

- L2- and page-aligned implementations are signifi cantly faster than default version.
- Aligning Morton arrays to page-level has the same effect as aligning them to L2-level.
- All Morton implementations are worse than or as badly as the CM implementation.



Cholesky-k on Pentium 4

- Alignment improves the default Morton performance only by a small factor.
- For large problem sizes aligning Morton arrays at the L2-level improves the performance slightly. Aligning at page-level has the same effect as aligning at the L2-level.

- Aligning at the L1-level has no effect (i.e. it's performance characteristics are similar to the default).
- Morton layout performs as badly as the CM version.

Figure 5.13: **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 significant sizes in the memory hierarchy (cache line lengths and page size). These results are further improved in Chapter 6, whose results are reported in Figure 6.11.

Chapter 6

Unrolling Loops Over Morton Arrays

This chapter discusses how loops operating over Morton arrays can be unrolled with appropriate strength reduction. This chapter is based on our paper presented at Languages and Compilers for Parallel Computing Workshop (LCPC) 2003 [82].

6.1 Introduction

In the previous chapter, we explored how spatial locality of Morton layout depends on the alignment of the array's base address. In this chapter, we see how loops over Morton arrays can be unrolled and how they have to be aligned to reduce address calculation overhead. Although unrolling is a straightforward process, strength reduction over loops operating over Morton arrays is not. We discuss this in detail in this chapter and we show how unrolling can be effective over Morton arrays, using a small suite of micro-benchmark kernels.

6.2 Unrolling Loops

Let $\mathcal{L}\left(\binom{i}{j}\right)$ be the address calculation function, which returns the offset from the array base address of the array, of an element stored at (i, j), expressed by an index vector $\binom{i}{j}$. Then, for any offset-vector $\binom{k}{l}$, linear layouts have the following property:

$$\mathcal{L}\left(\binom{i}{j}+\binom{k}{l}\right)=\mathcal{L}\binom{i}{j}+\mathcal{L}\binom{k}{l} \quad . \tag{6.1}$$

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

However, with Morton layout the strength reduction is not straightforward when unrolling loops. As stated in Chapter 3, Section 3.4, the Morton address mapping is $\mathcal{S}_{mz}(i, j)=\mathcal{D}_1(i) \mid \mathcal{D}_0(j)$, where "|" denotes bitwise-or, which can be implemented as addition. If given offset k,

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

The problem is that there is no general way of simplifying $\mathcal{D}_0(j+k)$ for all j and all k. Proposition 6.1 simplifies this.

```
double mmijk_unrolled(unsigned sz,double *A,double *B,double *C)
{
   unsigned i,j,k;
   unsigned int t1i, t0j;

   for (i=0;i<sz;i++){
       t1i= T1[i];
       for (j=0;j<sz;j++){
           t0j= T0[j];
           for (k=0;k<sz;k+=4){
               C[ t1i + t0j ] += A[ t1i + T0[k]  ]  * B[ T1[k]   + t0j ];
               C[ t1i + t0j ] += A[ t1i + T0[k+1]]  * B[ T1[k+1] + t0j ];
               C[ t1i + t0j ] += A[ t1i + T0[k+2]]  * B[ T1[k+2] + t0j ];
               C[ t1i + t0j ] += A[ t1i + T0[k+3]]  * B[ T1[k+3] + t0j ];
           }
       }
   }
}
```

(a) Naive unrolled Morton order matrix multiply

```
double mmijk_unrolled(unsigned sz,double *A,double *B,double *C)
{
   unsigned i,j,k;
   unsigned int t1i, t0j;

   for (i=0;i<sz;i++){
       t1i= T1[i];
       for (j=0;j<sz;j++){
           t0j= T0[j];
           for (k=0;k<sz;k+=4){
               C[ t1i + t0j ] += A[ t1i + T0[k]        ] * B[ T1[k] + t0j         ];
               C[ t1i + t0j ] += A[ t1i + T0[k] + T0[1]] * B[ T1[k] + T1[1] + t0j ];
               C[ t1i + t0j ] += A[ t1i + T0[k] + T0[2]] * B[ T1[k] + T1[2] + t0j ];
               C[ t1i + t0j ] += A[ t1i + T0[k] + T0[3]] * B[ T1[k] + T1[3] + t0j ];
           }
       }
   }
}
```

(b) Improved version of unrolled Morton order matrix multiply

```
double mmijk_unrolled(unsigned sz,double *A,double *B,double *C)
{
   unsigned i,j,k;
   unsigned int t1i, t0j;

   for (i=0;i<sz;i++){
       t1i= T1[i];
       for (j=0;j<sz;j++){
           t0j= T0[j];
           for (k=0;k<sz;k+=4){
               C[ t1i + t0j ] += A[ t1i + T0[k]    ] * B[ T1[k] +     t0j ];
               C[ t1i + t0j ] += A[ t1i + T0[k] + 2] * B[ T1[k] + 1 + t0j ];
               C[ t1i + t0j ] += A[ t1i + T0[k] + 4] * B[ T1[k] + 4 + t0j ];
               C[ t1i + t0j ] += A[ t1i + T0[k] + 8] * B[ T1[k] + 5 + t0j ];
           }
       }
   }
}
```

(c) Strength-reduced version of unrolled Morton order matrix multiply

Figure 6.1: **Unrolling Morton-order matrix-multiply implementation using table lookup scheme.** The figure shows unrolled Morton order matrix multiply being strength reduced using table lookup scheme. `T0[.]` corresponds to $\mathcal{D}_0(.)$ and `T1[.]` corresponds to $\mathcal{D}_1(.)$. As can be seen, when strength-reduced, most of the table lookups need to be performed only once per iteration.

Proposition 6.1 (Strength-reduction of Morton address calculation)
Let $u = 2^n$ for some integer $n > 0$. Assume that $j \bmod u = 0$ and that $k < u$. Then,

$$\mathcal{D}_0(j+k) = \mathcal{D}_0(j) + \mathcal{D}_0(k) \quad . \tag{6.2}$$

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

□

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

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

An analogous result holds for the i index. Therefore, by carefully choosing the *alignment* of the starting loop iteration variable with respect to the array indices used in the loop body and by choosing a power-of-two unrolling factor, loops over Morton order arrays can benefit from strength-reduction in unrolled loops.

In our implementation, this means that memory references for the Morton tables are replaced by simple addition of constants. Existing production compilers cannot find this transformation automatically. We therefore implemented this unrolling scheme by hand in order to quantify the possible benefit. We report very promising initial performance results in Section 6.5.

Figure 6.1 illustrates the unrolling with strength reduction for the matrix multiply (MMikj variant) kernel. As we strength reduce, most of the table lookups need to be performed only once per iteration. Figures 6.3–6.6 show the unrolled implementation of our micro-benchmarks.

6.3 Validity of Strength Reduction Over Morton Arrays

Let j_0 be the starting value of a loop. Unrolling a loop with an unrolling factor $u = 2^n$ along with strength reduction technique we have discussed in Section 6.2 is valid only if $j_0 \bmod u = 0$. However, this is only a necessary condition but not sufficient: it does not guarantee that a loop can be unrolled with strength-reduction, as illustrated below.

- If $j_0 \bmod u \neq 0$ (starting value is not aligned at multiple of power-of-two), for example as in Jacobi2D kernels, the unrolling has to be done along with loop peeling. This means that, when the starting value of a loop induction variable is not a multiple of power-of-two, the loop should be split into three loops consisting of a pre-loop, a main-loop and a post-loop. The pre-loop and the post-loop enable the main loop to start and stop at desired values by performing any remaining iterations.

- Strength reduction is easy when index variables are straightforward expressions of loop induction variable. Complicated expressions, consisting of non-constants, may prevent strength reduction. Presence of non-constant terms in index expressions does not guarantee the unrolled loop index to be within bounds.
- Loops with variable stride cannot be unrolled.

One other interesting issue is the issue of loop alignment: In the presence of multiple arrays and/or multiple instances of an array, there exists multiple possible alignments. An example code shown in Figure 6.2 illustrates this.

6.4 Experimental Evaluation

With the same experimental setup and framework discussed in previous chapters (see Section 4.4), we have evaluated the effect of unrolling with appropriate strength reduction on a number of micro-benchmarks. The unrolling factor for the hand-unrolled version of the implementation is 4. For higher unrolling factors, we automated the code generation using the TaskGraph Library [10]. In overall, the data underlying the results presented in this chapter, consist of more than 27 million individual measurements.

6.5 Performance Results

Detailed performance results showing the impact of unrolling with appropriate strength reduction are shown in Figures 6.7–6.11, and, as in the previous chapter, we make some comments directly in the figures.

Results in the previous chapter established the fact that aligning Morton arrays to significant memory boundaries, particularly at the page-level, can improve the performance of the default Morton layout. However, in majority of the cases, the performance improvement was so small and the performance of Morton layout, in general, was disappointing. From the results presented in this chapter, we notice that unrolling the best aligned Morton version (often page-level) has improved the performance very significantly over the basic performance.

In order to evaluate quantitatively whether Morton layout is a good compromise with respect to the better of the two canonical layouts, we establish a metric where the slowdown factor s (with respect to the better of the two) should fall within an acceptable range of $1 \leq s < c$, where c is the acceptable maximum slowdown factor. If we can accept a maximum slowdown factor of 2 ($c = 2$), we can make the following conclusions, categorised according to our experiments:

- For ADI (Figure 6.7), on all platforms, the slowdown with Morton layout is less 2.
- For Jacobi2D (Figure 6.8),
  - On Alpha, the slowdown factor is substantially less than 2.
  - On Sparc, Morton layout performs worse than the row-major layout, offering around 20% of the best speedup (or $s > 2$) for a range of problem sizes between 500×500–1000×1000. For larger problem sizes (beyond 1000×1000), the slowdown factor is slightly less than 2.

```
for (j=1; j<40; ++j){
   C[i][j] + = p*(A[i][j-1] + B[i+1][j+3]);
}
```

(a) Original code (to be unrolled by a factor of 4)

```
for (j=1; j<U; j += 4){
    C[i][D[j-1]+D[1]] + = p*(A[i][D[j-1]]       + B[i+1][D[j+3]]);
    C[i][D[j-1]+D[2]] + = p*(A[i][D[j-1]+D[1]]  + B[i+1][D[j+3]+D[1]]);
    C[i][D[j-1]+D[3]] + = p*(A[i][D[j-1]+D[2]]  + B[i+1][D[j+3]+D[2]]);
    C[i][D[j+3]]      + = p*(A[i][D[j-1]+D[3]]  + B[i+1][D[j+3]+D[3]]);
}
```

(b) Unrolled loop aligned at $j = 1$. Aligning at $j = 1$ requires 2 table lookups per iteration (`D[j-1]` and `D[j+3]`)

```
for (j=2; j<U; j += 4){
    C[i][D[j-2]+D[2]] + = p*(A[i][D[j-2]+ D[1]]    + B[i+1][D[j+2]+D[1]]);
    C[i][D[j-2]+D[3]] + = p*(A[i][D[j-2]+ D[2]]    + B[i+1][D[j+2]+D[2]]);
    C[i][D[j+2]]      + = p*(A[i][D[j-2]+ D[3]]    + B[i+1][D[j+2]+D[3]]);
    C[i][D[j+2]+D[1]] + = p*(A[i][D[j+2]      ]    + B[i+1][D[j+6]]);
}
```

(c) Unrolled loop aligned at $j = 2$. Aligning at $j = 2$ requires 3 table lookups per iteration (`D[j-2]`, `D[j+2]`, and `D[j+6]`)

```
for (j=4; j<40; j += 4){
    C[i][D[j]]      + = p*(A[i][D[j-1]]    +  B[i+1][D[j]+D[3]]);
    C[i][D[j]+D[1]] + = p*(A[i][D[j]]      +  B[i+1][D[j+4]]);
    C[i][D[j]+D[2]] + = p*(A[i][D[j]+D1]]  +  B[i+1][D[j+4]+D[1]]);
    C[i][D[j]+D[3]] + = p*(A[i][D[j]+D[2]] +  B[i+1][D[j+4]+D[2]]);
}
```

(d) Unrolled loop aligned at $j = 4$. This version also requires 3 table lookups per iteration (`D[j]`, `D[j-1]` and `D[j+4]`)

Figure 6.2: **Unrolling and loop alignment.** When unrolling loops, there may exist multiple alignment points to which the loop startup variable should be aligned. The figure illustrates such a case using an example code given in Figure 6.2(a). Here, we consider only the innermost loop. Aligned and unrolled versions of the loop are shown in Figures 6.2(b), 6.2(c) and 6.2(d). The number of table lookups vary depending on the alignment point of the loop. An optimal alignment point should minimise the number of table lookups per iteration. In all cases, `D[.]` denotes the appropriate table lookup.

```
for (t=0;t<nIters;t++){
    for (i=1;i<sz;++i){
        for (j=0; j< sz; ++j)
          A[T1[i] + T0[j]]     += A[T1[i-1] + T0[j]];
    for (i=0;i<sz;++i){
      for (j = 1; j< sz; ++j){
          A[T1[i] + T0[j]]     += A[T1[i] + T0[j-1]];
}/*end of time loop*/
```

```
for (t=0;t<nIters;t++){
    for (i=1;i<sz;++i){
        const unsigned int t1i = T1[i];
        const unsigned int t1im1 = T1[i-1];
        for (j=0; j+3 < sz; j += 4){
          const unsigned int t2j = T0[j];
          A[t1i + t2j]     += A[t1im1 + t2j];
          A[t1i + t2j + 1] += A[t1im1 + t2j + 1];
          A[t1i + t2j + 4] += A[t1im1 + t2j + 4];
          A[t1i + t2j + 5] += A[t1im1 + t2j + 5];
        }
        for (m = j; m < sz; ++m){
          const unsigned int t2m = T0[m];
          A[t1i + t2m]     += A[t1im1 + t2m];
        }
     }

    for (i=0;i<sz;++i){
      const unsigned int t1i = T1[i];
      for (j = 1; j+3 < sz; j += 4){
          const unsigned int t2jm1 = T0[j-1];
          unsigned int t2jp3 = T0[j+3];
          A[t1i + t2jm1 + 1]     += A[t1i + t2jm1];
          A[t1i + t2jm1 + 4]     += A[t1i + t2jm1 + 1];
          A[t1i + t2jm1 + 5]     += A[t1i + t2jm1 + 4];
          A[t1i + t2jp3  ]       += A[t1i + t2jm1 + 5];
      }
      for (m = j; m < sz; ++m){
          const unsigned int t2mm1 = T0[m-1];
          unsigned int t2m = T0[m];
          A[t1i + t2m]     += A[t1i + t2mm1];
      }
    }
}/*end of time loop*/
```

Figure 6.3: **Unrolled Morton-order ADI implementation using table lookup scheme.** The figure shows the original implementation of Morton-order ADI kernel (top) and a strength-reduced implementation of the unrolled ADI kernel (bottom). Both the implementations use the table lookup scheme. In the figure, `T0[.]` corresponds to $\mathcal{D}_0(.)$ and `T1[.]` corresponds to $\mathcal{D}_1(.)$. Notice that the main loop has been peeled in order to enable alignment at the start of loop iteration.

```
for (k=0;k<sz;k++){
    A[T1[k]+T0[k]] = sqrt(A[T1[k]+T0[k]]);
    for (i=k;i<sz;i++){
      A[T1[i]+T0[k]] /= A[T1[k]+T0[k]];
      for (j = k; j < i; ++j){
        A[T1[i]+T0[j]]   -= A[T1[i]+T0[k]] * A[T1[j]+T0[k]];
      }
    }
}
```

```
for (k=0;k<sz;k++){
    const unsigned int t1k = T1[k];
    const unsigned int t2k = T0[k];
    A[t1k+t2k] = sqrt(A[t1k+t2k]);
    for (i=k;i<sz;i++){
      const unsigned int t1i = T1[i];
      A[t1i+t2k] /= A[t1k+t2k];

      /* Pre-Loop */
      const unsigned int j1_end = ((k>>2)<<2);
      for (j1 = k; j1 < j1_end; j1++){
        const unsigned int t1j1 = T1[j1];
        const unsigned int t2j1 = T0[j1];
        A[t1i+t2j1]   -= A[t1i+t2k] * A[t1j1+t2k];
      }

      for (j2 = j1; j2+3 < i; j2 +=4){
        const unsigned int t1j2 = T1[j2];
        const unsigned int t2j2 = T0[j2];
        A[t1i+t2j2]   -= A[t1i+t2k] * A[t1j2+t2k];
        A[t1i+t2j2+1] -= A[t1i+t2k] * A[t1j2+2+t2k];
        A[t1i+t2j2+4] -= A[t1i+t2k] * A[t1j2+8+t2k];
        A[t1i+t2j2+5] -= A[t1i+t2k] * A[t1j2+10+t2k];
      }

      for (j3 = j2; j3 < i; j3 ++){
        const unsigned int t1j3 = T1[j3];
        const unsigned int t2j3 = T0[j3];
        A[t1i+t2j3]   -= A[t1i+t2k] * A[t1j3+t2k];
      }


    }
  }
```

Figure 6.4: **Unrolled Morton-order Cholesky-K implementation using table lookup scheme.** Strength-reduced version of the unrolled Morton-order Cholesky (K-Variant) implementation using table lookup scheme (bottom). We show the original implementation of the same kernel (top) for ease of reference. In the figure, `T0[.]` corresponds to $\mathcal{D}_0(.)$ and `T1[.]` corresponds to $\mathcal{D}_1(.)$. Notice that the main loop has been peeled in order to enable alignment at the start of loop iteration.

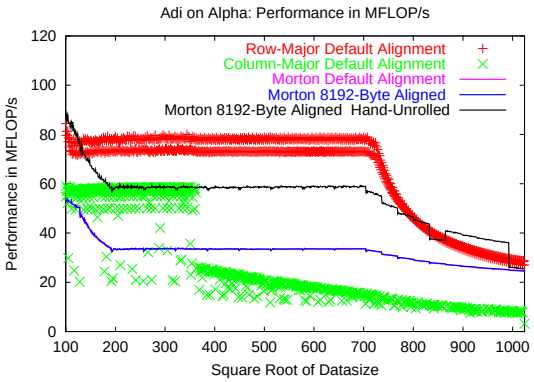
```
for ( it = 0; it < nIters; it++ ) {
   for ( i = 1; i < sz - 1; i++ ){
       const unsigned int t1i   = T1[i];
       const unsigned int t1im1 = T1[i-1];
       const unsigned int t1ip1 = T1[i+1];
       for ( j = 1; j < j_end; j += 4 ){
           const unsigned int t0jm1 = T0[j-1];
           const unsigned int t0jp3 = T0[j+3];
           B[t1i + (t0jm1 + 1)] = 0.25 * ( A[t1im1 + (t0jm1 + 1)] +  A[t1ip1 + (t0jm1 + 1)] +
                                           A[t1i   + (t0jm1)    ] +  A[t1i   + (t0jm1 + 4)] );
           B[t1i + (t0jm1 + 4)] = 0.25 * ( A[t1im1 + (t0jm1 + 4)] +  A[t1ip1 + (t0jm1 + 4)] +
                                           A[t1i   + (t0jm1 + 1)] +  A[t1i   + (t0jm1 + 5)] );
           B[t1i + (t0jm1 + 5)] = 0.25 * ( A[t1im1 + (t0jm1 + 5)] +  A[t1ip1 + (t0jm1 + 5)] +
                                           A[t1i   + (t0jm1 + 4)] +  A[t1i   + (t0jp3)    ] );
           B[t1i + (t0jp3)    ] = 0.25 * ( A[t1im1 + (t0jp3)    ] +  A[t1ip1 + (t0jp3)    ] +
                                           A[t1i   + (t0jm1 + 5)] +  A[t1i   + (t0jp3 + 1)] );
       }
       for ( j = j_end; j < sz - 1; j++ ){
           const unsigned int t0j   = T0[j];
           const unsigned int t0jm1 = T0[j-1];
           const unsigned int t0jp1 = T0[j+1];
           B[t1i + t0j] = 0.25 * ( A[t1im1 + t0j  ] + A[t1ip1 + t0j  ] +
                                   A[t1i   + t0jm1] + A[t1i   + t0jp1] );
       }
   }

   it++;
   if (it >= nIters) break;

   for ( i = 1; i < sz - 1; i++ ){
       const unsigned int t1i   = T1[i];
       const unsigned int t1im1 = T1[i-1];
       const unsigned int t1ip1 = T1[i+1];

       for ( j = 1; j < j_end; j += 4 ){
           const unsigned int t0jm1 = T0[j-1];
           const unsigned int t0jp3 = T0[j+3];
           A[t1i + (t0jm1 + 1)] = 0.25 * ( B[t1im1 + (t0jm1 + 1)] + B[t1ip1 + (t0jm1 + 1)] +
                                           B[t1i   + (t0jm1)    ] + B[t1i   + (t0jm1 + 4)] );
           A[t1i + (t0jm1 + 4)] = 0.25 * ( B[t1im1 + (t0jm1 + 4)] + B[t1ip1 + (t0jm1 + 4)] +
                                           B[t1i   + (t0jm1 + 1)] + B[t1i   + (t0jm1 + 5)] );
           A[t1i + (t0jm1 + 5)] = 0.25 * ( B[t1im1 + (t0jm1 + 5)] + B[t1ip1 + (t0jm1 + 5)] +
                                           B[t1i   + (t0jm1 + 4)] + B[t1i   + (t0jp3)    ] );
           A[t1i + (t0jp3)    ] = 0.25 * ( B[t1im1 + (t0jp3)    ] + B[t1ip1 + (t0jp3)    ] +
                                           B[t1i   + (t0jm1 + 5)] + B[t1i   + (t0jp3 + 1)] );
       }
       for ( j = j_end; j < sz - 1; j++ ){
           const unsigned int t0j   = T0[j];
           const unsigned int t0jm1 = T0[j-1];
           const unsigned int t0jp1 = T0[j+1];
           A[t1i + t0j] = 0.25 * ( B[t1im1 + t0j  ] + B[t1ip1 + t0j  ] +
                                   B[t1i   + t0jm1] + B[t1i   + t0jp1] );
       }
   }
}
```

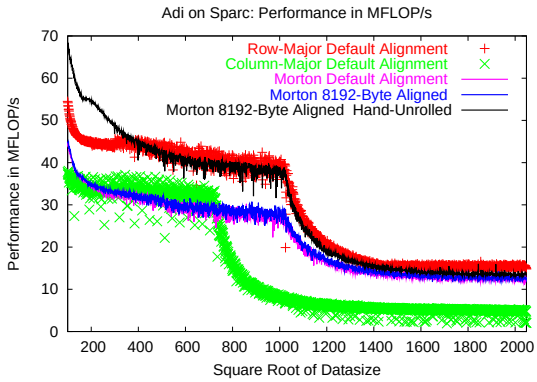
Figure 6.5: **Unrolled Morton-order Jacobi2D implementation using table lookup scheme.** Strength-reduced version of the unrolled Morton-order Jacobi2D implementation using table lookup scheme. Please see the original version in Figure 4.5 of Chapter 4. In this figure, `T0[.]` corresponds to $\mathcal{D}_0(.)$ and `T1[.]` corresponds to $\mathcal{D}_1(.)$. Notice that the main loop has been peeled at the tail of the loop by introducing a post-loop (no pre-loop) in order to enable alignment at the start of loop iteration. Also, the main loop is aligned at 0 though the loop starting value is 1, with an additional table-lookup for an index which crosses the index boundary.

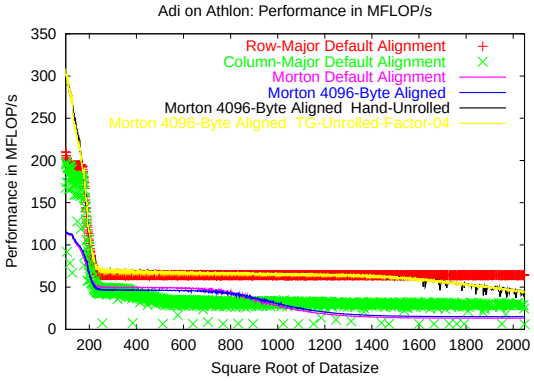
```
for( i = 0; i < sz; ++i ){
   for( k = 0; k < sz; ++k ){
      const double r = A[T1[i] + T0[k]];
      for( j = 0; j < sz; ++j){
        C[T1[i] + T0[j]]     += r * B[T1[k] + T0[j]];
      }
   }
}
```

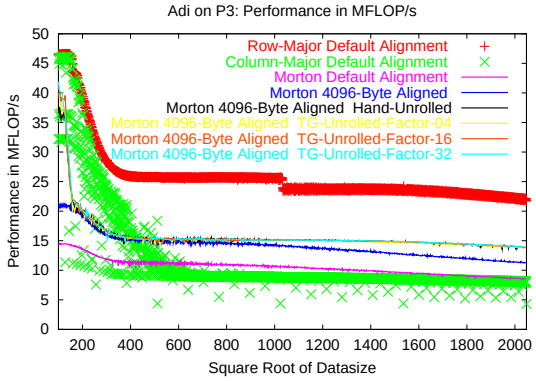
```
const unsigned int j_end   = sz - mod( sz, 4 );
for( i = 0; i < sz; ++i ){
   const unsigned int t1i = T1[i];
   for( k = 0; k < sz; ++k ){
      const unsigned int t1k = T1[k];
      const unsigned int t0k = T0[k];
      const double r      = A[t1i + t0k];

      for( j = 0; j < j_end; j += 4 ){
        const unsigned int t0j   = T0[j];
        const unsigned int c_idx = t1i + t0j;
        const unsigned int b_idx = t1k + t0j;

        C[c_idx]     += r * B[b_idx];
        C[c_idx + 1] += r * B[b_idx + 1];
        C[c_idx + 4] += r * B[b_idx + 4];
        C[c_idx + 5] += r * B[b_idx + 5];
      }

      for( j = j_end; j < sz; ++j ){
        const unsigned int t0j   = T0[j];
        const unsigned int c_idx = t1i + t0j;
        const unsigned int b_idx = t1k + t0j;
        C[c_idx]     += r * B[b_idx];
      }

    }
  }
```

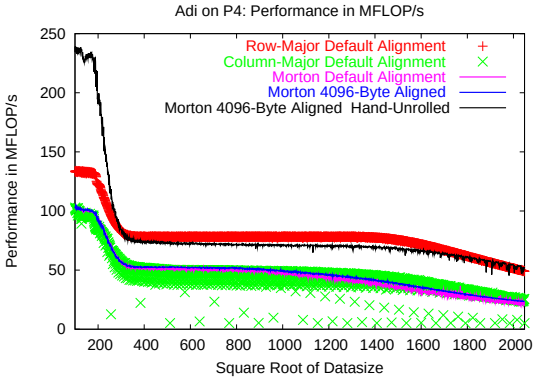
Figure 6.6: **Unrolled Morton-order MMikj implementation using table lookup scheme** The figure shows the original implementation of MMikj kernel (top) and a corresponding strength-reduced version of the unrolled Morton-order implementation. Both the kernels use the table lookup scheme. In the figure, `T0[.]` corresponds to $\mathcal{D}_0(.)$ and `T1[.]` corresponds to $\mathcal{D}_1(.)$. Notice that the main loop has been peeled and a post-loop has been introduced to cover the remaining iterations.







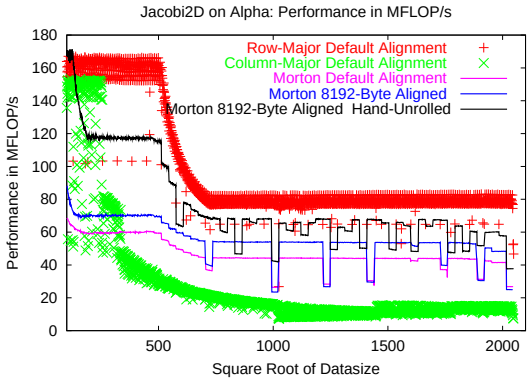


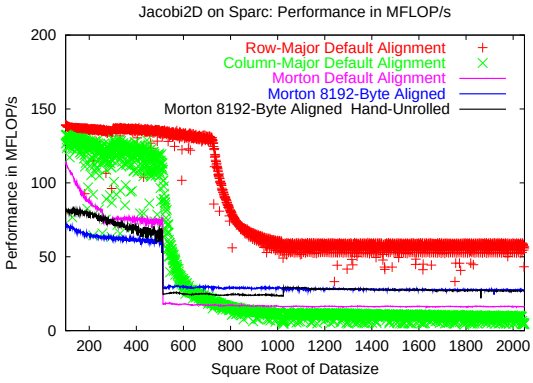


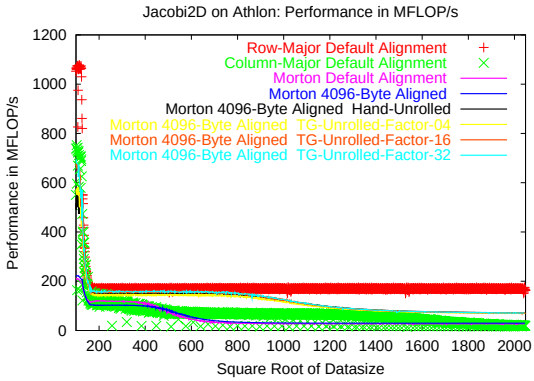
- Unrolling signifi cantly improves performance of the best aligned Morton array on nearly all platforms.
- Notice for Alpha, the upper limit is 1024×1024.

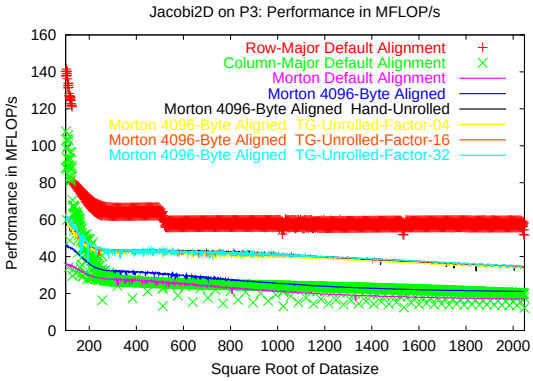
- For Alpha (Sun), the fall-off in RM performance occurs at 725×725 (1024×1024) when the total datasize exceeds L2 cache size of 4MB (8MB), direct mapped. This assumes a working set of 725×725 (1024×1024) doubles.
- Performance drop on Pentium 4 and Athlon around 1024×1024 broadly agrees with our hypothesis in Section 7.3.1, where the row-length exceeds the limits of conflict-free region.
- In summary, for ADI, Morton layout is a good compromise between row- and column-major layouts, on nearly all platforms. However, arguably, P3 may be an exception, where for small data sizes, canonical layouts are faster than Morton layout.

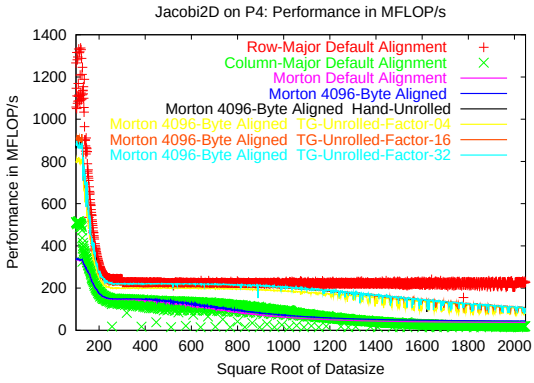
Figure 6.7: **ADI performance in MFLOPs on different platforms.** We compare row-major, column-major, Morton with default alignment of the base address of the array, Morton with page-aligned base address and unrolled-Morton with page-aligned base address and factor 4 loop unrolling.







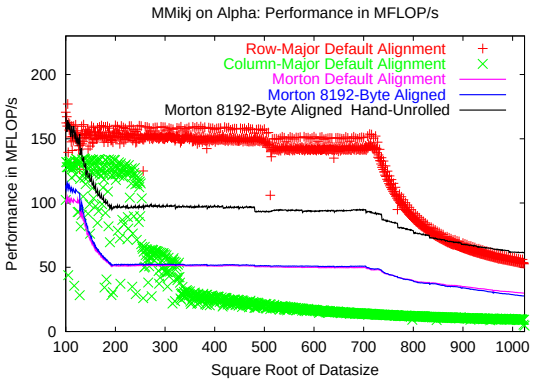


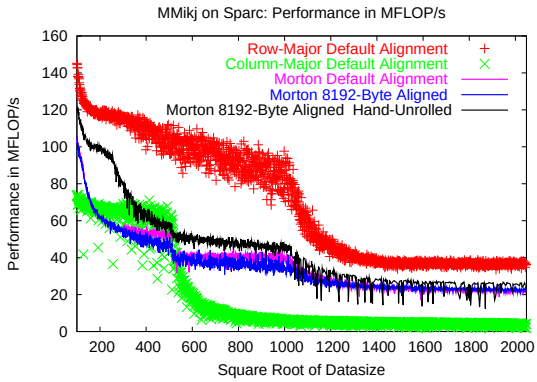


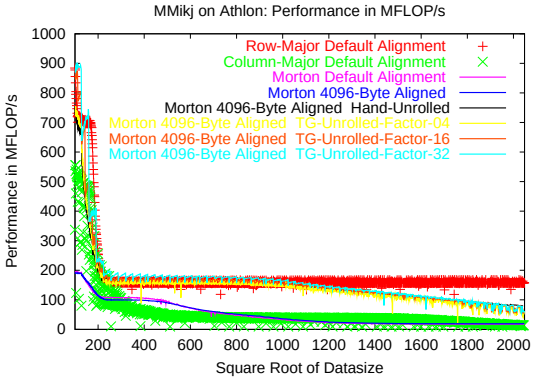
- Unrolling improves performance of the best aligned Morton implementation, in particular on x86 where the unrolled Morton performance is within reach of the best canonical.

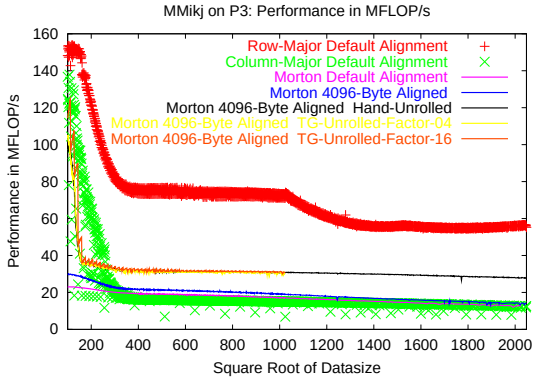
- Performance drop on Pentium 4 and Athlon around half of 1024×1024 ((Jacobi2D uses two arrays) broadly agrees with our hypothesis in Section 7.3.1, where the row-length exceeds the limits of conflict-free region.
- In summary, for Jacobi2D, Morton layout is an attractive compromise, depending on the platform and problem size. On Sparc, Morton layout is a useful compromise for problem sizes larger than 600×600 where the crossover occurs.

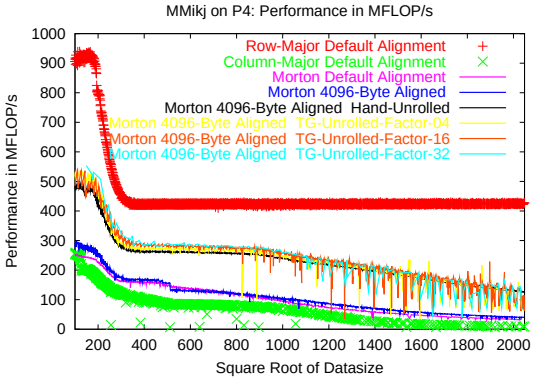
Figure 6.8: **Jacobi2D performance in MFLOPs on different platforms.** We compare row-major, column-major, Morton with default alignment of the base address of the array, Morton with page-aligned base address and Morton with page-aligned base address and factor 4 loop unrolling.







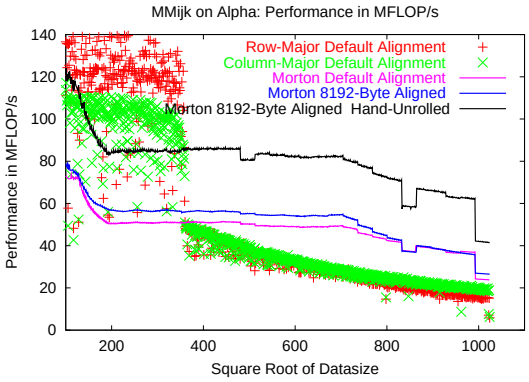


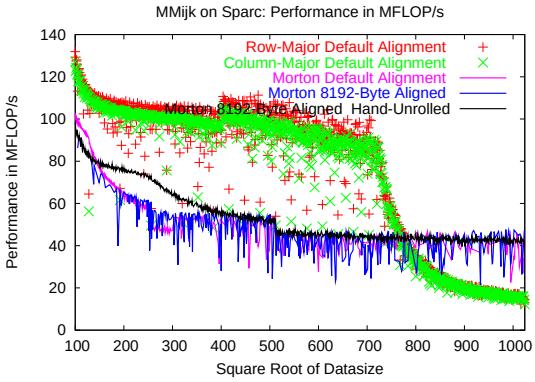


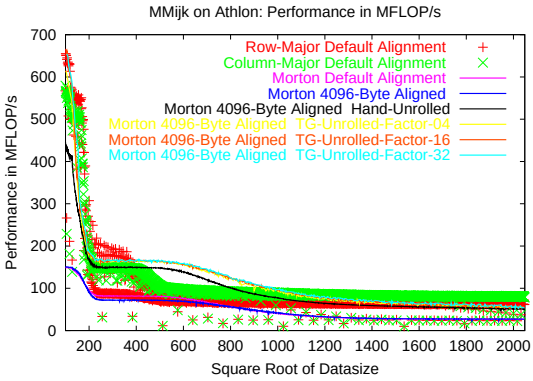
- Unrolling improves the performance of best aligned Morton implementations.
- On x86 platforms, high unrolling factors makes the Morton performance very comparable to the row-major performance, especially on Athlon.

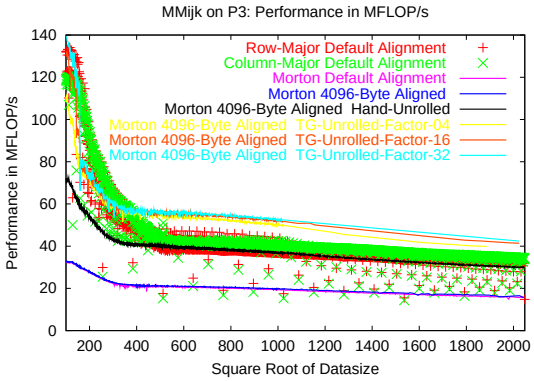
- For Alpha, notice that upper limit is 1024×1024.
- Performance drop on Pentium 4 and Athlon around third of 1024×1024 (MMikj uses three arrays) broadly agrees with our hypothesis in Section 7.3.1, where the row-length exceeds the limits of conflict-free region.
- In summary, Morton layout is an attractive compromise between row-major and column-major layouts, on nearly all platforms.

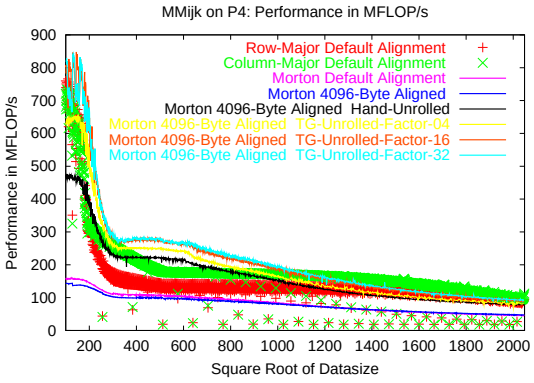
Figure 6.9: **MMikj performance in MFLOPs on different platforms.** We compare row-major, column-major, Morton with default alignment of the base address of the array, Morton with page-aligned base address and Morton with page-aligned base address and factor 4 loop unrolling.







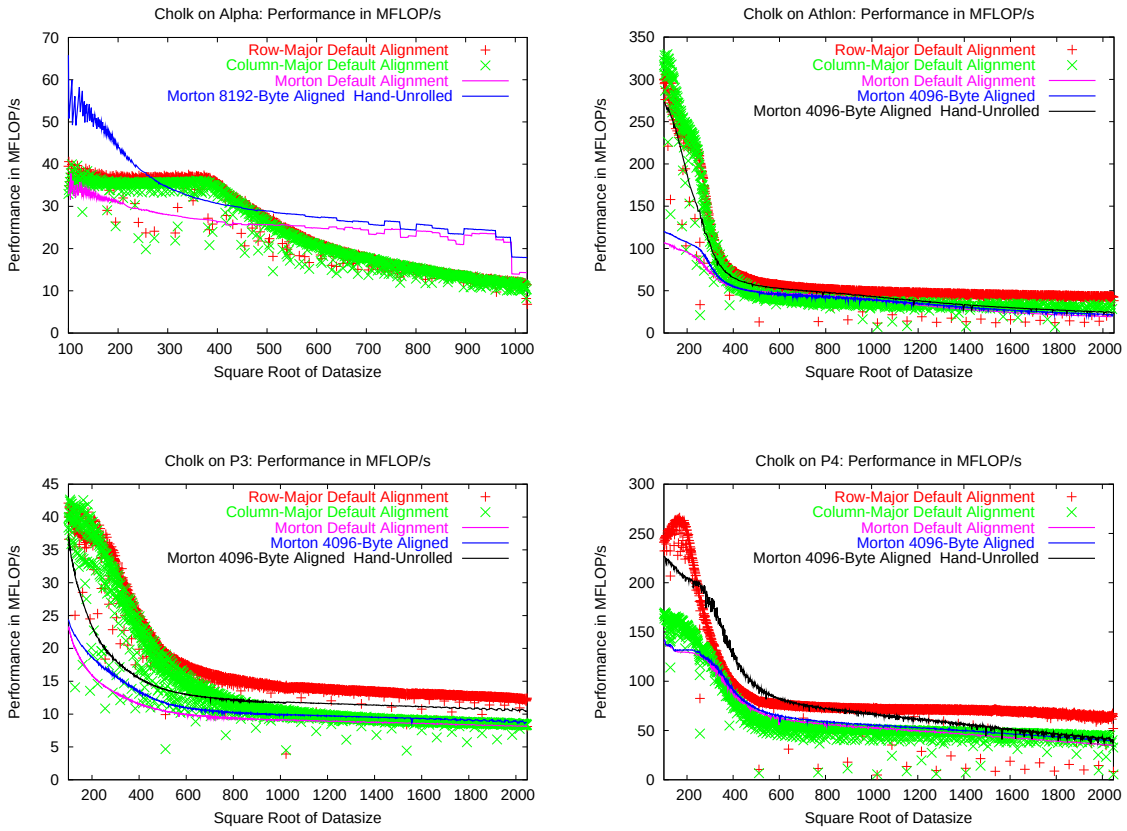




- Unrolled and page-aligned Morton is considerably faster than default on most platforms.
- For Alpha, notice that the upper limit is 1024×1024.
- Notice the sharp drop in RM and CM performance on Alpha (around 360×360) and on Sparc (around 700×700) platforms .

- Performance drop on Pentium 4 and Athlon around thirds of 1024×1024 (MMijk uses three arrays) broadly agrees with our hypothesis in Section 7.3.1, where the row-length exceeds the limits of conflict-free region.
- In most of the cases, Morton layout is a compromise between row-major and column-major layouts except on Alpha and Sparc.

Figure 6.10: **MMijk performance in MFLOPs on different platforms.** We compare row-major, column-major, Morton with default alignment of the base address of the array, Morton with page-aligned base address and Morton with page-aligned base address and factor 4 loop unrolling.



- Unrolling improves the performance of the best aligned Morton version considerably.
- On Pentium III and Pentium 4, the speedup gain is sustained for larger range of problem sizes than on Athlon.
- For Alpha, notice that the upper limit is 1024×1024.
- Notice the sharp drop in RM and CM performance on Alpha (around 400×400).
- Except on Alpha and Pentium III, the worst canonical layout (column-major, in our case) performs better than or even closer to the performance of unrolled, best aligned Morton implementation, for larger problem sizes. Although the slowdown compared to the best canonical layout is smaller, the slowdown factor is as badly as that of the worst canonical layout. On Alpha, Morton layout performs better than both the canonical layouts starting from problem size around 500×500. On Pentium III, Morton layout is a compromise starting from problem size around 800×800.
- In summary, Morton layout can be a compromise layout on restricted set of platforms and never performs worse than the column-major layout.

Figure 6.11: **Cholesky-k performance in MFLOPs on different platforms.** We compare row-major, column-major, Morton with default alignment of the base address of the array, Morton with page-aligned base address and Morton with page-aligned base address and factor 4 loop unrolling.

  - On Athlon, Pentium III and Pentium 4, the slowdown factor is less than 2. However, on Pentium 4, for problem sizes larger than 1800×1800, the slowdown factor is only slightly larger than 2.

- For MMikj (Figure 6.9),
  - On Alpha, the worst slowdown factor is substantially less than 2.
  - On Sparc, for larger problem sizes (larger than 1200×1200), the slowdown factor is less than 2. However, for a range of problem sizes (500×500–1000×1000), the slowdown factor can vary as the variation in the RM performance is very high. The average slowdown factor is less than 2.
  - On Pentium III, for all problem sizes up to 1100×1100, Morton layout runs only about 40% as fast as the best layout (or $s > 2$). For problem sizes beyond 1100×1100, the slowdown factor less than 2.
  - On Athlon and Pentium 4, the worst slowdown factor is less than 2 for all problem sizes up to 1800×1800. For larger problem sizes (larger than 1800×1800), the slowdown factor is larger than 2.
- For MMijk (Figure 6.10), on all platforms the slowdown factor is less than 2.
- For Cholesky-k (Figure 6.11), the slowdown is less than 2 on nearly all platforms. On Alpha, Morton version is faster than both the canonical layouts for problem sizes larger than 500×500.

In other words, summarising architecture-by-architecture:

- On Alpha, for all benchmark kernels, Morton layout is a good compromise with respect to the better of the two canonical layouts.
- On Sparc, Morton layout is guaranteed to be a good compromise only for the ADI and MMikj kernels. Morton layout falls above the maximum slowdown of 2 on some subranges of problem sizes, for the Jacobi2D and MMijk kernels. However, for some restricted range of problem sizes, Morton layout is a good compromise.
- On Athlon, Morton layout is a good compromise.
- On Pentium III, except for the MMikj kernel, Morton layout is a good compromise. Even with the MMikj kernel, Morton layout fails only for a small subrange of problem sizes.
- On Pentium 4, except for the MMikj and Jacobi2D kernels, Morton layout is a good compromise. On failed benchmarks, Morton layout fails to offer the acceptable slowdown factor for small range of problem sizes.

Our evaluation of Morton layout with the best possible alignment and unrolling points out that Morton layout is a good compromise to the better of the two canonical layouts on nearly all platforms, with very few exceptions. These exceptions occur on Sparc, Pentium III and on Pentium 4 platforms on subranges of problem sizes of different kernels.

6.6 Conclusions

By inspecting the assembly code, we established that at least the `icc` compiler on x86 architectures does automatically unroll our benchmark kernels for row-major layout. Our results in this chapter show that unrolling the loops over Morton arrays, using the technique described in Section 6.2, can result in a significant performance improvement of the Morton code: On several architectures, the unrolled Morton codes are, for part of the spectrum of problem sizes, very close to, or even better than, the performance of the best canonical code. We have done a preliminary investigation on the effect of larger unrolling factors on performance and the results are promising.

In summary, we improved the performance of basic Morton scheme (whose results are presented in Chapter 4) by aligning the starting address of the Morton arrays (corresponding results are in Chapter 5). We further improved the performance by loop unrolling with strength reduction which partly minimised the address computation overheads. Results presented in this chapter are the best obtained results for these benchmarks when using Morton layout. Our evaluation of Morton layout on number of different platforms, using a suite of benchmarks has shown that Morton layout can be an attractive and promising compromise and even an alternative layout to canonical layouts, with few exceptions.

In this chapter and in the chapters so far, we have evaluated the performance basic Morton layout and improved the performance by two simple optimisation techniques. These steps can be considered as a pre-cursor to incorporating support for Morton layout in compilers. In the next chapter, we describe the design issues for implementing a prototype compiler to support Morton layout in real Fortran programs.

Chapter 7

Conclusions and Directions for Further Work

In this chapter, we review the contributions of this thesis and we discuss the design considerations for the implementation of a prototype compiler to support Morton layout in scientific applications. Following this, we point out a number of possible directions for future research where we could extend the work presented in this thesis.

7.1 Review of the Contributions of this Thesis

The goal of this thesis was to analyse alternative array storage layouts for regular scientific programs. As presented in previous chapters, we found that recursive array layouts, Morton layouts in particular, are an attractive alternative to canonical layouts. As evidenced by our experimental results, on suitable hardware platforms, we demonstrated that Morton layout can be an attractive compromise storage layout compared with canonical layouts, provided that the techniques developed in this thesis are used. Towards this, the thesis has made the following contributions:

- We have reviewed existing techniques for locality optimisation under three different categories.

  Most importantly, we paid particular attention to recursive and non-linear array layouts and how both iteration space and data layout techniques can be integrated for better results. The valuable part of this survey is to point out the current state of the optimisation techniques and how data-layout transformations can aid in optimisation.

- We have exhaustively evaluated the Morton layout over a full range of problem sizes, on various representative architectures for a suite of benchmark kernels.

  As pointed out in the earlier chapters, Morton layout is only applicable to arrays whose sizes are a power-of-two. However, by padding the rows and columns, we were able to apply the Morton layout to non-power-of-two array sizes. With this technique, we exhaustively evaluated the Morton layout. Such an exhaustive evaluation has enabled us to comprehensively observe and analyse performance of Morton layout in a selected set of scientific benchmark kernels. Further, this evaluation has also demonstrated the effectiveness of the lookup-table scheme compared to the original dilated arithmetic scheme.

- Performance of the straightforward implementation of Morton layout is often poor. In Chapters 5 and 6, we have illustrated and evaluated how simple optimisations can improve the performance of the basic Morton scheme by a significant factor.

  - As discussed in Chapter 5, aligning the base address of Morton arrays significantly improved the performance of basic Morton layout. Aligning Morton arrays at significant words corresponding to levels of the memory hierarchy has improved the performance in a majority of the architecture/benchmark pairs. We often found that aligning to the largest significant sizes of the memory hierarchy (page-level) guarantees improved performance, though the same could be achieved by aligning the arrays to one of the levels below, in some cases.
  - Then, in Chapter 6, we suggested a technique to unroll the loops with Morton array accesses with appropriate strength reduction. We found that unrolling with strength reduction has always improved the performance of basic Morton layout, often dramatically.

  Conventional compilers, due to lack of information passed about the underlying storage layout, fail to perform these optimisations.

- We have evaluated the hypothesis that Morton layout is a good compromise between row-major and column-major layouts.

  Our best results, in Chapter 6, have shown that Morton layout is often a good compromise between row-major and column-major layouts, with few exceptions.

We view our contributions presented in this thesis as a pre-cursor to incorporating support for alternative storage layouts, especially for hierarchical storage layouts, in compilers. Our results are promising, however restricted to micro-benchmarks and one of the hierarchical storage layouts. In order to evaluate the effectiveness of hierarchical layouts in real scientific applications, such compiler support is necessary. A prototype compiler could provide a framework for generalising and automating the addressing and optimisations for different hierarchical storage layouts. In relation to this, we present our initial findings and plans in the design of a prototype compiler, in the next Section.

7.2 Design Considerations for a Compiler to Support Morton Layout

It is essential to use an existing compiler infrastructure like ROSE [69] or Stanford University Intermediate Format (SUIF-1) [88] to limit the investment in time and resources. Since the internal functionalities and mechanisms of these frameworks may vary very greatly, exact implementation of the prototype compiler is entirely dependent on the underlying framework. Our design considerations are generic as much as possible so that the techniques could be implemented using any of the available compiler frameworks.

7.2.1 Structure of the Prototype Compiler

The original source code is converted into an intermediate format supported by the underlying compiler framework. Then transformations to support Morton layout are applied to this intermediate

representation. The corresponding source code is then generated with the help of the framework. Figure 7.1 illustrates the overall structure of the prototype compiler.

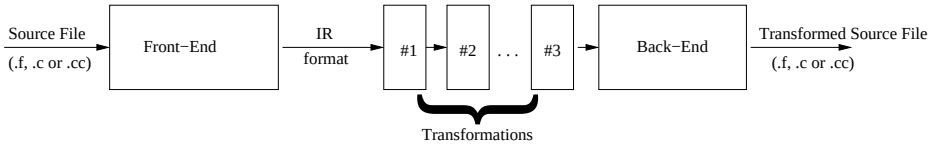


Figure 7.1: **General Structure of the Prototype Compiler to support Morton layout in scientific programs** The Figure shows the overall structure of the prototype compiler, assuming an underlying compiler framework is used. The source program is converted to an intermediate representation (IR) supported by the underlying compiler infrastructure using a front–end. Different transformations to support Morton layout are then applied to this intermediate format in a preferred order. Finally, the transformed intermediate format is translated to a source form (either C/C++ or Fortran format), using a back–end tool.

The front-end of the underlying compiler infrastructure translates the source file into an intermediate representation that the chosen infrastructure supports. For example, in SUIF-1, the `scc` tool is a front-end to translate C and Fortran sources to SUIF-1 intermediate format. The ROSE [69] framework, for example, uses Edison Design Group's [24] front-end to handle C++ sources.

The intermediate representation of a source program is the basis for developing compiler transformations — transformations to support hierarchical storage layouts in our context. The intermediate representation should retain almost all the high-level information from the source code such that both high-level program-restructuring transformations and low-level analyses and optimisations can be supported. Different compiler frameworks usually have at-least one intermediate format and should support plugging transformations to manipulate the intermediate representation of the program.

For example, in SUIF, this intermediate representation is a mixed-mode program representation for fine grain control of transformations — supporting low-level instructions, basic blocks and trees of blocks. This representation is based on an object oriented class library, but source language independent. The ROSE framework uses Edison Design Group's [24] intermediate representation and their own representation (as AST). Both the frameworks support plugging transformations together to operate on corresponding intermediate representations.

The back-end is then used to generate a source program from the intermediate format so that it could be compiled using one of the vendor supplied compilers. SUIF-1, for example, has a back-end to generate sources for C and Fortran. The ROSE supports generation of C++ source code.

7.2.2 Transformations to Support Morton Layout

To implement hierarchical storage layouts in real Fortran or C++ programs, we need to convert existing multi-dimensional (most specifically two-dimensional arrays) row- or column-major arrays to a desired member of the hierarchical storage layouts. A transformation pass is necessary to achieve this. Additional passes or transformations are required to support optimisations for a chosen layout. Required transformations to perform these operations are discussed below.

Converting to Morton Layout

When converting row- or column-major arrays to Morton arrays, the following operations should be performed:

- **Flattening**: Although Morton arrays are viewed as two-dimensional arrays, from the implementation point of view, they are one-dimensional arrays. For this reason, declaration of existing two-dimensional arrays should be flattened. This involves modifying corresponding symbol table entries and procedure signatures. Symbol table entries for flattened arrays are altered to reflect the fact that their dimensions and size have changed. Similarly, subroutine signatures need to be modified to conform with the typing rules.
- **Padding**: Morton layout is available only to power-of-two size matrices. If the original array size is not a power-of-two (sizes in each dimension), the array should be padded to the next power-of-two size in addition to flattening. Again, corresponding entries in the symbol table should be modified.
- **Addressing**: As illustrated in Chapter 3, Morton order addressing is a complicated process. We demonstrated that table lookup scheme offers an easier implementation solution at the cost of slight performance penalty, compared to the dilated arithmetic scheme. Further, when implemented with unrolling, the advantages obtained from the dilated arithmetic scheme could easily be matched. This necessitates lookup tables to be setup (i.e. to be declared, entered in to symbol tables) and filled up with correct values. Using a lookup table scheme has an additional flexibility of mixing storage layouts and this requires separate lookup tables for each array. If the layout of all arrays is consistent throughout the program, a common set of tables could be used for all arrays. An important phase of addressing is that all two-dimensional array references need to be re-indexed using the appropriate addressing scheme, for example for Z-Morton layout using the scheme illustrated in Chapter 3.

In summary, these set of transformations, when combined, should convert all existing two-dimensional arrays to Morton arrays with appropriate re-indexing and lookup tables.

Unrolling with Strength Reduction

This is one of the key transformations to improve the performance of basic Morton layout and amortise the cost of addressing. In order to unroll Morton loops with appropriate strength reduction, following steps are required:

- **Identify Candidate Loops**: When unrolling loops over Morton arrays, innermost loops should be considered as candidate loops for unrolling. However, the unrolling can be restricted only to the loops with Morton array accesses.
- **Loop Alignment**: Aligning the starting value of the loop to a power-of-two value (see Section 6.3) is the next step in the transformation. In order to do this, it may be necessary to do loop peeling, converting the original loop into three loops: a pre-loop, a main loop and a post-loop. The pre-loop and post-loop should cover remaining iterations and should ensure the loop alignment of the main loop. However, if the step value of the loop is not a constant or if it is a non-power-of-two value, then it is not possible to align the loop at power-of-two values.

- **Unroll/Strength Reduce**: Once the loop is aligned, the loop body should be replicated with appropriate re-indexing. Further, when necessary conditions are satisfied, the strength reduction should be be applied (as in Chapter 6) with the help of pre-calculated lookup tables. The strength reduction should be applied, only in the presence of Morton arrays.

Transformations discussed above are the minimal set of transformations required to support Morton layout, as discussed in the previous chapters of this thesis. Having a prototype compiler with these set of transformations should enable the effectiveness of Morton layout in real scientific programs to be tested.

7.3 Further Directions for Future Work

The work described in this thesis analysed alternative array layouts for regular scientific programs and demonstrated the concept using one of the hierarchical layouts — Morton layout. We have also discussed how well these layouts match with the hierarchical memory model present in modern machines. However, the lack of support from compilers, as evidenced by our experimental results, and the lack of support from performance tools prevent the widespread usage of these layouts. One of the key reasons for lack of support for these layouts is that performance characteristics of these layouts are not well known. Contemporary performance tools do not exploit any context information specific to these layouts. We argue that our work can be used and extended to provide support for these layouts in scientific programs. Most specifically, we argue that providing an abstraction for these array layouts at the compiler level is the right approach.

The key aspects towards providing an abstraction are discussed in the following sub-sections.

7.3.1 Associativity Conflicts in Morton layout

It is our hypothesis that associativity of various levels of the memory hierarchy influences the overall performance of Morton layout.

In Figure 7.2, we illustrate the row-to-row associativity conflicts in Morton layout, for a direct-mapped cache. The diagram shows how systematic associativity conflicts occur with both row- and column-major traversals when problem size exceeds the square root of the cache-size (for a direct-mapped cache). In general, for a w-way associative cache with capacity C, addresses aligned at $(\frac{C}{w})$ bytes are mapped to the same set. If each word is l bytes, each way holds $\frac{C}{w.l}$ words. This means any two arbitrary addresses s and t should collide when $|s-t|\%\frac{C}{w.l} = 0$. With row-major layout, this happens with `A[i,s]` and `A[i,t]`, or for elements separated by a multiple of $|s-t|$. With Z-Morton layout this happens with `A[i,u]` and `A[i,v]` where $|u-v| = \sqrt{\frac{C}{w.l}}$. Thus, with row-major traversal, Z-Morton layout is expected to suffer from associativity conflicts for smaller problem size than row-major. We call a region within which canonical traversal of a layout is free from conflicts as a conflict-free region. The size of the conflict-free region for row-major traversal of row-major array is $\frac{C}{w.l}$. The size for the row-major traversal of a Morton array is $\sqrt{\frac{C}{w.l}}$.

For Pentium 4 and Athlon systems with the cache parameters given in Table 4.1 (P4: 512KB/8-way/128B, Athlon: 512KB/16-way/64B), for a benchmark with single Morton array, it can be expected that the associativity conflicts to have greater impact on performance when the array size exceeds

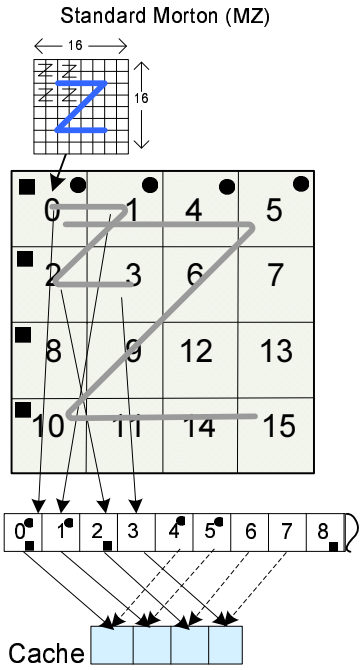


Figure 7.2: **Conflict Misses in Morton Layout.** The figure illustrates the nature of associativity conflicts in Morton layout. We show the mapping of page-sized Morton blocks into linear memory and into a direct-mapped 4-page cache. When accessing the Morton array in row-major order, pages 0 and 1 conflict with pages 4 and 5. Similarly, when traversed in column-major order, pages 0 and 2 conflict with pages 8 and 10. This generates 2 misses per row or column for each traversal order. The analysis can easily be extended for caches with higher levels of associativity .

1024 words (P4: 128×7, Athlon: 64×16). From our experimental results reported in last three chapters, it can be noticed that Morton performance drops near the problem sizes stated above. These observations broadly confirm our hypothesis. In our technical report [84], we perform an in-depth analysis of associativity conflicts in Morton layout and an alternative layout is proposed to minimize the associativity conflicts.

7.3.2 Implementation of High Level of Abstraction for Hierarchical Storage Layouts

The first step towards providing a compiler support for hierarchical storage layouts is to develop efficient abstract representations of a wide range of hierarchical storage layouts. Abstractions should include meta-data so that an underlying compiler infrastructure could use that information to selectively apply appropriate optimisations. Padua *et al.* [3] provide such an abstraction for hierarchical storage layouts but without any domain specific compiler support.

7.3.3 Combining Iteration and Data-Space Transformations

As discussed in Chapter 2, Kandemir *et al.* [41], Cierniak and Li [21], O'Boyle *et al.* [62] have demonstrated the profitability of combined transformations. Support for combined transformations is very crucial with hierarchical storage layouts and this involves automatic tiling and recursion of control flow structures.

7.3.4 Data Structure Specific Performance Metrics

Existing performance measurement techniques do not provide data structure specific performance results. Such data structure specific performance metrics are crucial, for example, in justifying a choice of a layout for an array. Gerndt *et al.* [47] discuss a memory access monitoring environment for Fortran programs. Combining such methods with performance measurement techniques which use hardware performance counters [13] is useful for developing data structure specific performance tools.

7.3.5 Low Level Support for Hierarchical Storage Layouts

When compiling the code generated from the domain-specific abstractions, the instruction set and hardware play an important role. Investigation of providing low level support for hierarchical storage layouts is important. For example:

- *Instruction set support for address computation*: As illustrated in Chapter 3, the address computation for hierarchical layouts is fairly expensive process. While traditional layouts access neighbouring elements just by incrementing one of the array indices, hierarchical layouts often involve operations on dilated indices, which are expensive to implement using existing instruction sets. Instruction set support for address computation is particularly important for random accesses, for example, with many levels of indirection, where other optimisation techniques, such as unrolling, do not help. The effect of hardware support for dilated operations could be evaluated, initially in a simulated environment like SimpleScalar [77].

- *Caching/Prefetching mechanisms*: Modern architectures, for example Pentium IV, detect any constant stride access for arrays in a loop nest and successfully prefetch elements to hide access latency. This technique has been proven to be an effective optimisation [36, 38] on many applications. However, with hierarchically stored arrays, the strides are non-constant but follow a predictable pattern. The issue of prefetching for hierarchical storage layouts is an interesting issue to be addressed.

- *Interaction with SIMD Instructions*: Modern processors support SIMD instructions — the VIS instructions in UltraSparc processors [79], SSE/SSE2/SSE3 in Pentium IV [37] and AltiVec in Motorola processors [25, 27]. However, these instructions inherently assume that data elements are arranged in one of the lexicographic orders when performing most of the cacheability and data management instructions. Direct SIMD operations on Morton array elements is not possible without involving additional data movements. If the interaction between Morton, or hierarchical layouts in general, and SIMD instruction set is improved, many classes of applications could benefit from such layouts.

7.3.6 Hierarchical Storage Layouts for Sparse Matrix Computations

There are numerous storage formats for sparse matrices. However, these storage formats lack the concept of compromise layout. Though Morton layout offers some compromise, it is not readily applicable for storing sparse matrices. Initially, we would like to investigate the performance impact of using Morton layout in sparse matrix computations and in libraries like Sparsity [35]. Following this, we would like to find a compromise storage layout for sparse matrices, based on recursive blocking.

7.3.7 Address Computation

In this thesis, we looked at two different methods of address computation for Morton layouts, which is generally applicable for any hierarchical storage layout. Raman and Wise discuss alternative ways of converting to and from dilated integers in one of their recent papers [70]. This leads us to consider applying different methods for address calculations based on the underlying layout. More investigation is needed in evaluating the most beneficial method of computation for a given layout.

7.3.8 Optimal Layouts

In this thesis, we have taken the approach of assigning all arrays with the same array layout. However, as discussed in Chapter 2, different layouts could be chosen to improve locality of individual arrays. We believe that cross interference effects between arrays could be minimised by mixing layouts.

In addition to the techniques suggested by Kandemir *et al.* [41], the optimal layouts for arrays can be determined at compile time by using the profiled or trace data, as discussed by Rubin *et al.* [73].

Beckmann *et al.* [9] use a technique whereby they delay the execution of a program to determine optimal data placement techniques for parallel programs. It is possible to adopt this technique to perform runtime layout transformation. If we assume that the layouts are dynamically changed, we will be delaying the execution of a program, as long as possible so that optimal layouts could be chosen minimising the layout changes.

7.3.9 Multi-dimensional Arrays

This thesis is entirely focused on applications/kernels that use two-dimensional arrays. Though the layouts and associated techniques are equally applicable for higher-dimensional arrays, the potential performance benefit should be worse than for two-dimensional arrays. For example, for three-dimensional arrays, the basic block which to be contained in memory levels is now a three-dimensional cube. If we assume that Morton layout is used, the spatial locality along one of the dimensions is cube-root of the cubic block size, much worse than the square-root value of two-dimensional case. However, we are interested in particular application classes, like ray casting [55], that might benefit from such layouts. One of the ways to improve the spatial locality is to select a layout with the highest number of orientations, such as the Hilbert curve which may minimise the self-interference effects.

7.4 Summary

This chapter has pointed out the contributions made by this thesis and discussed the design considerations for compiler support for hierarchical layouts. The chapter has also outlined a number of directions for further research. The main challenge in using these layouts in scientific programs is to provide compiler support for which this thesis stands as a starting point.

Bibliography

[1] Nawaaz Ahmed and Keshav Pingali. Automatic generation of block-recursive codes. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par '00: Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, volume 1900 of *LNCS*, pages 368–378. Springer-Verlag, October 2000.
Cited on page 25

[2] Lelac Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools*, pages 231–239. ACM Press, 2004.
Cited on page 24

[3] George Almasi, Luiz De Rose, Jose Moreira, and David Padua. Programming for locality and parallelism with hierarchically tiled arrays. In Lawrence Rauchwerger, editor, *Proceedings of Languages and Compilers for Parallel Computing*, volume 2958 of *LNCS*, pages 162–176. Springer-Verlag, October 2003.
Cited on page 102

[4] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, 1994.
Cited on page 33, 41, 68

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
Cited on page 16, 24

[6] Utpal Banerjee. *Loop transformations for restructuring compilers: The foundations*. Kluwer Academic Publishers, 1993.
Cited on page 20

[7] Ioana Banicescu and Susan Flynn Hummel. Balancing processor loads and exploiting data locality in N-body simulations. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*, page 43. ACM/IEEE, December 1995.
Cited on page 34

[8] Cedric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In Lawrence Rauchwerger, editor, *Proceedings of Languages and Compilers for Parallel Computing*, volume 2958 of *LNCS*, pages 209–225. Springer-

Verlag, October 2003.
Cited on page 23

[9] Olav Beckmann. *Interprocedural Optimisation of Regular Parallel Computations at Runtime*. PhD thesis, Department of Computing, Imperial College, London, England, January 2001.
Cited on page 29, 104

[10] Olav Beckmann, Alastair Houghton, Paul H J Kelly, and Michael Mellor. Run-time code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, number 3016 in LNCS, March 2004.
Cited on page 84

[11] Theodore Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, IT-15(6):658–664, November 1992.
Cited on page 34, 38, 39

[12] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
Cited on page 20

[13] Shirley Browne, Jack J. Dongarra, Nathan Garner, Kevin London, and Philip J. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, page 42. IEEE Computer Society, 2000.
Cited on page 103

[14] Steve Carr and Rich B. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Transactions on Mathematical Software*, 23(3):336–361, September 1997.
Cited on page 24

[15] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *ASPLOS '94: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 28, pages 252–262, October 1994.
Cited on page 20

[16] Larry Carter, Jeanne Ferrante, Susan Flynn Hummel, Bowen Alpern, and Kang Su Gatlin. Hierarchical tiling: A methodology for high performance. Technical Report CS96-508, Department of Computer Science, University of California, San Diego, USA, November 1996.
Cited on page 24

[17] Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, pages 492–499. ACM Press, 1999.
Cited on page 24

[18] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–28. ACM Press, 1993.
Cited on page 29

[19] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, pages 444–453. ACM Press, 1999.
Cited on page 32, 33

[20] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *SPAA '99: Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231. ACM Press, 1999.
Cited on page 38, 39, 68

[21] Michał Cierniak and Wei Li. Unifying data and control transformations for distributed shared-memory machines. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 205–217. ACM Press, 1995.
Cited on page 16, 20, 28, 30, 102

[22] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming language design and implementation*, pages 279–290. ACM Press, 1995.
Cited on page 24

[23] Peter Drakenberg, Fredrik Lundevall, and Björn Lisper. An efficient semi-hierarchical array layout. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*. Kluwer, January 2001. Available via `www.mrtc.mdh.se`.
Cited on page 35

[24] Edison Design Group, `http://www.edg.com/`.
Cited on page 99

[25] Freescale Semiconductor. Inc. *AltiVec Technology: Programming Environments Manual*, February 2002. `www.freescale.com/files/32bit/doc/fact_sheet/ALTIVECEM.pdf`.
Cited on page 103

[26] Matteo Frigo. A fast Fourier transform compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 169–180. ACM Press, 1999.
Cited on page 25

[27] Sam Fuller. Motorola's AltiVec technology. 2004. `http://www.freescale.com/files/-32bit/doc/fact_sheet/ALTIVECWP.pdf`.
Cited on page 103

[28] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, November 1999.
Cited on page 35

[29] Fred G. Gustavson. New generalized data structures for matrices lead to a variety of high performance algorithms. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Waniewski, editors, *PPAM 2001: Proceedings of the 4th International Conference on Parallel Processing and Applied Mathematics*, volume 2328 of *LNCS*, pages 418–436. Springer-Verlag, September 2002.
Cited on page 33

[30] Fred G. Gustavson. High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM Journal of Research and Development*, 47(1), January 2003.
Cited on page 33

[31] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers, Third edition, 2003.
Cited on page 17, 18, 19

[32] Sambuddhi Hettiaratchi, Peter Y. K. Cheung, and Thomas J. W. Clarke. Energy efficient address assignment through minimized memory row switching. In *ICCAD 2002: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 577–581. ACM Press, 2002.
Cited on page 31

[33] David Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, March 1891. In German.
Cited on page 13, 34, 35, 68

[34] Yu Charlie Hu, S. Lennart Johnsson, and Shang-Hua Teng. High performance Fortran for highly irregular problems. In *PPOPP '97: Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24. ACM Press, 1997.
Cited on page 34

[35] Eun-Jin Im, Katherine A. Yelick, and Richard Vuduc. SPARSITY: An optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.
Cited on page 103

[36] Intel Corporation. *IA-32 Intel Architecture Optimization: Reference Manual*, 2004.
Cited on page 103

[37] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual: Basic Architecture*, 2004.
Cited on page 103

[38] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual: System Programming Guide*, 2004.
Cited on page 103

[39] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 332–342. ACM Press, May 1990.
Cited on page 34

[40] Y.-J. Ju and Henry G. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of Languages and Compilers for Parallel Computing*, volume 589 of *LNCS*, pages 344–358. Springer-Verlag, August 1992.
Cited on page 30

[41] Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, N. Shenoy, and Prithviraj Banerjee. Enhancing spatial locality via data layout optimizations. In D. Pritchard and J. Reeve, editors, *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, volume 1470 of *LNCS*, pages 422–434. Springer-Verlag, September 1998.
Cited on page 16, 31, 102, 104

[42] Mahmut T. Kandemir, Alok N. Choudhary, N. Shenoy, Prithviraj Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, February 1999.
Cited on page 29

[43] Mahmut T. Kandemir, J. Ramanujam, Alok N. Choudhary, and Prithviraj Banerjee. A layout-conscious iteration space transformation technique. *IEEE Transactions on Computers.*, 50(12):1321–1336, 2001.
Cited on page 28

[44] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Department of Computer Science, University of Maryland, April 93.
Cited on page 22, 23

[45] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*.
Cited on page 20

[46] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *ICS '92: Proceedings of the 6th International Conference on Supercomputing*, pages 323–334. ACM Press, 1992.
Cited on page 20

[47] Edmond Kereku, Tianchao Li, Michael Gerndt, and Josef Weidendorfer. A data structure oriented monitoring environment for Fortran OpenMP programs. In Macro Danelutto, Marco Vanneschi, and Dominico Laforenza, editors, *Euro-Par 2004 Parallel Processing:Proceedings of the 10th International Euro-Par Conference*, volume 3149 of *LNCS*, pages 133–140. Springer-Verlag, September 2004.
Cited on page 103

[48] Peter M. W. Knijnenburg, Eduard Ayguagé, and Jordi Torres. Multi-transformations of nested loops for parallelizing compilers. Technical Report TR-96-14, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Leiden, The Netherlands, 1996.
Cited on page 23

[49] Peter M. W. Knijnenburg, Toru Kisuki, and Michael F. P. O'Boyle. Iterative compilation. In F. Deprettere, J. Teich, and S. Vassiliadis, editors, *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, volume 2268 of *LNCS*, pages 171–187. Springer-Verlag, July 2003.
Cited on page 24, 25

[50] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. *ACM SIGPLAN Notices*, 32(5):346–357, May 1997.
Cited on page 24

[51] 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.
Cited on page 24, 25

[52] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974.
Cited on page 21

[53] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.
Cited on page 29

[54] Shun-Tak Leung and John Zahorjan. Optimizing data locality by array restructuring. Technical Report TR-95-09-01, Department of Computer Science, Washington State University, Washington, USA, September 1995.
Cited on page 31

[55] Marc Levoy. A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics and Applications*, 10(2):33–40, March 1990.
Cited on page 104

[56] Wei Li. *Compiling for NUMA Parallel Machines*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, USA, 1994.
Cited on page 20, 31

[57] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.
Cited on page 23

[58] Morris Mano. *Digital Design*. Prentice Hall, Third edition, 2002.
Cited on page 35

[59] 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.
Cited on page 20

[60] Nicholas Mitchell, Karin Högstedt, Larry Carter, and Jeanne Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.
Cited on page 24, 42

[61] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, volume 36, pages 89–104. ACM Press, 2002.
Cited on page 41

[62] Michael F. P. O'Boyle and Peter M. W. Knijnenburg. Integrating loop and data transformations for global optimisation. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 12–19. IEEE Computer Society Press, October 1998.
Cited on page 16, 102

[63] Avigail Orni. Measuring the locality of space-filling curves. Master's thesis, Department of Computer Science, Technion, Haifa, Israel, 1999.
Cited on page 40

[64] Preeti Ranjan Panda and Nikil D. Dutt. Low-power memory mapping through reducing address bus activity. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(3):309–320, 1999.
Cited on page 31

[65] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt, and Alexandru Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2):142–149, February 1999.
Cited on page 25

[66] Giuseppe Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36:157–160, January 1890. In French.
Cited on page 13, 34, 35

[67] John R. Pilkington and Scott B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves:. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–300, March 1996.
Cited on page 34

[68] William Pugh and Evan Rosser. Iteration space slicing and its application to communication optimization. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 221–228. ACM Press, July 1997.
Cited on page 25

[69] Daniel Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, September 2000.
Cited on page 98, 99

[70] Rajeev Raman and David S. Wise. Converting to and from dilated integers, 2004. Available via `http://www.cs.indiana.edu/~dswise/Arcee/`.
Cited on page 104

[71] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 38–49. ACM Press, 1998.
Cited on page 25, 64

[72] Gabriel Rivera and Chau-Wen Tseng. Eliminating conflict misses for high performance architectures. In *ICS '98: Proceedings of the 12th International Conference on Supercomputing*, pages 353–360. ACM Press, July 1998.
Cited on page 25

[73] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. *ACM SIGPLAN Notices*, 37(1):140–153, January 2002.
Cited on page 104

[74] Lothar Sachs. *Statistische Methoden*. Springer Verlag, 5th edition, 1982.
Cited on page 51

[75] Hans Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.
Cited on page 13, 35

[76] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 117–127. ACM Press, 2000.
Cited on page 24, 42

[77] SimpleScalar, `http://www.simplescalar.com/`.
Cited on page 103

[78] SPEC2000, `http://www.specbench.org/`.
Cited on page 48

[79] Sun Microsystems. *UltraSPARC-III*, 2004. User Manual.
Cited on page 103

[80] Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 410–419. ACM Press, 1993.
Cited on page 25

[81] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul H. J. Kelly. An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays. In Stephen A. Jarvis, editor, *Performance Engineering:* 19^{th} *Annual UK Performance Engineering Workshop*, pages 340–351. University of Warwick, UK, July 2003.
Cited on page 14, 44

[82] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul H. J. Kelly. Improving the performance of morton layout by array alignment and loop unrolling reducing the price of naivety. In Lawrence Rauchwerger, editor, *Proceedings of 16th International Workshop on Languages and Compilers for Parallel Computing*, volume 2958 of *LNCS*, pages 241–257. Springer-Verlag, October 2003.
Cited on page 14, 64, 81

[83] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul H. J. Kelly. Is morton layout competitive for large two dimensional arrays, yet? In *Concurrency And Computation: Practice and Experience*, 2005. To appear.
Cited on page 14, 44, 64

[84] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul H. J. Kelly. Minimizing Associativity Conflicts in Morton Layout. Technical report, Department of Computing, Imperial College London, U.K, May 2005.
Cited on page 102

[85] Jeyarajan Thiyagalingam and Paul H. J. Kelly. Is Morton layout competitive for large two-dimensional arrays? In B. Monien and R. Feldman, editors, *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, volume 2400 of *LNCS*, pages 280–288. Springer-Verlag, October 2002.
Cited on page 14, 44

[86] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–839, August 2002.
Cited on page 35

[87] Richard Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–25, 2001.
Cited on page 25

[88] Robert P. Wilson, Monica S. Lam, John L. Hennessy, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, and Mary W. Hall. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.
Cited on page 20, 98

[89] David S. Wise. Ahnentafel indexing into morton-ordered arrays, or matrix locality for free. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par '00: Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, volume 1900 of *LNCS*, pages 774–783. Springer-Verlag, October 2000.
Cited on page 37, 45

[90] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for morton-order matrices. In *PPoPP '01: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 24–33. ACM Press, 2001.
Cited on page 34, 45

[91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.
Cited on page 20, 31

[92] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 274–286, Paris, December 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.
Cited on page 24

[93] Michael E. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.
Cited on page 21

[94] Michael E. Wolfe and Mitsuru Ikei. Automatic array alignment for distributed memory multicomputers. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the 27th Annual Hawaii International Conference on System Sciences : Software Technology*, volume 2, pages 23–32. IEEE Computer Society Press, January 1994.
Cited on page 29

[95] Qing Yi, Vikram Adve, and Ken Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 169–181, 2000.
Cited on page 25

[96] Kamen Yotov, Peng Wu, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. A comparison of empirical and model-driven optimization. In Jr. James B. Fenwick and Cindy Norris, editors, *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 63–76. ACM Press, 2003.
Cited on page 25